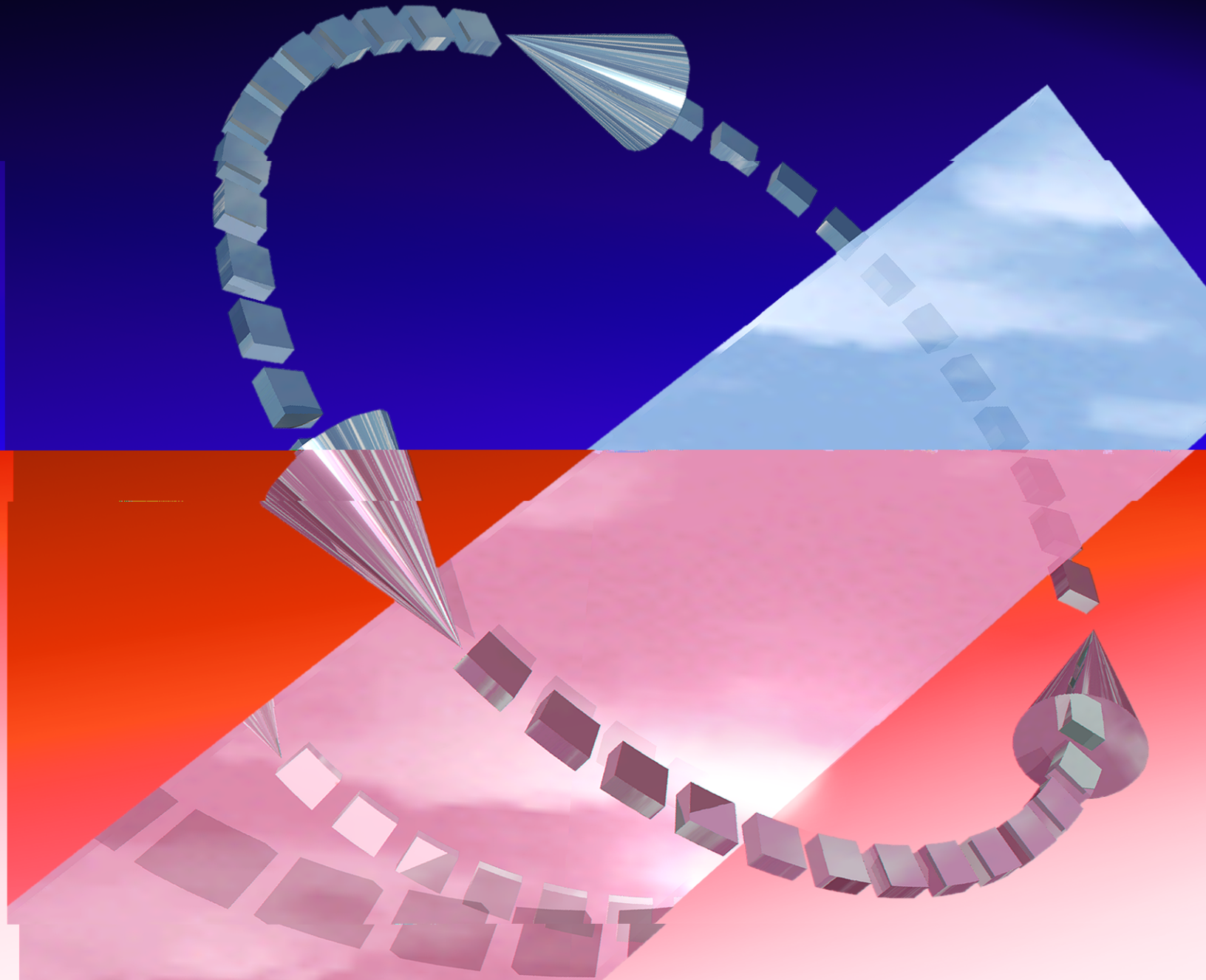
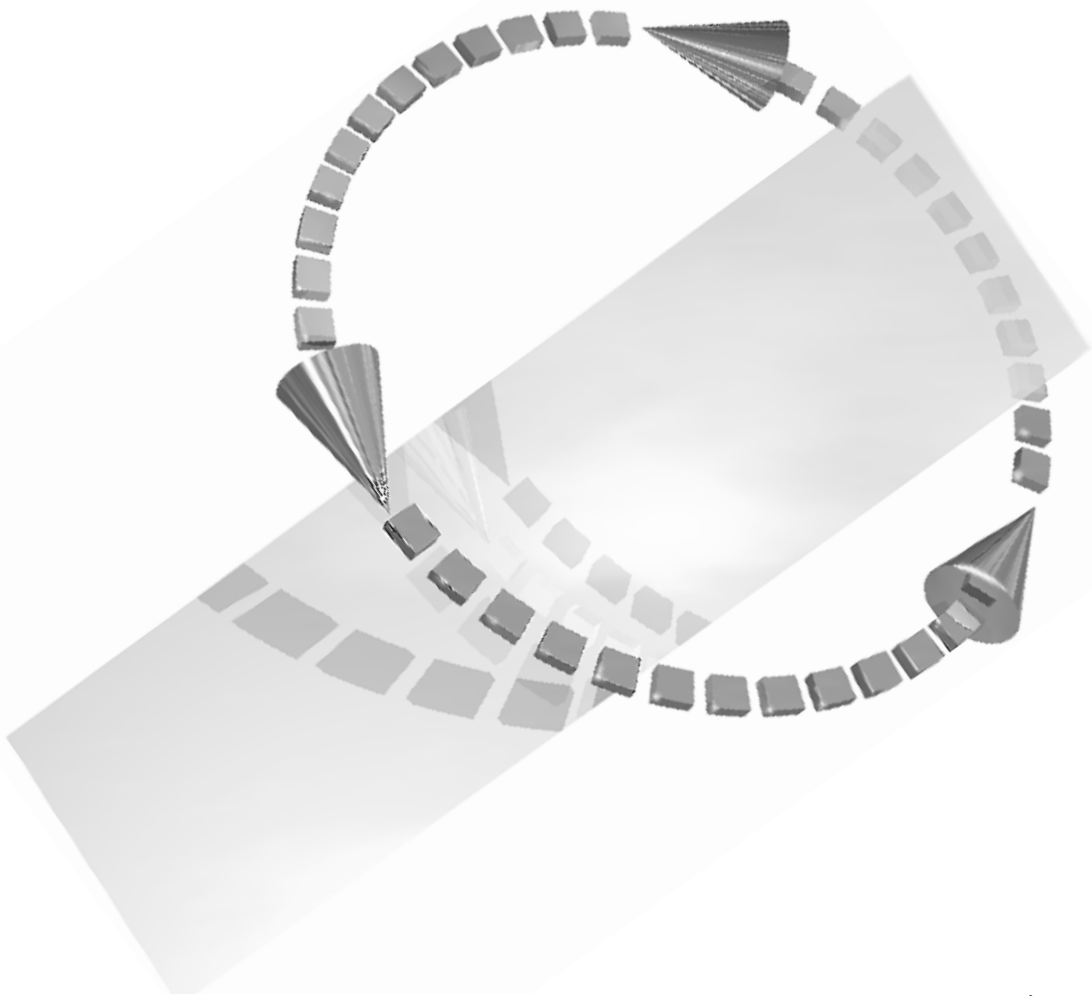




VR-Link

*The Networking Toolkit
for the Virtual World*



VR-Link

*The Networking Toolkit
for the Virtual World*



MÄK Technologies
185 Alewife Brook Parkway
Cambridge, MA 02138 USA

Copyright 1998, MÄK Technologies

All rights Reserved. Printed in the United States.

First Printing: February, 1998

Under copyright laws, no part of this document may be copied or reproduced in any form without prior written consent of MÄK Technologies, Inc.

VR-Link, MÄK Logger, and MÄK Stealth are trademarks of MÄK Technologies, Inc.

All other trademarks are owned by their respective companies.

MÄK Technologies

185 Alewife Brook Parkway

Cambridge, MA 02138 USA

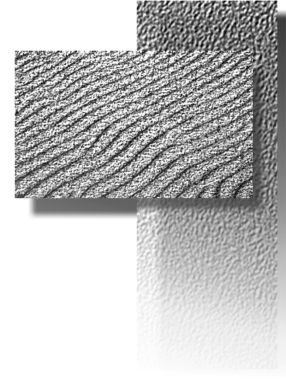
Voice: 617-876-8085

Fax: 617-876-9208

info@mak.com

www.mak.com

Revision VRL-3.3-1-981012



Contents

Preface

How this Guide is Organized	xvii
About the Hypertext Version	xviii
MÄK Products for the Virtual World	xix
Getting Technical Support	xx
Document Conventions	xxi

Chapter 1 Introduction

1.1. Main Features	1-2
1.1.1. Files and Executables in VR-Link	1-2
1.2. Protocol Versions Supported	1-3
1.2.1. Obtaining Further Information about HLA and DIS	1-3

Chapter 2 Installing and Configuring VR-Link

2.1. System Requirements	2-2
2.2. Installing VR-Link	2-4
2.2.1. Installing VR-Link on UNIX Systems	2-4
2.2.2. Installing VR-Link on PCs	2-4
2.2.3. VR-Link Files Installed	2-4
2.3. Setting Up and Using the FLEXlm License Manager	2-6
2.3.1. Setting up the License Server	2-6
2.3.2. Running the License Server	2-8
2.3.3. Troubleshooting the FLEXlm License Manager	2-9
2.4. Obtaining and Installing a Runtime Infrastructure (RTI)	2-10
2.4.1. Installing the MÄK RTI	2-10
2.4.2. Installing the DMSO RTI	2-11

2.5. Configuring the Network for DIS Applications	2-13
2.5.1. Broadcast Address and Netmask (UNIX)	2-13
2.5.2. Network Mask (Windows)	2-14
2.5.3. UDP Port	2-14

Chapter 3 VR-Link Concepts

3.1. A Conceptual Overview of VR-Link	3-2
3.1.1. VR-Link Classes in Brief	3-2
3.2. Basic VR-Link Examples	3-4
3.2.1. A Listen-Only Example	3-4
3.2.2. A Send-Only Example	3-7
3.2.3. Notes about the Examples	3-10
3.3. Compiling and Linking a VR-Link Application	3-13
3.3.1. Libraries and Header Files	3-13
3.3.2. Compiling for a Particular Protocol Version under UNIX	3-13
3.3.3. Selecting an Example Application's Protocol under UNIX	3-14
3.3.4. Compiling for a Particular Protocol Version under Windows	3-15
3.3.5. Selecting an Example Application's Protocol under Windows	3-17
3.4. Using VR-Link from C	3-18

Chapter 4 The Protocol-Independent Interface

4.1. Connecting to Exercises	4-2
4.1.1. Creating an Exercise Connection for HLA	4-2
4.1.2. Creating an Exercise Connection for DIS	4-3
4.1.3. Compiling Applications for Protocol-Independence	4-3
4.1.4. DtExerciseConn Member Functions	4-4
4.2. Managing Interactions	4-7
4.2.1. Sending Interactions	4-7
4.2.2. Receiving Interactions	4-8
4.3. Managing Entities	4-11
4.3.1. Managing Locally Simulated Entities	4-11
4.3.2. Managing Remote Entities	4-15
4.4. Coordinate Views	4-25
4.4.1. Topographic Coordinate Views	4-26
4.4.2. UTM Coordinate Views	4-27
4.4.3. Cartesian Coordinate Views	4-27
4.5. Articulated Parts	4-28
4.5.1. Inspecting Articulated Parts Data	4-28
4.5.2. Setting Articulated Parts Data	4-30

Chapter 5 DIS-Specific Interface

5.1. Working with PDUs	5-2
5.1.1. Sending PDUs	5-2
5.1.2. Receiving PDUs	5-3

5.2. Inspecting and Setting Data in A PDU Header	5-7
5.2.1. Getting a PDU's Network Representation	5-8
5.2.2. DtPdu Constructors	5-8
5.2.3. DtPduFactory	5-10
5.2.4. Using External Buffers in a PDU Class	5-11
5.2.5. Copying PDUs	5-12
5.3. Creating Your Own PDUs	5-13
5.3.1. Using DtUnknownPdu	5-13
5.3.2. Deriving Classes from DtPdu	5-14
5.4. Configuring Your Connection to the DIS Network	5-21
5.4.1. Configuring a DtNetSocket	5-22
5.4.2. Configuring a DtClientSocket	5-23
5.4.3. Multicast	5-24
5.4.4. Filtering	5-24
5.4.5. Bundling and Unbundling PDU Packets	5-25
5.4.6. DtSocket Specifics	5-26
5.5. Intercepting Incoming Entity State PDUs	5-29

Chapter 6 HLA-Specific Interface

6.1. The RTI	6-2
6.1.1. Federate-Initiated Services	6-2
6.1.2. RTI-Initiated Services	6-2
6.2. Managing HLA Objects	6-4
6.2.1. Managing Locally Simulated Objects	6-4
6.2.2. Subscribing to Object Classes	6-7
6.2.3. Discovering and Removing Objects	6-8
6.3. Intercepting Reflected Attribute Updates	6-10
6.4. Working With Interactions	6-11
6.4.1. Publishing Interactions	6-11
6.4.2. Subscribing to Interactions	6-11
6.4.3. Using the DtInteraction Class	6-12
6.4.4. Using DtInteractionFactory	6-13
6.5. Creating Interaction Classes	6-14
6.5.1. Using DtUnknownInteraction	6-14
6.5.2. Deriving Classes From DtInteraction	6-17

Chapter 7 Utilities

7.1. Vector and Matrix Classes	7-2
7.1.1. DtVector and DtDcm Semantics	7-3
7.1.2. Vector and Matrix Manipulation Functions	7-4
7.2. Orientation, Euler Angles, and DtTaitBryan	7-5
7.2.1. Converting Between Euler Angles and Matrix Representation	7-6

7.3. Coordinate Conversions	7-7
7.3.1. Geocentric Coordinates	7-7
7.3.2. Geodetic Coordinates	7-8
7.3.3. Topographic Coordinates	7-10
7.3.4. UTM Coordinates	7-12
7.3.5. Considerations for Using UTM Coordinates	7-13
7.3.6. Lower Level Coordinate Conversion Functions	7-15
7.4. Managing Time	7-17
7.5. Creating Linked Lists with DtList	7-19
7.6. Diagnostic Utilities	7-21
7.7. IP Address Manipulation Functions	7-22
7.8. Miscellaneous Functions	7-23

Chapter 8 Example and Utility Applications

8.1. About the Applications	8-2
8.1.1. License Enforcement for the Example Applications	8-2
8.2. The f18 Program	8-3
8.2.1. The f18 Configuration File	8-4
8.2.2. Firing Munitions	8-5
8.2.3. Reactions to Detonate Messages	8-5
8.2.4. Absolute timestamping (DIS only)	8-5
8.3. netdump (DIS Only)	8-6
8.4. hlaNetdump (HLA only)	8-8
8.5. nethex (DIS only)	8-9
8.6. talk and listen	8-9
8.7. vdc – the VR-Link Daemon Controller	8-10
8.7.1. vdc Examples	8-11
8.7.2. vdc Diagnostics	8-11
8.8. The VR-Link Packet Server (DIS and UNIX Only)	8-12
8.8.1. Packet Server Modes	8-12
8.8.2. Packet Server Configurations	8-12
8.8.3. How Applications Connect to the Packet Server	8-12
8.8.4. Using the Packet Server	8-13
8.8.5. Starting the Packet Server	8-15
8.8.6. Stopping the Packet Server	8-17
8.8.7. Packet Server Controls	8-17
8.8.8. Pinging Clients	8-18
8.8.9. Unbundling Packets	8-18
8.8.10. Status Commands	8-18
8.8.11. Queue Sizes	8-19
8.8.12. Technical Notes	8-19
8.9. buffgrep	8-21
8.9.1. buffgrep Examples	8-21

Chapter 9 Guide to Classes and Header Files

9.1. Convenience Header Files	9-2
9.1.1. Header Files for Protocol Independence	9-2
9.2. Class-to-Header Cross Reference	9-4
9.3. Header-to-Class Cross Reference	9-16
9.4. Other Header Files	9-28
9.5. Class Hierarchy Diagrams and Tables	9-29

Appendix A PDU Class Reference

DtAcknowledgePdu	A-2
DtActionRequestPdu	A-4
DtActionResponsePdu	A-6
DtCollisionPdu	A-8
DtCommentPdu	A-10
DtCreateEntityPdu	A-12
DtDataPdu	A-13
DtDataQueryPdu	A-15
DtDesignatorPdu	A-17
DtDetonationPdu	A-19
DtEmissionPdu	A-22
DtEntityStatePdu	A-27
DtEventReportPdu	A-32
DtFirePdu	A-34
DtLaserPdu	A-36
DtMessagePdu	A-38
DtPduWithData	A-44
DtReceiverPdu	A-51
DtRemoveEntityPdu	A-52
DtRepairCompletePdu	A-53
DtRepairResponsePdu	A-54
DtResupplyCancelPdu	A-55
DtResupplyOfferPdu	A-56
DtResupplyReceivedPdu	A-58
DtSetDataPdu	A-60
DtSignalPdu	A-62
DtSimanPdu	A-64
DtStartResumePdu	A-66
DtStopFreezePdu	A-68
DtTransmitterPdu	A-70
DtViewControlPdu	A-75

Appendix B Header File Reference

absNotifyObs.h	B-2
ackPdu.h	B-3
ackRPdu.h	B-5
actionPdu.h	B-7
actReqRPdu.h	B-9
actRespRPdu.h	B-11
aggAggList.h	B-13
aggCbInfo.h	B-15
aggEntDec.h	B-17
aggEntEnc.h	B-19
aggEntList.h	B-21
aggPub.h	B-23
aggStatePdu.h	B-24
aggregateSR.h	B-30
antPatRec.h	B-34
apSpec.h	B-37
arealObjStPdu.h	B-38
artPart.h	B-41
artPartList.h	B-44
artPartThresh.h	B-48
attPartList.h	B-51
baseEmbedDec.h	B-54
baseEmbedEnc.h	B-56
baseEmittrDec.h	B-58
baseEmittrEnc.h	B-60
baseEntDec.hr	B-62
baseEntEnc.h	B-64
baseEntitySR.h	B-67
baseExConn.h	B-71
baseHashKey.h	B-73
baseMsg.h	B-74
baseReflAgg.h	B-76
baseReflEnt.h	B-77
beamAntRec.h	B-78
beamCbInfo.h	B-80
cartView.h	B-82
ClientSocket.h	B-84
collidePdu.h	B-87
collisElasPdu.h	B-89
commentPdu.h	B-92
commentRPdu.h	B-93
createEntRPdu.h	B-95
createPdu.h	B-97
dAggDec.h	B-98

dAggEnc.h	B-99
dAggPub.h	B-101
dataPdu.h	B-102
dataQueryRPdu.h	B-103
dataRPdu.h	B-105
datumIdType.h	B-107
datumMap.h	B-108
datumRecSet.h	B-110
dcm.h	B-116
deadReckoner.h	B-117
decFactory.h	B-120
dEntIdHashLst.h	B-122
dEntityDec.h	B-123
dEntityEnc.h	B-124
dEntityPub.h	B-126
designatePdu.h	B-127
designatorDec.h	B-129
designatorEnc.h	B-131
designatorSR.h	B-133
designCbInfo.h	B-135
detonatePdu.h	B-137
dExerciseConn.h	B-140
dInteraction.h	B-143
directRotRate.h	B-144
dirRtiAmb.h	B-145
disEnums.h	B-155
disPduFactory.h	B-194
dRefAggList.h	B-195
dReflectedAgg.h	B-198
dReflectedEnt.h	B-200
dRefEntList.h	B-202
dscvObjCbInfo.h	B-205
dStateMsg.h	B-207
elecEmitPdu.h	B-208
embedSystemSR.h	B-212
emissionPdu.h	B-213
emitrSysCbInf.h	B-216
emitterSysDec.h	B-218
emitterSysEnc.h	B-220
emitterSysSR.h	B-222
emitrBeamSR.h	B-224
encFactory.h	B-227
entCbInfo.h	B-230
entitySR.h	B-232
EntityTypes.h	B-237
enum2string.h	B-253

envDec.h	B-254
envEnc.h	B-255
enviroTypeRec.h	B-257
envProcPdu.h	B-258
esPdu.h	B-260
esUpdatePdu.h	B-263
Euler.h	B-266
eventId.h	B-267
eventReptPdu.h	B-269
eventReptRPdu.h	B-270
filename.h	B-272
firePdu.h	B-273
fom.h	B-275
fomMapper.h	B-277
gedBasFixWing.h	B-280
gedBasRotWing.h	B-282
gedBasSoldier.h	B-285
gedBasVehicle.h	B-287
gedEnhFixWing.h	B-289
gedEnhRotWing.h	B-291
gedEnhSoldier.h	B-293
gedEnhVehicle.h	B-295
gedLogVehicle.h	B-297
geodCoord.h	B-299
globalObjDes.h	B-301
globIdHLst.h	B-302
gridAxiRecFix.h	B-303
gridAxiRecVar.h	B-305
gridDataPdu.h	B-307
gridDataRec.h	B-311
gridDataRecT0.h	B-312
gridDataRecT1.h	B-314
gridDataRecT2.h	B-316
hAckDec.h	B-318
hAckEnc.h	B-319
hAckInter.h	B-320
hActReqDec.h	B-322
hActReqEnc.h	B-323
hActReqInter.h	B-324
hAggPub.h	B-326
hAppRadSigDec.h	B-328
hAppRadSigEnc.h	B-329
hashlist.h	B-330
hCollideDec.h	B-332
hCollideEnc.h	B-334
hCollideInter.h	B-335

hDbiRadSigDec.h	B-337
hDbiRadSigEnc.h	B-338
hDesignatrPub.h	B-339
hDetDec.h	B-340
hDetEnc.h	B-342
hDetInter.h	B-343
hEmitrBeamPub.h	B-346
hEmittrSysPub.h	B-348
hEncARSigDec.h	B-350
hEncARSigEnc.h	B-351
hEntityPub.h	B-352
hExerciseConn.h	B-353
hFireDec.h	B-357
hFireEnc.h	B-358
hFireInter.h	B-359
hInteraction.h	B-361
hInterFactory.h	B-364
hInterWData.h	B-365
hIntWEncDec.h	B-369
hlaObj.h	B-371
hlaObjFact.h	B-375
hlaObjWithSR.h	B-377
hLcDec.h	B-379
hLcEnc.h	B-380
hLcInter.h	B-381
hObjPub.h	B-382
hostStructs.h	B-384
hRadioRcvrPub.h	B-389
hRadioXmitPub.h	B-390
hRawBinRSEnc.h	B-392
hRefBeamList.h	B-393
hRefDesigList.h	B-395
hRefEmSysList.h	B-397
hRefEntHList.h	B-400
hReflAggList.h	B-401
hReflBeam.h	B-403
hReflDesigntr.h	B-405
hReflectedAgg.h	B-407
hReflectedEnt.h	B-409
hReflectedObj.h	B-411
hRefObjList.h	B-414
hRefEmitrSys.h	B-418
hRefEntList.h	B-420
hRefRadRecvr.h	B-422
hRefRadXmitr.h	B-424
hRefRadRvList.h	B-426

hRefRadXmList.h	B-428
hSignalInter.h	B-430
hStartDec.h	B-432
hStartEnc.h	B-433
hStartInter.h	B-434
hStateDecoder.h	B-436
hStateEncoder.h	B-439
hStateMsg.h	B-442
hStopDec.h	B-444
hStopEnc.h	B-445
hStopInter.h	B-446
hUnknownInter.h	B-448
hVcDec.h	B-450
hVcEnc.h	B-451
hVcInter.h	B-452
iffAtcNavBeam.h	B-453
iffAtcNavLayr.h	B-455
iffAtcNavOper.h	B-456
iffAtcNavOpr2.h	B-458
iffAtcNavParm.h	B-459
iffAtcNavPdu.h	B-461
iffAtcNavSyID.h	B-464
intClassDesc.h	B-466
intClassMap.h	B-468
intDecFact.h	B-469
intEncFact.h	B-471
intercmCtlPdu.h	B-473
intercmEntDst.h	B-476
intercmGrpAsg.h	B-478
intercmGrpDst.h	B-480
intercmParRec.h	B-481
intercmSigPdu.h	B-482
interDecoder.h	B-484
interEncoder.h	B-487
interMgr.h	B-489
intHashKey.h	B-491
isGroupOfPdu.h	B-492
isPartOfPdu.h	B-495
laserPdu.h	B-497
lcEnd.h	B-498
lcEofAction.h	B-499
lcHeartbeat.h	B-500
lcInter.h	B-501
lcPause.h	B-502
lcPdu.h	B-503
lcPlay.h	B-506

lcPlayAction.h	B-507
lcPlayFile.h	B-508
lcQuit.h	B-509
lcRecAction.h	B-510
lcRecFile.h	B-511
lcRecord.h	B-512
lcSetTime.h	B-513
lcSkipTime.h	B-514
lcStart.h	B-515
lcStop.h	B-516
lcTimeScale.h	B-517
lcTimeout.h	B-518
leAppearPdu.h	B-519
leArtPartsPdu.h	B-522
leDeadReck.h	B-525
leDetonPdu.h	B-526
leEntityID.h	B-530
leEntityOri.h	B-531
leEntityVel.h	B-533
leFirePdu.h	B-534
leLocEntity.h	B-537
leLocWorld.h	B-538
leOriError.h	B-540
lePosError.h	B-541
leTspiPdu.h	B-542
LibMatrix.h	B-545
linObjStPdu.h	B-550
linSegParam.h	B-552
locObjMgr.h	B-554
MachineTypes.h	B-555
mathTypes.h	B-557
messagePdu.h	B-558
milEntDec.h	B-560
milEntEnc.h	B-562
milPlatDec.h	B-564
milPlatEnc.h	B-566
mineDataPdu.h	B-568
minePerimeter.h	B-574
mineQueryPdu.h	B-575
mineRsNackPdu.h	B-578
mineStatePdu.h	B-580
munEntDec.h	B-583
munEntEnc.h	B-584
netBoolean.h	B-586
NetSocket.h	B-587
NetStructs.h	B-591

netTimeStamp.h	B-598
NetTypes.h	B-599
objClassDesc.h	B-602
objDesList.h	B-605
objectType.h	B-606
objIdHashKey.h	B-607
objIdHashLst.h	B-608
objRefCbInfo.h	B-609
pairSock.h	B-610
pdu.h	B-611
pduCbInfo.h	B-613
pduCbMgr.h	B-614
pduDatRecSet.h	B-615
pduFactory.h	B-616
pduWithData.h	B-617
physEntDec.h	B-621
physEntEnc.h	B-623
pointObjStPdu.h	B-625
printUtil.h	B-628
ProcControl.h	B-629
queryPdu.h	B-630
radXmitCbInfo.h	B-631
radioRecvrDec.h	B-632
radioRecvrEnc.h	B-634
radioRecvrSR.h	B-636
radioXmitDec.h	B-637
radioXmitEnc.h	B-639
radioXmitSR.h	B-641
radRcvrCbInfo.h	B-644
receivePdu.h	B-645
recIntrCbInfo.h	B-646
recordPdu.h	B-647
recordSet.h	B-649
recordSetPdu.h	B-651
recQueryPdu.h	B-653
refAttrCbInfo.h	B-655
refAttrCbList.h	B-656
reflAggList.h	B-658
reflectedAgg.h	B-659
refObjCbInfo.h	B-660
refObjMgr.h	B-661
regCtlCbInfo.h	B-663
regCtlCbList.h	B-664
remObjCbInfo.h	B-666
removeEntRPdu.h	B-667
removePdu.h	B-669

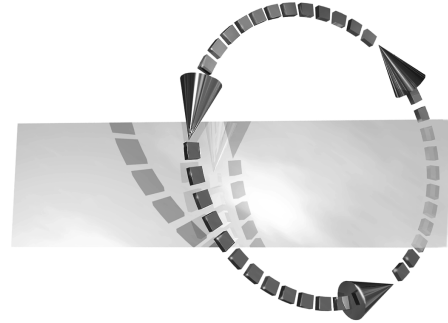
repairPdu.h	B-670
resupplyPdu.h	B-672
rprDecFact.h	B-675
rprEncFact.h	B-676
rprFomMap.h	B-677
rprIDecFact.h	B-679
rprIEncFact.h	B-680
rprInterFact.h	B-681
rprObjFact.h	B-682
script.h	B-683
servReqPdu.h	B-684
setDataPdu.h	B-686
setDataRPdu.h	B-687
setRecordPdu.h	B-689
signalPdu.h	B-691
simanPdu.h	B-693
simpIntWED.h	B-694
smoother.h	B-695
smoothOriWDR.h	B-697
smoothPosWDR.h	B-698
smoothUtil.h	B-699
smUtil.h	B-702
Socket.h	B-703
soldierDec.h	B-706
soldierEnc.h	B-707
startPdu.h	B-709
startResRPdu.h	B-710
stateRep.h	B-712
stIntCbInfo.h	B-713
stIntCbList.h	B-714
stopFrzRPdu.h	B-716
stopPdu.h	B-718
stopUpdCbInfo.h	B-719
stopUpdCbList.h	B-720
str.h	B-722
strHashKey.h	B-724
strHashLst.h	B-725
strtUpdCbInfo.h	B-726
strtUpdCbList.h	B-727
supEmisEsNozz.h	B-729
supEmisEsPdu.h	B-730
supEmisEsProp.h	B-732
taitBryan.h	B-733
thresolder.h	B-734
topoCoord.h	B-736
topoView.h	B-737

Contents

transCtlPdu.h	B-738
transmitPdu.h	B-740
trkJamBeamDec.h	B-743
trkJamBeamEnc.h	B-745
tStampType.h	B-747
undwaAcBeam.h	B-748
undwaAcEmtSys.h	B-750
undwaAcPasAct.h	B-752
undwaAcPdu.h	B-753
undwaAcShaft.h	B-756
unknownHObj.h	B-757
unknownPdu.h	B-759
updCtlCbInfo.h	B-760
updCtlCbList.h	B-761
utmCoord.h	B-763
utmView.h	B-766
varDatumRec.h	B-768
vcPdu.h	B-770
vlList.h	B-777
vlprint.h	B-779
vlTime.h	B-780
vlutil.h	B-781
vlVector.h	B-784
vrlFedAmb.h	B-786
vrlFedTime.h	B-790
vrlRtiAmb.h	B-793

Glossary

Index



Preface

This guide, like VR-Link itself, is intended for use by developers creating simulators and other DIS and HLA applications. It assumes that you are thoroughly familiar with the C++ language and have a working knowledge of DIS and HLA concepts.

We are constantly improving VR-Link to meet user needs. Consequently, some of the latest new features may have been implemented after this documentation went to press. Please read the *RELNOTES* file distributed with your version of VR-Link as well as any supplementary documentation for the most recent information.

How this Guide is Organized

The chapters are organized as follows:

Chapter 1, *Introduction*, describes the main features of VR-Link and contains information about simulation protocols.

Chapter 2, *Installing and Configuring VR-Link*, explains how to install the software, and get technical support.

Chapter 3, *VR-Link Concepts*, provides a general description of what VR-Link does and shows how its classes work together. This chapter also discusses code examples for simple VR-Link applications, and describes compiling and linking procedures.

Chapter 4, *The Protocol-Independent Interface*, describes the most commonly-used VR-Link classes and functions, and shows examples of their use.

Chapter 5, *DIS-Specific Interface*, and Chapter 6, *HLA-Specific Interface*, describe toolkit functionality that pertains only to a specific protocol. Most simulator projects do not need to make use of these “lower-level” features.

Chapter 6, *HLA-Specific Interface*, describes classes and functions that support the High Level Architecture for simulations.

Chapter 7, *Utilities*, explains how to use VR-Link's utility functions to perform time management, coordinate conversions and other tasks.

Chapter 8, *Example and Utility Applications*, explains how to use the example and utility applications that ship with VR-Link, including *buffgrep*, *netdump*, *nethex*, *hlaNetdump*, *vdc*, *vrlinkd3i*, *buffgrep talk*, *listen*, and *fl8*.

Chapter 9, *Guide to Classes and Header Files*, contains cross-reference tables to help you find information about particular classes and header files listed in Appendix A, *PDU Class Reference*.

Appendix A, *PDU Class Reference*, provides descriptive notes about DIS 2.0.3, 2.0.4, IEEE1278.1 and 2.1.4 PDUs. This information appeared in man pages versions of VR-Link prior to version 3.0.

Appendix B, *Header File Reference*, contains edited versions of all VR-Link header files.

About the Hypertext Version

The *./docs* directory contains an on-line version of this document in Adobe's Portable Document Format (PDF). For easy navigation, hypertext links are provided for the entries in the table of contents, for all index entries, and for cross references. PDF files also allow full text searching.

To view and print PDF documents, you need a copy of the free Acrobat Reader software. It is available for SunOS and Solaris, IRIX, HP-UX, Digital UNIX, Windows, and other platforms. To obtain a copy, visit www.adobe.com.

MÄK Products for the Virtual World

The VR-Link is a member of the MÄK Technologies line of HLA/DIS software products designed to streamline the process of developing and using networked simulated environments.

In addition to the VR-Link, the MÄK Technologies product line includes:

- ♦ The MÄK RTI. The MÄK RTI is the first commercial run-time infrastructure for HLA simulations.
- ♦ The MÄK Stealth. The Stealth displays a realistic, three-dimensional view of your virtual world. You can view this world from the inside of a simulated moving vehicle, or place the eyepoint at another moving or stationary location. The Stealth lets you switch rapidly among several predefined viewpoints while the simulation is underway.
- ♦ The MÄK Logger. The *Logger*, also called the *Data Logger*, can record HLA and DIS exercises and play them back for after-action review. You can play a recorded file at speeds above or below normal, or in reverse, and locate areas of interest quickly. The Logger lets you choose between a graphical interface, or a shell-style, command-line interface.

The Logger comes with a set of utility programs that let you inspect, combine and edit Logger recordings.

- ♦ Dial-a-Sim. Dial-a-Sim is a reconfigurable DIS simulator program that lets you create and simulate vehicles quickly, with or without programming. Dial-a-Sim's graphical interface lets you assemble vehicles by choosing from a collection of predefined components. You can adjust the physical properties and positioning of these components to meet your exact needs.

Once your vehicle is complete, you can drive it in a networked simulation. Dial-a-Sim provides a set of on-screen controls, and supports PC-compatible steering wheel assemblies and other hardware controls.

- ♦ The MÄK Plan View Display. The Plan View Display (PVD) provides a two-dimensional, "big picture" of a virtual environment. It shows simulated entities, such as vehicles, soldiers, and tanks moving over a 2D-map display. The MÄK PVD is compatible with both Distributed Interactive Simulation (DIS) and High Level Architecture (HLA) simulation standards.

Contact MÄK Technologies for more information about these products and their supported platforms.

Getting Technical Support

For VR-Link technical support, information about upgrades, and information about other MÄK products, you can reach us in the following ways:

For sales and upgrade information by e-mail:	vrlink-info-sales@mak.com
For technical support by e-mail:	vrlink-tech-spt@mak.com
For general MÄK information via the Web:	http://www.mak.com

Send postal correspondence to:	MÄK Technologies 185 Alewife Brook Parkway Cambridge, MA 02138

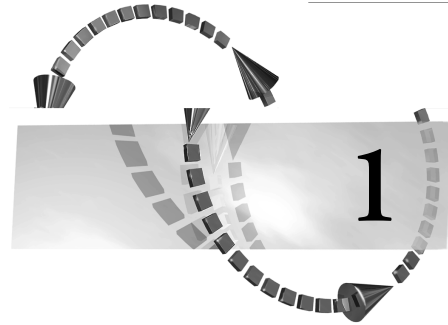
Call or fax us at:	Fax:	617-876-9208
	Voice Phone:	617-876-8085

Document Conventions

This document uses the following typographic conventions:

Monospaced Type	Indicates commands or values you enter.
<i>Monospaced Italic</i>	Indicates variables that you replace with appropriate values.
[]	Indicates optional arguments.
	Separates options in a command where only one option may be chosen at a time.
/	Indicates a directory. Since MÄK products run on both UNIX and PC platforms, we use the / (slash) for generic discussions of pathnames. If you are running on a PC, substitute a \ (backslash) when you type pathnames.
Bold type	Indicates a new term.
<i>Italic</i>	Indicates a file name or path, or a class name.
sansSerif	Indicates a parameter or argument.
>	Indicates a one-step procedure.
Menu→Option	Indicates a menu choice. For example, an instruction to select the Save option from the File menu appears as: Choose File→Save.

Directory names are preceded with dot and slash characters that show their position with respect to the VR-Link home directory. For example, the directory *vrlink/binrpr* appears in the text as *./binrpr*.



Introduction

The *VR-Link Networking Toolkit* is an object-oriented library of C++ functions and definitions that minimize the effort required to create networked simulators and virtual reality applications. VR-Link's powerful, easy-to-use programmer's interface greatly reduces development cost, time and risk.

With VR-Link's protocol-independent API, you can simulate local entities, set their state and automatically send entity information to other applications through either a DIS network or through HLA's Runtime Infrastructure (RTI). VR-Link also simplifies receiving and processing information from other applications, providing easy access to the state of remote entities. VR-Link handles dead reckoning, thresholding, responding to attribute requests, filtering, and many other tasks.

Because VR-Link supports both DIS and HLA through the same API, you can easily switch your applications between the two by changing just a few lines of initialization code and recompiling. This means that your VR-Link-based applications can maintain their DIS compliance vital to ongoing projects, while migrating to HLA.

You can extend VR-Link's default functionality to work with new user-defined DIS PDUs and new HLA objects or interactions. You can also use VR-Link's C++ interface from C applications.

If you are upgrading from a previous version, be sure to read the *RELNOTES* files in the VR-Link home directory for the latest information about new features.

1.1. Main Features

VR-Link provides a range of features to speed the creation or maintenance of DIS and HLA applications:

- ♦ *Support for Multiple Protocols through a Single API.* VR-Link supports version 0.3 of the HLA RPR FOM, as well as DIS versions 2.0.3, 2.0.4 and version IEEE 1278.1-1995. You can easily select the desired protocol at compile time.
- ♦ *C++ Classes for all DIS PDUs and HLA RPR FOM Interactions.* VR-Link's classes provide mutator and accessor functions to set and inspect each field. They can automatically fill out header information and handle variable length messages intuitively. Functions are also provided to print ASCII representations.
- ♦ *C++ Classes for HLA RPR FOM Objects.* These classes handle registration, subscription and removal, and respond to attribute requests automatically.
- ♦ *Exercise Connection Object.* The *exercise connection*, is a VR-Link application's interface to the HLA or DIS exercise. It sends and receives messages through the DIS network or HLA's RTI, manages entity and event IDs, and performs time-stamping on outgoing messages.
- ♦ *Reflected Entity List.* The *reflected entity list* keeps track of participants in your virtual world, and facilitates dead reckoning, trajectory smoothing and filtering. It automatically receives entity state updates, submits attribute update requests as needed, and provides notice when an entity enters or leaves an exercise.
- ♦ *Entity Publisher Object.* An entity's *entity publisher object* automatically sends entity state updates when needed. It also maintains an *entity state repository* that keeps track of state data, including the state of articulated and attached parts.
- ♦ *Utility Functions.* VR-Link has a rich set of utility functions including vector and matrix manipulation functions, a platform-independent interface to the system clock, and support for discrete simulation time. A full set of coordinate conversion utilities is also included.
- ♦ *User Extensibility.* You can define and add your own DIS PDUs or HLA objects and interactions, and override default functionality through subclassing.
- ♦ *Example Applications.* VR-Link comes with a set of DIS and HLA utility programs. For example, the *netdump* utility prints incoming DIS data in an easy-to-read format. The *f18* utility is a simple networked entity simulator that serves as a flexible debugging tool. The source code is provided to demonstrate the use of much of VR-Link's functionality.

1.1.1. Files and Executables in VR-Link

VR-Link includes the header files and libraries necessary to build VR-Link-based applications, plus executables and source code for the utility applications *f18*, *netdump*, *nethex*, *hlaNetdump*, *talk*, *listen* and *vdc*. VR-Link also includes the *vrlinkd3i* packet server and *buffgrep* executables without source code. The unlimited version of the packet server, *vrlinkdu*, is available as an option. You can find more information about these utility applications in Chapter 8, [Example and Utility Applications](#).

1.2. Protocol Versions Supported

DIS only

In DIS, the protocol specifies the content, format and communication mechanism for information exchanged among simulators.

HLA only

HLA allows more flexibility than DIS. The RTI developers determine the wire formats and mechanisms for data delivery, while each individual simulation domain may develop its own *Federation Object Model* (FOM) which defines the types of information to be shared among simulators.

The particular FOM supported by VR-Link 3.x is the Realtime-Platform Reference FOM (RPR FOM), a FOM developed jointly by many companies that have previously used DIS for their simulation networking. The RPR FOM objects and interactions are based on DIS PDUs. VR-Link's HLA libraries support version 0.1.6 of the RPR FOM, plus enhancements that were agreed on at the June 12-13th 1997 RPR FOM meeting. More specifically, VR-Link supports the FOM defined within the Federation Execution Data (FED) file called *VR-Link.fed* stored in the VR-Link home directory.

With version 3.1, VR-Link provides support for DIS version 2.1.4 in addition to HLA. VR-Link continues to support DIS versions 2.0.3, 2.0.4 and the IEEE 1278.1-1995.

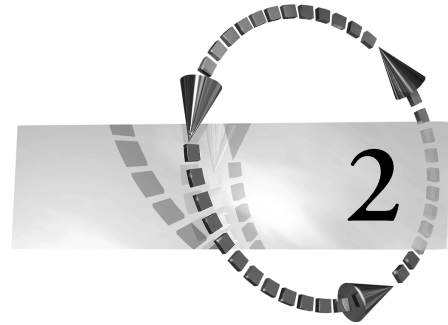
1.2.1. Obtaining Further Information about HLA and DIS

For information about HLA rules and specifications, visit the HLA Web site at:

<http://hla.dmsi.mil/>

For documentation of the IEEE 1278.1-1995 version of DIS, obtain the publication, *1278.1-1995 IEEE Standard for Distributed Interactive Simulation, Application Protocols*. For more information, visit the IEEE World Wide Web site at:

<http://stdsbbs.ieee.org/products/catalog/catalog.html>



Installing and Configuring VR-Link

This chapter covers the following topics:

- ♦ [System Requirements](#), on page 2-2
- ♦ [Installing VR-Link](#), on page 2-4
- ♦ [Configuring the Network for DIS Applications](#), on page 2-13

2.1. System Requirements

VR-Link 3 is available for IRIX6, Solaris, DEC, and Windows NT/95 platforms, and will be available for HP in early 1998. On SunOS4 and IRIX5, RTI support is not available, so VR-Link continues to support DIS-only development on these platforms. Please contact MÄK Technologies for the most current list of supported platforms.

Note: On each platform, we support only the specific C++ compilers listed below.

Silicon Graphics with IRIX6

- ♦ IRIX 6.2 or later
- ♦ MipsPro 7.1 C++ compiler, available from SGI

Note: On SGI, VR-Link uses the N32 ABI. Therefore, when building for IRIX6, use the `-n32` flag on your compile and link lines.

On IRIX6, the RTI requires the following IRIX patches: 1403, 1404, 1644, 1918, 2000, 2044.

Silicon Graphics with IRIX5

- ♦ IRIX 5.3 or later
- ♦ IRIS C++ compiler version 4.0 or later, available from SGI

Sun Sparcstation with SunOS 4

- ♦ Sun Sparcstation with SunOS 4.1.3 or later.
- ♦ GNU C++ compiler (g++) version 2.7.2.3, and g++ libraries version 2.7.2. This software is available in source form via anonymous ftp from *prep.ai.mit.edu/pub/gnu*. Obtain the files *gcc-2.7.2.3.tar.gz* and *libg++-2.7.2.tar.gz*.

Sun Sparcstation with Solaris 2

- ♦ Solaris version 2.5 or later.
- ♦ SPARCompiler C++ version 4.1 or later for Solaris, available from Sun.

DEC Alpha

- ♦ OSF/1 V4.0 operating system.
- ♦ DEC C++ for OSF/1, version 5.5, available from DEC.

PC with Windows NT or Windows 95

- ♦ Windows NT 4.0 or higher.
- ♦ 486-class processor or higher.
- ♦ Microsoft Visual C++ 4.2 or higher.

2.2. Installing VR-Link

This section explains how to install VR-Link on different operating systems and describes the file structure.

2.2.1. Installing VR-Link on UNIX Systems

1. Mount the CD-ROM and run:
`./install`
2. Follow the on-screen instructions.

You can install the VR-Link tree into any directory for which you have write permissions.

2.2.2. Installing VR-Link on PCs

1. From Windows Explorer, run the *Setup.exe* program, located in the root directory of the CD-ROM.
2. Follow the on-screen instructions.

2.2.3. VR-Link Files Installed

The installation process creates a directory tree like that in [Figure 2-1](#). The installation process creates the VR-Link home directory (*vrlinkx.x*) within your installation directory. [Table 2-1](#) describes the contents of the subdirectories.

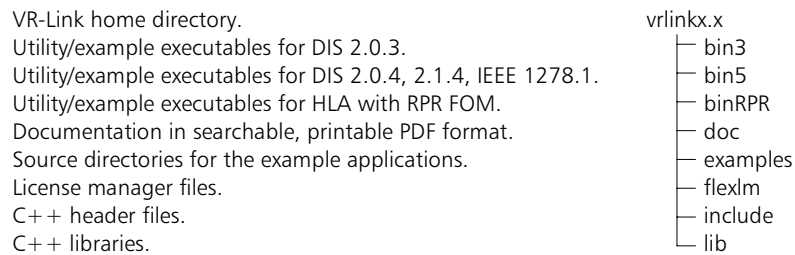


Figure 2-1. VR-Link file structure

Table 2-1: Contents of VR-Link subdirectories

Directory	Directory contents
<i>./bin3</i>	Executables for use with DIS version 2.0.3.
<i>./bin5</i>	Executables for use with DIS versions 2.0.4, 2.1.4, and IEEE 1278.
<i>./binRPR</i>	Executables for use with the HLA RPR FOM.
<i>./doc</i>	VR-Link documentation in a Adobe portable document format (PDF).
<i>./examples</i>	Source directories for the example applications.
<i>./flexlm</i>	Files for the FLEXlm license manager.
<i>./include</i>	C + + header files.
<i>./lib</i>	VR-Link C + + libraries.

2.3. Setting Up and Using the FLEXlm License Manager

VR-Link, like other MÄK products, uses the FLEXlm license manager. Before you can use VR-Link, you must have a license server installed and a valid *license.dat* file. (For information about FLEXlm, view the FAQ page at www.globetrotter.com/flmfaq.htm.) Contact the MÄK Technologies sales department for information about special licensing agreements.

2.3.1. Setting up the License Server

When you install VR-Link, FLEXlm files are installed in the *./flexlm* directory. The procedure you need to follow to set up license management depends on whether or not you have previously installed a MÄK product that uses the license manager.

If you have no other MÄK products installed

The machine on which you run the license manager is the license server. It can be the same machine as the one on which you installed VR-Link, or a different one. If you want to use a different machine as the license server, copy the files in the *./flexlm* directory to that machine.

To set up the FLEXlm license server:

1. Change to the *flexlm* directory on the license server.
2. Execute *lmhostid*. This program generates a unique number that MÄK uses to generate your license activation codes.
3. Call or e-mail MÄK with the name of the server and the number obtained by *lmhostid*. MÄK will e-mail or FTP you a file named *license.dat*.
4. Put the *license.dat* file in the *flexlm* directory on the license server and on all machines on which you have VR-Link installed.
5. On each machine that has a *license.dat* file, set the environment variable LM_LICENSE_FILE to the absolute path to *license.dat*.

Windows 95

On Windows 95, set environment variables in the *Autoexec.bat* file, for example:

```
set LM_LICENSE_FILE="C:\Program Files\MÄK Technologies\  
flexlm\license.dat"
```

Note: Do not put spaces around the = sign. Put quotes around the path.

Windows NT

On Windows NT, you set environment variables in the Environment tab of the Properties dialog box for the machine:

1. Right-click the My Computer icon on the Desktop.
2. On the pop-up menu, select Properties. The System Properties dialog box opens.
3. Select the Environment tab.
4. In the Variable text field, type LM_LICENSE_FILE.
5. In the Value field, type the pathname of the file, for example:

"C:\Program Files\MÄK Technologies\flexlm\license.dat"

UNIX

On UNIX, set the variable similarly to the following example:

```
setenv LM_LICENSE_FILE /usr/local/dir/flexlm/license.dat
```

6. Edit the *license.dat* file to specify the location of the *maklmgrd* executable on the server machine. The file supplied by MÄK represents this location with a dot as shown below:

```
DAEMON maklmgrd .
```

Replace the dot with your absolute path to *maklmgrd*, for example:

```
DAEMON maklmgrd "C:\Program Files\MÄK Technologies\
flexlm\maklmgrd"
```

Note: On PCs, you must enclose the path in quotes.

If you already have a license.dat file from other MÄK products

If you have other MÄK products, you can set up a separate license server for VR-Link, in which case follow the procedure on the previous page. Or, you can merge the license information for VR-Link with that of previous products and use the same license server.

To obtain and merge license information:

1. Change to the *flexlm* directory on the license server.
2. Execute *lmhostid*. This program generates a unique number that MÄK uses to generate your license activation codes.
3. Call or e-mail MÄK with the name of the server, the number obtained by *lmhostid*. MÄK will e-mail or FTP you a file named *license.dat*. The file is in a format similar to the following:

```
SERVER mollusc 006097752f93 4567
DAEMON maklmgrd .
```

```
PACKAGE dir maklmgrd 1999.177 60D00041FA200D8A795D \  
  COMPONENTS=dir1  
INCREMENT dir maklmgrd 1999.177 1-jan-0 1 0BB480511615C175B8CF \  
  PLATFORMS=i86_n
```

Your existing *license.dat* file is similar to the following:

```
SERVER mollusc 006097752f93 4567  
DAEMON maklmgrd "c:\Program Files\MÄK Technologies\stealth\flexlm\  
  maklmgrd"  
PACKAGE makproduct maklmgrd 1999.177 40C020D12D609AB04D55 \  
  COMPONENTS="hprod1 dprod1 prod1"  
INCREMENT makproduct maklmgrd 1999.177 1-jan-0 1 AB4450916DD1944435D2 \  
  PLATFORMS=i86_n
```

Note: The line continuation marks (\) in the examples are for clarity of presentation. They may not be present in the files you receive.

4. Open up the two files in a text editor. Append everything except the Server and Daemon lines (the Server lines will be identical) from the new *license.dat* file to the end of the old *license.dat* file. For example:

```
SERVER mollusc 006097752f93 4567  
DAEMON maklmgrd "c:\Program Files\MÄK Technologies\stealth\flexlm\  
  maklmgrd"  
PACKAGE makproduct maklmgrd 1999.177 40C020D12D609AB04D55 \  
  COMPONENTS="hprod1 dprod1 prod1"  
INCREMENT makproduct maklmgrd 1999.177 1-jan-0 1 AB4450916DD1944435D2 \  
  PLATFORMS=i86_n  
PACKAGE dir maklmgrd 1999.177 60D00041FA200D8A795D \  
  COMPONENTS=dir1  
INCREMENT dir maklmgrd 1999.177 1-jan-0 1 0BB480511615C175B8CF \  
  PLATFORMS=i86_n
```

5. Put the updated *license.dat* file in the *flexlm* directory on the license server and on all machines on which you have VR-Link installed.

2.3.2. Running the License Server

The FLEXlm license server must be running on the server machine and the `LM_LICENSE_FILE` variable must be set on the local machine before you can run your MÄK applications.

- > To start the license server, on the server machine only, execute *runLm* from the *.flexlm* directory.
- > To obtain the current status of the server, run *lmstat*.

Shutting Down the License Server

- > To force all applications to check in their licenses and shut down the license server, run *lmdown*.

2.3.3. Troubleshooting the FLEXlm License Manager

For general troubleshooting information, refer to the FLEXlm FAQ at www.globe-trotter.com/flmfaq.htm. It describes common problems, including the majority that have been reported to MÄK support engineers. Please consult the FAQ first when you have a problem.

Unable to Get A License - Port in Use

If you are unable to get a license for a MÄK application and the FLEXlm log file indicates that the port is in use, it is possible that you have more than one *lmgrd* or *maklmgrd* processes running. It is also possible that an application that was using a license is thought to have terminated, but is still alive. To verify and resolve these possibilities:

1. See if more than one *lmgrd* or *maklmgrd* process is running. On Windows NT, check the Task Manager. On UNIX, enter the following command:

```
ps -aux | grep lmgrd
ps -aux | grep maklmgrd
```

2. If more than one of either process is running, note the process ids (PID) for both executables and kill all of them, for example:

```
kill -9 1385 878
```

where 1385 and 878 are process IDs.

It is important that you kill all the processes to ensure a clean restart for the license server.

3. See if there are old processes running that may be using a license. On Windows NT, check the task manager. On UNIX, use `ps -aux`.

Note: Some UNIX systems use `ps -aux`, others `ps -ef` to check processes. Use the command appropriate for your operating system.

4. If there are old processes present that may be using a license, kill them.
5. If in step 2 you killed the license server, start it with the `runLm` command.

Preventing Multiple License Manager Processes

We recommend that you keep the license server running and that you not start and stop it as you run applications. If you prefer to stop the license server when you are not using it, always use `lmdown` to stop it.

Whenever you start the license manager, first run `lmstat` to be sure that there is not already a license server process running. Following this practice will ensure that you do not start multiple license servers.

2.4. Obtaining and Installing a Runtime Infrastructure (RTI)

An RTI (Run-Time Infrastructure) is a software library (and perhaps supporting executables) that implements the HLA Interface Specification. In HLA, applications exchange FOM data through RTI calls, which means that all HLA applications need to use an RTI. (Applications that use VR-Link typically do not need to make RTI calls directly, but only because VR-Link makes the RTI calls for them.)

Because of differences in the low-level network mechanisms used by different RTI implementations (which include, but are not limited to packet layout), applications that want to interoperate in the same federation execution must use the same RTI implementation. Because RTIs are provided as dynamic libraries that implement a fixed API, a federation can easily switch from one RTI implementation to another between runs (without even recompiling the applications), but during each run, all participants must agree on which RTI to use, much as they must also agree on which FOM to use.

The two RTI implementations against which all MÄK products have been tested are the MÄK RTI (which comes with all MÄK products), and the DMSO RTI (which is available from the DMSO web site).

2.4.1. Installing the MÄK RTI

The MÄK RTI is included with all MÄK products and you have the option of installing it as part of product installation.

UNIX

The installation process creates a link called *RTI* in each product's root directory, which points to the *makRti* directory, in which the MÄK RTI software is located.

PCs

The project files point to the default installation directory for the RTI.

Configuring Your System to Use the MÄK RTI

The RTI dynamic library needs to be located somewhere on your dynamic library search path. This path is indicated by an environment variable as follows:

- ♦ For IRIX6n32 use LD_LIBRARYN32_PATH
- ♦ For DEC Alpha, use PATH
- ♦ On other UNIX platforms use LD_LIBRARY_PATH
- ♦ On PCs, use PATH

The MÄK RTI needs to know where to find the following configuration files:

- ♦ The FED file (*.fed*) for your federation execution (required)
- ♦ The RID file (*rid.mtl*) (optional)

Put the configuration files in the directory from which you are running, or set the environment variable RTI_CONFIG to the directory that contains them.

Running Application with the MÄK RTI

You do not need a server, such as the *rtiexec*, or central application to use the MÄK RTI, however you can run the *rtiexec* if you want to. Be sure the license server is running, then start the applications, and they should be able to communicate. (For information about the license server, see “Setting Up and Using the FLEXlm License Manager”.) See the *MÄK RTI Reference Manual* for information about changing the networking parameters such as multicast addresses, UDP ports, and the *rtiexec*.

Documentation for the MÄK RTI

The documentation for the MÄK RTI consists of:

- ♦ *MÄK RTI Reference Manual*
- ♦ The documentation set for the DMSO RTI. Aside from the installation and configuration instructions, the documentation for the DMSO RTI also applies to the MÄK RTI. It is included on the MÄK product CD.

2.4.2. Installing the DMSO RTI

You can get the DMSO RTI from DMSO's web site (<http://hla.dmsol.com>). Click on Software Distribution Center, where you can register for an account, then later download the RTI software. Documentation for the DMSO RTI is provided on the MÄK product CD because the documentation applies to the MÄK RTI as well.

Consult the DMSO Release Notes to determine version information and whether or not any patches are required.

Install the RTI according to the instructions provided.

Before you use the DMSO RTI for the first time, run the *configRTI* scripts provided in the root directory of any MÄK product. Provide the location of the RTI directory as an argument, as in these examples:

UNIX

```
configRTI /usr/RTI1.3
```

PCs

```
configRTI "c:\program files\rti\rti1.3"
```

configRTI copies the *VR-Link.fed* file from the MÄK products root directory to the DMSO RTI's config directory.

Configuring Your System to Use the DMSO RTI

To configure the environment:

- > On UNIX systems, source the *RTI.env* file.
- > On Windows systems, set environment variables and add the RTI location to your path. The RTI's README file contains detailed instructions.

Running Applications Using the DMSO RTI

To use the DMSO RTI, you need to run the RTI executive (*rtiexec*). This executable resides in the RTI's *bin/platform_name* directory (UNIX) or in the RTI's *bin* directory (PCs). Run *rtiexec* only on one machine, regardless of how many HLA applications you are running.

Once the RTI is running, you can run HLA applications.

2.5. Configuring the Network for DIS Applications

In a typical DIS exercise, VR-Link applications communicate with other DIS applications using broadcast UDP/IP over a local area network. In such a configuration:

- ♦ All applications in the exercise use the same IP broadcast address, and each application's host machine has a network interface device configured for that address.
- ♦ All machines must share a common netmask.
- ♦ All participating applications must use the same UDP port.

Typically, machines on a LAN will already have the correct broadcast addresses, so VR-Link applications will be able to communicate with each other immediately. However, if applications are unable to achieve two-way communication, the network devices may be improperly configured.

2.5.1. Broadcast Address and Netmask (UNIX)

On UNIX-based systems, the `ifconfig` command can be used to examine and set the configuration of network devices. The devices should be configured so that all machines agree on the netmask and on the part of the network address (inet) covered by the netmask.

Because some machines fill unspecified bits in a broadcast address with 1's (as VR-Link does), while others fill it with 0's, VR-Link computes the broadcast address itself using the device's netmask and host address. Thus, the broadcast address used by VR-Link is the same as that reported by `ifconfig` on an SGI, but not the same under SunOS 4.1.x.

Some machines have more than one network device capable of broadcast. Unless your application is designed to specify a specific device or address (which is possible using VR-Link routines), a VR-Link application will use the broadcast address of the first device listed in the interface table by default.

2.5.2. Network Mask (Windows)

Under Windows, VR-Link assumes you are running on a class C network using a netmask of 255.255.255.0. If your machine is configured to use a different netmask, create a *DtNetmask* environment variable to reflect your netmask.

The DtNetmask variable should specify your netmask in a C-language style hexadecimal format. For example:

if your netmask is:	255.255.0.0
the hexadecimal representation is:	0xFFFF0000

The default value of DtNetmask is 0xFFFF0000, which corresponds to the netmask 255.255.255.0.

Setting the Network Mask for Windows NT

To set the DtNetmask variable for Windows NT:

1. Start the Windows NT Control Panel. Open the *System* applet.
2. Enter the DtNetmask variable name and appropriate hexadecimal value in the fields at the bottom of the System dialog box, then click Set.

Setting the Network Mask for Windows 95

- > To set the DtNetmask variable for Windows 95, add the following line to your *AUTOEXEC.BAT* file, then reboot. Replace the hexadecimal number with one that represents your netmask:

```
SET DtNetmask=0xFFFF0000
```

2.5.3. UDP Port

In addition to the broadcast address, the applications in an exercise must also agree on the UDP port number.

The pre-built applications that ship with VR-Link use port 3000 by default (with the exception of *vrlinkd*, which requires that you specify a port number as a command-line argument). Most applications let you set the port number with the **-P port** option.



VR-Link Concepts

This chapter provides a general explanation of how VR-Link works. It covers the following topics:

- ♦ [A Conceptual Overview of VR-Link](#), on page 3-2
- ♦ [Basic VR-Link Examples](#), on page 3-4
- ♦ [Notes about the Examples](#), on page 3-10
- ♦ [Compiling and Linking a VR-Link Application](#), on page 3-13
- ♦ [Compiling and Linking a VR-Link Application](#), on page 3-13

3.1. A Conceptual Overview of VR-Link

A simulator or other DIS/HLA application performs several tasks in order to interact with other players in a virtual world. These tasks typically include:

- ♦ Connecting to an exercise¹. The application must create an object through which it can send and receive messages. That object should provide an efficient way to direct incoming messages to other parts of the application.
- ♦ Managing state information. The application must inform other exercise participants about the states of its own locally generated entities. The application typically must receive, process, and manage state information about remote entities.
- ♦ Managing information about events. The application must inform the world about interactions such as collisions or fire/detonations involving a locally simulated entity, and receive and process events and interactions sent by other applications.

3.1.1. VR-Link Classes in Brief

VR-Link provides classes that facilitate the tasks outlined above:

- ♦ Exercise Connection – A class which serves as the application's interface to the RTI or to the DIS network.
- ♦ Entity management classes – Classes, with a protocol-independent API, that maintain state information of local and remote entities, and handle the sending and receiving of state updates.
- ♦ Interaction classes – Classes that provide a protocol-independent API to either HLA interactions or DIS PDUs that represent events.

At a lower level, VR-Link provides protocol-specific classes, although in most cases, your code will not need to interact directly with them:

- ♦ To represent things having state, VR-Link provides classes for HLA objects and for DIS PDUs that represent state updates.
- ♦ To represent events, classes are provided for HLA interactions and for DIS PDUs representing events.

Finally, to make the job of writing an application easier, VR-Link includes:

- ♦ Utilities – Conversion routines, mathematical functions, and other useful functions.

1. The HLA counterpart to a DIS **exercise** is a **federation execution**. To improve the readability of topics that pertain to both HLA and DIS, we use the term exercise to represent both.

Figure 3-1 shows how classes in a VR-Link application interact. Your application code can interact with classes at any level, and with the RTI or network directly. However, for most applications, you can rely on the higher-level classes (shown with heavy outlines), which are generally easier to use and provide protocol independence.

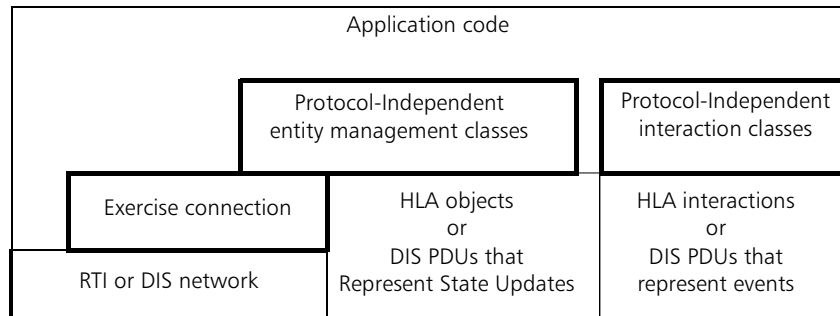


Figure 3-1. Structure of a **VR-Link** Application

3.2. Basic VR-Link Examples

This section describes two simple VR-Link applications. The first is a listen-only application that observes an exercise without simulating any entity on the network. The second is a write-only program that does not process information about remote entities.

3.2.1. A Listen-Only Example

[Example 1](#) illustrates a simple, listen-only VR-Link application. With each iteration of the loop, the program prints an entity's updated, dead-reckoned position in topographic coordinates. In addition, if a fire PDU or interaction is detected on the network, the program prints a message showing the entity ID of the attacker.

Example 1

```
1  #include "stdio.h"
2  #include "exerciseConn.h"
3  #include "reflEntList.h"
4  #include "ProcControl.h"
5  #include "LibMatrix.h"
6  #include "entitySR.h"
7  #include "reflectedEnt.h"
8  #include "fireInter.h"
9  #include "topoView.h"
10
11 int keybrdTick(void);
12
13 // Define a callback to process fire interactions
14 void fireCb(DtFireInteraction *fire, void *usr)
15 {
16     printf("Fire Interaction from %s\n", fire->attackerId().string());
17 }
18
19 main()
20 {
21     // Create a connection to the exercise or federation execution
22     #if DtHLA
23         char *execName      = "VR-Link";
24         char *fedName       = "VR-Link talk";
25         DtExerciseConn exConn(execName, fedName);
26     #else
27         int port            = 3000;
28         int exerciseId      = 1;
29         int siteId          = 1;
30         int applicationNum  = 15;
31         DtExerciseConn exConn(port, exerciseId, siteId, applicationNum);
32     #endif
33
34     // Register a callback to handle fire interactions
35     DtFireInteraction::addCallback(&exConn, fireCb, NULL);
36
37     // Create an object to manage entities that we hear about
38     // on the network.
39     DtReflectedEntityList rel(&exConn);
40
41     // Initialize VR-Link time.
```

```

42   DtTimeInit();
43
44   int forever = 1;
45   while (forever)
46   {
47       // Check if user hit 'q' to quit
48       if (keybrdTick() == -1)
49           break;
50
51       // Tell VR-Link the current value of simulation time
52       DtSetSimTime(DtGetElapsedRealTime());
53       // Process any incoming messages
54       exConn.drainInput();
55
56       // Find the first entity in the reflected entity list
57       DtReflectedEntity *first = rel.first();
58
59       if (first)
60       {
61           // Grab its state repository, where we can inspect its data
62           DtEntityStateRepository *esr = first->entityStateRep();
63
64           // Create a topographic view on the state repository, so we
65           // can look at position information in topographic coordinates
66           double refLatitude = DtDeg2Rad( 35.699760);
67           double refLongitude = DtDeg2Rad(-121.326577);
68           DtTopoView topoView(esr, refLatitude, refLongitude);
69
70           // Print the position
71           printf("Position of first entity: %s\n",
72               topoView.location().string());
73       }
74
75       // Sleep till next iteration
76       DtSleep(0.1);
77   }
78   return 0;
79 }
80 int keybrdTick()
81 {
82     char *keyPtr = DtPollInputLine();
83     if (keyPtr && (*keyPtr == 'q' || *keyPtr == 'Q'))
84         return -1;
85     else
86         return 0;
87 }

```

Before the Loop

Early in the program, in lines [21-32](#), the program creates a *DtExerciseConnection*. This connection serves as the program's interface to an HLA or DIS exercise. A different constructor is used depending on whether the program is compiled for HLA or DIS. This is the only protocol-specific part of the code.

Applications based on VR-Link typically use callbacks to handle interactions such as fire, detonations, and collisions. For example, a callback named *fireCb* is registered with the *FireInteraction* class at line 34. This callback (defined at line 13) prints a message containing the attacker ID. It executes whenever the exercise connection receives a Fire PDU or interaction.

We create a **reflected entity list** in line 39 to keep track of entities found on the network. The entity list tracks the arrival and departure of entities, performs dead reckoning, manages timeouts, and performs other entity-tracking tasks.

Finally, *DtTimeInit* (line 42) initializes VR-Link time. This is necessary for the proper operation of other VR-Link time functions. Make this call once, early in your application if you are calling other VR-Link time functions, either directly or indirectly.

Within the Loop

At the start of each iteration, the program sets *VR-Link simulation time* (line 52) to provide a common time value for use by time-related operations that occur within an iteration of the loop (such as the dead reckoning of multiple entities).

The *drainInput()* call (line 54) reads and processes any PDUs or HLA messages arriving through the exercise connection. This call triggers the execution, if needed, of any callbacks you have registered for that exercise connection.

In line 57, the program finds the first entity in the entity list, then in line 62, retrieves the pointer to the entity's **entity state repository**. Line 68 creates a topographic view of that entity state repository, allowing us to retrieve its position data in topographic coordinates rather than geocentric.

Finally, line 71 obtains and prints the dead-reckoned entity location.

3.2.2. A Send-Only Example

[Example 2](#) illustrates a simple send-only application that simulates the flight of an F18 aircraft. On execution, the program begins by sending a fire PDU or HLA fire interaction. Thereafter, the F18 flies north for 10 seconds, updating its position by sending DIS entity state PDUs or HLA attribute updates.

Example 2

```

1  #include "stdio.h"
2  #include "exerciseConn.h"
3  #include "topoView.h"
4  #include "ProcControl.h"
5  #include "EntityTypes.h"
6  #include "entityPub.h"
7  #include "entitySR.h"
8  #include "fireInter.h"
9
10 main()
11 {
12     // Create a connection to the exercise or federation execution
13     #if DtHLA
14         char *execName      = "VR-Link";
15         char *fedName       = "VR-Link talk";
16         DtExerciseConn exConn(execName, fedName);
17     #else
18         int port            = 3000;
19         int exerciseId      = 1;
20         int siteId          = 1;
21         int applicationNum = 16;
22         DtExerciseConn exConn(port, exerciseId, siteId, applicationNum);
23     #endif
24
25     DtEntityType f18Type(DtPlatform, DtPlatformDomainAir,
26                          DtUnitedStates, DtFighter, DtF18, 0, 0);
27
28     // Create an entity publisher for the entity we are simulating
29     DtEntityPublisher f18EntityPub(f18Type, &exConn,
30                                   DtDrDrmRvw, DtForceFriendly,
31                                   DtEntityPublisher::guiseSameAsType());
32
33     // Hold on to its state repository, where we can set data
34     DtEntityStateRepository *esr = f18EntityPub.entityStateRep();
35
36     // Create a topographic view on the state repository, so we
37     // can set position information in topographic coordinates
38     double refLatitude = DtDeg2Rad( 35.699760);
39     double refLongitude = DtDeg2Rad(-121.326577);
40     DtTopoView f18TopoView(esr, refLatitude, refLongitude);
41
42     // We can use the ESR to set state
43     esr->setMarkingText("VR-Link");
44
45     // Initialize VR-Link time.
46     DtTimeInit();
47
48     DtVector position(0, 0, -100);
49     DtVector velocity(20, 0, 0);

```

```
47
48 // Send a Fire Interaction
49 DtFireInteraction fire;
50 fire.setAttackerId(f18EntityPub.id());
51 exConn.sendStamped(fire);
52 // Main loop
53 DtTime timestep = 0.05;
54 DtTime simTime = 0;
55 while (simTime <= 10.0)
56 {
57     // Tell VR-Link the current value of simulation time
58     DtSetSimTime(simTime);
59
60     // Process any incoming messages
61     exConn.drainInput();
62
63     // Set the current position information
64     f18TopoView.setLocation(position);
65     f18TopoView.setVelocity(velocity);
66
67     // Call tick, which insures that any data that needs to be
68     // updated is sent.
69     f18EntityPub.tick();
70
71     // Set up for next iteration
72     position[0] += velocity[0] * timestep;
73     simTime     += timestep;
74
75     // Wait till real time equals simulation time of next step
76     DtSleep(simTime - DtGetElapsedRealTime());
77 }
78
79 return 0;
80 }
```

Before the Loop

Like the listen-only example, this program creates a *DtExerciseConnection* to provide an interface to the RTI or DIS network (lines 12 through 23).

Line 25 defines the entity type the F18 will use.

To be visible to other applications in the exercise, each locally-simulated entity requires a *DtEntityPublisher*, created here in line 28. The entity publisher manages the generation of messages for this particular entity. It provides an entity state repository where you can set state values, and a `tick()` function, which causes state information to be sent to the network if necessary.

Line 31 sets up a pointer to the entity state repository, then line 37 creates a topographic view on that repository. This lets us set the entity's positional data using topographic coordinates, rather than the original geocentric. Line 40 shows an example of storing a non-positional state value in the repository.

An example of sending an interaction appears in lines 49-51. You can send the interaction using the exercise connection's `sendStamped()` or `send()` function.

Note: You can use these functions to send state updates as well, but it is preferable to let the entity publisher send state updates for you, using data in the entity state repository, as shown in lines 64 through 69.

Within the Loop

The main loop executes twenty times per second for ten seconds. As in the listen-only example, this program sets simulation time at the start of each iteration (line 58).

Note: While this program's main purpose is to demonstrate the sending of data to the network, it is not a true send-only application. Incoming data is also processed with the `drainInput()` call on line 61. This call is required for HLA, as this is where we tick the RTI.

The program updates the F18's positional data in its entity state repository in lines 64 and 65, and ticks the entity publisher in line 69 to send the updated data onto the network.

Thereafter, the only remaining tasks are to increment the F18's position, increment the simulation time, and sleep until it's time to begin the next iteration.

3.2.3. Notes about the Examples

If you are upgrading from a version of VR-Link prior to version 3.x, you will notice several ways in which the previous examples are different from similar programs created with earlier versions of VR-Link. This section briefly describes the differences that are apparent in the examples.

Connecting To An Exercise

The *DtExerciseConn* now has constructors to support both DIS and HLA. As shown in the [Example 1](#), line 25, the HLA constructor requires an execution name and a federate name. For DIS, you create a *DtExerciseConn* using any of the constructors available in previous VR-Link versions.

DtExerciseConn still has `send()`, `sendStamped()`, and `drainInput()` member functions to send and receive data. See “[Connecting to Exercises](#)” on page 4-2 for more information.

Interactions and Callbacks

VR-Link 3.x provides classes to handle DIS and HLA interactions such as fire, detonate or collision events. For de08.2(o(9(e)(o)-2.9x)1A2(t)6r)8(e)9.87(049.9(Twi”ma4(e)-58(e)9.. F)37.n, (ent)42(d)8

Communicating Entity Data

Before VR-Link 3.x, the sending of an entity's state information was handled by *DtOutbound*, and incoming state information was handled by *DtRva*, which maintained a list of entities (*DtDraEntities*) in an exercise.

VR-Link 3.x replaces these three classes with new classes: *DtOutbound* has been replaced by *DtEntityPublisher*, *DtRva* has been replaced by *DtReflectedEntityList*, and *DtDraEntity* has been replaced by *DtReflectedEntity*.

These classes work much like their predecessors, except for their method of setting and retrieving state data. In VR-Link 2.4.6, *DtOutbound* and *DtDraEntity* let you access that data through public members such as *DtLocation* and *DtVelocity*. In VR-Link 3.x, the portion of an entity's state that can be communicated among applications is encapsulated in a *DtEntityStateRepository* object which provides the necessary functions.

Both *DtReflectedEntities* and *DtEntityPublishers* have the function `entityStateRep()` that returns a pointer to a *DtEntityStateRepository*. This is illustrated in [Example 1](#), line 62, and [Example 2](#), line 31.

For an illustration of retrieving and setting state values in the entity state repository, directly and indirectly, see [Example 1](#) (lines 68 and 71) and [Example 2](#) (lines 37, 40, 64, and 65).

When sending, use `DtEntityStateRepository::setLocation()`, `setVelocity()`, etc., to set the entity's state. Then, call

`DtEntityPublisher::tick()`, which decides what information (if any) needs to be sent to other applications, and sends that information (replacing `DtOutbound::DtTick()`).

When receiving, you can use the *DtReflectedEntity*'s *DtEntityStateRepository*'s functions (such as `location()` and `velocity()`) to inspect the current dead-reckoned state of the entity.

For more information, see [dEntityPub.h](#), [hEntityPub.h](#), [entitySR.h](#), [dReflEntList.h](#), [hReflEntList.h](#), [dReflectedEnt.h](#), [hReflectedEnt.h](#), and Chapter 4, *The Protocol-Independent Interface*.

Dead Reckoning

Conspicuous by its absence from the examples is any call to `deadReckonAllEntites()`. This call is no longer needed, since all data access is through member functions, and these functions typically ensure that dead reckoning occurs before they return a result. If you don't want dead reckoning, set the *DtEntityStateRepository*'s approximator to NULL using `DtEntityStateRepository::setApproximator()`.

Using UTM and Topographic Views

In versions of VR-Link prior to version 3.x, you could use *DtTopoOutbound* and *DtUtmOutbound* to set the state of a locally simulated entity in various coordinate systems. VR-Link 3.x and later provides new classes that serve this purpose for both outgoing and incoming data:

- ♦ *DtTopoView*
- ♦ *DtUtmView*
- ♦ *DtCartesianView*

Create a *DtTopoView* (or *DtUtmView*) on a *DtEntityPublisher*'s *DtEntityStateRepository* for outgoing data (as in [Example 2](#), line 37), or on a *DtReflectedEntity*'s *DtEntityStateRepository* for incoming data ([Example 1](#), line 68). You can then use the view's inspectors or mutators instead of *DtEntityStateRepository*'s. For more information, see [cartView.h](#), [topoView.h](#), and [utmView.h](#).

3.3. Compiling and Linking a VR-Link Application

To build a VR-Link application, you must tell the compiler where to find the VR-Link header files (and for HLA, the RTI header files). In addition, you must link with VR-Link's libraries (and for HLA, the RTI's libraries). If you are not using a C++ linker to link, you must explicitly link with C++ system libraries such as *libC.a*.

3.3.1. Libraries and Header Files

VR-Link consists of four libraries: *mtl*, *vl*, *matrix*, and *vlutil*. The following versions of the *vl* library are provided:

	UNIX:	Windows:
– For HLA using the RPR FOM	<i>libvlRPR.a</i>	<i>vlRPR.lib</i>
– For DIS 2.0.4, IEEE1278.1, or 2.1.4	<i>libvl5.a</i>	<i>vl5.lib</i>
– For DIS 2.0.3	<i>libvl3.a</i>	<i>vl3.lib</i>

For each library, we have provided a single header file that includes all other header files in that library. The files are:

- ♦ *vlInc.h*
- ♦ *mtlInc.h*
- ♦ *matrixInc.h*
- ♦ *vlutilInc.h*

For convenience, you can include just these four general header files. However, you might speed up compilation of your source files by including only the header files you need (as our examples do).

On the PC platform, VR-Link uses multithreaded (MT) libraries by default, but also provides multithreaded DLL (MD) libraries. They are in the *.\lib\libmd* directory.

3.3.2. Compiling for a Particular Protocol Version under UNIX

You can select your program's protocol at compile time by using the flags described in the following sections. *vlhome* represents your VR-Link home directory. RTI_HOME and RTI_ARCH come from the environment, assuming you've configured the environment according to the instructions in section 2.4, [“Obtaining and Installing a Runtime Infrastructure \(RTI\)”](#).

Note: The order of the libraries in the link line is significant. See the Makefiles in *./examples* for further guidance.

Table 3-1:

Protocol	Compile flag	Link flag
HLA	-Ivlhome/include -I\${RTI_HOME}/lang/C++/include (path to the RTI include directory) -DDtDISVERS=5 -DDtHLA=1 -DDtRPR=1 (for conditional compilation)	-Lvlhome/lib -lmtl -lvlRPR -lmatrix -lvlutil -L\${RTI_HOME}/lang/C++/lib /\${RTI_ARCH} -lRTI -lpthread
DIS 2.0.4, 2.1.4 or IEEE 1278.1	-Ivlhome/include -DDtDISVERS=5 (for conditional compilation)	-Lvlhome/lib -lmtl -lvl5-lmatrix -lvlutil
DIS 2.0.3	-Ivlhome/include -DDtDISVERS=3 (for conditional compilation)	-Lvlhome/lib -lmtl -lvl3-lmatrix -lvlutil
IRIX6	-n32 for HLA and DIS -exceptions for HLA only	-n32 for HLA and DIS -exceptions for HLA only

3.3.3. Selecting an Example Application's Protocol under UNIX

When you build the example programs, you can choose the protocol without editing the Makefile. Use the VERS argument, as shown in these examples for *netdump*:

```
make VERS=3 netdump      for DIS 2.0.3
make VERS=5 netdump      for DIS 2.0.4, IEEE 1278.1 or 2.1.4
make VERS=RPR netdump    for HLA using the RPR FOM
```

Depending on the value you specify for VERS, a particular file is included in the Makefile, as determined by the following line in the Makefile:

```
include ./makeMods.mk$(VERS)
```

The included file is *makeMods.mk3*, *makeMods.mk5*, or *makeMods.mkRPR* (which are in the example program's source directory). By including a *makeMods* file in this manner, we effectively insert version-specific commands that control what is built.

If desired, you can still choose the protocol by editing the Makefile manually. To do so, comment and uncomment the appropriate lines according to the instructions in the file. You'll also need to remove the include line shown above, or replace the *makeMods* files with empty versions.

3.3.4. Compiling for a Particular Protocol Version under Windows

Under Windows, you can set your program's protocol following the procedures in this section. You need to provide build settings in Microsoft Developer Studio to define constants and specify paths for header files and RTI libraries.

When Compiling for HLA

1. Choose Build→Settings, then select the C/C++ tab (Figure 3-2).

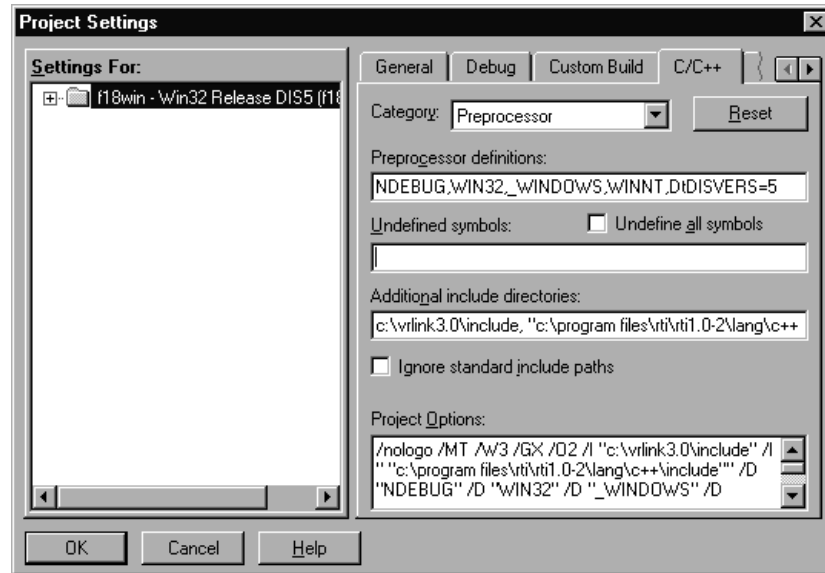


Figure 3-2. Developer Studio Project Settings

2. Enter the following #define symbols in the space provided for preprocessor definitions:

```
DtDISVERS=5, DtHLA=1, DtRPR=1
```

3. In the space provided for additional include libraries, enter your VR-Link include path and the path to your RTI include directory, for example:

```
..\..\include, "c:\program files\rti\rti1.0-2\lang\c++\include"
```

4. Select the Link tab.

5. In the space provided for object/library modules, enter the appropriate RTI, VR-Link and Visual C++ library files, for example:

```
"c:\program files\rti\rti1.0-2\lang\c++\lib\release\rti10.lib"
..\..\lib\vlutil.lib ....\lib\mtl.lib ....\lib\matrix.lib
..\..\lib\vlRPR.lib netapi32.lib wsock32.lib
```

When Compiling for DIS 2.0.4, 2.1.4 or IEEE 1278.1

1. On the C/C++ page under Build→Settings, enter the following #define symbol in the space provided for preprocessor definitions:

```
DtDISVERS=5
```

2. In the space provided for additional include libraries, enter your VR-Link include path, for example:

```
..\..\include
```

3. On the Link page, enter the appropriate VR-Link and Visual C++ library files, for example:

```
..\..\lib\vlutil.lib ....\lib\mtl.lib ....\lib\matrix.lib  
..\..\lib\vl5.lib netapi32.lib wsock32.lib
```

When Compiling for DIS 2.0.3

1. On the C/C++ page under Build→Settings, enter the following #define symbol in the space provided for preprocessor definitions:

```
DtDISVERS=3
```

2. In the space provided for additional include libraries, enter your VR-Link include path, for example:

```
..\..\include
```

3. On the Link page, enter the appropriate VR-Link and Visual C++ library files, for example:

```
..\..\lib\vlutil.lib ....\lib\mtl.lib ....\lib\matrix.lib  
..\..\lib\vl3.lib netapi32.lib wsock32.lib
```

3.3.5. Selecting an Example Application's Protocol under Windows

In the Microsoft C++ IDE, you can choose a protocol for an example application by choosing the desired protocol from the Default Project Configuration list.

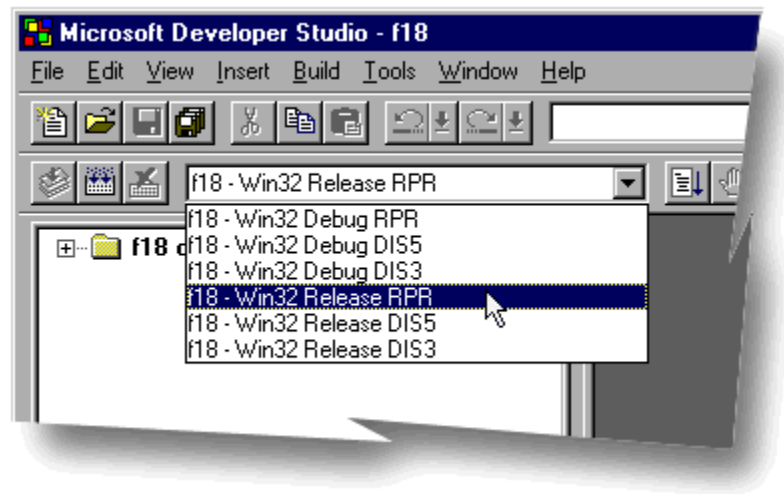


Figure 3-3. Selecting a protocol in the Microsoft C++ IDE.

3.4. Using VR-Link from C

The directory *./examples/simpleC* contains an example of using VR-Link with an application written in C. The example is a C version of the listen-only program described in section 3.2.1, “A Listen-Only Example”. See the header files *interface.h* and *simpleC.h* for details.

Calling C++ from C

Since VR-Link is a C++ toolkit, any code that directly calls a VR-Link function must be compiled by a C++ compiler. You can compile this code in such a way that it may be called from C. To do so, you need at least one C++ file from which all calls to VR-Link functions can be made. In our example, the file *interface.cc* serves this purpose. It contains several functions that interact with VR-Link code: *initVRLink()*, *tickVRLink()*, and *firstEntityPosition()*. To alert the C++ compiler that these functions should be compiled to be callable from C, their declarations are surrounded by the following keywords, as seen by the C++ compiler:

```
extern "C" {  
}
```

The file *listenC.c* represents the rest of the application. It is compiled by a C compiler, and does not directly call any VR-Link functions. It does, however, call the functions defined in *interface.cc* that were compiled to be called from C.

Calling C from C++

You can create a C++ function that interfaces with VR-Link code and can also call C functions. For example, in *interface.cc*, the callback on Fire PDUs (*fireCb*) is a C++ function which calls the C function *firePrint()*.

To alert the C++ compiler that an external function is a C function, its declaration is surrounded by the following keywords, as seen by the C++ compiler:

```
extern "C" {  
}
```

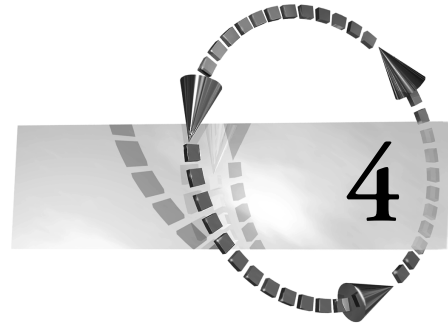
Header Files that are Included by C and C++

When C and C++ code interact, it is necessary that the declarations of many of the same functions be included by both C and C++ code. As described above, these declarations need to be surrounded by

```
extern "C" {  
}
```

when presented to the C++ compiler. However, these keywords are meaningless to a C compiler, so that they should always be surrounded by:

```
#ifdef __cplusplus  
#endif
```

The Protocol-Independent Interface

You can access most of VR-Link's features through its protocol-independent API, described in this chapter. Protocol-specific portions of the API are described in Chapter 5, *DIS-Specific Interface* and Chapter 6, *HLA-Specific Interface*.

Topics covered in this chapter include:

- ♦ [Connecting to Exercises](#), on page 4-2
- ♦ [Managing Interactions](#), on page 4-7
- ♦ [Managing Locally Simulated Entities](#), on page 4-11
- ♦ [Managing Remote Entities](#), on page 4-15
- ♦ [Dead Reckoning](#), on page 4-20
- ♦ [Intercepting State Updates](#), on page 4-22
- ♦ [Timeouts](#), on page 4-23
- ♦ [Coordinate Views](#), on page 4-25
- ♦ [Articulated Parts](#), on page 4-28

4.1. Connecting to Exercises

In general, an HLA federation or DIS exercise can be defined as a collection of simulation applications communicating with each other through the exchange of FOM-defined messages (for HLA) or PDUs (for DIS). A simulation application (also called a federate in HLA) can in turn be defined as a participant in an exercise. Such an application may simulate one or more entities, with each entity having a unique entity ID.

VR-Link's class *DtExerciseConn* is a simulation application's interface to an exercise. The application sends messages that it generates, and receives messages from the network, through its *DtExerciseConn*. Virtually all VR-Link-based applications make use of a *DtExerciseConn*.

There are both HLA and DIS versions of *DtExerciseConn*. The HLA version is defined in *hExerciseConn.h*. The DIS version is defined in *dExerciseConn.h*. Rather than include both, you can simply include *exerciseConn.h*, which will include the appropriate version based on whether or not you have included the *DtHLA=1* definition in your compile line.

Although the two versions of *DtExerciseConn* are defined separately, most public member function names and prototypes are shared across the two versions, so that you can make most of these calls regardless of which protocol you are using. Much of this commonality is enforced because both versions are derived from *DtBaseExerciseConn*, an abstract class that defines much of the *DtExerciseConn* interface.

Creating a *DtExerciseConn* must be done differently for HLA and DIS because different initialization parameters are required.

4.1.1. Creating an Exercise Connection for HLA

An exercise connection for HLA requires a federation execution name and a federate ID.

The federation execution name must match the name of a *.fed* file that appears in the *config* directory within your RTI installation. When you run the *configRTI* script during VR-Link installation, a file called *VR-Link.fed* is created in that directory, meaning you can use the federation execution name VR-Link.

If you want to use a federation name other than VR-Link, rename the *.fed* file in the RTI's */config* directory.

The same federation execution name must be used by all applications that wish to interact in the same HLA world.

The federate name is a name you give to your application. It is used for RTI diagnostics. Federate names do not have to be unique in the federation.

The standard *DtExerciseConn* constructor initializes the application's RTI ambassadors (which causes the *.fed* file to be read), creates the named federation execution if it does not already exist, and joins the execution. If you are using the DMSO RTI, make sure that the *rtiexec* is running, otherwise you will not be able to create or join the federation execution. *rtiexec* is optional if you use the MÄK RTI. (See [“Obtaining and Installing a Runtime Infrastructure \(RTI\)”](#) on page 2-10).

4.1.2. Creating an Exercise Connection for DIS

To create an exercise connection for DIS, you need a:

- ♦ UDP port number
- ♦ exercise ID number
- ♦ site number
- ♦ application number

All applications that want to interact in a DIS exercise must use the same UDP port number and exercise ID number.

If a Packet Server is running on the specified UDP port, the *DtExerciseConn* will connect to that Packet Server, rather than connecting directly to the network. (For information about the Packet Server and its benefits, see [“The VR-Link Packet Server \(DIS and UNIX Only\)”](#) on page 8-12.)

By default, DIS PDUs sent through the *DtExerciseConn* are sent to the broadcast address of the machine's primary network interface. The *DtExerciseConn* will receive any unicast or broadcast packets destined for the machine on the specified port. Additional constructors and functions of the DIS version of *DtExerciseConn* allow greater flexibility in setting up your network connection, and are discussed in [“Configuring Your Connection to the DIS Network”](#) on page 5-21.

4.1.3. Compiling Applications for Protocol-Independence

Applications that can be built for both DIS and HLA typically use conditional compilation as in this example:

```
#if DtHLA
    char *execName      = "VR-Link";
    char *fedName       = "myApp";
    DtExerciseConn exConn(execName, fedName);
#else
    int port            = 3000;
    int exerciseId      = 1;
    int siteId          = 1;
    int applicationNum  = 15;
    DtExerciseConn exConn(port, exerciseId, siteId, applicationNum);
#endif
```

4.1.4. *DtExerciseConn* Member Functions

The most commonly used protocol-independent member functions for *DtExerciseConn* are described in [Table 4-1](#).

Table 4-1: DtExerciseConn member functions

Function	Description
send() sendStamped()	Sends interactions to the exercise (See “Managing Interactions” on page 4-7.)
drainInput()	Causes VR-Link to read and process input from the network. The HLA version repeatedly calls RTI::tick() until it fails to return RTI::RTI_TRUE (meaning there is no more data waiting to be read). The DIS version reads PDUs from the network directly, or from the Packet Server, depending on configuration. See “Using drainInput()” on page 4-4.
nextId()	Obtains an ID for use with an entity (or other object) that you are simulating. Both the HLA and DIS versions of nextId() return an object of the type <i>DtObjectId</i> , defined in <i>hostStructs.h</i> . See “Using nextId() in HLA” on page 4-5 and “Using nextId() in DIS” on page 4-5.
nextEventId()	Both the DIS and HLA versions of nextEventId() work similarly to the DIS version of nextId(). You can use nextEventId() to generate consecutive event IDs for use in fire, detonate or collision interactions, or you can simply choose your own event IDs in both the DIS and HLA cases. Event IDs, being part of the RPR FOM rather than a fundamental HLA concept, are not managed by the RTI as Object IDs are. Therefore it is up to the application to ensure that event IDs are unique.
addPostDrainCallback()	Allows you to register functions with the <i>DtExerciseConn</i> that you wish to have called automatically each time drainInput() is called. See “Callbacks” on page 4-5.
removePostDrainCallback()	Allows you to unregister functions with the <i>DtExerciseConn</i> that you wish to have called automatically each time drainInput() is called. See “Callbacks” on page 4-5.

Using drainInput()

From within *DtExerciseConn::drainInput()*, VR-Link executes any callbacks that you have registered on *DtInteractions* (for either HLA or DIS), *DtPdus* (for DIS), or various RTI services (for HLA).

Your application should call *DtExerciseConn::drainInput()* periodically, usually once per simulation loop. This is true for HLA even if your application doesn’t care about remotely simulated objects or remotely generated interactions, as the RTI requires RTI::tick() to be called periodically.

The optional timeout argument to `drainInput()` lets you specify a time (in seconds) after which `drainInput()` should return even if there is more data left to be read. A value of -1 (the default) is an infinite timeout. A value of 0 means return immediately. The less important second argument to `drainInput()` specifies the number of packets to be read or the number of times to call `RTI::tick()` in between tests of whether timeout has been exceeded.

Using `nextId()` in HLA

All object IDs should be obtained through `DtExerciseConn::nextId()` (or directly from the RTI). `DtExerciseConn::nextId()` gets an ID from the RTI (which ensures that it will be unique in the exercise.)

DtObjectId is a wrapper around the `RTI::ObjectID()` – a single number that the RTI uses to identify an object.

Using `nextId()` in DIS

In DIS, you may choose your own entity IDs rather than obtain them using `nextId()`.

In DIS, *DtObjectId* is typedefed to *DtEntityIdentifier* (also defined in [hostStructs.h](#)) – a class that represents the DIS identification triplet of site ID, application number and entity number. *DtObjectId* is used rather than *DtEntityIdentifier* or `RTI::ObjectID()` when we want to refer to an identifier in a protocol-independent fashion.

The DIS version returns an identifier consisting of the *DtExerciseConn*'s site and host, and an entity number that is one greater than the one returned in the previous call to `nextId`. There is no guarantee that the entity ID is unique in the DIS exercise.

Callbacks

Post drain callbacks are called from `drainInput()` immediately before it returns, and after all callbacks on interactions, PDUs, or RTI services have been made.

Note: This point is significant, because the RTI does not allow calls to any RTI services from within RTI callbacks. RTI calls can be made freely from within post drain callbacks.

Post drain callbacks should be regular C-style functions or static member functions, but cannot be non-static member functions. They must have the following prototype:

```
void myCallback(void* usr);
```

The value passed to your callback function as **usr** will be the value that was passed as the **usr** argument to `addPostDrainCallback()`. Often, this **usr** argument is used to pass a pointer to an object on which you would like to call a member function from within your callback. Cast **usr** back to the object's type within the callback. For example:

```
class MyObj
{
public:
    void someFunc();
    static void theCallback(void* usr);
};

void MyObj::theCallback(void* usr)
{
    MyObj* obj = (MyObj*) usr;
    obj->someFunc();
}

main()
{
    DtExerciseConn exerciseConn(...);
    ...
    MyObj obj;
    exerciseConn.addPostDrainCallback(MyObj::theCallback, &obj);
}
```

In cases where you would like to pass more than one object to your callback, you must create a simple structure consisting of pointers to your objects, then pass a pointer to the structure as **usr**.

Note: Any pointer you pass as **usr** must be the address of a variable whose lifetime is at least as long as the period for which the callback function is registered.

4.2. Managing Interactions

Within VR-Link, we use the term **interaction** to refer to the messages that are exchanged among simulation applications in an exercise to describe events, such as the firing of a munition, detonation of a munition, or collision of entities. While the term is typically used only in HLA (rather than DIS), we use it in a protocol-independent sense to refer to an HLA interaction, or to a DIS PDU that describes an event.

For example, there are both DIS and HLA versions of the classes *DtFireInteraction*, *DtDetonationInteraction*, and *DtCollisionInteraction*, representing fire, detonate and collision interactions respectively.

Although the two versions of each of these classes are defined separately, most of the public member function names and prototypes are shared across the two versions, so you can use most of them in the same way regardless of whether you are using DIS or HLA.

For both DIS and HLA, the individual interaction classes are derived from *DtInteraction*. *DtInteraction* is defined in *hInteraction.h* (HLA) and *dInteraction.h* (DIS). If you include *interaction.h*, an appropriate version will be included.

The HLA versions of *DtFireInteraction*, *DtDetonationInteraction*, and *DtCollisionInteraction* are in *hFireInter.h*, *hDetInter.h*, and *hCollideInter.h*. The DIS versions are in *firePdu.h*, *detonatePdu.h*, and *collidePdu.h*. However, you can simply include *fireInter.h*, *detonateInter.h* or *collideInter.h* in your applications. These header files include the appropriate version of the classes depending on the protocol for which you are building.

DIS only

For DIS applications, *DtFireInteraction*, *DtDetonationInteraction*, and *DtCollisionInteraction* are equivalent to *DtFirePdu*, *DtDetonationPdu*, and *DtCollisionPdu*. That is, one is typedefed to the other. You can use the *DtPdu* class names if your application is only for DIS. However, you must use the *DtInteraction* names if you are writing code for both DIS and HLA, and this is what we recommend.

In DIS, *DtInteraction* is derived from *DtPdu*. The *DtInteraction* class itself has no real functionality, but the extra level gives our class hierarchy some structure. For example, *DtFirePdus* are (and can be implicitly cast to) *DtInteractions* and *DtPdus*. Entity state PDUs are not interactions, and thus *DtEntityStatePdu* is derived from *DtPdu*, but not from *DtInteraction*.

4.2.1. Sending Interactions

To send interactions without regard to the protocol:

1. Create an interaction class (derived from *DtInteraction*).
2. Use its mutator functions to set values for the different parameters.
3. Send it using *DtExerciseConn*'s *send()* or *sendStamped()* functions.

The following code is an example of creating and sending a fire interaction, which informs the exercise that you have fired a munition. The example assumes that *myId*, *targetId*, and *missileId* are existing *DtObjectIds*.

```
DtExerciseConn exerciseConn(...);

...
// Create a DtFireInteraction
DtFireInteraction fireInter;

// Fill the DtFireInteraction with data
fireInter.setAttackerID(myId);
fireInter.setTargetID(targetId);
fireInter.setMunitionId(missileId);
fireInter.setEventId(exerciseConn.nextEventId());
...

// Send to the exercise
exerciseConn.sendStamped(fireInter);
```

For a description of the mutator functions for a particular interaction class, see the appropriate header files.

HLA only

For HLA, `send()` and `sendStamped()` are identical.

DIS only

For DIS, `sendStamped()` sends a copy of the PDU, which has the exercise ID and time-stamp fields in the PDU header set to the *DtExerciseConn*'s exercise ID and the current time. `send()` does not touch the PDU header.

4.2.2. Receiving Interactions

Incoming interactions are handled through callbacks – functions that application developers write, and register with VR-Link's *DtInteraction* classes.

Each interaction class has static `addCallback()` and `removeCallback()` member functions. These callback functions allow you to register and unregister functions that you want to have called automatically when that type of interaction is received during a call to `DtExerciseConn::drainInput()`.

Interaction callbacks should be regular C-style functions or static member functions of a class, but cannot be non-static member functions. There is a different callback function type for each interaction class. For example, only callback functions that take a *DtFireInteraction* pointer as an argument can be registered with *DtFireInteraction*. A *DtFireInteraction* callback might look like this:

```
void myFireCallback(DtFireInteraction* inter, void* usr)
{
    printf("Got a Fire Interaction from %s\n",
        inter->attackerId().string());
}
```


It would be registered like this:

```
DtFireInteraction::addCallback(exerciseConn,
                               myFireCallback, NULL);
```

Note: Since `addCallback()` is a static member of *DtFireInteraction*, it can be called by simply using the qualified name, rather than calling it on a particular instance of *DtFireInteraction*.

When the callback function is executed by VR-Link, the interaction that triggered the callback will be passed to the function as the first argument.

The value passed to your callback function as `usr` will be the value that was passed as the `usr` argument to `addCallback()`. Often, this `usr` argument is used to pass a pointer to an object on which you would like to call a member function from within your callback. Simply cast `usr` back to the object's type within the callback. For example:

```
class MyObj
{
public:
    void someFunc(DtFireInteraction* inter);
    static void theCallback(DtFireInteraction* inter, void* usr);
};

void MyObj::theCallback(DtFireInteraction* inter, void* usr)
{
    MyObj* obj = (MyObj*) usr;
    obj->someFunc(inter);
}

main()
{
    DtExerciseConn exerciseConn(...);
    ...
    MyObj obj;
    DtFireInteraction::addCallback(&exerciseConn,
                                   MyObj::theCallback, &obj);
}
```

In cases where you would like to pass more than one object to your callback, you must create a simple structure consisting of pointers to your objects, then pass a pointer to the structure as `usr`.

Note: Any pointer you pass as `usr` must be the address of a variable whose lifetime is at least as long as the period for which the callback function is registered.

When an interaction is received for which there are multiple callbacks registered, the callbacks are called in the opposite order from that in which they were registered.

DtInteractions are transient, that is, they are deleted by VR-Link immediately after the last callback function registered for the interaction is called. Therefore, do not attempt to delete or save a pointer to the *DtInteraction* passed to your callback function. If you need to save the data, make a copy of the *DtInteraction*.

Managing Callbacks in HLA

In HLA applications, interaction callbacks are called from within `RTI::tick()` (which is called by `DtExerciseConn::drainInput()`). The RTI does not allow calls to RTI services from within other RTI service calls (including `RTI::tick()`). This means that you cannot directly or indirectly make any RTI calls from within your interaction callbacks.

If you need to make RTI calls based on the receipt of a *DtInteraction*, you can accomplish this as follows:

- From within your interaction callback, use `DtExerciseConn::postDrainCallback()` to register a function with VR-Link that will get executed right before `drainInput()` returns, when it is safe to make RTI calls.

For example, the following code will ensure that your application sends out a fire interaction each time it receives a fire interaction:

```
void sendFire(void *usr)
{
    // Cast the usr pointer back to a DtExerciseConn
    DtExerciseConn* exConn = (DtExerciseConn*) usr;

    // Create and send a DtFireInteraction
    DtFireInteraction fireInter;
    fireInter.setAttackerId(...);
    ...
    exConn->sendStamped(fireInter);

    // Remember to deregister ourselves, so that we don't get
    // called again the next time drainInput is called.
    exConn->removePostDrainCallback(sendFire, exConn);
}

void fireCb(DtFireInteraction* inter, void *usr)
{
    // Cast the usr pointer back to a DtExerciseConn
    DtExerciseConn* exConn = (DtExerciseConn*) usr;

    // Can't send a fire interaction here, since that
    // would involve an RTI call from within an RTI-service
    // callback. So instead, register a fire-sending function
    // (sendFire) as a postDrainCallback.
    exConn->addPostDrainCallback(sendFire, exConn);
}

main()
{
    DtExerciseConn exerciseConn(...);
    ...
    // Register sendFire as a fireInteraction callback,
    // passing a pointer to the DtExerciseConn as the usr pointer,
    // so that it will be available to us within the callback.
    DtFireInteraction::addCallback(&exerciseConn, fireCb,
                                   &exerciseConn);
    ...
}
```

4.3. Managing Entities

VR-Link provides many classes that facilitate informing the exercise about the state of your locally simulated entities and informing your application about the state of remotely simulated entities. These classes include:

- ♦ *DtEntityStateRepository* encapsulates an entity's state
- ♦ *DtEntityPublisher* sends updates to the exercise
- ♦ *DtReflectedEntityList* keeps track of remote entities
- ♦ *DtReflectedEntity* represents a single remote entity

4.3.1. Managing Locally Simulated Entities

In order to inform the world about the state of locally-simulated entities with VR-Link, you must:

1. Create a *DtEntityPublisher* for each locally-simulated entity.
2. Update its current state at every frame through its *DtEntityStateRepository*.
3. Call *DtEntityPublisher::tick()*.

tick() ensures that any needed information is sent to the network through the exercise connection. *DtEntityPublisher*'s destructor removes the entity from the exercise.

Creating a *DtEntityPublisher*

There are both HLA and DIS versions of *DtEntityPublisher*, defined in [hEntityPub.h](#) and [dEntityPub.h](#) respectively. Rather than include both, you can include *entityPub.h*, which will include the appropriate version based on whether or not you have included the *DtHLA=1* definition in your compile line.

Although the two versions of *DtEntityPublisher* are defined separately, most of the public member function names and prototypes are shared across the two versions, so you can make most of these calls regardless of which protocol you are using.

A protocol-independent instance of *DtEntityPublisher* can be created through either of two constructors. Both take a *DtEntityType*, and a *DtExerciseConn* as arguments. The *DtEntityPublisher* sends its entity updates through the *DtExerciseConn* that you pass to the constructor.

The *DtEntityType* (defined in [hostStructs.h](#)) represents a seven-component enumeration defined by the DIS protocol and by the RPR FOM. It is used in outgoing entity state PDUs and in HLA updates, and also determines what HLA object class is used to represent the entity in an HLA exercise. Please see the DIS Data Dictionary for a comprehensive list of valid entity types.

All versions of the *DtEntityPublisher* constructor have an optional final argument that specifies the ID to use for the entity. IDs are represented by the type *DtObjectId* (defined in [hostStructs.h](#)).

HLA only

In HLA, *DtObjectId* is a wrapper around the `RTI::ObjectID()` – a single number that the RTI uses to identify an object.

DIS only

In DIS, *DtObjectId* is typedefed to *DtEntityIdentifier* (also defined in [hostStructs.h](#)) – a class that represents the DIS identification triplet of site ID, application number and entity number.

Use *DtObjectId* when you want to refer to an identifier in a protocol-independent fashion.

The default behavior is to have the *DtEntityPublisher* obtain an ID automatically from the *DtExerciseConn* (which, for HLA, requests an ID from the RTI). If you specify your own entity ID, you should still obtain it using `DtExerciseConn::nextId()` (or, for HLA, directly from the RTI).

The protocol-independent constructor that takes the following arguments:

```
DtEntityPublisher::DtEntityPublisher(const DtEntityType& type,  
    DtExerciseConn* conn,  
    const DtObjectId &id = findOwnId());
```

The other constructor takes three additional arguments not likely to change during a simulation: dead reckoning algorithm, guise (alternate entity type), and force ID:

```
DtEntityPublisher(const DtEntityType& type,  
    DtExerciseConn *conn,  
    DtDeadReckonTypes algorithm,  
    DtForceType forceId,  
    const DtEntityType& guise = guiseSameAsType(),  
    const DtObjectId& id = findOwnId());
```

The following example creates a *DtEntityPublisher*. The entity type is that of an M1A1 tank. Entity type enumerations can be found in the DIS Data Dictionary, which is also referenced by the RPR FOM in HLA. The optional **DtObjectId** argument is omitted, meaning that the *DtEntityPublisher* will choose its own ID.

```
DtExerciseConn exConn(...);  
DtEntityType tankType(1, 1, 225, 1, 1, 0, 0);  
DtEntityPublisher tankPub(tankType, exConn);
```

Other protocol-dependent constructors exist that allow you to specify explicit parameters such as the HLA object class to represent the object. These are described in the chapters on the protocol-dependent interface.

After creating a *DtEntityPublisher*, you can find its ID using the member function `id()`.

Setting an Entity's State

An entity's state can be set through a *DtEntityStateRepository* – a container for an entity's state. It is defined in [entitySR.h](#).

In each iteration of your simulation, you should update the *DtEntityStateRepository* of each entity you are simulating locally so that it contains the current state of the entity before calling *DtEntityPublisher::tick()*. You can obtain a pointer to a *DtEntityPublisher's* *DtEntityStateRepository* using the member function *DtEntityPublisher::entity-StateRep()*.

DtEntityStateRepository has mutator functions that allow you to set the various components of an entity's state. Time-space-position information can be set with:

- ♦ *setLocation()*
- ♦ *setVelocity()*
- ♦ *setAcceleration()*
- ♦ *setOrientation()*
- ♦ *setRotationalVelocity()*

Components that affect the outward appearance of an entity can be set with functions such as:

- ♦ *setDamageState()*
- ♦ *setFlamesPresent()*
- ♦ *setEngineSmokeOn()*
- ♦ *setHatchState()*

See [entitySR.h](#) for a complete list.

Corresponding inspector functions also exist within *DtEntityStateRepository*, but these are more typically used to inspect a remote entity's state, rather than that of a locally simulated entity which you have set yourself.

Positions, velocities, and accelerations must be set in geocentric coordinates, as specified in the DIS Standard. In this coordinate system, the origin represents the center of the earth, the X-Y plane goes through the equator, with the X axis passing through the prime meridian, the Y axis passing through 90 degree east longitude, and the Z axis pointing through the north pole.

Orientation is specified as a *DtTaitBryan*, a class that represents three angles – successive rotations needed to transform from the geocentric coordinate system to the entity coordinate system (origin at the center of mass, X points out the front, Y points out the right side, Z points out the bottom.) The specific sequence of rotations is known as the Tait-Bryan sequence, in which you first rotate about Z, then about the new Y, then about the newest X axis. The angles are specified in radians.

See “[Coordinate Conversions](#)” on page 7-7 for information about converting rotation matrices to Euler angles, and converting among the many different coordinate systems supported by VR-Link.

```
DtEntityPublisher tankPub(...);
...
// Grab a pointer to our entity publisher's ESR
DtEntityStateRepository *esr = tankPub.entityStateRep();

// Set location to somewhere near Ft. Hunter Liggett, CA, in
// geocentric coordinates.
esr->setLocation(DtVector(-2696545.0, -4430407.0, 3701906.0));

// Velocity and acceleration are also in geocentric coordinates
esr->setVelocity(DtVector(100.0, 100.0, 100.0));
esr->setAcceleration(DtVector(0.0, 0.0, 0.0));

// Set orientation as three Euler angles in radians
esr->setOrientation(DtTaitBryan(-2.11, 0.948, 2.469);

// Set angular velocity vector in body coordinates
esr->setRotationalVelocity(DtVector(0.0, 0.10, -0.125);

// Indicate that the entity is on fire
esr->setFlamesPresent(DtTrue);

// Indicate that the entity is slightly damaged
// The DtDamageState enumeration is in disEnums.h
esr->setDamageState(DtDamageSlight);
...
...
```

`DtEntityStateRepository::printData()` can be used to print the current contents of a *DtEntityStateRepository* in human readable form.

The *DtEntityPublisher* tick() Function

`DtEntityPublisher::tick()` is the function that does most of the *DtEntityPublisher*'s work. It should be called for each *DtEntityPublisher* once per simulation frame, after the entity's state has been updated using the *DtEntityStateRepository*'s mutator functions.

The tick function decides what data needs to be sent to the network (if any) and formats and sends that data through the *DtEntityPublisher*'s exercise connection.

DtEntityPublisher internally maintains a second *DtEntityStateRepository* representing the way remote applications currently see the entity, based on prior entity state updates, and dead-reckoning calculations. Deciding what needs to be sent involves comparing the current state with this lower fidelity representation of state. If position or orientation differences exceed the *DtEntityPublisher*'s thresholds, or if any other attributes have changed at all, data needs to be sent so that remote applications will have access to the most current information.

DIS only

Current data is sent periodically whenever a time threshold has been exceeded since the last update was sent, even if nothing has changed. (This DIS rule is sometimes known as an entity **heartbeat**.)

If any data needs to be sent, an Entity State PDU is sent to the exercise containing current values for all fields.

HLA only

An attribute update is sent containing values for only those attributes that have changed.

Setting Thresholds

DtEntityPublisher's thresholds for position and orientation are configurable through the static member functions `setDfltTranslationThreshold()` and `setDfltRotationThreshold()` respectively. These thresholds apply to all instances of *DtEntityPublisher*. Translation threshold is specified in meters, and defaults to 1.0. Rotation threshold is specified in radians and defaults to the radian equivalent of 3.0 degrees.

DIS only

For DIS, there is a time threshold that defaults to 5.0 seconds. You can configure the time threshold for all *DtEntityPublishers* with the static member function `setDfltTimeThreshold()`, or override it on a per-entity basis using the member function `setTimeThreshold()`. Time is specified in seconds.

4.3.2. Managing Remote Entities

A *DtReflectedEntityList* (REL) maintains the current state of entities that VR-Link learns about through updates received from the exercise. For the most part, this consists of entities simulated by other applications. For DIS, an application receives entity state PDUs sent by itself along with those sent by other applications. Therefore, in DIS the *DtReflectedEntityList* contains a representation of locally simulated entities as well.

Each entity in the list is represented by an instance of *DtReflectedEntity*. The list provides member functions that let you look up a *DtReflectedEntity* by ID, or iterate through the list.

VR-Link provides both HLA and DIS versions of *DtReflectedEntityList* and *DtReflectedEntity*, which are defined in *hReflEntList.h*, *hReflectedEnt.h*, *dReflEntList.h*, and *dReflectedEnt.h*. Rather than include all of these header files, you can include *reflEntList.h* and *reflectedEnt.h*, which include the appropriate versions based on whether or not you have included the *DtHLA=1* definition in your compile line.

Although the two versions of these classes are defined separately, most of the public member function names and prototypes are shared across the two versions, so that you can make most of these calls regardless of which protocol you are using.

For *DtReflectedEntity*, some of this commonality is enforced by deriving both versions from a single base: *DtBaseReflectedEntity*, which is defined in *baseReflEnt.h*.

Creating Reflected Entity Lists

A *DtReflectedEntityList* is created on an exercise connection as follows:

```
DtExerciseConn exConn(...);  
...  
DtReflectedEntityList(&exConn);
```

Whenever a new entity is detected by the REL, it automatically creates a new *DtReflectedEntity* to represent it. Whenever a state update is received, the corresponding *DtReflectedEntity* is updated to reflect the current state. These two events will automatically occur as long as *DtExerciseConn::drainInput()* is called periodically within your application. No further action is required before inspecting data on remote entities.

HLA only

For HLA, you can never be sure when inspecting data that values for all attributes have been received.

DtReflectedEntityList has *first()* and *last()* member functions which return the first and last entities in the list. *DtReflectedEntity* has *next()* and *prev()* member functions, which return the next and previous entity in the list. These two functions return NULL if you try to read past the end or beginning of the list. You can iterate through all entities in the list as follows:

```
DtReflectedEntityList rel(...);  
...  
for (DtReflectedEntity* ent = rel.first();  
     ent;  
     ent = ent->next())  
{  
    ...  
}
```

DtReflectedEntity also has *wrapNext()* and *wrapPrev()* member functions that are similar to *next()* and *prev()*, but which loop back to the first or last entity in the list when you try to move past the end or beginning of the list.

You can obtain the total number of entities in the REL with the *count()* member function.

If you would like to look up a *DtReflectedEntity* by its ID, the REL provides the member function `lookup()`.

An ID is represented by a *DtObjectId*, defined in [hostStructs.h](#).

DtObjectId is used rather than *DtEntityIdentifier* or `RTI::ObjectID()` when we want to refer to an identifier without regard for the protocol.

For example:

```
DtReflectedEntityList rel(...);
...
// HLA IDs are one number, DIS IDs have three components.
#ifdef DtHLA
DtObjectId id(100002);
#else
DtObjectId id(1, 1, 1);
#endif
DtReflectedEntity* ent = rel.lookup(id);
```

Once you have a pointer to a *DtReflectedEntity*, you can obtain its ID using its `id()` member function (inherited from *DtBaseReflectedEntity*).

HLA only

In HLA, *DtObjectId* is a wrapper around the `RTI::ObjectID()` – a single number that the RTI uses to identify an object.

DIS only

In DIS, *DtObjectId* is typedefed to *DtEntityIdentifier* (also defined in [hostStructs.h](#)) – a class that represents the DIS identification triplet of site ID, application number and entity number.

Inspecting an Entity's State

DtReflectedEntity uses a *DtEntityStateRepository* to store the current state of the entity. This is the same class used by *DtEntityPublisher* to store the state of a locally simulated entity; see “[Setting an Entity's State](#)” on page 4-13 for more details. Individual state data are inspected through *DtEntityStateRepository* inspector functions rather than directly through the *DtReflectedEntity*.

A pointer to a *DtReflectedEntity*'s ESR can be obtained using `DtReflectedEntity::entity-StateRep()`.

DtEntityStateRepository has inspector functions that allow you to look at various components of an entity's state. Time-space-position information can be obtained using:

- ♦ `location()`
- ♦ `velocity()`
- ♦ `acceleration()`
- ♦ `orientation()`
- ♦ `bodyToGeoc()`
- ♦ `rotationalVelocity()`

Components that affect the outward appearance of an entity can be examined with functions such as:

- ♦ `damageState()`
- ♦ `flamesPresent()`
- ♦ `engineSmokeOn()`
- ♦ `hatchState()`

See [entitySR.h](#) for the complete list.

Corresponding mutator functions also exist within *DtEntityStateRepository*, but these are more typically used to set a locally simulated entity's state, rather than that of a reflected entity, which is set automatically from data in state updates received from the exercise.

Positions, velocities, and accelerations are in geocentric coordinates, as specified in the DIS Standard. Orientation is available in one of the following forms:

- ♦ The `orientation()` member function returns a *DtTaitBryan* (three euler angles in radians, representing ordered rotations needed to transform from the geocentric coordinate system to the entity coordinate system).
- ♦ The `bodyToGeoc()` member function returns a rotation matrix that will rotate from the body to geocentric coordinate system.

See “[Coordinate Conversions](#)” on page 7-7 for information about how to convert rotation matrices to Euler angles, and how to convert among the many different coordinate systems supported by VR-Link. In addition, the section on *DtEntityStateRepository* under “[Managing Locally Simulated Entities](#)” on page 4-11 provides more information about the coordinate systems mentioned above.

VR-Link also provides various View classes, described in “[Coordinate Views](#)” on page 4-25, which provide the ability to inspect data in a *DtEntityStateRepository* in other coordinate systems without explicitly performing coordinate conversions.

Some of the inspector functions take enumerations as arguments. Values for these enumerations are in [disEnums.h](#). When a *DtBoolean* is returned, it will be either *DtTrue* or *DtFalse*.

The following example shows how to use *DtEntityStateRepository*'s inspector functions to examine the current state of a reflected entity, within a function that prints part of the state of the first entity in the list:

```
void printStateOfFirstEnt(DtReflectedEntityList* rel)
{
    // Grab a pointer to the first entity
    DtReflectedEntity* firstEnt = rel.first();

    // Make sure the list isn't empty
    if (!firstEnt)
    {
        return;
    }

    // Print the entity's ID
    printf("ID: %s\n", firstEnt->entityId().string());

    // Grab a pointer to the entity's ESR
    DtEntityStateRepository *esr = firstEnt->entityStateRep();

    // DtVector and DtTaitBryan's string member functions return
    // a string representation of their data
    printf("Loc: %s\n", esr->location().string());
    printf("Vel: %s\n", esr->velocity().string());
    printf("Accel: %s\n", esr->acceleration().string());
    printf("Orient: %s\n", esr->orientation().string());
    printf("AngVel:%s\n", esr->rotationalVelocity().string());

    if(esr->flamesPresent == DtTrue)
    {
        printf("Flaming!\n");
    }

    // The DtDamageState enumeration is in disEnums.h
    if (esr->damageState() == DtDamageDestroyed)
    {
        printf("Destroyed!\n");
    }
}
```

DtEntityStateRepository::printData() works in a similar way, and can be used to print the current contents of a *DtEntityStateRepository* in human readable form.

Dead Reckoning

By default, the position, velocity and orientation returned by a *DtReflectedEntity*'s *DtEntityStateRepository*'s member functions (`location()`, `velocity()`, `orientation()`, and `bodyToGeoc()`), are dead-reckoned values. That is, they are not merely the values last received via state updates from the exercise. They are extrapolated forward to the current value of VR-Link simulation time from acceleration, velocity, and angular velocity based on the entity's current dead-reckoning algorithm.

The current value of VR-Link simulation time is the last value passed to *DtSetSimTime*, which should be called once per frame by an application. In this way, all entities are dead-reckoned to the same time, regardless of the order in which the locations are inspected. Time of validity of the inputs to the dead-reckoning equation is the value of VR-Link simulation time when the data was received from the exercise.

Notes:

- ♦ There is no performance penalty for calling the inspectors for dead-reckoned values more than once during a particular frame. As long as the value of VR-Link simulation time has not changed, the dead-reckoning code is not executed again; a cached value is used. Dead reckoning is not performed for entities whose positions and orientations are never inspected.
- ♦ Dead reckoning of orientation as a set of Euler angles is a more expensive operation than dead reckoning orientation as a rotation matrix. So use `bodyToGeoc()` rather than `orientation` in cases where performance is an issue, and you do not have an explicit need for the Euler angle representation.

A *DtEntityStateRepository* uses a *DtDeadReckoner* (defined in [deadReckoner.h](#)), which is derived from *DtApproximator* (defined in [approximator.h](#)), to perform its dead reckoning.

In order to turn dead reckoning off, so that `location()`, `velocity()`, `orientation()`, and `bodyToGeoc()` return the last values received from the exercise rather than current dead-reckoned values, set the *DtEntityStateRepository*'s approximator to NULL, using its `setApproximator()` member function. In fact, this is what a *DtEntityPublisher* does with its ESR, so that you do not get dead-reckoned values when inspecting the ESR of a locally simulated entity.

To restore dead-reckoning capability to the ESR, save a pointer to its current approximator (obtained by using the approximator member function) before setting it to NULL, then restore it by passing the approximator to a second call to `setApproximator`. For example:

```
DtEntityStateRepository* esr = ... ;
// Save a pointer to the original approximator
DtApproximator* origApprox = esr->approximator();
// Turn off dead-reckoning to set approximator to NULL
esr->setApproximator(NULL);
// location returns non-dead-reckoned value
DtVector nonDR = esr->location();
// Restore the approximator
esr->setApproximator(origApprox);
// Now location returns dead-reckoned value again
DtVector deadReckoned = esr->location();
```

Entities Joining and Leaving the Exercise

Often, an application will want to be notified when an entity joins or leaves the exercise. Immediately after a new entity is added to a *DtReflectedEntityList*, the virtual function `entityAdded()` is called. Similarly, when an entity is removed from an REL, the virtual function `removeAndDelete()` is called.

By subclassing *DtReflectedEntityList*, and overriding these two functions, an application can provide code that will be executed when these events occur. Make sure to use your derived *DtReflectedEntityList* in place of the default *DtReflectedEntityList* when creating your REL. Also, be sure to call down to `DtReflectedEntityList::removeAndDelete()` from within your version of the function so that the entity actually gets removed from the list.

In the following example, our application will print *HELLO* or *GOODBYE* when an entity comes or goes, along with the ID of the entity:

```
class NotifyingREL : public DtReflectedEntityList
{
public:
    // constructor
    NotifyingREL(DtExerciseConn* exConn) :
        DtReflectedEntityList(exConn) {}

    // Override entityAdded
    virtual void entityAdded(DtReflectedEntity* ent)
    {
        printf("HELLO %s\n", ent->id().string());
    }
}
```

```
// Override removeAndDelete
virtual void removeAndDelete(DtReflectedEntity* ent,
                             char* msg = NULL)
{
    printf("GODDBYE %s\n", ent->id().string());

    // Remember to call the base class version
    DtReflectedEntityList::removeAndDelete(ent, msg);
}
};

main()
{
    DtExerciseConn exConn(...);
    ...
    // Create a Notifying REL, rather than the base REL
    NotifyingREL rel(&exConn);
    ...
}
```

HLA only

An entity is added to the REL as soon as VR-Link receives a `discoverObject()` service call from the RTI. This occurs before the first attribute update is presented to us by the RTI. For this reason, the *DtReflectedEntity*'s ESR will not contain any data at the time when the entity is passed to your overridden version of `entityAdded()`. Only the entity's ID will have been set at this point. Within `entityAdded()` you should be doing only things like saving a pointer to the entity, rather than trying to inspect any data.

Typically, in HLA, the first update immediately follows the `discoverObject` invocation for a new entity. Therefore, by the time `drainInput` returns, most data will be valid for your application to inspect. If you do want to be notified immediately after the first update arrives, see the next section, [“Intercepting State Updates” on page 4-22](#).

Another caveat is that `entityAdded()` and `removeAndDelete()` are called from within RTI callbacks. RTI rules prohibit making RTI calls from within an RTI callback, so do not make any RTI calls (or call any functions that make RTI calls) from within `entityAdded()` or `removeAndDelete()`.

DIS only

`entityAdded()` is called after the first entity state PDU for the entity has been processed. Therefore, the state of the entity is available within `entityAdded()`, through the entity's ESR. However in the interests of writing protocol-independent code, you might not want to inspect the data here in DIS either.

Intercepting State Updates

Some applications might want to have access to state updates for particular entities. Since state updates are different in DIS and HLA, this must be done with protocol-specific code, and is described in the protocol-specific chapters.

Timeouts

DtReflectedEntityList has the capability to time entities out, removing them from the list if an update has not been received within some period of time.

DIS only

This capability is on by default in DIS, where the rules state that an entity state PDU must be sent periodically (usually once every 5 seconds), even if no data has changed.

HLA only

In HLA, there is no such rule; it is perfectly valid for an entity to go several minutes or more without updating attributes, as long as nothing has changed. For this reason, timeouts are off by default in HLA. When timeout processing is off, a *DtReflectedEntity* is not removed from the REL until we are notified by the RTI that the entity has left the exercise.

For both DIS and HLA, timeouts can be turned on and off using the static member function *DtReflectedEntityList::setTimeoutProcessing()*. The argument is a *DtBoolean*, either *DtTrue* or *DtFalse*.

If timeout processing is on, the REL checks to see if any entities need to be timed out each time *DtExerciseConn::drainInput()* is called. The timeout interval (the amount of time that can elapse since the last update before an entity is timed out) defaults to 12.0 seconds, but can be configured with the static member function *DtReflectedEntityList::setTimeoutInterval()*.

Subclassing *DtReflectedEntity*

Some applications might want to associate additional data or functionality with a *DtReflectedEntity*. One way to achieve this is by subclassing *DtReflectedEntity*. For example, in the Stealth, we use a subclass of *DtReflectedEntity* to associate graphics data with the entity.

If you do choose to subclass *DtReflectedEntity* (rather than associating data through composition, for example), it is necessary to subclass *DtReflectedEntityList* as well. The reason for this is simple: *DtReflectedEntities* are created by the REL, and the base REL only knows how to create base *DtReflectedEntities*. You need to create a derived REL that knows how to create your derived *DtReflectedEntities*.

An REL uses the virtual function *newReflectedEntity()* to create *DtReflectedEntities*. It is this function that you must override with a definition that returns a new instance of your derived *DtReflectedEntity*.

The constructors for DIS and HLA versions of *DtReflectedEntity* take different arguments; therefore the two versions of *DtReflectedEntityList::newReflectedEntity()*, (which basically just passes its arguments to the *DtReflectedEntity* constructor) take different arguments as well.

The following example shows how to subclass the two classes:

```
class myReflectedEntity : public DtReflectedEntity
{
public:
    // Constructor
#ifdef DtHLA
    myReflectedEntity(DtHlaObject* obj, DtExerciseConn* conn) :
        DtReflectedEntity(obj, conn)
#else
    myReflectedEntity(DtExerciseConn* conn,
                      const DtEntityIdentifier& id,
                      const DtEntityType& type) :
        DtReflectedEntity(conn, id, type)
#endif
    {
        // The two versions may be able to share a body
        ...
    }
    // Specifics of myReflectedEntity
    ...
};

class myREL : public DtReflectedEntityList
{
public:
    // Constructor (same for both DIS and HLA)
    myREL(DtExerciseConn* exConn) :
        DtReflectedEntityList(exConn) {}
#ifdef DtHLA
    virtual DtReflectedEntity* newReflectedEntity(
        DtHlaObject* obj) const
    {
        return new myReflectedEntity(obj, exerciseConn());
    }
#else
    virtual DtReflectedEntity* newReflectedEntity(
        const DtEntityIdentifier& id,
        const DtEntityType& type) const
    {
        return new myReflectedEntity(exerciseConn(), id, type);
    }
#endif
};
```


4.4. Coordinate Views

As described in the previous sections, *DtEntityStateRepository*'s mutators and inspectors expect and return position and orientation and their derivatives in geocentric coordinates. If you would like to use a different coordinate system in your application, you can use VR-Link's coordinate conversion routines to convert outgoing data to a geocentric form before setting values in a *DtEntityPublisher*'s ESR, and to convert incoming data obtained from a *DtReflectedEntity*'s ESR from geocentric to your coordinate system of choice. However, there is an easier way.

VR-Link provides several View classes which can be created on an ESR, providing access to the data in a coordinate system other than geocentric. After creating the appropriate view, simply use its mutators and inspectors, rather than directly calling the ESR's functions. The View's functions call down to the ESR's functions before or after doing the conversions, so that the ESR always stores geocentric data.

There are three View classes:

- ♦ *DtUtmView* supports UTM coordinates
- ♦ *DtCartesianView* supports any arbitrary Cartesian coordinate system
- ♦ *DtTopoView* is derived from *DtCartesianView* and supports topographic coordinate systems (Cartesian systems that have their X-Y plane tangent to the Earth's surface at their origin, with the X axis pointing north, the Y axis pointing east, and the Z axis pointing down).

These classes are defined in [utmView.h](#), [cartView.h](#), and [topoView.h](#). See “Coordinate Conversions” on page 7-7 for more detail on these coordinate systems.

The ESR inspector and mutator functions that can be circumvented using Views are:

- ♦ `location()`
- ♦ `setLocation()`
- ♦ `velocity()`
- ♦ `setVelocity()`
- ♦ `acceleration()`
- ♦ `setAcceleration()`
- ♦ `orientation()`
- ♦ `setOrientation()`
- ♦ `bodyToGeoc()`
- ♦ `rotationalVelocity()`
- ♦ `setRotationalVelocity()`

However, since `rotationalVelocity()` is always specified in body rather than geocentric coordinates, the Views' `rotationalVelocity()` and `setRotationalVelocity()` functions just pass the ESR values through without any conversion.

4.4.1. Topographic Coordinate Views

The *DtTopoView* constructor takes a pointer to the ESR that we will be accessing, and a latitude and longitude (in radians) that define the particular topographic coordinate system we'd like to work in:

```
DtEntityPublisher entPub(...);
DtEntityStateRepository* esr = entPub.entityStateRep();
DtTopoView topoView(esr, DtDeg2Rad(36.0), DtDeg2Rad(-121.0));
```

We can now use *DtTopoView*'s mutators to set positions and orientations specified with respect to a topographic coordinate system whose origin is at a lat/long of {36.0, -121.0}.

Note: Topographic Euler angles correspond to the familiar heading, pitch and roll.

```
// Topographic coordinates
topoView.setLocation(DtVector(100.0, 100.0, 0.0));
topoView.setVelocity(DtVector(10.0, 10.0, 0.0));
topoView.setAcceleration(DtVector(1.0, 1.0, 0.0));

// Topographic Euler angles - heading, pitch and roll
topoView.setOrientation(DtTaitBryan(0.0, DtDeg2Rad(10.0), 0.0));

// Rotational velocity is always in body coordinates
topoView.setRotationalVelocity(DtVector(0.0, 0.10, -0.125));
```

Going the other way, we see how a *DtTopoView* is used to inspect a remote entity's data in topographic coordinates:

```
DtReflectedEntityList rel(...);
...
DtReflectedEntity* ent = rel.first();
DtEntityStateRepository* esr = ent->entityStateRep();
DtTopoView topoView(esr, DtDeg2Rad(36.0), DtDeg2Rad(-121.0));
```

We can now use *DtTopoView*'s inspectors to get positions and orientations specified with respect to the topographic coordinate system.

```
// Topographic coordinates
DtVector topoLoc = topoView.location();
DtVector topoVel = topoView.velocity();
DtVector topoAccel = topoView.acceleration();

// Topographic Euler angles - heading, pitch and roll
DtTaitBryan topoOrient = topoView.orientation();
// Or choose a matrix representation of orientation
DtDcm bodyToLocal = topoView.bodyToLocal();

// Rotational velocity is always in body coordinates
DtVector angVel = topoView.rotationalVelocity();
```

4.4.2. UTM Coordinate Views

DtUtmView is a View that allows you to work in UTM coordinates. Its constructor takes only an ESR, but relies on the fact that you've already initialized VR-Link's UTM coordinate system using *DtUtmInit*. (See [“Coordinate Conversions”](#) on page 7-7). Once you've chosen a particular UTM coordinate system in this way, you can use the inspectors and mutators like you do with *DtTopoView*.

Locations and their derivatives are in UTM coordinates, and are expressed as *DtUtmCoords*, defined in [utmCoord.h](#). UTM orientations are treated in the same way as topographic orientations. The three Euler angles represent heading, pitch and roll, while *bodyToLocal()* returns a rotation matrix that can rotate a vector from the body frame to a topographic frame centered at the entity's current location. As always, rotational velocity is represented in body coordinates.

4.4.3. Cartesian Coordinate Views

DtCartesianView allows the creation of a view defined by any Cartesian coordinate system. Along with an ESR, its constructor takes a *DtVector* indicating the origin of the new system, expressed in geocentric coordinates, and a *DtDcm* representing a rotation matrix that can rotate a vector from the geocentric frame to the frame we are defining.

Alternatively, you can use the constructor that takes only an ESR, and initialize the View by passing the two parameters to the *setOffset()* and *setRotation()* member functions.

Since *DtTopoView*, inherits its inspectors and mutators from *DtCartesianView*, the descriptions of these functions in [“Topographic Coordinate Views”](#) on page 4-26 applies here, except that Euler angles do not correspond to heading, pitch and roll, when expressed with respect to an arbitrary Cartesian system.

4.5. Articulated Parts

DIS (and the RPR FOM in HLA) define articulated parts as movable parts of an entity, such as a turret, gun, or landing gear. The state of an entity's articulated parts is part of an entity's state, and is accessible through a *DtEntityStateRepository*.

A *DtEntityStateRepository* uses a *DtArtPartList*, defined in [artPartList.h](#), to store the articulated parts portion of its state. You can obtain a pointer to an ESR's *DtArtPartList* with its `artPartList()` member function.

4.5.1. Inspecting Articulated Parts Data

DtArtPartList provides many inspector functions for examining the state of an entity's articulated parts. The `numArtParts()` member function returns the number of articulated parts the entity has. For example, if a tank had a turret and a gun, `numArtParts()` returns 2. The `numArtParams()` member function returns the number of articulation parameters the entity is using to convey articulated parts information to the exercise. For example, if the tank is publishing information on its turret's azimuth and azimuth rate, as well as its gun's elevation, `numArtParams()` returns 3.

`DtArtPartList::getArtParamVal()` allows you to query the list for the value of a particular articulation parameter. The part type should be a member of the *DtArtPartType* enumeration, while the parameter type should be a member of the *DtArtParamType* enumeration, both defined in [disEnums.h](#). The `val` argument is the variable you would like filled with the value being inspected, passed by reference. The function returns `DtTrue` if successful, and `DtFalse` if the part-type/param-type combination is not valid for the entity.

The value returned is dead-reckoned to the current value of VR-Link simulation time, if a rate for the parameter in question has been included in the parameter list for the entity.

For example, here's how to inspect the value of the turret azimuth of a particular entity:

```
DtReflectedEntityList rel(...);
....
DtReflectedEntity* firstEnt = rel.first();
DtEntityStateRepository* esr = firstEnt->entityStateRep();
DtArtPartList* artParts = esr->artPartsList();
float turAz = 0.0;
if (artParts->getArtParamVal(DtPrimaryTurret1, DtApAzimuth,
    turAz) != DtTrue)
{
    printf("Entity has no turret azimuth);
}
```

To obtain more detailed information about articulated parts, including obtaining non-dead-reckoned values, you must look at the individual *DtArtPart* objects that a *DtArtPartList* uses to represent individual parts. *DtArtPart* is defined in [artPart.h](#).

The *DtArtPart* Class

You can obtain a pointer to a particular *DtArtPart* by part type or by index (in the range of 1 to numArtParts), through the *DtArtPartList* functions artPartFromPartType() or artPartFromPartNum() respectively. Both return NULL if there is no part identified by the given part type or part number.

You can iterate through the list of *DtArtParts* as follows:

```
DtArtPartList *parts = ...;
for (int i = 1; DtArtPart* part =
    parts->artPartFromPartNum(i); i++)
```

A *DtArtPart*'s part type (a member of the *DtArtPartType* enumeration) and part number (its index in a part list) can be obtained with partType() and partNum(). partAttachedTo() returns the type of the part to which the current part is attached. For example, a gun is often attached to a turret, whereas a turret is usually attached to the base entity (indicated by a return value of DtAttachedToBase).

The number of parameters used for this part is found using numParams(), while firstValidParamType() and nextValidParamType() are used to iterate through the set of parameters as follows:

```
DtArtPart* part = ...;
for (int param = part->firstValidParamType;
    param;
    param = part->nextValidParamType(param));
```

DtArtPart::getVal() lets you inspect the value of a parameter by parameter type, filling in a variable passed by reference, and returning DtFalse if the parameter is not valid for the part. DtArtPart::val() doesn't do the validity checking, so should be called only when you know the parameter is valid.

Disabling Dead Reckoning

By default, values returned by these functions are dead reckoned to the current time if the parameter's rate is also a valid parameter for the part. However, dead reckoning can be disabled using the setApproximating() member function, passing DtFalse as an argument.

If, rather than inspect individual parameters, you would like to obtain the state of the part as a whole, expressed as a vector of X, Y, and Z translations, or as a set of three angles representing azimuth, elevation and rotation, then use the functions translations() and angles().

Both *DtArtPart* and *DtArtPartList* have print() functions that print their data in human readable form.

4.5.2. Setting Articulated Parts Data

On the outgoing side, you can use *DtEntityStateRepository*'s `artPartList()` function to access a pointer to a *DtEntityPublisher*'s ESR's articulated part list, then use *DtArtPartList* and *DtArtPart* mutator functions to set current data for an entity you are simulating.

Note: A *DtArtPartList* is initialized with the function `init()`, which takes an array of *DtArtPartSpecs*, or articulated part specifications. `init()` must be called to indicate what parts and parameters are valid for the entity, before you try to set values for any of the parameters.

A *DtArtPartList* has the following definition, which is in [apSpec.h](#):

```
typedef struct DtArtPartSpec
{
    int partType;
    int attachedType;
    int params[DtMAX_PARAMS_PER_PART];
} DtArtPartSpec;
```

The array should have one *DtArtPartSpec* for each of the entity's articulated parts for which you will be communicating information to the exercise, plus a terminating *DtArtSpec* that has a part type of zero.

`partType()` should be a member of the *DtArtPartType* enumeration found in [disEnums.h](#), and should specify the type of the part. **attachedType** is the part type of the part to which the current part is attached. If the part is attached to the base entity, rather than to another part, the attached type should be *DtAttachedToBase*. **params** is a zero-terminated array indicating which parameters will be used to communicate information on this part to the exercise. Elements of the array should be members of the *DtArtParamType* enumeration.

The following example shows how to statically create an array of *DtArtPartSpecs* for an entity with a turret and a gun, and use it to initialize a *DtArtPartList*. We will use two parameters (azimuth and azimuth rate) for the turret, and one parameter (elevation) for the gun. Note the use of the *DtNullArtPartSpec* macro to create the terminating NULL *DtArtPartSpec*.

```
DtArtPartSpec* specList =
{
    {
        DtPrimaryTurret1,
        DtAttachedToBase,
        {DtApAzimuth, DtApAzimuthRate, 0}
    },
    {
        DtPrimaryGun1,
        DtPrimaryTurret1,
        {DtApElevation, 0}
    },
    {
        DtNullArtPartSpec
    }
};

DtEntityPublisher entityPub(...);
DtEntityStateRepository* esr = entityPub.entityStateRep();
DtArtPartList* artParts = esr->artPartList();
artParts->init(specList);
```

An equivalent way to initialize an ESR's articulated parts is to pass the array of *DtArtPartSpecs* to *DtEntityStateRepository::initArtParts()*.

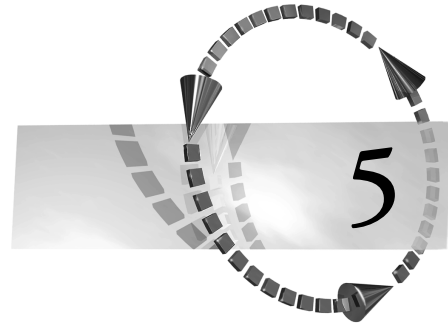
Once the *DtArtPartList* is initialized you can use its *setArtParamVal()* member function to set values for the parameters. This function works like its complementary inspector *artParamVal()*:

```
artParts->setArtParamVal(DtPrimaryTurret1, DtApAzimuth,
                        DtDeg2Rad(90.0));
```

Alternatively, you can obtain a pointer to an individual *DtArtPart*, and use the *setVal()* function:

```
DtArtPart* turret = artParts->artPartFromPartType(DtPrimaryTurret1);
if (turret)
{
    turret->setVal(DtApAzimuth, DtDeg2Rad(90.0));
}
```

If you would like to change the set of parts or parameters used for the entity, call *init()* again with a new array of *DtArtPartSpecs*.



DIS-Specific Interface

VR-Link's protocol-independent interface provides the functionality most simulations need to interact with a DIS or HLA exercise. However, some applications need greater control over protocol-specific parameters, or must access functionality that exists only in one domain or the other. VR-Link's protocol-specific layer provides this flexibility. This chapter discusses the *DtPdu* class and its subclasses, which represent the various DIS PDUs.

This chapter covers the following topics:

- ♦ [Sending PDUs](#), on page 5-2
- ♦ [Inspecting and Setting Data in A PDU Header](#), on page 5-7
- ♦ [Getting a PDU's Network Representation](#), on page 5-8
- ♦ [Configuring Your Connection to the DIS Network](#), on page 5-21
- ♦ [DtSocket Specifics](#), on page 5-26
- ♦ [Intercepting Incoming Entity State PDUs](#), on page 5-29

Note: If your application is intended for both DIS and HLA, or if you plan to add this support in the future, enclose your DIS-specific code within a preprocessor directive pair to ensure that it will not be compiled when you try to build for HLA:

```
#if !DtHLA
// DIS-specific code
#endif
```

5.1. Working with PDUs

VR-Link's *DtPdu* class, defined in [pdu.h](#), is used to represent DIS PDUs. *DtPdu* is an abstract base class. Many classes that represent the individual types of PDUs are derived from *DtPdu*. The derived classes provide mutator and inspector functions to set and examine the values of PDU fields. Header information is accessed through *DtPdu* member functions.

All PDU classes are derived from *DtPdu*, including those that represent state updates, and those that represent events. Of this latter category of PDUs, those PDUs that have HLA counterparts have been derived from *DtInteraction*, an essentially empty class, which is in turn derived from *DtPdu*. In addition, we have provided typedefs so that you can refer to a *DtFirePdu* as a *DtFireInteraction* in order to use it in protocol-independent code.

All DIS 2.0.3, 2.0.4, IEEE1278.1 and 2.1.4 PDUs have PDU classes implemented in VR-Link. Since there are nearly 70 PDUs, we do not describe all of their inspectors and mutators here in the narrative section of this manual. In general though, an inspector function has the same names as the field being inspected, and returns the value it is inspecting. For example *DtEntityStatePdu::entityType()*. Mutator functions have the same names, preceded by the word “set”, for example, *DtEntityStatePdu::setEntityType()*.

You can find descriptive information about the DIS 2.0.3, 2.0.4 and IEEE 1278.1 PDU classes in Appendix A, *PDU Class Reference*. Please see the header files for the exact prototypes of all PDU class member functions, and descriptions of DIS 2.1.4 classes. Mappings between class names and header file names appear in [Table 9-2](#) and [Table 9.3](#).

5.1.1. Sending PDUs

To send a PDU:

1. Create an instance of a PDU class.
2. Use its mutator functions to set values for the different fields.
3. Send it using *DtExerciseConn*'s *send()* or *sendStamped()*.

sendStamped() sends a copy of the PDU that has the exercise ID and timestamp fields in the PDU header set to the *DtExerciseConn*'s exercise ID, and the current time, respectively; *send()* leaves the PDU header unmodified.

The following example shows how to create and send a Transmitter PDU:

```
DtExerciseConn exConn(...);
...
// Create a DtTransmitterPdu
DtTransmitterPdu pdu;
// Fill the PDU with data
pdu.setEntityId(DtEntityIdentifier(1,2,3));
pdu.setRadioId(4);
```

```
pdu.setTransmitState(DtOn);  
...  
// Send to the exercise  
exConn.sendStamped(pdu);
```

Normally, `send()` and `sendStamped()` send the PDU to the *DtExerciseConn*'s default destination IP address (see “[Creating an Exercise Connection for DIS](#)” on page 4-3 for details). However, the functions have an optional argument that specifies a different destination address. For example, to send a PDU to the machine with address 207.86.232.1, use:

```
exConn.sendStamped(pdu, DtStringToInetAddr("207.86.232.1"));
```

Note: Entity state PDUs are typically not sent individually by application code. Rather, an application usually uses *DtEntityPublisher* to manage the sending of entity state PDUs.

5.1.2. Receiving PDUs

Incoming PDUs are handled through callbacks – functions that application developers write, and register with VR-Link's *DtPdu* classes.

Each *DtPdu* subclass has static `addCallback()` and `removeCallback()` member functions that allow you to register and unregister functions that you wish to have called automatically each time that type of PDU is received by a particular *DtExerciseConn* during a call to `DtExerciseConn::drainInput()`. Within your callbacks, you can use a *DtPdu* subclass's inspector functions to examine the fields of the PDU.

PDU callbacks should be C-style functions or static member functions of a class, but cannot be non-static member functions. There is a different callback function type for each PDU class. For example, only callback functions that take a *DtTransmitterPdu* pointer as an argument may be registered with *DtTransmitterPdu*. A *DtTransmitterPdu* callback might look like this:

```
void myTransCallback(DtTransmitterPdu* pdu, void* usr)  
{  
    printf("Got a Transmitter PDU from %s\n",  
           pdu->entityId().string());  
}
```

It would be registered like this:

```
DtTransmitterPdu::addCallback(exerciseConn, myTransCallback,  
                              NULL);
```

Note: Since `addCallback()` is a static member of *DtTransmitterPdu*, it can be called using the qualified name (as above), rather than calling it on a particular instance of *DtTransmitterPdu*.

When the callback function is executed by VR-Link, the PDU that triggered the callback is passed to the function as the first argument.

The value passed to your callback function as `usr` is the value that was passed as the `usr` argument to `addCallback()`. Often, this `usr` argument is used to pass a pointer to an object on which you would like to call a member function from within your callback. Simply cast `usr` back to the object's type within the callback. For examples, see [“Receiving Interactions”](#) on page 4-8.

In cases where you would like to pass more than one object to your callback, you must create a simple structure consisting of pointers to your objects, then pass a pointer to the structure as `usr`.

Note: Any pointer you pass as `usr` must be the address of a variable whose lifetime is at least as long as the period for which the callback function is registered.

When a PDU is received for which multiple callbacks are registered, the callbacks are called in the opposite order from that in which they were registered.

DtPdus are transient, that is, they are deleted by VR-Link immediately after the last callback function registered for the PDU is called. Therefore, you should not attempt to delete or save a pointer to the *DtPdu* passed to your callback function. If you need to save the data, make a copy of the PDU.

An alternate way to control PDU callbacks is to use *DtExerciseConn*'s `addPduCallback()` and `removePduCallback()` member functions, which take a PDU kind along with your callback function and user data. In fact the PDU class' `addCallback()` and `removeCallback()` functions use these functions. In general, this alternate method is discouraged, since you do not get the type-safety that you do with the PDU class's functions. (You need to cast a *DtPdu** to a pointer to a specific derived class from within your callback). However, the *DtExerciseConn* functions are useful when implementing your own PDU classes.

More Control Over Receiving PDUs

Most applications use VR-Link's callback mechanism to handle incoming PDUs, leaving the job of actually reading the PDUs and dispatching them to the callbacks to *DtExerciseConn*'s `drainInput()`. However, *DtExerciseConn* does provide access to the functions used by `drainInput()` in order to give you more flexibility and control over reading PDUs from the network.

`drainInput()` works by repeatedly calling `readAndProcess()`. `readAndProcess()` tries to read a single PDU from the network using `netRead()`, then passes the PDU to `processPdu()`, from which all callbacks are invoked. Any of these functions can be used by application code, instead of calling `drainInput()`.

`netRead()` tries to read a packet from the network, constructs a PDU class from the network representation, and returns it.

If you use `netRead()` directly, note the following considerations:

- For efficiency reasons, successive calls to `netRead()` return *DtPdus* that all use the same buffer for their network representations. Applications must delete the *DtPdu* returned by `netRead()` before another call to `netRead()` is made. Otherwise, the previous *DtPdus* are trashed. Using `readAndProcess()` (or `drainInput()`) insures that each *DtPdu* is deleted before the next call to `netRead()`, which is why it is safer to use.
- Because PDU filtering based on PDU kind is done at the *DtSocket* level (before the packets get to `netRead()`), `netRead()` returns only those kinds of PDUs for which interest has been registered, if filtering is enabled. If you are using `netRead()`, you should either turn filtering off with `DtExerciseConn::disableFiltering()`, or explicitly register interest in the kinds of PDUs you would like to receive, using `DtExerciseConn::addInterestInPduKind()`.

The optional return code argument to `netRead()` is a pointer to a return code which is set depending on the results of the `netRead()` operation as follows:

- If a PDU was successfully read, **retCode** is set to `DtNET_READ_SUCCESS`. If there are no PDUs to be read, **retCode** is set to `DtNET_READ_NO_PACKETS`, and `NULL` is returned.
- If the *DtExerciseConn* is connected to a Packet Server through a *DtClientSocket*, and a packet is received that represents a control message from the Packet Server, rather than a DIS PDU, **retCode** is set to `DtNET_READ_SERVER_MSG`.
- If data is read, but it is not a PDU of the correct DIS version, `DtNET_READ_BAD_PACKET` is indicated.
- If the PDU length field conflicts with the actual size of the packet, the packet is rejected with status `DtNET_READ_SIZE_MISMATCH`.

So, an application can use `netRead()` as follows:

```
DtExerciseConn exConn(...);
...
int retCode;
while (1)
{
    DtPdu* pdu = exConn.netRead(&retCode);
    if (retCode == DtNET_READ_SUCCESS)
    {
        // do something with the PDU
        ...
        delete pdu;
    }
}
```

`readUntil()` is similar to `drainInput()` in that it repeatedly reads and processes PDUs, but it returns when `predicate()` returns true when called on the PDU being processed, or after **timeout** seconds, whichever happens first. If **timeout** seconds elapse, it returns *DtOtherPduKind*. Otherwise, it returns the kind of the PDU that caused the **predicate** to return true. `predicate()` is a function you provide that has the following signature:

```
int predicate(DtPdu *pdu, void *arg);
```

`readUntil()`'s signature is as follows:

```
DtPduKind readUntil(DtPredicate predicate, void* arg,  
                   DtTime timeout, DtTime sleepTime);
```

The **arg** argument passed to `readUntil()` is passed as the **arg** argument to `predicate()` each time it is called.

If no PDU is available to be read, then `readUntil()` will sleep for **sleepTime** seconds before polling the socket again. A timeout less than 0 causes `readUntil()` to poll continuously, without sleeping.

5.2. Inspecting and Setting Data in A PDU Header

The *DtPdu* class, provides several functions to inspect and set the data in a DIS PDU's header. Other fields of the various PDUs are accessed through member functions derived from *DtPdu*.

The following functions return the values of their respective fields in the PDU's header:

- ♦ `protocolVersion()`
- ♦ `exerciseId()`
- ♦ `kind()`
- ♦ `protocolFamily()`
- ♦ `length()`

Note: `protocolFamily()` is not available when linking with *libv13*, because it was not a field in DIS 2.0.3 headers. The *DtPduKind* and *DtProtocolFamily* enumerations are found in [disEnums.h](#).

Other functions for setting and returning data are described in [Table 5-1](#):

Table 5-1: DtPdu member functions

Function	Description
timeStamp()	returns a floating point number representing the number of seconds past the hour that is encoded in the PDU's time stamp. The time encoded in a PDU's timestamp indicates a number of seconds past a particular hour, but contains no information about what particular hour the sender had in mind. <code>guessTimeValid()</code> makes a guess at the full time the sender had in mind, by returning the time closest to <code>referenceTime</code> such that the number of seconds past the hour equals the value specified in the PDU's timestamp. Both <code>referenceTime</code> and the value returned represent seconds since Jan 1, 1970.
timeStampType()	returns either <code>DtRELATIVE</code> or <code>DtABSOLUTE</code> , depending on the type of the PDU's timestamp.
setExerciseId()	sets the value of the exercise ID field of the PDU's header to <code>id</code> .
setTimeStamp()	sets the PDU's timestamp to time modulo 3600, and the timestamp type to <code>relOrAbs</code> . Time is the time in seconds at which the PDU's data is to be considered valid. <code>relOrAbs</code> must be either <code>DtRELATIVE</code> or <code>DtABSOLUTE</code> . The default is <code>DtRELATIVE</code> .
print()	prints the entire contents of a PDU in human readable form, including header and data, by calling <code>printHeader()</code> followed by <code>printData()</code> . This is the function used by the <code>netdump</code> utility to print the contents of PDUs. <code>printHeader()</code> prints the information contained in the PDU's header, while <code>printData()</code> is a virtual function that prints the data below the header. The base class version of this function prints the data in hexadecimal format.

Other header information is set at the time the *DtPdu* is constructed, and cannot be set later. For details, see “[DtPdu Constructors](#)” on page 5-8.

5.2.1. Getting a PDU’s Network Representation

DtPdu::packet() returns a pointer to a buffer containing the network representation of the PDU – the exact bytes that would be sent onto the network if you were to send a *DtPdu* at any given time. This is a pointer to an internal buffer which, in most circumstances, you should not try to modify. Use member functions to set values for particular fields.

However, if you want to change the data that will go out to the network, you can modify this network representation. If you choose to do so, it is your responsibility to keep the data consistent and to ensure that the buffer represents a valid PDU.

Note: This will only work for those *DtPdu* classes that store their current data in their network representation buffer, (as opposed to storing it in some other representation and merely converting to network representation when *packet* is called). While this has been the chosen implementation for all PDU classes so far, we might choose other implementations in the future.

5.2.2. *DtPdu* Constructors

Since *DtPdu* is an abstract class, you cannot directly create a *DtPdu*. You can only create instances of classes derived from *DtPdu*.

All PDU classes have at least two constructors, a blank PDU constructor, and a from-network-representation constructor.

Constructing a Blank PDU

The blank PDU constructor constructs a blank, minimal PDU. By minimal, we mean that for variable length PDUs the size is equal to the smallest size that a PDU of that type can legally have. For example, a *DtDataPdu* created with the blank PDU constructor will have zero fixed datum fields and zero variable datum fields; a *DtEntity-StatePdu* so created will contain zero articulated parts. By blank, we mean that all bytes of the PDU below the header typically contain zeros (although in some cases other values more accurately represent the concept of a field being blank).

PDU Header Information

The protocol version is set to the value of the VR-Link global variable `DtProtocolVersionToSend`, declared in [pdu.h](#), as follows:

- ♦ The default is 5 (IEEE 1278.1) if you're building with `DtDISVERS=5`. Applications built with `DtDISVERS=5` accept both version 4 and version 5 PDUs. They transmit only version 5 PDUs. To send version 4 PDUs, set `DtProtocolVersionToSend` to 4.
- ♦ The default is 3 (DIS 2.0.3) if you're building with `DtDISVERS=3`.
- ♦ Set `DtProtocolVersionToSend` to 4 to indicate DIS version 2.0.4, or to 6 to indicate DIS 2.1.4.

PDU kind and protocol family depend on what type of PDU class you are creating (based on the value returned by the virtual function `internalGetPduKind()` which must be defined for each derived PDU class). The length is the PDU's minimal length, as defined above. The `exerciseId` and time stamp are zero.

There is an optional buffer argument to the blank PDU constructor. In most cases, you can ignore this argument and use the default, `DtUSE_INTERNAL_BUFFER`. Blank *DtPdu*'s are constructed as follows:

```
DtFirePdu pdu();
```

or simply

```
DtFirePdu pdu;
```

Alternate uses of the buffer argument are outlined in [“Using External Buffers in a PDU Class”](#) on page 5-11.

The blank PDU constructor is typically used when creating PDUs that you are sending. After creating the PDU, you can use its member functions to set the various data, then pass it to `DtExerciseConn::sendStamped()` for sending.

Constructing a PDU from a Network Representation

The from-network-representation constructor takes a pointer to a network representation of a PDU as an argument, and constructs a *DtPdu* object, which represents this PDU. By network representation, we mean a buffer containing the exact bytes that would be used to represent the PDU on the network.

It is rare that an application will need to use the from-network-representation constructor for a *DtPdu*. It is most commonly used only by a *DtPduFactory*, which is used by *DtExerciseConn* after receiving a packet from the network in order to construct the PDU class object that is passed to application callbacks.

Like the blank PDU constructor, the from-network-representation PDU class constructor has an optional buffer argument. This argument is used in the same way as the buffer argument to the blank PDU constructor, and is discussed in [“Using External Buffers in a PDU Class”](#) on page 5-11.

Each of the PDU class from-network-representation constructors requires as an argument a network representation of specifically that type of PDU. For instance, the *DtFirePdu* constructor specifically requires a *DtNetFirePdu*.

When constructing a *DtPdu* from a network representation through a PDU class constructor, the argument passed must be a valid network representation of a PDU. The `status()` member function may be used to determine whether a *DtPdu* represents valid data after construction. If `status` returns `DtSTATUS_OK`, it represents valid data. Otherwise, results of using the *DtPdu* are undefined. The meaning of the value returned by `status` depends on the type of PDU, and is meant to give some indication of why the PDU representation is invalid.

Currently, `status` is not verified by all *DtPdu* constructors. Therefore, a `status` of `DtSTATUS_OK` does not necessarily mean that a PDU is valid. However, a `status` other than that means that the PDU is not valid.

To construct a *DtPdu* object from a network representation of a PDU whose type you do not know, you can use a *DtPduFactory* (defined in [pduFactory.h](#)) – specifically, its `createPdu()` member function.

5.2.3. *DtPduFactory*

A *DtPduFactory* implements a kind of “virtual construction”, by maintaining a table of associations between PDU kinds and functions that create a corresponding PDU class object.

The `createPdu()` function checks the PDU kind in the packet, then passes it and the buffer argument to the right creator function from the table. The creator in turn, is just a wrapper around the appropriate PDU class from-network-representation constructor (a constructor itself can’t be added to a table). A pointer to the newly constructed object is returned by `createPdu()`.

If the PDU kind in a packet does not have an individual PDU class implemented (that is, there’s no creator registered with the *DtPduFactory* for the PDU kind), a *DtUnknownPdu* is created. See “[Using DtUnknownPdu](#)” on page 5-13.

A pointer to the PDU factory that a *DtExerciseConn* uses to create PDU class objects can be obtained using `DtExerciseConn::pduFactory()`.

Given a buffer containing the network representation of a PDU (perhaps you read it from a file), you can use this *DtPduFactory* to construct the right type of PDU class as follows:

```
DtExerciseConn exConn(...);
...
// buffer contains the network representation of some PDU
char *buffer = ...;

// Grab a pointer to our DtExerciseConn's DtPduFactory
DtPduFactory* fact = exConn.pduFactory();

// Cast the buffer to a DtNetPacket* before passing it to
// createPdu to indicate you are treating it as the network
```

```
// representation of a PDU.
DtPdu* pdu = fact->createPdu((DtNetPacket*) buffer);
```

`createPdu()` rejects packets (returns NULL) that have a protocol version less than `DtProtocolVersionToRecvMin` or greater than `DtProtocolVersionToRecvMax`. These are global variables declared in [pdu.h](#). When using `DtDISVERS=5`, these default to 4 and 6 respectively so that DIS 2.0.4, IEEE 1278.1 and 2.1.4 PDUs are accepted. When using `DtDISVERS=3`, both default to 3 so that only DIS 2.0.3 PDUs are accepted.

By default, the *DtPduFactory* that a *DtExerciseConn* uses is a *DtDisPduFactory* (defined in [disPduFactory.h](#)) – a PDU factory that knows about all of the DIS PDU kinds and their corresponding VR-Link *DtPdu* classes. However, if you would like to change the mappings between PDU kinds and PDU classes, you can do so through *DtPduFactory*'s `addCreator()` function. (The most common situation when this is necessary is when you derived your own type of *DtPdu*s.)

Each PDU class has a static member function called `create()`, which can serve as that PDU's creator function. These creator functions have the following prototype.

```
DtPdu *create(const DtNetPacket *initial,
              DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
```

So in order to change a *DtExerciseConn*'s *DtPduFactory* so that it will create a *DtEntityStatePdu* to represent PDU kind number 10, you can do this:

```
DtExerciseConn exConn(...);
...
exConn.pduFactory()->addCreator(DtPduKind(10),
                                DtEntityStatePdu::create);
```

A NULL creator function can be added for a PDU kind to indicate that there is no PDU class for that kind, and *DtUnknownPdu* should be used instead.

5.2.4. Using External Buffers in a PDU Class

Each *DtPdu* object stores a network representation of the PDU it represents. It is a pointer to this network representation that is returned by `packet()`. Normally, the memory used for this network representation is allocated internally by the PDU class constructor and deleted by the destructor.

In some cases, you might want a specific area of memory to be used by the *DtPdu* class for storage of its network representation (for example, memory from the stack, or from a shared memory pool. For this purpose, all PDU class constructors have an optional buffer argument. Users can cast a pointer to a particular area of memory to a `DtBufferPtr` and pass it as the buffer argument to indicate that a *DtPdu* should store its network representation there:

```
char buffer[144];
// construct blank PDU, but use buffer to store network
// representation
DtEntityStatePdu pdu(DtBufferPtr(buffer));
```

However, the most common reason why we would want to have a *DtPdu* use an external buffer for storage of its network representation is increased efficiency. When constructing a *DtPdu* from a network representation of a PDU, if you specify that the memory already occupied by the network representation be used by the PDU to store its network representation, a buffer copy may be avoided:

```
DtNetFirePdu* netPacket = .....;
DtFirePdu(netPacket, DtBufferPtr(netPacket));
```

In fact, this is what *DtExerciseConn* does when it creates a PDU class object. It reads a packet from the network into a buffer, then instructs the PDU class object to use this buffer for its network representation. The data does not need to be copied into a buffer internal to the *DtPdu*.

There are four considerations to keep in mind when passing a pointer to external memory to the blank *DtPdu* constructor:

1. It is your responsibility to insure that this external buffer is large enough to accommodate the largest network representation that this particular *DtPdu* object will need.
2. It is your responsibility to insure that the lifetime of the buffer memory extends for the lifetime of the *DtPdu*.
3. Once you pass a buffer pointer to a *DtPdu* constructor, it belongs to the *DtPdu* object for the duration of its lifetime. Modifying the buffer will result in undefined behavior when later using the *DtPdu*'s mutator or inspector functions.
4. This memory will NOT be freed by the *DtPdu* object; since it was allocated by the application, it is the application's responsibility to free it.

5.2.5. Copying PDUs

DtPdu has the copy constructor and assignment operator overloaded. When you do an assignment, such as:

```
DtFirePdu pdu1, pdu2;
pdu1 = pdu2;
```

pdu1 now contains the same data as *pdu2*. The two objects do not, however, share any data objects.

As with the standard PDU class constructors, the *DtPdu* constructor has an optional buffer argument. (See [“Using External Buffers in a PDU Class”](#) on page 5-11.)

In all cases,

```
DtFirePdu pdu1;
DtFirePdu pdu2(pdu1, buffer);
```

has the same effect as

```
DtFirePdu pdu1;
DtFirePdu pdu2(buffer);
pdu2 = pdu1;
```

5.3. Creating Your Own PDUs

Many VR-Link users need to use various PDUs that are not part of the DIS standard, and thus not implemented in VR-Link. However, VR-Link can treat user-defined PDUs, just like it treats PDUs that are implemented in VR-Link.

There are two ways to create user-defined PDUs:

- ♦ Use *DtUnknownPdu* to represent your type of PDU
- ♦ Derive a class from *DtPdu*

Using *DtUnknownPdu* is sometimes easier for simple PDUs that can be represented by C-style structures. In essence, the *DtUnknownPdu* just serves as a wrapper around such a structure. Deriving your own *DtPdu* class requires some extra work up front, but it is easier to use in your application, especially if it's a variable-length PDU, or one that has several variants.

5.3.1. Using *DtUnknownPdu*

A *DtUnknownPdu* (defined in [unknownPdu.h](#)) can be used to represent PDUs for which there is no PDU class. This includes new or experimental PDUs that you develop.

To send a PDU using the *DtUnknownPdu* class:

1. Fill a buffer with the exact bytes you want to send on the network (probably by filling in a C-structure). Make sure this structure includes the DIS PDU header whose structure is defined in *pduHeader.h* (in the *include/packets* directory).
2. Pass this network representation to the *DtUnknownPdu* from-network-representation constructor.
3. Send the PDU using the normal *DtExerciseConn::send()* or *sendStamped()*.

For example, suppose you create a Test PDU with two fields called *a* and *b*. We will choose PDU kind number 220 to represent this PDU:

```
// Define the structure of your network representation
typedef struct NetTestPdu
{
    DtNetPduHeader header;
    DtNetInt32 a;
    DtNetInt32 b;
} NetTestPdu;

// Create an instance of the structure and fill in the net
// representation
NetTestPdu netPdu;
netPdu.header.version = 5;
netPdu.header.DtExercise = 1;
netPdu.header.DtKind = 220
...
netPdu.a = 10;
netPdu.b = 11;
```

```
// Create a DtUnknownPdu from this network representation
DtUnknownPdu pdu((DtNetPduHeader*) &netPdu);

// Send the PDU as you normally would
exConn.sendStamped(pdu);
```

On the receiving side, *DtUnknownPdu* does not have `addCallback()` or `removeCallback()` functions, since a *DtUnknownPdu* does not know what PDU kind you are interested in. But you can still register callbacks with the *DtExerciseConn* on a particular PDU kind, in our case 220. Note the cast of the integer 220 to a *DtPduKind* enum here.

```
exConn.addPduCallback(DtPduKind(220), testCallback, NULL);
```

Since 220 is not a PDU kind that VR-Link knows about, it will create a *DtUnknownPdu* and pass that to your callback function. You can cast the *DtPdu** to a *DtUnknownPdu** within your callback if you want to, but it's not necessary since the only useful thing you can do with a *DtUnknownPdu* is get its network representation, which is done through a *DtPdu* member function.

```
void testCallback(DtPdu* pdu, void *usr)
{
    // Get the PDU's network representation, and cast it to a
    // NetTestPdu structure, so that you can access the data
    NetTestPdu* netPdu = pdu->packet();

    int a = netPdu->a;
    int b = netPdu->b;
}
```

By using our “Net” types, such as *DtNetInt32* (defined in [NetTypes.h](#)), you insure platform independence. When you are on little endian machines, byte swapping is performed when assigning to a Net type, and when a Net type is implicitly cast to a native type.

5.3.2. Deriving Classes from *DtPdu*

VR-Link's *DtExerciseConn::send()* and *DtExerciseConn::sendStamped()* can send any type of PDU, as long as the object passed to it is of a type derived from *DtPdu*. So, in order to send user-defined PDUs, you can write a derived class, after which you can immediately use your derived class with these functions.

When *DtExerciseConn* receives a packet from the network, it creates an object of the appropriate class derived from *DtPdu*, based on the PDU type in the packet. For instance, a *DtEntityStatePdu* is created if a packet representing an Entity State PDU is received. For this reason, VR-Link needs to know that your derived class exists.

To add a new *DtPdu* class:

1. Write the derived class (as described in “[Writing a DtPdu Class](#)” on page 5-15).
2. Let VR-Link know about the class (described in “[Telling VR-Link about your New Class](#)” on page 5-20).

Writing a *DtPdu* Class

To derive a class from *DtPdu*:

1. Create the PDU layout, a structure that is used for the network representation of the PDU. For all of the VR-Link PDUs, these network representations appear in the *include/packets* directory.

The first field in all network representations should be a *DtNetPduHeader*.

If your PDU is variable length, it will not be possible to fully describe the PDU layout with a structure like this. Try to include as much of the layout as possible.

By using our “Net” types, such as *DtNetInt32* (defined in [NetTypes.h](#)), you ensure platform independence. When you are on little endian machines, byte swapping is performed when assigning to a Net type, and when a Net type is implicitly cast to a native type.

As an example, we’ll use the same Test PDU we worked with above. This test PDU has two fields, *a* and *b*, which are both ints. Here is the definition of the network representation structure:

```
typedef struct NetTestPdu
{
    DtNetPduHeader header;
    DtNetInt32 a;
    DtNetInt32 b;
} NetTestPdu;
```

2. Create the TestPdu class definition. Each class derived from *DtPdu* must have the two standard constructors possessed by all PDU classes. Your type of *DtPdu* may have additional constructors, but it must have the two standard ones. PDU classes also need to provide a definition for the virtual function *internalGetPduKind()*, which returns the PDU kind value being used for this type of PDU.

In addition your class should provide the static member functions *addCallback()* and *removeCallback()* for callback management, and the static member function *create()*, that returns a new instance of the class. The purpose of this function is discussed in [“Telling VR-Link about your New Class”](#) on page 5-20.

3. Optionally, provide inspector and mutator functions to examine and set the fields of your PDU, but VR-Link does not require that you do so.

The definition for the TestPdu class might look like this:

```
// First define the prototype for a TestPdu-specific
// callback function
class TestPdu;
typedef void (*TestPduCb)(TestPdu *pdu, void *usr);

class TestPdu : public DtPdu
{
public:
    // Blank PDU constructor
```

```
TestPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);  
// From-network-representation-constructor  
TestPdu(const NetTestPdu *initial,  
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);  
// Returns a new instance of TestPdu  
static DtPdu *create(const DtNetPacket *initial,  
                    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);  
// Callback management  
static void addCallback(DtExerciseConn *conn,  
                       TestPduCb cb, void *usr);  
static void removeCallback(DtExerciseConn *conn,  
                           TestPduCb cb, void *usr);  
  
// Set/inspect a  
void setA(int val);  
int a() const;  
// Set/inspect b  
void setB(int val);  
int b() const;  
  
protected:  
    DtPduKind TestPdu::internalGetPduKind() const;  
};
```

The blank PDU constructor should be defined as follows:

```
TestPdu::TestPdu(DtBufferPtr buffer)  
{  
    int staticSize = sizeof(NetTestPdu);  
    initPdu(staticSize, buffer);  
}
```

The size passed to `initPdu()` should be the size (in bytes) of the smallest legal PDU of this type. In the case of our `TestPdu`, which is not a variable length PDU, there can be only one size, which is the size of the `NetTestPdu` structure.

The from-network-representation constructor should look like this:

```
TestPdu::TestPdu(const NetTestPdu *initial, DtBufferPtr buffer)  
{  
    initPdu(initial, buffer);  
}
```

`initPdu()` is a member of the base class *DtPdu*, and it handles allocating memory for the network representation of the PDU, and other related items.

Our definition of `internalGetPduKind()` should return the value 220, which we are using for our `TestPdu`.

```
// Extend the DtPduKind enumeration with our chosen value  
const DtPduKind TestPduKind = DtPduKind(220);  
  
DtPduKind TestPdu::internalGetPduKind() const  
{  
    return TestPduKind;  
}
```



```
}
```

create() can be implemented as follows:

```
DtPdu *TestPdu::create(const DtNetPacket *initial,
                      DtBufferPtr buffer)
{
    return new TestPdu((NetTestPdu *) initial, buffer);
}
```

while the callback-related functions should look like this:

```
void TestPdu::addCallback(DtExerciseConn *conn,
                        TestPduCb cb, void *usr)
{
    conn->addPduCallback(TestPduKind, (DtPduCallbackFcn) cb, usr);
}

void TestPdu::removeCallback(DtExerciseConn *conn,
                           TestPduCb cb, void *usr)
{
    conn->removePduCallback(TestPduKind, (DtPduCallbackFcn)
                           cb, usr);
}
```

Writing the mutator and inspector functions to set and examine the data in your PDU is fairly straightforward. A pointer to the PDU's network representation is stored in *netPdu*, a member of the base class *DtPdu*. This void* must be cast to a pointer to your network representation structure (in this case *NetTestPdu*) before fields are examined and set. For instance:

```
#define netTestPdu ((NetTestPdu *) netPdu)

void TestPdu::setA(int x)
{
    netTestPdu->a = x;
}

int TestPdu::a() const
{
    return netTestPdu->a;
}

void TestPdu::setB(int x)
{
    netTestPdu->b = x;
}

int TestPdu::b() const
{
    return netTestPdu->b;
}
```

Variable-Length PDUs

Variable length PDUs are more difficult to work with than fixed-length PDUs. You need to call one of the following *DtPdu* member functions from your mutator functions every time you change the size or layout of the PDU:

- ♦ `insertBytes()`
- ♦ `deleteBytes()`
- ♦ `changeSize()`

You may need to do some pointer arithmetic to access certain fields of the PDU that cannot be accessed through members of the static structure.

`insertBytes()` inserts the indicated number of blank (zeroed) bytes at the desired offset into the PDU's network representation.

`deleteBytes()` deletes the desired number of bytes, starting with the indicated offset.

You can use `changeSize()` instead of `insertBytes()` or `deleteBytes()` to insert or delete bytes from the end of the PDU. Its argument indicates the new total size of the PDU.

These functions handle reallocating the memory for the network representation if necessary, and will also make sure the size field in the header is updated to reflect the new size.

Example

As an example, we'll use a hypothetical Variable PDU that has three fields, an integer *a*, an array of integers *b*, whose cardinality is the value of *a*, and a float called *c*, in that order.

We cannot represent this layout with a static structure, since we don't know the size of the array *b*, but we lay out as much as we can:

```
typedef struct NetVarPdu
{
    DtNetPduHeader header;
    DtNetInt32 a;
    // Arbitrarily choose a cardinality of 1,
    // but we won't rely on this.
    DtNetInt32 b[1];
} NetVarPdu;
```

We will be able to use this structure to get to fields *a* and *b*, but we'll need to do some pointer arithmetic to get to *c*.

The definition of our *VarPdu* class is very similar to the definition of our *TestPdu* in the previous example. The differences are confined to the implementation of the mutators and inspectors, and to the implementation of the blank PDU constructor:

```
VarPdu::VarPdu(DtBufferPtr buffer)
{
    int initialSize = sizeof(DtNetPduHeader) +
                     sizeof(DtNetInt32) + sizeof(DtNetFloat32);
```

```
    initPdu(initialSize, buffer);
}
```

The size passed to `initPdu()` is the size of the smallest legal *VarPdu*; that is, one with zero elements in the array *b*. We calculate this size by adding the size of the PDU header to the size of our integer *a* and our float *c*.

Since *a* represents the number of elements in the array *b*, we must insert or delete bytes from the PDU layout whenever we change *a* through its mutator:

```
#define netVarPdu ((NetVarPdu *) netPdu)

void VarPdu::setA(int num)
{
    // Get the old value of a
    int oldA = a();
    if (num > oldA)
    {
        // insert bytes after the end of the exiting b array
        insertBytes(netVarPdu->b[oldA], num - oldA);
    }
    else if (num < oldA)
    {
        // delete trailing elements of b
        deleteBytes(netVarPdu->b[num - 1], old - num);
    }

    // Now set the value of the a field
    netVarPdu->a = num;
}
```

The inspector for *a* can be implemented just as in the static case:

```
int VarPdu::a() const
{
    return netVarPdu->a;
}
```

Our *b* and `setB()` functions should take an index into the array, so that we know which element to inspect or set. We can do some bounds checking if we want:

```
void VarPdu::setB(int index, int val)
{
    if (index < 0 || index >= a())
    {
        printf("VarPdu::setB bounds error\n");
        return;
    }
    netVarPdu->b[index] = val;
}

int VarPdu::b(int index) const
{
    if (index < 0 || index >= a())
    {
        printf("VarPdu::b bounds error\n");
        return -1;
    }
}
```

```
    }  
    return netVarPdu->b[index];  
}
```

The inspector and mutator for the *c* field must do some pointer arithmetic to find where the data is stored, since it can't be accessed through the structure.

```
float VarPdu::c() const  
{  
    // We can find c after the last element of b  
    DtNetFloat32* netC = (DtNetFloat32*) &(netVarPdu->b[a()]);  
    // netC will be implicitly cast to a native float  
    return *netC;  
}  
  
void VarPdu::setC(float val)  
{  
    // We can find c after the last element of b  
    DtNetFloat32* netC = (DtNetFloat32*) &(netVarPdu->b[a()]);  
    // val will be implicitly cast to a DtNetFloat32  
    *netC = val;  
}
```

Telling VR-Link about your New Class

You must tell VR-Link about a new *DtPdu* class, so that a *DtExerciseConn* can create an instance of your class and pass it to your callbacks whenever it receives the appropriate type of PDU from the network.

Notification is done through the *DtExerciseConn*'s PDU factory, an instance of the *DtPduFactory* class defined in [pduFactory.h](#).

Use `DtPduFactory::addCreator()` to register your *DtPdu*'s creator function as follows:

```
DtExerciseConn exConn(...);  
...  
exConn.pduFactory()->addCreator(TestPduKind, TestPdu::create);
```

5.4. Configuring Your Connection to the DIS Network

As described in “[Connecting to Exercises](#)” on page 4-2, a *DtExerciseConn* serves as an application’s connection to a DIS or HLA exercise. The DIS version is defined in [dExerciseConn.h](#).

A DIS *DtExerciseConn* uses a *DtSocket*, defined in [Socket.h](#), as its low-level interface to a virtual network. *DtSocket* is an abstract class whose virtual *DtSend()* and *DtRecv()* functions allow you to send and receive a packet of data among applications without regard to the actual communication mechanism used. Subclasses of *DtSocket* implement different methods of communication and take different configuration parameters.

For example, a *DtNetSocket* (defined in [NetSocket.h](#)) is used when a *DtExerciseConn* wants to communicate directly with an IP/UDP network, while a *DtClientSocket* (defined in [ClientSocket.h](#)) is used when we wish to communicate through the VR-Link packet server.

DtExerciseConn has several constructors. Two of them create a *DtSocket* based on their arguments, and the third takes an existing *DtSocket* as an argument.

All constructors take an exercise ID as an argument. The exercise ID is copied into the header of outgoing PDUs when *sendStamped()* is used, and is used for filtering of incoming PDUs as well. Only those PDUs on the matching exercise are passed to the application by the exercise connection. If a *DtExerciseConn*’s exercise ID is set to zero, it will read and process PDUs from all exercises. However, PDUs should never be sent to exercise zero, so exercise ID zero should usually be used only in applications that are listen-only. Once a *DtExerciseConn* is constructed, you can change or examine the current exercise ID with *setExerciseId()* and *exerciseId()*.

The standard *DtExerciseConn* constructor, as described in “[Creating an Exercise Connection for DIS](#)” on page 4-3, checks to see if a packet server is running on the specified UDP port, creates a *DtClientSocket* to communicate with it if there is one, or creates a *DtNetSocket* to communicate directly to the network if there isn’t. In either case (when the packet server is not in internal mode), the *DtExerciseConn* will receive any unicast or broadcast packets destined for the machine on the specified port, and send packets to the broadcast address of the machine’s primary network interface by default (when no address is specified in a call to *send()* or *sendStamped()*).

The second constructor is similar, but takes a destination address in addition to the four arguments expected by the standard constructor. The destination address is used as the default IP address to which to send outgoing packets. It can be a unicast, multicast or broadcast address.

For example, to indicate that packets should be sent only to the machine with the address 207.86.232.1:

```
DtExerciseConn(3000, DtStringToInetAddr("207.86.232.1"));
```

In this case, we will still receive packets destined for our broadcast and unicast addresses.

This constructor is also useful when you want to use a device other than a machine's primary network device. Pass the broadcast address associated with the desired device as the destination address.

Note: The two devices must be on different networks. There is no way for a UDP socket to distinguish between two devices on the same network.

The third *DtExerciseConn* constructor takes a pointer to a *DtSocket*, and thus gives you full control of how the *DtSocket* is created. You can even derive your own type of *DtSocket* and pass an instance of it to a *DtExerciseConn*. (Perhaps you would like to create a *SharedMemorySocket*.) In this case, the socket will not be deleted by the *DtExerciseConn* object; since it was allocated by the user, it is his responsibility to delete it. However, the user must not delete the *DtSocket* until after the *DtExerciseConn* has been destroyed.

See the following sections for more configuration information. Once a *DtExerciseConn* is constructed, its `socket()` member function will return a pointer to the *DtSocket* it is using.

5.4.1. Configuring a *DtNetSocket*

If you choose to create your own *DtSocket* before passing it to a *DtExerciseConn*, you have several additional configuration choices.

If you are constructing a *DtNetSocket*, (defined in [NetSocket.h](#)), that class's constructors take a flags argument in addition to the port and destination address arguments described above. Flags can be a bitwise OR of zero or more of the following:

- ♦ `DtRead`
- ♦ `DtWrite`
- ♦ `DtReadWrite`
- ♦ `DtQuiet`
- ♦ `DtBindToDestAddr`.

By default, a *DtSocket*, and therefore a *DtExerciseConn*, can be used for both reading and writing to the network. But by creating a *DtNetSocket* using either the `DtRead` or `DtWrite` flag alone, you can create a read-only or write only connection. For example:

```
DtNetSocket sock(3000, DtWrite);
DtExerciseConn conn(&sock, 1, 1, 1);
```

The `DtQuiet` flag keeps the constructor from printing its startup diagnostics.

The `DtBindToDestAddr` flag is used to configure the socket to listen for only those packets destined for exactly the address specified as the destination address. For example, if you want to receive packets on a certain multicast address only, but not broadcast packets or unicast packets directed to your machine, you can specify the desired address as the destination address, and use `DtBindToDestAddr`:

```
DtNetSocket sock(3000, DtStringToInetAddr("225.5.6.7"),
                DtBindToDestAddr | DtReadWrite);
DtExerciseConn conn(&sock, 1, 1, 1);
```

5.4.2. Configuring a *DtClientSocket*

A *DtClientSocket*, defined in [ClientSocket.h](#), is a *DtSocket* that sends and receives packets with the help of the VR-Link packet server. Depending on the state of the packet server, packets may be sent to and received from the network or may be passed only among the server's clients on the local host.

The constructor tries to establish communication with the VR-Link Packet Server running on the specified port. If there is no server running on the port, there is a fatal error. To avoid this, an application can check whether a server is alive before attempting to create a *DtClientSocket* using the C-style function `DtPacketServerIsAlive()`, declared as follows:

```
int DtPacketServerIsAlive(int port);
```

For example:

```
DtSocket *skt;
if (DtPacketServerIsAlive(disPort))
{
    skt = new DtClientSocket(disPort);
}
else
{
    skt = new DtNetSocket(disPort);
}
```

In addition to the port and destination address arguments described in the *DtExerciseConn* section, *DtClientSocket* takes two additional optional arguments: a size (in bytes) for the queue through which the packet server passes incoming messages, and a flags argument, which is a bitwise OR of zero or more configuration flags. See [ClientSocket.h](#) for the list of valid flags.

When using a *DtClientSocket*, the function `isInternal` can be used to find out if the packet server we are connected to is in internal mode or not.

5.4.3. Multicast

DtExerciseConn allows subscription to one or more multicast addresses. Valid multicast addresses are those that lie between 224.0.0.0 and 239.255.255.255, but you should probably not use any that start with 224.0.0, in order to avoid potential conflict with addresses used by operating systems and network services.

Note: Not all platforms support multicast.

Interest in a particular multicast address is indicated using *DtExerciseConn::addInterestInMcastAddr()*. When a multicast group is no longer of interest, call *subtractInterestInMcastAddr()*.

Different code modules may add and subtract interest in a particular group independently. As long as the number of calls to *addInterest()* outnumbers calls to *subtractInterest()* for a particular group, packets on that multicast address are received.

- To send to a particular multicast address, include that address as the **destAddr** argument to *send()* or *sendStamped()*.

You do not have to be subscribed to a multicast address in order to send to it.

If the destination address passed to a *DtExerciseConn* constructor is a multicast address, you do not need to call *addInterest()* for that address. You will automatically be subscribed.

Note: On many platforms, multicast subscription works on a per-host basis, rather than on a per-application basis. This means, that if you have several applications listening to the same port on the same machine, and one of them subscribes to a particular multicast address, all of the applications will receive packets on that multicast address, even if they did not explicitly subscribe.

If there are several routers on your network between machines that are exchanging PDUs using multicast, it may be necessary to increase the Time To Live (TTL) for outgoing multicast packets, to insure that the packets are actually forwarded across the routers. This can be achieved using *DtSocket::setMcastTtl()*.

5.4.4. Filtering

A *DtExerciseConn* can instruct its *DtSocket* to filter out PDUs based on certain criteria. In this way, unwanted PDUs require no processing by the *DtExerciseConn*. This is especially valuable if the *DtExerciseConn* is using a *DtClientSocket* connected to a Packet Server. Filtering is then done in the asynchronous Packet Server, which may be running on a different processor than your application.

By default, packets that are not of an expected protocol version and packets whose exercise IDs do not match the exercise ID of the *DtExerciseConn* are filtered out. Expected protocol versions are those between the values of the global variables **DtProtocolVersionToRecvMin** and **DtProtocolVersionToRecvMax**, declared in [pdu.h](#). These parameters default to 4 and 6 respectively, if you are building with DtDISVERS=5 (so that we accept DIS 2.0.4, IEEE 1278.1 and 2.1.4 PDUs), and to 3 and 3 if you are building with DtDISVERS=3 (so that we accept DIS 2.0.3 PDUs only).

In addition, the *DtExerciseConn* maintains a list of all PDU kinds in which it is interested. This includes all PDU kinds on which callbacks have been registered, as well as any PDU kinds in which an interest has been explicitly added using *DtExerciseConn::addInterestInPduKind()*. (*subtractInterestInPduKind()* decrements interest in a kind). The *DtExerciseConn* will have its *DtSocket* filter out all PDU kinds in which it has no interest. The *DtPduKind* enumeration can be found in [disEnums.h](#).

Filtering can be turned off and on with *DtExerciseConn*'s *disableFiltering()* and *enableFiltering()* functions.

5.4.5. Bundling and Unbundling PDU Packets

VR-Link supports the bundling of multiple PDUs into a single network packet, and the unbundling of such packets into their constituent PDUs.

Bundling is off by default, but may be turned on using *DtSocket*'s *setBundling()* function. Pass to it the desired maximum size for a packet. When bundling is on, a sequence of packets passed to a *DtSocket*'s *DtSend()* function (for example, from *DtExerciseConn::sendStamped()*), are concatenated together. The bundle is passed to *DtSocket::sendTo()* whenever a packet would push the bundle size beyond **maxSize**.

A good choice for **maxSize** is the maximum size of the data portion of an ethernet packet: 1464 bytes. If you are using a networking scheme other than ethernet, use the maximum packet size for that scheme. Bundling is turned off by passing a **maxSize** of 0 to *setBundling()*.

Sending of the current bundle can be forced with *DtSocket::flush()*. If you are using bundling, *flush()* should be called at the end of each simulation frame, to ensure that all packets generated during a frame get sent before the next frame begins.

isBundling() returns 0 if bundling is off, and the maximum bundle size otherwise.

Unbundling is enabled by default, but the capability can be turned on or off by passing 0 or 1 to *DtSocket::setUnbundling()*. With unbundling on, when a packet arrives containing more than one PDU, successive calls to *DtRecv()* (the call made by *DtExerciseConn::netRead()*) return successive PDUs from the packet. On the next call to *DtRecv()* after the last PDU from a packet has been returned, the first PDU from the next packet is returned.

If unbundling is disabled, but a bundled packet is received, all but the first PDU in the packet is ignored.

When you are using the packet server, unbundling takes place in the server. So for *DtClientSockets*, `setUnbundling()` merely instructs the packet server to turn unbundling on. Since another *DtClientSocket* might turn unbundling on or off in your client's packet server unbeknownst to you, it is possible that your *DtClientSocket* will unbundle when you have turned unbundling off with `setUnbundling()` or vice versa.

5.4.6. *DtSocket* Specifics

A *DtSocket* is an object through which packets can be sent to and received from other applications, without regard to the actual communications mechanism used.

Sending Packets

There are two functions involved in the sending of packets: `DtSend()` and `sendTo()`. `sendTo()` is a virtual function that, in derived classes, does the actual sending of the data. `DtSend()` performs any processing common to all *DtSockets* (such as bundling) before passing data on to `sendTo()` for actual sending.

A buffer, the number of bytes to send, and an optional address are passed to these functions. If the address is omitted, the *DtSocket's* default destination address is used. If the packet cannot be sent, -1 is returned. Otherwise, the number of bytes actually sent is returned. `DtExerciseConn::send()` and `sendStamped()` use `DtSocket::DtSend()` to send DIS PDUs.

DtSockets don't know anything about byte swapping or PDU layout (with the exception of some knowledge of PDU header layout used for filtering). The data contained in the buffer passed to *DtSocket's* sending functions are the exact bytes that will go out onto the network.

Receiving Packets

`DtRecv()` is used to receive packets through a *DtSocket*. This is the function called by `DtExerciseConn::netRead()` to read a PDU from the exercise. There are three different versions of `DtRecv()`. The version below reads the next packet available, and copies up to *size* bytes into buffer *buffer*. The size of the packet is returned. If there is no packet available, 0 is returned:

```
int DtRecv(caddr_t buffer, size_t size);
```

The following version sets *buffptr* to point to an internal buffer containing the packet read, and returns the size of the packet. If there is no packet available, *buffptr* is set to NULL, and size 0 is returned:

```
int DtRecv(caddr_t *buffptr, int *status);
```

If status is non-NULL, then it is set to indicate a return status. The status is DtSOCKET_PACKET if a packet was successfully received, and DtSOCKET_NO_PACKET if no packets are available. If a packet is received but is filtered out, the return code is set to DtSOCKET_FILTERED_PACKET, *buffptr* is set to NULL and 0 is returned. Return codes other than these may be used when DtRecv() returns because it received a control message that does not represent an actual packet on the virtual network.

For callers uninterested in the packet size, the following version simply returns a pointer to an internal buffer containing the packet read, or NULL if none is available:

```
caddr_t DtRecv();
```

***DtSocket* Functions**

Table 5-2 describes *DtSocket* member functions other than send and receive functions.

Table 5-2: DtSocket member functions (Sheet 1 of 2)

Function	Description
lastSourceInetAddr()	returns the IP address of the sender of the packet received in the last call to any of the DtRecv() functions. If an implementation does not support report of source IP address, an address of 0 is returned. DtReportPktsRcvd() returns the total number of packets read through the <i>DtSocket</i> since it was constructed.
DtBecomeNonBlocking()	ensures that reads and writes are non-blocking. This is called by all <i>DtExerciseConn</i> constructors, even for sockets which are passed to <i>DtExerciseConn</i> already constructed. If this function is not called, <i>DtNetSocket</i> reads and writes and <i>DtClientSocket</i> reads will block. <i>DtClientSocket</i> writes are always non-blocking.
readFd()	return the read file descriptors. Those types of <i>DtSockets</i> that do not have associated file descriptors return -1.
writeFd()	return the write file descriptors. Those types of <i>DtSockets</i> that do not have associated file descriptors return -1.
andnotifyMe()	lets the <i>DtSocket</i> know that the caller is using the presence of data on the <i>DtSocket's</i> read file descriptor as an indicator that there are packets available on the virtual network. This mechanism is useful for applications that are event-driven in nature, and wish to wait via a DtSelect() for input from one of various devices. The <i>DtSocket</i> will ensure that there is data on the read file descriptor whenever there are packets available, even if the underlying implementation does not cause this to be true automatically.

Table 5-2: DtSocket member functions (Sheet 2 of 2)

Function	Description
listenToMulticastGroup() dropMulticastGroup()	do what their names imply, but rarely need to be called directly, since they are called by the preferred DtExerciseConn::addInterestInMcastAddr() and subtractInterestInMcastAddr(). Do not use these functions if you are using the DtExerciseConn versions in the same application.
setMcastTtl()	Sets the IP time-to-live (TTL) field for multicast packets to ttl hops (or seconds). The TTL field controls the number of routers a packet is forwarded through. The DtSocket constructor does not initialize the multicast TTL to any particular value. Instead, the system default is used. Rather than calling setMcastTtl() in an application, you may want to change the system-wide default for multicast TTL (on platforms that support this). For example, under Solaris2, the following sets the system default to 64 hops: <pre>ndd -set /dev/ip ip_broadcast_ttl 64</pre> setMcastTtl() returns 1 on success and 0 on error (for example, on a platform that does not support multicast).
isOk()	Returns 1 if the DtSocket is currently able to send and receive packets, 0 otherwise (for example, if a DtClientSocket's packet server is down).
exerciseld() protocolVersion() pduKind() ipAddr()	Return pointers to DtFilterTests (defined in Socket.h) that control filtering on various attributes of PDUs. However, most of these features are better accessed through the relevant DtExerciseConn functions.

5.5. Intercepting Incoming Entity State PDUs

In the Chapter 4, *The Protocol-Independent Interface*, we mentioned that intercepting incoming state updates for a particular entity had to account for the protocol, since the concept of an HLA attribute update is fairly different from the concept of a DIS entity state PDU.

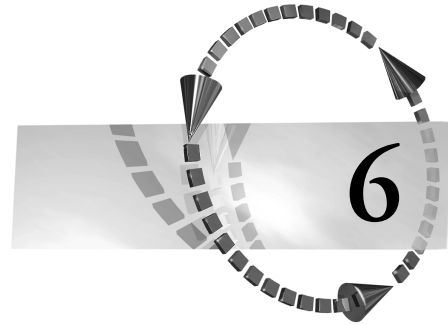
In DIS, there are two ways to intercept incoming entity state PDUs:

- ♦ Register a callback on entity state PDUs, just as you would on any other kind of PDU.
- ♦ Override `DtReflectedEntity::processEntityState()`.

When an entity state PDU arrives, it is dispatched to a callback that *DtReflectedEntityList* registers with the *DtExerciseConn*. Within this callback, the REL calls the appropriate *DtReflectedEntity*'s `processEntityState()`, depending on the entity ID in the PDU.

DtReflectedEntity's default implementation of this virtual function is to update its own *DtEntityStateRepository* based on the contents of the PDU. But by subclassing *DtReflectedEntity*, you can provide an alternate implementation. Your version of this function should probably call down to `DtReflectedEntity::processEntityState()` to make sure that the ESR is properly updated even if there is additional work you want to perform here.

Remember that when subclassing *DtReflectedEntity*, you must also subclass *DtReflectedEntityList* so that it knows to create your new type of *DtReflectedEntity*. See [“Creating Reflected Entity Lists”](#) on page 4-16 for examples of subclassing these two classes.



HLA-Specific Interface

This chapter describes VR-Link's HLA-specific parameters and features:

- ♦ [The RTI](#), on page 6-2
- ♦ [Managing HLA Objects](#), on page 6-4
- ♦ [Managing HLA Objects](#), on page 6-4
- ♦ [Subscribing to Object Classes](#), on page 6-7
- ♦ [Discovering and Removing Objects](#), on page 6-8
- ♦ [Working With Interactions](#), on page 6-11

Note: If your application is intended for both DIS and HLA, or if you plan to add this support in the future, enclose your HLA-specific code within a preprocessor directive pair to ensure that it will not be compiled when you try to build for DIS:

```
#if DtHLA
// HLA-specific code
#endif
```

6.1. The RTI

Although most applications based on VR-Link will never need to access the RTI directly (VR-Link functions make the RTI calls for you), some applications will want the flexibility of making their own calls to the RTI. Often, you will want to mix and match, letting VR-Link handle most interaction with the RTI, while handling some of it directly within the same application.

The RTI's API contains two main classes: the *RTIambassador* and the *FederateAmbassador*. When a *DtExerciseConn* is constructed, it creates instances of both of these classes.

6.1.1. Federate-Initiated Services

Federate-initiated services (such as `publishObjectClass()`, `updateAttributeValues()`, and `sendInteraction()`) are invoked by making calls to *RTI::RTIambassador* member functions.

A *DtExerciseConn*'s RTI ambassador is available through the `rtiAmb()` member function. Use this object to invoke any federate-initiated RTI services.

For example, to call the RTI's `requestPause()` service:

```
DtExerciseConn exConn(...);  
...  
RTI::RTIambassador* rtiAmb = exConn.rtiAmb();  
rtiAmb.requestPause(...);
```

See the separate RTI documentation for information on the RTI's functions.

6.1.2. RTI-Initiated Services

RTI-initiated services (such as `discoverObject()`, `reflectAttributeValues()`, and `receiveInteraction()`) are invoked when the RTI calls virtual member functions of *RTI::FederateAmbassador()* from within *RTI::RTIambassador::tick()*. Federates are responsible for deriving a class from *RTI::FederateAmbassador* and providing definitions for these pure virtual functions.

Normally, this responsibility is satisfied through VR-Link's *DtVrlFederateAmbassador* class, defined in [vrlFedAmb.h](#). This class is derived from *RTI::FederateAmbassador*, and includes definitions for all virtual functions (though some are empty definitions).

Subclassing *DtVrlFederateAmbassador*

For direct access to the data the RTI provides through RTI-initiated services, you can derive from *DtVrlFederateAmbassador*, and implement new definitions for the virtual functions in your subclass. In general, you should call the *DtVrlFederateAmbassador* versions of these functions from your versions, to insure that any VR-Link functionality that may depend on this continues to work.

The implementation of *DtVrlFederateAmbassador*'s `initiatePause()` member function is empty. VR-Link does not take any action by default when this service is invoked by the RTI. In order for your application to be able to handle this service invocation, you can provide your own definition for the `initiatePause()` virtual function, for example:

```
class MyFedAmb : public DtVrlFederateAmbassador
{
public:
    virtual void initiatePause(const RTI::PauseLabel label)
        throw (RTI::FederateAlreadyPaused,
              RTI::FederateInternalError);
};

void MyFedAmb::initiatePause(const RTI::PauseLabel label)
    throw (RTI::FederateAlreadyPaused,
          RTI::FederateInternalError)
{
    // Your code to handle the pause
    ...
    // Call superclass version in case it's
    // doing anything important
    DtVrlFederateAmbassador::initiatePause(label);
}
```

Telling *DtExerciseConn* About Your Derived Class

After you create a subclass, you need to tell *DtExerciseConn* that it should use an instance of your class rather than the default *DtVrlFederateAmbassador* as its federate ambassador. This is achieved using the static member function `DtExerciseConn::setFedAmbCreator()`. This function expects a `DtFedAmbCreator()` function as an argument – a function that returns a new instance of a *DtVrlFederateAmbassador* or a subclass thereof. `setFedAmbCreator()` needs to be called before a *DtExerciseConn* is constructed to take effect, since it is within the *DtExerciseConn* constructor that the current `DtFedAmbCreator()` is called to produce the federate ambassador to use.

A `FedAmbCreator()` function for our *MyFedAmb* class might look like this:

```
DtVrlFederateAmbassador* MyFedAmbCreator()
{
    return new MyFedAmb();
}
```

And we tell the *DtExerciseConn* class about this function before creating a *DtExerciseConn* instance as follows:

```
DtExerciseConn::setFedAmbCreator(MyFedAmbCreator);
...
// The DtExerciseConn we create here will use a MyFedAmb as
// its federate ambassador
DtExerciseConn exConn(...);
```

A *DtExerciseConn*'s *DtVrlFederateAmbassador* is available through its `fedAmb()` member function.

6.2. Managing HLA Objects

In HLA, information about the state of simulated objects is exchanged through updates of published attributes of objects. Each object in the HLA world must belong to some object class defined in the FOM. The set of valid attributes of each Object Class is also defined in the FOM.

VR-Link uses instances of the *DtHlaObject* class to represent individual HLA objects – both locally simulated and reflected. *DtHlaObject* is defined in [hlaObj.h](#).

Because most entity management functionality is available through VR-Link's protocol-independent layer, most applications do not need to interact with *DtHlaObjects* (or even know they exist). However, there are always *DtHlaObjects* behind the scenes. On the outgoing side, *DtEntityPublisher* creates and uses a *DtHlaObject* to help manage the sending of attribute updates. On the incoming side, a *DtReflectedEntity* uses a *DtHlaObject* to help manage the reception of incoming attribute updates.

6.2.1. Managing Locally Simulated Objects

Create a *DtHlaObject* for each object that you are simulating in your application. Normally, you do not need to explicitly create one, since the *DtEntityPublisher* constructor will do this for you. You can obtain a pointer to a *DtEntityPublisher*'s *DtHlaObject* using `DtEntityPublisher::hlaObject()`.

Choosing an Object Class

The particular HLA object class that is used to represent your object is chosen by *DtEntityPublisher* based on the arguments passed to its constructor as follows:

- ♦ The standard *DtEntityPublisher* constructor takes a *DtEntityType* enumeration to describe the type of entity you are simulating. If this constructor is used, *DtEntityPublisher* chooses an appropriate class from the RPR FOM object class hierarchy, by which to represent the object in the shared HLA world.

For example, if you pass an entity type that represents an M1A1 Tank (1:1:225:1:1:0:0), *DtEntityPublisher* chooses the *MilitaryPlatformLand* class.

- ♦ If you would like to explicitly choose the object class to represent a particular entity, you can use a second *DtEntityPublisher* constructor, that takes an *RTI::ObjectClassHandle* as an argument. Since the RTI assigns handles to the different object classes at run-time, class handles must not be hard-coded. Instead, use the RTI's `getObjectClassHandle()` function to obtain a handle from a class name. For example:

```
DtExerciseConn exConn(...);
...
// Obtain the handle for the class we want from the RTI
RTI::ObjectClassHandle handle =
    exConn.rtiAmb()->getObjectClassHandle("MilitaryPlatformLand");
// Create the DtEntityPublisher that uses this class
DtEntityPublisher entityPub(handle, &exConn);
```

After it creates a *DtHlaObject*, *DtEntityPublisher* immediately calls the *DtHlaObject*'s *setupUpdating()* function. This function publishes the object class (if not already published), registers the object's ID with the RTI through the *registerObject* service, and adds the object to a VR-Link-maintained list of all HLA objects that are currently set up for updating.

Note: “Publishing” an object class means informing the RTI that the application will be simulating objects of this class. Contrast this with “updating”, or actually sending attribute values for a specific object that you are simulating.

DtHlaObject's *cleanupFromUpdating()* function invokes the *deleteObject* RTI service to inform the exercise that the object no longer exists in the shared world, and removes the object from VR-Link's list of objects currently set up for updating. This function is normally called by the *DtEntityPublisher*'s destructor immediately before deleting the *DtHlaObject*.

> To find out what object class is actually being used to represent an entity, call *DtHlaObject*'s *objectClassHandle()* member function.

Similarly, its HLA object ID is available through *objectId()*:

```
DtExerciseConn exConn(...);
...
DtEntityPublisher entityPub(DtEntityType("1:1:225:1:1:0:0", ),
                             &exConn);
DtHlaObject* hlaObj = entityPub.hlaObject();
RTI::ObjectID = hlaObj->objectId();
RTI::ObjectClassHandle handle = hlaObj->objectClassHandle();
```

The Local Object Manager

The list of all HLA objects simulated by your application is available through your *DtExerciseConn*'s Local Object Manager, which can be obtained through *DtExerciseConn*'s *localObjectManager()* function. The *DtLocalHlaObjectManager* class is defined in *locObjMgr.h*. Its *allHlaObjects()* member function returns a *DtList* of pointers to *DtHlaObjects* that are currently set up for updating.

For example, to print the ID's of all objects currently being locally simulated:

```
DtExerciseConn exConn(...);
...
// Create some locally simulated objects, either through
// DtEntityPublisher, or by creating DtHlaObjects directly, and
// calling setupUpdating
...
DtLocalHlaObjectManager* localObjMgr =
    exConn.localObjectManager();
DtList* allObjects = localObjMgr.allHlaObjects();
for (DtListItem* item = allObjects.first(); item; item =
    item->next())
```

```
{  
    // Cast the generic void* to a DtHlaObject*  
    DtHlaObject* obj = (DtHlaObject*) item->data();  
    print("ID: %ld\n", obj->objectId());  
}
```

The local object manager can also be used to look up a *DtHlaObject* by ID, using the function `hlaObject()`. NULL is returned if no such HLA object is being locally simulated.

Updating Attributes

DtHlaObject has a virtual function called `update()`, which determines what data needs to be sent to the exercise, and sends that data by making a call to the RTI's `updateAttributeValues` service. If this sounds similar to the *DtEntityPublisher*'s `tick()` function, that's because `tick()` calls its *DtHlaObject*'s `update()` to do its work.

There are several HLA rules that indicate when the value of a particular attribute of an object needs to be updated by the federate that owns that attribute:

- Attributes to which no federate is subscribed never need to be updated.
- Only those attributes whose values are “out of date” with respect to some update criteria need to be sent to an HLA exercise during a particular simulation frame. An attribute whose value hasn't changed since it was last updated, for example, need not be updated again.
- Any federate may request that a particular set of attributes of an object (or of all objects of a particular class) be updated even if their values are not out of date. (This HLA feature is used most often by new arrivals in an exercise, who need to come up to speed on the current state of the world.)

DtHlaObjects implement these rules, which helps insure that only necessary data is sent.

When an object class is set up for updating in VR-Link, callbacks are automatically registered on the `startUpdates()` and `stopUpdates()` RTI services. Using the information provided by these service calls, VR-Link keeps track of all attributes that are currently subscribed to by some federate in the exercise. This set of “needed” attributes of a *DtHlaObject*'s object class can be obtained using the *DtHlaObject* function `attributesNeededByFederation()`.

A *DtHlaObject* instance also registers itself for callbacks on the `provideAttributeValues` RTI service, and in this way, tracks which attributes have been requested of the object since its last update. This set of “requested” attributes is available through `DtHlaObject::requestedAttributes()`.

These functions are used by the *DtHlaObjects* `update()` function as it makes decisions on what attributes need to be updated, and sends values for those attributes.

If you want to register your own callback functions that will be executed when the `startUpdates()` or `stopUpdates()` RTI services are invoked, you can do so using *DtVrlFederateAmabassador*'s `addStartUpdatesCb()` or `addStopUpdatesCb()`. Similarly, `removeStartUpdatesCb()` and `removeStopUpdatesCb()` can be used to unregister callbacks. These functions take an `ObjectClassHandle` along with a callback function, to indicate the particular class whose control messages you are interested in.

Callback functions should look like:

```
void myStartUpdatesCb(RTI::ObjectClassHandle objectClass,
                     const RTI::AttributeHandleSet& attributeSet,
                     void* usr);
void myStopUpdatesCb(RTI::ObjectClassHandle objectClass,
                     const RTI::AttributeHandleSet& attributeSet,
                     void* usr);
```

as defined in [striUpdCbList.h](#) and [stopUpdCbList.h](#) respectively.

6.2.2. Subscribing to Object Classes

The RTI presents remotely simulated objects to a federate (through the `discoverObject` and `reflectAttributeValues` services) only if the federate has subscribed to the object's class (or a superclass), using the RTI's subscription management services. Otherwise, objects are ignored.

Normally, you won't need to subscribe to object classes explicitly, since *DtReflectedEntityList* manages subscription for you.

When you create a *DtReflectedEntityList* using the standard protocol-independent constructor, VR-Link automatically subscribes to all object classes that represent entities in the RPR FOM – *BaseEntity*, *PhysicalEntity*, and their subclasses. (In addition to subscribing, we also invoke the `requestClassAttributeValueUpdate()` service in order to request that all objects of the class that currently exist in the exercise publish values for their attributes.)

If you want to have *DtReflectedEntityList* (REL) track only a subset of these object classes, you can use a secondary constructor that takes a list of *RTI::ObjectClassHandles* as an argument. The REL will subscribe only to the handles in the list. For example, if you are interested only in munitions and military platforms in the land domain:

```
DtExerciseConn exConn(...);
...
// Create the list
DtList handleList;
// Add desired handles to the list
handleList.add((void *) exConn.rtiAmb()->
               getObjectClassHandle("MilitaryPlatformLand"));
handleList.add((void *) exConn.rtiAmb()->
               getObjectClassHandle("MunitionEntityLand"));
// Create the REL using this list
DtReflectedEntityList rel(&exConn, &handleList);
```

For complete control of subscription and attribute update requests, including for object classes not managed by *DtReflectedEntityList*, you can call the RTI services directly. See “RTI-Initiated Services” on page 6-2 for details.

6.2.3. Discovering and Removing Objects

The section “Entities Joining and Leaving the Exercise” on page 4-21 describes a protocol-independent way for receiving notification about the coming and going of entities. This method involves subclassing *DtReflectedEntityList* and providing new definitions for the *entityAdded()* and *removeAndDelete()* functions.

Although this is the preferred method (due to protocol-independence, and ability to access the *DtReflectedEntity* associated with an object), this section describes an alternate method, which works at the level of *DtHlaObjects*. This method is particularly useful if you are dealing with non-entity objects that are not handled by the *DtReflectedEntityList* (such as *RadioTransmitters*).

DtHlaObjects are used to represent reflected objects. When a new object is discovered, the *DtExerciseConn* creates a *DtHlaObject* to represent it, based on the object class received through the *discoverObject* service.

After the *DtHlaObject* is created, it is added to a list maintained by VR-Link of all reflected HLA objects, then passed to any callback functions that have been registered with the *DtExerciseConn* on *discoverObjects()* for that object class. You can register callbacks on *discoverObject()* for any object class, using *DtExerciseConn*’s *addDiscoverObjectCallback()* member function. Callbacks can be unregistered with *removeDiscoverObjectCallback()*.

If you would like to be notified when any type of object is discovered, you can pass an *RTI::ObjectClassHandle* of 0 to *addDiscoverObjectCallback()*.

Callback functions that you provide should match the signature defined in *dscvObjCbInfo.h*, for example:

```
void myDiscoverObjectCb(DtHlaObject* obj, void* usr);
```

If you would like to be notified when an object is removed, you can use *DtHlaObject::addRemoveObjectCb()* to register a removal callback directly with the *DtHlaObject* of interest. Such functions can be unregistered with *removeRemoveObjectCb()*.

Alternatively, you can use *DtExerciseConn*’s *addRemoveObjectCallback()* and *removeRemoveObjectCallback()*, in which case you need to pass an object ID to specify which object you are interested in.

In either case, your callback functions should match the signature defined in *remObjCbInfo.h*. For example:

```
void myRemoveObjectCb(DtHlaObject* obj, void* usr);
```

When the *removeObject()* service is invoked by the RTI, the corresponding *DtHlaObject* is removed from VR-Link’s internal list of reflected HLA objects, passed to any callback functions registered on the removal of the object, then deleted.

As an example, these callback mechanisms are how *DtReflectedEntityList* is notified of the arrival and removal of new entities. When the REL's *discoverObject()* callback is invoked, it creates a new *DtReflectedEntity* by passing the *DtHlaObject* to its *newReflectedEntity()* function. (This function, in turn, passes the *DtHlaObject* to the *DtReflectedEntity* constructor.) When its *removeObject()* callback is invoked (REL registers one for each entity in the REL), the *DtReflectedEntity* is deleted using *DtReflectedEntityList::removeAndDelete()*.

The Reflected Object Manager

A list of all HLA objects currently being reflected to your application is available through your *DtExerciseConn*'s Reflected Object Manager, which can be obtained through *DtExerciseConn*'s *reflectedObjectManager()* function. The *DtReflectedHlaObjectManager* class is defined in *refObjMgr.h*. Its *allHlaObjects()* member function returns a *DtList* of pointers to *DtHlaObjects* that are currently being reflected.

For example, to print the ID's of all objects currently being reflected:

```
DtExerciseConn exConn(...);
...
// Subscribe to various object class, process discoverObjects
// and reflectAttributeValues calls. (This is usually achieved
// simply by creating a DtReflectedEntityList).
...
DtReflectedHlaObjectManager* refObjMgr =
    exConn.reflectedObjectManager();
DtList* allObjects = refObjMgr.allHlaObjects();
for (DtListItem* item = allObjects.first(); item; item =
    item->next())
{
    // Cast the generic void* to a DtHlaObject*
    DtHlaObject* obj = (DtHlaObject*) item->data();
    print("ID: %ld\n", obj->objectId());
}
```

The reflected object manager can also be used to look up a *DtHlaObject* by ID, using the function *hlaObject()*. NULL is returned if no such HLA object is being reflected.

6.3. Intercepting Reflected Attribute Updates

In Chapter 4, *The Protocol-Independent Interface*, we mentioned that intercepting incoming state updates for a particular entity had to account for the protocol, since the concept of an HLA attribute update is fairly different from the concept of a DIS entity state PDU.

In HLA, VR-Link routes attribute updates directly to the *DtHlaObject* whose attributes the update describes, which may process it however it desires. (Typically, they decode the message into a *DtReflectedEntity*'s *DtEntityStateRepository*). If you would like to be notified when an update message has been processed by a *DtHlaObject*, you can register a callback function with the *DtHlaObject*, using its `addPostReflectCb()` member function. (You can use `removePostReflectCb()` to unregister the callback).

These callback functions should look like the following, as defined in [objRefCbInfo.h](#):

```
void postReflectCb(const DtStateMsg& stateMsg,
                  DtHlaObject* obj,
                  void* usr);
```

Your function will be called immediately after the *DtHlaObject* has finished processing the message. Along with the *DtHlaObject* (and the `usr` pointer, of course), the message itself will be passed to your function.

DtStateMsg, defined in [hStateMsg.h](#), is basically a wrapper that contains the ID of an object whose state the message is updating, and the message itself – in the form in which we get it from the RTI: an `RTI::AttributeHandleValuePairSet()`. These two components of a *DtStateMsg* are available through its `objectId()` and `ahvps()` functions respectively.

Note: These callback functions are registered with a *DtHlaObject*, which does not know about the *DtReflectedEntity* through which you typically get attribute values. If you would like a pointer to the corresponding *DtReflectedEntity* to be available to your callback function, you can pass it as the `usr` pointer when you register the callback. For example:

```
// Our callback function
void postReflectCallback(const DtStateMsg& msg,
                        DtHlaObject* obj, void *usr)
{
    // Cast the usr pointer to the DtReflectedEntity that
    // we know it to be
    DtReflectedEntity* ent = (DtReflectedEntity* usr);

    // The message has already been processed, so the
    // DtReflectedEntity's ESR has already been updated
    //with the values contained in the message:
    DtEntityStateRepository* esr = ent->entityStateRep();
    ...
}
```



```
main()
{
    DtExerciseConn exConn(...);
    DtReflectedEntityList rel(&exConn);

    // Grab a pointer to the first entity in the list
    DtReflectedEntity* ent = rel.first();

    // Register a callback function on updates for this entity,
    // passing a pointer to our DtReflectedEntity as usr
    ent->hlaObject()->addPostReflectCb(postReflectCb, ent);
}
```

6.4. Working With Interactions

This section describes classes and functions used for HLA interactions.

6.4.1. Publishing Interactions

Just like for object classes, a federate must indicate to the RTI its intention to send interactions of a particular interaction class (publishing the interaction class), before actually sending any interactions of that class.

If you are using *DtExerciseConn*'s `send()` or `sendStamped()` in order to send *DtInteractions*, you need not take any other action in order to publish interaction classes. Each interaction class will be automatically published for you the first time you try to send an interaction of that class.

6.4.2. Subscribing to Interactions

VR-Link also automatically handles subscription to HLA interaction classes. Whenever you register a callback function with the *DtExerciseConn* on a particular interaction class, VR-Link subscribes to that class through the RTI.

One exception is if you pass an `InteractionClassHandle` of zero to *DtExerciseConn*'s `addInteractionCallback()` function to indicate that your callback function should be executed when an interaction of any subscribed class is received. In this case, we do not automatically subscribe to all classes. The application is responsible for subscribing to all classes in which it has an interest. This can be done directly through the RTI's `subscribeInteractionClass()` service.

6.4.3. Using the *DtInteraction* Class

The HLA version of the *DtInteraction* class is defined in [hInteraction.h](#). Interaction classes typically store their parameter values in their own native representation, but you can obtain this data in the RTI's representation (an `RTI::ParameterHandleValuePairSet()`) using `DtInteraction::phvps()`. In fact, this is what *DtExerciseConn*'s `send()` and `sendStamped()` functions use to obtain the object to pass to the RTI for sending.

A *DtInteraction* can be created or assigned from an RTI representation, using a constructor designed for this purpose, or the `setFromPhvps()` member function. These are used by *DtExerciseConn* to create the *DtInteractions* that get passed to an application's callback functions, as a result of a `receiveInteraction()` RTI service invocation.

DtInteraction has the virtual functions `interactionClassHandle()` and `numParameters()`, implemented in derived classes to return the appropriate class handles and number of possible parameters.

Note: `interactionClassHandle()` requires a pointer to an RTI ambassador as an argument, so that it can obtain the handle issued at run-time for its interaction class.

The interaction classes all have a static `classHandle()` member function, that can be used to obtain the handle for an interaction class without actually creating an instance of the class, or asking the RTI directly. For example, to obtain the handle being used for the *WeaponFire* interaction class:

```
DtExerciseConn exConn(...);
RTI::InteractionClassHandle handle =
DtFireInteraction::classHandle(exConn.rtiAmb());
```

DtInteraction Print-Related Functions

DtInteraction has three different virtual printing functions, which are partially implemented in derived classes. The version of `print()` with no arguments is the standard one – the one used by VR-Link's *netdump* utility. It uses the virtual `name()` function to print the name of the interaction, and uses `printData()` to print all fields of the interaction independently of protocol.

The version of `print()` that takes an RTI ambassador is used to print the actual hex values of each HLA parameter (as they would be passed to the RTI), with the help of `printHeader()` and `printParams()`. The ambassador is needed so that the data can be encoded in a `ParameterHandleValuePairSet()` before the values are printed. If `printParams()` is used by itself, you can set its second argument to 0 rather than the default 1 to indicate that only parameter name and handles should be printed and values omitted.

6.4.4. Using *DtInteractionFactory*

The object that a *DtExerciseConn* uses to create an appropriate interaction class object, is its interaction factory – an instance of the *DtInteractionFactory* class defined in [hInterFactory.h](#).

A *DtInteractionFactory* implements a kind of “virtual construction”, by maintaining a table of associations between interaction class handles and functions that create a corresponding interaction class object. Its `createInteraction()` function looks up the class handle of an incoming interaction, then passes the `RTI::ParameterHandleValuePairSet()` (along with the handle) to the creator function it finds to create the object. A pointer to the newly constructed object is returned.

Each interaction class has a static member function called `create()`, which serves as the creator function for the class. This is typically just a wrapper around the appropriate interaction class from-RTI-representation constructor (a constructor itself can't be added to a table).

If the interaction kind does not have an individual interaction class implemented (that is, there's no creator registered with the *DtInteractionFactory* for the class handle), a *DtUnknownInteraction* is created.

The interaction factory that a *DtExerciseConn* uses to create interaction class objects can be obtained using `DtExerciseConn::interactionFactory()`.

By default, the *DtInteractionFactory* that a *DtExerciseConn* uses is a *DtRprInteractionFactory* (defined in [rprInterFact.h](#)) – an interaction factory that knows about all of the RPR FOM interaction classes, and their corresponding VR-Link *DtInteraction* classes. However, if you want to change the mappings between interaction class handles and interaction class to construct you can do this through *DtInteractionFactory*'s `addCreator()` function. This capability is particularly useful when you derived your own type of *DtInteraction*.

You can add a NULL creator function for an interaction kind to indicate that there is no interaction class for that kind, and *DtUnknownInteraction* should be used instead.

6.5. Creating Interaction Classes

You might want to take advantage of HLA's flexibility to add new interaction classes and parameters that do not exist in the standard RPR FOM, and hence are not implemented in VR-Link.

VR-Link can deal with user-defined interactions, and can treat them just like it treats those interactions that are implemented in VR-Link.

There are two ways in which VR-Link can deal with user-defined interactions. You can either use VR-Link's *DtUnknownInteraction* to represent your type of interaction, or you can derive your own class from *DtInteraction* to represent it.

DtUnknownInteraction is a wrapper around the RTI representation of an interaction – `anRTI::ParameterHandleValuePairSet()` – so if you use this method, application code must encode and decode data to and from this form. Deriving your own *DtInteraction* class involves some extra work up front, but it is easier to use the interactions in your application.

In either case, you need to add the definition of your interaction to your FOM (which will probably be an extension of the RPR FOM), and most importantly, you need to add it to the FED file that you are using.

In our examples, we'll assume we are creating an interaction called *TestInteraction* that has two parameters – an integer called *Number*, and a character string called *String*.

- To use a new interaction in your federation, add the following lines to your FED file, within the list of interaction classes:

```
(class TestInteraction FED_BEST_EFFORT FED_TIMESTAMP
  (parameter Number)
  (parameter String)
)
```

6.5.1. Using *DtUnknownInteraction*

You can use a *DtUnknownInteraction* (defined in [bUnknownInter.h](#)) to represent interactions for which VR-Link has not implemented a *DtInteraction* subclass. This includes new interactions that you develop, for which you choose not to derive your own *DtInteraction* class.

Sending An Interaction Using *DtUnknownInteraction*

To send an interaction using the *DtUnknownInteraction* class:

1. Creating and fill out an `RTI::ParameterHandleValuePairSet()` – the RTI's representation of an interaction. Make sure that the sizes, data types, and byte ordering and alignment of all parameter values are exactly as dictated by the FOM.
2. Pass this RTI representation to the *DtUnknownInteraction*'s constructor, along with the correct interaction class handle, and a pointer to the RTI ambassador.

3. Send the interaction using the normal `DtExerciseConn::send()` or `sendStamped()`.

To send an *DtUnknownInteraction* representing our example *TestInteraction* from previous examples, with the values 1, and “hello” for the **Number** and **String** parameters respectively, you can do the following:

```
DtExerciseConn exConn(...);
...
// Create the RTI representation - a phvps with space for
// two parameters
RTI::ParameterHandleValuePairSet* pvSet =
    RTI::ParameterSetFactory::create(2);

// Obtain the interaction class handle for TestInteraction
RTI::InteractionClassHandle classHandle =
    exConn.rtiAmb()->getInteractionClassHandle("TestInteraction");

// Add the parameter values
RTI::ParameterHandle numberHandle =
    exConn.rtiAmb()->getParameterHandle("Number", classHandle);
DtNetInt32 netInt = 1;
pvSet.add(numberHandle, (const char*) &netInt,
    sizeof(DtNetInt32));

RTI::ParameterHandle stringHandle =
    exConn.rtiAmb()->getParameterHandle("String", classHandle);
char* netString = "hello";
pvSet.add(stringHandle, (const char*) netString,
    strlen(netString));

// Create the DtUnknownInteraction
DtUnknownInteraction inter(classHandle, *pvSet,
    exConn.rtiAmb());

// Send it!
exConn.sendStamped(inter);
```

Note: By using our “Net” types, such as *DtNetInt32* (defined in [NetTypes.h](#)), you insure platform independence. When you are on little endian machines, byte swapping is performed when assigning to a Net type, and when a Net type is implicitly cast to a native type.

Receiving Interactions

DtUnknownInteraction does not have `addCallback()` or `removeCallback()` functions, since a *DtUnknownInteraction* does not know what interaction class you are interested in. But you can still register callbacks with the *DtExerciseConn* on a particular interaction kind:

```
DtExerciseConn exConn(...);
...
// Obtain the interaction class handle for TestInteraction
RTI::InteractionClassHandle classHandle =
    exConn.rtiAmb()->getInteractionClassHandle("TestInteraction");
```

```
// The callback will need a pointer to the DtExerciseConn,  
// so pass it as the usr pointer  
exConn.addInteractionCallback(classHandle,  
                             testCallback, &exConn);
```

Since the *TestInteraction* is not an interaction class that VR-Link knows about, it will create a *DtUnknownInteraction* and pass that to your callback function. You can cast the *DtInteraction** to a *DtUnknownInteraction** within your callback if you want to, but it's not necessary since the only useful thing you can do with a *DtUnknownInteraction* is get its RTI representation, which is done through a *DtInteraction* member function. The signature for a receive interaction callback is defined in [recIntrCbInfo.h](#).

Example

```
void testCallback(DtInteraction* inter, void *usr)  
{  
    // Cast the usr pointer back to a DtExerciseConn*  
    DtExerciseConn* exConn = (DtExerciseConn*) usr;  
  
    // Get the interaction's RTI representation (phvps) so  
    // that you can access the data  
    RTI::ParameterHandleValuePairSet& pvSet = inter->phvps();  
  
    // Get the interaction class handle  
    RTI::InteractionClassHandle classHandle =  
        inter->interactionClassHandle();  
  
    RTI::ParameterHandle numberHandle =  
        exConn->rtiAmb()->getParameterHandle("Number", classHandle);  
  
    RTI::ParameterHandle stringHandle =  
        exConn->rtiAmb()->getParameterHandle("String", classHandle);  
  
    // Grab the parameter values  
    for (RTI::ULong i = 0; i < pvSet.size(); i++)  
    {  
        RTI::ParameterHandle currentHandle = pvSet.getHandle(i);  
  
        if (currentHandle == numberHandle)  
        {  
            RTI::ULong length;  
            DtNetInt32 netNum;  
            // copy the value of the Number parameter into netNum  
            pvSet.getValue(i, (const char*) &netNum, length);  
            // By implicitly casting from our Net type to a  
            // native type, bytes swapping occurs if necessary  
            int num = netNum;  
        }  
        else if (currentHandle == stringHandle)  
        {  
            RTI::ULong length;  
            char str[100];  
            // copy the value of the String parameter into str  
            pvSet.getValue(i, (const char*) str, length);  
        }  
    }  
}
```

6.5.2. Deriving Classes From *DtInteraction*

The `DtExerciseConn::send()` and `DtExerciseConn::sendStamped()` functions can send any type of interaction, as long as the object passed to it is of a type derived from *DtInteraction*. So, to send user-defined interactions, you can write a derived class, then use the class with these functions.

When VR-Link's *DtExerciseConn* receives an HLA interaction from the RTI, it creates an object of the appropriate class derived from *DtInteraction*, based on the interaction class handle received. For instance, a *DtFireInteraction* is created if the class handle corresponds to the RPR FOM's *WeaponFire* interaction class. For this reason, VR-Link needs to know that your derived class exists.

To add a new *DtInteraction* class:

1. Write the derived class (as described in the next section).
2. Let VR-Link know about the class, described in [“Registering A Class With DtExerciseConn”](#) on page 6-22.

In our example, we use the *TestInteraction* class described in [“Using DtUnknownInteraction”](#) on page 6-14. Again, the class has two parameters – an integer called **Number**, and a string called **String**.

Writing A *DtInteraction* Class

Interaction class typically store values for the various parameters in protected member variables in native representation.

Public member functions should include the following:

- ♦ A “blank” constructor that creates an interaction with zero values for all attributes
- ♦ A from-RTI-representation constructor and `setFromPhvps()` function that set the values from those in an `RTI::ParameterHandleValuePairSet()`
- ♦ A `phvps()` function that converts the values to a `RTI::ParameterHandleValuePairSet()` representation
- ♦ Implementations for the virtual `interactionClassHandle()`, `name()`, `numParameters()`, and `print()` functions
- ♦ Static `classHandle()`, `addCallback()` and `removeCallback()` functions
- ♦ A static `create()` function that returns a new instance of the interaction.

In addition, interaction classes typically have inspectors and mutators that provide access to the parameter values.

The class definition for our *TestInteraction* class might look like this:

```
// Define a type for TestInteraction callback functions
class TestInteraction;
typedef void (*TestInteractionCb)(TestInteraction* inter,
                                void* usr);
```

```
class TestInteraction : public DtInteraction
{
public:
    // Blank constructor
    TestInteraction();

    // From-RTI-representation constructor
    TestInteraction(const RTI::ParameterHandleValuePairSet& pvList,
        RTI::RTIambassador* amb);

    // destructor
    virtual ~DtViewControlInteraction();

    // Get the parameter handle value pair set for the
    // interaction.
    const RTI::ParameterHandleValuePairSet& phvps(
        RTI::RTIambassador* amb) const;

    // Set values from parameter value list.
    virtual void setFromPhvps(const RTI::ParameterHandleValuePairSet& pvList,
        RTI::RTIambassador* amb);

    // Get the interaction class handle.
    virtual RTI::InteractionClassHandle interactionClassHandle(
        RTI::RTIambassador* amb) const;

    // Get the number of parameters.
    virtual int numParameters() const;

    // Name of the interaction
    virtual const char* name() const;

    // Print the interaction's data.
    virtual void printData() const;

    // Get/set the "number" parameter
    int number() const;
    void setNumber(int num);

    // Get/set the "string" parameter
    const char* string() const;
    void setString(const char* str);

public:
    // Get the interaction class handle.
    static RTI::InteractionClassHandle classHandle(
        RTI::RTIambassador* amb);

    // Create a TestInteraction. Caller responsible for deletion.
    static DtInteraction* create(RTI::InteractionClassHandle h,
        const RTI::ParameterHandleValuePairSet& p,
        RTI::RTIambassador* amb);
```



```
// Add (register)/Remove (deregister) function to be
// called when interaction occurs.
static void addCallback(DtExerciseConn* conn,
    TestInteractionCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    TestInteractionCb cb, void* usr);

protected:
    int myNumber;
    char myString[100];

protected:
    // Used as return value by phvps - does not store state
    static RTI::ParameterHandleValuePairSet* thePvList;
};
```

The blank interaction constructor is usually very simple – just initialize data to zeros:

```
TestInteraction::TestInteraction() :
    DtInteraction(),
    myNumber(0)
{
    myString[0] = 0;
}
```

The from-RTI-representation constructor will just call `setFromPhvps()`, which we'll define later:

```
TestInteraction::TestInteraction(
    const RTI::ParameterHandleValuePairSet& pvList,
    RTI::RTIambassador* amb):
    DtInteraction(),
    myNumber(0)
{
    myString[0] = 0;
    setFromPhvps(pvList, amb);
}
```

Our destructor can be empty:

```
TestInteraction::~TestInteraction()
{
}
```

Most of the virtual function definitions are fairly straightforward:

```
RTI::InteractionClassHandle TestInteraction::
    interactionClassHandle(RTI::RTIambassador* amb) const
{
    return amb->getInteractionClassHandle(name());
}

int TestInteraction::numParameters() const
{
    // We have two parameters
    return 2;
}
```

```
const char* TestInteraction::name() const
{
    return "TestInteraction";
}

void TestInteraction::printData() const
{
    // Use our inspector functions
    printf("Number: %d\n", number());
    printf("String: %s\n", string());
}
```

Mutator and inspector functions are straightforward as well:

```
int TestInteraction::number() const
{
    return myNumber;
}

void TestInteraction::setNumber(int num)
{
    myNumber = num;
}

const char* TestInteraction::string() const
{
    return myString;
}

void TestInteraction::setString(const char* str)
{
    strcpy(myString, str);
}
```

Callback management functions are merely wrappers around similar functions in *DtExerciseConn* – where the actual mechanism resides:

```
void TestInteraction::addCallback(DtExerciseConn* conn,
    TestInteractionCb cb, void* usr)
{
    conn->addInteractionCallback(classHandle(conn->rtiAmb()),
        (DtReceiveInteractionCb) cb, usr);
}

void TestInteraction::removeCallback(DtExerciseConn* conn,
    TestInteractionCb cb, void* usr)
{
    conn->removeInteractionCallback(classHandle(conn->rtiAmb()),
        (DtReceiveInteractionCb) cb, usr);
}
```

The create function is implemented as follows. We'll see how this function is used in the section on letting VR-Link know about your interaction class:

```
DtInteraction* TestInteraction::create(
    RTI::InteractionClassHandle h,
    const RTI::ParameterHandleValuePairSet& p,
    RTI::RTIambassador* amb)
{
    return new TestInteraction(p, amb);
}
```

Most of the work in an interaction class is done by the phvps() and setFromPhvps() functions. These functions encode and decode the data stored in the interaction to or from the RTI's representation:

```
const RTI::ParameterHandleValuePairSet& TestInteraction::phvps(
    RTI::RTIambassador* amb) const
{
    // Create the static thePvList, if it has not been created yet
    if (!thePvList)
    {
        thePvList = RTI::ParameterSetFactory::create(
            numParameters());
    }
    // Empty the parameter/value list
    thePvList->empty();

    RTI::InteractionClassHandle handle =
        interactionClassHandle(amb);

    // Add the current values

    // Implicitly casting to a Net type insures that byte swapping
    // will occur if necessary
    DtNetInt32 netNum = number();
    thePvList->add(amb->getParameterHandle("Number", handle),
        (char*) &netNum, sizeof(DtNetInt32));
    thePvList->add(amb->getParameterHandle("String", handle),
        (char*) string(), strlen(string()));

    return *thePvList;
}

void TestInteraction::setFromPhvps(
    const RTI::ParameterHandleValuePairSet& pvSet,
    RTI::RTIambassador* amb)
{
    RTI::ParameterHandle numberHandle =
        amb->getParameterHandle("Number",
            interactionClassHandle(amb));

    RTI::ParameterHandle stringHandle = amb->getParameterHandle(
        "String", interactionClassHandle(amb));
```

```
for (RTI::ULong i = 0; i < pvSet.size(); i++)
{
    RTI::ParameterHandle currentHandle = pvSet.getHandle(i);
    if (currentHandle == numberHandle)
    {
        RTI::ULong length;
        DtNetInt32 netNum;
        // copy the value of the Number parameter into netNum
        pvSet.getValue(i, (const char*) &netNum, length);

        // Call the mutator function to set the value
        setNumber(netNum);
    }
    else if (currentHandle == stringHandle)
    {
        RTI::ULong length;
        char str[100];
        // copy the value of the String parameter into str
        pvSet.getValue(i, (const char*) str, length);

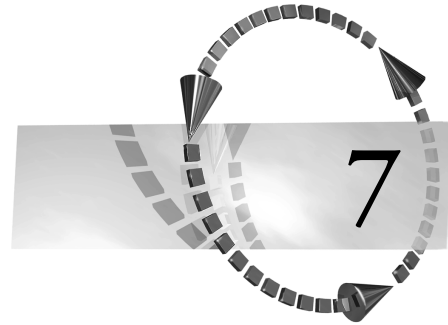
        // Call the mutator function to set the value
        setString(str);
    }
}
}
```

Registering A Class With *DtExerciseConn*

After you derive a class from *DtInteraction*, you must register your class's creator function with the *DtExerciseConn*'s interaction factory. Registering the class allows *DtExerciseConn* to create an instance of the class and pass it to your callbacks whenever it receives the appropriate type of interaction from the network.

You access the interaction factory using `DtExerciseConn::interactionFactory()`. *DtInteractionFactory*'s `addCreator()` function is used to actually register your own interaction class's creator function with VR-Link:

```
DtExerciseConn exConn(...);
...
exConn.interactionFactory()->addCreator(
    TestInteraction::classHandle(exConn.rtiAmb()),
    TestInteraction::create);
```



Utilities

VR-Link provides several utilities to help make it easier to write a simulation application. These utilities include coordinate conversion routines, matrix and vector classes and functions, time-related functions, and others.

This chapter describes the utilities as follows:

- ♦ [Vector and Matrix Classes](#), on page 7-2
- ♦ [Coordinate Conversions](#), on page 7-7
- ♦ [Managing Time](#), on page 7-17
- ♦ [Creating Linked Lists with DtList](#), on page 7-19
- ♦ [Diagnostic Utilities](#), on page 7-21
- ♦ [IP Address Manipulation Functions](#), on page 7-22
- ♦ [Miscellaneous Functions](#), on page 7-23

7.1. Vector and Matrix Classes

You can use the *DtVector* class, found in [v/Vector.h](#), to represent vectors in three dimensional space.

You can initialize *DtVectors* with their three components, as in:

```
DtVector vec(10.0, 20.0, 30.0);
```

The default constructor, which takes no arguments, constructs a vector containing three zeros.

The copy constructor, assignment operator, equivalence and subscripting operator are defined, so you can do things like:

```
DtVector vec(10.0, 20.0, 30.0);
DtVector vec2(vec);
DtVector vec3 = vec;
DtVector vec4;
vec4 = vec;
if (vec == vec2) {...}
```

and

```
double x = vec[0];
double y = vec[1];
double z = vec[2];
```

or

```
vec[0] = x;
vec[1] = y;
vec[2] = z;
```

DtVector also has a `string()` member function that returns a text representation of the contents of the vector, for example:

```
{    10.000,    20.000,    30.000}
```

The following static member functions of *DtVector* will return some commonly used vectors:

```
zero()  {0,0,0}
i()     {1,0,0}
j()     {0,1,0}
k()     {0,0,1}
ones()  {1,1,1}
```

DtDcm, found in [dcm.h](#), is used to represent 3x3 matrices. The name DCM stands for direction cosine matrix, and reflects the fact that we often use the class to store that particular type of matrix in order to represent an orientation. (Alternatively, orientation can be represented as a set of three Euler angles in the Tait Bryan sequence, using the *DtTaitBryan* class).

A *DtDcm* can be constructed from its nine components, as in:

```
DtDcm mat( 1.0, 2.0, 3.0,
           4.0, 5.0, 6.0,
           7.0, 8.0, 9.0);
```

or from three component vectors:

```
DtDcm ident(DtVector::i(),
            DtVector::j(),
            DtVector::k());
```

If the constructor is used with no arguments, a matrix containing nine zeros is constructed.

The copy constructor and assignment operators are defined, so you can do things like:

```
DtDcm mat(1,2,3,4,5,6,7,8,9);
DtDcm mat2(mat);
DtDcm mat3 = mat;
DtDcm mat4;
mat4 = mat;
```

The subscripting operator is defined as well, so you can obtain any of the three vectors that make up the matrix:

```
DtVector firstRow = mat[0];
```

In conjunction with *DtVector*'s subscripting operator, the familiar double subscript syntax is supported:

```
double upperLeft = mat[0][0];
double lowerRight = mat[2][2];
```

or

```
mat[0][0] = 10.0;
mat[1][1] = 20.0;
```

DtDcm has static member functions, `zero()` and `identity()`, that return the zero matrix and identity matrix respectively.

7.1.1. *DtVector* and *DtDcm* Semantics

Although VR-Link functions usually employ pass-by-pointer semantics when expecting objects that they will modify, functions that operate on *DtVectors* and *DtDcms* are exceptions. These objects are usually passed to VR-Link functions by reference, so that the subscripting operator can be used without the awkwardness of having to dereference a pointer. When these objects are not to be modified by a function, they are passed by

7.1.2. Vector and Matrix Manipulation Functions

LibMatrix.h contains a variety of C-style functions that operate on *DtVector* and *DtDcm*. The standard operations are included:

- ♦ addition
- ♦ subtraction
- ♦ multiplication
- ♦ scaling
- ♦ dot products
- ♦ cross products
- ♦ negation
- ♦ normalization
- ♦ determinant calculation
- ♦ transpose
- ♦ inverse
- ♦ and so on.

7.2. Orientation, Euler Angles, and DtTaitBryan

Although direction cosine matrices are often used in VR-Link to represent orientation, an alternative representation is a set of three Euler angles. Euler angles represent the orientation of a body in space with respect to a certain coordinate system as three ordered rotations needed to go from the reference coordinate system to the body coordinate system.

DIS and the RPR FOM in HLA use Euler angles to represent the orientation of an entity with respect to the geocentric coordinate axes. (The body coordinate system is defined by X out the front, Y out the right side, Z out the bottom of the entity.)

There are many possible ways to order the three rotations, but the one used by DIS and the RPR FOM, and hence VR-Link, is known as the Tait-Bryan sequence. The first rotation is about the Z axis, the second about the new Y axis, and the third about the newest X axis. The three angles are usually denoted by psi, theta, and phi respectively.

Note: These three angles do not necessarily correspond to the familiar heading, pitch and roll of an entity. Euler angles correspond to heading, pitch and roll only when the reference coordinate system is a topographic system. Specifically, the Euler angles sent in DIS entity state PDUs or RPR FOM entity updates use the geocentric coordinate system as a reference, and thus do not correspond to heading, pitch and roll. (See [“Coordinate Conversions”](#) on page 7-7 for information on converting between geocentric-reference Euler angles and topographic-reference Euler angles.)

VR-Link's *DtTaitBryan* class, defined in [taitBryan.h](#), is used to represent a set of Euler angles. A *DtTaitBryan* can be constructed from its three component angles, specified in radians, for example:

```
DtTaitBryan angles(3.14, 0.0, 1.57);
```

The class has inspector and mutator functions that provide access to the individual angles:

- ♦ psi()
- ♦ setPsi()
- ♦ theta()
- ♦ setTheta()
- ♦ phi()
- ♦ setPhi()

The assignment operator and copy constructor are implemented for *DtTaitBryan* as well. The `string()` member function provides a string representation of the contents of the object, in a similar fashion to `DtVector::string()`.

7.2.1. Converting Between Euler Angles and Matrix Representation

Euler.h contains functions that convert between the two representations of orientation.

Note: These functions are not converting orientations from one coordinate system to another; they are converting between two ways of expressing an orientation within a given coordinate system.

Given a set of Euler angles that represent the orientation of a body with respect to some reference frame (say, the geocentric frame, or a topographic frame),

DtEuler_to_BodyToRef() produces a rotation matrix (or *DtDcm*) that can rotate from body coordinates to reference coordinates, while DtEuler_to_RefToBody() produces its inverse – a matrix that can rotate from reference to body coordinates.

For example:

```
DtTaitBryan geocEulerAngles(...);  
DtDcm bodyToGeoc;  
DtEuler_to_BodyToRef(geocEulerAngles, bodyToGeoc);
```

The functions DtBodyToRef_to_Euler(), and DtRefToBody_to_Euler() go the other way – producing Euler angles from the matrix representation.

7.3. Coordinate Conversions

VR-Link supports many different coordinate systems and provides classes and functions for converting locations, vectors, and orientations from one system to another.

7.3.1. Geocentric Coordinates

DIS and the RPR FOM in HLA specify that world locations, velocities, accelerations and orientations be represented with respect to a right-hand geocentric Cartesian coordinate system. The origin of this geocentric coordinate system is the center of the earth, the positive X-axis passes through the prime meridian at the equator, the positive Y-axis passes through 90 degrees east longitude at the equator, and the positive Z-axis passes through the north pole (Figure 7-1).

The *DtVector* class is typically used to represent locations and vectors in geocentric coordinates, while *DtTaitBryan* or *DtDcm* is used to represent orientation.

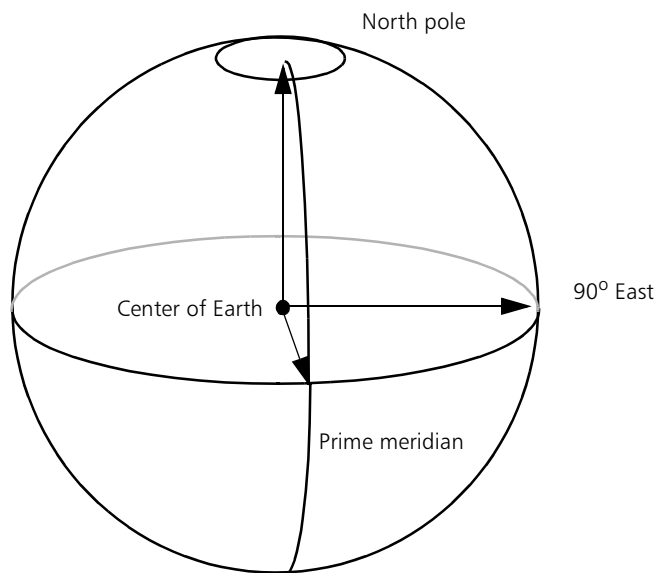


Figure 7-1. Geocentric coordinate system

7.3.2. Geodetic Coordinates

Geodetic coordinates are another means of specifying location in the world. VR-Link geodetic coordinates consist of latitude and longitude in radians, and altitude in meters above a certain reference ellipsoid.

VR-Link uses the class *DtGeodeticCoord* (defined in [geodCoord.h](#)) to represent geodetic coordinates. A *DtGeodeticCoord* can be initialized from a latitude, longitude and altitude. Negative latitudes are south latitudes; negative longitudes are west longitudes.

```
DtGeodeticCoord myLoc(DtDeg2Rad(45.0), DtDeg2Rad(30.0), 1000.0);
```

If the default constructor is used, the resulting object will represent a latitude, longitude and altitude of zero.

DtGeodeticCoord has the following inspectors and mutators to get and set values for the individual components of a geodetic coordinate:

- ♦ lat()
- ♦ setLat()
- ♦ lon()
- ♦ setLon()
- ♦ alt()
- ♦ setAlt()

DtGeodeticCoord also has member functions that enable you to convert to and from geocentric coordinates. *DtGeodeticCoord::geocentric()* returns the geocentric equivalent of a geodetic coordinate. For example:

```
DtGeodeticCoord geod(DtDeg2Rad(30.0), DtDeg2Rad(100.0), 1000.0);  
DtVector geoc = geod.geocentric();
```

After these lines of code, *geoc* will contain the coordinate:

```
{-960122.075, 5445122.868, 3170873.735}
```

which represents the same point in space as 30 degrees north latitude, 100 degrees east longitude, 1000 meters altitude.

A similar function is *DtGeodeticCoord::getGeocentric()*, which, rather than returning a geocentric coordinate, sets the value of an existing *DtVector*, for example,

```
DtGeodeticCoord geod(DtDeg2Rad(30.0), DtDeg2Rad(100.0), 1000.0);  
DtVector geoc;  
geod.getGeocentric(geoc);
```

Going the other direction, *DtGeodeticCoord::setGeocentric()* sets the value of a *DtGeodeticCoord* to the lat/long/alt equivalent of a given geocentric coordinate:

```
DtVector geoc(-960122.075, 5445122.868, 3170873.735);  
DtGeodeticCoord geod;  
geod.setGeocentric(geoc);
```

Also in [geodCoord.h](#), you'll find the C-style functions `DtGeocToGeod()` and `DtGeocToGeod()` that can be used instead of the member functions described previously to convert between geocentric and geodetic coordinates, although their use is discouraged in favor of the member functions.

Choosing a Reference Ellipsoid

The default reference ellipsoid used for geocentric to geodetic conversions is WGS84, but this is configurable through the function `DtUseMapDatum()`, declared in [geodCoord.h](#). Any *DtMapDatum* can be passed to this function. *DtMapDatum* is defined as:

```
typedef struct DtSpheroid
{
    DtFloat64 semiMajor;    // semimajor axis of the ellipsoid,
                           // in meters
    DtFloat64 semiMinor;    // semiminor axis of the ellipsoid,
                           // in meters
} DtSpheroid;

typedef struct DtMapDatum
{
    DtSpheroid spheroid;
    DtFloat64 datumShift[3]; // added to wgs84 to get GCC
                           // in this datum
} DtMapDatum;
```

VR-Link has the following *DtMapDatum*s pre-defined in [geodCoord.h](#):

```
DtWGS84, DtED50, DtNASANAD27, NASBNAD37, DtCONUSNAD27.
```

If you'd like to use the ED50 reference, for example, try

```
DtUseMapDatum(&DtED50);
```

7.3.3. Topographic Coordinates

VR-Link defines a topographic coordinate system as a right-handed Cartesian coordinate system whose X-Y plane is tangent to the earth's surface at the origin, with the positive X axis pointing north, the positive Y axis pointing east, and the positive Z axis pointing down (Figure 7-2). Obviously, there are an infinite number of topographic coordinate systems – one for each point on the earth's surface.

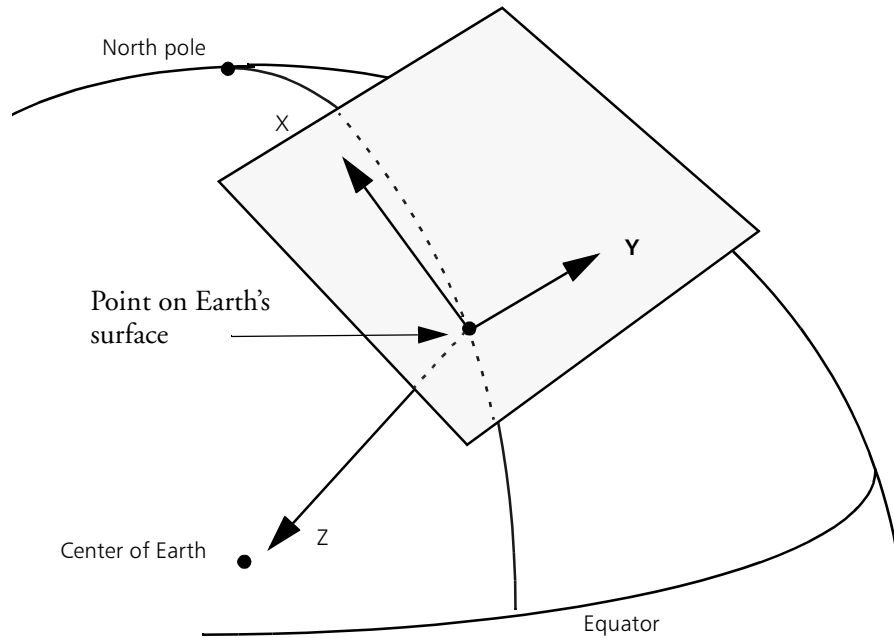


Figure 7-2. Topographic coordinate system

A *DtVector* is usually used in VR-Link to represent a topographic coordinate.

DtCoordTransform

The easiest way to convert between geocentric and topographic coordinates is by using a *DtCoordTransform* object, defined in [LibMatrix.h](#). A *DtCoordTransform* is an object that can transform locations, vectors or orientations from one cartesian coordinate system to another. In general, a *DtCoordTransform* can be constructed by passing to its constructor the origin of one system with respect to the other, and a rotation matrix that indicates the orientation of one system with respect to the other.

In the case of a geocentric/topographic transformation, however, VR-Link provides the function `DtGeocToTopoTransform()` to initialize a *DtCoordTransform* for you. This function, which is declared in [topoCoord.h](#), takes the latitude and longitude of a point on the earth, computes the rotation matrix that expresses the orientation of the topographic system defined by that point, and initializes a *DtCoordTransform* using this matrix along with the geocentric representation of the chosen point.

For example, to create a *DtCoordTransform* that can convert between geocentric coordinates and the topographic coordinate system whose origin is at 30 degrees north latitude, 100 degrees east longitude, you can do the following:

```
DtCoordTransform geocToTopo;  
DtGeocToTopoTransform(DtDeg2Rad(30.0), DtDeg2Rad(100.0),  
                      &geocToTopo);
```

Now, this transform can be used to convert from geocentric to topographic coordinates. For example, if we have a geocentric location in a *DtEntityStateRepository*, and we wish to convert it to our topographic frame, we can use *DtCoordTransform::coordTrans()*:

```
DtEntityStateRepository *esr = ...;  
DtVector topoLocation;  
geocToTopo.coordTrans(esr->location(), topoLocation);
```

If we want to do the reverse, we can initialize a second *DtCoordTransform* to convert from topographic to geocentric using the member function *setByInverse()*:

```
DtCoordTransform topoToGeoc;  
topoToGeoc.setByInverse(geocToTopo);
```

Now, we can use *topoToGeoc()* to convert to geocentric, say, for filling in a locally simulated entity's ESR:

```
DtEntityStateRepository *esr = ...;  
DtVector geocLocation;  
topoToGeoc.coordTrans(topoLocation, geocLocation);  
esr->setLocation(geocLocation);
```

In addition to *coordTrans()*, *DtCoordTransform* also has *vecTrans()* and *eulerTrans()* member functions for converting vectors (such as velocity and acceleration), and orientations (expressed as Euler angles) between two different Cartesian coordinate systems.

Heading, Pitch and Roll

Converting Euler angles from geocentric to topographic coordinates (or vice versa) is often particularly useful, since topographic-referenced Euler angles correspond to heading, pitch and roll.

Using a *DtCoordTransform* that can convert from geocentric to topographic coordinates, we can obtain an entity's heading, pitch, and roll as follows:

```
DtTaitBryan topoEuler;  
geocToTopo.eulerTrans(esr->orientation, &topoEuler);  
double heading = topoEuler.psi();  
double pitch = topoEuler.theta();  
double roll = topoEuler.phi();
```

7.3.4. UTM Coordinates

Locations in the world can be specified using a UTM coordinate system. UTM coordinates are mapped to a reference ellipsoid approximating the surface of the earth using a Universal Transverse Mercator projection. The coordinates consist of easting and northing from an origin or reference location, and an altitude above a reference ellipsoid, all in meters.

In a true UTM coordinate system, the origin or reference point (0,0,0) is often a point on the equator at the center of a particular UTM zone. (The world is longitudinally divided into 60 UTM zones, each centered on an odd multiple of 3 degrees). However, VR-Link allows you to define an “offset UTM” coordinate system by passing any arbitrary reference point, to the function `DtUtmInit()`.

`DtUtmInit()` must be called before using any of the UTM coordinate conversion functions. Currently, VR-Link supports the use of only one UTM reference point at a time.

Using `DtUtmInit()`

The first two arguments to `DtUtmInit()` are `DtDegMinSec` structures, representing the latitude and longitude of the reference point. `DtDegMinSec` has the following definition:

```
enum DtPolarDirection { DtEast, DtWest, DtNorth, DtSouth };  
typedef struct  
{  
    double deg, min, sec;  
    DtPolarDirection direction;  
} DtDegMinSec;
```

The following code initializes VR-Link's UTM conversion routines so that they use 35° north latitude and 122° west longitude as the reference point:

```
DtDegMinSec latRef = {35.0, 0.0, 0.0, DtNorth};  
DtDegMinSec lonRef = {122.0, 0.0, 0.0, DtWest};  
DtUtmInit(latRef, lonRef, 0);
```

The optional third argument to `DtUtmInit()` indicates that a particular convention should be used, whereby 500,000 meters are added to all easting values and 10,000,000 meters are added to all north values, in order to eliminate the use of negative numbers in UTM coordinates. Rare applications that need to use this convention should pass a value of 1 for this argument as opposed to the more common 0.

The optional fourth argument (not shown) controls the level of diagnostic information printed by the function.

A UTM coordinate is represented using `DtUtmCoord`, defined in [utmCoord.h](#). A `DtUtmCoord` can be constructed by passing its three components (in meters) to the constructor:

```
DtUtmCoord utmLoc(50.0, 100.0, 70.0);
```


Assuming that `DtUtmInit()` was called as above, this *DtUtmCoord* would represent a point of altitude 70 meters, that is 50 meters east and 100 meters north of our reference point of 35 degrees north by 122 degrees west.

The class has a copy constructor and assignment operator defined, and has the following inspectors and mutators to access the various components: `east`, `setEast`, `north`, `setNorth`, `up`, `setUp`.

DtUtmCoord also has member functions that enable you to convert to and from geocentric or geodetic coordinates. `DtUtmCoord::geocentric()` and `DtUtmCoord::geodetic()` return the geodetic and geocentric equivalents of a UTM coordinate respectively. For example:

```
DtUtmInit(...);  
...  
DtUtmCoord utm(50.0, 60.0, 70.0);  
DtGeodeticCoord geod = utm.geodetic();
```

The member functions `getGeocentric()` and `getGeodetic()` work similarly, but fill in an existing object passed by reference to the function, rather than returning the desired object.

Alternatively, the C-style functions `DtGeodToUtm()`, `DtUtmToGeod()`, `DtGeocToUtm()`, and `DtUtmToGeoc()` can also convert locations in UTM coordinates to geocentric or geodetic coordinates, and vice-versa. However their use is discouraged in favor of the member functions.

7.3.5. Considerations for Using UTM Coordinates

Trouble often arises when one fails to see the differences between a UTM coordinate system and a topographic coordinate system that share the same origin.

Although the coordinates of a point near the origin may be similar in the two systems, they are different in several ways.

- ♦ `Z` in a UTM system represents altitude above the earth's surface, while `-Z` in the topographic frame represent height above a plane tangent to the earth's surface, which deviates from altitude above the earth's surface itself as one moves away from the origin ([Figure 7-3](#)).

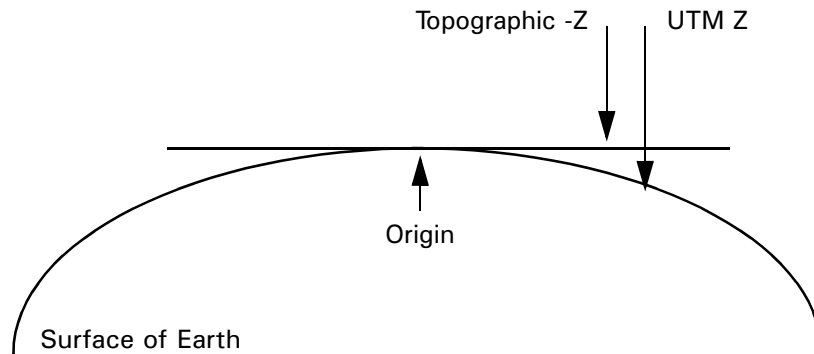


Figure 7-3. Comparison of UTM Z and topographic -Z

- ♦ Neglecting the small angle between northing and north, UTM coordinates expressed X, Y, and Z correspond to east, north, and up, while the topographic X, Y, and Z correspond to north, east, and down.

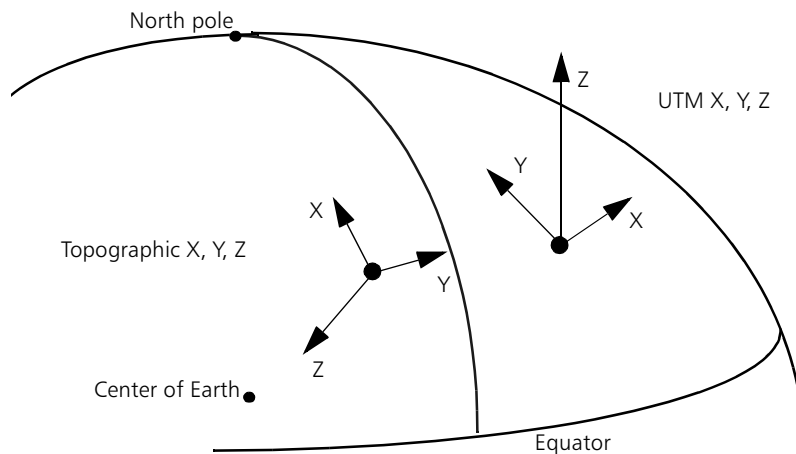


Figure 7-4. Topographic X, Y, and Z compared to UTM

- ♦ Although lines of constant easting in UTM coordinates run very close to north/south, they do not run truly north/south as lines of longitude do. This means that two points with equal easting values in their UTM coordinates, such as {100, 100, 0} and {200, 100, 0} do not lie exactly on the same line of longitude. The second point is not due north of the first. Depending on location within a UTM zone, the deviation can be plus or minus a couple of degrees. The function `DtUtmCoord::gridDeclination()` returns the magnitude (in radians) of the angle between northing and true north at a particular UTM location.

7.3.6. Lower Level Coordinate Conversion Functions

If you would like a greater level of control over your coordinate conversions than that offered by the *DtCoordTransform* class described in “[DtCoordTransform](#)” on page 7-10, there are several lower-level functions that can help.

If you have a rotation matrix that expresses a rotation between two coordinate systems, you can multiply that matrix by a vector, such as velocity, or acceleration using *DtDcmVecMul()* in order to convert that vector between the coordinate systems.

For geocentric/topographic conversions, the needed rotation matrix can be obtained using *DtLatLon_to_GeocToTopo()* or *DtLatLon_to_TopoToGeoc()*, declared in [topo-Coord.h](#). These functions take a latitude and longitude and return rotation matrices to convert between the geocentric frame and the topographic frame defined by that point on earth.

For example:

```
DtGeodeticCoord myOriginInGeod(35.0, -122.0, 0.0);

// Obtain a rotation matrix for the topo frame in question
DtDcm geoc2Topo;
DtLatLon_to_GeocToTopo(myOriginInGeod, geoc2Topo);

DtEntityStateRepository* esr = ...;
DtVector topoVel;

// Perform the vector rotation, using the matrix.
DtDcmVecMul(geoc2Topo, esr->velocity(), topoVel);
```

You can also use a rotation matrix to rotate a set of Euler angles from one coordinate system to another, using the function *DtEulerToEuler()*, declared in [Euler.h](#), for example:

```
DtTaitBryan topoOrient;
DtEulerToEuler(esr->orientation(), geoc2Topo, topoOrient);
```

But if your goal is to obtain an entity's orientation as topographic Euler angles (heading, pitch, and roll), a more efficient way would be:

1. Use *esr->bodyToGeoc()* rather than *orientation*.
2. Multiply this returned matrix by *geoc2Topo()*.
3. Convert the result to Euler angles using *DtBodyToRef_to_Euler()* as follows:

```
DtDcm bodyToTopo;
DtDcmDcmMul(geoc2Topo, esr->bodyToGeoc(), bodyToTopo);
DtTaitBryan topoEuler;
DtBodyToRef_to_Euler(bodyToTopo, topoEuler);
```

Coordinate locations cannot be transformed using only a rotation matrix. The translational offset between the two coordinate systems is also required. To transform from one cartesian system to another:

1. Subtract the origin of the second system from the point being transformed.
2. Perform the rotation as above.

For example, to transform a location from geocentric to a topographic coordinate:

```
// Find the local origin expressed in geocentric coordinates
DtVector myOriginInGeoc = myOriginInGeod.geocentric();
// Subtract the local origin from the point being transformed
DtVector tmp;
DtVecSub(esr->location(), myOriginInGeoc, tmp);
// Rotate the result
DtVector topoLoc;
DtDcmVecMul(geoc2Topo, tmp, topoLoc);
```

7.4. Managing Time

VR-Link's time-related functions are declared in *vlTime.h*.

Before any of VR-Link's time-related functions are used, it is necessary to call *DtTimeInit*. This applies to almost all applications, since *DtExerciseConn*, *DtReflectEntityList*, and *DtEntityPublisher* all use time-related functions.

The VR-Link toolkit maintains the concept of VR-Link simulation time for use in dead reckoning of remote entities and thresholding of local entities. Typically, VR-Link simulation time is set once during each iteration of the application's main simulation loop so that all entities are dead reckoned based on the same value of current time. VR-Link simulation time is set by calling *DtSetSimTime()*.

Usually, you will want to advance VR-Link simulation time each frame, in proportion to the amount of real time that has elapsed since the previous frame, so that it becomes a discrete approximation to (possibly scaled or offset) real time.

- The following functions may be of assistance in this task: *DtAbsRealTime()* provides a machine-independent interface to obtaining the absolute current time, in seconds, while *DtGetElapsedRealTime()* returns the time, in seconds, since *DtTimeInit()* was called.

In a fixed frame rate application, like VR-Link's *fl8* example, VR-Link simulation time can be managed as follows:

```
DtTimeInit();
DtTime dt = .05;           // Each time step is .05 seconds
DtTime simTime = 0.0;      // Represents current time
...
// Main simulation loop
while(...)
{
    DtSetSimTime(simTime);
    ...
    // Do stuff
    ...
    // Advance simTime by dt
    simTime = simTime + dt;
    // Sleep till the next multiple of .05 seconds
    DtTime timeTillNextFrame = simTime -
        DtGetElapsedRealTime();
    DtSleep(timeTillNextFrame);
}
```

In a floating frame rate application – where you go on to the next frame as soon as you're done with the previous one – you can pass the current time to `DtSetSimTime()` each frame:

```
DtTimeInit();
...
// Main simulation loop
while(...)
{
    DtSetSimTime(DtAbsRealTime());
    ...
    // Do stuff
    ...
}
```

7.5. Creating Linked Lists with *DtList*

DtList, defined in [vList.h](#), provides a general linked-list capability in VR-Link. To enable *DtList* to be used to represent a list of any type of object, pointers to the objects being listed are represented by `void*`, and must be explicitly cast back to the data type you are listing when you want to inspect data in the list.

Within VR-Link, a *DtList* is used to represent the list of articulated parts that an entity possesses, the list of HLA objects in the exercise, and other such lists. For these reasons, you may need to work with *DtLists* even if you do not choose to use them to represent your own lists.

A *DtList* is constructed as follows:

```
DtList list;
```

Elements can be added to a *DtList* using:

- ♦ `addToStart()`
- ♦ `addToEnd()`
- ♦ `addBefore()`
- ♦ `addAfter()`
- ♦ `add()` – which is equivalent to `addToEnd()`

These functions return a pointer to a *DtListItem* – an envelope in which your data gets wrapped before insertion into the list.

These functions take a pointer to the element to be added. (As an exception, we sometimes store simple data types that are ≤ 32 bits directly in the list rather than by using pointers.) These pointers must remain valid for the duration of the time they are in the list. For example, the address of automatic variables should not be put in a list if the list remains beyond the scope of the automatic variables. As a result, in general, the pointers in a list should typically be from new'd data.

DtList's `first()` and `last()` functions return pointers to *DtListItems* that hold the first and last elements of the list.

DtListItem's `data()` member function can be used to get back the pointer you added to the list.

`first()` and `last()` return `NULL` when the list is empty.

DtListItem has `prev()` and `next()` member functions that facilitate iterating through a list. These functions return `NULL` when the beginning and end of the list have been reached, respectively. Thus, you can iterate through the elements in a list as follows:

```
DtList list;
...
for (DtListItem* item = list.first(); item; item = item->next());
```

The following example adds several items to a list (in this case instances of class *A*), and pulls them out of the list later:

```
// Definition of class A
class A
{
public:
    A(int num) { a = num; }
    int a;
};

// Create the list
DtList list;

// Create three instances of class A
A* a1 = new A(1);
A* a2 = new A(2);
A* a3 = new A(3);

// Add the instances to the list
list.add(a1);
list.add(a2);
list.add(a3);

// Iterate through the list, inspect the data
for (DtListItem* item = list.first(); item; item = item->next())
{
    // Use DtListItem::data to get back the pointer you added.
    // You'll need to cast the pointer back to an A* from a void*
    A* current = (A*) item->data();
    printf("%d\n", current->a);
}
```


7.6. Diagnostic Utilities

The amount of diagnostic information printed by VR-Link can be controlled by setting the global variable `DtNotifyLevel`, declared in [vlprint.h](#). `DtNotifyLevel` may be set to any of the following values:

<code>DtNIFatal</code>	only fatal messages are printed.
<code>DtNIWarn</code>	warning messages are printed as well.
<code>DtNIInfo</code>	some diagnostic information is printed.
<code>DtNIVerbose</code>	maximum information is printed.

The default is `DtNIInfo`.

You can use this notify level to control messages generated by your own application as well, if you use VR-Link's printing functions rather than `printf()` or `cout()` directly. These printing functions are also declared in [vlprint.h](#), and have the same prototype as `printf()`. They include:

- ♦ `DtFatalPerror()` and `DtWarnPerror()`, which print perror information in addition to the information you pass to the function
- ♦ `DtFatalError()`
- ♦ `DtWarn()`
- ♦ `DtInfo()`

The fatal functions call `DtAbort` after printing their messages.

7.7. IP Address Manipulation Functions

In [vlutil.h](#), you can find a set of functions for obtaining and manipulating IP addresses. These functions are often useful in generating addresses to use in initializing a DIS exercise connection.

VR-Link represents IP addresses with the *DtInetAddr* type, which is equivalent to an unsigned 32 bit integer.

`DtStringToInetAddr()` constructs a *DtInetAddr* from a string containing the common dot notation for an IP address. For example:

```
DtInetAddr myAddr = DtStringToInetAddr("207.86.232.1");
```

`DtInetAddrString()` goes the other way, returning a string representation of a *DtInetAddr*.

`DtInetAddrOfDevice()` returns the IP address of a particular device, specified by its device name. Device names are typically something like `le0` or `ec0`. Similarly, `DtNetMaskOfDevice()` returns the netmask of a device.

`DtInetBroadcastOfDevice()` computes and returns the broadcast address of a device, by using those portions of the device's IP address that are covered by the netmask, and setting the lower order bits to 1's. For example, if the IP address of a device is 207.86.232.1, and the netmask is 0xfffff00 (class C network), then the broadcast address will be 207.86.232.255.

Note: On some UNIX platforms, this does not match the erroneous broadcast address reported by `ifconfig`.

`DtFirstHostInetAddr()` returns the IP address of the host's primary network device.

`DtIsMulticastAddr()` returns 1 if the address passed to it is a valid multicast address, that is, one between 224.0.0.0 and 239.255.255.255.

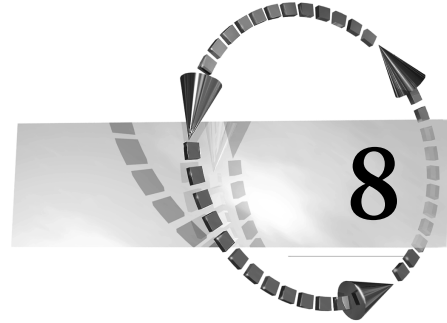
7.8. Miscellaneous Functions

DtSleep(), declared in [ProcControl.h](#), provides a machine-independent way to halt program execution for some period of time. Its argument is the time, in seconds, for which to “sleep”. The implementation of DtSleep() is machine-dependent, but it uses system calls such as UNIX's select, rather than a CPU-intensive busy loop.

DtSelect(), declared in [vlutil.h](#), provides a machine-independent interface to the UNIX select system call, which takes different arguments on different platforms. DtSelect()'s arguments are a number of file descriptors, three arrays of file descriptors to monitor for presence of data to read, availability for writing, and pending exceptional conditions, respectively, and a timeout in seconds. See [vlutil.h](#) for more information.

DtAbort(), declared in [vlprint.h](#), is called whenever VR-Link generates a fatal error. In turn, it makes the abort system call. If you need to modify VR-Link's abort behavior, you can write your own definition for DtAbort(). Because VR-Link's definition for DtAbort() resides in its own object file, your definition will merely supersede ours with no other effects.

If your definition is linked into your application through a library that you create (rather than through an object file on your link line), make sure that your library appears before *libvlutil.a* on your link line.



Example and Utility Applications

VR-Link includes a set of supporting applications that can help you develop and troubleshoot your own applications.

The following example programs are included with VR-Link, with source code:

- ♦ [The fl8 Program, on page 8-3](#), a simple VR-Link simulator.
- ♦ [netdump \(DIS Only\), on page 8-6](#) and [hlaNetdump \(HLA only\), on page 8-8](#), which display DIS PDUs or HLA attribute updates and interactions in an easy-to-read form.
- ♦ [nethex \(DIS only\), on page 8-9](#), which displays DIS PDUs in a hexadecimal format.
- ♦ [talk and listen, on page 8-9](#), simple send-only and receive-only programs.
- ♦ [vdc – the VR-Link Daemon Controller, on page 8-10](#), a program that controls the Packet Server daemon.

In addition, the following executables are included without source code:

- ♦ The Packet Server, which improves network access for DIS applications, and enables communication among multiple DIS applications on the same machine ([The VR-Link Packet Server \(DIS and UNIX Only\), on page 8-12](#)).
- ♦ *buffgrep*, which searches the standard input for buffers containing a specified pattern ([buffgrep, on page 8-21](#)). This program is typically used with *netdump* or *hlaNetdump*.

8.1. About the Applications

VR-Link's example and utility applications are provided to make your development and troubleshooting tasks easier, and to demonstrate how VR-Link's classes and functions are used in practice.

The executables for these programs are in the three *.bin* directories described in section 2.2.3, “VR-Link Files Installed”. You can find source code for the example applications in subdirectories under *.examples*.

The *buffgrep* program and *vrlinkd3i* packet server are provided only as utilities, and do not include source code. The *vrlinkdu* packet server is available as a separate option.

8.1.1. License Enforcement for the Example Applications

The prebuilt example applications in the *.bin* directories do not cause a VR-Link license to be checked out, so you can run them freely. However, if you rebuild them, be careful not to overwrite the original versions. Your new versions will require VR-Link licenses in order to run.

8.2. The f18 Program

The *f18* program simulates a simple HLA or DIS vehicle. By default, that vehicle is an FA18, flying in a circle, just off the coast of Fort Hunter-Liggett, but you can control the entity type, location and route.

f18 generates reliable HLA entity update messages (or DIS entity state PDUs), so it is a useful tool for debugging applications that receive and interpret these messages. *f18* can also fire munitions and react to detonations.

The *f18* syntax is:

```
f18 -h
f18 [options...]
```

> To exit *f18*, press “q” followed by Enter or Return.

Table 8-1 summarizes *f18*’s command line options.

Table 8-1: f18 Startup Options (Sheet 1 of 2)

Option	Description
Protocol Independent	
-d <i>secs</i>	Set the delay (in seconds) between the time <i>f18</i> receives a detonation interaction (or PDU), and the time it broadcasts its final message and exits.
-D <i>algo</i>	Set dead reckoning algorithm to <i>algo</i> , where <i>algo</i> is one of: 0 - Other 1 - Static 2 - Fixed rotation, positional DR, world coordinates 3 - Rotational DR, positional DR, world coordinates 4 - Rotational DR, velocital DR, world coordinates 5 - Fixed rotation, velocital DR, world coordinates
-h	Displays a summary of command-line options, then exits.
-H <i>deg</i>	Sets the initial heading of the f18 to <i>deg</i> degrees.
-l <i>file</i>	Loads the specified <i>.mtl</i> (MÄK Technologies Lisp) configuration <i>file</i> .
-L <i>n, e, u</i>	Set initial position, as north,east,up in topographic coordinates.
-O <i>lat, long</i>	Sets the reference latitude and longitude for the topographic coordinate system.
-r <i>radius</i>	Sets turn <i>radius</i> of the f18 to the specified number of meters. Setting the <i>radius</i> to 0.0 will cause the f18 to move in a straight line.
-s <i>mps</i>	Sets initial speed of the f18 to <i>mps</i> meters/second.
-T <i>type</i>	Sets the entity type, specified as a string of the form: kind:domain:country:category:subcategory:specific:extra
-W <i>x, y, z</i>	Sets the initial position in geocentric coordinates.

Table 8-1: *f18* Startup Options (Sheet 2 of 2)

Option	Description
-x <i>ex-name</i>	Sets the exercise ID for IS (default is 1), or the federation execution name for HLA (default is VR-Link).
DIS Only	
-a <i>ID</i>	Set the second component of the entity ID to <i>ID</i> . By default, the entity ID is 1:2:1, however if one desires to run multiple <i>f18</i> 's at once, each needs a unique entity ID.
-A <i>addr</i>	Sets the default destination IP address for outgoing PDUs to <i>addr</i> . <i>addr</i> is of the form 123.123.123.123.
-P <i>portnum</i>	In DIS versions, specifies the UDP port. Defaults to 3000.
-S <i>addr</i>	In DIS versions, subscribes to the multicast address specified by <i>addr</i> . Multiple -S's can appear on a command line. Multicast is not supported on all platforms. <i>addr</i> is of the form 123.123.123.123.
-V <i>ver</i>	Sets the DIS protocol version contained in outgoing PDUs. Replace <i>n</i> with an integer as follows: 3 - DIS 2.0.3 5 - IEEE 1278.1 4 - DIS 2.0.4 6 - DIS 2.1.4

8.2.1. The *f18* Configuration File

By default, *f18* loads the configuration file *f18.mtl*, if it exists in the current directory. This file sets many of the *f18*'s default parameters, including notify level, dead-reckoning algorithm, and initial heading, position, and speed.

To change default parameters:

- ♦ Edit the *f18.mtl*, file
- ♦ Use command-line options to override parameters
- ♦ Use the -l option to specify an alternate configuration file.

You can rebuild *f18*, or create a new application based on *f18*, that does not use a configuration file.

- > To write a variation of *f18* that does not use a configuration file, remove the `initMtl()` call.

You can also specify a different configuration file in your code, at the `DtcLoadLispFile()` function call.

8.2.2. Firing Munitions

- > To cause the F18 to fire at another entity, press the Enter or Return key.

If other entities exist in the exercise, *f18* issues a fire message directed at the closest one. Though there is no tracked munition, *f18* issues a detonate message three seconds later to indicate that the target was hit.

- > To change the delay, edit the `munitionFlightTime` parameter in the configuration file.

8.2.3. Reactions to Detonate Messages

The F18 reacts to detonation PDUs or RPR FOM Detonation Interactions. If a detonation occurs within the `lethalDetonationRange`, or if the detonation result is entity impact or entity proximate, the F18 is destroyed. Updates reflect a damage state of `DamageDestroyed` for a period of `destroyedToFinalDelay` seconds, at which point the entity leaves the exercise and *f18* exits.

`LethalDetonationRange` defaults to 20 meters, and `destroyedToFinalDelay` defaults to zero. You can change these parameters in *f18.mtl*.

8.2.4. Absolute timestamping (DIS only)

The value of the `timeStampType` flag in *f18.mtl* determines whether absolute or relative timestamps are used in *f18*'s outgoing PDUs. The flag defaults to relative (0), but may instead be set to absolute (1). When the `timeStampType` is absolute, *f18* will also dead reckon remote entities based on their absolute timestamps, when applicable. Absolute timestamping should only be used when the local machine's clock is synchronized with the clock being used by other applications in the exercise for their absolute timestamps.

8.3. netdump (DIS Only)

netdump is a DIS debugging tool that displays the contents of arriving DIS PDUs in an easy-to-read form. (Its HLA counterpart, *hlaNetdump*, is described on page 8-8.)

Netdump writes its output to stdout. For each PDU, netdump prints the number of PDUs it has received, and the PDU's arrival time (relative to *netdump*'s initialization).

You can filter PDUs based on specific fields by piping *netdump*'s output through VR-Link's *buffgrep* utility.

netdump's usage is:

```
netdump -h
netdump [-c] [-e 0|1] [-P port] [-S address]
```

Table 8-2 describes the command-line options.

Table 8-2: netdump Startup Options

Option	Description
-c	Clears the screen between PDUs. This option is especially useful for debugging single-entity applications. The screen is cleared between PDUs, so corresponding fields of successive PDUs appear in the same place. This makes changes in various parameters easily visible.
-e <i>level</i>	Sets level of aggressiveness in trying to print erroneous PDUs. If <i>level</i> is 0, <i>netdump</i> always tries to print each PDU. If <i>level</i> is 1, <i>netdump</i> doesn't try to print PDUs when the size indicated in the DIS header doesn't match the size of the packet received.
-h	Displays a summary of command-line options, then exits.
-P <i>portnum</i>	Specifies the UDP port. Defaults to 3000.
-S <i>addr</i>	Subscribes to the multicast address specified by <i>addr</i> . Multiple -S's can appear on a command line. Multicast is not supported on all platforms. <i>addr</i> is of the form 123.123.123.123.

Notes:

- ♦ If *netdump* has been linked with the *libv1* in the *.bin5* directory, it prints all fields of DIS 2.0.4, IEEE 1278.1 or DIS 2.1.4 PDUs.
- ♦ If it has been linked with the *libv3* in the *.bin3* directory, it prints fields from DIS 2.0.3 PDUs.
- ♦ When it receives PDUs of the wrong DIS version, *netdump* prints the DIS header information and a hex dump of the rest of the PDU.
- ♦ *netdump* prints version information as described in [Table 8-3](#):

Table 8-3: PDU versions and DIS protocol versions

PDU Version	DIS Protocol Version
3	2.0.3
4	2.0.4
5	IEEE 1278.1
6	2.1.4

8.4. *hlaNetdump* (HLA only)

hlaNetdump listens to an HLA federation execution and prints out data whenever it receives an attribute update or interaction.

hlaNetdump handles interactions similarly to *netdump* (page 8-6) — it interprets and prints out the data in the interaction.

When *hlaNetdump* receives a message, it is decoded into a *DtEntityStateRepository*, then the state repository is printed. *hlaNetdump* prints default values (usually zero or false) for attributes not contained in the update. A list appears above the state information, showing the names of attributes actually contained in the update.

The usage for *hlaNetdump* is:

```
hlaNetdump -h
hlaNetdump [-c] [-r] [-x exec-name]
```

Table 8-4 describes the command-line options.

Table 8-4: *hlaNetdump* Startup Options

Option	Description
-c	Clears the screen between messages (updates or interactions). This option is especially useful for debugging single entity applications. The screen is cleared between messages, so corresponding fields of successive messages appear in the same place. This makes changes in various parameters easily visible.
-h	Displays a summary of command-line options, then exits.
-r	Run <i>hlaNetdump</i> in raw mode.
-x <i>exec-name</i>	Specifies the name of the federation execution to be monitored. Default is <i>VR-Link</i> .

When you run *hlaNetdump* with the -r option, it behaves like *nethex*. For each attribute update or interaction received, it prints the name of the attribute, the size of the value of that attribute, and a hexadecimal dump of the value of the attribute.

8.5. *nethex* (DIS only)

nethex is a DIS debugging tool that prints out arriving DIS PDUs in raw, hexadecimal format. Using *nethex*, you can see exactly the information contained in each byte of a PDU. *nethex* writes its output to stdout.

nethex's usage is:

```
nethex -h
nethex [-P portnum] [-S addr]
```

[Table 8-5](#) describes the command-line options.

Table 8-5: *nethex* Startup Options

Option	Description
-h	Displays a summary of command-line options, then executes.
-P <i>portnum</i>	Specifies the UDP port. Defaults to 3000.
-S <i>addr</i>	Subscribes to the multicast address specified by <i>addr</i> . Multiple -S's can appear on a command line. Multicast is not supported on all platforms. <i>addr</i> is of the form 123.123.123.123.

8.6. *talk and listen*

talk and *listen* are simple send-only and receive-only programs. They demonstrate many VR-Link fundamentals.

talk simulates the flight of an F18 aircraft. *listen* repeatedly prints an entity's updated position, and if a fire PDU or interaction occurs, prints the entity ID of the attacker. Both programs are written for protocol-independence.

For detailed descriptions, see [“A Listen-Only Example” on page 3-4](#) and [“A Send-Only Example” on page 3-7](#).

8.7. *vdc* – the VR-Link Daemon Controller

vdc, the **vrlink-daemon controller**, lets you send messages that control the Packet Server (described in “[The VR-Link Packet Server \(DIS and UNIX Only\)](#)” on page 8-12). Its source code is provided as a working example of how a VR-Link application can control the Packet Server at runtime.

vdc controls the Packet Server whether it is running in the background (as a daemon) or in the foreground. However, *vdc*’s terminate command is ignored if the Packet Server is running in the foreground.

To issue a *vdc* command, enter the following:

```
vdc port [cmd]
```

where *port* is the required UDP port number and *cmd* is a one-letter command from Table 8-6 below. If you omit the command, *vdc* sends an *a* command (test if alive).

Table 8-6: *vdc* Control Keys

Key	Purpose
e	Set Packet Server to external mode.
i	Set Packet Server to internal mode.
s	Print server status.
c	Print server client status.
p	Cause server to ping clients.
q	Terminate Packet Server.
u	Turn unbundling on.
n	Turn unbundling off.
a	Test if alive.
t	Quietly test if alive.

Note: All commands except *a* and *t* have the same meaning as those used by the Packet Server itself.

The *a* and *t* commands are unique to *vdc*. You can use them to perform a conditional action depending on whether the Packet Server is running. See “[vdc Examples](#)” on [page 8-11](#) for examples.

When you control the Packet Server through *vdc*, status messages and diagnostics are displayed by *vdc*, rather than by the Packet Server. A terminate request sent by *vdc* is ignored if the Packet Server is running in foreground.

8.7.1. *vdc* Examples

vdc accepts multiple commands in a string, as in these examples:

<code>vdc 3000 is</code>	Change to internal mode and then display server status.
<code>vdc 3000 pc</code>	Ping clients and then display client status.

You can use the test commands *a* and *t* to perform a different action depending on whether the Packet Server is running:

<code>vdc 3000 tq</code>	If the server does not exist, exit <i>vdc</i> quietly; otherwise terminate the server.
<code>vdc 3000 a vrlinkd3i 3000</code>	Start server if not already running.
<code>vdc 3000 t && echo hi echo bye</code>	Display “hi” if the server is running; otherwise display “bye”.

8.7.2. *vdc* Diagnostics

If you receive the message

```
no vrlink server on port nnnn
```

and you know that the Packet Server is running on the specified port, the server's AF-UNIX socket file may have become unlinked. See [“The Server Socket File” on page 8-20](#) for more information.

8.8. The VR-Link Packet Server (DIS and UNIX Only)

The Packet Server, also called the *vrlink* daemon or *vrlinkd*, distributes DIS packets between a network connection and multiple VR-Link applications running on the same host. It offers several benefits to DIS applications running on UNIX systems.

Note: The Packet Server applies to DIS applications only, since in HLA, the RTI handles low-level communications.

8.8.1. Packet Server Modes

The Packet Server operates in two modes, external and internal.

In external mode, the Packet Server serves as an intermediary between its client DIS applications and the UDP socket, allowing the applications to communicate with the network through the same UDP port. It asynchronously reads packets from a specified UDP port and forwards the packets to client applications through a shared memory interface. It provides client-specific filtering based on exercise ID, DIS protocol version, and PDU kind.

In internal mode, the Packet Server allows client applications on a single machine to exchange packets among themselves, without interacting with the network.

The server can switch between internal and external mode, bringing its clients into and out of an exercise.

8.8.2. Packet Server Configurations

The Packet Server is available in two configurations, distinguished by the number of clients they support:

- ♦ *vrlinkd3i* supports three clients in internal mode, or one client in external mode. This version is included as a standard part of the VR-Link toolkit.
- ♦ *vrlinkdu* supports an unlimited number of clients. It is available as a separate product.

You can start multiple Packet Servers to service more than one UDP port on a host.

8.8.3. How Applications Connect to the Packet Server

A MÄK VR-Link application (or any VR-Link application that uses the default exercise connection) automatically uses the Packet Server if one is running on its port. Otherwise, the application connects directly to the network.

If a Packet Server is running, but is already handling its maximum number of clients, further clients will terminate when they attempt to connect to the Packet Server.

8.8.4. Using the Packet Server

Some UNIX platforms allow only a single application to listen to a particular port. On these platforms the Packet Server lets you run multiple VR-Link-based applications on a single machine. However, even on platforms where the Packet Server is not strictly necessary, you may want to use it anyway to take advantage of the additional functionality it provides. (For more information on platforms that can run multiple VR-Link applications, see the Packet Server discussion in the *RELNOTES* file.)

Note: You need a VR-Link license for each VR-Link application that you run concurrently, regardless of platform.

Using the Packet Server with Only One Client Application

If only one DIS application based on VR-Link is running per machine, the applications can communicate directly with the network and no Packet Server is necessary. However, the Packet Server can be beneficial even when there is only one client.

Because the asynchronous server can respond more quickly than a fixed-rate client to network traffic, the Packet Server may reduce the number of dropped packets. Moreover, if the client cannot drain packets from the Packet Server quickly enough, the Packet Server ensures that the oldest packets are dropped rather than the newest. The opposite would be true if the client were reading directly from the network. In addition, the Packet Server can enable a higher packet throughput when it can run on a separate processor.

When only one client needs to connect to the Packet Server, you can use *vrlinkd3i*, which comes standard with VR-Link.

Using the Packet Server with Multiple Clients per Host

On many UNIX platforms, you need a Packet Server to run more than one VR-Link DIS application on a single machine. You can use the *vrlinkd3i* version if you are running two or three applications and the applications communicate only among themselves. Otherwise, you need the *vrlinkdu* version.

Figure 8-7 depicts some common configurations of applications using a Packet Server. In scenario A, each machine hosts one application, so no Packet Server is necessary. However, Machine #3 uses one anyway for improved packet handling. In scenario B, one machine hosts two applications. On some platforms, the unlimited *vrlinkdu* version is required in such a case to connect multiple applications to an external network. In scenario C, the Packet Server is used in internal mode to connect three applications on the same machine.

8.8.5. Starting the Packet Server

- > To start the Packet Server, run *vrlinkd3i* or *vrlinkdu* with the number specifying the UDP port to be serviced.

The example below starts a Packet Server on port 3000, which is the default port for VR-Link applications:

```
vrlinkd3i 3000
```

By default, the Packet Server starts in external mode.

- > To run the Packet Server in internal mode, start it with the **-I** option.
- > To run the Packet Server in the background after initialization (as a daemon), start it with the **-B** option.

For example, the following command starts an internal Packet Server in the background on UDP port 3001.

```
vrlinkd3i -I -B 3001
```

Startup options appear before the port number on the command line. The options in [Table 8-8](#) apply to both Packet Server configurations.

Table 8-8: Packet Server Startup Options (Sheet 1 of 2)

Option	Description
portnum	The UDP port number is the Packet Server's only required argument.
-A <i>addr</i>	Specifies where to send packets which are received from clients created with VR-Link versions prior to 2.4.2. <i>addr</i> is an IP address, and may be a broadcast, point-to-point or multicast address. This option does not apply to applications created with newer VR-Link versions, which choose their own destination address. Do not use this option with -D.
-B	Tells the Packet Server to execute in the background (as a daemon), and sets the notify level (option -N) to 2.
-D <i>device</i>	Tells the Packet Server to listen to the network on which <i>device</i> listens. <i>device</i> is the name of a network device, (e.g. <i>ec0</i> or <i>le0</i>). Use <code>netstat -i</code> to list available devices. The destination address for packets sent by pre-2.4.2 clients is set to the broadcast address of <i>device</i> . The -D and -A should not be used together.
-I	Starts the Packet Server in internal mode, ignoring the network.
-n	Tells the Packet Server to drop new packets rather than old ones when the queue is full and the drop mode is not specified by the client.

Table 8-8: Packet Server Startup Options (Sheet 2 of 2)

Option	Description
-N <i>level</i>	Sets notify level to one of the following: 0 FATAL 2 NOTIFY 1 WARNINGS 3 INFO (default)
-r <i>size</i>	Sets the size of the queue the Packet Server will use to receive packets from pre-2.4.2 clients for clients that do not specify a size. The default is 20 kB.
-s <i>size</i>	sets the size of the queue the Packet Server will use to send packets to clients for clients that do not specify a size. The default is 200 kB.
-u <i>state</i>	Turns unbundling on or off depending on whether <i>state</i> is 1 or 0 respectively. Unbundling is on by default.
-w	Tells the Packet Server, when dropping old packets, not to wait for the queue lock when the client has the lock, and to drop the packet instead. The Packet Server may run slightly faster in this mode while dropping relatively few new packets.

8.8.6. Stopping the Packet Server

- > If the Packet Server is running in the foreground, stop it by typing ‘q’, then Enter, or Ctrl-C.
- > If the Packet Server is running in the background, stop it using the VR-Link daemon controller:

```
vdc server-port q
```

The Packet Server also can terminate in response to a SIGTERM or SIGINT signal, or a terminate message from another process.

When terminating, the Packet Server first notifies each client that it is closing the connection to the client. Typically, the next read or write by a client after such notification causes a fatal error in the client.

The Packet Server also closes connections if you change its mode and it is serving too many clients for the new mode. For example, if you connect three clients to *vrlinkd3i* running in internal mode, then to switch to external mode, the connections to all but the first client are closed.

8.8.7. Packet Server Controls

When the Packet Server is running in the foreground, you can control it using its keyboard controls. When it is running in the background, you can control it through messages from another process such as the *vdc* program (see “[vdc – the VR-Link Daemon Controller](#)” on page 8-10) or your own application.

The keyboard controls consist of the single-character commands summarized in [Table 8-9](#). Each command is followed by the Enter key.

Table 8-9: Packet Server Control Keys

Key	Purpose
e	Set Packet Server to external mode.
i	Set Packet Server to internal mode.
s	Print server status.
c	Print server client status.
p	Cause server to ping clients.
q	Terminate Packet Server.
u	Turn unbundling on.
n	Turn unbundling off.

When you run the Packet Server in the background, the *vdc* daemon controller program uses these same single character commands and some additional ones. See [“Background Operation” on page 8-19](#) for more information.

8.8.8. Pinging Clients

> To cause the Packet Server to ping its clients, press the “p” key.

It determines whether the clients are still alive, and then sends them a control message. Client applications then typically display a reception message, thus indicating that packet-server connections are intact.

If a client is no longer alive, the Packet Server removes it from its list of clients.

8.8.9. Unbundling Packets

The Packet Server can extract individual PDUs that are concatenated into the same IP packet. The PDUs are served to clients one at a time as if each had arrived in its own packet. Unbundling is on by default.

> To turn off packet bundling, use the `-u 0` startup option, or press the “n” key.

8.8.10. Status Commands

The server status command (the “s” key) displays the following information:

- ♦ The Packet Server’s internal/external mode status
- ♦ The UDP port number
- ♦ The bundling mode
- ♦ The number of packet clients and control-only clients
- ♦ The maximum number of clients allowed for the current mode

The client status command (the c key) displays the server status followed by this information for each client:

- ♦ Whether the client is a packet client or control client
- ♦ The client’s ID (used for IPC keys) and process ID
- ♦ The number of messages sent and received
- ♦ Old messages and new messages dropped
- ♦ Size of the send and receive queues
- ♦ The list of mode flags currently in effect (Read, Write, Notify, and so on)

8.8.11. Queue Sizes

Packets are dropped when a client does not drain the buffers often enough for a given queue size. You may be able to compensate for this problem by increasing the size of the Packet Server's send queue.

Note: Shared memory is always allocated in full page sizes, typically 4kBytes.

The Packet Server allocates a single shared memory segment equal to the combined size of the send and receive queues. Therefore, for the most efficient use of system resources, specify send and receive queue sizes that add up to a multiple of 4kB.

> To set the send and receive queue sizes, use the `-S` and `-r` startup options, respectively.

8.8.12. Technical Notes

This section contains additional technical information.

External-Mode Operation with a Single Client

vrlinkd3i supports one client in external mode. Using the packet server in this way moves filtering into a separate process, which is useful on multiprocessing machines. It also allows the application to vary the effective amount of network packet buffering, which can be especially helpful to applications that do not service the network often. Finally, it allows the network to be turned on and off from the application or from an external controller such as *vdc*.

Background Operation

Using the `-B` option to run the Packet Server as a daemon is similar to executing a shell background job, but differs in the following respects:

- ♦ Control is returned to caller only after port is tested for availability
- ♦ Termination by the control client is enabled
- ♦ Diagnostics are sent to syslog
- ♦ The notify level is set to ignore Info.

The server behaves as a polite daemon (own process group, no control terminal, and cwd set to */tmp*).

The Application's Interface to the Packet Server

Client applications use the *DtClientSocket* packet interface to send and receive packets through the server. The *DtClientSocket* is a *DtSocket* that sends and receives packets with the help of the Packet Server. For details, see [Socket.h](#) on page B-703.

Constraints Imposed by System Resources

While *vrlinkdu* has no inherent limit to the number of clients it can support, it may be limited by the availability of System V IPC resources. Each client uses a separate semaphore id. Some UNIX systems are configured to allow only ten. You can modify this by reconfiguring the kernel. For large numbers of clients, you may also need to increase the total system-wide number of semaphores (which is different from the limit on semaphore ids) and the number of shared memory segments.

The allowable maximum queue size is also constrained by the system limit on the size of shared memory segments.

The Server Socket File

The server accepts control requests (for example, add new client) on an AF-UNIX socket file in */tmp* (for example, */tmp/vld-3000*). If this file is removed, then no further control requests will be accepted. In fact, control requests will return with the error:

```
no vrlink server on port nnnn
```

Without its socket file, the server will continue to function with existing clients, but will not be able to add new clients or change communication mode.

- > To restore these capabilities, restart the server and reestablish connections for the clients.
- > To terminate a background server that has lost its socket file, use `kill -15`.

8.9. *buffgrep*

The *buffgrep* utility searches the standard input for buffers matching one or more patterns. It is typically used with *netdump* (“*netdump (DIS Only)*” on page 8-6) or *hlaNetdump* (“*hlaNetdump (HLA only)*” on page 8-8).

A buffer is defined as all the lines from the end of the last buffer (or the beginning of the stream) up to and including the first line that matches a specified ending pattern. Buffers that contain any of the search patterns specified are copied to the standard output. *buffgrep* patterns are regular expressions like those used by *grep* and *ed*.

The *buffgrep* syntax is:

```
buffgrep [ -v] endpattern pat1 [pat2...]
```

The *-v* option causes *buffgrep* to print a buffer only if no line contains any of the specified patterns *pat1*, *pat2*, and so on.

buffgrep is often useful in conjunction with *netdump* (or *hlaNetdump*) and *awk*. When using *buffgrep* in this manner, use the following sequence as the *endpattern*:

```
'^$'
```

8.9.1. *buffgrep* Examples

Print out all DIS EntityState PDUs:

```
netdump | buffgrep '^$' EntityState
```

Print out all HLA updates or PDUs from entity 1:8:1 or 1:9:X:

```
hlaNetdump | buffgrep '^$' 'From.*1:8:1' 'From.*1:9:*'
netdump | buffgrep '^$' 'From.*1:8:1' 'From.*1:9:*'
```

Print out all HLA fire interactions:

```
hlaNetdump | buffgrep '^$' FireInteraction
```

Print out all EntityState PDUs from entity 1:8:1:

```
netdump | buffgrep '^$' EntityState | buffgrep '^$'
'From.*1:8:1'
```

Print out all EntityState PDUs except those from entity 1:8:1:

```
netdump | buffgrep '^$' EntityState | buffgrep -v '^$'
'From.*1:8:1'
```

Print out only the PDU banner, TimeStamp, and Velocity for Collision PDUs:

```
netdump | buffgrep '^$' Collision | awk '/PDU/ /TimeStamp/
/Velocity/'
```

Print out the TimeStamp changes in PDUs from entity 1:8:1:

```
netdump | buffgrep '^$' 'From.*1:8:1' |  
    awk '/PduKind/ /TimeStamp/ {dt=$2-t; t=$2; print $0, "  
", dt}'
```



Guide to Classes and Header Files

This chapter covers the following topics:

- ♦ [Convenience Header Files, on page 9-2](#)
- ♦ [Header Files for Protocol Independence, on page 9-2](#)
- ♦ Quick-reference tables to help navigate the header files:
 - [Class-to-Header Cross Reference, on page 9-4](#)
 - [Header-to-Class Cross Reference, on page 9-16](#)
 - [Other Header Files, on page 9-28](#)
- ♦ [Class Hierarchy Diagrams and Tables, on page 9-29](#)

In order to keep VR-Link's printed documentation manageable, the header files themselves are in an appendix that appears only in the on-line PDF version of the manual. In that format, you can use Acrobat's search and bookmark features to find information quickly. In addition, the on-line version of this chapter's cross-reference tables contain hypertext links that give you easy access to all header files and classes in the appendix.

9.1. Convenience Header Files

Some VR-Link header files exist simply for convenience or to provide protocol independence, and do not require further description in this reference.

The following four files each include the other header files for their respective libraries:

- ♦ *matrixInc.h*
- ♦ *vlInc.h*
- ♦ *mtlInc.h*
- ♦ *vlutilInc.h*

For convenience, you can include just these general header files in your code. However, you might speed up compilation by including only the header files you need.

The following files group other related header files together for convenience:

- ♦ *mathTypes.h* groups math-related headers, defines math macros
- ♦ *pdus.h* groups all PDU headers

9.1.1. Header Files for Protocol Independence

Table 9-1 lists header files that are used for protocol-independence.

Table 9-1: Header files for protocol independence

This file:	provides protocol independence for:
<i>ackInter.h</i>	Acknowledge interactions.
<i>actReqInter.h</i>	Acton request interactions
<i>aggPub.h</i>	Local aggregates
<i>collideInter.h</i>	Collision interactions
<i>detonateInter.h</i>	Detonation interactions
<i>entityPub.h</i>	Local entities
<i>exerciseConn.h</i>	Exercise connections

Table 9-1: Header files for protocol independence

This file:	provides protocol independence for:
fireInter.h	Fire interactions
interaction.h	Base interactions
lclInter.h	Logger control interactions
reflectedAgg.h	Reflected aggregates
reflectedEnt.h	Reflected entities
reflAggList.h	Reflected aggregate lists
reflEntList.h	Reflected entity lists
signalInter.h	Signal interactions
startInter.h	Start and resume interactions
stateMsg.h	State messages
stopInter.h	Stop and freeze interactions
vcInter.h	View control interactions

9.2. Class-to-Header Cross Reference

Table 9-2 contains an alphabetical list of VR-Link classes along with the header file that contains each one. In the on-line version of this manual, the header names are hyper-text links to the corresponding pages in Appendix B.

Table 9-2: Classes and their Corresponding Header Files

Class	Header File
DtAbsNotifyObserver	absNotifyObs.h
DtAbsoluteVCPdu	vcPdu.h
DtAcknowledgeDecoder	hAckDec.h
DtAcknowledgeEncoder	hAckEnc.h
DtAcknowledgeInteraction	hAckInter.h
DtAcknowledgePdu	ackPdu.h
DtAcknowledgeRPdu	ackRPdu.h
DtAckPredOpt	smUtil.h
DtActionRequestDecoder	hActReqDec.h
DtActionRequestEncoder	hActReqEnc.h
DtActionRequestInteraction	hActReqInter.h
DtActionRequestPdu	actionPdu.h
DtActionRequestRPdu	actReqRPdu.h
DtActionResponsePdu	actionPdu.h
DtActionResponseRPdu	actRespRPdu.h
DtAggregateCallbackInfo	aggCbInfo.h
DtAggregateEntityDecoder	aggEntDec.h
DtAggregateEntityEncoder	aggEntEnc.h
DtAggregatePublisher	dAggPub.h
DtAggregatePublisher	hAggPub.h
DtAggregateStateDecoder	dAggDec.h
DtAggregateStateEncoder	dAggEnc.h
DtAggregateStateRepository	aggregateSR.h
DtAggregateStatePdu	aggStatePdu.h
DtAntennaPatternRecord	antPatRec.h
DtApproximator	approximator.h
DtAppSpecificRadioSignalDecoder	hAppRadSigDec.h
DtAppSpecificRadioSignalEncoder	hAppRadSigEnc.h
DtArealObjectStatePdu	arealObjStPdu.h
DtArtPart	artPart.h
DtArtPartList	artPartList.h
DtArtPartSpec	apSpec.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtArtPartThresholder	artPartThresh.h
DtAttachedPartList	attPartList.h
DtBaseEmbeddedSystemDecoder	baseEmbedDec.h
DtBaseEmbeddedSystemEncoder	baseEmbedEnc.h
DtBaseEmitterBeamDecoder	baseEmittDec.h
DtBaseEmitterBeamEncoder	baseEmittEnc.h
DtBaseEntityDecoder	baseEntDec.h
DtBaseEntityEncoder	baseEntEnc.h
DtBaseEntityStateRepository	baseEntitySR.h
DtBaseExerciseConn	baseExConn.h
DtBaseHashKey	baseHashKey.h
DtBaseMsg	baseMsg.h
DtBaseReflectedAggregate	baseRefIAgg.h
DtBaseReflectedEntity	baseRefIEnt.h
DtBasicFixedWingAircraftRecord	gedBasFixWing.h
DtBasicGroundCombatSoldierRecord	gedBasSoldier.h
DtBasicGroundCombatVehicleRecord	gedBasVehicle.h
DtBasicRotorWingAircraftRecord	gedBasRotWing.h
DtBeamAntennaPatternRecord	beamAntRec.h
DtBurstDescriptor	hostStructs.h
DtCartesianView	cartView.h
DtClientSocket	ClientSocket.h
DtCollisionDecoder	hCollideDec.h
DtCollisionEncoder	hCollideEnc.h
DtCollisionElasticPdu	collisElasPdu.h
DtCollisionInteraction	hCollideInter.h
DtCollisionPdu	collidePdu.h
DtCommentPdu	commentPdu.h
DtCommentRPdu	commentRPdu.h
DtCompassVCPdu	vcPdu.h
DtCoordTransform	LibMatrix.h
DtCoordTranslation	LibMatrix.h
DtCreateEntityPdu	createPdu.h
DtCreateEntityRPdu	createEntRPdu.h
DtCsFilterTest	ClientSocket.h
DtDatabaseIndexRadioSignalDecoder	hDbiRadSigDec.h
DtDatabaseIndexRadioSignalEncoder	hDbiRadSigEnc.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtDataPdu	dataPdu.h
DtDataQueryPdu	queryPdu.h
DtDataQueryRPdu	dataQueryRPdu.h
DtDataRPdu	dataRPdu.h
DtDatumMap	datumMap.h
DtDatumMapEntry	datumMap.h
DtDatumRecordSet	datumRecSet.h
DtVarDatumSpec	
DtDcm	dcm.h
DtDeadReckoner	deadReckoner.h
DtStateDecoderFactory	decFactory.h
DtDesignatorCallbackInfo	designCbInfo.h
DtDesignatorDecoder	designatorDec.h
DtDesignatorEncoder	designatorEnc.h
DtDesignatorPdu	designatePdu.h
DtDesignatorPublisher	hDesignatrPub.h
DtDesignatorRepository	designatorSR.h
DtDetonationDecoder	hDetDec.h
DtDetonationEncoder	hDetEnc.h
DtDetonationInteraction	hDetInter.h
DtDetonationPdu	detonatePdu.h
DtDirectedRotRate	directRotRate.h
DtDirectRtiAmbassador	dirRtiAmb.h
DtDiscoverObjectCbInfo	dscvObjCbInfo.h
DtEntityPublisher	dEntityPub.h
DtExerciseConn	dExerciseConn.h
DtDisPduFactory	disPduFactory.h
DtReflectedEntity	dReflectedEnt.h
DtReflectedEntityList	dReflEntList.h
DtDmeGlobalFcns	datumMap.h
DtElectromagneticEmissionPdu	elecEmitPdu.h
DtEmbeddedSystemRepository	embedSystemSR.h
DtEmissionPdu	emissionPdu.h
DtEmitterBeamCallbackInfo	beamCbInfo.h
DtEmitterBeamPublisher	hEmitrBeamPub.h
DtEmitterBeamRepository	emitttrBeamSR.h
DtEmitterSystemCallbackInfo	emitrSysCbInf.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtEmitterSystemDecoder	emitterSysDec.h
DtEmitterSystemEncoder	emitterSysEnc.h
DtEmitterSystemPublisher	hEmitttrSysPub.h
DtEmitterSystemRepository	emitterSysSR.h
DtEncodedAudioRadioSignalDecoder	hEncARSigDec.h
DtEncodedAudioRadioSignalEncoder	hEncARSigEnc.h
DtStateEncoderFactory	encFactory.h
DtEnhancedFixedWingAircraftRecord	gedEnhFixWing.h
DtEnhancedGroundCombatSoldierRecord	gedEnhSoldier.h
DtEnhancedGroundCombatVehicleRecord	gedEnhVehicle.h
DtEnhancedRotorWingAircraftRecord	gedEnhRotWing.h
DtEnvironmentalEntityDecoder	envDec.h
DtEnvironmentalEntityEncoder	envEnc.h
DtEntityCallbackInfo	entCbInfo.h
DtEntityIdentifier	hostStructs.h
DtEntityIdHashList	dEntIdHashLst.h
DtEntityPublisher	dEntityPub.h hEntityPub.h
DtEntityStateDecoder	dEntityDec.h
DtEntityStateEncoder	dEntityEnc.h
DtEntityStatePdu	esPdu.h
DtEntityStateRepository	entitySR.h
DtEntityStateUpdatePdu	esUpdatePdu.h
DtEntityType	hostStructs.h
DtEnvironmentalProcessPdu	envProcPdu.h
DtEnvironmentTypeRecord	enviroTypeRec.h
DtEventId	eventId.h
DtEventReportPdu	eventReptPdu.h
DtEventReptRPdu	eventReptRPdu.h
DtExerciseConn	hExerciseConn.h
DtFederateId	hostStructs.h
DtFilename	filename.h
DtFilterTest	Socket.h
DtFireDecoder	hFireDec.h
DtFireEncoder	hFireEnc.h
DtFireInteraction	hFireInter.h
DtFirePdu	firePdu.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtFom	fom.h
DtFomMapper	fomMapper.h
DtFundamentalParameterData	elecEmitPdu.h
DtFundamentalParameterData	emissionPdu.h
DtGeodeticCoord	geodCoord.h
DtGlobalIdHashList	globldHLst.h
DtGlobalObjectDesignatorList	objDesList.h
DtGridAxisRecordFixed	gridAxiRecFix.h
DtGridAxisRecordVariable	gridAxiRecVar.h
DtGridDataRecord	gridDataRec.h
DtGridDataRecordType0	gridDataRecT0.h
DtGridDataRecordType1	gridDataRecT1.h
DtGridDataRecordType2	gridDataRecT2.h
DtGriddedDataPdu	gridDataPdu.h
DtGroundLogisticsVehicleRecord	gedLogVehicle.h
DtGroupVCPdu	vcPdu.h
DtHashedItem	DtOldHashlist, DtOldIntrusiveHashlist
DtHashlist	hashlist.h
DtHlaObject	hlaObj.h
DtHlaObjectFactory	hlaObjFact.h
DtHlaObjectWithStateRep	hlaObjWithSR.h
DtHlaStateDecoder	hStateDecoder.h
DtHlaStateEncoder	hStateEncoder.h
DtIffAtcNavAidsBeamData	iffAtcNavBeam.h
DtIffAtcNavAidsLayerHeader	iffAtcNavLayr.h
DtIffAtcNavAidsOperational	iffAtcNavOper.h
DtIffAtcNavAidsParameter	iffAtcNavParm.h
DtIffAtcNavAidsPdu	iffAtcNavPdu.h
DtIffAtcNavAidsSecondOperational	iffAtcNavOpr2.h
DtIffAtcNavAidsSystemID	iffAtcNavSyID.h
DtInteraction	dInteraction.h
DtInteraction	hInteraction.h
DtInteractionDecoder	interDecoder.h
DtInteractionDecoderFactory	intDecFact.h
DtInteractionEncoder	interEncoder.h
DtInteractionEncoderFactory	intEncFact.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtInteractionFactory	hInterFactory.h
DtInteractionManager	interMgr.h
DtInteractionWithData	hInterWData.h
DtInteractionWithEncDec	hIntWEncDec.h
DtInterClassDesc	intClassDesc.h
DtInterClassMapping	intClassMap.h
DtIntercomControlPdu	intercmCtlPdu.h
DtIntercomEntityDestRec	intercmEntDst.h
DtIntercomGroupAsgmtRec	intercmGrpAsg.h
DtIntercomGroupDestRec	intercmGrpDst.h
DtIntercomParmRec	intercmParRec.h
DtIntercomSignalPdu	intercmSigPdu.h
DtIntHashKey	intHashKey.h
DtIntrusiveList	vList.h
DtIsGroupOfPdu	isGroupOfPdu.h
DtIsPartOfPdu	isPartOfPdu.h
DtLaserPdu	laserPdu.h
DtLeAppearancePdu	leAppearPdu.h
DtLeArticulatedPartsPdu	leArtPartsPdu.h
DtLeDeadReckoningParameters	leDeadReck.h
DtLeDetonationPdu	leDetonPdu.h
DtLeEntityID	leEntityID.h
DtLeEntityLinearVelocity	leEntityVel.h
DtLeEntityLocationEntityCoordinates	leLocEntity.h
DtLeEntityLocationWorldCoordinates	leLocWorld.h
DtLeEntityOrientation	leEntityOri.h
DtLeFirePdu	leFirePdu.h
DtLeOrientationError	leOriError.h
DtLePositionError	lePosError.h
DtLeTspiPdu	leTspiPdu.h
DtLgrControlDecoder	hLcDec.h
DtLgrControlEncoder	hLcEnc.h
DtLgrControlInteraction	hLcInter.h
DtLgrControlPdu	lcPdu.h
DtLgrEndPdu	lcEnd.h
DtLgrEofActionPdu	lcEofAction.h
DtLgrHeartbeatPdu	lcHeartbeat.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtLgrPausePdu	lcPause.h
DtLgrPlayPdu	lcPlay.h
DtLgrPlayActionPdu	lcPlayAction.h
DtLgrPbFilePdu	lcPlayFile.h
DtLgrQuitPdu	lcQuit.h
DtLgrRecordActionPdu	lcRecAction.h
DtLgrRecFilePdu	lcRecFile.h
DtLgrRecordPdu	lcRecord.h
DtLgrSetTimePdu	lcSetTime.h
DtLgrSkipTimePdu	lcSkipTime.h
DtLgrStartPdu	lcStart.h
DtLgrStopPdu	lcStop.h
DtLgrTimeScalePdu	lcTimeScale.h
DtLgrTimeoutPdu	lcTimeout.h
DtLinearObjectStatePdu	linObjStPdu.h
DtLinearSegmentParameter	linSegParam.h
DtList	vllist.h
DtListedObj	vllist.h
DtListItem	vllist.h
DtLocalHlaObjectManager	locObjMgr.h
DtMessagePdu	messagePdu.h
DtMilitaryEntityDecoder	milEntDec.h
DtMilitaryEntityEncoder	milEntEnc.h
DtMilitaryPlatformEntityDecoder	milPlatDec.h
DtMilitaryPlatformEntityEncoder	milPlatEnc.h
DtMimicTrackVCPdu	vcPdu.h
DtMimicVCPdu	vcPdu.h
DtMinefieldDataPdu	mineDataPdu.h
DtMinefieldPerimeterRecord	minePerimeter.h
DtMinefieldQueryPdu	mineQueryPdu.h
DtMinefieldResponseNackPdu	mineRsNackPdu.h
DtMinefieldStatePdu	mineStatePdu.h
DtModelessVCPdu	vcPdu.h
DtModulationType	transmitPdu.h
DtMunitionEntityDecoder	munEntDec.h
DtMunitionEntityEncoder	munEntEnc.h
DtNet32Vector	NetStructs.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtNet64Vector	NetStructs.h
DtNetBoolean	netBoolean.h
DtNetBurstDescriptor	NetStructs.h
DtNetEntityIdentifier	NetStructs.h
DtNetEntityMarking	NetStructs.h
DtNetEntityType	NetStructs.h
DtNetEulerAngles	NetStructs.h
DtNetEventId	NetStructs.h
DtNetFederateId	NetStructs.h
DtNetFloat32	NetTypes.h
DtNetFloat64	NetTypes.h
DtNetGlobalObjectDesignator	NetStructs.h
DtNetInt16	NetTypes.h
DtNetInt32	NetTypes.h
DtNetRadioEntityType	NetStructs.h
DtNetSimulationAddress	NetStructs.h
DtNetSocket	NetSocket.h
DtNetTimeStamp	netTimeStamp.h
DtNetU16	NetTypes.h
DtNetU32	NetTypes.h
DtObjClassDesc	objClassDesc.h
DtObjectId	hostStructs.h
DtObjectIdHashKey	objIdHashKey.h
DtObjectIdHashlist	objIdHashLst.h
DtObjectPublisher	hObjPub.h
DtObjectReflectAttributeValuesCbInfo	objRefCbInfo.h
DtObjectType	objectType.h
DtOldHashlist, DtOldIntrusiveHashlist In VR-Link 3.3, we significantly changed the semantics of our DtHashlist class. The new version is defined in hashlist.h. The older DtHashlist class has been renamed DtOldHashlist, and is still defined here in this file for backwards compatibility. While we suggest that you port your code to the new DtHashlist, which is cleaner and easier to use, if you do not want to make that change, you can just use DtOldHashlist wherever you used to use DtHashlist, and the rest of your old code should work.	
DtOrbitVCPdu	vcPdu.h
DtPdu	pdu.h
DtPduCallbackInfo	pduCbInfo.h
DtPduCallbackManager	pduCbMgr.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtPduDatumRecordSet	pduDatRecSet.h
DtPduFactory	pduFactory.h
DtPduWithData	pduWithData.h
DtPhysicalEntityDecoder	physEntDec.h
DtPhysicalEntityEncoder	physEntEnc.h
DtPointObjectStatePdu	pointObjStPdu.h
DtPropulsionSystem	supEmisEsProp.h
DtRadioEntityType	hostStructs.h
DtRadioReceiverCallbackInfo	radRcvrCbInfo.h
DtRadioReceiverDecoder	radioRecvrDec.h
DtRadioReceiverEncoder	radioRecvrEnc.h
DtRadioReceiverPublisher	hRadioRcvrPub.h
DtRadioReceiverRepository	radioRecvrSR.h
DtRadioTransmitterCallbackInfo	radXmitCbInfo.h
DtRadioTransmitterDecoder	radioXmitDec.h
DtRadioTransmitterEncoder	radioXmitEnc.h
DtRadioTransmitterPublisher	hRadioXmitPub.h
DtRadioTransmitterRepository	radioXmitSR.h
DtRawBinaryRadioSignalDecoder	hRawBinRSDec.h
DtRawBinaryRadioSignalEncoder	hRawBinRSEnc.h
DtReceiveInteractionCbInfo	recIntrCbInfo.h
DtReceiverPdu	receivePdu.h
DtRecordPdu	recordPdu.h
DtRecordQueryPdu	recQueryPdu.h
DtRecordSet	recordSet.h
DtRecordSetPdu	recordSetPdu.h
DtReflectAttributeValuesCbInfo	refAttrCbInfo.h
DtReflectAttributeValuesCbList	refAttrCbList.h
DtReflectedAggregate	dReflectedAgg.h
DtReflectedAggregate	hReflectedAgg.h
DtReflectedAggregateList	dRefIAggList.h
DtReflectedAggregateList	hRefIAggList.h
DtReflectedEmitterBeamList	hRefBeamList.h
DtReflectedDesignatorList	hRefDesigList.h
DtReflectedEmitterSystemList	hRefEmSysList.h
DtReflectedEmitterBeam	hRefIBeam.h
DtReflectedDesignator	hRefIDesigntr.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtReflectedEmitterSystem	hRefIEmitrSys.h
DtReflectedEntity	dReflectedEnt.h hReflectedEnt.h
DtReflectedEntityHashlist	dRefEntList.h
DtReflectedEntityHashlist	hRefEntHList.h
DtReflectedEntityList	hRefEntList.h dRefEntList.h
DtReflectedHlaObjectManager	refObjMgr.h
DtReflectedObject	hReflectedObj.h
DtReflectedObjectCallbackInfo	refObjCbInfo.h
DtReflectedObjectList	hRefObjList.h
DtReflectedRadioReceiver	hRefIRadRecvr.h
DtReflectedRadioTransmitter	hRefIRadXmitr.h
DtReflectedRadioRecieverList	hRefRadRvList.h
DtReflectedRadioTransmitterList	hRefRadXmList.h
DtRegControlCbInfo	regCtlCbInfo.h
DtRegControlCbList	regCtlCbList.h
DtRemoveEntityPdu	removePdu.h
DtRemoveEntityRPdu	removeEntRPdu.h
DtRemoveObjectCbInfo	remObjCbInfo.h
DtRepairCompletePdu	repairPdu.h
DtRepairResponsePdu	repairPdu.h
DtResupplyCancelPdu	resupplyPdu.h
DtResupplyOfferPdu	resupplyPdu.h
DtResupplyReceivedPdu	resupplyPdu.h
DtRprFomMapper	rprFomMap.h
DtRprHlaObjectFactory	rprObjFact.h
DtRprInteractionDecoderFactory	rprlDecFact.h
DtRprInteractionEncoderFactory	rprlEncFact.h
DtRprInteractionFactory	rprlInterFact.h
DtRprStateDecoderFactory	rprDecFact.h
DtRprStateEncoderFactory	rprEncFact.h
DtScript	script.h
DtSelectParam	vlutil.h
DtServiceRequestPdu	servReqPdu.h
DtSetDataPdu	setDataPdu.h
DtSetDataRPdu	setDataRPdu.h
DtSetRecordPdu	setRecordPdu.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtSignalInteraction	hSignalInter.h
DtSignalPdu	signalPdu.h
DtSilentAggregateList	aggAggList.h
DtSilentEntityList	aggEntList.h
DtSimanPdu	simanPdu.h
DtSimpleInteractionWithEncDec	simpIntWED.h
DtSmoother	smoother.h
DtSmoothOrient	smoothUtil.h
DtSmoothOriWithDr	smoothOriWDR.h
DtSmoothPos	smoothUtil.h
DtSmoothPosWithDr	smoothPosWDR.h
DtSocket	Socket.h
DtSocketPairSocket	pairSock.h
DtSoldierDecoder	soldierDec.h
DtSoldierEncoder	soldierEnc.h
DtStartResumeDecoder	hStartDec.h
DtStartResumeEncoder	hStartEnc.h
DtStartResumeInteraction	hStartInter.h
DtStartResumePdu	startPdu.h
DtStartResumeRPdu	startResRPdu.h
DtStartStopInterCbInfo	stIntCbInfo.h
DtStartStopInterCbList	stIntCbList.h
DtStartUpdatesCbInfo	strtUpdCbInfo.h
DtStartUpdatesCbList	strtUpdCbList.h
DtStateMsg	dStateMsg.h
DtStateMsg	hStateMsg.h
DtStateRepository	stateRep.h
DtStopFreezeDecoder	hStopDec.h
DtStopFreezeEncoder	hStopEnc.h
DtStopFreezeInteraction	hStopInter.h
DtStopFreezePdu	stopPdu.h
DtStopFreezeRPdu	stopFrzRPdu.h
DtStopUpdatesCbInfo	stopUpdCbInfo.h
DtStopUpdatesCbList	stopUpdCbList.h
DtString	str.h
DtStringHashKey	strHashKey.h
DtStringHashlist	strHashLst.h

Table 9-2: Classes and their Corresponding Header Files (Continued)

Class	Header File
DtSuppEmisEntStatePdu	supEmisEsPdu.h
DtTaitBryan	taitBryan.h
DtTetherTrackVCPdu	vcPdu.h
DtTetherVCPdu	vcPdu.h
DtTimeStampType	tStampType.h
DtThresholder	thresholder.h
DtTopoView	topoView.h
DtTrackJamBeamDecoder	trkJamBeamDec.h
DtTrackJamBeamEncoder	trkJamBeamEnc.h
DtTrackJamTarget	elecEmitPdu.h
DtTrackJamTarget	emissionPdu.h
DtTrackVCPdu	vcPdu.h
DtTransferControlRequestPdu	transCtlPdu.h
DtTransmitterPdu	transmitPdu.h
DtUnderwaterAcousticAddtnlPassiveAct	undwaAcPasAct.h
DtUnderwaterAcousticBeam	undwaAcBeam.h
DtUnderwaterAcousticEmitterSystem	undwaAcEmitSys.h
DtUnderwaterAcousticPdu	undwaAcPdu.h
DtUnderwaterAcousticShaftSpeed	undwaAcShaft.h
DtUnknownHlaObject	unknownHObj.h
DtUnknownInteraction	hUnknownInter.h
DtUnknownPdu	unknownPdu.h
DtUnknownVCPdu	vcPdu.h
DtUpdateControlCbInfo	updCtlCbInfo.h
DtUpdateControlCbList	updCtlCbList.h
DtUtmCoord	utmCoord.h
DtUtmView	utmView.h
DtVariableDatumRecord	varDatumRec.h
DtVector	vlVector.h
DtVectoringNozzleSystem	supEmisEsNozz.h
DtViewControlDecoder	hVcDec.h
DtViewControlEncoder	hVcEnc.h
DtViewControlInteraction	hVcInter.h
DtViewControlPdu	vcPdu.h
DtVrIFederateAmbassador	vrlFedAmb.h
DtVrIFedTime	vrlFedTime.h
DtVrIRtiAmbassador	vrlRtiAmb.h

9.3. Header-to-Class Cross Reference

Table 9-3 contains an alphabetical list of VR-Link header files, along with the classes they contain. In the on-line version of this manual, the header file names are hypertext links to the corresponding pages in Appendix B.

Table 9-3: Header Files and the Classes they Contain

Header File	Class
absNotifyObs.h	DtAbsNotifyObserver
ackPdu.h	DtAcknowledgePdu
ackRPdu.h	DtAcknowledgeRPdu
actionPdu.h	DtActionRequestPdu DtActionResponsePdu
actReqRPdu.h	DtActionRequestRPdu
actRespRPdu.h	DtActionResponseRPdu
aggAggList.h	DtSilentAggregateList
aggCbInfo.h	DtAggregateCallbackInfo
aggEntDec.h	DtAggregateEntityDecoder
aggEntEnc.h	DtAggregatedEntityEncoder
aggEntList.h	DtSilentEntityList
aggregateSR.h	DtAggregateStateRepository
aggStatePdu.h	DtAggregateStatePdu
antPatRec.h	DtAntennaPatternRecord
approximator.h	DtApproximator
apSpec.h	DtArtPartSpec
arealObjStPdu.h	DtArealObjectStatePdu
artPart.h	DtArtPart
artPartList.h	DtArtPartList
artPartThresh.h	DtArtPartThresholder
attPartList.h	DtAttachedPartList
baseEmbedDec.h	DtBaseEmbeddedSystemDecoder
baseEmbedEnc.h	DtBaseEmbeddedSystemEncoder
baseEmittrDec.h	DtBaseEmitterBeamDecoder
baseEmittrEnc.h	DtBaseEmitterBeamEncoder
baseEntDec.hr	DtBaseEntityDecoder
baseEntEnc.h	DtBaseEntityEncoder
baseEntitySR.h	DrBaseEntityStateRepository
baseExConn.h	DtBaseExerciseConn
baseHashKey.h	DtBaseHashKey
baseMsg.h	DtBaseMsg
baseReflAgg.h	DtBaseReflectedAggregate

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
baseRefEnt.h	DtBaseReflectedEntity
beamAntRec.h	DtBeamAntennaPatternRecord
beamCbInfo.h	DtEmitterBeamCallbackInfo
cartView.h	DtCartesianView
ClientSocket.h	DtClientSocket DtCsFilterTest
collidePdu.h	DtCollisionPdu
collisElasPdu.h	DtCollisionElasticPdu
commentPdu.h	DtCommentPdu
commentRPdu.h	DtCommentRPdu
createEntRPdu.h	DtCreateEntityRPdu
createPdu.h	DtCreateEntityPdu
dAggDec.h	DtAggregateStateDecoder
dAggEnc.h	DtAggregateStateEncoder
dAggPub.h	DtAggregatePublisher
dataPdu.h	DtDataPdu
dataQueryRPdu.h	DtDataQueryRPdu
dataRPdu.h	DtDataRPdu
datumMap.h	DtDatumMap DtDatumMapEntry DtDmeGlobalFcns
datumRecSet.h	DtVarDatumSpec DtDatumRecordSet
dcm.h	DtDcm
deadReckoner.h	DtDeadReckoner
decFactory.h	DtStateDecoderFactory
dEntIdHashLst.h	DtEntityIdHashList
dEntityDec.h	DtEntityStateDecoder
dEntityEnc.h	DtEntityStateEncoder
dEntityPub.h	DtEntityPublisher
designatePdu.h	DtDesignatorPdu
designatorDec.h	DtDesignatorDecoder
designatorEnc.h	DtDesignatorEncoder
designatorSR.h	DtDesignatorRepository
designCbInfo.h	DtDesignatorCallbackInfo
detonatePdu.h	DtDetonationPdu
dExerciseConn.h	DtExerciseConn
dInteraction.h	DtInteraction

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
directRotRate.h	DtDirectedRotRate
dirRtiAmb.h	DtDirectRtiAmbassador
disPduFactory.h	DtDisPduFactory
dRefIAggList.h	DtReflectedAggregateList
dReflectedAgg.h	DtReflectedAggregate
dReflectedEnt.h	DtReflectedEntity
dRefIEntList.h	DtReflectedEntityList DtReflectedEntityHashlist
dscvObjCbInfo.h	DtDiscoverObjectCbInfo
dStateMsg.h	DtStateMsg
elecEmitPdu.h	DtElectromagneticEmissionPdu DtFundamentalParameterData DtTrackJamTarget
embedSystemSR.h	DtEmbeddedSystemRepository
emissionPdu.h	DtEmissionPdu DtFundamentalParameterData DtTrackJamTarget
emitrSysCbInf.h	DtEmitterSystemCallbackInfo
emitterSysDec.h	DtEmitterSystemDecoder
emitterSysEnc.h	DtEmitterSystemEncoder
emitterSysSR.h	DtEmitterSystemRepository
emitrBeamSR.h	DtEmitterBeamRepository
encFactory.h	DtStateEncoderFactory
entCbInfo.h	DtEntityCallbackInfo
entitySR.h	DtEntityStateRepository
envDec.h	DtEnvironmentalEntityDecoder
envEnc.h	DtEnvironmentalEntityEncoder
enviroTypeRec.h	DtEnvironmentTypeRecord
envProcPdu.h	DtEnvironmentalProcessPdu
esPdu.h	DtEntityStatePdu
esUpdatePdu.h	DtEntityStateUpdatePdu
eventId.h	DtEventId
eventReptPdu.h	DtEventReportPdu
eventReptRPdu.h	DtEventReptRPdu
filename.h	DtFilename
firePdu.h	DtFirePdu
fom.h	DtFom
fomMapper.h	DtFomMapper

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
gedBasFixWing.h	DtBasicFixedWingAircraftRecord
gedBasRotWing.h	DtBasicRotorWingAircraftRecord
gedBasSoldier.h	DtBasicGroundCombatSoldierRecord
gedBasVehicle.h	DtBasicGroundCombatVehicleRecord
gedEnhFixWing.h	DtEnhancedFixedWingAircraftRecord
gedEnhRotWing.h	DtEnhancedRotorWingAircraftRecord
gedEnhSoldier.h	DtEnhancedGroundCombatSoldierRecord
gedEnhVehicle.h	DtEnhancedGroundCombatVehicleRecord
gedLogVehicle.h	DtGroundLogisticsVehicleRecord
geodCoord.h	DtGeodeticCoord
globldHLst.h	DtGloballdHashList
gridAxiRecFix.h	DtGridAxisRecordFixed
gridAxiRecVar.h	DtGridAxisRecordVariable
gridDataPdu.h	DtGriddedDataPdu
gridDataRec.h	DtGridDataRecord
gridDataRecT0.h	DtGridDataRecordType0
gridDataRecT1.h	DtGridDataRecordType1
gridDataRecT2.h	DtGridDataRecordType2
hAckDec.h	DtAcknowledgeDecoder
hAckEnc.h	DtAcknowledgeEncoder
hAckInter.h	DtAcknowledgeInteraction
hActReqDec.h	DtActionRequestDecoder
hActReqEnc.h	DtActionRequestEncoder
hActReqInter.h	DtActionRequestInteraction
hAggPub.h	DtAggregatePublisher
hAppRadSigDec.h	DtAppSpecificRadioSignalDecoder
hAppRadSigEnc.h	DtAppSpecificRadioSignalEncoder
hashlist.h	DtHashlist
hCollideDec.h	DtCollisionDecoder
hCollideEnc.h	DtCollisionEncoder
hCollideInter.h	DtCollisionInteraction
hDbiRadSigDec.h	DtDatabaseIndexRadioSignalDecoder
hDbiRadSigEnc.h	DtDatabaseIndexRadioSignalEncoder
hDesignatrPub.h	DtDesignatorPublisher
hDetDec.h	DtDetonationDecoder
hDetEnc.h	DtDetonationEncoder
hDetInter.h	DtDetonationInteraction

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
hEmitrBeamPub.h	DtEmitterBeamPublisher
hEmittrSysPub.h	DtEmitterSystemPublisher
hEncARSigDec.h	DtEncodedAudioRadioSignalDecoder
hEncARSigEnc.h	DtEncodedAudioRadioSignalEncoder
hEntityPub.h	DtEntityPublisher
hExerciseConn.h	DtExerciseConn
hFireDec.h	DtFireDecoder
hFireEnc.h	DtFireEncoder
hFireInter.h	DtFireInteraction
hInteraction.h	DtInteraction
hInterFactory.h	DtInteractionFactory
hInterWData.h	DtInteractionWithData
hIntWEncDec.h	DtInteractionWithEncDec
hlaObj.h	DtHlaObject
hlaObjFact.h	DtHlaObjectFactory
hlaObjWithSR.h	DtHlaObjectWithStateRep
hLcDec.h	DtLgrControlDecoder
hLcEnc.h	DtLgrControlEncoder
hLcInter.h	DtLgrControlInteraction
hObjPub.h	DtObjectPublisher
hostStructs.h	DtBurstDescriptor
	DtEntityIdentifier
	DtEntityType
	DtFederateId
	DtObjectId
	DtRadioEntityType
hRadioRcvrPub.h	DtRadioReceiverPublisher
hRadioXmitPub.h	DtRadioTransmitterPublisher
hRawBinRSDec.h	DtRawBinaryRadioSignalDecoder
hRawBinRSEnc.h	DtRawBinaryRadioSignalEncoder
hRefBeamList.h	DtReflectedEmitterBeamList
hRefDesigList.h	DtReflectedDesignatorList
hRefEmSysList.h	DtReflectedEmitterSystemList
hRefEntHList.h	DtReflectedEntityHashlist
hRefIAggList.h	DtReflectedAggregateList
hRefIBeam.h	DtReflectedEmitterBeam
hRefIDesigntr.h	DtReflectedDesignator
hReflectedAgg.h	DtReflectedAggregate

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
hReflectedObj.h	DtReflectedObject
hRefEmitrSys.h	DtReflectedEmitterSystem
hReflectedEnt.h	DtReflectedEntity
hRefEntList.h	DtReflectedEntityList
hRefIRadRecvr.h	DtReflectedRadioReceiver
hRefIRadXmitr.h	DtReflectedRadioTransmitter
hRefObjList.h	DtReflectedObjectList
hRefRadRvList.h	DtReflectedRadioReceiverList
hRefRadXmList.h	DtReflectedRadioTransmitterList
hSignalInter.h	DtSignalInteraction
hStartDec.h	DtStartResumeDecoder
hStartEnc.h	DtStartResumeEncoder
hStartInter.h	DtStartResumeInteraction
hStateDecoder.h	DtHlaStateDecoder
hStateEncoder.h	DtHlaStateEncoder
hStateMsg.h	DtStateMsg
hStopDec.h	DtStopFreezeDecoder
hStopEnc.h	DtStopFreezeEncoder
hStopInter.h	DtStopFreezeInteraction
hUnknownInter.h	DtUnknownInteraction
hVcDec.h	DtViewControlDecoder
hVcEnc.h	DtViewControlEncoder
hVcInter.h	DtViewControlInteraction
iffAtcNavBeam.h	DtIffAtcNavAidsBeamData
iffAtcNavLayr.h	DtIffAtcNavAidsLayerHeader
iffAtcNavOper.h	DtIffAtcNavAidsOperational
iffAtcNavOpr2.h	DtIffAtcNavAidsSecondOperational
iffAtcNavParm.h	DtIffAtcNavAidsParameter
iffAtcNavPdu.h	DtIffAtcNavAidsPdu
iffAtcNavSyID.h	DtIffAtcNavAidsSystemID
intClassDesc.h	DtInterClassDesc
intClassMap.h	DtInterClassMapping
intDecFact.h	DtInteractionDecoderFactory
intEncFact.h	DtInteractionEncoderFactory
intercmCtlPdu.h	DtIntercomControlPdu
intercmEntDst.h	DtIntercomEntityDestRec
intercmGrpAsg.h	DtIntercomGroupAsgmtRec

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
intercmGrpDst.h	DtIntercomGroupDestRec
intercmParRec.h	DtIntercomParmRec
intercmSigPdu.h	DtIntercomSignalPdu
interDecoder.h	DtInteractionDecoder
interEncoder.h	DtInteractionEncoder
interMgr.h	DtInteractionManager
intHashKey.h	DtIntHashKey
isGroupOfPdu.h	DtIsGroupOfPdu
isPartOfPdu.h	DtIsPartOfPdu
laserPdu.h	DtLaserPdu
lcEnd.h	DtLgrEndPdu
lcEofAction.h	DtLgrEofActionPdu
lcHeartbeat.h	DtLgrHeartbeatPdu
lcPause.h	DtLgrPausePdu
lcPdu.h	DtLgrControlPdu
lcPlay.h	DtLgrPlayPdu
lcPlayAction.h	DtLgrPlayActionPdu
lcPlayFile.h	DtLgrPbFilePdu
lcQuit.h	DtLgrQuitPdu
lcRecAction.h	DtLgrRecordActionPdu
lcRecFile.h	DtLgrRecFilePdu
lcRecord.h	DtLgrRecordPdu
lcSetTime.h	DtLgrSetTimePdu
lcSkipTime.h	DtLgrSkipTimePdu
lcStart.h	DtLgrStartPdu
lcStop.h	DtLgrStopPdu
lcTimeScale.h	DtLgrTimeScalePdu
lcTimeout.h	DtLgrTimeoutPdu
leAppearPdu.h	DtLeAppearancePdu
leArtPartsPdu.h	DtLeArticulatedPartsPdu
leDeadReck.h	DtLeDeadReckoningParameters
leDetonPdu.h	DtLeDetonationPdu
leEntityID.h	DtLeEntityID
leEntityOri.h	DtLeEntityOrientation
leEntityVel.h	DtLeEntityLinearVelocity
leFirePdu.h	DtLeFirePdu
leLocEntity.h	DtLeEntityLocationEntityCoordinates

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
leLocWorld.h	DtLeEntityLocationWorldCoordinates
leOriError.h	DtLeOrientationError
lePosError.h	DtLePositionError
leTspiPdu.h	DtLeTspiPdu
LibMatrix.h	DtCoordTransform DtCoordTranslation
linObjStPdu.h	DtLinearObjectStatePdu
linSegParam.h	DtLinearSegmentParameter
locObjMgr.h	DtLocalHlaObjectManager
messagePdu.h	DtMessagePdu
milEntDec.h	DtMilitaryEntityDecoder
milEntEnc.h	DtMilitaryEntityEncoder
milPlatDec.h	DtMilitaryPlatformEntityDecoder
milPlatEnc.h	DtMilitaryPlatformEntityEncoder
mineDataPdu.h	DtMinefieldDataPdu
minePerimeter.h	DtMinefieldPerimeterRecord
mineQueryPdu.h	DtMinefieldQueryPdu
mineRsNackPdu.h	DtMinefieldResponseNackPdu
mineStatePdu.h	DtMinefieldStatePdu
munEntDec.h	DtMunitionEntityDecoder
munEntEnc.h	DtMunitionEntityEncoder
netBoolean.h	DtNetBoolean
NetSocket.h	DtNetSocket
NetStructs.h	DtNet32Vector DtNet64Vector DtNetBurstDescriptor DtNetEntityIdentifier DtNetEntityMarking DtNetEntityType DtNetEulerAngles DtNetEventId DtNetFederateId DtNetGlobalObjectDesignator DtNetRadioEntityType DtNetSimulationAddress
netTimeStamp.h	DtNetTimeStamp

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
NetTypes.h	DtNetFloat32 DtNetFloat64 DtNetInt16 DtNetInt32 DtNetU16 DtNetU32
objClassDesc.h	DtObjClassDesc
objDesList.h	DtGlobalObjectDesignatorList
objectType.h	DtObjectType
objIdHashKey.h	DtObjectIdHashKey
objIdHashLst.h	DtObjectIdHashlist
objRefCbInfo.h	DtObjectReflectAttributeValuesCbInfo
oldHashList.h	In VR-Link 3.3, we significantly changed the semantics of our DtHashlist class. The new version is defined in hashlist.h. The older DtHashlist class has been renamed DtOldHashlist, and is still defined here in this file for backwards compatibility. While we suggest that you port your code to the new DtHashlist, which is cleaner and easier to use, if you do not want to make that change, you can just use DtOldHashlist wherever you used to use DtHashlist, and the rest of your old code should work.
pairSock.h	DtSocketPairSocket
pdu.h	DtPdu
pduCbInfo.h	DtPduCallbackInfo
pduCbMgr.h	DtPduCallbackManager
pduDatRecSet.h	DtPduDatumRecordSet
pduFactory.h	DtPduFactory
pduWithData.h	DtPduWithData
physEntDec.h	DtPhysicalEntityDecoder
physEntEnc.h	DtPhysicalEntityEncoder
pointObjStPdu.h	DtPointObjectStatePdu
queryPdu.h	DtDataQueryPdu
radioRecvrDec.h	DtRadioReceiverDecoder
radioRecvrEnc.h	DtRadioReceiverEncoder
radioRecvrSR.h	DtRadioReceiverRepository
radioXmitDec.h	DtRadioTransmitterDecoder
radioXmitEnc.h	DtRadioTransmitterEncoder

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
radioXmitSR.h	DtRadioTransmitterRepository
radRcvrCbInfo.h	DtRadioReceiverCallbackInfo
radXmitCbInfo.h	DtRadioTransmitterCallbackInfo
receivePdu.h	DtReceiverPdu
recIntrCbInfo.h	DtReceiveInteractionCbInfo
recordPdu.h	DtRecordPdu
recordSet.h	DtRecordSet
recordSetPdu.h	DtRecordSetPdu
recQueryPdu.h	DtRecordQueryPdu
refAttrCbInfo.h	DtReflectAttributeValuesCbInfo
refAttrCbList.h	DtReflectAttributeValuesCbList
refObjCbInfo.h	DtReflectedObjectCallbackInfo
refObjMgr.h	DtReflectedHlaObjectManager
regCtlCbInfo.h	DtRegControlCbInfo
regCtlCbList.h	DtRegControlCbList
remObjCbInfo.h	DtRemoveObjectCbInfo
removeEntRPdu.h	DtRemoveEntityRPdu
removePdu.h	DtRemoveEntityPdu
repairPdu.h	DtRepairCompletePdu DtRepairResponsePdu
resupplyPdu.h	DtResupplyCancelPdu DtResupplyOfferPdu DtResupplyReceivedPdu
rprDecFact.h	DtRprStateDecoderFactory
rprEncFact.h	DtRprStateEncoderFactory
rprFomMap.h	DtRprFomMapper
rprlDecFact.h	DtRprlInteractionDecoderFactory
rprlEncFact.h	DtRprlInteractionEncoderFactory
rprlInterFact.h	DtRprlInteractionFactory
rprObjFact.h	DtRprHlaObjectFactory
script.h	DtScript
servReqPdu.h	DtServiceRequestPdu
setDataPdu.h	DtSetDataPdu
setDataRPdu.h	DtSetDataRPdu
setRecordPdu.h	DtSetRecordPdu
signalPdu.h	DtSignalPdu
simanPdu.h	DtSimanPdu
simplntWED.h	DtSimpleInteractionWithEncDec

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
smoother.h	DtSmoother
smoothOriWDR.h	DtSmoothOriWithDr
smoothPosWDR.h	DtSmoothPosWithDr
smoothUtil.h	DtSmoothOrient DtSmoothPos
smUtil.h	DtAckPredOpt
Socket.h	DtFilterTest DtSocket
soldierDec.h	DtSoldierDecoder
soldierEnc.h	DtSoldierEncoder
startPdu.h	DtStartResumePdu
startResRPdu.h	DtStartResumeRPdu
stateRep.h	DtStateRepository
stIntCbInfo.h	DtStartStopInterCbInfo
stIntCbList.h	DtStartStopInterCbList
stopFrzRPdu.h	DtStopFreezeRPdu
stopPdu.h	DtStopFreezePdu
stopUpdCbInfo.h	DtStopUpdatesCbInfo
stopUpdCbList.h	DtStopUpdatesCbList
str.h	DtString
strHashKey.h	DtStringHashKey
strHashLst.h	DtStringHashlist
strtUpdCbInfo.h	DtStartUpdatesCbInfo
strtUpdCbList.h	DtStartUpdatesCbList
supEmisEsNozz.h	DtVectoringNozzleSystem
supEmisEsPdu.h	DtSuppEmisEntStatePdu
supEmisEsProp.h	DtPropulsionSystem
taitBryan.h	DtTaitBryan
threshold.h	DtThresholder
topoView.h	DtTopoView
transCtlPdu.h	DtTransferControlRequestPdu
transmitPdu.h	DtModulationType DtTransmitterPdu
trkJamBeamDec.h	DtTrackJamBeamDecoder
trkJamBeamEnc.h	DtTrackJamBeamEncoder
tStampType.h	DtTimeStampType
undwaAcBeam.h	DtUnderwaterAcousticBeam
undwaAcEmitSys.h	DtUnderwaterAcousticEmitterSystem

Table 9-3: Header Files and the Classes they Contain (Continued)

Header File	Class
undwaAcPasAct.h	DtUnderwaterAcousticAddtnlPassiveAct
undwaAcPdu.h	DtUnderwaterAcousticPdu
undwaAcShaft.h	DtUnderwaterAcousticShaftSpeed
unknownHObj.h	DtUnknownHlaObject
unknownPdu.h	DtUnknownPdu
updCtlCbInfo.h	DtUpdateControlCbInfo
updCtlCbList.h	DtUpdateControlCbList
utmCoord.h	DtUtmCoord
utmView.h	DtUtmView
varDatumRec.h	DtVariableDatumRecord
vcPdu.h	DtAbsoluteVCPdu
	DtCompassVCPdu
	DtGroupVCPdu
	DtMimicTrackVCPdu
	DtMimicVCPdu
	DtModelessVCPdu
	DtOrbitVCPdu
	DtTetherTrackVCPdu
	DtTetherVCPdu
	DtTrackVCPdu
	DtUnknownVCPdu
	DtViewControlPdu
vlList.h	DtIntrusiveList
	DtList
	DtListItem
	DtListedObj
vlutil.h	DtSelectParam
vlVector.h	DtVector
vrlFedAmb.h	DtVrlFederateAmbassador
vrlFedTime.h	DtVrlFedTime
vrlRtiAmb.h	DtVrlRtiAmbassador

9.4. Other Header Files

Table 9-4 lists header files that do not contain classes.

Table 9-4: Other Header Files

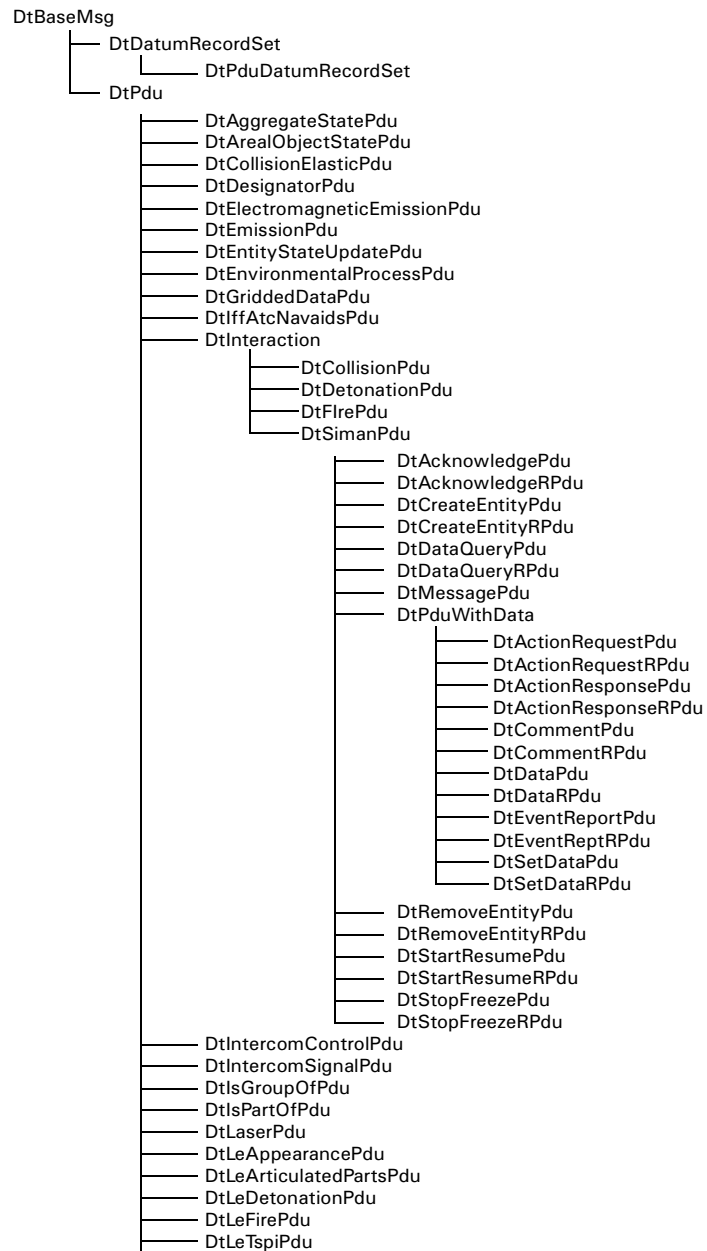
Header File	Contents
datumIdType.h	Definitions and functions that map from DtDatumParam to type enumerations.
disEnums.h	Contains enumerations specified by the DIS protocol and RPR FOM.
EntityTypes.h	Contains enumerations of entity-type fields.
enum2string.h	Contains functions that map from DtPduKind enumerations to strings.
Euler.h	Contains functions that convert between Euler angles and transformation matrices.
MachineTypes.h	Defines primitive data types in a machine-independent manner.
mathTypes.h	Contains math macros, includes other math-related headers.
printUtil.h	Defines printing functions.
ProcControl.h	Defines functions for sleeping and polling stdin.
topoCoord.h	Defines functions to generate matrices that convert between geodetic and topographic orientation.
vlprint.h	Contains functions related to the printing of notification messages.
vlTime.h	Contains functions related to time.

9.5. Class Hierarchy Diagrams and Tables

This section contains class hierarchy diagrams as follows:

- ♦ [Figure 9-1, DIS PDU Class Hierarchy](#)
- ♦ [Figure 9-2, HLA State and Interaction Class Hierarchy](#)
- ♦ [Table 9-5, All VR-Link classes](#)

Figure 9-1. DIS PDU Class Hierarchy



(continued on next page)

Figure 9-1 continued

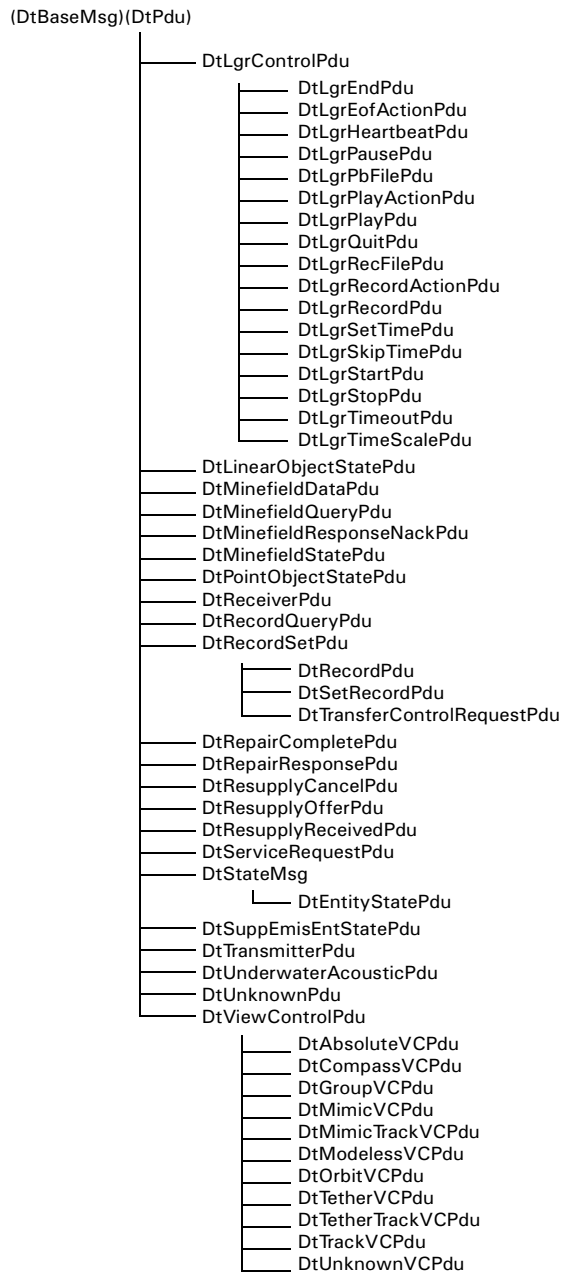


Figure 9-2. HLA State and Interaction Class Hierarchy

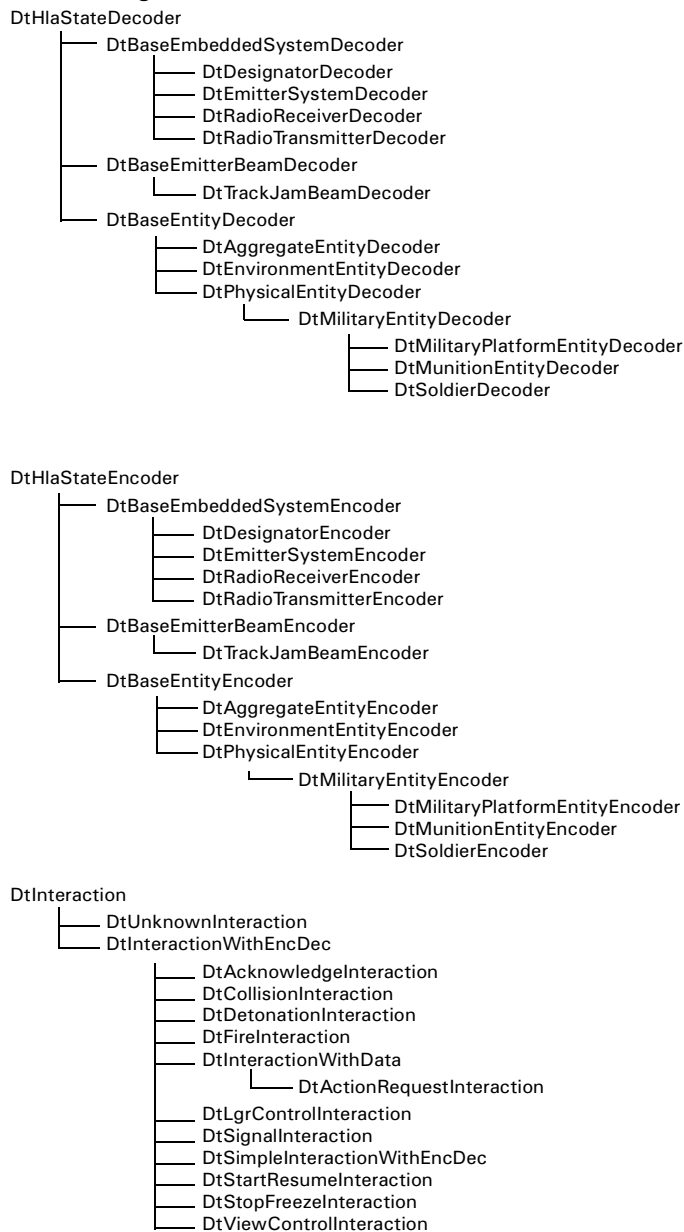


Table 9-5 lists all VR-Link classes, sorted on the first column, with the classes from which they are derived listed to the right to the base class. Classes that are not derived from another class are listed in the first column.

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes			
DtAbsNotifyObserver					
DtAbsoluteVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg	
DtAcknowledgeDecoder	DtInteractionDecoder				
DtAcknowledgeEncoder	DtInteractionEncoder				
DtAcknowledgeInteraction	DtInteractionWith-EncDec	DtInteraction	DtPdu	DtBaseMsg	
DtAcknowledgePdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg	
DtAcknowledgeRPdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg	
DtAckPredOpt					
DtActionRequestDecoder	DtInteractionDecoder				
DtActionRequestEncoder	DtInteractionEncoder				
DtActionRequestInteraction	DtInteractionWithData	DtInteractionWithEncDec	DtInteraction	DtPdu	DtBaseMsg
DtActionRequestPdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtActionRequestRPdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtActionResponsePdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtActionResponseRPdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtAggregateCallbackInfo	DtListedObj				
DtAggregateEntityDecoder	DtBaseEntityDecoder	DtHlaStateDecoder			
DtAggregateEntityEncoder	DtBaseEntityEncoder	DtHlaStateEncoder			
DtAggregatePublisher (DIS)					
DtAggregatePublisher (HLA)	DtObjectPublisher				
DtAggregateStateDecoder					
DtAggregateStateEncoder					
DtAggregateStatePdu	DtPdu	DtBaseMsg			
DtAggregateStateRepository	DtBaseEntityStateRepository		DtStateRepository		
DtAntennaPatternRecord					
DtApproximator					
DtAppSpecificRadioSignalDecoder		DtInteractionDecoder			
DtAppSpecificRadioSignalEncoder		DtInteractionEncoder			
DtArealObjectStatePdu	DtPdu	DtBaseMsg			
DtArtPart					
DtArtPartList					
DtArtPartSpec					

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes				
DtArtPartThresholder						
DtAttachedPartList						
DtBaseEmbeddedSystemDecoder		DtHlaStateDecoder				
DtBaseEmbeddedSystemEncoder		DtHlaStateEncoder				
DtBaseEmitterBeamDecoder	DtHlaStateDecoder					
DtBaseEmitterBeamEncoder	DtHlaStateEncoder					
DtBaseEntityDecoder	DtHlaStateDecoder					
DtBaseEntityEncoder	DtHlaStateEncoder					
DtBaseEntityStateRepository	DtStateRepository					
DtBaseExerciseConn						
DtBaseHashKey						
DtBaseMsg						
DtBaseReflectedAggregate	DtListedObj					
DtBaseReflectedEntity	DtListedObj					
DtBasicFixedWingAircraftRecord						
DtBasicGroundCombatSoldierRecord						
DtBasicGroundCombatVehicleRecord						
DtBasicRotorWingAircraftRecord						
DtBeamAntennaPatternRecord		DtAntennaPatternRecord				
DtBurstDescriptor						
DtCartesianView						
DtClientSocket	DtSocket					
DtCollisionDecoder	DtInteractionDecoder					
DtCollisionElasticPdu	DtPdu	DtBaseMsg				
DtCollisionEncoder	DtInteractionEncoder					
DtCollisionInteraction	DtInteractionWith-EncDec	DtInteraction	DtPdu	DtBaseMsg		
DtCollisionPdu	DtInteraction	DtPdu	DtBaseMsg			
DtCommentPdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg	
DtCommentRPdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg	
DtCompassVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg		
DtCoordTransform						
DtCoordTranslation						
DtCreateEntityPdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg		
DtCreateEntityRPdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg		
DtCsFilterTest	DtFilterTest					
DtDatabaseIndexRadioSignalDecoder		DtInteractionDecoder				
DtDatabaseIndexRadioSignalEncoder		DtInteractionEncoder				

Table 9-5: All VR-Link classes

Derived classes ...			... Base classes		
DtDataPdu	DtPduWithData	DtSimanPdu	DtInterac- tion	DtPdu	DtBaseMs g
DtDataQueryPdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseM sg	
DtDataQueryRPdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseM sg	
DtDataRPdu	DtPduWithData	DtSimanPdu	DtInterac- tion	DtPdu	DtBaseMs g
DtDatumMap					
DtDatumMapEntry					
DtDatumRecordSet	DtBaseMsg				
DtDcm					
DtDeadReckoner	DtApproximator				
DtDesignatorCallbackInfo	DtListedObj				
DtDesignatorDecoder	DtBaseEmbeddedSystemDecoder		DtHlaStateDecoder		
DtDesignatorEncoder	DtBaseEmbeddedSystemEncoder		DtHlaStateEncoder		
DtDesignatorPdu	DtPdu	DtBaseMsg			
DtDesignatorPublisher	DtObjectPublisher				
DtDesignatorRepository	DtEmbeddedSystemRepository		DtStateRepository		
DtDetonationDecoder	DtInteractionDecoder				
DtDetonationEncoder	DtInteractionEncoder				
DtDetonationInteraction	DtInteractionWith- EncDec	DtInteraction	DtPdu	DtBaseM sg	
DtDetonationPdu	DtInteraction	DtPdu	DtBaseMsg		
DtDirectedRotRate					
DtDirectRtiAmbassador	DtVrIRtiAmbassador				
DtDiscoverObjectCbInfo					
DtDisPduFactory	DtPduFactory				
DtDmeGlobalFcns	DtDatumMapEntry				
DtElectromagneticEmissionPdu		DtPdu	DtBaseMsg		
DtEmbeddedSystemRepository		DtStateRepository			
DtEmissionPdu	DtPdu	DtBaseMsg			
DtEmitterBeamCallback- Info	DtListedObj				
DtEmitterBeamPublisher	DtObjectPublisher				
DtEmitterBeamRepository	DtStateRepository				
DtEmitterSystemCallbackInfo		DtListedObj			
DtEmitterSystemDecoder	DtBaseEmbeddedSystemDecoder		DtHlaStateDecoder		
DtEmitterSystemEncoder	DtBaseEmbeddedSystemEncoder		DtHlaStateEncoder		
DtEmitterSystemPublisher	DtObjectPublisher				
DtEmitterSystemReposi- tory	DtEmbeddedSystemRepository		DtStateRepository		
DtEncodedAudioRadioSignalDecoder		DtInteractionDecoder			
DtEncodedAudioRadioSignalEncoder		DtInteractionEncoder			
DtEnhancedFixedWingAircraftRecord					
DtEnhancedGroundCombatSoldierRecord					

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes			
DtEnhancedGroundCombatVehicleRecord					
DtEnhancedRotorWingAircraftRecord					
DtEntityCallbackInfo	DtListedObj				
DtEntityIdentifier					
DtEntityIdHashlist	DtOldIntrusiveHashlist				
DtEntityPublisher (DIS)					
DtEntityPublisher (HLA)	DtObjectPublisher				
DtEntityStateDecoder					
DtEntityStateEncoder					
DtEntityStatePdu	DtStateMsg	DtPdu		DtBaseMsg	
DtEntityStateRepository	DtBaseEntityStateRepository			DtStateRepository	
DtEntityStateUpdatePdu	DtPdu	DtBaseMsg			
DtEntityType					
DtEnvironmentalProcessPdu	DtPdu	DtBaseMsg			
DtEnvironmentEntityDecoder		DtBaseEntityDecoder		DtHlaStateDecoder	
DtEnvironmentEntityEncoder		DtBaseEntityEncoder		DtHlaStateDecoder	
DtEnvironmentTypeRecord	DtEntityType				
DtEventId					
DtEventReportPdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtEventReptRPdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtExerciseConn	DtBaseExerciseConn				
DtFederateId					
DtFilename	DtString				
DtFilterTest					
DtFireDecoder	DtInteractionDecoder				
DtFireEncoder	DtInteractionEncoder				
DtFireInteraction	DtInteractionWith-EncDec	DtInteraction	DtPdu	DtBaseMsg	
DtFirePdu	DtInteraction	DtPdu	DtBaseMsg		
DtFom	DtFedTarget				
DtFomMapper					
DtFundamentalParameterData					
DtGeodeticCoord	DtVector				
DtGlobalIdHashlist					
DtGlobalObjectDesignatorList					
DtGridAxisRecordFixed					
DtGridAxisRecordVariable	DtGridAxisRecordFixed				
DtGridDataRecord					
DtGridDataRecordType0	DtGridDataRecord				
DtGridDataRecordType1	DtGridDataRecord				
DtGridDataRecordType2	DtGridDataRecord				
DtGriddedDataPdu	DtPdu	DtBaseMsg			
DtGroundLogisticsVehicleRecord					

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes		
DtGroupVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg
DtHashedItem				
DtHashlist				
DtHlaObject				
DtHlaObjectFactory				
DtHlaObjectWithStateRep	DtHlaObject			
DtHlaStateDecoder				
DtHlaStateEncoder				
DtIffAtcNav aidsBeamData				
DtIffAtcNav aidsLayer-Header				
DtIffAtcNav aidsOperational				
DtIffAtcNav aidsParameter				
DtIffAtcNav aidsPdu	DtPdu	DtBaseMsg		
DtIffAtcNav aidsSecondOperational				
DtIffAtcNav aidsSystemID				
DtInteraction (HLA)				
DtInteraction (DIS)	DtPdu	DtBaseMsg		
DtInteractionDecoder				
DtInteractionDecoderFactory				
DtInteractionEncoder				
DtInteractionEncoderFactory				
DtInteractionFactory				
DtInteractionManager				
DtInteractionWithData	DtInteractionWith-EncDec	DtInteraction	DtPdu	DtBaseMsg
DtInteractionWithEncDec	DtInteraction	DtPdu	DtBaseMsg	
DtInterClassDesc				
DtInterClassMapping				
DtIntercomControlPdu	DtPdu	DtBaseMsg		
DtIntercomEntityDestRec	DtIntercomParmRec			
DtIntercomGroupAssignmentRec	DtIntercomParmRec			
DtIntercomGroupDestRec	DtIntercomParmRec			
DtIntercomParmRec				
DtIntercomSignalPdu	DtPdu	DtBaseMsg		
DtIntHashKey	DtBaseHashKey			
DtIntrusiveList	DtList			
DtIsGroupOfPdu	DtPdu	DtBaseMsg		
DtIsPartOfPdu	DtPdu	DtBaseMsg		
DtLaserPdu	DtPdu	DtBaseMsg		
DtLeAppearancePdu	DtPdu	DtBaseMsg		
DtLeArticulatedPartsPdu	DtPdu	DtBaseMsg		

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes		
DtLeDeadReckoningParameters				
DtLeDetonationPdu	DtPdu	DtBaseMsg		
DtLeEntityID				
DtLeEntityLinearVelocity				
DtLeEntityLocationEntityCoordinates				
DtLeEntityLocationWorldCoordinates				
DtLeEntityOrientation				
DtLeFirePdu	DtPdu	DtBaseMsg		
DtLeOrientationError				
DtLePositionError				
DtLeTspiPdu	DtPdu	DtBaseMsg		
DtLgrControlDecoder	DtInteractionDecoder			
DtLgrControlEncoder	DtInteractionEncoder			
DtLgrControlInteraction	DtInteractionWith-EncDec	DtInteraction	DtPdu	DtBaseMsg
DtLgrControlPdu (DIS)	DtInteraction	DtPdu	DtBaseMsg	
DtLgrControlPdu (HLA)	DtPdu	DtBaseMsg		
DtLgrEndPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrEofActionPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrHeartbeatPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrPausePdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrPbFilePdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrPlayActionPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrPlayPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrQuitPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrRecFilePdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrRecordActionPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrRecordPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrSetTimePdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrSkipTimePdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrStartPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrStopPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrTimeoutPdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLgrTimeScalePdu	DtLgrControlPdu	DtPdu	DtBaseMsg	
DtLinearObjectStatePdu	DtPdu	DtBaseMsg		
DtLinearSegmentParameter				
DtList				
DtListedObj				
DtListItem				
DtLocalHlaObjectManager				
DtMessagePdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtMilitaryEntityDecoder	DtPhysicalEntityDecoder	DtBaseEntityDecoder		DtHlaStateDecoder

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes		
DtMilitaryEntityEncoder	DtPhysicalEntityEncoder	DtBaseEntityEncoder		DtHlaStateEncoder
DtMilitaryPlatformEntityDecoder	DtMilitaryEntityDecoder	DtPhysicalEntityDecoder	DtBaseEntityDecoder	DtHlaStateDecoder
DtMilitaryPlatformEntityEncoder	DtMilitaryEntityEncoder	DtPhysicalEntityEncoder	DtBaseEntityEncoder	DtHlaStateEncoder
DtMimicTrackVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg
DtMimicVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg
DtMinefieldDataPdu	DtPdu	DtBaseMsg		
DtMinefieldPerimeterRecord				
DtMinefieldQueryPdu	DtPdu	DtBaseMsg		
DtMinefieldResponseNackPdu		DtPdu	DtBaseMsg	
DtMinefieldStatePdu	DtPdu	DtBaseMsg		
DtModelessVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg
DtModulationType				
DtMunitionEntityDecoder	DtMilitaryEntityDecoder	DtPhysicalEntityDecoder	DtBaseEntityDecoder	DtHlaStateDecoder
DtMunitionEntityEncoder	DtMilitaryEntityEncoder	DtPhysicalEntityEncoder	DtBaseEntityEncoder	DtHlaStateEncoder
DtNet32Vector				
DtNet64Vector				
DtNetBoolean				
DtNetBurstDescriptor				
DtNetEntityIdentifier				
DtNetEntityMarking				
DtNetEntityType				
DtNetEulerAngles				
DtNetEventId				
DtNetFederateId				
DtNetFloat32				
DtNetFloat64				
DtNetGlobalObjectDesignator				
DtNetInt16				
DtNetInt32				
DtNetRadioEntityType				
DtNetSimulationAddress				
DtNetSocket	DtSocket			
DtNetTimeStamp				
DtNetU16				
DtNetU32				

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes			
DtObjClassDesc					
DtObjectId					
DtObjectIdHashKey	DtBaseHashKey				
DtObjectIdHashlist					
DtObjectPublisher					
DtObjectReflectAttributeValuesCbInfo					
DtObjectType					
DtOldIntrusiveHashlist	DtOldHashlist				
DtOrbitVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg	
DtPdu	DtBaseMsg				
DtPduCallbackInfo					
DtPduCallbackManager					
DtPduDatumRecordSet	DtDatumRecordSet	DtBaseMsg			
DtPduFactory					
DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg	
DtPhysicalEntityDecoder	DtBaseEntityDecoder	DtHlaStateDecoder			
DtPhysicalEntityEncoder	DtBaseEntityEncoder	DtHlaStateEncoder			
DtPointObjectStatePdu	DtPdu	DtBaseMsg			
DtPropulsionSystem					
DtRadioEntityType	DtEntityType				
DtRadioReceiverCallbackInfo		DtListedObj			
DtRadioReceiverDecoder	DtBaseEmbeddedSystemDecoder	DtHlaStateDecoder			
DtRadioReceiverEncoder	DtBaseEmbeddedSystemEncoder	DtHlaStateEncoder			
DtRadioReceiverPublisher	DtObjectPublisher				
DtRadioReceiverRepository	DtEmbeddedSystemRepository	DtStateRepository			
DtRadioTransmitterCallbackInfo		DtListedObj			
DtRadioTransmitterDecoder	DtBaseEmbeddedSystemDecoder	DtHlaStateDecoder			
DtRadioTransmitterEncoder	DtBaseEmbeddedSystemEncoder	DtHlaStateEncoder			
DtRadioTransmitterPublisher	DtObjectPublisher				
DtRadioTransmitterRepository		DtEmbeddedSystemRepository	DtStateRepository		
DtRawBinaryRadioSignalDecoder		DtInteractionDecoder			
DtRawBinaryRadioSignalEncoder		DtInteractionEncoder			
DtReceiveInteractionCbInfo					
DtReceiverPdu	DtPdu	DtBaseMsg			
DtRecordPdu	DtRecordSetPdu	DtPdu	DtBaseMsg		
DtRecordQueryPdu	DtPdu	DtBaseMsg			
DtRecordSet					
DtRecordSetPdu	DtPdu	DtBaseMsg			
DtReflectAttributeValuesCbInfo					
DtReflectAttributeValuesCbList					

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes		
DtReflectedAggregate				
DtReflectedAggregate (HLA)	DtReflectedObject	DtListedObject		
DtReflectedAggregate (DIS)	DtListedObject			
DtReflectedAggregateList (DIS)				
DtReflectedAggregateList (HLA)		DtReflectedObjectList		
DtReflectedDesignator	DtReflectedObject	DtListedObject		
DtReflectedDesignatorList	DtReflectedObjectList			
DtReflectedEmitterBeam	DtReflectedObject	DtListedObject		
DtReflectedEmitterBeam-List	DtReflectedObjectList			
DtReflectedEmitterSystem	DtReflectedObject	DtListedObject		
DtReflectedEmitterSystemList		DtReflectedObjectList		
DtReflectedEntity (HLA)	DtReflectedObject	DtListedObject		
DtReflectedEntity (DIS)	DtListedObject			
DtReflectedEntityHashlist	DtOldIntrusiveHashlist			
DtReflectedEntityList (DIS)				
DtReflectedEntityList (HLA)	DtReflectedObjectList			
DtReflectedHlaObjectManager				
DtReflectedObject	DtListedObject			
DtReflectedObjectCallbackInfo		DtListedObject		
DtReflectedObjectList				
DtReflectedRadioReceiver	DtReflectedObject	DtListedObject		
DtReflectedRadioReceiverList		DtReflectedObjectList		
DtReflectedRadioTransmitter	DtReflectedObject	DtListedObject		
DtReflectedRadioTransmitterList		DtReflectedObjectList		
DtRegControlCbInfo				
DtRegControlCbList				
DtRemoveEntityPdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtRemoveEntityRPdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtRemoveObjectCbInfo				
DtRepairCompletePdu	DtPdu	DtBaseMsg		
DtRepairResponsePdu	DtPdu	DtBaseMsg		
DtResupplyCancelPdu	DtPdu	DtBaseMsg		
DtResupplyOfferPdu	DtPdu	DtBaseMsg		
DtResupplyReceivedPdu	DtPdu	DtBaseMsg		
DtRprFomMapper	DtFomMapper			
DtRprHlaObjectFactory	DtHlaObjectFactory			
DtRprInteractionDecoderFactory		DtInteractionDecoderFactory		
DtRprInteractionEncoderFactory		DtInteractionEncoderFactory		
DtRprInteractionFactory	DtInteractionFactory			

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes			
DtRprStateDecoderFactory	DtStateDecoderFactory				
DtRprStateEncoderFactory	DtStateEncoderFactory				
DtScript	DtList				
DtSelectParam					
DtServiceRequestPdu	DtPdu	DtBaseMsg			
DtSetDataPdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtSetDataRPdu	DtPduWithData	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg
DtSetRecordPdu	DtRecordSetPdu	DtPdu	DtBaseMsg		
DtSignalInteraction	DtInteractionWith-EncDec	DtInteraction	DtPdu	DtBaseMsg	
DtSignalPdu	DtInteraction	DtPdu	DtBaseMsg		
DtSilentAggregateList					
DtSilentEntityList					
DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg		
DtSimpleInteractionWith-EncDec	DtInteractionWith-EncDec	DtInteraction	DtPdu	DtBaseMsg	
DtSmoother	DtDeadReckoner	DtApproximator			
DtSmoothOrient					
DtSmoothOriWithDr	DtSmoothOrient				
DtSmoothPos					
DtSmoothPosWithDr	DtSmoothPos				
DtSocket					
DtSocketPairSocket	DtSocket				
DtSoldierDecoder	DtMilitaryEntityDecoder	DtPhysicalEntityDecoder	DtBaseEntityDecoder	DtHlaStateDecoder	
DtSoldierEncoder	DtMilitaryEntityEncoder	DtPhysicalEntityEncoder	DtBaseEntityEncoder	DtHlaStateEncoder	
DtStartResumeDecoder	DtInteractionDecoder				
DtStartResumeEncoder	DtInteractionEncoder				
DtStartResumeInteraction	DtInteractionWith-EncDec	DtInteraction	DtPdu	DtBaseMsg	
DtStartResumePdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg	
DtStartResumeRPdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg	
DtStartStopInterCbInfo					
DtStartStopInterCbList					
DtStartUpdatesCbInfo					
DtStartUpdatesCbList					
DtStateDecoderFactory					
DtStateEncoderFactory					
DtStateMsg (HLA)					
DtStateMsg (DIS)	DtPdu	DtBaseMsg			

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes			
DtStateRepository					
DtStopFreezeDecoder	DtInteractionDecoder				
DtStopFreezeEncoder	DtInteractionEncoder				
DtStopFreezeInteraction	DtInteractionWith-EncDec	DtInteraction	DtPdu	DtBaseMsg	
DtStopFreezePdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg	
DtStopFreezeRPdu	DtSimanPdu	DtInteraction	DtPdu	DtBaseMsg	
DtStopUpdatesCbInfo					
DtStopUpdatesCbList					
DtString					
DtStringHashKey	DtBaseHashKey				
DtStringHashlist	DtOldIntrusiveHashlist				
DtSuppEmisEntStatePdu	DtPdu	DtBaseMsg			
DtTaitBryan					
DtTetherTrackVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg	
DtTetherVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg	
DtThresholder					
DtTimeStampType					
DtTopoView	DtCartesianView				
DtTrackJamBeamDecoder	DtBaseEmitterBeamDecoder			DtHlaStateDecoder	
DtTrackJamBeamEncoder	DtBaseEmitterBeamEncoder			DtHlaStateEncoder	
DtTrackJamTarget					
DtTrackVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg	
DtTransferControlRequestPdu		DtRecord-SetPdu	DtPdu	DtBaseMsg	
DtTransmitterPdu	DtPdu	DtBaseMsg			
DtUnderwaterAcousticAddtnlPassiveAct					
DtUnderwaterAcoustic-Beam					
DtUnderwaterAcousticEmitterSystem					
DtUnderwaterAcousticPdu	DtPdu	DtBaseMsg			
DtUnderwaterAcousticShaftSpeed					
DtUnknownHlaObject	DtHlaObject				
DtUnknownInteraction	DtInteraction	DtPdu	DtBaseMsg		
DtUnknownPdu	DtPdu	DtBaseMsg			
DtUnknownVCPdu	DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg	
DtUpdateControlCbInfo					
DtUpdateControlCbList					
DtUtmCoord	DtVector				
DtUtmView					

Table 9-5: All VR-Link classes

Derived classes ...		... Base classes		
DtVarDatumSpec				
DtVariableDatumRecord				
DtVector				
DtVectoringNozzleSystem				
DtViewControlDecoder	DtInteractionDecoder			
DtViewControlEncoder	DtInteractionEncoder			
DtViewControlInteraction	DtInteractionWith- EncDec	DtInteraction	DtPdu	DtBaseM sg
DtViewControlPdu	DtInteraction	DtPdu	DtBaseMsg	
DtViewControlPdu	DtPdu	DtBaseMsg		
DtVrIFederateAmbassador	RTI::NullFederateAmbassador			
DtVrIFedTime	RTI::FedTime			
DtVrIRtiAmbassador				



PDU Class Reference

This appendix contains descriptive notes pertaining to DIS 2.0.3, 2.0.4, IEEE1278.1 and 2.1.4 PDUs.

<i>DtAcknowledgePdu</i>	A-2	<i>DtPduWithData</i>	A-44
<i>DtActionRequestPdu</i>	A-4	<i>DtReceiverPdu</i>	A-51
<i>DtActionResponsePdu</i>	A-6	<i>DtRemoveEntityPdu</i>	A-52
<i>DtCollisionPdu</i>	A-8	<i>DtRepairCompletePdu</i>	A-53
<i>DtCommentPdu</i>	A-10	<i>DtRepairResponsePdu</i>	A-54
<i>DtCreateEntityPdu</i>	A-12	<i>DtResupplyCancelPdu</i>	A-55
<i>DtDataPdu</i>	A-13	<i>DtResupplyOfferPdu</i>	A-56
<i>DtDataQueryPdu</i>	A-15	<i>DtResupplyReceivedPdu</i>	A-58
<i>DtDesignatorPdu</i>	A-17	<i>DtSetDataPdu</i>	A-60
<i>DtDetonationPdu</i>	A-19	<i>DtSignalPdu</i>	A-62
<i>DtEmissionPdu</i>	A-22	<i>DtSimanPdu</i>	A-64
<i>DtEntityStatePdu</i>	A-27	<i>DtStartResumePdu</i>	A-66
<i>DtEventReportPdu</i>	A-32	<i>DtStopFreezePdu</i>	A-68
<i>DtFirePdu</i>	A-34	<i>DtTransmitterPdu</i>	A-70
<i>DtLaserPdu</i>	A-36	<i>DtViewControlPdu</i>	A-75
<i>DtMessagePdu</i>	A-38		

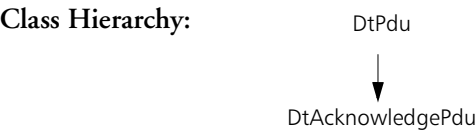
DtAcknowledgePdu

Name

DtAcknowledgePdu (DIS Acknowledge PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Constructors

Standard constructors derived from *DtPdu*:

- Blank PDU constructor
- From-network-representation constructor

Class-specific constructors:

- The constructor creates a blank *DtAcknowledgePdu* using **buffer**, then sets the originating entity ID to “from”, the receiving entity ID to “to”, the acknowledge flag to “aFlag” and the response flag to “rFlag”.

Functions

Table 1-1: DtAcknowledgePdu member functions(Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> (see “DtSimanPdu” on page A-64)
acknowledgeFlag() responseFlag()	return the values of the flags.
setAcknowledge-Flag()	sets the acknowledge flag to “flag”.
setResponseFlag()	sets the response flag to “flag”.

Table 1-1: DtAcknowledgePdu member functions(Sheet 2 of 2)

Function	Description
requestId() setRequestId()	inspect and set the request ID field.

Note: requestId() and setRequestId() are unavailable when linking with *libv13*.

See Also

[DtPdu](#), [DtSimanPdu](#), and this PDU's header file ([ackPdu.h](#)).

DtActionRequestPdu

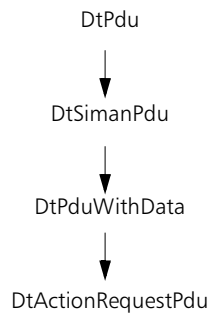
Name

DtActionRequestPdu (DIS Action Request PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-2: DtActionRequestPdu member functions (Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> . See “DtSimanPdu” on page A-64

Table 1-2: DtActionRequestPdu member functions (Sheet 2 of 2)

Function	Description
numFixedFields() numVarFields() setNumFixedFields() setNumVarFields() setDatumParam() setVarDataBytes() initDataSection() datumSizeBits() datumSizeBytes() datumId() getIndex() fixedDatumId() varDatumId() datumVal() functions setDatumVal() functions	are inherited from <i>DtPduWithData</i> (see “DtPduWithData” on page A-44)
requestId() actionId() setRequestId() setActionId()	inspect and set their PDU fields

See Also

[DtPdu](#), [DtSimanPdu](#), [DtPduWithData](#), and this PDU’s header file ([actionPdu.h](#)).

DtActionResponsePdu

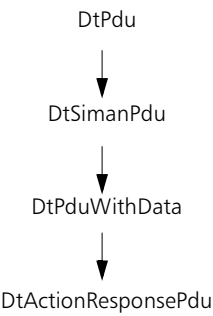
Name

DtActionResponsePdu (DIS Action Response PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-3: DtActionResponsePdu member functions (Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> . See “DtSimanPdu” on page A-64

Table 1-3: DtActionResponsePdu member functions (Sheet 2 of 2)

Function	Description
numFixedFields() numVarFields() setNumFixedFields() setNumVarFields() setDatumParam() setVarDataBytes() initDataSection() datumSizeBits() datumSizeBytes() datumId() getIndex() fixedDatumId() varDatumId() datumVal() functions setDatumVal() functions	are inherited from <i>DtPduWithData</i> (see “DtPduWithData” on page A-44)
requestId() requestStatus() setRequestId() setRequestStatus()	inspect and set their PDU fields

See Also

[DtPdu](#), [DtPduWithData](#), [DtSimanPdu](#), and this PDU’s header file ([actionPdu.h](#)).

DtCollisionPdu

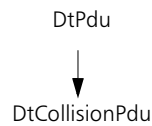
Name

DtCollisionPdu (DIS Collision PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-4: DtCollisionPdu member functions

Function	Description
issueld()	return the values of the appropriate data.
collideld()	
eventld()	
velocity()	
mass()	
entityCoords()	
setIssueld()	
setCollideld()	
setEventld()	
setVelocity()	
setMass()	
setEntityCoords()	
collisionType()	inspect and set the value of the collision type field. This field was introduced in the IEEE 1278.1-1995 version of the DIS protocol, replacing 32 bits of padding.
setCollisionType()	

Notes:

- ♦ Since C and C++ do not allow assignment to arrays, you must use *DtVecCopy* for copying the values returned by *velocity()* and *entityCoords()* into local variables:

```
DtCollisionPdu pdu;  
DtVector vec;  
vec = pdu.velocity();           /* ILLEGAL!!! */  
DtVecCopy(pdu.velocity(), vec); /* CORRECT    */
```
- ♦ *collisionType()* and *setCollisionType()* are not available when using DIS 2.0.3 (linking with *libvl3.a*). When linking with *libvl.a*, when the protocol version of a *DtCollisionPdu* is 4 (indicating DIS 2.0.4), these functions print a warning as they access a place in the PDU structure that is supposed to be padding in DIS 2.0.4.

See Also

DtPdu, and this PDU's header file (*collidePdu.h*).

DtCommentPdu

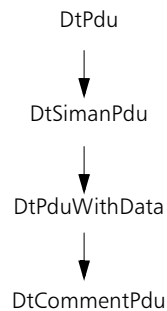
Name

DtCommentPdu (DIS Comment PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



The Comment PDU was introduced in DIS version 5 (1278.1-1995). It was previously known as the Message PDU. When using DIS 2.0.3 or 2.0.4, *DtMessagePdu* should be instead, as *DtCommentPdu* is unavailable.

Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Class-specific constructors:

- ♦ The constructor takes a sender ID, receiver ID, and a text message. The PDU created will have one variable datum record which contains the message.

Functions

Table 1-5: DtCommentPdu member functions (Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> . See “DtSimanPdu” on page A-64

Table 1-5: DtCommentPdu member functions (Sheet 2 of 2)

Function	Description
numFixedFields() numVarFields() setNumFixedFields() setNumVarFields() setDatumParam() setVarDataBytes() initDataSection() datumSizeBits() datumSizeBytes() datumId() getIndex() fixedDatumId() varDatumId() datumVal() functions setDatumVal() functions	are inherited from <i>DtPduWithData</i> (see “DtPduWithData” on page A-44)
setComment()	takes a text message and a size of this message as arguments. After a call to this function, the PDU has one variable length datum record that contains the message.

See Also

[DtPdu](#), [DtSimanPdu](#), [DtPduWithData](#), [DtMessagePdu](#), and this PDU’s header file ([commentPdu.h](#)).

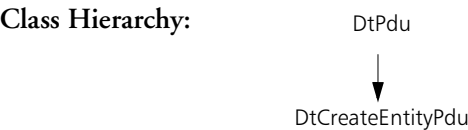
DtCreateEntityPdu

Name

DtCreateEntityPdu (DIS Create Entity PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Constructors

Standard constructors derived from *DtPdu*:

- Blank PDU constructor
- From-network-representation constructor

Class-specific constructors:

- The constructor creates a blank *DtCreateEntityPdu* using **buffer**, then sets the originating entity ID to “from”, and the receiving entity ID to “to”.

Functions

Table 1-6: DtCreateEntityPdu member functions

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> . See “DtSimanPdu” on page A-64
requestId() setRequestId()	inspect and set this field.

Note: requestId() and setRequestId() are unavailable when linking with *libv13*.

See Also

DtPdu, *DtSimanPdu*, and this PDU’s header file ([createPdu.h](#)).

DtDataPdu

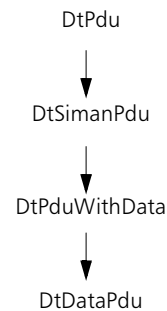
Name

DtDataPdu (DIS Data PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-7: DtDataPdu member functions (Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> . See “DtSimanPdu” on page A-64

Table 1-7: DtDataPdu member functions (Sheet 2 of 2)

Function	Description
numFixedFields() numVarFields() setNumFixedFields() setNumVarFields() setDatumParam() setVarDataBytes() initDataSection() datumSizeBits() datumSizeBytes() datumId() getIndex() fixedDatumId() varDatumId() datumVal() functions setDatumVal() functions	are inherited from <i>DtPduWithData</i> (see “DtPduWithData” on page A-44)
requestId() setRequestId()	inspect and set this field.

See Also

DtPdu, *DtSimanPdu*, *DtPduWithData*, and this PDU’s header file (*dataPdu.h*).

DtDataQueryPdu

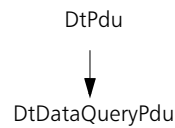
Name

DtDataQueryPdu (DIS Data Query PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-8: DtDataQueryPdu member functions (Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> . See “DtSimanPdu” on page A-64 for details.
requestId() timeInterval()	return the value of those data. The value returned by <code>timeInterval()</code> is in seconds.
numFixedFields()	returns the number of fixed datum IDs in the PDU.
numVarFields()	returns the number of variable datum IDs in the PDU.
fixedDatumId()	returns the <i>n</i> th fixed datum ID in the PDU, where <i>n</i> = <i>index</i> . <code>DtDatumParam(0)</code> is returned if <i>index</i> is either less than 1 or greater than the number of fixed datum IDs in the PDU.
varDatumId()	returns the <i>n</i> th variable datum ID in the PDU, where <i>n</i> = <i>index</i> . <code>DtDatumParam(0)</code> is returned if <i>index</i> is either less than 1 or greater than the number of variable datum IDs in the PDU.

Table 1-8: DtDataQueryPdu member functions (Sheet 2 of 2)

Function	Description
setRequestId() setTimeInterval()	set the values of those data in the PDU.
initDatumIds()	sets the number of fixed datum IDs in the PDU to numFixedFields and the number of variable datum IDs to numVarFields . It changes the size of the PDU to accommodate the new number of fixed and variable fields. It also sets all fixed and variable datum IDs to DtDatumParam(0).
setFixedDatumId()	sets the <i>n</i> th fixed datum ID to <i>id</i> , where <i>n</i> = index .
setVarDatumId()	sets the <i>n</i> th variable datum ID to <i>id</i> , where <i>n</i> = index .

See Also

[DtPdu](#), [DtSimanPdu](#), and this PDU's header file ([queryPdu.h](#)).

DtDesignatorPdu

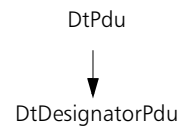
Name

DtDesignatorPdu (DIS Designator PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Note: This class is unavailable when linking with *libvl3*.

Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-9: DtDesignatorPdu member functions (Sheet 1 of 2)

Function	Description
designatingEntityId() codeName() designatedEntityId() designatorCode() power() wavelength() designatorSpotRelative() designatorSpotWorld()	return the values of the appropriate data.
setDesignatingEntityId() setCodeName() setDesignatedEntityId() setDesignatorCode() setPower() setWavelength() setDesignatorSpotRelative() setDesignatorSpotWorld()	set the values of the appropriate data.

Table 1-9: DtDesignatorPdu member functions (Sheet 2 of 2)

Function	Description
deadReckoningAlgorithm() setDeadReckoningAlgorithm() linearAcceleration() setLinearAcceleration()	inspect and set fields that were introduced in the IEEE 1278.1-1995 version of the DIS protocol.

Note: The dead reckoning and linear acceleration functions are not available when using DIS 2.0.3 (linking with *libvl3.a*). When linking with *libvl.a*, when the protocol version of a *DtCollisionPdu* is 4 (indicating DIS 2.0.4), these functions print a warning as they access fields that do not exist in DIS 2.0.4. In this case, `deadReckoningAlgorithm()` returns 0, and `linearAcceleration()` returns {0,0,0}. The corresponding mutator functions have no effect on the PDU.

See Also

[DtPdu](#), [DtLaserPdu](#), and this PDU's header file ([designatePdu.h](#)).

DtDetonationPdu

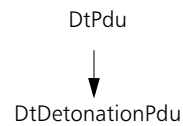
Name

DtDetonationPdu (DIS Detonation PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Class-specific constructors:

- ♦ The constructor creates a blank *DtDetonationPdu* using **buffer**, then calls *DtDetonationPdu* mutator functions to set the values of each of the data items specified as arguments. It also sets the number of articulation parameters and initializes those parameters by passing **specList** to **setArtParts()**.

Functions

Table 1-10: DtDetonationPdu member functions (Sheet 1 of 2)

Function	Description
attackerId()	return the values of the appropriate data.
targetId()	
munitionId()	
eventId()	
velocity()	
worldLocation()	
burst()	
entityLocation()	
result()	
numArtParams()	

Table 1-10: DtDetonationPdu member functions (Sheet 2 of 2)

Function	Description
artPartSpecs()	returns an array of <i>DtArtPartSpecs</i> , representing the articulation parameters contained in the PDU (up to 25 articulated parts). The array is an exact copy of the last array of <i>DtArtPartSpecs</i> with which the PDU was initialized (either through a call to <i>setArtParts()</i> , or by passing it to a <i>DtDetonationPdu</i> constructor).
getArtParamVal()	The two overloaded versions of <i>getArtParamVal()</i> fills value with the value of a particular articulation parameter. The first is used to select an articulation parameter by index (the <i>n</i> th parameter in the PDU, where <i>n</i> = paramIndex); the second is used to select a parameter by part type and parameter type. The first function returns zero if paramIndex is out of bounds (that is, less than 1 or greater than the number of parameters in this PDU), one otherwise. The second returns zero if no articulation parameter exists in this PDU with the specified part type and parameter type, otherwise it returns one.
setAttackerId() setTargetId() setMunitionId() setEventId() setVelocity() setWorldLocation() burst() setEntityLocation() setResult()	set the appropriate PDU data to the values passed as arguments.
setArtParts()	changes the size of the PDU and sets the number of articulation parameters to reflect specList . It also sets the part type, and part-attached-to fields of each articulation record appropriately.
setArtParamVal()	The two overloaded versions of <i>setArtParamVal()</i> set the value of a particular articulation parameter to value . Using the first, an articulation parameter is chosen by part type and parameter type. In the second, the <i>n</i> th articulation parameter is chosen, where <i>n</i> = paramIndex .

specList is an array of *DtArtPartSpecs*, each of which describes one articulated part (which may have several articulation parameters associated with it).

A *DtArtPartSpec* has the following form:

```
typedef struct
{
    int partType;
    int attachedType;
    int params[DtMaxNumArtParts];
} DtArtPartSpec;
```

where **partType** is the DIS enumeration for the part type, **attachedType** is the DIS enumeration for the parent part type, (or the token *DtAttachedToBase* if the part is attached to the entity base), and **params** is a NULL terminated list of articulation parameters that are to be used for this part.

The last element of the array of *DtArtPartSpecs* must be *DtNullArtPartSpec*, indicating that the previous *DtArtPartSpec* described the last articulated part. If **specList** is NULL, the list contains no articulated parts.

The array of *DtArtPartSpecs* can be defined statically, for example:

```
int numParts = 2;
static DtArtPartSpec specList[numParts +1] =
{
    {
        DtPrimaryTurret1,
        DtAttachedToBase,
        {DtApAzimuth, DtApAzimuthRate, 0}
    },
    {
        DtPrimaryGun1,
        DtPrimaryTurret1,
        {DtApElevation, 0}
    },
    {DtNullArtPartSpec}
};
```

See Also

[*DtPdu*](#), [*artPart.h*](#), and this PDU's header file ([*detonatePdu.h*](#)).

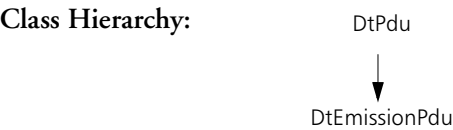
DtEmissionPdu

Name

DtEmissionPdu (DIS Emission PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Note: This class is only available when linking with *libvl3*. It is replaced by *DtElectromagneticEmissionPdu* in *libvl*.

Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-11: DtEmissionPdu member functions (Sheet 1 of 4)

Function	Description
emittingEntityId() eventId() stateUpdateIndicator() numSystems()	return the values of the appropriate data.
systemDataWords() numBeams() systemName() systemFunction() emitterId() location()	inspect various data associated with a particular emitter system record. The emitter system to inspect is dictated by the sysNum argument to each of these functions. Emitter system records are numbered 0 through numSystem() - 1. For the meaning of the retcode argument, see "Return Codes" on page A-26.

Table 1-11: DtEmissionPdu member functions (Sheet 2 of 4)

Function	Description
beamDataWords() beamId() beamParameterIndex() parameterData() beamFunction() numTargetsInBeam() trackingAllTargets() jammingMode Sequence()	inspect various data associated with a particular beam belonging to a particular emitter system (each emitter system record may contain one or more beam records). The particular emitter system record to inspect is dictated by the sysNum argument to each of these functions. The particular beam record within the specified emitter system record to inspect is dictated by the beamNum argument. Emitter system records are numbered 0 through numSystems() - 1. Beam records within a particular emitter system record sysNum are numbered 0 through numBeams(sysNum). For the meaning of the retcode argument, see “Return Codes” on page A-26.
trackJamTarget()	A particular emitter beam record may contain one or more track/jam targets records. The function trackJamTarget() is used to inspect a particular target record. sysNum and beamNum identify the particular emitter system and beam within that system whose target we would like to inspect. targetNum identifies a particular target within the beam record. Target records within a particular beam record indexed as described above by sysNum and beamNum , are indexed 0 through numTargetsInBeam(sysNum, beamNum). For the meaning of the retcode argument, see “Return Codes” on page A-26.
setEmittingEntityId() setEventId() setStateUpdate Indicator()	set the values of the appropriate fields.
setNumSystems()	resizes the PDU to accommodate num emitter system records. If num is less than the number of emitter system records previously in the PDU, trailing system records are deleted. Otherwise, additional emitter system records are created at the end of the PDU. The newly created emitter system records will each contain zero emitter beam records, and the system data size fields are set to reflect this.
setSystemName() setSystemFunction() setEmitterId() setLocation()	set the appropriate fields of the emitter system record indexed by sysNum . sysNum has the same meaning as in the corresponding inspector functions described above. For the meaning of the retcode argument, see “Return Codes” on page A-26.

Table 1-11: DtEmissionPdu member functions (Sheet 3 of 4)

Function	Description
setNumBeams()	resizes the PDU, and the emitter system record indexed by sysNum to accommodate the new number of beam records. If numBeams is less than the previous number of beams in the system record, trailing beam records are deleted. Otherwise, the appropriate number of new beam records are created following the last existing beam record. These new beam records will contains zeros in all fields, and will have no track/jam target records. The data sizes in the beam records, system records, and PDU header will all remain consistent after a call to setNumBeams(). For the meaning of the retcode argument, see “Return Codes” on page A-26.
setBeamId() setBeamParameter Index() setParameterData() setBeamFunction() setTrackingAllTargets() setJammingMode Sequence()	set the appropriate values with a beam records indexed by sysNum and beamNum (beamNum). sysNum and beamNum have the same meanings as in the corresponding inspector functions described above. For the meaning of the retcode argument, see “Return Codes” on page A-26. The <i>DtFundamentalParameterData</i> class is described above. The individual components of the object may be set directly before it is passed to setParameterData().
setNumTargets InBeam()	resizes the PDU, beam record, and emitter system record to accommodate the new number of track/jam target records. If num is less than the previous number of targets, trailing target records are deleted. Otherwise additional target records are inserted, following the last existing record. New track/jam target records created will have all values set to zero. The data sizes in the beam records, system records, and PDU header will all remain consistent after a call to setNumTargetsInBeam() . For the meaning of the retcode argument, see “Return Codes” on page A-26.
setTrackJamTarget()	sets the values of a particular track/jam target record to the data contained in targ . The <i>DtTrackJamTarget</i> class is described above. Its individual components may be set directly before it is passed to setTrackJamTarget(). sysNum , beamNum and targetNum have the same meanings as in trackJamTarget(). For the meaning of the retcode argument, see “Return Codes” on page A-26.

Table 1-11: DtEmissionPdu member functions (Sheet 4 of 4)

Function	Description
status()	returns one of the following values: DtSTATUS_OK if the values of all of the system data lengths, beam data lengths, number of targets and PDU length are all consistent. DtINCONSISTENT_SYS_LENGTHS_PDU_LENGTH if the system record data lengths are inconsistent with the PDU length. DtINCONSISTENT_BEAM_LENGTHS_SYS_LENGTH if for any emitter system record, the beam data lengths are inconsistent with the system data length. DtINCONSISTENT_NUM_TARGETS_BEAM_LENGTH if for any beam record, the number of targets is inconsistent with the beam record length.

The value returned by parameterData() is a reference to an object of type *DtFundamentalParameterData*. *DtFundamentalParameterData* has the following definition, and the values of the individual components can be inspected directly:

```
class DtFundamentalParameterData
{
public:
    float frequency;
    float frequencyRange;
    float effectiveRadiatedPower;
    float pulseRepetitionFrequency;
    float pulseWidth;
    float beamAzCenter;
    float beamAzSweep;
    float beamElCenter;
    float beamElSweep;
    float beamSweepSync;
};
```

The value returned by trackJamTarget() is a reference to an object of type *DtTrackJamTarget*. Its definition appears below, and individual components may be inspected directly:

```
class DtTrackJamTarget
{
public:
    DtEntityIdentifier entityId;
    int emitterId;
    int beamId;
};
```

Return Codes

Many inspector and mutator functions have an optional **retcode** argument. If it is non-NULL, the integer to which it points is set to indicate whether the inspection or mutation of the PDU was successful. For inspector functions, a return code of DtEMISSION_SUCCESS indicates that the value returned is valid. For mutator functions, a return code of DtEMISSION_SUCCESS indicates that the PDU has been successfully mutated. Any other value to which the integer to which **retcode** points might be set, indicates failure for one of the reasons cited below.

For functions that have a **sysNum** argument, DtBAD_SYS_NUM indicates that **sysNum** is not a valid emitter system record index for the PDU. DtBAD_SYS_ADDR indicates that although **sysNum** is within the legal range, system data lengths are inconsistent with the PDU length, and the specified emitter system records could not be found.

For functions with a **beamNum** argument, DtBAD_BEAM_NUM indicates that **sysNum** is a valid emitter system record index for the PDU but **beamNum** is not a valid beam record index for that emitter system. DtBAD_SYS_ADDR indicates that although **sysNum** and **beamNum** are within the legal ranges, and the specified emitter system record was found, the length of the data in the beam records is inconsistent with either the PDU length or the enclosing emitter system record length, and the specified beam record could not be found.

For functions with **sysNum**, **beamNum** and **targetNum** arguments, DtBAD_TARGET_NUM indicates that while **sysNum** and **beamNum** are valid, **targetNum** is not valid. DtBAD_SYS_ADDR indicates that although **sysNum**, **beamNum** and **targetNum** are within the legal ranges, and the specified beam record was found, the number of targets in this beam record is inconsistent with either the PDU length or the enclosing beam record length, and the specified target record could not be found.

See Also

[*DtPdu*](#), and this PDU's header file ([*emissionPdu.h*](#)).

DtEntityStatePdu

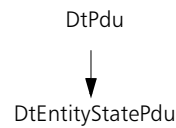
Name

DtEntityStatePdu (DIS Entity State PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Class-specific constructors:

- ♦ The constructors create a blank *DtEntityStatePdu* using **buffer**, then call *DtEntityStatePdu* mutator functions to set the values of each of the data items specified as arguments. For details on the meanings of each of the arguments, see the sections on the mutator functions *setId()*, *setEntityType()*, *setGuise()*, *setAlgorithm()*, *setForceId()*, and *setArtAndAttachedParts()* in this section.

After construction from a network representation, the status of the *DtEntityStatePdu* as reported by *DtPdu::status()* is *DtSTATUS_OK* unless the number of articulation parameters is inconsistent with the size of the PDU.

Functions

Table 1-12: DtEntityStatePdu member functions (Sheet 1 of 3)

Function	Description
id()	return the current values of those data.
forceld()	All of these functions return by value or reference, so their values can be copied into local objects as follows: <pre>DtEntityStatePdu pdu;</pre>
entityType()	
guise()	
orientation()	
appearance()	
algorithm()	
markingCharSet()	
capabilities()	Alternatively, the data can be used immediately, as in: <pre>DtEntityStatePdu pdu1; DtEntityStatePdu pdu2; pdu2.setEntityType(pdu1.entityType);</pre> or <pre>printf("Entity Type: %s\n", pdu1.entityType().string());</pre>
velocity()	return references to <i>DtConstVectors</i> (which is a <i>DtVector</i>) whose components cannot be modified. Because C and C++ prohibit assignment to arrays, the following is illegal: <pre>DtEntityStatePdu pdu1; DtVector vec = pdu.location();</pre> You must use <i>DtVecCopy</i> when you need to copy the value of one of these data into a local object: <pre>DtEntityStatePdu pdu1; DtVector vec; DtVecCopy(pdu.location(), vec);</pre> You can also use the returned value directly by passing it to any function that expects a <i>DtConstVector</i> or reference to one (actually, <i>DtVecCopy</i> is just such a function).
location()	
acceleration()	
rotationalVelocity()	
markingText()	returns a pointer to memory internal to <i>DtEntityStatePdu</i> . Do not save a pointer to this buffer, since it may be updated by the <i>DtEntityStatePdu</i> at any time, and will contain garbage after the <i>DtEntityStatePdu</i> is deleted. If the value of the marking text is not to be used immediately, use <code>strcpy</code> to copy it into a local buffer. The marking text will never be longer than 12 characters, and will always contain a trailing NULL character: <pre>char mark[12]; strcpy(mark, pdu.markingText());</pre>
appearance Attribute()	returns the current value of appearance attribute attr . A list of constants for many appearance attributes and values is in disEnums.h . For example: <pre>DtAppearAttrVal val = pdu.appearanceAttribute(DtAppHatch); if (val == DtHatchPopped) printf("Popped\n"); . .</pre>

Table 1-12: DtEntityStatePdu member functions (Sheet 2 of 3)

Function	Description
numArtParams()	returns the number of articulation parameter records in the PDU.
numAttached-Params()	returns the number of attached part records in the PDU.
numArtPlusAttachedParams()	returns the number of attached part records plus articulation parameter records in the PDU.
artPartSpecs()	returns an array of <i>DtArtPartSpecs</i> , representing the articulation parameters contained in the PDU (up to 25 articulated parts). The array is an exact copy of the last array of <i>DtArtPartSpecs</i> with which the PDU was initialized (either through a call to <i>setArtAndAttachedParts()</i> , or by passing it to a <i>DtEntityStatePdu</i> constructor).
attachedPartStations()	returns a null-terminated array of integers representing the attached part stations possessed by this entity. The array is an exact copy of the last array of station numbers with which the PDU was initialized (either through a call to <i>setArtAndAttachedParts()</i> or by passing it to the constructor).
getArtParamVal()	The two overloaded versions of <i>getArtParamVal()</i> fills value with the value of a particular articulation parameter. The first is used to select an articulation parameter by index (the <i>n</i>th parameter in the PDU, where <i>n</i> = paramIndex); the second is used to select a parameter by part type and parameter type. The first function returns zero if paramIndex is out of bounds (that is, less than 0 or greater than the number of parameters in this PDU), otherwise it returns one. The second returns zero if no articulation parameters exist in this PDU with the specified part type and parameter type, otherwise it returns one.
attachedPartChange()	returns the value of the change indicator in the attached part record associated with station . If there is no record with station station , DtNO_SUCH_ATTACH_STATION is returned.
getAttachedType()	fills type with the entity type in the type field of the attached part record whose station number is equal to station . If no such record exists, DtNO_SUCH_ATTACH_STATION is returned. Otherwise, DtATTACHED_PART_SUCCESS is returned.

Table 1-12: DtEntityStatePdu member functions (Sheet 3 of 3)

Function	Description
setId() setForceId() setEntityType() setGuise() setVelocity() setLocation() setOrientation() setAppearance() setAlgorithm() setAcceleration() setRotational Velocity() setMarking CharSet() setCapabilities()	set the values of their respective data to their arguments.
setAppearance Attribute()	sets the value of appearance attribute attr to val . A list of constants for many appearance attributes and values may be found in disEnums.h . For example: <pre>pdu.setAppearanceAttribute(DtAppHatch, DtHatchPopped);</pre>
setMarkingText()	sets the marking text to be the string contained in text and sets the character set to be charSet . The default value for charSet is 1.
setArtAndAttached Parts()	changes the size of the PDU and sets the number of articulation parameters plus attached part records to reflect specList and stationList . It also sets the part type, and part-attached-to fields of each articulation record appropriately, and sets the station number of each attached part record appropriately. See further discussion following this table.
setArtParamVal()	The two overloaded versions of setArtParamVal() set the value of a particular articulation parameter to value . Using the first, an articulation parameter is chosen by part type and parameter type. In the second, the n th articulation parameter is chosen, where n = paramIndex . The change argument indicates the value to be placed into the Change field of the articulation record. Passing the value DtINCREMENT (which is the default), indicates that the value of the Change field should be set to one greater than its current value.
setAttachedPart Change()	sets the value of the change indicator in the attached part record whose station is station to change . If no such record exists, DtNO_SUCH_ATTACH_STATION is returned, otherwise DtATTACHED_PART_SUCCESS is returned.
setAttachedPart Change()	sets the value of the type field in the attached part record whose station is station to type . If no such record exists, DtNO_SUCH_ATTACH_STATION is returned, otherwise DtATTACHED_PART_SUCCESS is returned.

stationList is a null terminated list of attached part station numbers.

specList is an array of *DtArtPartSpecs*, each of which describes one articulated part (which may have several articulation parameters associated with it).

A *DtArtPartSpec* has the following form:

```
typedef struct
{
    int partType;
    int attachedType;
    int params[DtMaxNumArtParts];
} DtArtPartSpec;
```

where **partType** is the DIS enumeration for the part type, **attachedType** is the DIS enumeration for the parent part type, (or the token *DtAttachedToBase* if the part is attached to the entity base), and **params** is a NULL terminated list of articulation parameters that are to be used for this part.

The last element of the array of *DtArtPartSpecs* must be *DtNullArtPartSpec*, indicating that the previous *DtArtPartSpec* described the last articulated part. If **specList** is NULL, the list contains no articulated parts.

The array of *DtArtPartSpecs* can be defined statically, for example:

```
int numParts = 2;
static DtArtPartSpec specList[numParts + 1] =
{
    {
        DtPrimaryTurret1,
        DtAttachedToBase,
        {DtApAzimuth, DtApAzimuthRate, 0}
    },
    {
        DtPrimaryGun1,
        DtPrimaryTurret1,
        {DtApElevation, 0}
    },
    {DtNullArtPartSpec}
};
```

See Also

[*DtPdu*](#), [*artPart.h*](#), and this PDU's header file ([*esPdu.h*](#)).

DtEventReportPdu

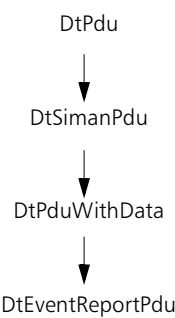
Name

DtEventReportPdu (DIS Event Report PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-13: DtEventReportPdu member functions (Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> . See “DtSimanPdu” on page A-64

Table 1-13: DtEventReportPdu member functions (Sheet 2 of 2)

Function	Description
numFixedFields() numVarFields() setNumFixedFields() setNumVarFields() setDatumParam() setVarDataBytes() initDataSection() datumSizeBits() datumSizeBytes() datumId() getIndex() fixedDatumId() varDatumId() datumVal() functions setDatumVal() functions	are inherited from <i>DtPduWithData</i> (see “DtPduWithData” on page A-44)
eventType() setEventType()	inspect and set that field in the PDU.

See Also

[DtPdu](#), [DtSimanPdu](#), and this PDU's header file ([eventReptPdu.h](#)).

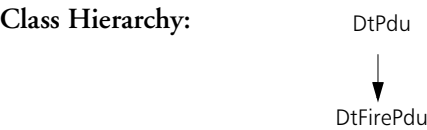
DtFirePdu

Name

DtFirePdu (DIS Fire PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Class-specific constructor:

The constructor creates a blank *DtFirePdu* using **buffer**, then calls *DtFirePdu* mutator functions to set the values of each of the data items specified as arguments.

Functions

Table 1-14: DtFirePdu member functions (Sheet 1 of 2)

Function	Description
attacker()	return the values of the appropriate data.
target()	
munition()	
eventId()	
location()	
burst velocity()	
range()	

Table 1-14: DtFirePdu member functions (Sheet 2 of 2)

Function	Description
setAttacker() setTarget() setMunition() setEventId() setLocation() setBurst() setVelocity() setRange()	set the appropriate PDU data to the values passed as arguments.
fireMissionIndex() setFireMission Index()	inspect and set the value of the fire mission field. This field was introduced in the IEEE 1278.1-1995 version of the DIS protocol, replacing 32 bits of padding.

Notes:

- Since C and C++ do not allow assignment to arrays, you must use *DtVecCopy* for copying the values returned by *location()* and *velocity()* into local variables:


```

DtFirePdu pdu;
DtVector vec;
vec = pdu.location();          /* ILLEGAL!!! */
DtVecCopy(pdu.location(), vec); /* CORRECT   */

```
- fireMissionIndex()* and *setFireMissionIndex()* are not available when using DIS 2.0.3 (linking with *libvl3.a*). When linking with *libvl.a*, when the protocol version of a *DtCollisionPdu* is 4 (indicating DIS 2.0.4), these functions print a warning as they access a place in the PDU structure that is supposed to be padding in DIS 2.0.4.

See Also

[*DtPdu*](#), and this PDU's header file ([*firePdu.h*](#)).

DtLaserPdu

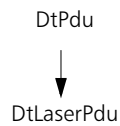
Name

DtLaserPdu (DIS Laser PDU)

Description

This class is only available when linking with *libvl3*. It is replaced by *DtDesignatorPdu* in *libvl*.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions**Table 1-15:** DtLaserPdu member functions

Function	Description
lasingEntityId() codeName() lasedEntityId() laserCode() power() wavelength() lasedSpotRelative() lasedSpotWorld()	return the values of the appropriate data.
setLasingEntityId() setCodeName() setLasedEntityId() setLaserCode() setPower() setWavelength() setLasedSpot Relative() setLasedSpot World()	set the values of the appropriate data.

See Also

[DtPdu](#), [DtDesignatorPdu](#), and this PDU's header file ([laserPdu.h](#)).

DtMessagePdu

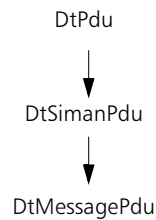
Name

DtMessagePdu (DIS Message PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



The *DtMessagePdu* class described here is available only when linking with *libvl3.a* (using DIS 2.0.3). In DIS version 5 (1278.1-1995) the Message PDU was replaced by the Comment PDU. When linking with *libvl.a*, and using DIS 2.0.4, *DtMessagePdu* is defined as:

```
typedef DtCommentPdu DtMessagePdu;
```

Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Class-specific constructors:

- ♦ The constructor can be used when the PDU is meant to convey a single text message. **message** is a null terminated character string, which is the message to be sent. The PDU will have a single variable length datum record whose length is equal to `strlen(message) + 1`. The datum parameter in the variable datum record is 0, or *DtNullDatumParam*.

Functions

Table 1-16: DtMessagePdu member functions

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> . See “DtSimanPdu” on page A-64 for details.
numVarFields()	returns the number of variable length datum fields in the PDU.
datumSizeBits()	returns the size of the <i>n</i> th variable datum field is returned, where <i>n</i> = Index . If Index is out of bounds, 0 is returned.
datumSizeBytes()	returns the value of <code>datumSizeBits()</code> , after converting to bytes.
setNumVarFields()	resizes the PDU to accommodate the new number of variable length datum records. If newNum is less than the previous number of records, trailing records are deleted. If newNum is greater than the previous number of records, additional records are created after the last existing record. The newly created records will have a data size of zero, and datum parameter ID of 0.
setDatumParam()	sets the <i>n</i> th variable datum ID in the PDU to param , where <i>n</i> = Index . Index should be between 1 and the number of variable datum fields in the PDU.
setVarDataBytes()	changes the data size in a particular variable length datum record. The PDU is resized to reflect the change. If the new value is less than the old, trailing bytes of the data are deleted. If the new value is greater than the old, bytes containing zeros are added after the last existing byte in the record. A PDU might be constructed and filled in as follows: <pre> DtMessagePdu messagePdu; messagePdu.setNumVarFields(3); messagePdu.setDatumParam(1, DtDatumXpos); messagePdu.setDatumParam(2, DtDatumYpos); messagePdu.setDatumParam(3, DtDatumZpos); messagePdu.setVarDatumBytes(1, 64); messagePdu.setVarDatumBytes(2, 64); messagePdu.setVarDatumBytes(3, 64); /* Now use the setDatumVal functions described below to set the actual datum values */ </pre>
setMessage()	can be used when the PDU is meant to contain only one variable length datum record, which contains a text message. The size of the field is set to size , and the data is set to message .

Inspecting and Setting Data Values

DtMessagePdu has many inspector and mutator functions that return and set the data value of a particular datum parameter respectively. The inspector functions' names start with "datumVal", and the mutator functions' names start with "setDatumVal()". The values of different datum parameters in a PDU can be of different data types. For each of the possible data types, we provide an inspector function that will return the value of a particular datum parameter, when that value is interpreted as belonging to that type, and a mutator function that sets the value of a particular datum parameter to a value of that type. The correct data type to be used for a particular datum parameter may be obtained using *DtDatumId2Type* (see "Data Types" on page A-41).

Particular datum records can be identified either by datum parameter ID, or by Index in the PDU. Indices range from 1 to numVarFields().

You can go back and forth between datum parameter IDs and indices using the functions datumId() and paramIndex(). paramIndex() will return the index of the variable datum record where the given datum parameter ID may be found. datumId() returns the *n*th variable datum ID in the PDU, where *n* = Index. *DtNullDatumParam* is returned when Index is less than 1 or greater than the number of variable datum fields in the PDU.

If no datum record with datum parameter ID *param* is found in the PDU, paramIndex() will return 0, and the integer to which *retcode* points is set to the value DtDATUM_BAD_ID (provided *retcode* is not NULL). If paramIndex() is successful, the value pointed to by *retcode* is set to DtDATUM_VALID.

Below are several examples of usage of the datumVal() inspectors, the setDatumVal() mutators, and the getIndex() function.

```
double xval = messagePdu.datumValFloat64(DtDatumXpos);
```

will assign to *xval* the value contained in the datum record of the PDU whose datum parameter ID is DtDatumXpos(), interpreting that value as a 64 bit floating point number.

An alternative way of doing the same thing is:

```
int Index, retcode;
/* first get the Index of the desired datum record */
Index = messagePdu.paramIndex(DtDatumXpos, &retcode);
/* now get the value contained in that datum record */
if (retcode == DtDATUM_VALID)
    double xval = messagePdu.datumValFloat64(Index);

messagePdu.setDatumValInt32(DtDatumKind, 12);
```

will set the value of the *kind* parameter to 12, using a 32 bit integer representation.

```
int Index, retcode;
/* first get the Index of the desired datum record */
Index = messagePdu.paramIndex(DtDatumMass, &retcode);
/* now set that datum record's value to 2000 */
if (retcode == DtDATUM_VALID)
```

```
{  
    messagePdu.setDatumValFloat32(Index, 2000.0);  
}
```

Note: Setting or inspecting datum values by `Index` is more efficient than by parameter ID (the latter type method first looks up the `Index` of the ID, then calls the former type of inspector (or mutator) with that `Index`). So if you plan to look up the value of the same ID more than once, it is more efficient to look up the `Index` once, then call the inspector with that `Index`.

Choosing among the different inspector and mutator functions for data types is fairly straightforward. However, the `datumValByteArray()` and `setDatumValByteArray()` functions can always be used to get a bitwise copy of a particular data value. The value returned by `datumValByteArray()` is a pointer to internal memory. If the value is not to be used immediately, the data should be copied into local memory:

```
{  
    char marking[12];  
    memcpy(marking,  
           messagePdu.datumValByteArray(Index, 12),  
           messagePdu.datumSizeBytes(Index));  
}
```

Return Codes

All of the datum value inspectors and mutators have an optional return code pointer argument, **retcode**. If this pointer is non-NULL, a return code for the function is stored in **retcode**. In all cases, the return code is `DtDATUM_VALID` when the function call was successful. This means that specified datum parameter record exists in the PDU, and its value was able to be interpreted as the data type whose inspector function was used, or set to a value of the data type whose mutator function was used.

The return code is set to `DtDATUM_BAD_LENGTH` when the length of the datum value field does not match the size of the data type whose function you are using.

The return code is set to `DtDATUM_BAD_INDEX` when **Index** is out of range.

The return code is set to `DtDATUM_BAD_ID` when no datum record exists in the PDU whose parameter ID is **param**.

Data Types

VR-Link maintains a table of mappings between *DtDatumIds* and data types. The function *DtDatumId2Type* is used to look up the data type associated with a particular datum ID.

The different data type values that may be returned by *DtDatumId2Type* are:

- ♦ DtTYPE_BYTE_ARRAY
- ♦ DtTYPE_FLOAT32
- ♦ DtTYPE_FLOAT64
- ♦ DtTYPE_INT32
- ♦ DtTYPE_UNSIGNED32
- ♦ DtTYPE_STRING.

If the table does not contain a mapping for *id*, then the value DtTYPE_NOT_DEFINED is returned.

All DIS 2.0.3 datum IDs are entered in this table by default along with reasonable data types. For instance, *DtDatumId2Type(DtDatumXpos)* will return DtTYPE_FLOAT64 while *DtDatumId2Type(DtDatumForceId)* will return DtTYPE_FLOAT32. This function is used by, among other things, *DtMessagePdu*'s implementation of the *DtPdu* virtual function *printData()*, which needs to know the type of each datum value, so it can properly format them for printing:

```
for (int i = 1; i <= numVarFields(); i++)
{
    DtDatumParam param = datumId(i);
    switch (DtDatumId2Type(param))
    {
        case DtTYPE_FLOAT32:
            DtDumpFloat(DtEnumString(param), datumValFloat32(i));
            break;
        case DtTYPE_FLOAT64:
            DtDumpFloat(DtEnumString(param), datumValFloat32(i));
            break;
        case DtTYPE_INT32:
            DtDumpInteger(DtEnumString(param),
                          datumValInt32(i));
            break;
        .
        .
        .
    }
}
```

The data type of a particular datum ID may be changed, or a mapping for a new datum ID may be added to the table, with the function *DtAddDatumId2TypeMapping()*. This function will return DtTYPE_NOT_DEFINED if there was no previous data type mapping for *id* in the table. Otherwise, it returns the old value. If the optional **warn** argument is non-zero, then a warning message is printed when an existing mapping is being overridden.

See Also

[*DtPdu*](#), [*DtSimanPdu*](#), [*DtPduWithData*](#), [*DtCommentPdu*](#), and this PDU's header file ([*messagePdu.h*](#)).

DtPduWithData

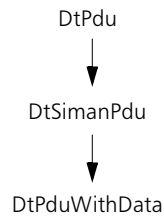
Name

DtPduWithData

Description

DtPduWithData is an abstract class that encapsulates the functionality common to all PDUs that contain fixed and variable length datum records. Specifically, *DtPduWithData* provides accessor functions to inspect and set the values in the “data” section of these PDUs.

Class Hierarchy:



After an instance of a class derived from *DtPduWithData* is constructed, inspecting and mutating PDU header data is done through *DtPdu* member functions. Similarly, inspecting and mutating of data common to all simulation management PDUs is done through *DtSimanPdu* member functions, and inspecting and mutating pieces of the “data” section of the PDU is done through *DtPduWithData* member functions.

Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-17: DtPduWithData member functions

Function	Description
numFixedFields()	returns the number of fixed length datum fields in the PDU.
numVarFields()	returns the number of variable length datum fields in the PDU.
setNumVarFields() setnumFixed Fields()	resize the PDU to accommodate the new number of datum records. If newNum is less than the previous number of records, trailing records are deleted. If newNum is greater than the previous number of records, additional records are created after the last existing record. Newly created variable length records will have a data size of zero. Both fixed and variable data records will have a datum parameter ID of 0.
setDatumParam()	sets the nth fixed or variable datum ID in the PDU to param, where n = Index, depending on whether fixedOrVar is DtFIXED or DtVAR. Index should be between 1 and the number of fields in the PDU.
setVarDataBytes()	is used to change the data size in a particular variable length datum record. The PDU is resized to reflect the change. If the new value is less than the old, trailing bytes of the data are deleted. If the new value is greater than the old, bytes containing zeros are added after the last existing byte in the record.
initDataSection()	Initializes all of the datum records in the PDU at once, so that calls to setnumFixedFields() and setNumVarFields(), and repeated calls to setDatumParam() and setNumVarDatumBytes() do not need to be made. Changes the size of the PDU to reflect the total size of the fixed and variable data records described by params and specList. Sets the number of fixed records, and number of variable records appropriately. fixedParams is an array of DtDatumParams to be used as fixed datum parameter IDs. The last element of the array must be <i>DtNullDatumParam</i>. specList is an array of <i>DtVarDatumSpecs</i>, each of which describes the datum parameter ID, and data size (in bits) of one variable datum record. The last element of the array must be <i>DtNullVarDatumSpec</i>. See additional discussion following this table.
datumSizeBits()	returns the size of the datum field specified by fixedOrVar and Index . That is, if fixedOrVar is DtFIXED, 32 is returned, and if fixedOrVar is DtVAR, the size of the nth variable datum field is returned, where n = Index . In either case, if Index is out of bounds, 0 is returned. Indices range from 1 to numFixedFields() or numVarFields().
datumSizeBytes()	returns the value of datumSizeBits(), after converting to bytes.

A *DtVarDatumSpec* looks like this:

```
typedef struct
{
    DtDatumParam param;
    int length;
} DtVarDatumSpec;
```

For example:

```
static DtDatumParam fixed[] = {DtDatumPsi, DtDatumTheta,
                               DtDatumPhi, DtNullDatumParam};
static DtVarDatumSpec var[] =
{
    {DtDatumXpos, 64},
    {DtDatumYpos, 64},
    {DtDatumZpos, 64},
    {DtNullVarDatumSpec}
};
pdu.initDataSection(fixed, var);
```

Note: The *DtNullDatumParam* and *DtNullVarDatumSpec* serve only to mark the end of the list of records, and are not meant to indicate extra records. In the example above, the number of fixed records in the PDU is 3, and the number of variable length records in the PDU is 3.

Inspecting and Setting Data Values

DtPduWithData has many inspector and mutator functions that return and set the data value of a particular datum parameter respectively. The names of inspector functions start with “datumVal”, and the names of mutator functions start with “setDatumVal()”. The values of different datum parameters in a PDU can be of different data types. For each of the possible data types, we provide an inspector function that returns the value of a particular datum parameter, when that value is interpreted as belonging to that type, and a mutator function that sets the value of a particular datum parameter to a value of that type. The correct data type to be used for a particular datum parameter can be obtained using *DtDatumId2Type* (see “Data Types” on page A-48).

Particular datum records can be identified either by datum parameter ID, or by Index in the PDU. We use Index in a general sense here to mean a pair of indices which specify first whether you are talking about a fixed or variable datum field, and secondly the value of *n* in the expression “the *n*th fixed datum field”. Indices range from 1 to `numFixedFields()` or `numVarFields()`.

You can go back and forth between datum parameter IDs and indices using the functions `datumId()` and `getIndex()`. `getIndex()` fills in the arguments **fixedOrVar** and **Index** with `DtFIXED` or `DtVAR`, and an index, respectively. Taken together, these two values determine the slot in the PDU where the datum record associated with this datum parameter ID can be found.

`fixedDatumId()` returns the `n`th fixed datum ID in the PDU, where `n = Index`.
`DtNullDatumParam` is returned when `Index` is less than 1, or greater than the number of fixed datum fields in the PDU.

`varDatumId()` returns the `n`th variable datum ID in the PDU, where `n = Index`.
`DtNullDatumParam` is returned when `Index` is less than 1, or greater than the number of variable datum fields in the PDU.

`datumId()` returns `fixedDatumId(Index)` or `varDatumId(Index)`, depending on whether `fixedOrVar` is `DtFIXED` or `DtVAR`.

If no datum record with datum parameter ID `param` is found in the PDU, the values in `Index` and `fixedOrVar` are invalid, and the value of the pointer passed as the `retcode` argument is set to the value `DtDATUM_BAD_ID` (provided `retcode` is not NULL.) If `getIndex()` is successful, `retcode` the value pointed to by `retcode` is set to `DtDATUM_VALID`.

Several examples of usage of the `datumVal()` inspectors, the `setDatumVal()` mutators, and the `getIndex()` function follow:

```
double xval = dataPdu.datumValFloat64(DtDatumXpos);
```

assigns to `xval` the value contained in the datum record of the PDU whose datum parameter ID is `DtDatumXpos`, interpreting that value as a 64 bit floating point number.

An alternative way of doing the same thing is:

```
int fixedOrVar, Index, retcode;
/* first get the Index of the desired datum record */
pdu.getIndex(DtDatumXpos, &fixedOrVar, &Index, &retcode);
/* now get the value contained in that datum record */
if (retcode == DtDATUM_VALID)
    double xval = dataPdu.datumValFloat64(fixedOrVar, Index);

dataPdu.setDatumValInt32(DtDatumKind, 12);
```

will set the value of the `kind` parameter to 12, using a 32 bit integer representation.

```
int fixedOrVar, Index, retcode;
/* first get the Index of the desired datum record */
pdu.getIndex(DtDatumMass, &fixedOrVar, &Index, &retcode);
/* now set that datum record's value to 2000 */
if (retcode == DtDATUM_VALID)
{
    dataPdu.setDatumValFloat32(fixedOrVar, Index, 2000.0);
}
```

Note: Setting or inspecting datum values by `Index` is more efficient than by parameter ID (the latter type method first looks up the `Index` of the ID, then calls the former type of inspector (or mutator) with that `Index`). So if you plan to look up the value of the same ID more than once, it is more efficient to look up the `Index` once, then call the inspector with that `Index`.

Choosing among the different inspector and mutator functions for the data types is fairly straightforward. However, the `datumValByteArray()` and `setDatumValByteArray()` functions may always be used to get a bitwise copy of a particular data value. The value returned by `datumValByteArray()` is a pointer to internal memory. If the value is not to be used immediately, copy the data into local memory:

```
{
    char marking[12];
    memcpy(marking,
           dataPdu.datumValByteArray(fixedOrVar, Index, 12),
           dataPdu.datumSizeBytes(fixedOrVar, Index));
}
```

Return Codes

All of the datum value inspectors and mutators have an optional return code pointer argument, **retcode**. If this pointer is non-NULL, a return code for the function is stored in **retcode**. In all cases, the return code is `DtDATUM_VALID` when the function call was successful. This means that a specified datum parameter record exists in the PDU, and its value was able to be interpreted as the data type whose inspector function was used, or set to a value of the data type whose mutator function was used.

The return code is set to `DtDATUM_BAD_LENGTH` when the length of the datum value field (32 bits for fixed data, varies for variable data) does not match the size of the data type whose function you are using. For example:

```
int retcode;
double val = pdu.datumValFloat64(DtFIXED, 1, &retcode);
```

Here, **retcode** is set to `DtDATUM_BAD_LENGTH` since the 32 bit fixed data field cannot be interpreted as a 64 bit floating point number.

The return code is set to `DtDATUM_BAD_INDEX` when **Index** is out of range. For instance, if there are only 3 fixed datum records,

```
int retcode;
pdu.setDatumValFloat64(DtFIXED, 4, 725.0, &retcode);
```

causes **retcode** to contain `DtDATUM_BAD_INDEX`, since we have asked for the value of the 4th fixed record.

The return code is set to `DtDATUM_BAD_ID` when no datum record exists in the PDU whose parameter ID is **param**.

Data Types

VR-Link maintains a table of mappings between *DtDatumIds* and data types. The function *DtDatumId2Type* is used to look up the data type associated with a particular datum ID.

The different data type values that can be returned by *DtDatumId2Type* are:

- ♦ DtTYPE_BYTE_ARRAY
- ♦ DtTYPE_FLOAT32
- ♦ DtTYPE_FLOAT64
- ♦ DtTYPE_INT32
- ♦ DtTYPE_UNSIGNED32
- ♦ DtTYPE_STRING

If the table does not contain a mapping for *id*, then the value DtTYPE_NOT_DEFINED is returned.

All DIS 2.0.3 datum IDs are entered in this table by default along with reasonable data types. For instance, *DtDatumId2Type(DtDatumXpos)* returns DtTYPE_FLOAT64 while *DtDatumId2Type(DtDatumForceId)* returns DtTYPE_FLOAT32. This function is used by, among other things, *DtPduWithData*'s implementation of the *DtPdu* virtual function *printData()*, which needs to know the type of each datum value, so it can properly format them for printing:

```
for (int i = 1; i <= numVarFields(); i++)
{
    DtDatumParam param = datumId(i);
    switch (DtDatumId2Type(param))
    {
        case DtTYPE_FLOAT32:
            DtDumpFloat(DtEnumString(param), datumValFloat32(i));
            break;
        case DtTYPE_FLOAT64:
            DtDumpFloat(DtEnumString(param), datumValFloat32(i));
            break;
        case DtTYPE_INT32:
            DtDumpInteger(DtEnumString(param),
                          datumValInt32(i));
            break;
        .
        .
        .
    }
}
```

The data type of a particular datum ID can be changed, or a mapping for a new datum ID can be added to the table with the function *DtAddDatumId2TypeMapping*. This function returns DtTYPE_NOT_DEFINED if there was no previous data type mapping for *id* in the table. Otherwise, it returns the old value. If the optional **warn** argument is non-zero, then a warning message is printed when an existing mapping is being overridden.

The following classes in VR-Link are derived from *DtPduWithData*:

- ♦ *DtActionRequestPdu*
- ♦ *DtActionResponsePdu*
- ♦ *DtDataPdu*
- ♦ *DtEventReportPdu*
- ♦ *DtMessagePdu*
- ♦ *DtSetDataPdu*.

See Also

[*DtPdu*](#), [*DtSimanPdu*](#), and this PDU's header file ([*pduWithData.h*](#)).

DtReceiverPdu

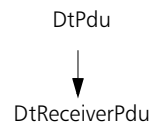
Name

DtReceiverPdu (DIS Receiver PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-18: DtReceiverPdu member functions

Function	Description
entityId() radioid() receiverState() receivedPower() transmitEntityId() transmitRadioid()	return the values of the appropriate data.
setEntityId() setRadioid() setReceiverState() setReceivedPower() setTransmitEntityId() setTransmitRadioid()	set the appropriate PDU data to the values passed as arguments.

See Also

[*DtPdu*](#), and this PDU's header file ([*receivePdu.h*](#)).

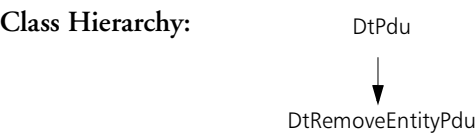
DtRemoveEntityPdu

Name

DtRemoveEntityPdu (DIS Remove Entity PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Constructors

Standard constructors derived from *DtPdu*:

- Blank PDU constructor
- From-network-representation constructor

Class-specific constructors:

- The constructor creates a blank *DtRemoveEntityPdu* using **buffer**, then sets the originating entity ID to **from**, and the receiving entity ID to **to**.

Functions

Table 1-19: DtRemoveEntityPdu member functions

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from DtSimanPdu. See “DtSimanPdu” on page A-64 for details.
requestId() setRequestId()	inspect and set this field.

Note: requestId() and setRequestId() are unavailable when linking with *libv13*.

See Also

DtPdu, *DtSimanPdu*, and this PDU’s header file (*removePdu.h*).

DtRepairCompletePdu

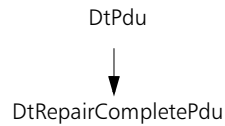
Name

DtRepairCompletePdu (DIS RepairComplete PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-20: DtRepairCompletePdu member functions

Function	Description
receivingId() repairingId() repairType()	return the values of the appropriate data.
setReceivingId() setRepairingId() setRepairType()	set the values of the appropriate data.

See Also

[*DtPdu*](#), and this PDU's header file ([*repairPdu.h*](#)).

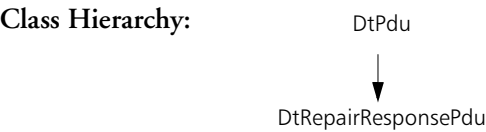
DtRepairResponsePdu

Name

DtRepairResponsePdu (DIS RepairResponse PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-21: DtRepairResponsePdu member functions

Function	Description
receivingId() repairingId() repairResult()	return the values of the appropriate data.
setReceivingId() setRepairingId() setRepairResult()	set the values of the appropriate data.

See Also

[DtPdu](#), and this PDU’s header file ([repairPdu.h](#)).

DtResupplyCancelPdu

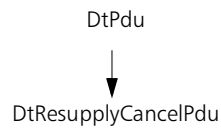
Name

DtResupplyCancelPdu (DIS ResupplyCancel PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Class-specific constructors:

- ♦ The constructor constructs a blank Resupply Cancel PDU, then sets the receiving and supplying entity IDs to the IDs passed as arguments.

Functions

Table 1-22: DtResupplyCancelPdu member functions

Function	Description
receivingId() supplyingId()	return the values of the appropriate data.
setReceivingId() setSupplyingId()	set the values of the appropriate data.

See Also

DtPdu, and this PDU's header file ([*resupplyPdu.h*](#)).

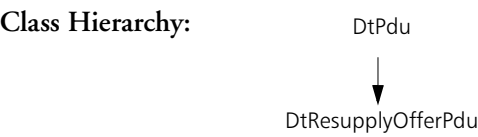
DtResupplyOfferPdu

Name

DtResupplyOfferPdu (DIS ResupplyOffer PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Class-specific constructors:

- ♦ The constructor may be used when the PDU needs to convey information about just one supply type. It constructs a Resupply Offer PDU with receiving and supplying entity IDs set to **receiver** and **supplier** respectively. The PDU will have one supply type record, with type **supply** and quantity **quantity**.

Functions

Table 1-23: DtResupplyOfferPdu member functions (Sheet 1 of 2)

Function	Description
receivingId() supplyingId() numSupplyTypes()	return the values of the appropriate data.
setReceivingId() setSupplyingId()	set the values of the appropriate data.

Table 1-23: DtResupplyOfferPdu member functions (Sheet 2 of 2)

Function	Description
nthSupply()	returns the entity type in the <i>n</i> th supply type record, where <i>n</i> is supplyIndex . Records are indexed from 0 to numSupplyTypes() - 1. If quantity is a non-NULL pointer, the integer to which it points is set to the quantity in which the <i>n</i> th supply type is being offered. If supplyIndex is not a valid index for the PDU, a <i>DtEntityType</i> with all fields set to zero is returned, and the integer pointed to by retCode (if retCode is non-NULL) is set to DtSUPPLY_BAD_INDEX. If the index is valid, the integer pointed to by retCode is set to DtSUPPLY_VALID.
supplyQuantity()	returns the quantity in which the supply type supply is being offered. If there is no supply type record whose supply type is supply , zero is returned.
setNumSupply-Types()	resizes the PDU to accommodate numTypes supply type records. If numTypes is greater than the previous number of supply type records, trailing records are deleted. Otherwise, supply type records with types and quantities consisting of all zeros are added to the end of the PDU.
setNthSupply()	is used to set the supply type and quantity that appears in the <i>n</i> th supply type record, where <i>n</i> is supplyIndex . Records are indexed from 0 to numSupplyTypes() - 1. The integer pointed to by retCode (if retCode is non-NULL) is set to DtSUPPLY_BAD_INDEX if supplyIndex is not in this range, and DtSUPPLY_VALID if it is. In the former case, the PDU remains unchanged.

See Also

[DtPdu](#), and this PDU's header file ([resupplyPdu.h](#)).

DtResupplyReceivedPdu

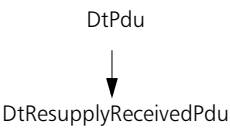
Name

DtResupplyReceivedPdu (DIS ResupplyReceived PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Class-specific constructors:

- ♦ The constructor may be used when the PDU needs to convey information about just one supply type. It constructs a Resupply Received PDU with receiving and supplying entity IDs set to **receiver** and **supplier** respectively. The PDU has one supply type record, with type **supply** and quantity **quantity**.

Functions

Table 1-24: DtResupplyReceivedPdu member functions (Sheet 1 of 2)

Function	Description
receivingId() supplyingId() numSupplyTypes()	return the values of the appropriate data.
setReceivingId() setSupplyingId()	set the values of the appropriate data.

Table 1-24: DtResupplyReceivedPdu member functions (Sheet 2 of 2)

Function	Description
nthSupply()	returns the entity type in the <i>n</i> th supply type record, where <i>n</i> is supplyIndex . Records are indexed from 0 to numSupplyTypes() - 1. If quantity is a non-NULL pointer, the integer to which it points is set to the quantity in which the <i>n</i> th supply type is being offered. If supplyIndex is not a valid index for the PDU, a <i>DtEntityType</i> with all fields set to zero is returned, and the integer pointed to by retCode (if retCode is non-NULL) is set to DtSUPPLY_BAD_INDEX. If the index is valid, the integer pointed to by retCode is set to DtSUPPLY_VALID.
supplyQuantity()	returns the quantity in which the supply type supply is being offered. If there is no supply type record whose supply type is supply , zero is returned.
setNumSupplyTypes()	resizes the PDU to accommodate numTypes supply type records. If numTypes is greater than the previous number of supply type records, trailing records are deleted. Otherwise, supply type records with types and quantities consisting of all zeros are added to the end of the PDU.
setNthSupply()	sets the supply type and quantity that appears in the <i>n</i> th supply type record, where <i>n</i> is supplyIndex . Records are indexed from 0 to numSupplyTypes() - 1. The integer pointed to by retCode (if retCode is non-NULL) is set to DtSUPPLY_BAD_INDEX if supplyIndex is not in this range, and DtSUPPLY_VALID if it is. In the former case, the PDU remains unchanged.

See Also

[DtPdu](#), and this PDU's header file ([resupplyPdu.h](#)).

DtSetDataPdu

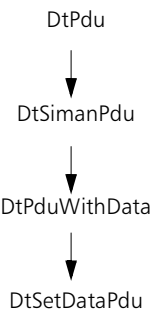
Name

DtSetDataPdu (DIS SetData PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-25: DtSetDataPdu member functions (Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> . See “DtSimanPdu” on page A-64

Table 1-25: DtSetDataPdu member functions (Sheet 2 of 2)

Function	Description
numFixedFields() numVarFields() setNumFixedFields() setNumVarFields() setDatumParam() setVarDataBytes() initDataSection() datumSizeBits() datumSizeBytes() datumId() getIndex() fixedDatumId() varDatumId() datumVal() functions setDatumVal() functions	are inherited from <i>DtPduWithData</i> (see “DtPduWithData” on page A-44)
requestId() setRequestId()	inspect and set this field.

See Also

DtPdu, *DtPduWithData*, *DtSimanPdu*, and this PDU’s header file ([setDataPdu.h](#)).

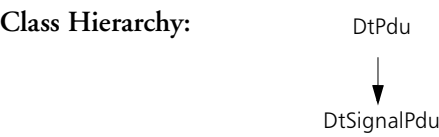
DtSignalPdu

Name

DtSignalPdu (DIS Signal PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-26: DtSignalPdu member functions (Sheet 1 of 2)

Function	Description
entityId() radioId() sampleRate() numSamples()	return the values of the appropriate data.
encodingClass()	returns the value of the upper 2 bits of the encoding scheme, one of the following: ♦ <i>DtSignalClassEncodedVoice</i> ♦ <i>DtSignalClassRawBinary</i> ♦ <i>DtSignalClassApplicationSpecific</i> ♦ <i>DtSignalClassPreRecorded</i>
encodingType()	returns the value in the lower 14 bits of the encoding scheme.
tdlType() setTdlType()	inspect and set the TDL type respectively.

Table 1-26: DtSignalPdu member functions (Sheet 2 of 2)

Function	Description
numDataBytes()	returns the size of the data section in bytes, while numDataBits() returns the size in bits.
dataByte()	returns the <i>n</i> th byte of the data section of the PDU, where <i>n</i> is byteNum . Data byte indices range from 0 to numDataBytes().
getData()	is used to copy the entire content of the data section of the PDU into the buffer buff . buff must be large enough to accommodate all of the data, that is, it should be at least numDataBytes() long.
setEntityId() setRadioId() setSampleRate() setNumSamples()	set the appropriate PDU data to the values passed as arguments.
setEncodingClass()	sets the value of the top two bits of the encoding scheme to one of the following: ♦ <i>DtSignalClassEncodedVoice</i> ♦ <i>DtSignalClassRawBinary</i> ♦ <i>DtSignalClassApplicationSpecific</i> ♦ <i>DtSignalClassPreRecorded</i>.
setEncodingType()	sets the value in the lower 14 bits of the encoding scheme to codeType .
setNumDataBits() setNumDataBytes()	s used to resize the PDU to accommodate the new size of the data section of the PDU. The PDU length and data length fields of the PDU are updated accordingly. If the new length is larger than the current length of the PDU, all new data bytes created are set to 0. If the new length is less than the old length, trailing data bytes are deleted.
setData()	will copy numBytes bytes starting from data into the data section of the PDU, effectively enabling setting the whole data section at one.
setDataByte()	may be used to set bytes in the data section one at a time. Bytes are indexed from 0 to numDataBytes() - 1.

Note: `tdlType()` and `setTdlType()` are not available when linking with *libvl3*.

See Also

DtPdu, and this PDU's header file (*signalPdu.h*).

DtSimanPdu

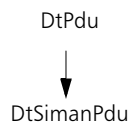
Name

DtSimanPdu (DIS Simulation Management PDU)

Description

DtSimanPdu is an abstract class that encapsulates the functionality common to all simulation management PDUs. Specifically, *DtSimanPdu* provides access to the sender and receiver IDs in instances of PDU classes that are derived from *DtSimanPdu*.

Class Hierarchy:



DtSimanPdu is derived from *DtPdu*, so that when an instance of a class derived from *DtSimanPdu* is constructed, inspecting and mutating PDU header data is done through *DtPdu* member functions. Since *DtSimanPdu* is an abstract class, an instance of *DtSimanPdu* cannot be directly constructed.

Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-27: DtSimanPdu member functions

Function	Description
senderId() and receiverId()	return the values of the appropriate data.
setSenderId() and setReceiverId()	set the appropriate PDU data to the values passed as arguments.

Notes

The following classes in VR-Link are derived directly from *DtSimanPdu*:

- ♦ *DtAcknowledgePdu*
- ♦ *DtCreateEntityPdu*
- ♦ *DtDataQueryPdu*
- ♦ *DtRemoveEntityPdu*
- ♦ *DtStartResumePdu*
- ♦ *DtStopFreezePdu*.

In addition, the following classes are derived from *DtPduWithData*, which is in turn derived from *DtSimanPdu*:

- ♦ *DtActionRequestPdu*
- ♦ *DtActionResponsePdu*
- ♦ *DtDataPdu*
- ♦ *DtEventReportPdu*
- ♦ *DtMessagePdu*
- ♦ *DtSetDataPdu*

See Also

[DtPdu](#), and this PDU's header file ([simanPdu.h](#)).

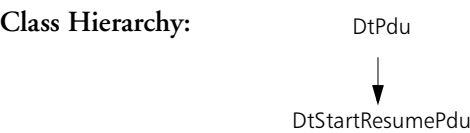
DtStartResumePdu

Name

DtStartResumePdu (DIS Start/Resume PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Class-specific constructors:

The constructor creates a blank *DtStartResumePdu* using **buffer**, then sets the originating entity ID to **from**, the receiving entity ID to **to**, the value of the real world time to **realTime**, and the value of the simulation time to **simTime**.

Functions

Table 1-28: DtStartResumePdu member functions (Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> (see “ DtSimanPdu ” on page A-64 for details)
realWorldTime()	returns the value of the real world time encoded in the PDU. The return value is a floating point number representing seconds since January 1, 1970.
simulationTime()	returns the value of the simulation time encoded in the PDU. The return value is a floating point number representing seconds since January 1, 1970.

Table 1-28: DtStartResumePdu member functions (Sheet 2 of 2)

Function	Description
setRealWorldTime() setSimulation- Time()	sets the values of those data. Time is a floating point number representing seconds since January 1, 1970.
requestId() setRequestId()	inspect and set this field.

Note: requestId() and setRequestId() are unavailable when linking with *libv3*.

See Also

[DtPdu](#), [DtSimanPdu](#), and this PDU's header file ([startPdu.h](#)).

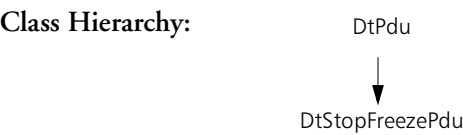
DtStopFreezePdu

Name

DtStopFreezePdu (DIS Stop/Freeze PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.



Constructors

Standard constructors derived from *DtPdu*:

- Blank PDU constructor
- From-network-representation constructor

Class-specific constructors:

- The constructor creates a blank *DtStopFreezePdu* using **buffer**, then sets the originating entity ID to **from**, the receiving entity ID to **to**, the value of the real world time to **realTime**, and the stop reason to **reason**.

Functions

Table 1-29: DtStopFreezePdu member functions (Sheet 1 of 2)

Function	Description
senderId() receiverId() setSenderId() setReceiverId()	are inherited from <i>DtSimanPdu</i> (see “ DtSimanPdu ” on page A-64 for details)
realWorldTime()	returns the value of the real world time encoded in the PDU. The return value is a floating point number representing seconds since January 1, 1970.
setRealWorldTime()	sets the real world time to Time , which is a floating point number representing seconds since January 1, 1970.

Table 1-29: DtStopFreezePdu member functions (Sheet 2 of 2)

Function	Description
frozenBehavior() setFrozenBehavior() requestId() setRequestId()	inspect and set the appropriate fields.
reason()	returns the reason for stopping
setReason()	sets the reason for stopping.

Note: These functions are unavailable when linking with *libv3*.

See Also

[DtPdu](#), [DtSimanPdu](#), and this PDU's header file ([stopPdu.h](#)).

DtTransmitterPdu

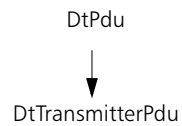
Name

DtTransmitterPdu (DIS Transmitter PDU)

Description

Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-30: DtTransmitterPdu member functions(Sheet 1 of 2)

Function	Description
entityId() radioId() radioEntityType() transmitState() inputSource() worldAntennaLoc() relativeAntenna Loc() antennaPattern Type() antennaParams Length() frequency() bandwidth() power() crypto- System() cryptoKeyId() modulationParams Length()	return the values of the appropriate data.
radioEntityType()	returns a <i>DtEntityType</i> when using DIS version 2.0.3, (linked with <i>libvl3.a</i>), and a <i>DtRadioEntityType</i> when using DIS version 4 or 5 (linked with <i>libvl.a</i>).
antenna- ParamsLength() and modulation- ParamsLength()	return values in bytes. As 64 bit integers are not supported on most platforms frequency has an optional argument to indicate whether one wishes to inspect the upper or lower 32 bit word of the 64 bit frequency field. The upperLower argument defaults to DtFREQ_LOWER, but may also be set to DtFREQ_UPPER.
modulationType()	returns a reference to a <i>DtModulationType</i> object. The <i>DtModulationType</i> member functions <code>spectrum()</code> , <code>majorModType()</code> , <code>detail()</code> , and <code>systemType()</code> may be used to obtain individual pieces of data from this object. For more information see the text following this table.

Table 1-30: DtTransmitterPdu member functions(Sheet 2 of 2)

Function	Description
setEntityId()	set the appropriate PDU data to the values passed as arguments. See text for further discussion
setRadioID()	
setRadioEntity- Type()	
setTransmitState()	
setInputSource()	
setWorldAntenna Loc()	
setRelativeAntenna Loc()	
setAntennaPattern- Type()	
setFrequency()	
setBandwidth()	
setPower()	
setModulation- Type()	
setCryptoSystem()	
setCryptoKey()	

modulationType() Example

```
int detail = transPdu.modulationType().detail();
```

The layout for the modulation parameters portion of the Transmitter PDU depends on the modulation type and is not described in the DIS standard. Therefore, VR-Link gives the user access to the bitwise network representation of this portion of the PDU. The function `getNetModulationParams()` copies this data into the buffer provided as the **destination** argument to the function. This buffer must be at least `modulationParamsLength()` long.

Similarly, one can obtain the bitwise network representation of the antenna pattern parameters, using `getNetAntennaParams()`. The size of the buffer passed as **destination** must be at least `antennaParamsLength()` bytes long.

If the antenna pattern type for a particular transmitter PDU is a beam antenna pattern (that is, `antennaPatternType()` returns *DtBeam*), the function `antennaParams()` can be used to interpret the antenna pattern parameters as a beam antenna pattern record. A reference to a *DtBeamAntennaPatternRecord* is returned. The *DtBeamAntennaPatternRecord* member functions can be used to inspect the values of its individual fields. They are:

- ♦ `angles()`
- ♦ `azimuthBeamwidth()`
- ♦ `elevationBeamwidth()`
- ♦ `referenceSystem()`
- ♦ `electricFieldZ()`
- ♦ `electricFieldX()`
- ♦ `phase()`

For example:

```
float phase = pdu.antennaParams().phase();
```

setRadioEntityType()

When linking with *libvl.a* (using DIS 2.0.4 or 1278.1-1995), `setRadioEntityType()` is overloaded to take either a *DtRadioEntityType* or a *DtEntityType*. However, use *DtRadioEntityType* in version 1278.1-1995 PDUs; use *DtEntityType* in DIS 2.0.4 PDUs.

Note: When linking with *libvl3.a*, (using DIS 2.0.3), only the version of `setRadioEntityType()` that takes a *DtEntityType* is available.

setFrequency()

There are two versions of `setFrequency()`. One allows you to set both the upper and lower words of the 64-bit frequency field. The other may be used when the frequency will fit completely in the lower word.

setModulationType()

The **type** argument to `setModulationType()` is reference to a *DtModulationType* object. The *DtModulationType* member functions `setSpectrum()`, `setMajor()`, `setDetail()`, and `setSystemType()` may be used to set the individual components of the object. Alternately, all of these data may be passed to the *DtModulationType* constructor. For example:

```
pdu.setModulationType(DtModulationType(spec, maj, det, sys));
```

DIS does not specify the layout of the modulation parameters record. The layout, in fact, depends on the modulation type. VR-Link allows you to set the exact network representation for the PDU's modulation parameter information using `setModulationParams()`. Typically, this would be used by creating an appropriate struct, filling out all of the fields, and passing it to this function. The **size** argument indicates the length, in bytes, of the modulation parameters.

setAntennaPatternParams()

Similarly, the function `setAntennaPatternParams()` may be used to set the network representation of the antenna pattern parameters. **size** indicates the length, in bytes, of the antenna pattern parameters.

If the antenna pattern type is a beam antenna pattern, the antenna pattern parameters may be represented as a *DtBeamAntennaPatternRecord*. An overloaded version of `setAntennaPatternParams()` takes a reference to an object of this type, and sets the PDU's antenna pattern parameter record appropriately. The *DtBeamAntennaPatternRecord* member functions can be used to set the individual fields of this object. They are:

- ♦ `setAngles()`
- ♦ `setBeamwidth()`
- ♦ `setReferenceSystem()`
- ♦ `setElectricField()`
- ♦ `setPhase()`

See Also

DtPdu, and this PDU's header file (*transmitPdu.h*).

DtViewControlPdu

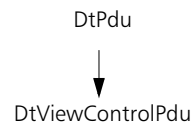
Name

DtViewControlPdu

Description

A *DtViewControlPdu* represents an instance of MÄK's ViewControl PDU. Inspecting and mutating PDU header data is done through *DtPdu* member functions.

Class Hierarchy:



When sent to the MÄK Stealth, View Control PDUs cause changes in the Stealth's View State.

There are currently 10 different types of View Control PDU that the MÄK Stealth accepts - one each for nine of its View Modes and one to indicate a change of attached entity or view group without setting the current mode. Each View Mode requires different parameters to describe the View State. A subclass of *DtViewControlPdu* represents each type of View Control PDU:

- ♦ *DtAbsoluteVCPdu*
- ♦ *DtTetherVCPdu*
- ♦ *DtCompassVCPdu*
- ♦ *DtMimicVCPdu*
- ♦ *DtOrbitVCPdu*
- ♦ *DtTrackVCPdu*
- ♦ *DtTetherTrackVCPdu*
- ♦ *DtMimicTrackVCPdu*
- ♦ *DtGroupVCPdu*
- ♦ *DtModelessVCPdu*.

In addition, the derived class *DtUnknownVCPdu* may be used to represent new experimental or unknown kinds of View Control PDU.

Data that is common to all types of View Control PDUs may be accessed through the base *DtViewControlPdu* mutators and inspectors.

All View Control PDUs have the same DIS PDU kind (*DtViewControlPduKind* = 202), but each type has its own Viewer Command. *viewerCommand()* returns the PDU's Viewer Command. (See [disEnums.h](#) for the *DtViewerCommand* enumeration.)

The Viewer ID in a View Control PDU is the entity ID of the Stealth to which the PDU is directed. A value of 0xFFFF may be used as a wildcard in any field of the entity ID. `viewerId()` and `setViewerId()` access the PDU's Viewer Id.

Constructors

Standard constructors derived from *DtPdu*:

- ♦ Blank PDU constructor
- ♦ From-network-representation constructor

Functions

Table 1-31: DtViewControlPdu member functions

Function	Description
<code>senderId()</code> and <code>setSenderId()</code>	inspect and set the entity ID of the sender of PDU.
<code>ignoreFlags()</code> and <code>setIgnoreFlags()</code>	inspect and set ignore flags.

All View Control PDUs have an Ignore Flags field that is used to indicate to the Stealth that certain viewing parameters should remain as they were, and should not be set from this PDU. The different Ignore commands that may be OR'ed into the Ignore bitfield are:

- ♦ `DtIGNORE_EAST`
- ♦ `DtIGNORE_PITCH_OFFSET`
- ♦ `DtIGNORE_NORTH`
- ♦ `DtIGNORE_FORWARD`
- ♦ `DtIGNORE_HEIGHT`
- ♦ `DtIGNORE_RIGHT`
- ♦ `DtIGNORE_HEADING`
- ♦ `DtIGNORE_UP`
- ♦ `DtIGNORE_PITCH`
- ♦ `DtIGNORE_BODY_PITCH`
- ♦ `DtIGNORE_ROLL`
- ♦ `DtIGNORE_BODY_ROLL`
- ♦ `DtIGNORE_EAST_OFFSET`
- ♦ `DtIGNORE_AZIMUTH_RATE`
- ♦ `DtIGNORE_NORTH_OFFSET`

- ♦ DtIGNORE_RADIAL_OFFSET
- ♦ DtIGNORE_HEIGHT_OFFSET
- ♦ DtIGNORE_PRIMARY_ID
- ♦ DtIGNORE_RADIUS
- ♦ DtIGNORE_SECONDARY_ID
- ♦ DtIGNORE_BODY_HEADING
- ♦ DtIGNORE_GROUP_IDS
- ♦ DtIGNORE_HEADING_OFFSET

The values of the parameters in a view Control PDU can be represented with an ASCII view command string. These strings are used in *viewsave.mtl* - the file written by the Stealth whenever the saved view states are to be written, and also in Stealth Scripting files, which record a sequence of View State changes. **string** returns a string representing the data in the PDU, and **setFromString()** sets the data in the PDU to the values in the string.

DtNewViewControlPdu creates an instance of the appropriate derived *DtViewControlPdu* based on the value of **command**.

Each derived *DtViewControlPdu* has individual accessor functions to inspect and mutate the parameters associated with its View Mode.

See Also

[DtPdu](#), and this PDU's header file ([vcPdu.h](#)).



Header File Reference

This chapter contains abbreviated versions of the VR-Link header files.

This appendix appears only in the on-line PDF version of the documentation. You can use Acrobat's *search* and *bookmark* features to find information quickly. In addition, the cross-reference tables on pages [9-4](#) through [9-28](#) contain hypertext links that allow easy navigation of all header files and classes in this appendix.

absNotifyObs.h

```
#include "MachineTypes.h"
#include "observee.h"

// class DtAbsNotifyObserver:
//
// DtAbsNotifyObserver is an abstract base class.
// Instances of DtAbsNotifyObserver are used by DtObservee to
// notify an observer object that something has happened to the
// observed object. Observers should subclass from DtAbsNotifyObserver
// to receive notification for their class.

class DtAbsNotifyObserver
{
public:

    // destructor
    virtual ~DtAbsNotifyObserver();

    // assignment operator
    DtAbsNotifyObserver& operator=(const DtAbsNotifyObserver& orig);

    // Called by DtObservee when something has happened to the
    // observed object.
    // Default does nothing. This should be overridden.
    virtual DtBoolean notifyFromObserved(DtObservee* observed,
        const DtObservee::DtWhyNotifyObserver& why);

protected:

    // Constructors are protected since this is an abstract base class.

    // default constructor
    DtAbsNotifyObserver();

    // copy constructor
    DtAbsNotifyObserver(const DtAbsNotifyObserver& orig);
};
```

ackPdu.h

```

#if !DtHLA

#include "hostStructs.h"
#include "simanPdu.h"

#define DtAcknowledgeStaticSize 32

class DtAcknowledgePdu;
typedef DtAcknowledgePdu DtAcknowledgeInteraction;
typedef void (*DtAcknowledgeInteractionCb)
    (DtAcknowledgeInteraction *pdu, void *usr);

class DtAcknowledgePdu : public DtSimanPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtAcknowledgePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtAcknowledgePdu(const DtNetAcknowledgePdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtAcknowledgeInteractionCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtAcknowledgeInteractionCb cb, void *usr);

/***** ADDITIONAL CONSTRUCTORS *****/

    DtAcknowledgePdu(const DtEntityIdentifier &to,
        const DtEntityIdentifier &from,
        DtAcknowledgeFlag aFlag,
        DtResponseFlag rFlag,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

    DtAcknowledgeFlag acknowledgeFlag() const;
    DtResponseFlag responseFlag() const;
#if DtDISVERS != 3
    int requestId() const;
#endif

/***** MUTATOR FUNCTIONS *****/

    void setAcknowledgeFlag(DtAcknowledgeFlag flag);
    void setResponseFlag(DtResponseFlag flag);
#if DtDISVERS != 3
    void setRequestId(int id);
#endif

```

```
#endif

    void printData() const;

protected:
    DtPduKind internalGetPduKind() const {return DtAcknowledgePduKind;}
};
#endif
```


ackRPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "simanPdu.h"
#include "packets/ackRP.h"
#include "exerciseConn.h"

// DtAcknowledgeRPdu callback function signature.
typedef void (*DtAcknowledgeRPduCb)(DtAcknowledgeRPdu* pdu, void* usr);

// class DtAcknowledgeRPdu:
//
// Instances of DtAcknowledgeRPdu are used to send reliable acknowledge
// PDUs.

class DtAcknowledgeRPdu : public DtSimanPdu
{
public:

    // Constructors
    DtAcknowledgeRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtAcknowledgeRPdu(const DtNetAcknowledgeRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtAcknowledgeRPdu(const DtAcknowledgeRPdu& orig);

    // destructor
    virtual ~DtAcknowledgeRPdu();

    // Assignment operator
    DtAcknowledgeRPdu& operator=(const DtAcknowledgeRPdu& orig);

    // Set/Get the acknowledge flag.
    virtual void setAcknowledgeFlag(DtAcknowledgeFlag flag);
    virtual DtAcknowledgeFlag acknowledgeFlag() const;

    // Set/Get the response flag.
    virtual void setResponseFlag(DtResponseFlag flag);
    virtual DtResponseFlag responseFlag() const;

    // Set/Get the request id.
    virtual void setRequestId(int id);
    virtual int requestId() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:

    // Create a DtAcknowledgeRPdu.
    // Caller is responsible for deletion.

```

```
static DtPdu* create(const DtNetPacket* initial,
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtAcknowledgeRPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtAcknowledgeRPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

protected:

    // Size of PDU with zero data bytes.
    static int theStaticSize;

};
#endif
#endif
```

actionPdu.h

```

#if !DtHLA

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"

/*****
***** ACTION REQUEST PDU *****/
*****/

#if DtDISVERS != 3
#define DtActionRequestStaticSize 40
#else
#define DtActionRequestStaticSize 48
#endif

typedef DtActionRequestPdu DtActionRequestInteraction;
typedef void (*DtActionRequestInteractionCb)
    (DtActionRequestInteraction *pdu, void *usr);

class DtActionRequestPdu : public DtPduWithData
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtActionRequestPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtActionRequestPdu(const DtNetActionRequestPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtActionRequestInteractionCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtActionRequestInteractionCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    unsigned long requestId() const;
    unsigned long actionId() const;

/***** MUTATOR FUNCTIONS *****/

    void setRequestId(unsigned long id);
    void setActionId(unsigned long id);

    void printData() const;

protected:

```

```
DtPduKind internalGetPduKind() const {return DtActionRequestPduKind;}
DtNetDataPduTail *dataSection() const;
};
/*****
/***** ACTION RESPONSE PDU *****/
/*****/

#if DtDISVERS != 3
#define DtActionResponseStaticSize 40
#else
#define DtActionResponseStaticSize 48
#endif

typedef DtActionResponsePdu DtActionResponseInteraction;
typedef void (*DtActionResponseInteractionCb)(DtActionResponsePdu *pdu,
void *usr);

class DtActionResponsePdu : public DtPduWithData
{
public:

/***** STANDARD CONSTRUCTORS *****/

DtActionResponsePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
DtActionResponsePdu(const DtNetActionResponsePdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
static void addCallback(DtExerciseConn *conn,
DtActionResponseInteractionCb cb, void *usr);
static void removeCallback(DtExerciseConn *conn,
DtActionResponseInteractionCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

unsigned long requestId() const;
DtRequestStatus requestStatus() const;

/***** MUTATOR FUNCTIONS *****/

void setRequestId(unsigned long id);
void setRequestStatus(DtRequestStatus rStatus);

void printData() const;

protected:

DtPduKind internalGetPduKind() const {return DtActionResponsePduKind;}
DtNetDataPduTail *dataSection() const;
};

#endif
```

actReqRPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"
#include "packets/actReqRP.h"
#include "exerciseConn.h"

// class DtActionRequestRPdu:
//
// Instances of DtActionRequestRPdu are used to send/receive
// reliable action request PDUs.

// DtActionRequestRPdu callback function signature.
typedef void (*DtActionRequestRPduCb)(DtActionRequestRPdu* pdu, void* usr);

class DtActionRequestRPdu : public DtPduWithData
{
public:

    // Constructors
    DtActionRequestRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtActionRequestRPdu(const DtNetActionRequestRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtActionRequestRPdu(const DtActionRequestRPdu& orig);

    // destructor
    virtual ~DtActionRequestRPdu();

    // Assignment operator
    DtActionRequestRPdu& operator=(const DtActionRequestRPdu& orig);

    // Set/Get the request id.
    virtual void setRequestId(unsigned long id);
    virtual unsigned long requestId() const;

    // Set/Get the action id.
    virtual void setActionId(unsigned long id);
    virtual unsigned long actionId() const;

    // Set/Get the required reliability service.
    virtual void setReqRelService(DtRequiredReliabilityService value);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:

    // Create a DtActionRequestRPdu.

```

```
// Caller is responsible for deletion.
static DtPdu* create(const DtNetPacket* initial,
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtActionRequestRPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtActionRequestRPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Returns the data portion of the PDU
    virtual DtNetDataPduTail* dataSection() const;

protected:

    static int theStaticSize;

};
#endif
#endif
```

actRespRPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"
#include "packets/actRespRP.h"
#include "exerciseConn.h"

// class DtActionResponseRPdu:
//
// Instances of DtActionResponseRPdu are used to send/receive
// reliable action response PDUs.

// DtActionResponseRPdu callback function signature.
typedef void (*DtActionResponseRPduCb)(DtActionResponseRPdu* pdu, void*
usr);

class DtActionResponseRPdu : public DtPduWithData
{
public:

    // Constructors
    DtActionResponseRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtActionResponseRPdu(const DtNetActionResponseRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtActionResponseRPdu(const DtActionResponseRPdu& orig);

    // destructor
    virtual ~DtActionResponseRPdu();

    // Assignment operator
    DtActionResponseRPdu& operator=(const DtActionResponseRPdu& orig);

    // Set/Get the request id.
    virtual void setRequestId(unsigned long id);
    virtual unsigned long requestId() const;

    // Set/Get the request status.
    virtual void setRequestStatus(DtRequestStatus rStatus);
    virtual DtRequestStatus requestStatus() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:

    // Create a DtActionResponseRPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

```

```
// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtActionResponseRPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtActionResponseRPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Returns the data portion of the net structure.
    virtual DtNetDataPduTail* dataSection() const;

protected:

    static int theStaticSize;

};
#endif
#endif
```


aggAggList.h

```

#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/aggStateP.h"

// class DtSilentAggregateList:
//
// DtSilentAggregateList is a class which contains the information needed
// to describe a list of silent aggregates for the aggregate state PDU.

class DtSilentAggregateList
{
public:

    // Constructors
    DtSilentAggregateList();
    DtSilentAggregateList(const DtSilentAggregateList& orig);

    // Destructor
    virtual ~DtSilentAggregateList();

    // Assignment operator
    DtSilentAggregateList& operator=(const DtSilentAggregateList& orig);
    DtSilentAggregateList& operator=(const DtNetSilentAggregateList& orig);

    // Comparison operators
    int operator==(const DtSilentAggregateList& otherguy) const;
    int operator!=(const DtSilentAggregateList& otherguy) const;

    // Set/Get the number of aggregates
    virtual void setNumberOfAggregates(const DtU16 value);
    virtual DtU16 numberOfAggregates() const;

    // Set/Get the aggregate type
    virtual void setAggregateType(const DtEntityType& type);
    virtual const DtEntityType& aggregateType() const;

    // Print the record
    virtual void print() const;

    // Load/unload data from a packet record
    virtual void unloadNetRecord(const DtNetSilentAggregateList& record);
    virtual void loadNetRecord(DtNetSilentAggregateList& record) const;

public:

    // Get the length of the network representation of a record

```

```
#ifndef DATA_LENGTH_OF_NETWORK_RECORD
#define DATA_LENGTH_OF_NETWORK_RECORD 16
#endif

#pragma pack(push, 1)
struct DtEntityRecord {
    DtU16 myNumberOfAggregates;
    DtEntityType myEntityType;
};
#pragma pack(pop)
```

aggCbInfo.h

```

#if DtDISVERS >= 5

#if WIN32 // DevStudio has a nested #if limit
#include "vlUtil.h"
#endif

#include "vlList.h"

#if DtHLA || DtHLA_GATEWAY
#include "hlaDefines.h"
#else
#include "disDefines.h"
#endif

// Aggregate state callback function signature.
typedef void (*DtAggregateCallbackFunction)(
    DtReflectedAggregate* agg, void* userData);

#include "chooseDefines.h"

// class DtAggregateCallbackInfo:
//
// Instances of DtAggregateCallbackInfo are used to contain the function
// and user data associated with a callback which delivers aggregate state
// information.

class DtAggregateCallbackInfo : public DtListedObj
{
public:
    // default constructor
    DtAggregateCallbackInfo();

    // constructor
    DtAggregateCallbackInfo(DtAggregateCallbackFunction cb, void* userData);

    // destructor
    virtual ~DtAggregateCallbackInfo();

    // copy constructor
    DtAggregateCallbackInfo(const DtAggregateCallbackInfo& orig);

    // assignment operator
    DtAggregateCallbackInfo& operator=(const DtAggregateCallbackInfo& orig);

    // equality operators
    virtual int operator==(const DtAggregateCallbackInfo& orig) const;
    virtual int operator!=(const DtAggregateCallbackInfo& orig) const;

    // set/get the callback function

```

```
virtual void setCallbackFunction(DtAggregateCallbackFunction funct);
virtual DtAggregateCallbackFunction callbackFunction() const;

// set/get the user data
virtual void setUserData(void* data);
virtual void* userData() const;

protected:

    DtAggregateCallbackFunction myCallbackFunction;
    void* myUserData;

};
#endif
```

aggEntDec.h

```

#if DtHLA

#include "baseEntDec.h"

// class DtAggregateEntityDecoder:
//
// Instances of DtAggregateEntityDecoder are used to decode an
// aggregate state message into a state repository in an HLA
// (as opposed to DIS) environment.

class DtAggregateEntityDecoder : public DtBaseEntityDecoder
{
public:

    // constructor
    DtAggregateEntityDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtAggregateEntityDecoder();

    // copy constructor
    DtAggregateEntityDecoder(const DtAggregateEntityDecoder& orig);

    // assignment operator
    DtAggregateEntityDecoder& operator=(const DtAggregateEntityDecoder&
orig);

public:

    // Create a DtAggregateEntityDecoder.
    // User is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

protected:

    // static decode<AttributeName> member functions.
    DtDECLARE_AGGREGATE_ATTR_DECODER(AggregateMarking);
    DtDECLARE_AGGREGATE_ATTR_DECODER(AggregateState);
    DtDECLARE_AGGREGATE_ATTR_DECODER(Dimensions);
    DtDECLARE_AGGREGATE_ATTR_DECODER(EntityIDs);

```

```
DtDECLARE_AGGREGATE_ATTR_DECODER(ForceID);
DtDECLARE_AGGREGATE_ATTR_DECODER(Formation);
DtDECLARE_AGGREGATE_ATTR_DECODER(NumberOfEntities);
DtDECLARE_AGGREGATE_ATTR_DECODER(SilentAggregates);
DtDECLARE_AGGREGATE_ATTR_DECODER(SilentEntities);
DtDECLARE_AGGREGATE_ATTR_DECODER(SubAggregateIDs);
DtDECLARE_AGGREGATE_ATTR_DECODER(VariableDatums);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_AGGREGATE_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtAggregateEntityDecoder, DtAggregateStateRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_AGGREGATE_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_AGGREGATE_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif
```

aggEntEnc.h

```

#if DtHLA

#include "baseEntEnc.h"
#include "aggregateSR.h"

// class DtAggregateEntityEncoder:
//
// Instances of DtAggregateEntityEncoder are used to encode an
// aggregate entity's state repository into a state message in an
// HLA environment.

class DtAggregateEntityEncoder : public DtBaseEntityEncoder
{
public:

    // constructor
    DtAggregateEntityEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtAggregateEntityEncoder();

    // copy constructor
    DtAggregateEntityEncoder(const DtAggregateEntityEncoder& orig);

    // assignment operator
    DtAggregateEntityEncoder& operator=(const DtAggregateEntityEncoder&
orig);

public:

    // Create a DtAggregateEntityEncoder.
    // Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // static need<Attribute> member functions.
    DtDECLARE_AGGREGATE_ATTR_CHECKER(AggregateMarking);
    DtDECLARE_AGGREGATE_ATTR_CHECKER(AggregateState);
    DtDECLARE_AGGREGATE_ATTR_CHECKER(Dimensions);
    DtDECLARE_AGGREGATE_ATTR_CHECKER(EntityIDs);
    DtDECLARE_AGGREGATE_ATTR_CHECKER(ForceID);
    DtDECLARE_AGGREGATE_ATTR_CHECKER(Formation);
    DtDECLARE_AGGREGATE_ATTR_CHECKER(NumberOfEntities);
    DtDECLARE_AGGREGATE_ATTR_CHECKER(SilentAggregates);
    DtDECLARE_AGGREGATE_ATTR_CHECKER(SilentEntities);
    DtDECLARE_AGGREGATE_ATTR_CHECKER(SubAggregateIDs);

```

```
DtDECLARE_AGGREGATE_ATTR_CHECKER(VariableDatums);

// static decode<Attribute> member functions.
DtDECLARE_AGGREGATE_ATTR_ENCODER(AggregateMarking);
DtDECLARE_AGGREGATE_ATTR_ENCODER(AggregateState);
DtDECLARE_AGGREGATE_ATTR_ENCODER(Dimensions);
DtDECLARE_AGGREGATE_ATTR_ENCODER(EntityIDs);
DtDECLARE_AGGREGATE_ATTR_ENCODER(ForceID);
DtDECLARE_AGGREGATE_ATTR_ENCODER(Formation);
DtDECLARE_AGGREGATE_ATTR_ENCODER(NumberOfEntities);
DtDECLARE_AGGREGATE_ATTR_ENCODER(SilentAggregates);
DtDECLARE_AGGREGATE_ATTR_ENCODER(SilentEntities);
DtDECLARE_AGGREGATE_ATTR_ENCODER(SubAggregateIDs);
DtDECLARE_AGGREGATE_ATTR_ENCODER(VariableDatums);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_AGG_ENTITY_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtAggregateEntityEncoder, \
        DtAggregateStateRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that checker whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_AGG_ENTITY_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtAggregateEntityEncoder, \
        DtAggregateStateRepository, \
        attrName, inspector)

#endif
```


aggEntList.h

```

#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/aggStateP.h"

// class DtSilentEntityList:
//
// DtSilentEntityList is a class which contains the information needed
// to describe a list of silent entities for the aggregate state PDU.

class DtSilentEntityList
{
public:

    // Constructors
    DtSilentEntityList();
    DtSilentEntityList(const DtSilentEntityList& orig);

    // Destructor
    virtual ~DtSilentEntityList();

    // Assignment operator
    DtSilentEntityList& operator=(const DtSilentEntityList& orig);
    DtSilentEntityList& operator=(const DtNetSilentEntityList& orig);

    // Comparison operators
    int operator==(const DtSilentEntityList& otherguy) const;
    int operator!=(const DtSilentEntityList& otherguy) const;

    // Set/Get the number of aggregates
    virtual void setNumberOfEntities(const DtU16 value);
    virtual DtU16 numberOfEntities() const;

    // Set/Get the entity type
    virtual void setEntityType(const DtEntityType& type);
    virtual const DtEntityType& entityType() const;

#if !DtHLA
    // Set/Get the number of appearance records. Set returns DtFalse if the
    // value is too large, old values are not retained when the number
    // changes.
    virtual DtBoolean setNumberOfAppearanceRecords(const DtU16 value);
    virtual DtU16 numberOfAppearanceRecords() const;

    // Set/Get the appearance of an entity (as a 32 bit integer). Returns
    // DtFalse if the index is out of range. Set will erase all previous
    // values.

```

```
virtual DtBoolean setAppearance(int index, DtU32 app);
virtual DtBoolean getAppearance(int index, DtU32& app) const;

// Set/Get appearance by attribute. Returns DtFalse if the index
// is out of range.
virtual DtBoolean setAppearanceAttribute(int index,
    const DtAppearAttribute attr, const DtAppearAttrVal val);
virtual DtBoolean getAppearanceAttribute(int index,
    const DtAppearAttribute attr, DtAppearAttrVal& val) const;
#endif

// Print the record
virtual void print() const;

// Load/unload data from a packet record
virtual void unloadNetRecord(const DtNetSilentEntityList& record);
virtual void loadNetRecord(DtNetSilentEntityList& record) const;

// Get the network byte length of this record
virtual int networkByteLength() const;

public:

    // Get the length of the network representation of a record
    static int byteLengthOfNetworkRecord(DtNetSilentEntityList* recPtr);

protected:

    // Data members:
    DtU16                myNumberOfEntities;
    DtEntityType          myEntityType;
#if !DtHLA
    DtU16                myNumberOfAppearanceRecords;
    DtU32*               myEntityAppearanceList;
#endif
};
#endif
```

aggPub.h

```
#if DtDISVERS >= 5

// This file provides protocol independence for local aggregates.

#include "hAggPub.h"
#include "dAggPub.h"

#endif
```

aggStatePdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"
#include "aggEntList.h"
#include "aggAggList.h"
#include "varDatumRec.h"

#include "mathTypes.h"

#include "packets/aggStateP.h"

// class DtAggregateStatePdu:
//
// Instances of DtAggregateStatePdu are used to communicate information
// about
// aggregating entities.

// DtAggregateStatePdu callback function signature.
typedef void (*DtAggregateStatePduCb)(DtAggregateStatePdu* pdu, void* usr);

class DtAggregateStatePdu : public DtPdu
{
public:

    // Constructors
    DtAggregateStatePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtAggregateStatePdu(const DtNetAggregateStatePdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtAggregateStatePdu(const DtAggregateStatePdu& orig);

    // destructor
    virtual ~DtAggregateStatePdu();

    // Assignment operator
    DtAggregateStatePdu& operator=(const DtAggregateStatePdu& orig);

    // Set/Get aggregate entity identifier
    virtual void setAggregateID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& aggregateID() const;

    // Set/Get the force ID
    virtual void setForceID(DtForceType force);
    virtual DtForceType forceID() const;

    // Set/Get the aggregate state
    virtual void setAggregateState(DtAggregateState value);
```

```
virtual DtAggregateState aggregateState() const;

// Set/Get the aggregate type
virtual void setAggregateType(const DtEntityType& type);
virtual const DtEntityType& aggregateType() const;

// Set/Get the formation
virtual void setFormation(DtAggregateFormation value);
virtual DtAggregateFormation formation() const;

// Set/Get marking character set
virtual void setMarkingCharacterSet(DtCharacterSet value);
virtual DtCharacterSet markingCharacterSet() const;

// Set/Get marking text (copies 31 bytes)
virtual void setMarkingText(DtU8* copyFrom);
virtual void getMarkingText(DtU8* copyTo) const;

// Set/Get the dimensions
virtual void setDimensions(DtConstVector dims);
virtual DtConstVector dimensions() const;

// Set/Get orientation
virtual void setOrientation(const DtTaitBryan& orientation);
virtual const DtTaitBryan& orientation() const;

// Set/Get the location of center of mass
virtual void setLocation(DtConstVector coords);
virtual DtConstVector location() const;

// Set/Get the velocity
virtual void setVelocity(DtConstVector coords);
virtual DtConstVector velocity() const;

// Set/Get the number of DIS sub-aggregates. Set returns DtFalse if the
// requested number is too large.
virtual DtBoolean setNumberOfDIS_Subaggregates(DtU16
newNumberOfAggregates);
virtual DtU16 numberOfDIS_Subaggregates() const;

// Set/Get the number of DIS entities. Set returns DtFalse if the
// requested number is too large.
virtual DtBoolean setNumberOfDIS_Entities(DtU16 newNumberOfEntities);
virtual DtU16 numberOfDIS_Entities() const;

// Set/Get the number of silent aggregates. Set returns DtFalse if the
// requested number is too large.
virtual DtBoolean setNumberOfSilentSubaggregates(
DtU16 newNumberOfAggregates);
virtual DtU16 numberOfSilentSubaggregates() const;

// Set/Get the number of silent entities. Set returns DtFalse if the
// requested number is too large.
virtual DtBoolean setNumberOfSilentEntities(DtU16 newNumberOfEntities);
virtual DtU16 numberOfSilentEntities() const;
```

```
// Set/Get a DIS sub-aggregate ID. Returns DtFalse if the index
// is invalid.
virtual DtBoolean setSubaggregateID(int index,
    const DtEntityIdentifier& aggID);
virtual DtBoolean getSubaggregateID(int index,
    DtEntityIdentifier& aggID) const;

// Set/Get a DIS entity ID. Returns DtFalse if the index is invalid.
virtual DtBoolean setEntityID(int index,
    const DtEntityIdentifier& entID);
virtual DtBoolean getEntityID(int index,
    DtEntityIdentifier& entID) const;

// Set/Get a silent aggregate list. Returns DtFalse if the index is
// invalid.
virtual DtBoolean setSilentAggregateList(int index,
    const DtSilentAggregateList& aggList);
virtual DtBoolean getSilentAggregateList(int index,
    DtSilentAggregateList& aggList) const;

// Set/Get a silent entity list. Returns DtFalse if the index is
// invalid.
virtual DtBoolean setSilentEntityList(int index,
    const DtSilentEntityList& entList);
virtual DtBoolean getSilentEntityList(int index,
    DtSilentEntityList& entList) const;

// Set/Get the number of datum records. Set returns DtFalse
// if the requested number is too large.
virtual DtBoolean setNumberOfDatumRecords(DtU32 newNumberOfRecords);
virtual DtU32 numberOfDatumRecords() const;

// Set/Get a variable datum record. Returns DtFalse if the index is
// invalid.
virtual DtBoolean setVariableDatumRecord(int index,
    const DtVariableDatumRecord& rec);
virtual DtBoolean getVariableDatumRecord(int index,
    DtVariableDatumRecord& rec) const;

// Get a constant reference to a variable datum record
virtual const DtVariableDatumRecord& variableDatumRecordConstRef(
    int index) const;

// Print the data contained by this PDU
virtual void printData() const;

// WARNING: These methods are obsolete but have been retained for
backward
// compatibility. These methods may be removed in a future release.
// Instead, the methods setVariableDatumRecord, getVariableDatumRecord,
// and variableDatumRecordConstRef should be used.

// Set/Get the datum length (in bits). Returns DtFalse if
// the index is invalid. Set may destroy data in the record.
```

```

    virtual DtBoolean setDatumBitLength(int index, DtU32
newRecordBitLength);
    virtual DtBoolean getDatumBitLength(int index, DtU32& length) const;

    // Set/Get the variable datum ID. Returns DtFalse if the index
    // is invalid.
    virtual DtBoolean setDatumID(int index, DtDatumParam value);
    virtual DtBoolean getDatumID(int index, DtDatumParam& value) const;

    // Set/Get datum value as a DtFloat64. Returns DtFalse if the index
    // is invalid or if the field length is not 64 bits.
    virtual DtBoolean setDatumValFloat64(int index, DtFloat64 value);
    virtual DtBoolean getDatumValFloat64(int index, DtFloat64& value) const;

    // Set/Get datum value as a DtFloat32. Returns DtFalse if the index
    // is invalid or if the field length is not 32 bits.
    virtual DtBoolean setDatumValFloat32(int index, DtFloat32 value);
    virtual DtBoolean getDatumValFloat32(int index, DtFloat32& value) const;

    // Set/Get datum value as a DtInt32. Returns DtFalse if the index
    // is invalid or if the field length is not 32 bits.
    virtual DtBoolean setDatumValInt32(int index, DtInt32 value);
    virtual DtBoolean getDatumValInt32(int index, DtInt32& value) const;

    // Set/Get datum value as a DtU32. Returns DtFalse if the index
    // is invalid or if the field length is not 32 bits.
    virtual DtBoolean setDatumValU32(int index, DtU32 value);
    virtual DtBoolean getDatumValU32(int index, DtU32& value) const;

    // Set/Get datum value as a DtInt16. Returns DtFalse if the index
    // is invalid or if the field length is not 16 bits.
    virtual DtBoolean setDatumValInt16(int index, DtInt16 value);
    virtual DtBoolean getDatumValInt16(int index, DtInt16& value) const;

    // Set/Get datum value as a DtU16. Returns DtFalse if the index
    // is invalid or if the field length is not 16 bits.
    virtual DtBoolean setDatumValU16(int index, DtU16 value);
    virtual DtBoolean getDatumValU16(int index, DtU16& value) const;

    // Set/Get datum value as a DtU8. Returns DtFalse if the index
    // is invalid or if the field length is not 8 bits.
    virtual DtBoolean setDatumValU8(int index, DtU8 value);
    virtual DtBoolean getDatumValU8(int index, DtU8& value) const;

    // Set/Get datum value as a byte array. Returns DtFalse if the index
    // is invalid. Data is copied to and from the network PDU structure
    // according to the datum length of the record.
    virtual DtBoolean setDatumValByteArray(int index, const DtU8* setFrom);
    virtual DtBoolean getDatumValByteArray(int index, DtU8* copyTo) const;

    // Get a pointer to datum value. Returns DtFalse if the index is
    invalid.
    virtual DtBoolean getDatumValPtr(int index, DtNetU8*& datumPtr);

public:

```

```
// Create a DtAggregateStatePdu.
// Caller is responsible for deletion.
static DtPdu* create(const DtNetPacket* initial,
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtAggregateStatePduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtAggregateStatePduCb cb, void* usr);

protected:

// Returns the internal PDU kind
virtual DtPduKind internalGetPduKind() const;

// Casts the netPdu element to our network struct
// Non-virtual since you can't override based on return type
virtual DtNetAggregateStatePdu* netPduStruct() const;

// Casts the 'this' pointer for changing elements in a const function
// Non virtual since you can't override based on return type
DtAggregateStatePdu* self() const;

// Returns a pointer to the first byte of variable length data
virtual char* var() const;

// Returns an offset from the beginning of variable length data to
// a particular subaggregate ID.
virtual int subaggregateID_Offset(int index) const;

// Returns an offset from the beginning of variable length data to
// a particular entity ID.
virtual int entityID_Offset(int index) const;

// Returns an offset from the beginning of variable length data to
// the padding following the entity ID records.
virtual int paddingOffset() const;

// Returns the number of padding bytes which must follow the entity
// ID records.
virtual int paddingBytes() const;

// Returns an offset from the beginning of variable length data to
// a particular silent aggregate list.
virtual int silentAggregateOffset(int index) const;

// Returns an offset from the beginning of variable length data to
// a particular silent entity system list.
virtual int silentEntityOffset(int index) const;

// Returns an offset from the beginning of variable length data to
// the number of variable datum records field.
```



```
virtual int numberOfDatumRecordsOffset() const;

// Returns an offset from the beginning of variable length data to
// a particular variable datum record.
virtual int datumRecordOffset(int index) const;

protected:

    // Size of PDU with zero data bytes.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
    DtEntityType myEntityType;
    DtVector myVector;
    DtTaitBryan myAngles;
    DtVariableDatumRecord myRefRec;
    DtVariableDatumRecord myDumRec;
};
#endif
#endif
```

aggregateSR.h

```
#if DtDISVERS >= 5

#include "baseEntitySR.h"
#include "hostStructs.h"
#include "mathTypes.h"

#include "aggEntList.h"
#include "aggAggList.h"
#include "varDatumRec.h"
#include "objDesList.h"

#include "chooseDefines.h"

// class DtAggregateStateRepository:
//
// Instances of DtAggregateStateRepository are used to maintain the
// state of an aggregate. It provides an interface which covers the
// various features which define the state of an aggregate.

class DtAggregateStateRepository : public DtBaseEntityStateRepository
{
public:

    // default constructor
    DtAggregateStateRepository();

    // destructor
    virtual ~DtAggregateStateRepository();

    // copy constructor
    DtAggregateStateRepository(const DtAggregateStateRepository& orig);

    // assignment operator
    DtAggregateStateRepository& operator=(
        const DtAggregateStateRepository& orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const;

    // Set/Get the aggregate's identifier. Calls down to base class.
    virtual void setAggregateID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& aggregateID() const;

    // Set/Get the force type.
    virtual void setForceId(DtForceType force);
    virtual DtForceType forceId() const;

    // Set/Get the aggregate state.
    virtual void setAggregateState(DtAggregateState value);
    virtual DtAggregateState aggregateState() const;
```

```
// Set/Get the aggregate type. Calls down to base class.
virtual void setAggregateType(const DtEntityType& type);
virtual const DtEntityType& aggregateType() const;

// Set/Get the formation.
virtual void setFormation(DtAggregateFormation value);
virtual DtAggregateFormation formation() const;

// Set/Get marking character set.
virtual void setMarkingCharacterSet(DtCharacterSet value);
virtual DtCharacterSet markingCharacterSet() const;

// Set/Get marking text (copies 31 bytes).
virtual void setMarkingText(DtU8* copyFrom);
virtual void getMarkingText(DtU8* copyTo) const;
virtual const char* markingText() const;

// Set/Get the dimensions.
virtual void setDimensions(DtConstVector dims);
virtual DtConstVector dimensions() const;

// Return the list of published entities in the aggregate.
const DtGlobalObjectDesignatorList& entities() const;
DtGlobalObjectDesignatorList* entities();

// Return the list of published subaggregates in the aggregate.
DtGlobalObjectDesignatorList* subAggregates();
const DtGlobalObjectDesignatorList& subAggregates() const;

// Set/Get the number of DIS sub-aggregates. Set returns DtFalse if the
// requested number is too large. Changing this number will erase all
// previous data. Can be done through subAggregates() instead.
virtual DtBoolean setNumberOfDIS_Subaggregates(int
newNumberOfAggregates);
virtual int numberOfDIS_Subaggregates() const;

// Set/Get the number of DIS entities. Set returns DtFalse if the
// requested number is too large. Changing this number will erase all
// previous data. Can be done through entities() instead.
virtual DtBoolean setNumberOfDIS_Entities(int newNumberOfEntities);
virtual int numberOfDIS_Entities() const;

// Set/Get the number of silent aggregates. Set returns DtFalse if the
// requested number is too large. Changing this number will erase all
// previous data.
virtual DtBoolean setNumberOfSilentSubaggregates(
    int newNumberOfAggregates);
virtual int numberOfSilentSubaggregates() const;

// Set/Get the number of silent entities. Set returns DtFalse if the
// requested number is too large. Changing this number will erase all
// previous data.
virtual DtBoolean setNumberOfSilentEntities(int newNumberOfEntities);
virtual int numberOfSilentEntities() const;
```

```
// Set/Get a DIS sub-aggregate ID. Returns DtFalse if the index
// is invalid. Can be done through subAggregates() instead.
virtual DtBoolean setSubaggregateID(int index,
    const DtGlobalObjectDesignator& aggID);
virtual DtBoolean getSubaggregateID(int index,
    DtGlobalObjectDesignator& aggID) const;

// Set/Get a DIS entity ID. Returns DtFalse if the index is invalid.
// Can be done through entities() instead.
virtual DtBoolean setEntityID(int index,
    const DtGlobalObjectDesignator& entID);
virtual DtBoolean getEntityID(int index,
    DtGlobalObjectDesignator& entID) const;

// Set/Get a silent aggregate list. Returns DtFalse if the index is
// invalid.
virtual DtBoolean setSilentAggregateList(int index,
    const DtSilentAggregateList& aggList);
virtual DtBoolean getSilentAggregateList(int index,
    DtSilentAggregateList& aggList) const;

// Set/Get a silent entity list. Returns DtFalse if the index is
// invalid.
virtual DtBoolean setSilentEntityList(int index,
    const DtSilentEntityList& entList);
virtual DtBoolean getSilentEntityList(int index,
    DtSilentEntityList& entList) const;

// Set the number of variable datum records. Returns DtFalse if the
number
// of records is too large. This will erase the contents of existing
// datum records.
virtual DtBoolean setNumberOfDatumRecords(int newNumberOfRecords);

// Get the number of variable datum records
virtual int numberOfDatumRecords() const;

// Set/Get a variable datum record. Returns DtFalse if the index is
// invalid.
virtual DtBoolean setVariableDatumRecord(int index,
    const DtVariableDatumRecord& rec);
virtual DtBoolean getVariableDatumRecord(int index,
    DtVariableDatumRecord& rec) const;

// Get a reference to a variable datum record.
virtual DtVariableDatumRecord& variableDatumRecordRef(int index);

// Get a constant reference to a variable datum record.
virtual const DtVariableDatumRecord& variableDatumRecordConstRef(
    int index) const;

public:

    // Print the state data.
```

```

    virtual void printData() const;

public:

    static DtAggregateStateRepository* create();

protected:

    // Cast off const-ness for trivial internal change.
    // Non-virtual since can't override on return type.
    DtAggregateStateRepository* self() const;

protected:

    DtForceType myForceID;
    DtAggregateState myAggregateState;
    DtAggregateFormation myFormation;
    DtCharacterSet myMarkingCharSet;

    // Aggregate state PDU marking is 31 bytes long, RPR marking struct is 11
    #ifdef DtHLA
    #define AGGREGATE_SR_MARKING_TEXT_LENGTH 11
    #else
    #define AGGREGATE_SR_MARKING_TEXT_LENGTH 31
    #endif

    char myMarkingText[AGGREGATE_SR_MARKING_TEXT_LENGTH];
    DtVector myDimensions;

    DtGlobalObjectDesignatorList myEntities;
    DtGlobalObjectDesignatorList mySubAggs;

    int myNumberOfSilentAggregateTypes;
    DtSilentAggregateList* mySilentAggregates;

    int myNumberOfSilentEntityTypes;
    DtSilentEntityList* mySilentEntities;

    int myNumberOfVariableDatumRecords;
    DtVariableDatumRecord* myVariableDatumRecords;

    DtVariableDatumRecord myDumRec;

};
#endif

```

antPatRec.h

```
#include "MachineTypes.h"

// class DtAntennaPatternRecord:
//
// Instances of DtAntennaPatternRecord are

class DtAntennaPatternRecord
{
public:

    // default constructor
    DtAntennaPatternRecord();

    // copy constructor
    DtAntennaPatternRecord(const DtAntennaPatternRecord& orig);

    // destructor
    virtual ~DtAntennaPatternRecord();

    // assignment operator
    DtAntennaPatternRecord& operator=
        (const DtAntennaPatternRecord& orig);

    // equality operator
    DtBoolean operator==
        (const DtAntennaPatternRecord& orig) const;

    // inequality operator
    DtBoolean operator!=
        (const DtAntennaPatternRecord& orig) const;

    // Set from network representation
    virtual void setFromNet(const void* net, int size);

    // Returns network representation, fills size with size of netRep.
    // Return value is pointer to internal object with same lifetime as
    // this. Do not delete.
    virtual const void* netRep(int* size = NULL) const;

    // Returns size of network representation.
    virtual int netSize() const;

    // Returns a new'ed clone of the instance. Caller deletes.
    virtual DtAntennaPatternRecord* clone() const;

    // prints the data
    virtual void printData() const;

protected:

    char* myNetRep;
```

```

    int myNetSize;
};
approximator.h

#include "mathTypes.h"
#include "vlTime.h"

// class DtApproximator:
//
// DtApproximator is an abstract base class. It is used to provide
// approximated position and orientation information for entities.
// Position and orientation information is approximated over time
// from the last set absolute (non-approximated) information.

class DtApproximator
{
public:

    // destructor
    virtual ~DtApproximator();

    // assignment operator
    DtApproximator& operator=(const DtApproximator& orig);

    // Create a copy of this object. Memory is allocated and it is the
    // responsibility of the caller to free the memory via delete.
    virtual DtApproximator* clone() const = 0;

    // Set absolute location.
    virtual void setLocation(DtConstVector loc,
        DtTime t = DtGetSimTime()) = 0;
    // Get the approximated location at the specified time.
    virtual DtConstVector approxLocation(
        DtTime t = DtGetSimTime()) const = 0;

    // Set absolute velocity.
    virtual void setVelocity(DtConstVector vel,
        DtTime t = DtGetSimTime()) = 0;
    // Get the approximated velocity at the specified time.
    virtual DtConstVector approxVelocity(
        DtTime t = DtGetSimTime()) const = 0;

    // Set absolute orientation.
    virtual void setOrientation(const DtTaitBryan& ori,
        DtTime t = DtGetSimTime()) = 0;
    // Get approximated orientation (euler angles) at the specified time.
    virtual const DtTaitBryan& approxEuler(
        DtTime t = DtGetSimTime()) const = 0;

    // Set/Get absolute acceleration.
    virtual void setAcceleration(DtConstVector acc,
        DtTime t = DtGetSimTime()) = 0;
    virtual DtConstVector acceleration(
        DtTime t = DtGetSimTime()) const = 0;

```

```
// Set/Get absolute rotational velocity.
virtual void setRotationalVelocity(DtConstVector vel,
    DtTime t = DtGetSimTime()) = 0;
virtual DtConstVector rotationalVelocity(
    DtTime t = DtGetSimTime()) const = 0;

// Get the approximated data body to geocentric coordinate matrix.
virtual DtConstDcm approxBodyToGeoc(DtTime t = DtGetSimTime()) const = 0;

protected:

    // Constructors are protected since this is an abstract base class.

    // default constructor
    DtApproximator();

    // copy constructor
    DtApproximator(const DtApproximator& orig);
};
```


apSpec.h

```
#include <stdlib.h>

const int DtMAX_PARAMS_PER_PART = 31;

typedef struct DtArtPartSpec
{
    int partType;
    int attachedType;
    int params[DtMAX_PARAMS_PER_PART];
} DtArtPartSpec;

const int DtAttachedToBase = 0;
#define DtNoArtParts ((DtArtPartSpec*) NULL)
#define DtNullArtPartSpec 0,0,{0}
```

arealObjStPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/arealObjStP.h"

#include "objectType.h"

// class DtArealObjectStatePdu:
//
// Instances of DtArealObjectStatePdu are used to communicate information
// about the addition/modification of a synthetic object which is
// geometrically
// anchored to the terrain with multiple points.

// DtArealObjectStatePdu callback function signature.
typedef void (*DtArealObjectStatePduCb)(
    DtArealObjectStatePdu* pdu, void* usr);

class DtArealObjectStatePdu : public DtPdu
{
public:

    // Constructors
    DtArealObjectStatePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtArealObjectStatePdu(const DtNetArealObjectStatePdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtArealObjectStatePdu(const DtArealObjectStatePdu& orig);

    // destructor
    virtual ~DtArealObjectStatePdu();

    // Create a DtArealObjectStatePdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtArealObjectStatePduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtArealObjectStatePduCb cb, void* usr);

    // Assignment operator
```

```

DtArealObjectStatePdu& operator=(const DtArealObjectStatePdu& orig);

// Set/Get object ID
virtual void setObjectID(const DtEntityIdentifier& id);
virtual const DtEntityIdentifier& objectID() const;

// Set/Get referenced object ID
virtual void setReferencedObjectID(const DtEntityIdentifier& id);
virtual const DtEntityIdentifier& referencedObjectID() const;

// Set/Get update number
virtual void setUpdateNumber(DtU16 field);
virtual DtU16 updateNumber() const;

// Set/Get the force ID
virtual void setForceID(DtForceType force);
virtual DtForceType forceID() const;

// Set/Get the modifications (by 8-bit integer)
virtual void setModifications(DtU8 value);
virtual DtU8 modifications() const;

// Set/Get the modifications (by field)
virtual void setModificationsField(
    DtModificationField field, DtBoolean value);
virtual DtBoolean modificationsField(
    DtModificationField field) const;

// Set/Get the object type
virtual void setObjectType(const DtObjectType& type);
virtual const DtObjectType& objectType() const;

// Set/Get appearance, the lower 16 bits (as an integer)
virtual void setAppearance(DtU16 app);
virtual DtU16 appearance() const;

// Set/Get appearance by attribute for the lower 16 bits
virtual void setAppearanceAttribute(
    DtAppearAttribute attr, DtAppearAttrVal val);
virtual DtAppearAttrVal appearanceAttribute(
    DtAppearAttribute attr) const;

// Set/Get specific appearance, the upper 32 bits (as a 32 bit integer)
virtual void setSpecificAppearance(DtU32 app);
virtual DtU32 specificAppearance() const;

// Set/Get specific appearance by attribute, the upper 32 bits
virtual void setSpecificAppearanceAttribute(
    const DtAppearAttribute attr, const DtAppearAttrVal val);
virtual DtAppearAttrVal specificAppearanceAttribute(
    const DtAppearAttribute attr) const;

// Set/Get the number of points
virtual DtBoolean setNumPoints(DtU16 num);
virtual DtU16 numPoints() const;

```

```
// Set/Get the requester ID
virtual void setRequesterID(const DtSimulationAddress& type);
virtual const DtSimulationAddress& requesterID() const;

// Set/Get the receiving ID
virtual void setReceivingID(const DtSimulationAddress& type);
virtual const DtSimulationAddress& receivingID() const;

// Set/Get a point location. Returns DtFalse if the index is out of
range.
virtual DtBoolean setPointLocation(int index,
    DtConstVector init);
virtual DtBoolean getPointLocation(int index,
    DtVectorRef record) const;

// Print the data contained by this PDU
virtual void printData() const;

protected:

// Returns the internal PDU kind
virtual DtPduKind internalGetPduKind() const;

// Casts the netPdu element to our network struct
virtual DtNetArealObjectStatePdu* netPduStruct() const;

// Casts the 'this' pointer for changing elements in a const function
// Non-virtual since you can't override based on return type
DtArealObjectStatePdu* self() const;

// Returns a pointer to the first byte of variable length data
virtual char* var() const;

protected:

// Size of PDU with zero parameter records.
static int theStaticSize;

protected:

// Temporary objects for reference returns
DtEntityIdentifier myEntityID;
DtObjectType myObjectType;
DtSimulationAddress mySimulationAddress;

};
#endif
#endif
```

artPart.h

```

#include "MachineTypes.h"
#include "mathTypes.h"
#include "apSpec.h"
#include "vlTime.h"

// Used only for internal representation of Art Param information

typedef struct DtInternalArtParam
{
    float myValue;
    int myParamNum;
    int myNext;
    DtTime myTimeSet;
} DtInternalArtParam;

// class DtArtPart:
//
// Instances of DtArtPart are used by DtArtPartList to handle individual
// articulated parts (i.e. this is a helper class to DtArtPartList).

// One Articulated Part
class DtArtPart
{
    friend class DtArtPartList;

public:
    // Set/Get the value for the specified parameter type.
    // Returns DtFalse if the parameter type has not been added,
    // returns DtTrue otherwise.
    virtual DtBoolean setVal(int paramType, DtFloat32 value);
    virtual DtBoolean getVal(int paramType, DtFloat32& value) const;

    // Get the value for the specified parameter type.
    // Note this does not check existence of the parameter type and
    // should only be called when you know that it does exist (as from
    // nextValidParamType()).
    virtual DtFloat32 val(int paramType) const;

    // Get the part type.
    virtual int partType() const;

    // Get the part number.
    virtual int partNum() const;

    // Get the part this part is attached to.
    virtual int partAttachedTo() const;

    // Get the number of parameters.
    virtual int numParams() const;

```

```
// Get the first valid (added) parameter type.
virtual int firstValidParamType() const;

// Get the next valid parameter type.
virtual int nextValidParamType(int paramType) const;

// Returns the index of the parameter in a multiple part list
// as is found in HLA messages and DIS PDUs.
virtual int paramNum(int paramType) const;

// Get the euler angles.
// Euler is filled with the angles phi, theta and psi of the part
// with respect to its neutral orientation.
virtual const DtTaitBryan& angles() const;
virtual void getAngles(DtTaitBryan& euler) const;

// Get the translations.
// Vec is filled in with a vector representing the current translation,
// in x, y and z of the part from its neutral position.
virtual DtConstVector translations() const;
virtual void getTranslations(DtVectorRef vec) const;

// Set/Get whether or not to do approximating.
virtual void setApproximating(DtBoolean onOff);
virtual DtBoolean approximating() const;

// Print the part data.
virtual void print();

public:

    // Get whether the parameter type is actually a rate.
    static int artParamIsRate(int paramType);

    // Get the part type from the part and parameter value.
    static int computePartType(int partPlusParam);

    // Get the parameter type from the part and parameter value.
    static int computeParameter(int partPlusParam);

protected:

    // Constructors are protected since this class is managed by
    // DtArtPartList.

    // default constructor
    DtArtPart();

    // constructor
    DtArtPart(const DtArtPartSpec& spec);

    // destructor
    virtual ~DtArtPart();

    // copy constructor
```

```

DtArtPart(const DtArtPart& orig);

// assignment operator
DtArtPart& operator=(const DtArtPart& orig);

// Set the parameter number of the specified parameter type.
virtual void setParamNum(int paramType, int num);

// Set the part number.
virtual void setPartNum(int num);

// Add the specified parameter type to the part.
// Note that this assumes that parameters are always added
// in order from lowest to highest.
virtual void addParam(int paramType);

// Find the next valid parameter type.
virtual int findNextValidParamType(int paramType) const;

// Get the approximated value.
virtual DtFloat32 approxVal(int paramType) const;

// Get the last set (absolute) value.
virtual DtFloat32 lastSetVal(int paramType) const;

// Get the time at which the value was last set.
virtual DtTime timeSet(int paramType) const;

// Cast off const-ness for trivial internal change.
// Non-virtual since can't override on return type.
DtArtPart* self() const;

protected:

    int myPartType;
    int myPartNum;
    int myPartAttachedTo;
    int myNumParams;
    // myVals is an array of DtMAX_PARAMS_PER_PART + 1 items
    // which are themselves pointers.
    DtInternalArtParam** myVals;
    int myFirstParam;
    int myLastParam;
    DtBoolean myApproxFlag;

    DtTaitBryan myAngles;
    DtVector myVector;
};

```

artPartList.h

```
#include "vList.h"
#include "artPart.h"
#include "NetStructs.h"

// class DtArtPartList:
//
// Instances of DtArtPartList are used to manage articulated parts.
// Articulated part spec lists are mainly used to generate these lists.

class DtArtPartList
{
public:

    // default constructor
    DtArtPartList();

    // constructor
    DtArtPartList(const DtArtPartSpec* specList);

    // destructor
    virtual ~DtArtPartList();

    // copy constructor
    DtArtPartList(const DtArtPartList& orig);

    // assignment operator
    DtArtPartList& operator=(const DtArtPartList& orig);

    // The init functions don't set values for the parameters,
    // they merely set up the part and parameter types.
    // Non-virtual since called from ctor.
    void init(const DtArtPartSpec* partSpecs);

#if DtHLA

#include "chooseDefines.h"

    // The init functions don't set values for the parameters,
    // they merely set up the part and parameter types. numParams
    // is the total number of records in netParams, which might include
    // attached parameter records that will be ignored.
    virtual void init(int numParams,
        const DtNetArtParamRecord* netParams);

    // sets values from net representation, but does not change the
    // set of valid parts and parameters. numParams
    // is the total number of records in netParams, which might include
    // attached parameter records that will be ignored.
    virtual void setValsFromNet(int numParams,
        const DtNetArtParamRecord* netParams);

#endif
}
```

```

    // initializes the part types and parameters, and sets values as well.
    // numParams is the total number of records in netParams, which might
    // include attached parameter records that will be ignored.
    virtual void setFromNet(int numParams,
        const DtNetArtParamRecord* netParams);

    // fills in netParams based on current values
    virtual void fillNetParams(DtNetArtParamRecord* netParams) const;

#endif    // DtHLA

#if    !DtHLA

#include "chooseDefines.h"

    // The init functions don't set values for the parameters,
    // they merely set up the part and parameter types. numParams
    // is the total number of records in netParams, which might include
    // attached parameter records that will be ignored.
    virtual void init(int numParams,
        const DtNetArtParamRecord* netParams);

    // sets values from net representation, but does not change the
    // set of valid parts and parameters. numParams
    // is the total number of records in netParams, which might include
    // attached parameter records that will be ignored.
    virtual void setValsFromNet(int numParams,
        const DtNetArtParamRecord* netParams);

    // initializes the part types and parameters, and sets values as well.
    // numParams is the total number of records in netParams, which might
    // include attached parameter records that will be ignored.
    virtual void setFromNet(int numParams,
        const DtNetArtParamRecord* netParams);

    // fills in netParams based on current values
    virtual void fillNetParams(DtNetArtParamRecord* netParams) const;

#endif // !DtHLA

    // removes all parts and parameters from list
    // Non-virtual since called from ctor/dtor.
    void clear();

    // Set/Get the specified part type's specified parameter type value.
    // Returns DtFalse if the part or parameter are not found, otherwise
    // returns DtTrue.
    virtual DtBoolean setArtParamVal(int partType,
        int paramType, DtFloat32 val);
    virtual DtBoolean getArtParamVal(int partType,
        int paramType, DtFloat32& val) const;

    // Get the number of articulated parts.
    virtual int numArtParts() const;

```

```
// Get the number of articulated parameters.
virtual int numArtParams() const;

// Get the articulated part from the part number.
virtual DtArtPart* artPartFromPartNum(int partNum) const;

// Get the first articulated part with the part type.
virtual DtArtPart* artPartFromPartType(int partType) const;

// returns 0 if there is no such part, otherwise the part number
virtual int partNumFromPartType(int partType) const;

// return the list of DtArtPartSpecs describing the parts in the list.
virtual const DtArtPartSpec* artPartSpecs() const;

// Print the articulated part list.
virtual void print() const;

public:

    // Fill the articulated part spec list from the net parameter
    // record list. numParams is the total number of records in
    // netParams, which might include attached parameter records that
    // will be ignored.
    #if DtHLA

    #include "chooseDefines.h"

    static void fillSpecList(
        int numParams,
        const DtNetArtParamRecord* netParams,
        DtArtPartSpec* specList);

    // Returns true if param is an articulated parameter record,
    // false if its an attached part record.
    static DtBoolean artNotAttached(const DtNetArtParamRecord* param);

    #endif // DtHLA

    #if !DtHLA

    #include "chooseDefines.h"

    static void fillSpecList(
        int numParams,
        const DtNetArtParamRecord* netParams,
        DtArtPartSpec* specList);

    // Returns true if param is an articulated parameter record,
    // false if its an attached part record.
    static DtBoolean artNotAttached(const DtNetArtParamRecord* param);

    #endif // !DtHLA

protected:
```

```
// Returns whether the part number is within bounds.
virtual DtBoolean partNumInBounds(int partNum) const;

// Cast off const-ness for trivial internal change.
// Non-virtual since can't override on return type.
DtArtPartList* self() const;

protected:

    DtArtPart** myPartList;
    int myNumArtParts;
    int myNumArtParams;
    DtArtPartSpec* mySpecList;
};
```

artPartThresh.h

```
#include "artPartList.h"

// class DtArtPartThresholder:
//
// DtArtPartThresholder is used to store art-part-related thresholds,
// and to perform some art part thresholding.

class DtArtPartThresholder
{
public:

    enum DtArtPartThresholdType
    {
        DtArtPartDfltThreshMode = -1,
        DtArtPartIndividualThresh = 0,
        DtArtPartAggregateThresh = 1
    };

public:

    // Constructor
    DtArtPartThresholder();

    // Destructor
    ~DtArtPartThresholder();

    // copy constructor
    DtArtPartThresholder(const DtArtPartThresholder& orig);

    // assignment operator
    DtArtPartThresholder& operator=(const DtArtPartThresholder& orig);

    // Set/Get rotation threshold in radians.
    // Negative number means use default.
    virtual void setRotationThreshold(DtReal thresh);
    virtual double rotationThreshold() const;

    // Returns the "RotationTrace" threshold
    virtual double rotationTraceThreshold() const;

    // Set/Get translation threshold in meters.
    // Negative number means use default.
    virtual void setTranslationThreshold(DtReal thresh);
    virtual double translationThreshold() const;

    // Set/Get fractional threshold (fraction between 0 and 1).
    // Negative number means use default.
    virtual void setFractionalThreshold(DtReal thresh);
    virtual double fractionalThreshold() const;
```

```

// Set/Get the theshold mode. Use DtArtPartDfltThreshMode
// to indicate use of the default.
virtual void setThreshMode(DtArtPartThresholdType mode);
virtual DtArtPartThresholdType threshMode() const;

// Get whether or not an aggregate threshold (such as position,
// translation, rotation, etc.) has been exceeded.
virtual DtBoolean artPartsExceedAggThreshold(
    const DtArtPartList& localList,
    const DtArtPartList& remoteList) const;

// Get whether or not an individual parameter threshold has been
// exceeded.
virtual DtBoolean artPartsExceedIndivThreshold(
    const DtArtPartList& localList,
    const DtArtPartList& remoteList) const;

// Get the thresholding value (functional, translational, or
// rotational) for the parameter type.
virtual float artPartThreshold(int paramType) const;

public:

    // Set/Get the theshold mode.
    static void setDfltThreshMode(DtArtPartThresholdType mode);
    static DtArtPartThresholdType dfltThreshMode();

    // Set/Get fractional threshold value.
    static void setDfltFractionalThreshold(DtFloat32 thresh);
    static DtFloat32 dfltFractionalThreshold();

    // Set/Get rotational threshold value.
    static void setDfltRotationThreshold(DtReal r);
    static double dfltRotationThreshold();

    // Set/Get translational threshold value.
    static void setDfltTranslationThreshold(DtReal t);
    static double dfltTranslationThreshold();

protected:

    DtArtPartThresholdType myThreshMode;
    float myFracThresh;
    double myTransThresh;
    double myRotThresh;
    double myTraceRotThresh;

    DtBoolean myUsingDfltMode;
    DtBoolean myUsingDfltTrans;
    DtBoolean myUsingDfltRot;
    DtBoolean myUsingDfltFrac;

protected:

    static DtArtPartThresholdType theDfltThreshMode;

```

```
static float theDfltFracThresh;  
static double theDfltTransThresh;  
static double theDfltRotThresh;  
static double theDfltTraceRotThresh;  
};
```

attPartList.h

```

#include "hostStructs.h"

// This constant can be passed to a VR-Link function that
// expects a list of attached part stations to indicate
// an empty list.
const int DtNoAttachedParts[1] = {0};

enum DtAttachedPartStatus
{
    DtAttachedPartSuccess = -2,
    DtNoSuchAttachStation = -1,
    DtPartNotAttached = 0,
    DtPartAttached = 1
};

// class DtAttachedPartList:
//
// Instances of DtAttachedPartList are lists of attached parts.

class DtAttachedPartList
{
public:

    // default constructor
    DtAttachedPartList();

    // copy constructor
    DtAttachedPartList(const DtAttachedPartList& orig);

    // assignment operator
    DtAttachedPartList& operator=(const DtAttachedPartList& orig);

    // destructor
    virtual ~DtAttachedPartList();

    // Set/Get the number of attached parts.
    // Set may return DtFalse if the number of requested parts is invalid.
    virtual DtBoolean setNumAttachedParts(int num);
    virtual int numAttachedParts() const;

    // Returns a zero-termination list of station numbers that are valid in
    // this attached part list.
    virtual const int* stationList() const;

    // Stations is a zero-terminated array of stations.
    // Init initializes the attPartList with the stations in the array.
    virtual void init(const int* stations);

    // Set/Get the station number for a part.
    // Returns DtFalse if the index is out of range.

```

```
virtual DtBoolean setStationNumber(int partIndex, int station);
virtual DtBoolean getStationNumber(int partIndex, int& station) const;

// Returns the station number for a part.
// This version of the function requires that partIndex be valid
// (between 0 and numAttachedParts()).
virtual int stationNumber(int partIndex) const;

// Set/Get the entity type for a part.
// Returns DtFalse if the index is out of range.
virtual DtBoolean setEntityType(int partIndex, const DtEntityType& type);
virtual DtBoolean getEntityType(int partIndex, DtEntityType& type) const;

// Returns the entityType for a part, by index.
// This version of the function requires that partIndex be valid
// (between 0 and numAttachedParts()).
virtual const DtEntityType& entityType(int partIndex) const;

// Set/Get the entity type for a part.
// Returns DtFalse if the index is out of range.
virtual DtAttachedPartStatus setEntityTypeAtStation(int station,
    const DtEntityType& type);
virtual DtAttachedPartStatus getEntityTypeAtStation(int station,
    DtEntityType& type) const;

// Returns the entityType at a particular station.
// If the list does not contain an entry for the station,
// DtEntityType::nullEntityType is returned;
virtual const DtEntityType& entityTypeAtStation(int station) const;

// After this call, this attached part list will reflect the information
// contained in netParams. Both the set of valid stations, and
// the entity types attached to those stations might be changed.
// numParams is the total number of records in netParams, which might
// include articulated parameter records that will be ignored.
virtual void unloadNetList(int numParams,
    const DtNetAttachedPart* partList);

// Load a list from network data representation.
// numAttachedParts() network part records will be written/read.
virtual void loadNetList(DtNetAttachedPart* partList) const;

// Get the network data length for this list.
virtual int netRecordLength() const;

// Returns true if we contain the exact same information as other.
DtBoolean equivalent(const DtAttachedPartList& other) const;

// Print the information contained in the record.
virtual void print() const;

protected:

    int myNumParts;
    // zero terminated list that should always be of length
```



```
// myNumParts + 1
int* myStationList;

// NULL if myNumArtParts = 0, otherwise has length = myNumParts.
DtEntityType* myTypeList;
};
```

baseEmbedDec.h

```
#if DtHLA

#ifndef DtBaseEmbeddedSystemDecoder_H_
#define DtBaseEmbeddedSystemDecoder_H_

#include "chooseDefines.h"
#include "hStateDecoder.h"

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLARE_EMBEDSYS_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtEmbeddedSystemRepository, attrName)

// class DtBaseEmbeddedSystemDecoder:
//
// Instances of DtBaseEmbeddedSystemDecoder are

class DtBaseEmbeddedSystemDecoder : public DtHlaStateDecoder
{
public:

    // default constructor
    DtBaseEmbeddedSystemDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtBaseEmbeddedSystemDecoder();

    // copy constructor
    DtBaseEmbeddedSystemDecoder(const DtBaseEmbeddedSystemDecoder& orig);

    // assignment operator
    DtBaseEmbeddedSystemDecoder& operator=
        (const DtBaseEmbeddedSystemDecoder& orig);

public:

    // Create a DtBaseEntityDecoder. Caller is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
```

```
DtDECLARE_EMBEDSYS_ATTR_DECODER(EntityID);
DtDECLARE_EMBEDSYS_ATTR_DECODER(HostObjectID);
DtDECLARE_EMBEDSYS_ATTR_DECODER(RelativePosition);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_BASE_EMBEDSYS_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtBaseEmbeddedSystemDecoder, DtEmbeddedSystemRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_BASE_EMBEDSYS_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_BASE_EMBEDSYS_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif //DtBaseEmbeddedSystemDecoder_H_
```

baseEmbedEnc.h

```
#if DtHLA

#ifndef DtBaseEmbeddedSystemEncoder_H_
#define DtBaseEmbeddedSystemEncoder_H_

#include "chooseDefines.h"
#include "hStateEncoder.h"

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_BASE_EMBEDSYS_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtEmbeddedSystemRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_BASE_EMBEDSYS_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtEmbeddedSystemRepository, attrName)

// class DtBaseEmbeddedSystemEncoder:
//
// Instances of DtBaseEmbeddedSystemEncoder are

class DtBaseEmbeddedSystemEncoder : public DtHlaStateEncoder
{
public:

    // default constructor
    DtBaseEmbeddedSystemEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtBaseEmbeddedSystemEncoder();

    // copy constructor
    DtBaseEmbeddedSystemEncoder(const DtBaseEmbeddedSystemEncoder& orig);

    // assignment operator
    DtBaseEmbeddedSystemEncoder& operator=(const
DtBaseEmbeddedSystemEncoder& orig);

public:

    // Create a DtBaseEntityEncoder. Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    DtDECLARE_BASE_EMBEDSYS_ATTR_CHECKER(EntityID);
```

```

DtDECLARE_BASE_EMBEDSYS_ATTR_CHECKER(HostObjectID);
DtDECLARE_BASE_EMBEDSYS_ATTR_CHECKER(RelativePosition);

DtDECLARE_BASE_EMBEDSYS_ATTR_ENCODER(EntityID);
DtDECLARE_BASE_EMBEDSYS_ATTR_ENCODER(HostObjectID);
DtDECLARE_BASE_EMBEDSYS_ATTR_ENCODER(RelativePosition);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_BASE_EMBEDSYS_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtBaseEmbeddedSystemEncoder, \
        DtEmbeddedSystemRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that check whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_BASE_EMBEDSYS_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtBaseEmbeddedSystemEncoder, \
        DtEmbeddedSystemRepository, \
        attrName, inspector)

#endif //DtBaseEmbeddedSystemEncoder_H_

```

baseEmittrDec.h

```
#if DtHLA

#ifndef DtBaseEmitterBeamDecoder_H_
#define DtBaseEmitterBeamDecoder_H_

#include "chooseDefines.h"
#include "hStateDecoder.h"

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLARE_EMITTERBEAM_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtEmitterBeamRepository, attrName)

// class DtBaseEmitterBeamDecoder:
//
// Instances of DtBaseEmitterBeamDecoder are

class DtBaseEmitterBeamDecoder : public DtHlaStateDecoder
{
public:

    // default constructor
    DtBaseEmitterBeamDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtBaseEmitterBeamDecoder();

    // copy constructor
    DtBaseEmitterBeamDecoder(const DtBaseEmitterBeamDecoder& orig);

    // assignment operator
    DtBaseEmitterBeamDecoder& operator=
        (const DtBaseEmitterBeamDecoder& orig);

public:

    // Create a DtBaseEntityDecoder. Caller is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
```

```

DtDECLARE_EMITTERBEAM_ATTR_DECODER(AzimuthCenter);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(AzimuthSweep);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(BeamFunction);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(BeamID);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(BeamParameterIndex);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(ElevationCenter);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(ElevationSweep);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(EmittingSystemID);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(ERP);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(EntityID);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(Frequency);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(FrequencyRange);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(PRF);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(Pulsewidth);
DtDECLARE_EMITTERBEAM_ATTR_DECODER(SweepSynch);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_BASE_EMITTERBEAM_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtBaseEmitterBeamDecoder, DtEmitterBeamRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_BASE_EMITTERBEAM_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif //DtBaseEmitterBeamDecoder_H_

```

baseEmittEnc.h

```
#if DtHLA

#ifndef DtBaseEmitterBeamEncoder_H_
#define DtBaseEmitterBeamEncoder_H_

#include "chooseDefines.h"
#include "hStateEncoder.h"

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtEmitterBeamRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtEmitterBeamRepository, attrName)

// class DtBaseEmitterBeamEncoder:
//
// Instances of DtBaseEmitterBeamEncoder are

class DtBaseEmitterBeamEncoder : public DtHlaStateEncoder
{
public:

    // default constructor
    DtBaseEmitterBeamEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtBaseEmitterBeamEncoder();

    // copy constructor
    DtBaseEmitterBeamEncoder(const DtBaseEmitterBeamEncoder& orig);

    // assignment operator
    DtBaseEmitterBeamEncoder& operator=(const DtBaseEmitterBeamEncoder&
orig);

public:

    // Create a DtBaseEntityEncoder. Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(AzimuthCenter);
```



```

DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(AzimuthSweep);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(BeamFunction);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(BeamID);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(BeamParameterIndex);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(ElevationCenter);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(ElevationSweep);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(EmittingSystemID);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(ERP);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(EntityID);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(Frequency);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(FrequencyRange);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(PRF);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(PulseWidth);
DtDECLARE_BASE_EMITTERBEAM_ATTR_CHECKER(SweepSynch);

DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(AzimuthCenter);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(AzimuthSweep);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(BeamFunction);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(BeamID);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(BeamParameterIndex);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(ElevationCenter);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(ElevationSweep);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(EmittingSystemID);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(ERP);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(EntityID);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(Frequency);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(FrequencyRange);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(PRF);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(PulseWidth);
DtDECLARE_BASE_EMITTERBEAM_ATTR_ENCODER(SweepSynch);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_BASE_EMITTERBEAM_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtBaseEmitterBeamEncoder, \
        DtEmitterBeamRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that check whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_BASE_EMITTERBEAM_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtBaseEmitterBeamEncoder, \
        DtEmitterBeamRepository, \
        attrName, inspector)

#endif //DtBaseEmitterBeamEncoder_H_

```

baseEntDec.hr

```
#if DtHLA

#include "hStateDecoder.h"

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLARE_ENTITY_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtEntityStateRepository, attrName)

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLAREAggregate_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtAggregateStateRepository, attrName)

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLARE_BASE_ENTITY_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtBaseEntityStateRepository, attrName)

// class DtBaseEntityDecoder:
//
// Instances of DtBaseEntityDecoder are used to decode a
// base entity state message into a state repository in an HLA
// (as opposed to DIS) environment.

class DtBaseEntityDecoder : public DtHlaStateDecoder
{
public:

    // constructor
    DtBaseEntityDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtBaseEntityDecoder();

    // copy constructor
    DtBaseEntityDecoder(const DtBaseEntityDecoder& orig);

    // assignment operator
    DtBaseEntityDecoder& operator=(const DtBaseEntityDecoder& orig);

public:

    // Create a DtBaseEntityDecoder. Caller is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
```

```

// Add decoding functions for the attributes.
// Note: Non-virtual since meant to be called from constructors to
//       add decoders.
void addMyDecoders();

// static decode<AttributeName> member functions.
DtDECLARE_BASE_ENTITY_ATTR_DECODER(AccelerationVector);
DtDECLARE_BASE_ENTITY_ATTR_DECODER(AngularVelocityVector);
DtDECLARE_BASE_ENTITY_ATTR_DECODER(DRAgorithm);
DtDECLARE_BASE_ENTITY_ATTR_DECODER(EntityType);
#if !DtrPR016
    DtDECLARE_BASE_ENTITY_ATTR_DECODER(EntityID);
#else
    DtDECLARE_BASE_ENTITY_ATTR_DECODER(FederateID);
#endif
DtDECLARE_BASE_ENTITY_ATTR_DECODER(IsFrozen);
DtDECLARE_BASE_ENTITY_ATTR_DECODER(Orientation);
DtDECLARE_BASE_ENTITY_ATTR_DECODER(Position);
DtDECLARE_BASE_ENTITY_ATTR_DECODER(VelocityVector);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DthlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_BASE_ENTITY_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtBaseEntityDecoder, DtBaseEntityStateRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_BASE_ENTITY_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_BASE_ENTITY_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif

```

baseEntEnc.h

```
#if DtHLA

#include "hStateEncoder.h"

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_ENTITY_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtEntityStateRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_ENTITY_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtEntityStateRepository, attrName)

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_AGGREGATE_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtAggregateStateRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_AGGREGATE_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtAggregateStateRepository, attrName)

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_BASE_ENTITY_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtBaseEntityStateRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_BASE_ENTITY_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtBaseEntityStateRepository, attrName)

// class DtBaseEntityEncoder:
//
// Instances of DtBaseEntityEncoder are used to encode a
// base entity state repository into a state message in an HLA
// (as opposed to DIS) environment.

class DtBaseEntityEncoder : public DtHlaStateEncoder
{
public:

    // constructor
    DtBaseEntityEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtBaseEntityEncoder();
}
```

```

    // copy constructor
    DtBaseEntityEncoder(const DtBaseEntityEncoder& orig);

    // assignment operator
    DtBaseEntityEncoder& operator=(const DtBaseEntityEncoder& orig);

public:

    // Create a DtBaseEntityEncoder. Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // Get whether or not rotational or translational thresholds have
    // been exceeded.
    static DtBoolean tpsiChanged(
        const DtBaseEntityStateRepository& stateRep,
        const DtBaseEntityStateRepository& asSeenByRemote);

protected:

    // static need<Attribute> member functions.
    DtDECLARE_BASE_ENTITY_ATTR_CHECKER(AccelerationVector);
    DtDECLARE_BASE_ENTITY_ATTR_CHECKER(AngularVelocityVector);
    DtDECLARE_BASE_ENTITY_ATTR_CHECKER(DRAlgorithm);
    DtDECLARE_BASE_ENTITY_ATTR_CHECKER(EntityType);
    #if !DtrPR016
        DtDECLARE_BASE_ENTITY_ATTR_CHECKER(EntityID);
    #else
        DtDECLARE_BASE_ENTITY_ATTR_CHECKER(FederateID);
    #endif
    DtDECLARE_BASE_ENTITY_ATTR_CHECKER(IsFrozen);
    DtDECLARE_BASE_ENTITY_ATTR_CHECKER(Orientation);
    DtDECLARE_BASE_ENTITY_ATTR_CHECKER(Position);
    DtDECLARE_BASE_ENTITY_ATTR_CHECKER(VelocityVector);

    // static encode<Attribute> member functions.
    DtDECLARE_BASE_ENTITY_ATTR_ENCODER(AccelerationVector);
    DtDECLARE_BASE_ENTITY_ATTR_ENCODER(AngularVelocityVector);
    DtDECLARE_BASE_ENTITY_ATTR_ENCODER(DRAlgorithm);
    DtDECLARE_BASE_ENTITY_ATTR_ENCODER(EntityType);
    #if !DtrPR016
        DtDECLARE_BASE_ENTITY_ATTR_ENCODER(EntityID);
    #else
        DtDECLARE_BASE_ENTITY_ATTR_ENCODER(FederateID);
    #endif
    DtDECLARE_BASE_ENTITY_ATTR_ENCODER(IsFrozen);
    DtDECLARE_BASE_ENTITY_ATTR_ENCODER(Orientation);
    DtDECLARE_BASE_ENTITY_ATTR_ENCODER(Position);
    DtDECLARE_BASE_ENTITY_ATTR_ENCODER(VelocityVector);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly

```

```
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_BASE_ENTITY_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtBaseEntityEncoder, \
        DtBaseEntityStateRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that check whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_BASE_ENTITY_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtBaseEntityEncoder, \
        DtBaseEntityStateRepository, \
        attrName, inspector)

#endif
```

baseEntitySR.h

```

#include "stateRep.h"
#include "hostStructs.h"
#include "mathTypes.h"
#include "tStampType.h"
#include "thresholder.h"

// class DtBaseEntityStateRepository:
//
// Instances of DtBaseEntityStateRepository are used to maintain basic state
// information needed by all kinds of entities.

class DtBaseEntityStateRepository : public DtStateRepository
{
public:

    // default constructor
    DtBaseEntityStateRepository();

    // destructor
    virtual ~DtBaseEntityStateRepository();

    // copy constructor
    DtBaseEntityStateRepository(const DtBaseEntityStateRepository& orig);

    // assignment operator
    DtBaseEntityStateRepository& operator=(
        const DtBaseEntityStateRepository& orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const;

    // Set/Get flag indicateing whether we're using absolute timestamps
    void setTimeStampType(DtTimeStampType type);
    DtTimeStampType timeStampType() const;

    // Set/Get the location of the center of mass in world coordinates.
    virtual void setLocation(DtConstVector location,
        DtTime timeNow = DtGetSimTime());
    virtual DtConstVector location(DtTime timeNow = DtGetSimTime()) const;
    // Return the value last passed to setLocation, regardless of
    // whether we have an approximator.
    virtual DtConstVector lastSetLocation() const;

    // Set/Get the velocity.
    virtual void setVelocity(DtConstVector Velocity,
        DtTime timeNow = DtGetSimTime());
    virtual DtConstVector velocity(
        DtTime timeNow = DtGetSimTime()) const;
    // Return the value last passed to setVelocity, regardless of
    // whether we have an approximator.
    virtual DtConstVector lastSetVelocity() const;

```

```
// Set/Get the acceleration.
virtual void setAcceleration(DtConstVector Acceleration,
    DtTime timeNow = DtGetSimTime());
virtual DtConstVector acceleration(
    DtTime timeNow = DtGetSimTime()) const;
// Return the value last passed to setAcceleration, regardless of
// whether we have an approximator.
virtual DtConstVector lastSetAcceleration() const;

// Set/Get the orientation.
virtual void setOrientation(const DtTaitBryan& orient,
    DtTime timeNow = DtGetSimTime());
virtual const DtTaitBryan& orientation(
    DtTime timeNow = DtGetSimTime()) const;
// Return the value last passed to setOrientation, regardless of
// whether we have an approximator.
virtual const DtTaitBryan& lastSetOrientation() const;

// Set/Get the rotational velocity.
virtual void setRotationalVelocity(DtConstVector BodyRates,
    DtTime timeNow = DtGetSimTime());
virtual DtConstVector rotationalVelocity(
    DtTime timeNow = DtGetSimTime()) const;
// Return the value last passed to setRotationalVelocity, regardless of
// whether we have an approximator.
virtual DtConstVector lastSetRotationalVelocity() const;

// Get the body to geocentric coordinate matrix.
virtual DtConstDcm bodyToGeoc(
    DtTime timeNow = DtGetSimTime()) const;

// Set/Get the dead reckoning algorithm.
virtual void setAlgorithm(DtDeadReckonTypes alg);
virtual DtDeadReckonTypes algorithm() const;

// Set/Get the entity's identifier.
virtual void setEntityId(const DtEntityIdentifier& id);
virtual const DtEntityIdentifier& entityId() const;

// Set/Get the entity type.
virtual void setEntityType(const DtEntityType& type);
virtual const DtEntityType& entityType() const;

#if DtHLA
    // Set/Get the federate ID (part of entity ID)
    virtual void setFederateId(const DtFederateId& id);
    virtual const DtFederateId& federateId() const;
#endif

#if DtDISVERS != 3
    // Set/Get whether frozen or not.
    virtual void setFrozen(DtBoolean onOff);
    virtual DtBoolean frozen() const;
#endif
```



```
// Get and set the approximator used to provide approximated position
// and orientation information for the entity. If your approximator
// is a DtDeadReckoner, use useDeadReckoner instead of setApproximator.
// If it's a smoother, use useSmoother. Caller
// of setApproximator has the responsibility to delete approx
// after the StateRep no longer needs it.
virtual void setApproximator(DtApproximator* approx);
virtual DtApproximator* approximator() const;

// Tells the ESR to use dr as its approximator, and to notify it
// when DR algorithm changes. Caller is responsible for deleting dr,
// but only after the StateRep no longer needs it.
virtual void useDeadReckoner(DtDeadReckoner* dr);

// Create and use a new (private) dead-reckoner as our approximator
virtual void useDeadReckoner();

// If the ESR is using a dead reckoner, returns it. Else returns NULL.
virtual DtDeadReckoner* usingDeadReckoner();

// Tells the ESR to use sm as its approximator, and to notify it
// when DR algorithm changes. Caller is responsible for deleting sm,
// but only after the StateRep no longer needs it.
virtual void useSmoother(DtSmoother* sm);

// Create and use a new (private) smoother as our approximator
virtual void useSmoother();

// If the ESR is using a smoother, returns it. Else returns NULL.
virtual DtSmoother* usingSmoother();

// Returns the SR's threshold, which contains thresholds needed
// for encoding
virtual DtThreshold* threshold();

// Const version
virtual const DtThreshold& threshold() const;

// Print the state data.
virtual void printData() const;
```

protected:

```
// Cast off const-ness for trivial internal change.
// Non-virtual since can't override on return type.
DtBaseEntityStateRepository* self() const;
```

protected:

```
// holds what's currently being used as our approximator
DtApproximator* myApproximator;

// holds what was most recently passed to useDeadReckoner
DtDeadReckoner* myDeadReckoner;
```

```
// holds what was most recently passed to useSmoother
DtSmoother* mySmoother;

// Created and deleted by this ESR if needed
DtDeadReckoner* myDfltDeadReckoner;
DtSmoother* myDfltSmoother;

DtVector myLocation;
DtVector myVelocity;
DtVector myAcceleration;

DtTaitBryan myOrientation;
DtVector myRotationalVelocity;

DtDeadReckonTypes myAlgorithm;

DtDcm myBodyToGeoc;
DtBoolean myBodyToGeocDirty;

DtTimeStampType myTimeStampType;

DtEntityIdentifier myEntityId;
#if DtHLA
    DtFederateId myFederateId;
#endif
    DtEntityType myEntityType;
#if DtDISVERS != 3
    DtBoolean myFrozen;
#endif

    DtThresholder myThresholder;
};
#if DtHLA
#endif
```

baseExConn.h

```

#include "vlTime.h"
#include "vlList.h"

// class DtBaseExerciseConn:
//
// DtBaseExerciseConn is an abstract base class. It provides an
// application's interface or connection to a network.

typedef void (*DtPostDrainCb)(void* usr);

class DtBaseExerciseConn
{
public:

    // destructor
    virtual ~DtBaseExerciseConn();

    // assignment operator
    DtBaseExerciseConn& operator=(const DtBaseExerciseConn& orig);

    // Send a state message to the network.
    virtual void send(const DtStateMsg& msg) = 0;

    // Send an interaction to the network.
    virtual void send(const DtInteraction& msg) = 0;

    // Send a state message to the network which has been
    // time stamped with the current time.
    virtual void sendStamped(const DtStateMsg& msg) = 0;

    // Send an interaction to the network which has been
    // time stamped with the current time.
    virtual void sendStamped(const DtInteraction& msg) = 0;

    // Read and process messages from the network until either no more
    // messages, or for timeout seconds, whichever happens first.
    // Will read nReadPerTimeCheck messages at a time before checking
    // for timeout.
    // Returns number of incoming events which have been processed.
    virtual int drainInput(DtTime timeout=-1,
        int nReadPerTimeCheck=100) = 0;

    // add/remove callbacks to be executed from within drainInput
    // right before it returns
    virtual void addPostDrainCallback(DtPostDrainCb cb, void *usr);
    virtual void removePostDrainCallback(DtPostDrainCb cb, void *usr);

protected:

    // Constructors are protected since this is an abstract base class.

```

```
// default constructor
DtBaseExerciseConn();

// copy constructor
DtBaseExerciseConn(const DtBaseExerciseConn& orig);

// should be called by derived class' drainInputs.
virtual void executePostDrainCbs();

protected:

    DtList myPostDrainCbs;
};
```

baseHashKey.h

```

#include "MachineTypes.h"
#include "str.h"

// class DtBaseHashKey:
//
// DtBaseHashKey is an abstract base class for hashkeys.

class DtBaseHashKey
{
public:

    // default constructor
    DtBaseHashKey();

    // destructor
    virtual ~DtBaseHashKey();

    // copy constructor
    DtBaseHashKey(const DtBaseHashKey& orig);

    // assignment operator
    DtBaseHashKey& operator=(const DtBaseHashKey& orig);

    // equality operator
    virtual DtBoolean operator==(const DtBaseHashKey& other) const;

    // equality operator
    virtual DtBoolean operator!=(const DtBaseHashKey& other) const;

public:

    // get the hashed value for this key
    virtual DtU32 hashValue() const;

    // make a copy of this hash key
    virtual DtBaseHashKey* clone() const = 0;

protected:

    // get the string representation of this key
    virtual const DtString& stringRep() const = 0;

    // compute the hashed value for this key
    virtual DtU32 computeHashValue() = 0;

protected:

    DtBoolean myHashValueIsComputed;
    DtU32 myHashValue;
};

```

baseMsg.h

```
#include "stdlib.h"
#include "stdio.h"

#define DtUSE_INTERNAL_BUFFER_REP NULL
#define DtUSE_INTERNAL_BUFFER (DtBufferPtr(NULL))

class DtBaseMsg
{
public:

    // Destructor
    virtual ~DtBaseMsg();

    // Copy constructor
    DtBaseMsg(const DtBaseMsg& msg,
              DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Assignment
    DtBaseMsg &operator=(const DtBaseMsg& msg);

    // Get network representation
    virtual const char* netRep() const;

    // Get size of message
    virtual int length() const;

protected:

    // Constructor
    // Object is not fully constructed until initMsg is called.
    DtBaseMsg();

    void initMsg(int size, DtBufferPtr bufferPtr);
    void initMsg(const void *initial,
int size, DtBufferPtr bufferPtr);

    // Changes the size of the message. If we are using an external
    // buffer, assumes that the full buffer size is big enough to
    // accommodate the new msg size. If using an internal buffer,
    // the call may result in allocating a new internal buffer, and copying
    // data from the old to the new buffer.
    virtual void changeSize(int size);

    // Inserts numBytes zeroed-out bytes at the indicated offset
    // into the net rep. Uses changeSize to increase the size of the
    // buffer.
    virtual void insertBytes(int offset, int numBytes);

    // Inserts numBytes zeroed-out bytes at the indicated address.
    // Assumes that address falls within the net rep.
    // Uses changeSize to increase the size of the buffer.
```

```
virtual void insertBytes(void *address, int numBytes);

// Deletes numBytes bytes at the indicated offset in the net rep.
// Uses change size to decrease the size of the buffer.
virtual void deleteBytes(int offset, int numBytes);

// Deletes numBytes bytes starting at the indicated address.
// Assumes that address falls within the net rep.
// Uses changeSize to decrease the size of the buffer.
virtual void deleteBytes(void *address, int numBytes);

// zeros out all bytes in the network representation
virtual void clearAll();

// Sets myInternalBuff to point to a malloc'ed buffer of
// size bytes. Sets myIntBuffSize to size. Returns myInternalBuff.
// Size must be a positive number.
virtual void *newInternalBuffer(int size);

// Called by changeSize. Derived versions can provide
// definitions that set the size field in network rep.
virtual void sizeChanged(int size) = 0;

protected:

    void *myNetMessage;
    void *myInternalBuff;
    int myIntBuffSize;
    int myMsgSize;
};
```

baseRef/Agg.h

```
#if DtDISVERS >= 5

#include "vllist.h"
#include "mathTypes.h"
#include "hostStructs.h"
#include "stateMsg.h"

#include "chooseDefines.h"

// class DtBaseReflectedAggregate:
//
// DtBaseReflectedAggregate is an abstract base class. It is used to
// maintain the representation of a simulation aggregate reflected
// across a network (i.e. this is not a full simulation running
// on this machine).

class DtBaseReflectedAggregate : public DtListedObj
{
public:

    // destructor
    virtual ~DtBaseReflectedAggregate();

    // assignment operator
    DtBaseReflectedAggregate& operator=(const DtBaseReflectedAggregate&
orig);

    // Get the aggregate's state repository.
    virtual DtAggregateStateRepository* aggregateStateRep() const = 0;

    // Get the aggregate's identifier.
    virtual const DtObjectId& id() const;

protected:

    // Constructors are protected since this is an abstract base class.

    // constructor
    DtBaseReflectedAggregate(const DtObjectId& id);

    // copy constructor
    DtBaseReflectedAggregate(const DtBaseReflectedAggregate& orig);

protected:

    DtObjectId myObjectId;

};

#endif
```


baseRefEnt.h

```

#include "vllist.h"
#include "mathTypes.h"
#include "hostStructs.h"
#include "stateMsg.h"

#include "vlprint.h"          // Andy - for a DtWarn() message

// class DtBaseReflectedEntity:
//
// DtBaseReflectedEntity is an abstract base class. It is used to
// maintain the representation of a simulation entity reflected
// across a network (i.e. this is not an full simulation running
// on this machine).

class DtBaseReflectedEntity : public DtListedObj
{
public:

    // destructor
    virtual ~DtBaseReflectedEntity();

    // assignment operator
    DtBaseReflectedEntity& operator=(const DtBaseReflectedEntity& orig);

    // Get the entity's state repository.
    virtual DtEntityStateRepository* entityStateRep() const = 0;

    // Get the entity's identifier.
    virtual const DtObjectId& id() const;

protected:

    // Constructors are protected since this is an abstract base class.

    // constructor
    DtBaseReflectedEntity(const DtObjectId& id);

    // copy constructor
    DtBaseReflectedEntity(const DtBaseReflectedEntity& orig);

protected:

    DtObjectId myObjectId;
};

```

beamAntRec.h

```
#include "disEnums.h"
#include "mathTypes.h"
#include "antPatRec.h"

// class DtBeamAntennaPatternRecord:
//
// Instances of DtBeamAntennaPatternRecord are beam antenna pattern records
// used in conjunction with DtRadioTransmitterStateRepository (TSR).
// A DtAntennaPatternRecord obtained using the TSR's antennaPattern()
// inspector can be used to initialize a DtBeamAntennaPatternRecord, whose
// inspectors you can then use to view the data (assuming the AntennaPattern
// Record really contained BeamAntennaPattern data.
// Similarly, a DtBeamAntennaPatternRecord can be filled out and passed to
// TSR's setAntennaPattern.

class DtBeamAntennaPatternRecord : public DtAntennaPatternRecord
{
public:

    // default constructor
    DtBeamAntennaPatternRecord();

    // copy constructor
    DtBeamAntennaPatternRecord(const DtBeamAntennaPatternRecord& orig);

    // destructor
    virtual ~DtBeamAntennaPatternRecord();

    // assignment operator
    DtBeamAntennaPatternRecord& operator=(
        const DtBeamAntennaPatternRecord& orig);

    // Initialize from DtAntennaPatternRecord
    DtBeamAntennaPatternRecord(const DtAntennaPatternRecord& orig);

    // Set/Get the beam direction
    virtual void setAngles(const DtTaitBryan &angles);
    virtual const DtTaitBryan &angles() const;

    // Set/Get the azimuth bandwidth and elevation
    virtual void setBeamwidth(DtFloat32 azimuth,
        DtFloat32 elevation);

    // Get the azimuth beam bandwidth
    virtual DtFloat32 azimuthBeamwidth() const;

    // Get the elevation beam width
    virtual DtFloat32 elevationBeamwidth() const;

    // Set/Get the beam reference system
    virtual void setReferenceSystem(const DtReferenceSys sys);
    virtual DtReferenceSys referenceSystem() const;
```

```

// Set electric field
virtual void setElectricField(DtFloat32 z, DtFloat32 x);

// Get electric field Z
virtual DtFloat32 electricFieldZ() const;

// Get electric field X
virtual DtFloat32 electricFieldX() const;

// Set/Get the phase
virtual void setPhase(DtFloat32 phase);
virtual DtFloat32 phase() const;

// print the class data
virtual void printData() const;

// Returns a clone of the instance
virtual DtAntennaPatternRecord* clone() const;

// Set from network representation
virtual void setFromNet(const void* net, int size);

// Returns network representation, fills size with size of netRep.
virtual const void* netRep(int* size = NULL) const;

// Returns size of network representation.
virtual int netSize() const;

// Set from network representation
virtual void setFromNetBeam(const DtNetBeamAntennaPatternRecord& net);

// Also sets from network representation. This function is here for
// backwards compatibility with pre-3.3 releases.
virtual void setFromNet(const DtNetBeamAntennaPatternRecord& net);

// Returns network representation
virtual const DtNetBeamAntennaPatternRecord& netBeam() const;

// Also returns network representation. This function is here for
// backwards compatibility with pre-3.3 releases.
virtual const DtNetBeamAntennaPatternRecord& networkRepresentation()
const;

protected:
    DtTaitBryan    myBeamDirection;
    DtFloat32      myAzimuthBandwidth;
    DtFloat32      myElevationBeamwidth;
    DtReferenceSys myReferenceSystem;
    DtFloat32      myEz;
    DtFloat32      myEx;
    DtFloat32      myPhase;
};

```

beamCbInfo.h

```
#if DtHLA || DtHLA_GATEWAY

#include "hlaDefines.h"
#include "vllist.h"

// Entity state callback function signature.
typedef void (*DtEmitterBeamCallbackFunction)(
    DtReflectedEmitterBeam* ent, void* userData);

#endif

// class DtEmitterBeamCallbackInfo:
//
// Instances of DtEmitterBeamCallbackInfo are

class DtEmitterBeamCallbackInfo : public DtListedObj
{
public:

    // default constructor
    DtEmitterBeamCallbackInfo();

    // constructor
    DtEmitterBeamCallbackInfo(
        DtEmitterBeamCallbackFunction cb,
        void* userData);

    // destructor
    virtual ~DtEmitterBeamCallbackInfo();

    // copy constructor
    DtEmitterBeamCallbackInfo(const DtEmitterBeamCallbackInfo& orig);

    // assignment operator
    DtEmitterBeamCallbackInfo& operator=
        (const DtEmitterBeamCallbackInfo& orig);

    // equality operators
    int operator==(const DtEmitterBeamCallbackInfo& orig) const;
    int operator!=(const DtEmitterBeamCallbackInfo& orig) const;

    // set/get the callback function
    virtual void setCallbackFunction(DtEmitterBeamCallbackFunction funct);
    virtual DtEmitterBeamCallbackFunction callbackFunction() const;

    // set/get the user data
    virtual void setUserData(void* data);
    virtual void* userData() const;

protected:
```

```
DtEmitterBeamCallbackFunction myCallbackFunction;  
void* myUserData;  
};
```

cartView.h

```
#include "LibMatrix.h"
// class DtCartesianView:
//
// Instances of DtCartesianView are used to create a Cartesian
// coordinate view of the geocentric coordinates in an
// Entity State Repository.

class DtCartesianView
{
public:

    // constructor
    DtCartesianView(DtBaseEntityStateRepository* ksr);

    DtCartesianView(DtBaseEntityStateRepository* ksr,
        DtConstVector originInGeoc,
        DtConstDcm geocToLocalRot);

    // destructor
    virtual ~DtCartesianView();

    // copy constructor
    DtCartesianView(const DtCartesianView& orig);

    // assignment operator
    DtCartesianView& operator=(const DtCartesianView& orig);

    // set the origin of the local system in geocentric coords
    virtual void setOffset(DtConstVector originInGeoc);

    // set the rotation matrix that converts from geocentric to local coords
    virtual void setRotation(DtConstDcm geocToLocalRot);

    // Set/Get the location in local coordinates.
    virtual void setLocation(DtConstVector Location);
    virtual DtConstVector location() const;

    // Set/Get the velocity.
    virtual void setVelocity(DtConstVector Velocity);
    virtual DtConstVector velocity() const;

    // Set/Get the orientation.
    virtual void setOrientation(const DtTaitBryan& Orient);
    virtual const DtTaitBryan& orientation() const;

    // Get the body to local coordinate matrix. This is just
    // a matrix representation of the euler angles returned by
    // orientation().
    virtual DtConstDcm bodyToLocal() const;

    // Set/Get the acceleration.
```

```
virtual void setAcceleration(DtConstVector Acceleration);
virtual DtConstVector acceleration() const;

// Set/Get the rotational velocity.
virtual void setRotationalVelocity(DtConstVector BodyRates);
virtual DtConstVector rotationalVelocity() const;

protected:

    // Cast off const-ness for trivial internal change.
    // Non-virtual since can't override on return type.
    DtCartesianView* self() const;

protected:

    DtBaseEntityStateRepository* myESR;
    DtCoordTransform myGeocToLocal;
    DtCoordTransform myLocalToGeoc;

    // used as return values by inspectors. Do not necessarily
    // hold current state.
    DtVector myLocalLoc;
    DtVector myLocalVel;
    DtVector myLocalAccel;
    DtTaitBryan myLocalOrient;
    DtDcm myBodyToLocal;
};
```

ClientSocket.h

```
#include "NetTypes.h"
#include "Socket.h"
#include "NetSocket.h"
#include "cliServer.h"

/*
 * ClientSocket uses the vrlink packet server to read and write packets
 * The destination address (incl. broadcast vs. internal-mode) are
 * controlled by the packet server.
 *
 * Constructor differences with respect to NetSocket:
 *   the destAddr is determined by the packet server (aka vrlink-daemon)
 *   all ClientSockets are presently ReadWrite
 */

#if DT_HAVE_DAEMON
class DtCsFilterTest : public DtFilterTest
{
public:
    DtCsFilterTest(int fd, DtU8 filterType, DtU32 clientId);
    virtual ~DtCsFilterTest();

    void setOperator(DtTestOperator op);
    void extendTest(int n, DtU32 *values);
    void reduceTest(int n, DtU32 *vals);
    // No-op. Real test done on server
    DtBoolean evalTest(char *, int , DtFilterType){return DtTrue;}

private:
    DtCsFilterMsg      filterMsg;
};
#endif

#define DtCsFlagsDefault (DtCsFlagsReadWrite |
DtCsFlagsServerChooseDropMode )

#define DtCsQDefaultSize DtCsUseDefaultSize

class DtClientSocket : public DtSocket
{
public:

    DtClientSocket(int port, int recvQSize =
DtCsQDefaultSize, unsigned short csFlags = DtCsFlagsDefault);

    DtClientSocket(int port, DtInetAddr destAddr, int recvQSize =
DtCsQDefaultSize, unsigned short csFlags = DtCsFlagsDefault);

    ~DtClientSocket();

    int sendTo(caddr_t packet, size_t size, DtInetAddr addr);
```



```

int DtRecv(caddr_t, size_t); /* this is slower as it does a memcpy() */
int DtRecv(caddr_t *buffptr, int *status=NULL);

int listenToMulticastGroup(DtInetAddr multicastAddr);
int dropMulticastGroup(DtInetAddr multicastAddr);
int setMcastTtl(int ttl) { return netSock->setMcastTtl(ttl); }

void setUnbundling(int onOff);
/* int isUnbundling(); is not currently implemented. The answer
   depends on the state of the Packet Server. */

DtInetAddr lastSourceInetAddr() { return sourceAddr; }
void DtBecomeNonBlocking(void);

int readFd()          { return recvFd; }
int writeFd()         { return sendFd; }
int isOk()            { return ok; }
DtBoolean isInternal();

void setPsMsgTimeout(int seconds);

void notifyMe();
void disconnectFromServer();
void setPacketServerInternal();
void setPacketServerExternal();
void pingClients();
void testClients();
void terminatePacketServer();
void requestServerState();
void requestServerCStates();
void setPacketServerUnbundling(int onOff);

#if DT_HAVE_DAEMON
DtFilterTest *exerciseId() { return exercise; }
DtFilterTest *protocolVersion() { return version; }
DtFilterTest *pduKind() { return kind; }
DtFilterTest *ipAddr() { return addr; }
#endif

static DtTime psMsgTimeout;

protected:

void init(int port, DtInetAddr destAddr, int recvQSize,
          unsigned short csFlags);

int timedRecv(caddr_t buf, size_t size, char *psKind);
void drainNotifySocket();
void checkGoAway();

#if DT_HAVE_DAEMON
DtShmMsgQueueReader *reader;
#endif

```

```
DtNetSocket *netSock;
DtBoolean notifyPending;

DtBoolean ok;
int recvFd;
int sendFd;
long clientId;

private:

void processControlMsg(DtCsMsg *msg);
DtCsMsg *getPsMsg (DtCsMsg *buf, char desiredKind);
void cleanup(void);

#if DT_HAVE_DAEMON
DtCsFilterTest *exercise;
DtCsFilterTest *version;
DtCsFilterTest *kind;
DtCsFilterTest *addr;
#endif

DtInetAddr destAddr;
char clientSocketPath[DtCsClientPathSize];
int serverPort;
DtBoolean internal;
DtInetAddr sourceAddr;
DtNetFloat64 pktBuff8[(DtMaxPacketSize + sizeof(DtCsServerPacket) + 7)
/ sizeof(DtNetFloat64) ];
char *shmAddr;
#if DT_HAVE_DAEMON
DtCsControlReqMsg *controlMsg;
#endif
};
```

collidePdu.h

```

#if !DtHLA

#include "hostStructs.h"
#include "interaction.h"
#include "packets/collisionP.h"
#include "mathTypes.h"
#include "exerciseConn.h"
#include "disEnums.h"

#include "chooseDefines.h"

const int DtCollisionStaticSize = 60;
typedef DtCollisionPdu DtCollisionInteraction;
typedef void (*DtCollisionInteractionCb)
    (DtCollisionInteraction *inter, void *usr);

class DtCollisionPdu : public DtInteraction
{
public:

//***** STANDARD CONSTRUCTORS *****

    DtCollisionPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCollisionPdu(const DtNetCollisionPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static void addCallback(DtExerciseConn *conn,
        DtCollisionInteractionCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtCollisionInteractionCb cb, void *usr);

//***** INSPECTOR FUNCTIONS *****

    const DtEntityIdentifier &issueId() const;
    const DtEntityIdentifier &collideId() const;
    const DtEventID &eventId() const;
    #if DtDISVERS >= 5
        DtCollisionType collisionType() const;
    #endif
    DtConstVector velocity() const;
    DtFloat32 mass() const;
    DtConstVector entityCoords() const;

//***** MUTATOR FUNCTIONS *****

    void setIssueId(const DtEntityIdentifier &id);
    void setCollideId(const DtEntityIdentifier &id);
    void setEventId(const DtEventID &id);

```

```
#if DtDISVERS >= 5
    void setCollisionType(DtCollisionType t);
#endif
    void setVelocity(DtConstVector veloc);
    void setMass(DtFloat32 mass);
    void setEntityCoords(DtConstVector coords);

    void printData() const;

protected:
    DtPduKind internalGetPduKind() const {return DtCollisionPduKind;}
    DtEntityIdentifier iss;
    DtEntityIdentifier col;
    DtEventID event;
    DtVector vel, ent;
};
#endif
```

collisElasPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/collisElasP.h"

// class DtCollisionElasticPdu:
//
// Instances of DtCollisionElasticPdu are used to send simulated elastic
// collision information.

// DtCollisionElasticPdu callback function signature.
typedef void (*DtCollisionElasticPduCb)(DtCollisionElasticPdu* pdu, void*
usr);

class DtCollisionElasticPdu : public DtPdu
{
public:

    // Constructors
    DtCollisionElasticPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCollisionElasticPdu(const DtNetCollisionElasticPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCollisionElasticPdu(const DtCollisionElasticPdu& orig);

    // destructor
    virtual ~DtCollisionElasticPdu();

    // Create a DtCollisionElasticPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtCollisionElasticPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtCollisionElasticPduCb cb, void* usr);

    // Assignment operator
    DtCollisionElasticPdu& operator=(const DtCollisionElasticPdu& orig);

    // Set/Get issuing entity identifier
    virtual void setIssuingEntityID(const DtEntityIdentifier& entity);

```

```
virtual const DtEntityIdentifier& issuingEntityID() const;

// Set/Get colliding entity identifier
virtual void setCollidingEntityID(const DtEntityIdentifier& entity);
virtual const DtEntityIdentifier& collidingEntityID() const;

// Set/Get event ID
virtual void setEventID(const DtEventID& event);
virtual const DtEventID& eventID() const;

// Set/Get contact velocity
virtual void setContactVelocity(DtConstVector velocity);
virtual DtConstVector contactVelocity() const;

// Set/Get mass
virtual void setMass(DtFloat32 value);
virtual DtFloat32 mass() const;

// Set/Get location of impact
virtual void setLocationOfImpact(DtConstVector coords);
virtual DtConstVector locationOfImpact() const;

// Set/Get collision intermediate result XX
virtual void setCollisionInterResultXX(DtFloat32 result);
virtual DtFloat32 collisionInterResultXX() const;

// Set/Get collision intermediate result XY
virtual void setCollisionInterResultXY(DtFloat32 result);
virtual DtFloat32 collisionInterResultXY() const;

// Set/Get collision intermediate result XZ
virtual void setCollisionInterResultXZ(DtFloat32 result);
virtual DtFloat32 collisionInterResultXZ() const;

// Set/Get collision intermediate result YY
virtual void setCollisionInterResultYY(DtFloat32 result);
virtual DtFloat32 collisionInterResultYY() const;

// Set/Get collision intermediate result YZ
virtual void setCollisionInterResultYZ(DtFloat32 result);
virtual DtFloat32 collisionInterResultYZ() const;

// Set/Get collision intermediate result ZZ
virtual void setCollisionInterResultZZ(DtFloat32 result);
virtual DtFloat32 collisionInterResultZZ() const;

// Set/Get unit surface normal
virtual void setUnitSurfaceNormal(DtConstVector coords);
virtual DtConstVector unitSurfaceNormal() const;

// Set/Get coefficient of restitution
virtual void setCoefficientOfRestitution(DtFloat32 coeff);
virtual DtFloat32 coefficientOfRestitution() const;

// Print the data contained by this PDU
```

```
    virtual void printData() const;

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    // Non-virtual since you can't override based on return type
    virtual DtNetCollisionElasticPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non virtual since you can't override based on return type
    DtCollisionElasticPdu* self() const;

protected:

    // Size of PDU with zero data bytes.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
    DtEventID myEventID;
    DtVector myVector;

};
#endif
#endif
```

commentPdu.h

```
#if !DtHLA

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"

#if DtDISVERS >= 5

#define DtCommentStaticSize 32

class DtCommentPdu;
typedef void (*DtCommentPduCb)(DtCommentPdu *pdu, void *usr);

class DtCommentPdu : public DtPduWithData
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtCommentPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCommentPdu(const DtNetCommentPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCommentPdu(const DtEntityIdentifier &sender,
const DtEntityIdentifier &receiver,
const char *message,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtCommentPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtCommentPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    /* none besides those inherited from DtPduWithData */

/***** MUTATOR FUNCTIONS *****/

    void setComment(const char *message, int size);

protected:

    DtPduKind internalGetPduKind() const {return DtCommentPduKind;}
    DtNetDataPduTail *dataSection() const;
};
typedef DtCommentPdu DtMessagePdu;

#endif // DtDISVERS */
#endif // !DtHLA
```


commentRPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"
#include "packets/commentRP.h"
#include "exerciseConn.h"

// class DtCommentRPdu:
//
// Instances of DtCommentRPdu are used to send/receive
// reliable comment PDUs.

// DtCommentRPdu callback function signature.
typedef void (*DtCommentRPduCb)(DtCommentRPdu* pdu, void* usr);

class DtCommentRPdu : public DtPduWithData
{
public:

    // Constructors
    DtCommentRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCommentRPdu(const DtNetCommentRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCommentRPdu(const DtEntityIdentifier& sender,
        const DtEntityIdentifier& receiver,
        const char* message,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCommentRPdu(const DtCommentRPdu& orig);

    // destructor
    virtual ~DtCommentRPdu();

    // Assignment operator
    DtCommentRPdu& operator=(const DtCommentRPdu& orig);

    // Set the comment.
    virtual void setComment(const char* message, int size);

public:

    // Create a DtCommentRPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtCommentRPduCb cb, void* usr);

```

```
static void removeCallback(DtExerciseConn* conn,
    DtCommentRPduCb cb, void* usr);

protected:

    // Set the comment.
    // Non-virtual since called from ctor.
    void setCommentInternal(const char* message, int size);

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Returns the data portion of the pdu.
    virtual DtNetDataPduTail* dataSection() const;

protected:

    static int theStaticSize;

};
#endif // DtDISVERS
#endif // !DtHLA
```

createEntRPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "simanPdu.h"
#include "packets/createEntRP.h"
#include "exerciseConn.h"

// class DtCreateEntityRPdu:
//
// Instances of DtCreateEntityRPdu are used to send/receive
// reliable create entity PDUs.

// DtCreateEntityRPdu callback function signature.
typedef void (*DtCreateEntityRPduCb)(DtCreateEntityRPdu* pdu, void* usr);

class DtCreateEntityRPdu : public DtSimanPdu
{
public:

    // Constructors
    DtCreateEntityRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCreateEntityRPdu(const DtNetCreateEntityRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCreateEntityRPdu(const DtEntityIdentifier& to,
        const DtEntityIdentifier& from,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCreateEntityRPdu(const DtCreateEntityRPdu& orig);

    // destructor
    virtual ~DtCreateEntityRPdu();

    // Assignment operator
    DtCreateEntityRPdu& operator=(const DtCreateEntityRPdu& orig);

    // Set/Get the request id.
    virtual void setRequestId(int id);
    virtual int requestId() const;

    // Set/Get the required reliability service.
    virtual void setReqRelService(DtRequiredReliabilityService value);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:

    // Create a DtCreateEntityRPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,

```

```
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
        DtCreateEntityRPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
        DtCreateEntityRPduCb cb, void* usr);

protected:

        // Returns the internal PDU kind
        virtual DtPduKind internalGetPduKind() const;

protected:

        static int theStaticSize;

};
#endif
#endif
```

createPdu.h

```

#if !DtHLA

#include "hostStructs.h"
#include "simanPdu.h"

#define DtCreateStaticSize 28

class DtCreateEntityPdu;
typedef void (*DtCreateEntityPduCb)(DtCreateEntityPdu *pdu, void *usr);

class DtCreateEntityPdu : public DtSimanPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtCreateEntityPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtCreateEntityPdu(const DtNetCreateEntityPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** ADDITIONAL CONSTRUCTORS *****/

    DtCreateEntityPdu(const DtEntityIdentifier &to,
        const DtEntityIdentifier &from,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtCreateEntityPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtCreateEntityPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

#if DtDISVERS != 3
    int requestId() const;
#endif

/***** MUTATOR FUNCTIONS *****/

#if DtDISVERS != 3
    void setRequestId(int id);
    void printData() const;
#endif

protected:
    DtPduKind internalGetPduKind() const {return DtCreateEntityPduKind;}
};
#endif

```

dAggDec.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "aggStatePdu.h"
#include "aggregateSR.h"

// class DtAggregateStateDecoder:
//
// Instances of DtAggregateStateDecoder are used to take state information
// from an aggregate state PDU and place it into an aggregate state
// repository.

class DtAggregateStateDecoder
{

public:

    // default constructor
    DtAggregateStateDecoder();

    // destructor
    virtual ~DtAggregateStateDecoder();

    // copy constructor
    DtAggregateStateDecoder(const DtAggregateStateDecoder& orig);

    // assignment operator
    DtAggregateStateDecoder& operator=(const DtAggregateStateDecoder& orig);

public:

    // Decode an aggregate state PDU and update a state repository
    virtual void decode(const DtAggregateStatePdu& pdu,
        DtAggregateStateRepository* stateRep);

};
#endif
#endif
```

dAggEnc.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "aggStatePdu.h"
#include "aggregateSR.h"

// class DtAggregateStateEncoder:
//
// Instances of DtAggregateStateEncoder are used to take state information
// from an aggregate state repository and place it into an aggregate state
// PDU.

class DtAggregateStateEncoder
{
public:

    // default constructor
    DtAggregateStateEncoder();

    // destructor
    virtual ~DtAggregateStateEncoder();

    // copy constructor
    DtAggregateStateEncoder(const DtAggregateStateEncoder& orig);

    // assignment operator
    DtAggregateStateEncoder& operator=(const DtAggregateStateEncoder& orig);

    // create a state message from a local and remote state repository
    virtual DtAggregateStatePdu* stateMsg(
        const DtAggregateStateRepository& stateRep,
        const DtAggregateStateRepository& asSeenByRemote);

    // force an state message to be generated
    virtual void setSendNeeded();

protected:

    // determine if a state message needs to be sent
    virtual DtBoolean sendNeeded(
        const DtAggregateStateRepository& stateRep,
        const DtAggregateStateRepository& asSeenByRemote);

    // fill out a state message which reflects a state repository
    virtual void fillOut(
        const DtAggregateStateRepository& stateRep,
        const DtAggregateStateRepository& asSeenByRemote,
        DtAggregateStatePdu* pdu);

protected:

```

```
DtAggregateStatePdu myAsPdu;  
int forceSendNeeded;  
};  
#endif  
#endif
```


dAggPub.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "exerciseConn.h"
#include "aggregateSR.h"
#include "dAggEnc.h"
#include "dAggDec.h"

#include "chooseDefines.h"

#ifdef __GNUC__ // g++
// Note: Here due to bug in g++ when using class static member function
// as a default argument. Don't call these otherwise.
#ifdef __GNUC__ // g++
#else
#undef gFindOwnId
#endif
#endif

#endif
#endif
```

dataPdu.h

```
#if !DtHLA

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"

#define DtDataStaticSize 40

class DtDataPdu;
typedef void (*DtDataPduCb)(DtDataPdu *pdu, void *usr);

class DtDataPdu : public DtPduWithData
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtDataPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtDataPdu(const DtNetDataPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtDataPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtDataPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    unsigned long requestId() const;

/***** MUTATOR FUNCTIONS *****/

    void setRequestId(unsigned long id);

    void printData() const;

protected:

    DtPduKind internalGetPduKind() const {return DtDataPduKind;}
    DtNetDataPduTail *dataSection() const;
};
#endif
```

dataQueryRPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "simanPdu.h"
#include "packets/dataQueryRP.h"
#include "exerciseConn.h"

// class DtDataQueryRPdu
//
// Instances of DtDataQueryRPdu are used to send/receive
// reliable data query PDUs.

// DtDataQueryRPdu callback function signature.
typedef void (*DtDataQueryRPduCb)(DtDataQueryRPdu* pdu, void* usr);

class DtDataQueryRPdu : public DtSimanPdu
{
public:

    // Constructors
    DtDataQueryRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtDataQueryRPdu(const DtNetDataQueryRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtDataQueryRPdu(const DtDataQueryRPdu& orig);

    // destructor
    virtual ~DtDataQueryRPdu();

    // Assignment operator
    DtDataQueryRPdu& operator=(const DtDataQueryRPdu& orig);

    // Set/Get the request id.
    virtual void setRequestId(unsigned long id);
    virtual unsigned long requestId() const;

    // Set/Get the time interval.
    virtual void setTimeInterval(DtTime seconds);
    virtual DtTime timeInterval() const;

    // Get the number of fixed fields.
    virtual unsigned long numFixedFields() const;

    // Get the number of variable fields.
    virtual unsigned long numVarFields() const;

    // Set/Get the fixed datum id at the specified index.
    virtual int setFixedDatumId(int index, DtDatumParam id);
    virtual DtDatumParam fixedDatumId(int index) const;

    // Set/Get the variable datum id at the specified index.

```

```
virtual int setVarDatumId(int index, DtDatumParam id);
virtual DtDatumParam varDatumId(int index) const;

// Set/Get the required reliability service.
virtual void setReqRelService(DtRequiredReliabilityService value);
virtual DtRequiredReliabilityService reqRelService() const;

// Initialize the fixed and variable fields.
virtual void initDatumIds(unsigned long numFixedFields,
    unsigned long numVarFields);

// Print the data contained by this PDU
virtual void printData() const;

public:

    // Create a DtDataQueryRPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtDataQueryRPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtDataQueryRPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

protected:

    static int theStaticSize;

};
#endif
#endif
```

dataRPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"
#include "packets/dataRP.h"
#include "exerciseConn.h"

// class DtDataRPdu
//
// Instances of DtDataRPdu are used to send/receive
// reliable data PDUs.

// DtDataRPdu callback function signature.
typedef void (*DtDataRPduCb)(DtDataRPdu* pdu, void* usr);

class DtDataRPdu : public DtPduWithData
{
public:

    // Constructors
    DtDataRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtDataRPdu(const DtNetDataRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtDataRPdu(const DtDataRPdu& orig);

    // destructor
    virtual ~DtDataRPdu();

    // Assignment operator
    DtDataRPdu& operator=(const DtDataRPdu& orig);

    // Set/Get the request id.
    virtual void setRequestId(unsigned long id);
    virtual unsigned long requestId() const;

    // Set/Get the required reliability service.
    virtual void setReqRelService(DtRequiredReliabilityService value);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:

    // Create a DtDataRPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

```

```
// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtDataRPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtDataRPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Returns the data portion of the net structure.
    virtual DtNetDataPduTail* dataSection() const;

protected:

    static int theStaticSize;

};
#endif
#endif
```

datumIdType.h

```
#include "disEnums.h"

#define DtTYPE_NOT_DEFINED 0
#define DtTYPE_BYTE_ARRAY 1
#define DtTYPE_FLOAT32 2
#define DtTYPE_FLOAT64 3
#define DtTYPE_INT32 4
#define DtTYPE_UNSIGNED32 5
#define DtTYPE_STRING 6

int DtDatumId2Type(DtDatumParam id);
int DtAddDatumId2TypeMapping(DtDatumParam id, int dataType,
    int warn = 0);
```

datumMap.h

```
#if !DtHLA

#include "vlconfig.h"
#include <stdio.h>

#if DT_HAVE_STD_H
#include <std.h>
#endif

#include "hashlist.h"
#include "disEnums.h"
#include "setDataPdu.h"
#include "queryPdu.h"
#include "dataPdu.h"

class DtDatumMapEntry
{
public:
    DtDatumMapEntry() {}
    virtual void setdata(const DtSetDataPdu &setDataPdu,
        int fixedOrVar, int index, void *usr) = 0;
    virtual void getdata(DtDataPdu *dataPdu, int fixedOrVar, int index,
        void *usr) = 0;
};

typedef void (*DtGlobalSetFcn)(const DtSetDataPdu &pdu, int fixedOrVar,
    int index, void *usr);
typedef void (*DtGlobalGetFcn)(DtDataPdu *pdu, int fixedOrVar,
    int index, void *usr);

class DtDmeGlobalFcns : public DtDatumMapEntry
{
public:
    DtDmeGlobalFcns(DtGlobalSetFcn setF, DtGlobalGetFcn getF)
    : DtDatumMapEntry(), setFcn(setF), getFcn(getF) {}

    void setdata(const DtSetDataPdu &setDataPdu,
        int fixedOrVar, int index, void *usr)
    {
        if (setFcn)
            (*setFcn)(setDataPdu, fixedOrVar, index, usr);
    }

    void getdata(DtDataPdu *dataPdu, int fixedOrVar, int index,
        void *usr)
    {
        if (getFcn)
            (*getFcn)(dataPdu, fixedOrVar, index, usr);
    }
protected:
    DtGlobalGetFcn getFcn;
    DtGlobalSetFcn setFcn;
};
```



```
};

class DtDatumMap {
public:
    DtDatumMap(void) : list(101) { }

    void setData(const DtSetDataPdu &data, void *usr);

    void fillDataPdu(DtDataPdu *dataPdu, void *usr);
    void initDataPdu(const DtSetDataPdu &setDataPdu, DtDataPdu *dataPdu);
    void initDataPdu(const DtDataQueryPdu &queryPdu, DtDataPdu *dataPdu);
    void makeResponseDataPdu(const DtSetDataPdu &setDataPdu,
        DtDataPdu *dataPdu, void *usr)
    {
        initDataPdu(setDataPdu, dataPdu);
        fillDataPdu(dataPdu, usr);
    }
    void makeResponseDataPdu(const DtDataQueryPdu &queryPdu,
        DtDataPdu *dataPdu, void *usr)
    {
        initDataPdu(queryPdu, dataPdu);
        fillDataPdu(dataPdu, usr);
    }

    DtDatumMapEntry *lookup(DtDatumParam id, int verbose=1);
    void install(DtDatumMapEntry *entry, DtDatumParam id);
    void remove(DtDatumParam id, int verbose=1);
protected:
    DtHashlist list;
};

#endif
```

datumRecSet.h

```
#include "hostStructs.h"
#include "datumIdType.h"
#include "baseMsg.h"
#include "packets/dataP.h"

enum DtDatumRetCode
{
    DtDatumValid,
    DtDatumBadLength,
    DtDatumBadId,
    DtDatumBadIndex
};

enum DtDatumRecordType
{
    DtFixed,
    DtVar
};

// This class has public data members for backwards
// compatibility with pre-VR-Link3.3 versions.
class DtVarDatumSpec
{
public:
    DtDatumParam param;
    int length;
};
#define DtNullDatumParam DtDatumParam(0)
#define DtNullVarDatumSpec DtNullDatumParam,0

class DtDatumRecordSet : public DtBaseMsg
{
public:

    // Constructor
    DtDatumRecordSet(DtBufferPtr bufferPtr = DtUSE_INTERNAL_BUFFER);

    // Constructor
    DtDatumRecordSet(const void* netRep, int size,
        DtBufferPtr bufferPtr = DtUSE_INTERNAL_BUFFER);

    // Destructor
    virtual ~DtDatumRecordSet();

    // Copy ctor
    DtDatumRecordSet(const DtDatumRecordSet& orig);

    // Assignment operator
    DtDatumRecordSet& operator=(const DtDatumRecordSet& orig);

    // returns number of fixed fields
```

```

virtual int numFixedFields() const;

// returns number of variable length fields
virtual int numVarFields() const;

// Returns the datum param found in the index'th variable
// datum record. Index is between 1 and numRecords. Returns
// DtDatumParam(0) if index is out of bounds.
virtual DtDatumParam varDatumId(int index) const;

// Returns the datum param found in the index'th fixed
// datum record. Index is between 1 and numRecords. Returns
// DtDatumParam(0) if index is out of bounds.
virtual DtDatumParam fixedDatumId(int index) const;

// Returns the datum param found in the index'th fixed or variable
// datum record, depending on whether fixedOrVar is DtFixed or DtVar.
// Index is between 1 and numRecords. Returns
// DtDatumParam(0) if index is out of bounds.
virtual DtDatumParam datumId(DtDatumRecordType fixedOrVar, int index)
const;

// Returns the number of bits of data contained in the value
// fields of the index'th fixed or variable datum record,
// depending on whether fixedOrVar is DtFixed or DtVar. If DtFixed,
// the answer is always 32.
virtual int datumSizeBits(DtDatumRecordType fixedOrVar, int index) const;

// Returns the number of bytes of data contained in the value
// fields of the index'th fixed or variable datum record,
// depending on whether fixedOrVar is DtFixed or DtVar. If DtFixed,
// the answer is always 4.
virtual int datumSizeBytes(DtDatumRecordType fixedOrVar, int index)
const;

// Looks through all fixed and variable records for one containing
// param as the parameter ID. If it finds one, fills supplied fixedOrVar
// pointer with DtFixed or DtVar, and index with the index to indicate
// which record was found, and fills retcode with DtDatumValid. If param
// was not found, retcode is filled with DtDatumBadId, and index and
// fixedOrVar are invalid.
virtual void getIndex(DtDatumParam param, DtDatumRecordType* fixedOrVar,
    int* index, DtDatumRetCode* retcode = NULL) const;

// ***** INSPECTING DATUM VALUES BY INDEX *****
// If you know the data type of the value contained in a particular
// record, use one of the following to return it. fixedOrVar (either
DtFixed
// or DtVar) and index (between 1 and numRecords) indicate which record
// you are inspecting. datumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It returns a pointer to a character string containing the
// exact bytes in the network representation that represents the value
// being inspected. The supplied retcode pointer will be filled with
// DtDatumValid if the value was successfully inspected, and

```

```
DtDatumBadLength
// if either the index was invalid, or the size of the data indicated that
// it could not be interpreted as the chosen data type.

    virtual DtFloat32 datumValFloat32(DtDatumRecordType fixedOrVar, int
index,
    DtDatumRetCode* retcode = NULL) const;
    virtual DtFloat64 datumValFloat64(DtDatumRecordType fixedOrVar, int
index,
    DtDatumRetCode* retcode = NULL) const;
    virtual unsigned long datumValUnsigned32(DtDatumRecordType fixedOrVar,
int index, DtDatumRetCode* retcode = NULL) const;
    virtual long datumValInt32(DtDatumRecordType fixedOrVar, int index,
DtDatumRetCode* retcode = NULL) const;
    virtual const char* datumValByteArray(DtDatumRecordType fixedOrVar,
int index) const;

// ***** INSPECTING DATUM VALUES BY PARAM ID *****
// If you know the data type of the value contained in a particular
// record, use one of the following to return it. The particular record
// inspected will be the first one containing param as the datum ID.
// datumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It returns a pointer to a character string containing the
// exact bytes in the network representation that represents the value
// being inspected. The supplied retcode pointer will be filled with
// DtDatumValid if the value was successfully inspected, and
DtDatumBadLength
// if either param was not found, or the size of the data indicated that
// it could not be interpreted as the chosen data type.

    virtual DtFloat32 datumValFloat32(DtDatumParam param,
DtDatumRetCode* retcode = NULL) const;
    virtual DtFloat64 datumValFloat64(DtDatumParam param,
DtDatumRetCode* retcode = NULL) const;
    virtual unsigned long datumValUnsigned32(DtDatumParam param,
DtDatumRetCode* retcode = NULL) const;
    virtual long datumValInt32(DtDatumParam param,
DtDatumRetCode* retcode = NULL) const;
    virtual const char* datumValByteArray(DtDatumParam param,
DtDatumRetCode* retcode = NULL) const;

// ***** MUTATOR FUNCTIONS *****

    // Initialize to reflect the fixed parameters
    // and variable parameters and sizes indicated. fixedParams is a NULL-
    // terminated array of DtDatumParams. specList is an array of
DtVarDatumSpecs
    // terminated with the DtNullVarDatumSpec constant. This call can be
made
    // instead of calls to setNumVarDatumRecords, setNumFixedDatumRecord,
    // and repeated calls to setDatumParam and setVarDataBytes.
    virtual void init(const DtDatumParam* fixedParams,
const DtVarDatumSpec* specList);
```

```

    // Sets the number of fixed or variable fields to newNum. Previously
    // existing fixed fields with indices greater than newNum are deleted.
    // If newNum exceeds the number of existing fields, trailing new fields
    // are created with zeros as datum params, sizes (for var only) and
values.
    virtual void setNumFixedFields(int newNum);
    virtual void setNumVarFields(int newNum);

    // Set the datum ID in the index'th fixed or variable datum record to
    // param, depending on whether fixedOrVar is DtFixed or DtVar. Supplied
    // retcode pointer is filled with DtDatumValid if successful, or
DtDatumBadIndex
    // if the index is not valid.
    virtual void setDatumParam(DtDatumRecordType fixedOrVar, int index,
DtDatumParam param,
        DtDatumRetCode* retcode = NULL);

    // Allocates length bytes for the index'th variable datum record.
    // Should be called before using one of the setDatumVal functions
    // to actually set the data. Supplied retcode pointer is set to
    // DtDatumValid if successful, or DtDatumBadIndex if the index
    // is not valid.
    virtual void setVarDataBytes(int index, int length, DtDatumRetCode*
retcode = NULL);

// ***** SETTING DATA VALUES BY INDEX *****
// Use one of the following to set the data value in a fixed or variable
datum
// record, depending on the data type of your data. fixedOrVar (either
DtFixed
// or DtVar) and index (between 1 and numRecords) indicate which record
// you are modifying. setDatumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It takes a pointer to a character string containing the
// exact bytes that should be used in the network representation.
// The supplied retcode pointer will be filled with
// DtDatumValid if the value was successfully set, DtDatumBadIndex if the
// index is invalid, and DtDatumBadLength if the size of the data indicated
that
// it could not be set from as the chosen data type.

    virtual void setDatumValFloat64(DtDatumRecordType fixedOrVar, int index,
DtFloat64 val,
        DtDatumRetCode* retcode = NULL);
    virtual void setDatumValFloat32(DtDatumRecordType fixedOrVar, int index,
DtFloat32 val,
        DtDatumRetCode* retcode = NULL);
    virtual void setDatumValInt32(DtDatumRecordType fixedOrVar, int index,
long val,
        DtDatumRetCode* retcode = NULL);
    virtual void setDatumValUnsigned32(DtDatumRecordType fixedOrVar, int
index,
        unsigned long val, DtDatumRetCode* retcode = NULL);
    virtual void setDatumValByteArray(DtDatumRecordType fixedOrVar, int
index,

```

```
    const char* val, DtDatumRetCode* retcode = NULL);

// ***** SETTING DATA VALUES BY PARAM ID *****
// Use one of the following to set the data value in a fixed or variable
datum
// record, depending on the data type of your data. The particular record
modified
// will be the first one containing param as its parameter ID.
// setDatumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It takes a pointer to a character string containing the
// exact bytes that should be used in the network representation.
// The supplied retcode pointer will be filled with
// DtDatumValid if the value was successfully set, DtDatumBadId if param
// is invalid, and DtDatumBadLength if the size of the data indicated that
// it could not be set from the chosen data type.

virtual void setDatumValFloat64(DtDatumParam param, DtFloat64 val,
    DtDatumRetCode* retcode = NULL);
virtual void setDatumValFloat32(DtDatumParam param, DtFloat32 val,
    DtDatumRetCode* retcode = NULL);
virtual void setDatumValInt32(DtDatumParam param, long val,
    DtDatumRetCode* retcode = NULL);
virtual void setDatumValUnsigned32(DtDatumParam param, unsigned long val,
    DtDatumRetCode* retcode = NULL);
virtual void setDatumValByteArray(DtDatumParam param, const char* val,
    DtDatumRetCode* retcode = NULL);

// prints the data
virtual void printData() const;

// Returns a pointer to the network representation, as might appear
// in a data PDU. Consists of numberOfFixedFields, numberOfVarFields,
// then the fixed and variable fields.
virtual DtNetDataPduTail* netDataRep() const;

// Return a pointer to the index'th fixed data record in the
// network representation. Index is between 1 and numRecords.
virtual DtNetFixDatum* nthFixedDataRecord(int index) const;

// Return a pointer to the index'th variable data record in the
// network representation. Index is between 1 and numRecords.
virtual DtNetVarDatum* nthVarDataRecord(int index) const;

// Sets the fixed data from a network representation of a
// fixed datum record array. Also pass the number of fixed fields.
virtual void setFromNetFixed(const char* fixedFields, int
numFixedFields);

// Sets the variable data from a network representation consisting of
// the variable length records concatenated together. Also pass
// the number of fields, and the total length of the variable data
records.
virtual void setFromNetVar(const char* varFields, int numVarFields,
    int totalVarFieldsLength);
```

protected:

```
// Really, should not be pure virtual in DtBaseMsg, but since
// it is, we need a null definition here.
virtual void sizeChanged(int size);

// Returns a pointer to the index'th fixed or variable record in the
// network representation, depending on whether fixedOrVar is DtFixed
// or DtVar.
virtual void* datumValPtr(DtDatumRecordType fixedOrVar, int index) const;
};

// For backwards compatibility:
#define DtFIXED DtFixed
#define DtVAR DtVar
#define DtDATUM_VALID DtDatumValid
#define DtDATUM_BAD_LENGTH DtDatumBadLength
#define DtDATUM_BAD_ID DtDatumBadId
#define DtDATUM_BAD_INDEX DtDatumBadIndex
```

dcm.h

```
#include "vlVector.h"

typedef const DtDcm &DtConstDcm;
typedef DtDcm &DtDcmRef;

class DtDcm
{
public:
    DtDcm();
    DtDcm(double a, double b, double c,
double d, double e, double f,
double g, double h, double i);

    DtDcm(DtConstVector a, DtConstVector b, DtConstVector c);
    DtDcm(const DtDcm &mat);
    DtDcm &operator=(const DtDcm &mat);

    operator DtVector* ();
    operator const DtVector* () const;

    DtVector &operator[](int i);
    const DtVector &operator[](int i) const;

public:
    static DtConstDcm zero();
    static DtConstDcm identity();

protected:
    DtVector rep[3];
};
// Inline functions
```


deadReckoner.h

```

#include "approximator.h"
#include "disEnums.h"

// class DtDeadReckoner:
//
// Instances of DtDeadReckoner are used to provide approximated
// position and orientation information for entities using various
// dead reckoning algorithms.
// Position and orientation information is approximated over time
// from the last set absolute (non-approximated) information.
// Functions that start with deadReckoned return dead-reckoned values.
// Functions that start with approx return approximated values - which
// default to the dead-reckoned values, but this might be overridden in
// classes derived from DtDeadReckoner.

class DtDeadReckoner : public DtApproximator
{
public:

    // default constructor
    DtDeadReckoner();

    // destructor
    virtual ~DtDeadReckoner();

    // copy constructor
    DtDeadReckoner(const DtDeadReckoner& orig);

    // assignment operator
    DtDeadReckoner& operator=(const DtDeadReckoner& orig);

    // Create a copy of this object. Memory is allocated and it is the
    // responsibility of the caller to free the memory via delete.
    virtual DtApproximator* clone() const;

    // Set absolute location.
    virtual void setLocation(DtConstVector loc,
        DtTime curTime = DtGetSimTime());
    // Get the approximated (i.e. dead-reckoned for base DtDeadReckoner)
    // location at the specified time.
    virtual DtConstVector approxLocation(
        DtTime curTime = DtGetSimTime()) const;
    // Get the dead-reckoned location at the specified time.
    virtual DtConstVector deadReckonedLocation(
        DtTime curTime = DtGetSimTime()) const;

    // Set absolute velocity.
    virtual void setVelocity(DtConstVector vel,
        DtTime curTime = DtGetSimTime());
    // Get the approximated (i.e. dead-reckoned for base DtDeadReckoner)
    // velocity at the specified time.

```

```
virtual DtConstVector approxVelocity(  
    DtTime curTime = DtGetSimTime()) const;  
// Get the dead-reckoned velocity at the specified time.  
virtual DtConstVector deadReckonedVelocity(  
    DtTime curTime = DtGetSimTime()) const;  
  
// Set absolute orientation.  
virtual void setOrientation(const DtTaitBryan& ori,  
    DtTime curTime = DtGetSimTime());  
// Get the approximated (i.e. dead-reckoned for base DtDeadReckoner)  
// orientation at the specified time.  
virtual const DtTaitBryan& approxEuler(  
    DtTime curTime = DtGetSimTime()) const;  
// Get dead-reckoned orientation (euler angles) at the specified time.  
virtual const DtTaitBryan& deadReckonedEuler(  
    DtTime curTime = DtGetSimTime()) const;  
  
// Set/Get absolute acceleration.  
virtual void setAcceleration(DtConstVector acc,  
    DtTime curTime = DtGetSimTime());  
virtual DtConstVector acceleration(  
    DtTime curTime = DtGetSimTime()) const;  
  
// Set/Get absolute rotational velocity.  
virtual void setRotationalVelocity(DtConstVector vel,  
    DtTime curTime = DtGetSimTime());  
virtual DtConstVector rotationalVelocity(  
    DtTime curTime = DtGetSimTime()) const;  
  
// Get the approximated (i.e. dead-reckoned for base DtDeadReckoner)  
// body to geocentric coordinate matrix.  
virtual DtConstDcm approxBodyToGeoc(  
    DtTime curTime = DtGetSimTime()) const;  
// Get the dead-reckoned body to geocentric coordinate matrix.  
virtual DtConstDcm deadReckonedBodyToGeoc(  
    DtTime curTime = DtGetSimTime()) const;  
  
// Set/Get the dead reckoning algorithm used for generating the  
// approximated values from the absolute values.  
virtual void setAlgorithm(DtDeadReckonTypes alg);  
virtual DtDeadReckonTypes algorithm() const;  
  
public:  
  
    // Calculate the dead reckon position information.  
    static void deadReckonPosition(DtTime dt, DtDeadReckonTypes algorithm,  
        DtConstVector oldPos, DtConstVector oldVel,  
        DtConstVector oldAccel,  
        DtVectorRef newVel, DtVectorRef newPos);  
  
    // Calculate the orientation matrix.  
    static void deadReckonOrientation(DtTime dt, DtDeadReckonTypes  
algorithm,  
        DtConstDcm old_b2w,  
        const DtDirectedRotRate& drr,
```

```
DtDcmRef new_b2w);
```

```
protected:
```

```
// If the last dead reckoned position is not valid for the
// specified time, recalculate it and same the validity time.
virtual void deadReckonLoc(DtTime curTime) const;

// If the last dead reckoned body to geocentri coordinate matrix
// is not valid for the specified time, recalculate it and same
// the validity time.
virtual void deadReckonBodyToGeoc(DtTime curTime) const;

// If the last dead reckoned orientation (euler angles) is not valid
// for the specified time, recalculate it and same the validity time.
virtual void deadReckonEuler(DtTime curTime) const;

// Get rid of constness on the "this" pointer in order to access
// non-const functions from const member functions (such as simple
// accessors). This should be used sparingly since it bypasses
// the const protection.
// Note: This is essentially like an operator and is
// non-virtual since can't override based on return type.
// Allows derived classes to also define this member.
DtDeadReckoner* self() const;
```

```
protected:
```

```
// Returns 1 if the dead reckoning algorithm is a rotational one,
// 0 otherwise.
static int isRotational(DtDeadReckonTypes algorithm);
```

```
protected:
```

```
DtVector myLocAtTimeOfValidity;
DtVector myVelAtTimeOfValidity;
DtTime mySimTimePosDataValid;
DtDcm myBodyToGeocAtTimeOfValidity;
DtTime mySimTimeOrientDataValid;

DtVector myDRLoc;
DtVector myDRVel;
DtTime myDRPostTime;
DtDcm myDRBodyToGeoc;
DtTime myDRBodyToGeocTime;
DtTaitBryan myDREuler;
DtTime myDREulerTime;
DtDirectedRotRate myDirectedRotRate;
DtDeadReckonTypes myAlgorithm;
DtVector myAccel;
DtBodyRates myRotVel;
};
```

decFactory.h

```
#if DtHLA

#include "hStateDecoder.h"
#include "MachineTypes.h"

// class DtStateDecoderFactory:
//
// Instances of DtStateDecoderFactory are used to create decoders which
// know how to decode an HLA state message (or network representation)
// into a state repository (or host representation).

class DtStateDecoderFactory
{
public:

    // constructor
    DtStateDecoderFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtStateDecoderFactory();

    // copy constructor
    // Copies across added decoder creation functions.
    DtStateDecoderFactory(const DtStateDecoderFactory& orig);

    // assignment operator
    // Copies across added decoder creation functions.
    DtStateDecoderFactory& operator=(const DtStateDecoderFactory& orig);

    // Create a decoder for the specified object class.
    // Note: Caller is responsible for deleting.
    // Returns NULL if decoder creation function is not in the factory.
    virtual DtHlaStateDecoder* createDecoder(RTI::ObjectClassHandle hand);

    // Add a decoder prototype to the factory. The decoder prototype
    // will be cloned by createDecoder when called for the appropriate
    // handle. Caller deletes the decoder, but AFTER the decoder
    // factory no longer needs it. Returns any previous decoder which was
    // replaced by the add.
    virtual DtHlaStateDecoder* addDecoder(
        RTI::ObjectClassHandle hand,
        DtHlaStateDecoder* decoder);

    // This function is used to add an attribute decoder function
    // to several prototype decoders all at once - namely, those
    // associated with the object class denoted by hand, as well
    // as all of its subclasses. If applyToSubclasses is DtFalse,
    // then the function is only registered with the one decoder
    // prototype, making it equivalent to
    // decoder(hand)->addDecoder(hand, decoder));
    virtual void addAttributeDecoder(
```

```

    RTI::ObjectClassHandle classHand,
    RTI::AttributeHandle attrHand,
    DtHlaStateDecoder::DtAttributeDecoder decoder,
    DtBoolean applyToSubclasses = DtTrue);

// Same as above, but identify class by name
void addAttributeDecoder(
    char* className,
    RTI::AttributeHandle attrHand,
    DtHlaStateDecoder::DtAttributeDecoder func,
    DtBoolean applyToSubclasses = DtTrue);

// Same as above, but identify attr by name
void addAttributeDecoder(
    RTI::ObjectClassHandle classHand,
    char* attrName,
    DtHlaStateDecoder::DtAttributeDecoder func,
    DtBoolean applyToSubclasses = DtTrue);

// Same as above, but identify class and attr by name
void addAttributeDecoder(
    char* className,
    char* attrName,
    DtHlaStateDecoder::DtAttributeDecoder func,
    DtBoolean applyToSubclasses = DtTrue);

// Returns the decoder prototype associated with handle by a previous
// call to addDecoder.
virtual DtHlaStateDecoder* decoder(RTI::ObjectClassHandle hand);

protected:

    // myProtos is an array of DtHlaObject::maxObjectClasses() items.
    DtHlaStateDecoder** myProtos;
    DtExerciseConn* myExConn;
};
#endif

```

dEntIdHashLst.h

```
#if !DtHLA

#include "hashlist.h"

class DtEntityIdHashlist : public DtIntrusiveHashlist
{
public:

    DtEntityIdHashlist();
    virtual unsigned int hash(DtHashKey k) const;
    virtual int isEqual(DtHashKey key1, DtHashKey key2) const;
};
#endif
```

dEntityDec.h

```

#if !DtHLA

#include "esPdu.h"
#include "entitySR.h"

// class DtEntityStateDecoder:
//
// Instances of DtEntityStateDecoder are objects capable of decoding
// EntityState PDUs into a DtEntityStateRepository

class DtEntityStateDecoder
{
public:

    // constructor
    DtEntityStateDecoder(DtExerciseConn* exConn);

    // destructor
    virtual ~DtEntityStateDecoder();

    // copy constructor
    DtEntityStateDecoder(const DtEntityStateDecoder& orig);

    // assignment operator
    DtEntityStateDecoder& operator=(const DtEntityStateDecoder& orig);

    // decode a PDU into an EntityStateRepository
    virtual void decode(const DtEntityStatePdu& pdu,
        DtEntityStateRepository* stateRep);

protected:

    // decode the articulated parts portion of a
    // PDU into an EntityStateRepository
    virtual void decodeArtParts(
        const DtEntityStatePdu& pdu,
        DtEntityStateRepository* stateRep);

    // decode the attached parts portion of a
    // PDU into an EntityStateRepository
    virtual void decodeAttachedParts(
        const DtEntityStatePdu& pdu,
        DtEntityStateRepository* stateRep);

protected:

    DtExerciseConn* myExConn;
};
#endif

```

dEntityEnc.h

```
#if !DtHLA

#include "esPdu.h"
#include "entitySR.h"

class DtEntityStateEncoder
{
public:

    // Constructor
    DtEntityStateEncoder();

    virtual DtEntityStatePdu*
        stateMsg(const DtEntityStateRepository& stateRep,
            const DtEntityStateRepository& asSeenByRemote);

    // Force the generation of an entity state PDU next time
    // stateMsg is called even if one would not otherwise be needed.
    virtual void setSendNeeded();

protected:

    // Returns true if a PDU needs to be sent, (based on thresholds, etc)
    // false otherwise.
    virtual DtBoolean sendNeeded(const DtEntityStateRepository& stateRep,
        const DtEntityStateRepository& asSeenByRemote);

    // Copy current ESR data to the pdu.
    virtual void fillOut(const DtEntityStateRepository& stateRep,
const DtEntityStateRepository& asSeenByRemote,
DtEntityStatePdu* pdu);

    // Returns true if art part info is out of date, false otherwise.
    virtual DtBoolean artPartsNeedUpdating(
        const DtEntityStateRepository& stateRep,
        const DtEntityStateRepository& asSeenByRemote);

    // Returns true if att part info is out of date, false otherwise.
    virtual DtBoolean attPartsNeedUpdating(
        const DtEntityStateRepository& stateRep,
        const DtEntityStateRepository& asSeenByRemote);

    // Copies current articulated and attached part info
    // from the ESR to the PDU.
    virtual void fillOutArtAndAttParts(
        const DtEntityStateRepository& stateRep,
        const DtEntityStateRepository& asSeenByRemote,
        DtEntityStatePdu* pdu);

protected:
```



```
    DtEntityStatePdu myEsPdu;  
    int myForceSendNeeded;  
};  
#endif
```

dEntityPub.h

```
#if !DtHLA

#include "exerciseConn.h"
#include "entitySR.h"
#include "vlprint.h"
#include "dEntityEnc.h"
#include "dEntityDec.h"

#include "chooseDefines.h"

#ifdef __GNUC__ // g++
// Note: Here due to bug in g++ when using class static member function
// as a default argument. Don't call these otherwise.
#ifdef __GNUC__ // g++
#else
#undef gFindOwnId
#undef gGuiseSameAsType
#endif
#endif

#endif // !DtHLA
```

designatePdu.h

```

#if !DtHLA

#if DtDISVERS != 3

#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "packets/designateP.h"

#define DtDesignatorStaticSizeVERS5 88
#define DtDesignatorStaticSizeVERS4 72
#if DtDISVERS >= 5
#define DtDesignatorStaticSize DtDesignatorStaticSizeVERS5
#else
#define DtDesignatorStaticSize DtDesignatorStaticSizeVERS4
#endif

class DtDesignatorPdu;
typedef void (*DtDesignatorPduCb)(DtDesignatorPdu *pdu, void *usr);

class DtDesignatorPdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtDesignatorPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtDesignatorPdu(const DtNetDesignatorPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtDesignatorPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtDesignatorPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &designatingEntityId() const;
    DtDesignatorCodeName codeName() const;
    const DtEntityIdentifier &designatedEntityId() const;
    DtDesignatorCode designatorCode() const;
    float power() const;
    float wavelength() const;
    DtConstVector designatorSpotRelative() const;
    DtConstVector designatorSpotWorld() const;
#if DtDISVERS >= 5
    DtDeadReckonTypes deadReckoningAlgorithm() const;
    DtConstVector linearAcceleration() const;
#endif
#endif

```

```
/****** MUTATOR FUNCTIONS *****/

void setDesignatingEntityId(const DtEntityIdentifier &id);
void setCodeName(DtDesignatorCodeName name);
void setDesignatedEntityId(const DtEntityIdentifier &id);
void setDesignatorCode(DtDesignatorCode code);
void setPower(float power);
void setWavelength(float wavelength);
void setDesignatorSpotRelative(DtConstVector loc);
void setDesignatorSpotWorld(DtConstVector loc);
#if DtDISVERS >= 5
void setDeadReckoningAlgorithm(DtDeadReckonTypes alg);
void setLinearAcceleration(DtConstVector acc);
#endif
void printData() const;
protected:
DtPduKind internalGetPduKind() const {return DtDesignatorPduKind;}
DtEntityIdentifier theId;
DtVector loc;
};
#endif // DtDISVERS
#endif // DtHLA
```

designatorDec.h

```

#if DtHLA

#ifndef DtDesignatorDecoder_H_
#define DtDesignatorDecoder_H_

#include "chooseDefines.h"
#include "baseEmbedDec.h"

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLARE_DESIGNATOR_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtDesignatorRepository, attrName)

// class DtDesignatorDecoder:
//
// Instances of DtDesignatorDecoder are

class DtDesignatorDecoder : public DtBaseEmbeddedSystemDecoder
{
public:

    // default constructor
    DtDesignatorDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtDesignatorDecoder();

    // copy constructor
    DtDesignatorDecoder(const DtDesignatorDecoder& orig);

    // assignment operator
    DtDesignatorDecoder& operator=(const DtDesignatorDecoder& orig);

public:

    // Create a DtBaseEntityDecoder. Caller is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
    DtDECLARE_DESIGNATOR_ATTR_DECODER(CodeName);

```

```
DtDECLARE_DESIGNATOR_ATTR_DECODER(DesignatedObject);
DtDECLARE_DESIGNATOR_ATTR_DECODER(DesignatorCode);
DtDECLARE_DESIGNATOR_ATTR_DECODER(DesignatorID);
DtDECLARE_DESIGNATOR_ATTR_DECODER(DesignatorPower);
DtDECLARE_DESIGNATOR_ATTR_DECODER(DesignatorSpotLocation);
DtDECLARE_DESIGNATOR_ATTR_DECODER(DesignatorWavelength);
DtDECLARE_DESIGNATOR_ATTR_DECODER(DRAgorithm);
DtDECLARE_DESIGNATOR_ATTR_DECODER(RelativeSpotLocation);
DtDECLARE_DESIGNATOR_ATTR_DECODER(SpotLinearAcceleration);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_DESIGNATOR_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtDesignatorDecoder, DtDesignatorRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_DESIGNATOR_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_DESIGNATOR_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif //DtDesignatorDecoder_H_
```

designatorEnc.h

```

#if DtHLA

#ifndef DtDesignatorEncoder_H_
#define DtDesignatorEncoder_H_

#include "chooseDefines.h"
#include "baseEmbedEnc.h"

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_DESIGNATOR_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtDesignatorRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_DESIGNATOR_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtDesignatorRepository, attrName)

// class DtDesignatorEncoder:
//
// Instances of DtDesignatorEncoder are

class DtDesignatorEncoder : public DtBaseEmbeddedSystemEncoder
{
public:

    // default constructor
    DtDesignatorEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtDesignatorEncoder();

    // copy constructor
    DtDesignatorEncoder(const DtDesignatorEncoder& orig);

    // assignment operator
    DtDesignatorEncoder& operator=(const DtDesignatorEncoder& orig);

public:

    // Create a DtBaseEntityEncoder. Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    DtDECLARE_DESIGNATOR_ATTR_CHECKER(CodeName);
    DtDECLARE_DESIGNATOR_ATTR_CHECKER(DesignatedObject);

```

```
DtDECLARE_DESIGNATOR_ATTR_CHECKER(DesignatorCode);
DtDECLARE_DESIGNATOR_ATTR_CHECKER(DesignatorID);
DtDECLARE_DESIGNATOR_ATTR_CHECKER(DesignatorPower);
DtDECLARE_DESIGNATOR_ATTR_CHECKER(DesignatorSpotLocation);
DtDECLARE_DESIGNATOR_ATTR_CHECKER(DesignatorWavelength);
DtDECLARE_DESIGNATOR_ATTR_CHECKER(DRAlgorithm);
DtDECLARE_DESIGNATOR_ATTR_CHECKER(RelativeSpotLocation);
DtDECLARE_DESIGNATOR_ATTR_CHECKER(SpotLinearAcceleration);

DtDECLARE_DESIGNATOR_ATTR_ENCODER(CodeName);
DtDECLARE_DESIGNATOR_ATTR_ENCODER(DesignatedObject);
DtDECLARE_DESIGNATOR_ATTR_ENCODER(DesignatorCode);
DtDECLARE_DESIGNATOR_ATTR_ENCODER(DesignatorID);
DtDECLARE_DESIGNATOR_ATTR_ENCODER(DesignatorPower);
DtDECLARE_DESIGNATOR_ATTR_ENCODER(DesignatorSpotLocation);
DtDECLARE_DESIGNATOR_ATTR_ENCODER(DesignatorWavelength);
DtDECLARE_DESIGNATOR_ATTR_ENCODER(DRAlgorithm);
DtDECLARE_DESIGNATOR_ATTR_ENCODER(RelativeSpotLocation);
DtDECLARE_DESIGNATOR_ATTR_ENCODER(SpotLinearAcceleration);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_DESIGNATOR_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtDesignatorEncoder, \
        DtDesignatorRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that check whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_DESIGNATOR_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtDesignatorEncoder, \
        DtDesignatorRepository, \
        attrName, inspector)

#endif //DtDesignatorEncoder_H_
```


designatorSR.h

```

#include "embedSystemSR.h"

// class DtDesignatorRepository:
//
// Instances of DtDesignatorRepository are

class DtDesignatorRepository : public DtEmbeddedSystemRepository
{
public:

    // default constructor
    DtDesignatorRepository();

    // destructor
    virtual ~DtDesignatorRepository();

    // copy constructor
    DtDesignatorRepository(const DtDesignatorRepository& orig);

    // assignment operator
    DtDesignatorRepository& operator=(const DtDesignatorRepository& orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const;

    static DtDesignatorRepository* create();

    virtual void printData() const;

    // Set/Get the code name.
    virtual void setCodeName(const DtDesignatorCodeName& name);
    virtual const DtDesignatorCodeName& codeName() const;

    // Set/Get the designated object name.
    virtual void setDesignatedObject(const DtGlobalObjectDesignator& name);
    virtual const DtGlobalObjectDesignator& designatedObject() const;

    // Set/Get the designator code.
    virtual void setDesignatorCode(const DtDesignatorCode& code);
    virtual const DtDesignatorCode& designatorCode() const;

    // Set/Get the designator object id.
    virtual void setDesignatorId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& designatorId() const;

    // Set/Get the designator power.
    virtual void setDesignatorPower(DtFloat32 power);
    virtual DtFloat32 designatorPower() const;

    // Set/Get the designator spot location.
    virtual void setDesignatorSpotLocation(DtConstVector location);

```

```
virtual DtConstVector designatorSpotLocation() const;

// Set/Get the designator wavelength.
virtual void setDesignatorWavelength(DtFloat32 wavelength);
virtual DtFloat32 designatorWavelength() const;

// Set/Get the dead reckoning algorithm.
virtual void setDRAlgorithm(DtDeadReckonTypes algorithm);
virtual DtDeadReckonTypes dRAlgorithm() const;

// Set/Get the relative spot location.
virtual void setRelativeSpotLocation(DtConstVector location);
virtual DtConstVector relativeSpotLocation() const;

// Set/Get the spot linear acceleration.
virtual void setSpotLinearAcceleration(DtConstVector acceleration);
virtual DtConstVector spotLinearAcceleration() const;

protected:
    DtDesignatorCodeName    myCodeName;
    DtGlobalObjectDesignator myDesignatedObject;
    DtGlobalObjectDesignator myDesignatorId;
    DtDesignatorCode        myDesignatorCode;
    DtFloat32               myDesignatorPower;
    DtVector                myDesignatorSpotLocation;
    DtFloat32               myDesignatorWavelength;
    DtDeadReckonTypes       myDRAlgorithm;
    DtVector                myRelativeSpotLocation;
    DtVector                mySpotLinearAcceleration;
};
```

designCbInfo.h

```

#include "vllist.h"

#if DtHLA || DtHLA_GATEWAY

#include "hlaDefines.h"

// Entity state callback function signature.
typedef void (*DtDesignatorCallbackFunction)(
    DtReflectedDesignator* ent, void* userData);

#endif

// class DtDesignatorCallbackInfo:
//
// Instances of DtDesignatorCallbackInfo are

class DtDesignatorCallbackInfo : public DtListedObj
{
public:

    // default constructor
    DtDesignatorCallbackInfo();

    // constructor
    DtDesignatorCallbackInfo(
        DtDesignatorCallbackFunction cb,
        void* userData);

    // destructor
    virtual ~DtDesignatorCallbackInfo();

    // copy constructor
    DtDesignatorCallbackInfo(const DtDesignatorCallbackInfo& orig);

    // assignment operator
    DtDesignatorCallbackInfo& operator=
        (const DtDesignatorCallbackInfo& orig);

    // equality operators
    int operator==(const DtDesignatorCallbackInfo& orig) const;
    int operator!=(const DtDesignatorCallbackInfo& orig) const;

    // set/get the callback function
    virtual void setCallbackFunction(DtDesignatorCallbackFunction funct);
    virtual DtDesignatorCallbackFunction callbackFunction() const;

    // set/get the user data
    virtual void setUserData(void* data);
    virtual void* userData() const;

protected:

```

```
DtDesignatorCallbackFunction myCallbackFunction;  
void* myUserData;  
};
```

detonatePdu.h

```

#if !DtHLA

#include "interaction.h"
#include "packets/detonationP.h"
#include "hostStructs.h"
#include "mathTypes.h"
#include "apSpec.h"
#include "fireInter.h"

#include "chooseDefines.h"

#define DtDetonateStaticSize 104

class DtDetonationPdu;
typedef DtDetonationPdu DtDetonationInteraction;
typedef void (*DtDetonationInteractionCb)(DtDetonationInteraction *inter,
    void *usr);

class DtDetonationPdu : public DtInteraction
{
public:

    /***** STANDARD CONSTRUCTORS *****/

    DtDetonationPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtDetonationPdu(const DtNetDetonationPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Fills in common fields with data from Fire Interaction
    DtDetonationPdu(const DtFireInteraction &fire,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtDetonationInteractionCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtDetonationInteractionCb cb, void *usr);

    /***** ADDITIONAL CONSTRUCTORS *****/
    DtDetonationPdu(const DtEntityIdentifier &attacker,
        const DtEntityIdentifier &target,
        const DtEntityIdentifier &munition,
        const DtEventID &eventId,
        DtConstVector linVelocity,
        DtConstVector location,
        const DtBurstDescriptor &burst,
        DtConstVector entLocation,
        DtDetonationResult result,
        const DtArtPartSpec *specList,

```

```
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

const DtObjectId &attackerId() const;
const DtObjectId &targetId() const;
const DtObjectId &munitonId() const;
const DtEventID &eventId() const;
DtConstVector velocity() const;
DtConstVector worldLocation() const;
DtConstVector entityLocation() const;
DtDetonationResult result() const;
DtDetonatorFuze fuseType() const;
const DtEntityType &munitonType() const;
int quantity() const;
int rate() const;
DtWarheadType warheadType() const;

// NOT USED IN HLA
const DtBurstDescriptor &burst() const;

int numArtParams() const;
const DtArtPartSpec *artPartSpecs() const;
int getArtParamVal(int paramIndex, DtFloat32 *value) const;
int getArtParamVal(int partType, int param, DtFloat32 *value) const;

/***** MUTATOR FUNCTIONS *****/

void setAttackerId(const DtObjectId &id);
void setTargetId(const DtObjectId &id);
void setMunitonId(const DtObjectId &id);
void setEventId(const DtEventID &id);
void setVelocity(DtConstVector velocity);
void setWorldLocation(DtConstVector location);
void setEntityLocation(DtConstVector location);
void setResult(DtDetonationResult result);
void setFuseType(DtDetonatorFuze fuse);
void setMunitonType(const DtEntityType &munType);
void setQuantity(int quant);
void setRate(int rate);
void setWarheadType(DtWarheadType type);

// NOT USED IN HLA
void setBurst(const DtBurstDescriptor &burst);

void initArtParts(const DtArtPartSpec *specList);
int setArtParamVal(int partType, int param, DtFloat32 value);
int setArtParamVal(int paramIndex, DtFloat32 value);

void printData() const;

protected:
DtPduKind internalGetPduKind() const {return DtDetonationPduKind;}
void setNumArtParams(int numParams);
```

```
DtNetArtPartRecord *artPartRecord(int partType, int param) const;
void setArtParamVal(DtNetArtPartRecord *part, DtFloat32 value);
float getArtParamVal(DtNetArtPartRecord *part) const;

DtObjectId att;
DtObjectId tar;
DtObjectId mun;
DtEventID event;
DtVector vel, world, ent;
DtBurstDescriptor theBurst;
DtEntityType myMunType;
};
#endif
```

dExerciseConn.h

```
#if !DtHLA

#include "Socket.h"
#include "pduCbMgr.h"
#include "disEnums.h"
#include "packets/pduHeader.h"
#include "hostStructs.h"
#include "mathTypes.h"
#include "hashlist.h"
#include "baseExConn.h"
#include "pduFactory.h"
#include "tStampType.h"

#include "chooseDefines.h"

#define DtMaxExerciseId 255
const DtExerciseId DtAnyExercise = 0;

typedef int (*DtPredicate)(DtPdu *pdu, void *arg);

#define DtNET_READ_SUCCESS 0
#define DtNET_READ_NO_PACKETS 1
#define DtNET_READ_BAD_PACKET 2
#define DtNET_READ_SERVER_MSG 3
#define DtNET_READ_SIZE_MISMATCH4
#define DtPROCESS_PDU_SUCCESS DtNET_READ_SUCCESS
#define DtPROCESS_PDU_WRONG_EXID 5
#define DtPROCESS_PDU_NO_PDU 6
#define DtNET_READ_FILTERED_PACKET7

class DtExerciseConn : public DtBaseExerciseConn
{
public:
    /* ----- CONSTRUCTORS ----- */
    DtExerciseConn(int port, int exercise_id,
int site_id, int application_num);
    DtExerciseConn(int port, DtInetAddr destAddr, int exercise_id,
int site_id, int application_num);
    DtExerciseConn(DtSocket *skt, int exercise_id,
int site_id, int application_num);
    virtual ~DtExerciseConn();

    /* ----- READING PDUs ----- */

    void addPduCallback(DtPduKind pduKind, DtPduCallbackFcn fcn,
void *usr);
    void removePduCallback(DtPduKind pduKind, DtPduCallbackFcn fcn,
void *usr);

    int readAndProcess(void);
```



```

virtual int processPdu(DtPdu *pdu);
/* Note: netRead() creates a new DtPdu using DtPacketToPdu().
   Caller, e.g. readAndProcess(), is responsible for deleting
   it when through */
virtual DtPdu *netRead(int *retCode = NULL);

int drainInput(DtTime timeout=-1, int nReadPerTimeCheck=100);
void addInterestInPduKind(DtPduKind kind);
void subtractInterestInPduKind(DtPduKind kind);

DtPduKind readUntil(DtPredicate predicate, void *arg,
    DtTime timeout, DtTime sleepTime=0.1);

void addInterestInMcastAddr(DtInetAddr addr);
void subtractInterestInMcastAddr(DtInetAddr addr);

/* ----- SENDING PDUS ----- */
virtual void send(const DtStateMsg &msg);
virtual void send(const DtInteraction &msg);

virtual void sendStamped(const DtStateMsg &msg);
virtual void sendStamped(const DtInteraction &msg);

virtual int send(const DtPdu &pdu,
    DtInetAddr addr = DtUSE_DEFAULT_IP_ADDR);
    int sendStamped(const DtPdu &pdu,
    DtInetAddr addr = DtUSE_DEFAULT_IP_ADDR);

virtual DtInetAddr destinationAddr(const DtPdu &pdu);

/* ----- ID MANAGEMENT ----- */
DtExerciseId exerciseId();
void setExerciseId(int exercise_id);
DtSimulationAddress applicationId();
const DtEntityIdentifier &nextId();
const DtEventID &nextEventId();
DtEntityIdentifier makeEntityId(int entityNum);

/* ----- MISC ----- */
DtSocket *socket() { return mysock; }
DtPduFactory *pduFactory();
void disableFiltering();
void enableFiltering();
static int nSizeMismatchComplain;

// Set/Get time stamp type (DtTimeStampAbsolute or DtTimeStampRelative)
virtual void setTimeStampType(DtTimeStampType relOrAbs);
virtual DtTimeStampType timeStampType() const;

// Set time stamp type
// This function has been obsoleted by the version above
// that use the DtTimeStampType enumeration, but is included
// for backwards compatibility.
virtual void setTimeStampType(int relOrAbs);

```

```
public:

    // Returns a string indicating the VR-Link version number
    static const char* vrlinkVersion();

    /* ----- PROTECTED ----- */
protected:

    // Return time to send with outgoing messages
    // (default is real time if absolute timestamping, sim time
    // otherwise.
    virtual DtTime currentTimeForStamping();

    void init0(int port, int exerId, int siteId, int applNum, int v);
    void init0(DtSocket *s, int exerId, int siteId, int applNum, int v);
    void init(DtSocket *s, int deleteSkt,
              int exerId, int siteId, int applNum, int v);
    DtExerciseId      exid;
    DtSimulationAddress appid;
    int               availEventNum;
    int               deleteSocket;
    DtSocket          *mysock;
    DtPduCallbackManager myPduCbMgr;
    int pduInterest[DtMAX_PDU_KINDS + 1];
    void zeroPduInterestList();
    int nSizeMismatch;
    DtBoolean filtering;
    DtHashlist mcastAddrs;
    DtPduFactory *myPduFact;
    DtEntityIdentifier myCurrentId;
    DtEventID myCurrentEventId;

    DtTimeStampType myRelOrAbs;
};
/* ----- INLINES ----- */

    int result;
    DtPdu *pdu = netRead(&result);
    if (result == DtNET_READ_SUCCESS)
result = processPdu(pdu);
    if (pdu)
delete pdu;
    return result;
}

    return exid;
}
    return appid;
}

#endif
```

dInteraction.h

```
#if !DtHLA

#include "pdu.h"

#include "chooseDefines.h"

// class DtInteraction:
//
// Instances of DtInteraction are used for representing DIS
// events.

class DtInteraction : public DtPdu
{
public:

    // default constructor
    DtInteraction();

    // destructor
    virtual ~DtInteraction();

    // copy constructor
    DtInteraction(const DtInteraction& orig);

    // assignment operator
    DtInteraction& operator=(const DtInteraction& orig);

};
#endif
```

directRotRate.h

```
// rotation rate given by single rate and axis of rotation
class DtDirectedRotRate
{
public:
    DtDirectedRotRate();
    DtDirectedRotRate(DtConstVector pqr);

    void setFromVec(DtConstVector pqr);

    double thetaDot; /* radians per sec */
    double n[3];      /* axis unit vector */
};
// Inline functions
```

dirRtiAmb.h

```

#if DtHLA

#if DtRTI103
#include "dirAmb103.h"
#else

#ifndef DtDirectRtiAmbassador_H_
#define DtDirectRtiAmbassador_H_

#include "RTI.hh"
#include "vrlRtiAmb.h"

// class DtDirectRtiAmbassador:
//
// Instances of DtDirectRtiAmbassador are for wrapping the calls to
// The RTI ambassador so that we may override them.

class DtDirectRtiAmbassador : public DtVrlRtiAmbassador
{
public:

    // default constructor
    DtDirectRtiAmbassador();

    // destructor
    virtual ~DtDirectRtiAmbassador();

    // copy constructor
    DtDirectRtiAmbassador(const DtDirectRtiAmbassador& orig);

    // assignment operator
    DtDirectRtiAmbassador& operator=(const DtDirectRtiAmbassador& orig);

    // Get the real RTI ambassador.
    virtual RTI::RTIAmbassador* rtiAmb();

    // Calls down to the RTI function of the same name.
    virtual void createFederationExecution (
        const char *executionName,
        const char *FED);

    // Calls down to the RTI function of the same name.
    virtual void destroyFederationExecution (
        const char *executionName);

    // Calls down to the RTI function of the same name.
    virtual RTI::FederateHandle joinFederationExecution (
        const char* yourName,
        const char* executionName,
        RTI::FederateAmbassadorPtr federateAmbassadorReference);

```

```
// Calls down to the RTI function of the same name.
virtual void resignFederationExecution(
    RTI::ResignAction theAction);

// Calls down to the RTI function of the same name.
virtual void registerFederationSynchronizationPoint(
    const char* label,
    const char* theTag);

// Calls down to the RTI function of the same name.
virtual void registerFederationSynchronizationPoint(
    const char* label,
    const char* theTag,
    const RTI::FederateHandleSet& syncSet);

// Calls down to the RTI function of the same name.
virtual void synchronizationPointAchieved(
    const char* label);

// Calls down to the RTI function of the same name.
virtual void requestFederationSave (
    const char* label,
    const RTI::FedTime& theTime);

// Calls down to the RTI function of the same name.
virtual void requestFederationSave (
    const char *label);

// Calls down to the RTI function of the same name.
virtual void federateSaveBegun();

// Calls down to the RTI function of the same name.
virtual void federateSaveComplete();

// Calls down to the RTI function of the same name.
virtual void federateSaveNotComplete();

// Calls down to the RTI function of the same name.
virtual void requestFederationRestore(const char* label);

// Calls down to the RTI function of the same name.
virtual void federateRestoreComplete();

// Calls down to the RTI function of the same name.
virtual void federateRestoreNotComplete();

// Calls down to the RTI function of the same name.
virtual void publishObjectClass(
    RTI::ObjectClassHandle theClass,
    const RTI::AttributeHandleSet& attributeList);

// Calls down to the RTI function of the same name.
virtual void unpublishObjectClass(
    RTI::ObjectClassHandle theClass);
```

```
// Calls down to the RTI function of the same name.
virtual void publishInteractionClass(
    RTI::InteractionClassHandle theInteraction);

// Calls down to the RTI function of the same name.
virtual void unpublishInteractionClass(
    RTI::InteractionClassHandle theInteraction);

// Calls down to the RTI function of the same name.
virtual void subscribeObjectClassAttributes(
    RTI::ObjectClassHandle theClass,
    const RTI::AttributeHandleSet& attributeList,
    RTI::Boolean active = RTI::RTI_TRUE);

// Calls down to the RTI function of the same name.
virtual void unsubscribeObjectClass(
    RTI::ObjectClassHandle theClass);

// Calls down to the RTI function of the same name.
virtual void subscribeInteractionClass(
    RTI::InteractionClassHandle theClass,
    RTI::Boolean active = RTI::RTI_TRUE);

// Calls down to the RTI function of the same name.
virtual void unsubscribeInteractionClass(
    RTI::InteractionClassHandle theClass);

// Calls down to the RTI function of the same name.
virtual RTI::ObjectHandle registerObjectInstance(
    RTI::ObjectClassHandle theClass, const char* theObject);

// Calls down to the RTI function of the same name.
virtual RTI::ObjectHandle registerObjectInstance(
    RTI::ObjectClassHandle theClass);

// Calls down to the RTI function of the same name.
virtual RTI::EventRetractionHandle updateAttributeValues(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleValuePairSet& theAttributes,
    const RTI::FedTime& theTime,
    const char* theTag);

// Calls down to the RTI function of the same name.
virtual void updateAttributeValues(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleValuePairSet& theAttributes,
    const char* theTag);

// Calls down to the RTI function of the same name.
virtual RTI::EventRetractionHandle sendInteraction(
    RTI::InteractionClassHandle theInteraction,
    const RTI::ParameterHandleValuePairSet& theParameters,
    const RTI::FedTime& theTime,
    const char* theTag);
```

```
// Calls down to the RTI function of the same name.
virtual void sendInteraction(
    RTI::InteractionClassHandle theInteraction,
    const RTI::ParameterHandleValuePairSet& theParameters,
    const char* theTag);

// Calls down to the RTI function of the same name.
virtual RTI::EventRetractionHandle deleteObjectInstance(
    RTI::ObjectHandle theObject,
    const RTI::FedTime& theTime,
    const char* theTag);

// Calls down to the RTI function of the same name.
virtual void deleteObjectInstance(
    RTI::ObjectHandle theObject,
    const char* theTag);

// Calls down to the RTI function of the same name.
virtual void localDeleteObjectInstance(
    RTI::ObjectHandle theObject);

// Calls down to the RTI function of the same name.
virtual void changeAttributeTransportationType (
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes,
    RTI::TransportationHandle theType);

// Calls down to the RTI function of the same name.
virtual void changeInteractionTransportationType (
    RTI::InteractionClassHandle theClass,
    RTI::TransportationHandle theType);

// Calls down to the RTI function of the same name.
virtual void requestObjectAttributeValueUpdate (
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes);

// Calls down to the RTI function of the same name.
virtual void requestClassAttributeValueUpdate (
    RTI::ObjectClassHandle theClass,
    const RTI::AttributeHandleSet& theAttributes);

// Calls down to the RTI function of the same name.
virtual void unconditionalAttributeOwnershipDivestiture(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes);

// Calls down to the RTI function of the same name.
virtual void negotiatedAttributeOwnershipDivestiture(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes,
    const char* theTag);

// Calls down to the RTI function of the same name.
virtual void attributeOwnershipAcquisition(
```



```
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& desiredAttributes,
    const char* theTag);

// Calls down to the RTI function of the same name.
virtual void attributeOwnershipAcquisitionIfAvailable(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& desiredAttributes);

// Calls down to the RTI function of the same name.
virtual RTI::AttributeHandleSet* attributeOwnershipReleaseResponse(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes);

// Calls down to the RTI function of the same name.
virtual void cancelNegotiatedAttributeOwnershipDivestiture(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes);

// Calls down to the RTI function of the same name.
virtual void cancelAttributeOwnershipAcquisition(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes);

// Calls down to the RTI function of the same name.
virtual void queryAttributeOwnership(
    RTI::ObjectHandle theObject,
    RTI::AttributeHandle theAttribute);

// Calls down to the RTI function of the same name.
virtual RTI::Boolean isAttributeOwnedByFederate(
    RTI::ObjectHandle theObject,
    RTI::AttributeHandle theAttribute);

// Calls down to the RTI function of the same name.
virtual void enableTimeRegulation (
    const RTI::FedTime& theFederateTime,
    const RTI::FedTime& theLookahead);

// Calls down to the RTI function of the same name.
virtual void disableTimeRegulation();

// Calls down to the RTI function of the same name.
virtual void enableTimeConstrained();

// Calls down to the RTI function of the same name.
virtual void disableTimeConstrained();

// Calls down to the RTI function of the same name.
virtual void timeAdvanceRequest(
    const RTI::FedTime& theTime);

// Calls down to the RTI function of the same name.
virtual void timeAdvanceRequestAvailable (
    const RTI::FedTime& theTime);
```

```
// Calls down to the RTI function of the same name.
virtual void nextEventRequest(
    const RTI::FedTime& theTime);

// Calls down to the RTI function of the same name.
virtual void nextEventRequestAvailable (
    const RTI::FedTime& theTime);

// Calls down to the RTI function of the same name.
virtual void flushQueueRequest (
    const RTI::FedTime& theTime);

// Calls down to the RTI function of the same name.
virtual void enableAsynchronousDelivery();

// Calls down to the RTI function of the same name.
virtual void disableAsynchronousDelivery();

// Calls down to the RTI function of the same name.
virtual void queryLBTS(RTI::FedTime& theTime);

// Calls down to the RTI function of the same name.
virtual void queryFederateTime(RTI::FedTime& theTime);

// Calls down to the RTI function of the same name.
virtual void queryMinNextEventTime (
    RTI::FedTime& theTime);

// Calls down to the RTI function of the same name.
virtual void modifyLookahead (
    const RTI::FedTime& theLookahead);

// Calls down to the RTI function of the same name.
virtual void queryLookahead (
    RTI::FedTime& theTime);

// Calls down to the RTI function of the same name.
virtual void retract(
    RTI::EventRetractionHandle theHandle);

// Calls down to the RTI function of the same name.
virtual void changeAttributeOrderType(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes,
    RTI::OrderingHandle theType);

// Calls down to the RTI function of the same name.
virtual void changeInteractionOrderType(
    RTI::InteractionClassHandle theClass,
    RTI::OrderingHandle theType);

// Calls down to the RTI function of the same name.
virtual void createRegion (
    RTI::SpaceHandle theSpace,
```

```

        RTI::ULong numberOfExtents);

// Calls down to the RTI function of the same name.
virtual void notifyAboutRegionModification (
    RTI::Region &theRegion);

// Calls down to the RTI function of the same name.
virtual void deleteRegion(RTI::Region *theRegion);

// Calls down to the RTI function of the same name.
virtual RTI::ObjectHandle registerObjectInstanceWithRegion(
    RTI::ObjectClassHandle theClass,
    const char* theObject,
    RTI::AttributeHandle theAttributes[],
    RTI::Region* theRegions[],
    RTI::ULong theNumberOfHandles);

// Calls down to the RTI function of the same name.
virtual RTI::ObjectHandle registerObjectInstanceWithRegion(
    RTI::ObjectClassHandle theClass,
    RTI::AttributeHandle theAttributes[],
    RTI::Region* theRegions[],
    RTI::ULong theNumberOfHandles);

// Calls down to the RTI function of the same name.
virtual void associateRegionForUpdates (
    RTI::Region& theRegion,
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes);

// Calls down to the RTI function of the same name.
virtual void unassociateRegionForUpdates(
    RTI::Region& theRegion,
    RTI::ObjectHandle theObject);

// Calls down to the RTI function of the same name.
virtual void subscribeObjectClassAttributesWithRegion(
    RTI::ObjectClassHandle theClass,
    RTI::Region& theRegion,
    const RTI::AttributeHandleSet &attributeList,
    RTI::Boolean active = RTI::RTI_TRUE);

// Calls down to the RTI function of the same name.
virtual void unsubscribeObjectClassWithRegion(
    RTI::ObjectClassHandle theClass,
    RTI::Region& theRegion);

// Calls down to the RTI function of the same name.
virtual void subscribeInteractionClassWithRegion (
    RTI::InteractionClassHandle theClass,
    RTI::Region& theRegion,
    RTI::Boolean active = RTI::RTI_TRUE);

// Calls down to the RTI function of the same name.
virtual void unsubscribeInteractionClassWithRegion (

```

```
RTI::InteractionClassHandle theClass,  
RTI::Region& theRegion);  
  
// Calls down to the RTI function of the same name.  
virtual RTI::EventRetractionHandle sendInteractionWithRegion(  
    RTI::InteractionClassHandle theInteraction,  
    const RTI::ParameterHandleValuePairSet &theParameters,  
    const RTI::FedTime& theTime,  
    const char* theTag,  
    const RTI::Region& theRegion);  
  
// Calls down to the RTI function of the same name.  
virtual void sendInteractionWithRegion(  
    RTI::InteractionClassHandle theInteraction,  
    const RTI::ParameterHandleValuePairSet &theParameters,  
    const char* theTag,  
    const RTI::Region& theRegion);  
  
// Calls down to the RTI function of the same name.  
virtual void requestClassAttributeValueUpdateWithRegion(  
    RTI::ObjectClassHandle theClass,  
    const RTI::AttributeHandleSet& theAttributes,  
    const RTI::Region& theRegion);  
  
// Calls down to the RTI function of the same name.  
virtual RTI::ObjectClassHandle getObjectClassHandle (  
    const char* theName);  
  
// Calls down to the RTI function of the same name.  
virtual char* getObjectClassName (  
    RTI::ObjectClassHandle theHandle);  
  
// Calls down to the RTI function of the same name.  
virtual RTI::AttributeHandle getAttributeHandle (  
    const char* theName,  
    RTI::ObjectClassHandle whichClass);  
  
// Calls down to the RTI function of the same name.  
virtual char* getAttributeName (  
    RTI::AttributeHandle theHandle,  
    RTI::ObjectClassHandle whichClass);  
  
// Calls down to the RTI function of the same name.  
virtual RTI::InteractionClassHandle getInteractionClassHandle (  
    const char* theName);  
  
// Calls down to the RTI function of the same name.  
virtual char* getInteractionClassName (  
    RTI::InteractionClassHandle theHandle);  
  
// Calls down to the RTI function of the same name.  
virtual RTI::ParameterHandle getParameterHandle (  
    const char* theName,  
    RTI::InteractionClassHandle whichClass);
```

```
// Calls down to the RTI function of the same name.
virtual char* getParameterName (
    RTI::ParameterHandle theHandle,
    RTI::InteractionClassHandle whichClass);

// Calls down to the RTI function of the same name.
virtual RTI::ObjectHandle getObjectInstanceHandle(
    const char* theName);

// Calls down to the RTI function of the same name.
virtual char* getObjectInstanceName(
    RTI::ObjectHandle theHandle);

// Calls down to the RTI function of the same name.
virtual RTI::SpaceHandle getRoutingSpaceHandle (
    const char* theName);

// Calls down to the RTI function of the same name.
virtual const char* getRoutingSpaceName (
    const RTI::SpaceHandle theHandle);

// Calls down to the RTI function of the same name.
virtual RTI::DimensionHandle getDimensionHandle(
    const char* theName,
    RTI::SpaceHandle whichSpace);

// Calls down to the RTI function of the same name.
virtual char* getDimensionName(
    RTI::DimensionHandle theHandle,
    RTI::SpaceHandle whichSpace);

// Calls down to the RTI function of the same name.
virtual RTI::SpaceHandle
    getAttributeRoutingSpaceHandle(
        RTI::AttributeHandle theHandle,
        RTI::ObjectClassHandle whichClass);

// Calls down to the RTI function of the same name.
virtual RTI::ObjectClassHandle getObjectClass (
    RTI::ObjectHandle theObject);

// Calls down to the RTI function of the same name.
virtual RTI::SpaceHandle getInteractionRoutingSpaceHandle(
    RTI::InteractionClassHandle theHandle);

// Calls down to the RTI function of the same name.
virtual RTI::TransportationHandle getTransportationHandle(
    const char *theName);

// Calls down to the RTI function of the same name.
virtual char* getTransportationName(
    RTI::TransportationHandle theHandle);

// Calls down to the RTI function of the same name.
virtual RTI::OrderingHandle getOrderingHandle(
```

```
        const char *theName);

    // Calls down to the RTI function of the same name.
    virtual char* getOrderingName(
        RTI::OrderingHandle theHandle);

    // Calls down to the RTI function of the same name.
    virtual void enableClassRelevanceAdvisorySwitch();

    // Calls down to the RTI function of the same name.
    virtual void disableClassRelevanceAdvisorySwitch();

    // Calls down to the RTI function of the same name.
    virtual void enableAttributeRelevanceAdvisorySwitch();

    // Calls down to the RTI function of the same name.
    virtual void disableAttributeRelevanceAdvisorySwitch();

    // Calls down to the RTI function of the same name.
    virtual void enableAttributeScopeAdvisorySwitch();

    // Calls down to the RTI function of the same name.
    virtual void disableAttributeScopeAdvisorySwitch();

    // Calls down to the RTI function of the same name.
    virtual void enableInteractionRelevanceAdvisorySwitch();

    // Calls down to the RTI function of the same name.
    virtual void disableInteractionRelevanceAdvisorySwitch();

    // Calls down to the RTI function of the same name.
    virtual RTI::Boolean tick();

    // Calls down to the RTI function of the same name.
    virtual RTI::Boolean tick(
        RTI::TickTime minimum,
        RTI::TickTime maximum);

    // Calls down to the RTI function of the same name.
    virtual RTI::RegionToken getRegionToken(RTI::Region*);

    // Calls down to the RTI function of the same name.
    virtual RTI::Region* getRegion(RTI::RegionToken);

public:

    // returns a new DtDirectRtiAmbassador
    static DtVrIRtiAmbassador* create();

protected:

    RTI::RTIambassador* myRtiAmb;
};
#endif
#endif
```

disEnums.h

```

#include "packets/disVersion.h"

#define DtEXTEND_ENUM(enumType, name, val) \
    const enumType name = enumType(val)

/* SECTION 1 */

enum DtDetonatorFuze
{
    DtFuzeOther= 0,
    DtFuzeContact = 1000,
    DtFuzeContactInstant = 1100,
    DtFuzeContactDelayed = 1200,
    DtFuzeTimed = 2000,
    DtFuzeProximity = 3000,
    DtFuzeCommand = 4000,
    DtFuzeAltitude = 5000,
    DtFuzeDepth = 6000,
    DtFuzeAcoustic = 7000
};

enum DtWarheadType
{
    DtWarheadOther= 0,
    DtWarheadHighExplosive= 1000,
    DtWarheadHighExplosivePlastic= 1100,
    DtWarheadHighExplosiveIncendiary= 1200,
    DtWarheadHighExplosiveFragmentation= 1300,
    DtWarheadHighExplosiveAntiTank= 1400,
    DtWarheadHighExplosiveBomblets= 1500,
    DtWarheadHighExplosiveShapedCharge= 1600,
    DtWarheadSmoke= 2000,
    DtWarheadIllumination= 3000,
    DtWarheadPractice= 4000,
    DtWarheadKinetic= 5000,
    DtWarheadUnused= 6000,
    DtWarheadNuclear= 7000,
    DtWarheadChemicalGeneral= 8000,
    DtWarheadChemicalBlisterAgent= 8100,
    DtWarheadChemicalBloodAgent= 8200,
    DtWarheadChemicalNerveAgent= 8300,
    DtWarheadBiologicalGeneral= 9000
};

/* Section 2 */

enum DtRepairType
{
    DtNoRepairsPerformed,
    DtRepairEverything,
    #if DtDISVERS != 3

```

```
DtDriveTrainMotor = 10,
DtDriveTrainStarter = 20,
DtDriveTrainAlternator = 30,
DtDriveTrainGenerator = 40,
DtDriveTrainBattery = 50,
DtDriveTrainEngineCoolantLeak = 60,
DtDriveTrainFuelFilter = 70,
DtDriveTrainTranmissionOilLeak = 80,
DtDriveTrainEngineOilLeak = 90,
DtDriveTrainPumps = 100,
DtDriveTrainFilters = 110,
#else
DtPropulsionMotor = 10,
DtPropulsionStarter = 20,
DtPropulsionAlternator = 30,
DtPropulsionGenerator = 40,
DtPropulsionBattery = 50,
DtPropulsionEngineCoolantLeak = 60,
DtPropulsionFuelFilter = 70,
DtPropulsionTranmissionOilLeak = 80,
DtPropulsionEngineOilLeak = 90,
DtPropulsionPumps = 100,
DtPropulsionFilters = 110,
#endif
DtBodyHull = 1000,
DtBodyAirframe = 1010,
DtInterfacePropeller = 1500,
#if DtDISVERS != 3
DtDriveTrainTransmission = 120,
DtDriveTrainBrakes = 130,
#else
DtInterfaceTransmission = 1510,
DtInterfaceBrakes = 1530,
#endif
DtInterfaceFilters = 1520,
DtWeaponGunElevationDrive = 2000,
DtWeaponGunStabilizationSys = 2010,
DtWeaponGunneryPrimarySight = 2020,
DtWeaponCommandersGPSExtension = 2030,
DtWeaponLoadingMechanism = 2040,
DtFuelTransferPump = 4000,
DtFuelLines = 4010,
DtFuelGauges = 4020,
#if DtDISVERS != 3
DtElectronicWarfareSys = 4500,
DtElectronicDetectionSys = 4600,
DtElectronicRangeFinder = 4700,
DtElectronicSys = 4800,
DtElectronicRadios = 5000,
DtElectronicIntercoms = 5100,
DtElectronicLasers = 4840,
DtElectronicCoders = 5200,
DtElectronicDecoders = 5300,
DtElectronicComputers = 5500,
#else
```



```

    DtElectronicWarfareSys = 4600,
    DtElectronicDetectionSys = 5600,
    DtElectronicLaserRangeFinder = 4700,
    DtElectronicCounterMeasureSys = 4500,
    DtElectronicRadios = 4800,
    DtElectronicIntercoms = 4900,
    DtElectronicLasers = 5200,
    DtElectronicCoders = 5000,
    DtElectronicDecoders = 5100,
    DtElectronicEmitters = 5400,
    DtElectronicComputers = 5300,
#endif
    DtLifeSupportAirSupply = 8000,
    DtLifeSupportFilters = 8010,
    DtLifeSupportWaterSupply = 8020,
    DtLifeSupportRefrigerationSys = 8030,
    DtLifeSupportCBRProtection = 8040,
    DtLifeSupportWaterWashSys = 8050,
    DtLifeSupportDecontaminationSys = 8060,
    DtActuatorWaterSupply = 9000,
    DtActuatorCoolingSys = 9010,
    DtActuatorWinches = 9020,
    DtActuatorCatapults = 9030,
    DtActuatorCranes = 9040,
    DtActuatorLaunchers = 9050,
    DtAuxCraftLifeBoat = 10000,
    DtAuxCraftLandingCraft = 10010,
    DtAuxCraftEjectionSeat = 10020
};

#if DtDISVERS >= 5
enum DtCollisionType
{

    // Elastic and Inelastic were incorrect prior to 3.1

    DtInelastic= 0,
    DtElastic= 1,

    // Other collision type is no longer a valid DIS enumeration.

    DtCollisionTypeOther= 2

};
#endif

/* Section 4 */

enum DtForceType
{
    DtForceOther,
    DtForceFriendly,
    DtForceOpposing,
    DtForceNeutral
};

```

```
enum DtDetonationResult
{
    DtDetResOther          = 0,
    DtDetResEntityImpact   = 1,
    DtDetResEntityProximate= 2,
    DtDetResGroundImpact   = 3,
    DtDetResGroundProximate= 4,
    DtDetResDetonation= 5,
#ifdef DtDISVERS == 3
    DtDetResNone           = 6
#else
    DtDetResNone           = 6,
    DtDetResHE_HitSmall    = 7,
    DtDetResHE_HitMedium   = 8,
    DtDetResHE_HitLarge    = 9,
    DtDetResArmorPiercingHit = 10,
    DtDetResDirtBlastSmall  = 11,
    DtDetResDirtBlastMedium = 12,
    DtDetResDirtBlastLarge  = 13,
    DtDetResWaterBlastSmall = 14,
    DtDetResWaterBlastMedium = 15,
    DtDetResWaterBlastLarge = 16,
    DtDetResAirHit          = 17,
    DtDetResBuildingHitSmall = 18,
    DtDetResBuildingHitMedium = 19,
    DtDetResBuildingHitLarge = 20,
    DtDetResMineClearingLineCharge = 21,
    DtDetResEnvironmentObjectImpact = 22,
    DtDetResEnvironmentObjectProximate = 23,
    DtDetResWaterImpact     = 24,
    DtDetResAirBurst        = 25
#endif
};

enum DtPduKind
{
    DtOtherPduKind =0,
    DtEntityStatePduKind =1,
    DtFirePduKind =2,
    DtDetonationPduKind =3,
    DtCollisionPduKind =4,
    DtServiceRequestPduKind =5,
    DtResupplyOfferPduKind =6,
    DtResupplyReceivedPduKind =7,
    DtResupplyCancelPduKind =8,
    DtRepairCompletePduKind =9,
    DtRepairResponsePduKind =10,
    DtCreateEntityPduKind =11,
    DtRemoveEntityPduKind =12,
    DtStartResumePduKind =13,
    DtStopFreezePduKind =14,
    DtAcknowledgePduKind =15,
    DtActionRequestPduKind =16,
    DtActionResponsePduKind =17,
```

```

        DtDataQueryPduKind =      18,
        DtSetDataPduKind =      19,
        DtDataPduKind = 20,
        DtEventReportPduKind =21,
#if DtDISVERS >= 5
        DtCommentPduKind =      22,
#endif
        DtMessagePduKind =      22,
#if DtDISVERS != 3
        DtElectromagneticEmissionPduKind = 23,
        DtDesignatorPduKind = 24,
#else
        DtEmissionPduKind =      23,
        DtLaserPduKind = 24,
#endif
        DtTransmitterPduKind =25,
        DtSignalPduKind = 26,
        DtReceiverPduKind =      27,

// DMO enums used by 2.1.4 PDUs
        DtIffAtcNavAidsPduKind =      28,
        DtUnderwaterAcousticPduKind = 29,
        DtSuppEmisEntStatePduKind = 30,
        DtIntercomSignalPduKind = 31,
        DtIntercomControlPduKind = 32,
        DtAggregateStatePduKind = 33,
        DtIsGroupOfPduKind = 34,
        DtTransferControlRequestPduKind = 35,
        DtIsPartOfPduKind = 36,
        DtMinefieldStatePduKind = 37,
        DtMinefieldQueryPduKind = 38,
        DtMinefieldDataPduKind = 39,
        DtMinefieldResponseNackPduKind = 40,
        DtEnvironmentalProcessPduKind = 41,
        DtGriddedDataPduKind = 42,
        DtPointObjectStatePduKind = 43,
        DtLinearObjectStatePduKind = 44,
        DtArealObjectStatePduKind = 45,
        DtLeTspiPduKind = 46,
        DtLeAppearancePduKind = 47,
        DtLeArticulatedPartsPduKind = 48,
        DtLeFirePduKind = 49,
        DtLeDetonationPduKind = 50,
        DtCreateEntityRPduKind = 51,
        DtRemoveEntityRPduKind = 52,
        DtStartResumeRPduKind = 53,
        DtStopFreezeRPduKind = 54,
        DtAcknowledgeRPduKind = 55,
        DtActionRequestRPduKind = 56,
        DtActionResponseRPduKind = 57,
        DtDataQueryRPduKind = 58,
        DtSetDataRPduKind = 59,
        DtDataRPduKind = 60,
        DtEventReptRPduKind = 61,
        DtCommentRPduKind = 62,

```

```
DtRecordQueryPduKind =          63,
DtSetRecordPduKind =           64,
DtRecordPduKind =              65,
DtCollisionElasticPduKind =     66,
DtEntityStateUpdatePduKind =    67
};

const DtPduKind DtNPCollisionPduKind = DtPduKind(200);
const DtPduKind DtNPFieldPduKind = DtPduKind(201);
const DtPduKind DtViewControlPduKind = DtPduKind(202);
const DtPduKind DtMOPMigratorPduKind = DtPduKind(203);
const DtPduKind DtNPSurfaceContactPduKind = DtPduKind(204);
const DtPduKind DtGenericPduKind = DtPduKind(254);
const DtPduKind DtLgrControlPduKind = DtPduKind(205);

/* backwards compatible */
#define DtViewControlKind DtViewControlPduKind

#if DtDISVERS != 3
enum DtProtocolFamily
{
    DtFamilyOther,
    DtFamilyEntityInteraction,
    DtFamilyWarfare,
    DtFamilyLogistics,
    DtFamilyRadioCommunication,
    DtFamilySimulationManagement,
    DtFamilyDistributedEmissionRegeneration,
    DtFamilyEntityManagement,
    DtFamilyMinefield,
    DtFamilySyntheticEnvironment,
    DtFamilySimulationManagementWithReliability,
    DtFamilyLiveEntity,
    DtFamilyNonRealTime
};
#endif

enum DtRepairResult
{
    DtRepairResultOther,
    DtRepairResultEnded,
    DtRepairResultInvalid,
    DtRepairResultInterrupted,
    DtRepairResultCancelled
};

enum DtServiceType
{
    DtServiceTypeOther,
    DtServiceTypeResupply,
    DtServiceTypeRepair
};

enum DtStopReason
{
```

```
DtStopReasonOther,  
DtStopReasonRecess,  
DtStopReasonTermination,  
DtStopReasonSystemFailure,  
DtStopReasonSecurityViolation,  
DtStopReasonEntityReconstitution  
};  
  
enum DtAcknowledgeFlag  
{  
    DtAckCreate = 1,  
    DtAckRemove,  
    DtAckStart,  
    DtAckStop  
};  
  
enum DtResponseFlag  
{  
    DtNoWay,  
    DtSureThing  
};  
  
enum DtActionId  
{  
    DtActionOther,  
    DtActionLocalStorage,  
    DtActionOutOfAmmo,  
    DtActionKilledInAction,  
    DtActionDamage,  
    DtActionMobilityDisabled,  
    DtActionFireDisabled,  
    DtActionOutOfFuel,  
    DtActionRecallCheckpointData,  
    DtActionRecallInitialParameters,  
    DtActionInitiateTetherLead,  
    DtActionInitiateTetherFollow,  
    DtActionUntether,  
    DtActionInitiateServiceStationResupply,  
    DtActionInitiateTailgateResupply,  
    DtActionInitiateHitchLead,  
    DtActionInitiateHitchFollow,  
    DtActionUnhitch,  
    DtActionMount,  
    DtActionDismount,  
    DtActionStartDRC,  
    DtActionStopDRC,  
    DtActionDataQuery,  
    DtActionStatusRequest,  
    DtActionSendObjectStateData,  
    DtActionReconstitute,  
    DtActionLockSiteConfiguration,  
    DtActionUnlockSiteConfiguration,  
    DtActionUpdateSiteConfiguration,  
    DtActionQuerySiteConfiguration,  
    DtActionTetheringInformation,
```

```
DtActionMountIntent,  
DtActionOther2,  
DtActionAcceptSubscription,  
DtActionUnsubscribe,  
DtActionTeleportEntity,  
DtActionChangeAggregateState,  
DtActionRequestStartPDU,  
DtActionWakeUpGetReadyForInitialization,  
DtActionInitializeInternalParameters,  
DtActionSendPlanData,  
DtActionSynchronizeInternalClocks,  
DtActionRun,  
DtActionSaveInternalParameters,  
DtActionSimulateMalfunction,  
DtActionJoinExercise,  
DtActionResignExercise,  
DtActionTimeAdvance  
};  
  
enum DtRequestStatus  
{  
    DtRequestStatusOther,  
    DtRequestStatusPending,  
    DtRequestStatusExecuting,  
    DtRequestStatusPartial,  
    DtRequestStatusComplete,  
    DtRequestStatusRejected,  
    DtRequestStatusRetransmitNow,  
    DtRequestStatusRetransmitLater,  
    DtRequestStatusInvalidTimeParameters,  
    DtRequestStatusSimulationTimeExceeded,  
    DtRequestStatusRequestDone  
};  
  
enum DtEventType  
{  
    DtEventOther,  
    DtEventOutOfFuel,  
    DtEventOutOfAmmo,  
    DtEventKilledInAction,  
    DtEventDamage  
};  
  
#if DtDISVERS != 3  
enum DtDatumParam  
{  
    DtDatumMass= 26000,  
    DtDatumForceId= 11200,  
  
    DtDatumKind= 11110,  
    DtDatumDomain= 11120,  
    DtDatumCountry= 11130,  
    /* missing from spec */  
    DtDatumCategory= 11140,  
    DtDatumSubCategory= 11150,
```

```
DtDatumSpecific= 11160,
DtDatumExtra= 11170,

DtDatumAltKind= 12110,
DtDatumAltDomain= 12120,
DtDatumAltCountry= 12130,
/* missing from spec */
DtDatumAltCategory= 12140,
DtDatumAltSubCategory = 12150,
DtDatumAltSpecific= 12160,
DtDatumAltExtra= 12170,

DtDatumXpos= 31210,
DtDatumYpos= 31220,
DtDatumZpos= 31230,
#if !PCFTP
DtDatumXvel= 36100,
DtDatumYvel= 36200,
DtDatumZvel= 36300,
#endif
DtDatumPhi= 32500,
DtDatumTheta= 32600,
DtDatumPsi= 32700,

#if !PCFTP
DtDatumDrAlgorithm= 71000,

DtDatumXaccel= 37100,
DtDatumYaccel= 37200,
DtDatumZaccel= 37300,
DtDatumXangvel= 100001,
DtDatumYangvel= 100002,
DtDatumZangvel= 100003,

DtDatumAppearance= 33000,
#endif

/* character set */
DtDatumMarkingCS= 13100,
/* 11 bytes */
DtDatumMarking= 13000,

#if !PCFTP
DtDatumNumArtParam= 90000,
DtDatumArtParamId= 100014,
DtDatumArtParamType= 100047,
DtDatumArtParamValue = 100048,
DtDatumAmountAmmoSupplies= 27000,
DtDatumAmmoQuantity= 24000,
DtDatumFuelQuantity= 23100,
DtDatumRadarSysStatus= 61400,
DtDatumRadioCommSysStatus= 62100,
DtDatumStationaryRadioTime= 62200,
DtDatumMovingRadioTime= 62300,
DtDatumBodyPartDamagedRatio= 35000,
```

```
DtDatumTerrainDatabase= 41000,
DtDatumLoggerFile= 43000,
#endif
DtDatumCapability= 34

};

#else

enum DtDatumParam
{
    DtDatumMass= 1,
    DtDatumForceId= 2,

    DtDatumKind= 3,
    DtDatumDomain= 4,
    DtDatumCountry= 5,
    /* missing from spec */
    DtDatumCategory= 128,
    DtDatumSubCategory= 6,
    DtDatumSpecific= 7,
    DtDatumExtra= 8,

    DtDatumAltKind= 9,
    DtDatumAltDomain= 10,
    DtDatumAltCountry= 11,
    /* missing from spec */
    DtDatumAltCategory= 129,
    DtDatumAltSubCategory = 12,
    DtDatumAltSpecific= 13,
    DtDatumAltExtra= 14,

    DtDatumXpos= 15,
    DtDatumYpos= 16,
    DtDatumZpos= 17,
    DtDatumXvel= 18,
    DtDatumYvel= 19,
    DtDatumZvel= 20,
    DtDatumPhi= 21,
    DtDatumTheta= 22,
    DtDatumPsi= 23,

    DtDatumDrAlgorithm= 24,

    DtDatumXaccel= 25,
    DtDatumYaccel= 26,
    DtDatumZaccel= 27,
    DtDatumXangvel= 28,
    DtDatumYangvel= 29,
    DtDatumZangvel= 30,

    DtDatumAppearance= 31,
    /* character set */
    DtDatumMarkingCS= 32,
    /* 11 bytes */
}
```



```

    DtDatumMarking= 33,
    DtDatumCapability= 34,
    DtDatumNumArtParam= 35,
    DtDatumArtParamId= 36,
    DtDatumArtParamType= 37,
    DtDatumArtParamValue = 38,
    DtDatumAmountAmmoSupplies= 39,
    DtDatumAmmoQuantity= 40,
    DtDatumFuelQuantity= 41,
    DtDatumRadarSysStatus= 42,
    DtDatumRadioCommSysStatus= 43,
    DtDatumStationaryRadioTime= 44,
    DtDatumMovingRadioTime= 45,
    DtDatumBodyPartDamagedRatio= 46,
    DtDatumTerrainDatabase= 47,
    DtDatumLoggerFile= 48
};

#endif

enum DtEmissionPduType
{
    DtStateUpdate,
    DtChangeDataUpdate
};

enum DtEmitterSysName
{
    DtEmitterSysOther
};

enum DtEmissionFunc
{
    DtEmissionFuncOther = 0,
    DtEmissionFuncMultiFunction = 1,
    DtEmissionFuncEarlyWarningSurveillance = 2,
    DtEmissionFuncHeightFinding = 3,
    DtEmissionFuncFireControl = 4,
    DtEmissionFuncAcquisitionDetection = 5,
    DtEmissionFuncTracking = 6,
    DtEmissionFuncGuidanceIllumination = 7,
    DtEmissionFuncFiringPointLaunchPointLocation = 8,
    DtEmissionFuncRanging = 9,
    DtEmissionFuncRadarAltimeter = 10,
    DtEmissionFuncImaging = 11,
    DtEmissionFuncMotionDetection = 12,
    DtEmissionFuncNavigation = 13,
    DtEmissionFuncWeather = 14,
    DtEmissionFuncInstrumentation = 15,
    DtEmissionFuncIdentificationClassification = 16,
    DtEmissionFuncJammingNoise = 64,
    DtEmissionFuncJammingDeception = 65,
    DtEmissionFuncDecoy = 66,
    DtEmissionFuncWeaponNonLethal = 96,
    DtEmissionFuncWeaponLethal = 97
};

```

```
};

enum DtBeamFunc
{
    DtBeamFuncOther = 0,
    DtBeamFuncSearch = 1,
    DtBeamFuncHeightFinder = 2,
    DtBeamFuncAcquisition = 3,
    DtBeamFuncTracking = 4,
    DtBeamFuncAcquisitionAndTracking = 5,
    DtBeamFuncCommandGuidance = 6,
    DtBeamFuncIllumination = 7,
    DtBeamFuncRangeOnlyRadar = 8,
    DtBeamFuncMissileBeacon = 9,
    DtBeamFuncMissileFuze = 10,
    DtBeamFuncActiveRadarMissileSeeker = 11,
    BeamFuncJammer = 12
};

enum DtTrackingJamming
{
    DtNotTrackingAllTargets,
    DtTrackingAllTargets
};

enum DtReferenceSys
{
    DtReferenceSysWorldCoordinates = 1,
    DtReferenceSysEntityCoordinates = 2
};

enum DtDesignatorCodeName
{
    DtDesignatorNameOther,
    DtDesignatorNameTBD
};

enum DtDesignatorCode
{
    DtDesignatorCodeOther
};

#if DtDISVERS == 3
enum DtLaserCodeName
{
    DtLaserNameOther
};

enum DtLaserCode
{
    DtLaserCodeOther
};
#endif /* DtDISVERS */

#define DtRadioOther 0
```

```
enum DtRadioTransmitState
{
    DtOff,
    DtRadioOnNotTrasmitting,
    DtOn
};

enum DtRadioInputSource
{
    DtRadioInputOther,
    DtPilot,
    DtCoPilot,
    DtFirstOfficer,
    DtDriver,
    DtLoader,
    DtGunner,
    DtCommander,
    DtDigitalDataDevice,
    DtIntercom
};

enum DtAntennaPatternType
{
    DtOmniDirectional,
    DtBeam,
    DtSphericalHarmonic
};

enum DtRadioSpreadSpectrum
{
    DtRadioSpreadOther,
    DtFrequencyHopping,
    DtPseudoNoise,
    DtTimeHopping,
    DtPseudoNoiseFrequencyHopping,
    DtFrequencyHoppingTimeHopping,
    DtPseudoNoiseTimeHopping,
    DtPseudoNoiseFrequencyHoppingTimeHopping
};

enum DtRadioMajorModulation
{
    DtRadioMajorModOther,
    DtAmplitude,
    DtAmplitudeAndAngle,
#ifdef DtDISVERS != 3
    DtAngle = 3,
    DtCombination = 4,
    DtPulse = 5,
    DtUnmodulated = 6
#else
    DtCombination = 3,
    DtPulse = 4,
    DtUnmodulated = 5
#endif
};
```

```
};

enum DtRadioDetailedModulationAmplitude
{
    DtRadioDetailModOther,
    DtAfsk,
    DtAm,
    DtCw,
    DtDsb,
    DtIsb,
    DtLsb,
    DtSsbFull,
    DtSsbReduc,
    DtUsb,
    DtVsb
};

typedef DtRadioDetailedModulationAmplitude DtRadioDetailedModulation;

enum DtDetailedModulationAmplitudeAngle
{
    DtRadioModAmpAngOther,
    DtDetailedAmplitudeAndAngle
};

enum DtDetailedModulationAngle
{
    DtRadioModeAngleOther,
    DtFm,
    DtFsk,
    DtPm
};

enum DtDetailedModulationCombination
{
    DtRadioDetailModCombo,
    DtAmplitudeAnglePulse
};

enum DtDetailedModulationUnmodulated
{
    DtRadioDetailModUnmod,
    DtContinuous = 1
};

enum DtDetailedModulationType
{
    DtDetailedModulationDummy
};

typedef DtDetailedModulationUnmodulated DetailedModulationUnmodulated;

enum DtRadioSystem
{
    DtRadioSystemOther,
```

```

    DtGeneric,
    DtHq,
    DtHqII,
    DtHqIIA,
    DtSincgards
};

enum DtRadioCryptoSystem
{
    DtRadioCryptoSysOther,
    DtRadioCryptoSysKY_28,
    DtRadioCryptoSysKY_8,
    DtRadioCryptoSysNarrowSpectrumSecureVoice_NSVE,
    DtRadioCryptoSysWideSpectrumSecureVoice_WSVE,
    DtRadioCryptoSysSINGARS_ICOM
};

enum DtEncodingClass
{
    DtSignalClassEncodedVoice,
    DtSignalClassRawBinary,
    DtSignalClassApplicationSpecific,
    DtSignalClassPreRecorded
};

typedef DtEncodingClass DtRadioSignalEncoding;
const DtEncodingClass DtEncodedVoice = DtEncodingClass(0);
const DtEncodingClass DtRawBinary = DtEncodingClass(1);
const DtEncodingClass DtApplicationSpecific = DtEncodingClass(2);
const DtEncodingClass DtPreRecordedVoice = DtEncodingClass(3);

enum DtRadioReceiverState
{
    DtRadioOff,
    DtRadioOnNotReceiving,
    DtRadioOn
};

enum DtTacticalDataLinkType
{
    DtTacticalDataLinkOther = 0
};

enum DtUserProtocol
{
    DtUserProtocolOther = 0
};

enum DtEncodingType
{
    DtEncodingTypeOther = 0
};

/* Section 7 */

```

```
enum DtDeadReckonTypes
```

```
{
    DtDrOther   = 0,
    DtDrStatic  = 1,
    DtDrDrmFpw  = 2,
    DtDrDrmRpw  = 3,
    DtDrDrmRvw  = 4,
    DtDrDrmFvw  = 5,
    DtDrDrmFpb  = 6,
    DtDrDrmRpb  = 7,
    DtDrDrmRvb  = 8,
    DtDrDrmFvb  = 9
};
```

```
enum DtViewerCommand
```

```
{
    DtViewerAbsolute= 0,
    DtViewerTether= 1,
    DtViewerCompass= 2,
    DtViewerMimic= 3,
    DtViewerTurret= 4,
    DtViewerOrbit= 5,
    DtViewerTrack= 6,
    DtViewerTetherTrack= 7,
    DtViewerMimicTrack= 8,
    DtViewerGroup= 9,
    DtViewerModeless= 10
};
```

```
enum DtLgrCommand
```

```
{
    DtLgrPbFile= 0,
    DtLgrRecFile= 1,
    DtLgrQuit= 2,
    DtLgrStart= 3,
    DtLgrEnd= 4,
    DtLgrPlay= 5,
    DtLgrPause= 6,
    DtLgrStop= 7,
    DtLgrRecord= 8,
    DtLgrSetTime= 9,
    DtLgrSkipTime= 10,
    DtLgrTimeScale= 11,
    DtLgrPlayAction= 12,
    DtLgrRecordAction= 13,
    DtLgrEofAction= 14,
    DtLgrHeartbeat= 15,
    DtLgrTimeout= 16
};
```

```
enum DtAppearAttributeBitShift
```

```
{
    DtAppPaintSchemeBitShift = 0,
    DtAppMobilityKillBitShift = 1,
    DtAppFirePowerKillBitShift = 2,
};
```

```

    DtAppDamageBitShift =      3,
    DtAppSmokeBitShift =      5,
    DtAppTrailingEffectsBitShift = 7,
    DtAppHatchBitShift = 9,
    DtAppLightsBitShift = 12,
    DtAppFlamingBitShift = 15,
#if DtDISVERS != 3
    DtAppFrozenBitShift = 21,
    DtAppPowerPlantStatusBitShift = 22,
#endif
#if !DtHLA
    DtAppFinalPduBitShift = 23,
#endif
    DtAppDummyBitShift =      99
};

// The rest are entity type specific, and thus there are several
// names with the same value. Define them as consts below.

// This macro creates definitions that look like this:
// const DtAppearAttributeBitShift DtAppDensityBitShift =
//     DtAppearAttributeBitShift(16);

#define DtEXTEND_BIT_SHIFT_ENUM(attr, val) \
    const DtAppearAttributeBitShift DtApp##attr##BitShift = \
        DtAppearAttributeBitShift(val)

DtEXTEND_BIT_SHIFT_ENUM(ActiveState, 23);
DtEXTEND_BIT_SHIFT_ENUM(Launcher, 16);
DtEXTEND_BIT_SHIFT_ENUM(CamoType, 17);
DtEXTEND_BIT_SHIFT_ENUM(Concealed, 19);
DtEXTEND_BIT_SHIFT_ENUM(Tent, 24);
DtEXTEND_BIT_SHIFT_ENUM(Ramp, 25);
DtEXTEND_BIT_SHIFT_ENUM(AfterBurner, 16);
DtEXTEND_BIT_SHIFT_ENUM(LaunchFlash, 16);
#if DtDISVERS != 3
DtEXTEND_BIT_SHIFT_ENUM(LifeformState, 16);
DtEXTEND_BIT_SHIFT_ENUM(Weapon1, 24);
DtEXTEND_BIT_SHIFT_ENUM(Weapon2, 26);
#else
DtEXTEND_BIT_SHIFT_ENUM(LifeformState, 16);
DtEXTEND_BIT_SHIFT_ENUM(Weapon1, 18);
DtEXTEND_BIT_SHIFT_ENUM(Weapon2, 19);
#endif
DtEXTEND_BIT_SHIFT_ENUM(Density, 16);

// DMO enums used by 2.1.4 PDUs

// Minefield appearance

DtEXTEND_BIT_SHIFT_ENUM(MinefieldType, 0);
DtEXTEND_BIT_SHIFT_ENUM(MinefieldActiveStatus, 2);
DtEXTEND_BIT_SHIFT_ENUM(MinefieldLane, 3);
DtEXTEND_BIT_SHIFT_ENUM(MinefieldState, 13);

```

```
// Point, Linear, and Areal Object appearance

DtEXTEND_BIT_SHIFT_ENUM(ObjectPercentComplete, 0);
DtEXTEND_BIT_SHIFT_ENUM(ObjectDamage, 8);
DtEXTEND_BIT_SHIFT_ENUM(ObjectPredistributed, 10);
DtEXTEND_BIT_SHIFT_ENUM(ObjectState, 11);
DtEXTEND_BIT_SHIFT_ENUM(ObjectSmoking, 12);
DtEXTEND_BIT_SHIFT_ENUM(ObjectFlaming, 13);

// Point object specific appearance

// This is for the log crib, abatis, vehicle defilade, and infantry
// fighting position breach appearance.
DtEXTEND_BIT_SHIFT_ENUM(ObjectAbatisBreach, 0);

DtEXTEND_BIT_SHIFT_ENUM(ObjectBurstOpacity, 0);
DtEXTEND_BIT_SHIFT_ENUM(ObjectBurstSize, 8);
DtEXTEND_BIT_SHIFT_ENUM(ObjectBurstHeight, 16);
DtEXTEND_BIT_SHIFT_ENUM(ObjectBurstNumOfBursts, 24);
DtEXTEND_BIT_SHIFT_ENUM(ObjectBurstChemicalSmoke, 30);

DtEXTEND_BIT_SHIFT_ENUM(ObjectCraterSize, 0);

DtEXTEND_BIT_SHIFT_ENUM(ObjectRibbonBridgeNumOfSegments, 0);

// Linear object specific appearance

// This is for the tank ditch and concertina wire breach appearance
DtEXTEND_BIT_SHIFT_ENUM(ObjectTankDitchBreach, 0);
DtEXTEND_BIT_SHIFT_ENUM(ObjectTankDitchBreachLength, 16);
DtEXTEND_BIT_SHIFT_ENUM(ObjectTankDitchBreachLocation, 24);

DtEXTEND_BIT_SHIFT_ENUM(ObjectExhaustSmokeOpacity, 0);
DtEXTEND_BIT_SHIFT_ENUM(ObjectExhaustSmokeAttached, 8);
DtEXTEND_BIT_SHIFT_ENUM(ObjectExhaustSmokeChemical, 9);

DtEXTEND_BIT_SHIFT_ENUM(ObjectMinefieldLaneMarkerVisibleSide, 0);

// Areal object specific appearance

DtEXTEND_BIT_SHIFT_ENUM(ObjectMinefieldBreach, 0);
DtEXTEND_BIT_SHIFT_ENUM(ObjectMinefieldMineCount, 16);

// if you put spaces between the "<<" and the surrounding strings,
// enumstr gets confused and thinks the shift is another value.
enum DtAppearAttribute
{
    DtAppPaintScheme = 0x1<<DtAppPaintSchemeBitShift,
    DtAppMobilityKill = 0x1<<DtAppMobilityKillBitShift,
    DtAppFirePowerKill = 0x1<<DtAppFirePowerKillBitShift,
    DtAppDamage = 0x3<<DtAppDamageBitShift,
    DtAppSmoke = 0x3<<DtAppSmokeBitShift,
    DtAppTrailingEffects = 0x3<<DtAppTrailingEffectsBitShift,
    DtAppHatch = 0x7<<DtAppHatchBitShift,
    DtAppLights = 0x7<<DtAppLightsBitShift,
```



```

    DtAppFlaming = 0x1<<DtAppFlamingBitShift,
#if DtDISVERS != 3
    DtAppFrozen = 0x1<<DtAppFrozenBitShift,
    DtAppPowerPlantStatus = 0x1<<DtAppPowerPlantStatusBitShift,
#endif
#if !DtHLA
    DtAppFinalPdu = 0x1<<DtAppFinalPduBitShift,
#endif
    DtAppDummy = 99
};

// This macro creates definitions that look like this:
// const DtAppearAttribute DtAppDensity =
//     DtAppearAttribute(16 << DtAppDensityBitShift);

#define DtEXTEND_ATTR_ENUM(attr, val) \
    const DtAppearAttribute attr = \
        DtAppearAttribute(val << attr##BitShift)

DtEXTEND_ATTR_ENUM(DtAppActiveState, 0x1);
DtEXTEND_ATTR_ENUM(DtAppLauncher, 0x1);
DtEXTEND_ATTR_ENUM(DtAppCamoType, 0x3);
DtEXTEND_ATTR_ENUM(DtAppConcealed, 0x1);
DtEXTEND_ATTR_ENUM(DtAppTent, 0x1);
DtEXTEND_ATTR_ENUM(DtAppRamp, 0x1);
DtEXTEND_ATTR_ENUM(DtAppAfterBurner, 0x1);
DtEXTEND_ATTR_ENUM(DtAppLaunchFlash, 0x1);
#if DtDISVERS != 3
DtEXTEND_ATTR_ENUM(DtAppLifeformState, 0xf);
DtEXTEND_ATTR_ENUM(DtAppWeapon1, 0x3);
DtEXTEND_ATTR_ENUM(DtAppWeapon2, 0x3);
#else
DtEXTEND_ATTR_ENUM(DtAppLifeformState, 0x3);
DtEXTEND_ATTR_ENUM(DtAppWeapon1, 0x1);
DtEXTEND_ATTR_ENUM(DtAppWeapon2, 0x1);
#endif
DtEXTEND_ATTR_ENUM(DtAppDensity, 0xf);

// DMO enums used by 2.1.4 PDUs

// Minefield appearance

DtEXTEND_ATTR_ENUM(DtAppMinefieldType, 0x3);
DtEXTEND_ATTR_ENUM(DtAppMinefieldActiveStatus, 0x1);
DtEXTEND_ATTR_ENUM(DtAppMinefieldLane, 0x1);
DtEXTEND_ATTR_ENUM(DtAppMinefieldState, 0x1);

// Point, Linear, and Areal Object appearance

DtEXTEND_ATTR_ENUM(DtAppObjectPercentComplete, 0xff);
DtEXTEND_ATTR_ENUM(DtAppObjectDamage, 0x3);
DtEXTEND_ATTR_ENUM(DtAppObjectPredistributed, 0x1);
DtEXTEND_ATTR_ENUM(DtAppObjectState, 0x1);
DtEXTEND_ATTR_ENUM(DtAppObjectSmoking, 0x1);
DtEXTEND_ATTR_ENUM(DtAppObjectFlaming, 0x1);

```

```
// Point object specific appearance

// This is for the log crib, abatis, vehicle defilade, and infantry
// fighting position breach appearance.
DtEXTEND_ATTR_ENUM(DtAppObjectAbatisBreach, 0x3);

DtEXTEND_ATTR_ENUM(DtAppObjectBurstOpacity, 0xff);
DtEXTEND_ATTR_ENUM(DtAppObjectBurstSize, 0xff);
DtEXTEND_ATTR_ENUM(DtAppObjectBurstHeight, 0xff);
DtEXTEND_ATTR_ENUM(DtAppObjectBurstNumOfBursts, 0x3f);
DtEXTEND_ATTR_ENUM(DtAppObjectBurstChemicalSmoke, 0x3);

DtEXTEND_ATTR_ENUM(DtAppObjectCraterSize, 0xff);

DtEXTEND_ATTR_ENUM(DtAppObjectRibbonBridgeNumOfSegments, 0xff);

// Linear object specific appearance

// This is for the tank ditch and concertina wire breach appearance
DtEXTEND_ATTR_ENUM(DtAppObjectTankDitchBreach, 0x3);
DtEXTEND_ATTR_ENUM(DtAppObjectTankDitchBreachLength, 0xff);
DtEXTEND_ATTR_ENUM(DtAppObjectTankDitchBreachLocation, 0xff);

DtEXTEND_ATTR_ENUM(DtAppObjectExhaustSmokeOpacity, 0xff);
DtEXTEND_ATTR_ENUM(DtAppObjectExhaustSmokeAttached, 0x1);
DtEXTEND_ATTR_ENUM(DtAppObjectExhaustSmokeChemical, 0x3);

DtEXTEND_ATTR_ENUM(DtAppObjectMinefieldLaneMarkerVisibleSide, 0x3);

// Areal object specific appearance

DtEXTEND_ATTR_ENUM(DtAppObjectMinefieldBreach, 0x3);
DtEXTEND_ATTR_ENUM(DtAppObjectMinefieldMineCount, 0xffff);

// Create an enumeration type, but since many of the names will have
// the same value, define them all as consts below.
enum DtAppearAttrVal
{
    DtAppearInvalid = -1 // For DEC alpha
};

#define DtEXTEND_ATTR_VAL_ENUM(name, val, attr) \
    DtEXTEND_ENUM(DtAppearAttrVal, name, \
        val << DtApp##attr##BitShift)

DtEXTEND_ATTR_VAL_ENUM(DtPaintSchemeUniformBitVal, 0, PaintScheme);
DtEXTEND_ATTR_VAL_ENUM(DtPaintSchemeCamouflageBitVal, 1, PaintScheme);

DtEXTEND_ATTR_VAL_ENUM(DtMobilityKillNoneBitVal, 0, MobilityKill);
DtEXTEND_ATTR_VAL_ENUM(DtMobilityKillDisabledBitVal, 1, MobilityKill);

DtEXTEND_ATTR_VAL_ENUM(DtFirePowerKillNoneBitVal, 0, FirePowerKill);
DtEXTEND_ATTR_VAL_ENUM(DtFirePowerKillDisabledBitVal, 1, FirePowerKill);
```

```

DtEXTEND_ATTR_VAL_ENUM(DtDamageNoneBitVal, 0, Damage);
DtEXTEND_ATTR_VAL_ENUM(DtDamageSlightBitVal, 1, Damage);
DtEXTEND_ATTR_VAL_ENUM(DtDamageModerateBitVal, 2, Damage);
DtEXTEND_ATTR_VAL_ENUM(DtDamageDestroyedBitVal, 3, Damage);

DtEXTEND_ATTR_VAL_ENUM(DtSmokeNoneBitVal, 0, Smoke);
DtEXTEND_ATTR_VAL_ENUM(DtSmokeRisingBitVal, 1, Smoke);
DtEXTEND_ATTR_VAL_ENUM(DtSmokeEngineBitVal, 2, Smoke);
DtEXTEND_ATTR_VAL_ENUM(DtSmokeEngineAndPlumeBitVal, 3, Smoke);

DtEXTEND_ATTR_VAL_ENUM(DtTrailingEffectsNoneBitVal, 0, TrailingEffects);
DtEXTEND_ATTR_VAL_ENUM(DtTrailingEffectsSmallBitVal, 1, TrailingEffects);
DtEXTEND_ATTR_VAL_ENUM(DtTrailingEffectsMediumBitVal, 2, TrailingEffects);
DtEXTEND_ATTR_VAL_ENUM(DtTrailingEffectsLargeBitVal, 3, TrailingEffects);

DtEXTEND_ATTR_VAL_ENUM(DtHatchNABitVal, 0, Hatch);
DtEXTEND_ATTR_VAL_ENUM(DtHatchClosedBitVal, 1, Hatch);
DtEXTEND_ATTR_VAL_ENUM(DtHatchPoppedBitVal, 2, Hatch);
DtEXTEND_ATTR_VAL_ENUM(DtHatchPoppedAndPersonBitVal, 3, Hatch);
DtEXTEND_ATTR_VAL_ENUM(DtHatchOpenBitVal, 4, Hatch);
DtEXTEND_ATTR_VAL_ENUM(DtHatchOpenAndPersonBitVal, 5, Hatch);

DtEXTEND_ATTR_VAL_ENUM(DtLightsNoneBitVal, 0, Lights);
DtEXTEND_ATTR_VAL_ENUM(DtLightsRunningBitVal, 1, Lights);
DtEXTEND_ATTR_VAL_ENUM(DtLightsNavigationBitVal, 2, Lights);
DtEXTEND_ATTR_VAL_ENUM(DtLightsFormationBitVal, 3, Lights);

DtEXTEND_ATTR_VAL_ENUM(DtFlamingNoBitVal, 0, Flaming);
DtEXTEND_ATTR_VAL_ENUM(DtFlamingYesBitVal, 1, Flaming);

#if !DtHLA
DtEXTEND_ATTR_VAL_ENUM(DtNotFinalPduBitVal, 0, FinalPdu);
DtEXTEND_ATTR_VAL_ENUM(DtFinalPduBitVal, 1, FinalPdu);
#endif

#if DtDISVERS != 3
DtEXTEND_ATTR_VAL_ENUM(DtNotFrozenBitVal, 0, Frozen);
DtEXTEND_ATTR_VAL_ENUM(DtFrozenBitVal, 1, Frozen);

DtEXTEND_ATTR_VAL_ENUM(DtPowerPlantOffBitVal, 0, PowerPlantStatus);
DtEXTEND_ATTR_VAL_ENUM(DtPowerPlantOnBitVal, 1, PowerPlantStatus);
#endif

DtEXTEND_ATTR_VAL_ENUM(DtActiveBitVal, 0, ActiveState);
DtEXTEND_ATTR_VAL_ENUM(DtDeactivatedBitVal, 1, ActiveState);

DtEXTEND_ATTR_VAL_ENUM(DtLauncherNotRaisedBitVal, 0, Launcher);
DtEXTEND_ATTR_VAL_ENUM(DtLauncherRaisedBitVal, 1, Launcher);

DtEXTEND_ATTR_VAL_ENUM(DtDesertCamouflageBitVal, 0, CamoType);
DtEXTEND_ATTR_VAL_ENUM(DtWinterCamouflageBitVal, 1, CamoType);
DtEXTEND_ATTR_VAL_ENUM(DtForestCamouflageBitVal, 2, CamoType);

DtEXTEND_ATTR_VAL_ENUM(DtNotConcealedBitVal, 0, Concealed);
DtEXTEND_ATTR_VAL_ENUM(DtConcealedBitVal, 1, Concealed);

```

```
DtEXTEND_ATTR_VAL_ENUM(DtTentNotExtendedBitVal, 0, Tent);
DtEXTEND_ATTR_VAL_ENUM(DtTentExtendedBitVal, 1, Tent);

DtEXTEND_ATTR_VAL_ENUM(DtRampUpBitVal, 0, Ramp);
DtEXTEND_ATTR_VAL_ENUM(DtRampDownBitVal, 1, Ramp);

DtEXTEND_ATTR_VAL_ENUM(DtAfterBurnerNotOnBitVal, 0, AfterBurner);
DtEXTEND_ATTR_VAL_ENUM(DtAfterBurnerOnBitVal, 1, AfterBurner);

DtEXTEND_ATTR_VAL_ENUM(DtLaunchFlashNotPresentBitVal, 0, LaunchFlash);
DtEXTEND_ATTR_VAL_ENUM(DtLaunchFlashPresentBitVal, 1, LaunchFlash);

DtEXTEND_ATTR_VAL_ENUM(DtLifeformNABitVal, 0, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformStandingBitVal, 1, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformWalkingBitVal, 2, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformRunningBitVal, 3, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformKneelingBitVal, 4, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformProneBitVal, 5, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformCrawlingBitVal, 6, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformSwimmingBitVal, 7, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformParachutingBitVal, 8, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformJumpingBitVal, 9, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformSittingBitVal, 10, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformSquattingBitVal, 11, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformCrouchingBitVal, 12, LifeformState);
DtEXTEND_ATTR_VAL_ENUM(DtLifeformWadingBitVal, 13, LifeformState);

DtEXTEND_ATTR_VAL_ENUM(DtWeapon1None, 0, Weapon1);
DtEXTEND_ATTR_VAL_ENUM(DtWeapon1Stowed, 1, Weapon1);
DtEXTEND_ATTR_VAL_ENUM(DtWeapon1Deployed, 2, Weapon1);
DtEXTEND_ATTR_VAL_ENUM(DtWeapon1InFirePosition, 3, Weapon1);

DtEXTEND_ATTR_VAL_ENUM(DtWeapon2None, 0, Weapon2);
DtEXTEND_ATTR_VAL_ENUM(DtWeapon2Stowed, 1, Weapon2);
DtEXTEND_ATTR_VAL_ENUM(DtWeapon2Deployed, 2, Weapon2);
DtEXTEND_ATTR_VAL_ENUM(DtWeapon2InFirePosition, 3, Weapon2);

DtEXTEND_ATTR_VAL_ENUM(DtDensityClearBitVal, 0, Density);
DtEXTEND_ATTR_VAL_ENUM(DtDensityHazyBitVal, 1, Density);
DtEXTEND_ATTR_VAL_ENUM(DtDensityDenseBitVal, 2, Density);
DtEXTEND_ATTR_VAL_ENUM(DtDensityVeryDenseBitVal, 3, Density);
DtEXTEND_ATTR_VAL_ENUM(DtDensityOpaqueBitVal, 4, Density);

// DMO used by 2.1.4 PDUs

// Minefield appearance

DtEXTEND_ATTR_VAL_ENUM(DtMixedMinefield, 0, MinefieldType);
DtEXTEND_ATTR_VAL_ENUM(DtPureAntiPersonnelMinefield, 1, MinefieldType);
DtEXTEND_ATTR_VAL_ENUM(DtPureAntiTankMinefield, 2, MinefieldType);

DtEXTEND_ATTR_VAL_ENUM(DtActiveMinefield, 0, MinefieldActiveStatus);
DtEXTEND_ATTR_VAL_ENUM(DtInactiveMinefield, 1, MinefieldActiveStatus);
```

```

DtEXTEND_ATTR_VAL_ENUM(DtMinefieldHasActiveLane, 0, MinefieldLane);
DtEXTEND_ATTR_VAL_ENUM(DtMinefieldHasInactiveLane, 1, MinefieldLane);

DtEXTEND_ATTR_VAL_ENUM(DtActiveMinefieldState, 0, MinefieldState);
DtEXTEND_ATTR_VAL_ENUM(DtDeactivatedMinefieldState, 1, MinefieldState);

// Point, Linear, and Areal Object appearance

DtEXTEND_ATTR_VAL_ENUM(DtObjectNoDamage, 0, ObjectDamage);
DtEXTEND_ATTR_VAL_ENUM(DtObjectDamaged, 1, ObjectDamage);
DtEXTEND_ATTR_VAL_ENUM(DtObjectDestroyed, 2, ObjectDamage);

DtEXTEND_ATTR_VAL_ENUM(DtObjectCreatedDuringExercise, 0,
ObjectPredistributed);
DtEXTEND_ATTR_VAL_ENUM(DtObjectPredistributed, 1, ObjectPredistributed);

DtEXTEND_ATTR_VAL_ENUM(DtObjectActive, 0, ObjectState);
DtEXTEND_ATTR_VAL_ENUM(DtObjectDeactivated, 1, ObjectState);

DtEXTEND_ATTR_VAL_ENUM(DtObjectNotSmoking, 0, ObjectSmoking);
DtEXTEND_ATTR_VAL_ENUM(DtObjectSmoking, 1, ObjectSmoking);

DtEXTEND_ATTR_VAL_ENUM(DtObjectNotFlaming, 0, ObjectFlaming);
DtEXTEND_ATTR_VAL_ENUM(DtObjectFlaming, 1, ObjectFlaming);

// Point object specific appearance

// This is for the log crib, abatis, vehicle defilade, and infantry
// fighting position breach appearance.
DtEXTEND_ATTR_VAL_ENUM(DtObjectAbatisNoBreaching, 0, ObjectAbatisBreach);
DtEXTEND_ATTR_VAL_ENUM(DtObjectAbatisBreached, 1, ObjectAbatisBreach);
DtEXTEND_ATTR_VAL_ENUM(DtObjectAbatisCleared, 2, ObjectAbatisBreach);

DtEXTEND_ATTR_VAL_ENUM(DtObjectBurstOtherSmoke, 0,
ObjectBurstChemicalSmoke);
DtEXTEND_ATTR_VAL_ENUM(DtObjectBurstHydrochloricSmoke, 1,
ObjectBurstChemicalSmoke);
DtEXTEND_ATTR_VAL_ENUM(DtObjectBurstWhitePhosphorousSmoke, 2,
ObjectBurstChemicalSmoke);
DtEXTEND_ATTR_VAL_ENUM(DtObjectBurstRedPhosphorousSmoke, 3,
ObjectBurstChemicalSmoke);

// Linear object specific appearance

DtEXTEND_ATTR_VAL_ENUM(DtObjectTankDitchBreachNoBreaching, 0,
ObjectTankDitchBreach);
DtEXTEND_ATTR_VAL_ENUM(DtObjectTankDitchBreachSlightBreaching, 1,
ObjectTankDitchBreach);
DtEXTEND_ATTR_VAL_ENUM(DtObjectTankDitchBreachModerateBreaching, 2,
ObjectTankDitchBreach);
DtEXTEND_ATTR_VAL_ENUM(DtObjectTankDitchBreachCleared, 3,
ObjectTankDitchBreach);

DtEXTEND_ATTR_VAL_ENUM(DtObjectExhaustSmokeNotAttached, 0,
ObjectExhaustSmokeAttached);

```

```
DtEXTEND_ATTR_VAL_ENUM(DtObjectExhaustSmokeIsAttached, 1,
    ObjectExhaustSmokeAttached);

DtEXTEND_ATTR_VAL_ENUM(DtObjectExhaustOtherSmoke, 0,
    ObjectExhaustSmokeChemical);
DtEXTEND_ATTR_VAL_ENUM(DtObjectExhaustHydrochloricSmoke, 1,
    ObjectExhaustSmokeChemical);
DtEXTEND_ATTR_VAL_ENUM(DtObjectExhaustWhitePhosphorousSmoke, 2,
    ObjectExhaustSmokeChemical);
DtEXTEND_ATTR_VAL_ENUM(DtObjectExhaustRedPhosphorousSmoke, 3,
    ObjectExhaustSmokeChemical);

DtEXTEND_ATTR_VAL_ENUM(DtObjectMinefieldLeftSideVisible, 0,
    ObjectMinefieldLaneMarkerVisibleSide);
DtEXTEND_ATTR_VAL_ENUM(DtObjectMinefieldRightSideVisible, 1,
    ObjectMinefieldLaneMarkerVisibleSide);
DtEXTEND_ATTR_VAL_ENUM(DtObjectMinefieldBothSidesVisible, 2,
    ObjectMinefieldLaneMarkerVisibleSide);

DtEXTEND_ATTR_VAL_ENUM(DtObjectMinefieldNoBreaching, 0,
    ObjectMinefieldBreach);
DtEXTEND_ATTR_VAL_ENUM(DtObjectMinefieldBreached, 1,
    ObjectMinefieldBreach);
DtEXTEND_ATTR_VAL_ENUM(DtObjectMinefieldCleared, 2,
    ObjectMinefieldBreach);

// This is for the tank ditch and concertina wire breach appearance
// Areal object specific appearance

enum DtDamageState
{
    DtDamageNone,
    DtDamageSlight,
    DtDamageModerate,
    DtDamageDestroyed
};

enum DtTrailState
{
    DtTrailingEffectsNone,
    DtTrailingEffectsSmall,
    DtTrailingEffectsMedium,
    DtTrailingEffectsLarge
};

enum DtHatchState
{
    DtHatchNA,
    DtHatchClosed,
    DtHatchPopped,
    DtHatchPoppedAndPerson,
    DtHatchOpen,
    DtHatchOpenAndPerson
};
```

```
enum DtLightState
{
    DtLightsNone,
    DtLightsRunning,
    DtLightsNavigation,
    DtLightsFormation
};

enum DtLifeformState
{
    DtLifeformNA,
    DtLifeformStanding,
    DtLifeformWalking,
    DtLifeformRunning,
    DtLifeformKneeling,
    DtLifeformProne,
    DtLifeformCrawling,
    DtLifeformSwimming,
    DtLifeformParachuting,
    DtLifeformJumping,
    DtLifeformSitting,
    DtLifeformSquatting,
    DtLifeformCrouching,
    DtLifeformWading
};

enum DtCamouflageType
{
    DtDesertCamouflage,
    DtWinterCamouflage,
    DtForestCamouflage
};

enum DtWeaponState
{
    DtNoWeapon,
    DtWeaponStowed,
    DtWeaponDeployed,
    DtWeaponInFirePosition
};

enum DtCapability
{
    DtAmmunitionSupplyCap,
    DtFuelSupplyCap,
    DtRecoveryCap,
    DtRepairCap
};

enum DtDensityType
{
    DtDensityClear,
    DtDensityHazy,
    DtDensityDense,
    DtDensityVeryDense,
```

```
DtDensityOpaque
};

// Articulated parts
enum DtArtParamType
{
    DtApPosition= 1,
    DtApPositionRate= 2,
    DtApExtension= 3,
    DtApExtensionRate= 4,
    DtApX= 5,
    DtApXRate= 6,
    DtApY= 7,
    DtApYRate= 8,
    DtApZ= 9,
    DtApZRate= 10,
    DtApAzimuth= 11,
    DtApAzimuthRate= 12,
    DtApElevation= 13,
    DtApElevationRate= 14,
    DtApRotation= 15,
    DtApRotationRate= 16
};

enum DtArtPartType
{
    DtEntityBase= -1,
    DtRudder = 1024,
    DtLeftFlap = 1056,
    DtRightFlap= 1088,
    DtLeftAileron= 1120,
    DtRightAileron= 1152,

    DtPeriscope= 2048,
    DtGenericAntenna= 2080,
    DtSnorkel= 2112,

    DtLandingGear= 3072,
    DtTailHook= 3104,
    DtSpeedBrake= 3136,
    DtLeftWeaponBayDoor = 3168,
    DtRightWeaponBayDoor= 3200,
    DtTankAPCHatch= 3232,
    DtWingSweep= 3164,

    DtPrimaryTurret1= 4096,
    DtPrimaryTurret2= 4128,
    DtPrimaryTurret3= 4160,
    DtPrimaryTurret4= 4192,
    DtPrimaryTurret5= 4224,
    DtPrimaryTurret6= 4256,
    DtPrimaryTurret7= 4288,
    DtPrimaryTurret8= 4320,
    DtPrimaryTurret9= 4352,
    DtPrimaryTurret10= 4384,
```



```
DtPrimaryGun1= 4416,  
DtPrimaryGun2= 4448,  
DtPrimaryGun3= 4480,  
DtPrimaryGun4= 4512,  
DtPrimaryGun5= 4544,  
DtPrimaryGun6= 4576,  
DtPrimaryGun7= 4608,  
DtPrimaryGun8= 4640,  
DtPrimaryGun9= 4672,  
DtPrimaryGun10= 4704,
```

```
DtPrimaryLauncher1 = 4736,  
DtPrimaryLauncher2 = 4768,  
DtPrimaryLauncher3 = 4800,  
DtPrimaryLauncher4 = 4832,  
DtPrimaryLauncher5 = 4864,  
DtPrimaryLauncher6 = 4896,  
DtPrimaryLauncher7 = 4928,  
DtPrimaryLauncher8 = 4960,  
DtPrimaryLauncher9 = 4992,  
DtPrimaryLauncher10 = 5024,
```

```
DtPrimaryDefenseSys1 = 5056,  
DtPrimaryDefenseSys2 = 5088,  
DtPrimaryDefenseSys3 = 5120,  
DtPrimaryDefenseSys4 = 5152,  
DtPrimaryDefenseSys5 = 5184,  
DtPrimaryDefenseSys6 = 5216,  
DtPrimaryDefenseSys7 = 5248,  
DtPrimaryDefenseSys8 = 5280,  
DtPrimaryDefenseSys9 = 5312,  
DtPrimaryDefenseSys10= 5344,
```

```
DtPrimaryRadar1= 5376,  
DtPrimaryRadar2= 5408,  
DtPrimaryRadar3 = 5440,  
DtPrimaryRadar4 = 5472,  
DtPrimaryRadar5 = 5504,  
DtPrimaryRadar6 = 5536,  
DtPrimaryRadar7 = 5568,  
DtPrimaryRadar8 = 5600,  
DtPrimaryRadar9 = 5632,  
DtPrimaryRadar10= 5664,
```

```
DtSecondaryTurret1= 5696,  
DtSecondaryTurret2= 5728,  
DtSecondaryTurret3= 5760,  
DtSecondaryTurret4= 5792,  
DtSecondaryTurret5= 5824,  
DtSecondaryTurret6= 5856,  
DtSecondaryTurret7= 5888,  
DtSecondaryTurret8= 5920,  
DtSecondaryTurret9= 5952,  
DtSecondaryTurret10= 5984,
```

```
DtSecondaryGun1= 6016,
DtSecondaryGun2= 6048,
DtSecondaryGun3= 6080,
DtSecondaryGun4= 6112,
DtSecondaryGun5= 6144,
DtSecondaryGun6= 6176,
DtSecondaryGun7= 6208,
DtSecondaryGun8= 6240,
DtSecondaryGun9= 6272,
DtSecondaryGun10= 6304,

DtSecondaryLauncher1      = 6336,
DtSecondaryLauncher2      = 6368,
DtSecondaryLauncher3      = 6400,
DtSecondaryLauncher4      = 6432,
DtSecondaryLauncher5      = 6464,
DtSecondaryLauncher6      = 6496,
DtSecondaryLauncher7      = 6528,
DtSecondaryLauncher8      = 6560,
DtSecondaryLauncher9      = 6592,
DtSecondaryLauncher10     = 6624,

DtSecondaryDefenseSys1    = 6656,
DtSecondaryDefenseSys2    = 6688,
DtSecondaryDefenseSys3    = 6720,
DtSecondaryDefenseSys4    = 6752,
DtSecondaryDefenseSys5    = 6784,
DtSecondaryDefenseSys6    = 6816,
DtSecondaryDefenseSys7    = 6848,
DtSecondaryDefenseSys8    = 6880,
DtSecondaryDefenseSys9    = 6912,
DtSecondaryDefenseSys10= 6944,

DtSecondaryRadar1        = 6976,
DtSecondaryRadar2        = 7008,
DtSecondaryRadar3        = 7040,
DtSecondaryRadar4        = 7072,
DtSecondaryRadar5        = 7104,
DtSecondaryRadar6        = 7136,
DtSecondaryRadar7        = 7168,
DtSecondaryRadar8        = 7200,
DtSecondaryRadar9        = 7232,
DtSecondaryRadar10       = 7264
};

// DMO enums used by 2.1.4 PDUs

// Intercom Control PDU enums

enum DtIntercomCtlType
{
    DtIntercomCtlStatus      = 1,
    DtIntercomReqAck         = 2,
    DtIntercomReqNoAck       = 3,
```

```

    DtIntercomAck          = 4,
    DtIntercomNack         = 5
};

enum DtIntercomCommClass
{
    DtIntercomSimCommChnl  = 0,
    DtIntercomSimCommSupChnl = 1
};

enum DtIntercomCommType
{
    DtIntercomFDX          = 1,
    DtIntercomHDX_RecOnly  = 2,
    DtIntercomHDX_TransOnly = 3,
    DtIntercomHDX          = 4
};

enum DtIntercomCmdType
{
    DtIntercomNoCmd        = 0,
    DtIntercomCmdStatus    = 1,
    DtIntercomConnect      = 2,
    DtIntercomDisconnect   = 3,
    DtIntercomReset        = 4,
    DtIntercomOn           = 5,
    DtIntercomOff          = 6
};

enum DtIntercomTransLineState
{
    DtIntercomNotApplicable = 0,
    DtIntercomNotTransmitting = 1,
    DtIntercomTransmitting  = 2
};

enum DtIntercomDestLineStateCmd
{
    DtIntercomLS_NoCmd      = 0,
    DtIntercomSetLS_Trans   = 1,
    DtIntercomSetLS_NotTrans = 2,
    DtIntercomRetLocLS_Ctl  = 3
};

enum DtIntercomCtlRecType
{
    DtIntercomEntDestRec    = 1,
    DtIntercomGrpDestRec    = 2,
    DtIntercomGrpAsgmtRec   = 3
};

// Environmental Process PDU enums

enum DtEnvironmentalModelType
{

```

```
DtEnvModelInvalid = -1 // For DEC alpha
};

enum DtEnvironmentalStatusBit
{
    DtEnvironmentalStatusLast = 0,
    DtEnvironmentalStatusOn   = 1
};

// Gridded data PDU enums

enum DtGridDataRecType
{
    DtGridDataRecType0      = 0,
    DtGridDataRecType1      = 1,
    DtGridDataRecType2      = 2
};

enum DtGridAxisType
{
    DtGridAxisFixed         = 0,
    DtGridAxisVariable      = 1
};

// IFF/ATC/NAVAIDS PDU enums

enum DtIffAtcNav aidsSysName
{
    DtIffAtcNav aidsOtherSysName      = 0,
    DtIffAtcNav aidsMarkX              = 1,
    DtIffAtcNav aidsMarkXII           = 2,
    DtIffAtcNav aidsATCRBS             = 3,
    DtIffAtcNav aidsSoviet             = 4,
    DtIffAtcNav aidsModes              = 5,
    DtIffAtcNav aidsMarkX_XII_ATCRBS   = 6,
    DtIffAtcNav aidsMarkX_XII_ATCRBS_Modes = 7
};

enum DtIffAtcNav aidsSysType
{
    DtIffAtcNav aidsOtherSysType       = 0,
    DtIffAtcNav aidsTransponder        = 1,
    DtIffAtcNav aidsInterrogator       = 2,
    DtIffAtcNav aidsSovietTransponder  = 3,
    DtIffAtcNav aidsSovietInterrogator = 4
};

enum DtIffAtcNav aidsStatusBit
{
    DtIffAtcNav aidsSystemOnOff        = 0,
    DtIffAtcNav aidsParameter1         = 1,
    DtIffAtcNav aidsParameter2         = 2,
    DtIffAtcNav aidsParameter3         = 3,
    DtIffAtcNav aidsParameter4         = 4,
    DtIffAtcNav aidsParameter5         = 5,
```

```

    DtIffAtcNavAidsParameter6          = 6,
    DtIffAtcNavAidsOperationalStatus   = 7
};

enum DtIffAtcNavAidsSysMode
{
    DtIffAtcNavAidsModeInvalid = -1 // For DEC alpha
};

enum DtIffAtcNavAidsChangeOptionsBit
{
    DtIffAtcNavAidsChangeIndicator = 0,
    DtIffAtcNavAidsAlternateMode4  = 1,
    DtIffAtcNavAidsAlternateMode5C = 2
};

enum DtIffAtcNavAidsParameterBit
{
    DtIffAtcNavAidsParameterTCAS      = 12,
    DtIffAtcNavAidsParameterStatus    = 13,
    DtIffAtcNavAidsParameterDamage    = 14,
    DtIffAtcNavAidsParameterMalfunction = 15
};

enum DtIffAtcNavAidsAltParm4
{
    DtIffAtcNavAidsAltParm4Other      = 0,
    DtIffAtcNavAidsAltParm4Valid      = 1,
    DtIffAtcNavAidsAltParm4Invalid    = 2,
    DtIffAtcNavAidsAltParm4NoResponse = 3
};

enum DtIffAtcNavAidsModifier
{
    DtIffAtcNavAidsModifierInvalid = -1 // For DEC alpha
};

enum DtIffAtcNavAidsLayerSpecificInfo
{
    DtIffAtcNavAidsLayerInfoInvalid = -1 // For DEC alpha
};

enum DtIffAtcNavAidsParm1
{
    DtIffAtcNavAidsParm1Invalid = -1 // For DEC alpha
};

enum DtIffAtcNavAidsParm2
{
    DtIffAtcNavAidsParm2Invalid = -1 // For DEC alpha
};

enum DtIffAtcNavAidsApplicableModes
{
    DtIffAtcNavAidsAppModeInvalid = -1 // For DEC alpha
};

```

```
};

// Underwater accoustic PDU enums

enum DtUnderwaterAcousticUpdate
{
    DtUnderwaterAcousticStateUpdate      = 0,
    DtUnderwaterAcousticChangedDataUpdate = 1
};

enum DtUnderwaterAcousticSystemName
{
    DtUnderwaterAcousticOtherSystem      = 0,
    DtUnderwaterAcousticAN_BQQ_5        = 1,
    DtUnderwaterAcousticAN_SSQ_62       = 2,
    DtUnderwaterAcousticAN_SQS_23       = 3,
    DtUnderwaterAcousticAN_SQS_26       = 4,
    DtUnderwaterAcousticAN_SQS_53       = 5,
    DtUnderwaterAcousticALFS            = 6,
    DtUnderwaterAcousticLFA              = 7,
    DtUnderwaterAcousticAN_AQS_901      = 8,
    DtUnderwaterAcousticAN_AQS_902      = 9
};

enum DtUnderwaterAcousticFunction
{
    DtUnderwaterAcousticOtherFunction    = 0,
    DtUnderwaterAcousticPlatformSearch_detect_track = 1,
    DtUnderwaterAcousticNavigation       = 2,
    DtUnderwaterAcousticMineHunting      = 3,
    DtUnderwaterAcousticWeaponSearch_detect_track_detect = 4
};

enum DtUnderwaterAcousticActiveEmissionParmIndex
{
    DtUnderwaterAcousticActiveOtherIndex = 0
};

enum DtUnderwaterAcousticScanPattern
{
    DtUnderwaterAcousticScanPatternNotUsed = 0,
    DtUnderwaterAcousticConical            = 1,
    DtUnderwaterAcousticHelical            = 2,
    DtUnderwaterAcousticRaster             = 3,
    DtUnderwaterAcousticSectorSearch       = 4,
    DtUnderwaterAcousticContinuousSearch   = 5
};

enum DtUnderwaterAcousticPassiveParmIndex
{
    DtUnderwaterAcousticOtherPassiveIndex = 0
};

enum DtUnderwaterAcousticPropulsionPlantConfig
{
```

```

    DtUnderwaterAcousticOtherConfig      = 0,
    DtUnderwaterAcousticDiesel_electric  = 1,
    DtUnderwaterAcousticDiesel           = 2,
    DtUnderwaterAcousticBattery           = 3,
    DtUnderwaterAcousticNuclear           = 4,
    DtUnderwaterAcousticTurbineReduction  = 5,
    DtUnderwaterAcousticSteam             = 6,
    DtUnderwaterAcousticGasTurbine        = 7
};

enum DtUnderwaterAcousticHullMountedMaskerStatus
{
    DtUnderwaterAcousticHullMountedMaskerOff = 0,
    DtUnderwaterAcousticHullMountedMaskerOn  = 1
};

enum DtUnderwaterAcousticAdditionalPassiveActivityParmIndex
{
    DtUnderwaterAcousticOtherAdditionalIndex = 0,
    DtUnderwaterAcousticGrahamsMKV_CoffeeMaker = 1
};

// Transfer Control Request PDU enums

enum DtTransferType
{
    DtOtherTransfer      = 0,
    DtCtlReqTransferEntity = 1,
    DtAppReqTransferEntity = 2,
    DtReqExchangeEntity   = 3,
    DtCtlReqTransferEnvProc = 4,
    DtAppReqTransferEnvProc = 5,
    DtReqExchangeEnvProc  = 6
};

// Reliable service PDU enums

enum DtRequiredReliabilityService
{
    DtAcknowledged    = 0,
    DtUnacknowledged = 1
};

// IsPartOf PDU enums

enum DtNature
{
    DtOtherNature           = 0,
    DtHostFireableMunition  = 1,
    DtMunitionCarriedAsCargo = 2,
    DtFuelCarriedAsCargo    = 3,
    DtGunmountAttachedToHost = 4,
    DtComputerGeneratedForcesCarriedAsCargo = 5,
    DtVehicleCarriedAsCargo  = 6,
    DtEmitterMountedOnHost   = 7,

```

```
DtMobileCommandAndControlEntityCarriedAboardHost = 8,  
DtEntityStationedAt                             = 9,  
DtTeamMemberInformationWith                     = 10  
};
```

```
enum DtPosition  
{  
    DtOtherPosition = 0,  
    DtOnTopOf       = 1,  
    DtInsideOf      = 2  
};
```

```
enum DtStationName  
{  
    DtOtherStationName           = 0,  
    DtAircraftWingstation        = 1,  
    DtShipsForwardGunmountStarboard = 2,  
    DtShipsForwardGunmountPort   = 3,  
    DtShipsForwardGunmountCenterline = 4,  
    DtShipsAftGunmountStarboard  = 5,  
    DtShipsAftGunmountPort       = 6,  
    DtShipsAftGunmountCenterline = 7,  
    DtForwardTorpedoTube         = 8,  
    DtAftTorpedoTube             = 9,  
    DtBombBay                    = 10,  
    DtCargoBay                   = 11,  
    DtTruckBed                   = 12,  
    DtTrailerBed                 = 13,  
    DtWellDeck                   = 14,  
    DtOnStation_RNG_BRG         = 15,  
    DtOnStation_XYZ             = 16  
};
```

```
// Environmental process PDU enums
```

```
enum DtObjectKind  
{  
    DtOtherObjectKind           = 0,  
    DtObstacle                  = 1,  
    DtPreparedPosition          = 2,  
    DtObjectCulturalFeature     = 3,  
    DtPassageway                = 4,  
    DtTacticalSmoke             = 5,  
    DtObstacleMarker            = 6,  
    DtObstacleBreach           = 7  
};
```

```
enum DtModificationField  
{  
    DtModLocation               = 0,  
    DtModOrientation            = 1  
};
```

```
// Minefield PDU enums
```



```

enum DtMinefieldDataFilterField
{
    DtMineGroundBurialDepthOffset    = 0,
    DtMineWaterBurialDepthOffset     = 1,
    DtMineSnowBurialDepthOffset      = 2,
    DtMineOrientation                 = 3,
    DtMineThermalContrast             = 4,
    DtMineReflectance                 = 5,
    DtMineEmplacementAge              = 6,
    DtMineTripDetonationWireLocation = 7,
    DtMineFusing                      = 8,
    DtMineScalarDetectionCoefficient = 9,
    DtMinePaintScheme                 = 10
};

enum DtMinefieldFusing
{
    DtNoFuse                        = 0,
    DtOtherFuse                     = 1,
    DtPressureFuse                  = 2,
    DtMagneticFuse                  = 3,
    DtTiltRodFuse                   = 4,
    DtCommandFuse                   = 5,
    DtTripWireFuse                  = 6
};

enum DtMinefieldAntiHandlingDevice
{
    DtNoAntiHandlingDevice          = 0,
    DtAntiHandlingDevice            = 1
};

enum DtSensorTypeCategory
{
    DtOtherSensorTypeCategory = 0,
    DtOpticalSensorType       = 1,
    DtFLIR_SensorType         = 2,
    DtRADAR_SensorType        = 3,
    DtMagneticSensorType      = 4,
    DtLaserSensorType         = 5,
    DtSONAR_SensorType        = 6,
    DtPhysicalSensorType      = 7,
    DtMultispectralSensorType = 8
};

enum DtOpticalSensorTypeSubcategory
{
    DtUnaidedEyeActivelySearching      = 0,
    DtUnaidedEyeNotActivelySearching   = 1,
    DtBinoculars                       = 2,
    DtImageIntensifier                 = 3,
    DtHMMWV_OccActivelySearching       = 4,
    DtHMMWV_OccNotActivelySearching    = 5,
    DtTruckOccActivelySearching         = 6,
    DtTruckOccNotActivelySearching      = 7,

```

```
DtTrackedVehicle0ccClosedHatchActSearching    = 8,  
DtTrackedVehicle0ccClosedHatchNotActSearching = 9,  
DtTrackedVehicle0ccOpenHatchActSearching      = 10,  
DtTrackedVehicle0ccOpenHatchNotActSearching   = 11  
};
```

```
enum DtFLIR_SensorTypeSubcategory  
{  
    DtGeneric3_5    = 0,  
    DtGeneric8_12   = 1,  
    DtASTAMIDS_I    = 2,  
    DtASTAMIDS_II   = 3,  
    DtGSTAMIDS3_5   = 4,  
    DtGSTAMIDS8_12  = 5,  
    DtHSTAMIDS3_5   = 6,  
    DtHSTAMIDS8_12  = 7,  
    DtCOBRA3_5      = 8,  
    DtCOBRA8_12     = 9  
};
```

```
enum DtRADAR_SensorTypeSubcategory  
{  
    DtRADAR_Generic = 0,  
    DtGenericGPR    = 1,  
    DtGSTAMIDS_I    = 2,  
    DtGSTAMIDS_II   = 3,  
    DtHSTAMIDS_I    = 4,  
    DtHSTAMIDS_II   = 5  
};
```

```
enum DtMagneticSensorTypeSubcategory  
{  
    DtMagneticGeneric = 0,  
    DtAN_PSS_11       = 1,  
    DtAN_PSS_12       = 2,  
    DtGSTAMIDS        = 3  
};
```

```
enum DtLaserSensorTypeSubcategory  
{  
    DtLaserGeneric    = 0,  
    DtASTAMIDS         = 1  
};
```

```
enum DtSONARSensorTypeSubcategory  
{  
    DtSONAR_Generic = 0  
};
```

```
enum DtPhysicalSensorTypeSubcategory  
{  
    DtGenericProbe      = 0,  
    DtProbeMetalContent = 1,  
    DtProbeNoMetalContent = 2  
};
```

```

enum DtMultispectralSensorTypeSubcategory
{
    DtMultispectralGeneric = 0
};

enum DtMinefieldAlgae
{
    DtNoAlgae                = 0,
    DtLightAlgae              = 1,
    DtModerateAlgae           = 2,
    DtHeavyAlgae              = 3
};

enum DtMinefieldPaintScheme
{
    DtOtherPaintScheme        = 0,
    DtStandardPaintScheme     = 1,
    DtCamouflageDesertPaintScheme = 2,
    DtCamouflageJunglePaintScheme = 3,
    DtCamouflageSnowPaintScheme = 4,
    DtCamouflageGravelPaintScheme = 5,
    DtCamouflagePavementPaintScheme = 6,
    DtCamouflageSandPaintScheme = 7,
    DtNaturalWoodPaintScheme   = 8,
    DtClearPaintScheme         = 9,
    DtRedPaintScheme           = 10,
    DtBluePaintScheme          = 11,
    DtGreenPaintScheme          = 12,
    DtOlivePaintScheme         = 13,
    DtWhitePaintScheme         = 14,
    DtTanPaintScheme           = 15,
    DtBlackPaintScheme         = 16,
    DtYellowPaintScheme        = 17,
    DtBrownPaintScheme         = 18
};

enum DtMinefieldProtocol
{
    DtHeartbeatMode = 0,
    DtQRP_Mode      = 1
};

// Live entity protocol PDU enums

enum DtSubProtocol
{
    DtSubProtocolInvalid = -1 // For DEC alpha
};

enum DtCharacterSet
{
    DtUnusedCharacterSet      = 0,
    DtASCII_CharacterSet      = 1
};

```

```
enum DtLE_TSPI_FieldFlag
{
    DtLE_TSPI_Reserved           = 7,
    DtLE_TSPI_SystemSpecific     = 6,
    DtLE_TSPI_MeasuredSpeed      = 5,
    DtLE_TSPI_DeadReckoning      = 4,
    DtLE_TSPI_OrientationError   = 3,
    DtLE_TSPI_PositionError      = 2,
    DtLE_TSPI_EntityOrientation  = 1,
    DtLE_TSPI_EntityLinearVelocity = 0
};
```

```
enum DtLE_AppearanceField1Flag
{
    DtLE_AppearanceFlagField2    = 7,
    DtLE_AppearanceIR            = 6,
    DtLE_AppearanceVisual        = 5,
    DtLE_AppearanceCapabilities  = 4,
    DtLE_AppearanceMarking       = 3,
    DtLE_AppearanceAlternativeType = 2,
    DtLE_AppearanceType          = 1,
    DtLE_AppearanceForceID       = 0
};
```

```
enum DtLE_AppearanceField2Flag
{
    DtLE_AppearanceAudio         = 1,
    DtLE_AppearanceEM            = 0
};
```

```
enum DtLE_FireFieldFlag
{
    DtLE_FireLocation            = 6,
    DtLE_FireBurstQuantityRate   = 5,
    DtLE_FireBurstWarheadFuse    = 4,
    DtLE_FireEventSiteApp        = 3,
    DtLE_FireMunitionID          = 2,
    DtLE_FireMunitionSiteApp     = 1,
    DtLE_FireTargetID            = 0
};
```

```
enum DtLE_DetonFieldFlag
{
    DtLE_DetonField2             = 7,
    DtLE_DetonLocationEntityCoords = 6,
    DtLE_DetonBurstQuantityRate   = 5,
    DtLE_DetonBurstWarheadFuse    = 4,
    DtLE_DetonEventSiteApp        = 3,
    DtLE_DetonMunitionID          = 2,
    DtLE_DetonMunitionSiteApp     = 1,
    DtLE_DetonTargetID            = 0
};
```

```
enum DtLE_DetonField2Flag
```

```

{
    DtLE_DetonEventNumber      = 1,
    DtLE_DetonMunitionOrientation = 0
};

// IsGroupOf PDU enums

enum DtGroupedEntityCategory
{
    DtUndefinedGroupEntityCategory = 0,
    DtBasicGroundCombatVehicle     = 1,
    DtEnhancedGroundCombatVehicle  = 2,
    DtBasicGroundCombatSoldier     = 3,
    DtEnhancedGroundCombatSoldier  = 4,
    DtBasicRotorWingAircraft       = 5,
    DtEnhancedRotorWingAircraft    = 6,
    DtBasicFixedWingAircraft       = 7,
    DtEnhancedFixedWingAircraft    = 8,
    DtGroundLogisticsVehicle       = 9
};

enum DtRestStatus
{
    DtSoldierNotRested           = 0,
    DtSoldier1HourSleep          = 1,
    DtSoldier2HourSleep          = 2,
    DtSoldier3HourSleep          = 3,
    DtSoldier4HourSleep          = 4,
    DtSoldier5HourSleep          = 5,
    DtSoldier6HourSleep          = 6,
    DtSoldier7HourSleep          = 7,
    DtSoldierFullyRested         = 8
};

// Aggregate State PDU enums

enum DtAggregateState
{
    DtOtherAggregateState        = 0,
    DtAggregated                 = 1,
    DtDisaggregated              = 2,
    DtFullyDisaggregated         = 3,
    DtPseudoDisaggregated        = 4,
    DtPartiallyDisaggregated     = 5
};

enum DtAggregateFormation
{
    DtOtherAggregateFormation    = 0,
    DtAssemblyFormation          = 1,
    DtVeeFormation               = 2,
    DtWedgeFormation             = 3,
    DtLineFormation              = 4,
    DtColumnFormation            = 5
};

```

disPduFactory.h

```
#if !DtHLA

#include "pduFactory.h"

// class DtDisPduFactory:
//
// Instances of DtDisPduFactory are used to create DIS PDUs.
// This class knows how to create the various DIS pdus.

class DtDisPduFactory : public DtPduFactory
{
public:

    // default constructor
    // Adds creators for the various DIS pdus.
    DtDisPduFactory();

    // destructor
    virtual ~DtDisPduFactory();

    // copy constructor
    DtDisPduFactory(const DtDisPduFactory& orig);

    // assignment operator
    DtDisPduFactory& operator=(const DtDisPduFactory& orig);
};
#endif
```

dReflAggList.h

```

#if !DtHLA
#if DtDISVERS >= 5

#if WIN32 // DevStudio has a nested #if limit
#include "vlUtil.h"
#endif

#include "objIdHashLst.h"
#include "aggStatePdu.h"
#include "reflectedAgg.h"
#include "exerciseConn.h"

// class DtReflectedAggregateList:
//
// Instances of this class provide a protocol independant interface to a
// list of reflected aggregates in a DIS environment

class DtReflectedAggregateList
{
public:

    // constructor
    DtReflectedAggregateList(DtExerciseConn* exConn);

    // destructor
    virtual ~DtReflectedAggregateList();

    // lookup an aggregate from its object ID
    virtual DtReflectedAggregate* lookup(const DtObjectId& id) const;

    // get the first and last elements of the aggregate list
    virtual DtReflectedAggregate* first() const;
    virtual DtReflectedAggregate* last() const;

    // Get number of aggregates
    virtual int count() const;

    // process an incoming aggregate state pdu
    virtual void processAggregateState(const DtAggregateStatePdu& pdu);

    // process aggregate timeouts
    virtual void doTimeouts();

    // Get my exerciseConn.
    virtual DtExerciseConn* exerciseConn() const;

    // Aggregate added callback registration/deregistration
    virtual void addAggregateAdditionCallback(
        DtAggregateCallbackFunction cb, void* userData);
    virtual void removeAggregateAdditionCallback(
        DtAggregateCallbackFunction cb, void* userData);

```

```
// Aggregate removed callback registration/deregistration
virtual void addAggregateRemovalCallback(
    DtAggregateCallbackFunction cb, void* userData);
virtual void removeAggregateRemovalCallback(
    DtAggregateCallbackFunction cb, void* userData);

// Print the ASRs of all aggs
virtual void print();

// Set/Get the timeout interval
virtual void setTimeoutInterval(DtTime time);
virtual DtTime timeoutInterval();

protected:

// Assignment operator is protected because it is not fully implemented
DtReflectedAggregateList& operator=(
    const DtReflectedAggregateList& orig);

// Copy constructor is protected because it is not fully implemented
DtReflectedAggregateList(const DtReflectedAggregateList& orig);

// Create a new reflected aggregate
virtual DtReflectedAggregate* newReflectedAggregate(
    const DtEntityIdentifier& id, const DtEntityType& type) const;

// Remove a reflected aggregate from the list and delete it
virtual void removeAndDelete(
    DtReflectedAggregate* aggregate, char* msg = NULL);

// Called after an aggregate is added and before addition callbacks are
// processed. Right now, this function only calls DtInfo.
virtual void aggregateAdded(DtReflectedAggregate* agg);

// Determines if a PDU should be processed or not. Right now, this
function
// returns 1.
virtual int acceptPdu(const DtAggregateStatePdu& pdu) const;

// Determines if an aggregate should be timed out
virtual DtBoolean shouldBeTimedOut(DtReflectedAggregate* entity);

// Process aggregate addition/removal callbacks
virtual void processAggregateAdditionCallbacks(
    DtReflectedAggregate* agg);
virtual void processAggregateRemovalCallbacks(
    DtReflectedAggregate* agg);

protected:

// callback for the exercise connection
static void aggregateStateCb(DtAggregateStatePdu* pdu, void* usr);

// calls doTimeouts
```



```
static void doTimeoutsPostDrainCb(void* usr);

protected:

    DtExerciseConn* myExConn;
    DtObjectIdHashList myAggregateList;
    DtIntrusiveList myAggregateAddedCallbacks;
    DtIntrusiveList myAggregateRemovedCallbacks;
    DtTime myTimeoutInterval;
};
#endif
#endif
```

dReflectedAgg.h

```
#if !DtHLA

#if DtDISVERS >= 5

#if WIN32 // DevStudio has a nested #if limit
#include "vlUtil.h"
#endif

#include "aggregateSR.h"
#include "exerciseConn.h"
#include "dAggDec.h"
#include "aggStatePdu.h"
#include "globalObjDes.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtAggregateCallbackFunction)(
    DtReflectedAggregate* obj, void* userData);

// class DtReflectedAggregate:
//
// Instances of this class provide a protocol independant interface to a
// reflected aggregate in a DIS environment

class DtReflectedAggregate : public DtListedObj
{
public:

    // constructor
    DtReflectedAggregate(DtExerciseConn* conn,
        const DtEntityIdentifier& id,
        const DtEntityType& type);

    // destructor
    virtual ~DtReflectedAggregate();

    // get the aggregate state repository
    virtual DtAggregateStateRepository* aggregateStateRep() const;

    // synonym for aggregateStateRep
    virtual DtAggregateStateRepository* asr() const;

    // get the entity ID
    virtual const DtEntityIdentifier& aggregateID() const;

    // Get the aggregate's id. (For DIS, DtObjectId is DtEntityIdentifier).
    virtual const DtObjectId& id() const;

    // Get the aggregate's object's global object designator -
    // for DIS, it's the same as its DtEntityIdentifier.
    virtual const DtGlobalObjectDesignator& globalId() const;
```

```

// Process an aggregate state PDU
virtual void processAggregateState(const DtAggregateStatePdu& pdu);

// get the last time an update was sent
virtual DtTime lastSimTimeUpdated() const;

// get the next and previous elements in the reflected aggregate list
virtual DtReflectedAggregate* next() const;
virtual DtReflectedAggregate* prev() const;

// get the next and previous elements, wrap at the ends of the list
virtual DtReflectedAggregate* wrapNext() const;
virtual DtReflectedAggregate* wrapPrev() const;

// Aggregate update callback registration/deregistration
virtual void addPostUpdateCallback(
    DtAggregateCallbackFunction cb, void* userData);
virtual void removePostUpdateCallback(
    DtAggregateCallbackFunction cb, void* userData);

protected:

// copy constructor does not copy callback list
DtReflectedAggregate(const DtReflectedAggregate& orig);

// assignment operator does not copy callback list
DtReflectedAggregate& operator=(const DtReflectedAggregate& orig);

// process the post update callbacks
virtual void processPostUpdateCallbacks();

protected:

DtEntityIdentifier myId;
DtExerciseConn* myConn;
DtAggregateStateRepository* myStateRep;
DtAggregateStateDecoder* myAggregateDecoder;
DtTime myLastSimTimeUpdated;
DtIntrusiveList myAggregateStateUpdateCallbacks;
};
#endif
#endif

```

dReflectedEnt.h

```
#if !DtHLA

#include "entitySR.h"
#include "exerciseConn.h"
#include "dEntityDec.h"
#include "esPdu.h"
#include "globalObjDes.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtEntityCallbackFunction)(
    DtReflectedEntity* obj, void* userData);

// class DtReflectedEntity:
//
// Instances of this class provide a protocol independant interface to a
// reflected entity in a DIS environment

class DtReflectedEntity : public DtListedObj
{
public:

    // Constructor
    DtReflectedEntity(DtExerciseConn* conn,
        const DtEntityIdentifier& id,
        const DtEntityType& type);

    // Destructor
    virtual ~DtReflectedEntity();

    // Returns a pointer to the EntityStateRepository - through
    // which you can inspect current state information.
    virtual DtEntityStateRepository* entityStateRep() const;

    // Synonym for entityStateRep()
    virtual DtEntityStateRepository* esr() const;

    // Returns a pointer to the Decoder being used to decode
    // state messages into our ESR.
    virtual DtEntityStateDecoder* decoder();

    // Returns the entity identifier
    virtual const DtEntityIdentifier& entityId() const;

    // Get the entity's id. (For DIS, DtObjectId is DtEntityIdentifier).
    virtual const DtObjectId& id() const;

    // Get the entity's object's global object designator -
    // for DIS, it's the same as its DtEntityIdentifier.
    virtual const DtGlobalObjectDesignator& globalId() const;

    // Process an entity state PDU
```

```
virtual void processEntityState(const DtEntityStatePdu& pdu);

// Get the last time an update was sent
virtual DtTime lastSimTimeUpdated() const;

// get the next and previous elements in the reflected entity list
virtual DtReflectedEntity* next() const;
virtual DtReflectedEntity* prev() const;

// get the next and previous elements, wrap at the ends of the list
virtual DtReflectedEntity* wrapNext() const;
virtual DtReflectedEntity* wrapPrev() const;

// Entity state callback registration/deregistration
virtual void addPostUpdateCallback(
    DtEntityCallbackFunction cb, void* userData);
virtual void removePostUpdateCallback(
    DtEntityCallbackFunction cb, void* userData);

protected:

    // copy constructor
    DtReflectedEntity(const DtReflectedEntity& orig);

    // assignment operator
    DtReflectedEntity& operator=(const DtReflectedEntity& orig);

    // process the post update callbacks
    virtual void processPostUpdateCallbacks();

protected:

    DtEntityIdentifier myId;
    DtExerciseConn* myConn;
    DtEntityStateRepository* myEntityStateRep;
    DtEntityStateDecoder* myEntityDecoder;
    DtTime myLastSimTimeUpdated;
    DtIntrusiveList myEntityStateUpdateCallbacks;
};
#endif
```

dRefEntList.h

```
#if !DtHLA

#include "objIdHashLst.h"
#include "dReflectedEnt.h"
#include "exerciseConn.h"

// class DtReflectedEntityList:
//
// Instances of this class provide a protocol independant interface to a
// list of reflected entities in a DIS environment

class DtReflectedEntityList
{
public:

    // Constructor
    DtReflectedEntityList(DtExerciseConn* exConn,
        DtBoolean smoothCapable = DtFalse);

    // Destructor
    virtual ~DtReflectedEntityList();

    // lookup an entity from its object ID
    virtual DtReflectedEntity* lookup(const DtObjectId& id) const;

    // get the first and last elements of the entity list
    virtual DtReflectedEntity* first() const;
    virtual DtReflectedEntity* last() const;

    // Get number of entities
    virtual int count() const;

    // process an incoming entity state pdu
    virtual void processEntityState(const DtEntityStatePdu& pdu);

    // process entity timeouts
    virtual void doTimeouts();

    // Get my exerciseConn.
    virtual DtExerciseConn* exerciseConn() const;

    // Entity added callback registration/deregistration
    virtual void addEntityAdditionCallback(
        DtEntityCallbackFunction cb, void* userData);
    virtual void removeEntityAdditionCallback(
        DtEntityCallbackFunction cb, void* userData);

    // Entity removed callback registration/deregistration
    virtual void addEntityRemovalCallback(
        DtEntityCallbackFunction cb, void* userData);
    virtual void removeEntityRemovalCallback(
```

```

        DtEntityCallbackFunction cb, void* userData);

// Print the ESRs of all entities
virtual void print();

// Set/Get the timeout interval.
virtual void setTimeoutInterval(DtTime time);
virtual DtTime timeoutInterval();

// Set whether or not timeout processing occurs.
// Defaults to on.
virtual void setTimeoutProcessing(DtBoolean onOff);

protected:

// Assignment operator is protected because it is not fully implemented
DtReflectedEntityList& operator=(
    const DtReflectedEntityList& orig);

// Copy constructor is protected because it is not fully implemented
DtReflectedEntityList(const DtReflectedEntityList& orig);

// Create a new reflected entity
virtual DtReflectedEntity* newReflectedEntity(const DtEntityIdentifier&
id,
    const DtEntityType& type) const;

// Remove a reflected entity from the list and delete it
virtual void removeAndDelete(DtReflectedEntity* entity, char* msg =
NULL);

// Called after an entity is added and before addition callbacks are
// processed. Right now, this function only calls DtInfo.
virtual void entityAdded(DtReflectedEntity* ent);

// Determines if a PDU should be processed or not. Right now, this
function
// returns 1.
virtual int acceptPdu(const DtEntityStatePdu& pdu) const;

// Called when an entity sends its final PDU. Currently, this removes
// and deletes the entity.
virtual void finalPduIsSetAction(DtReflectedEntity* entity);

// Determines if an entity should be timed out
virtual DtBoolean shouldBeTimedOut(DtReflectedEntity* entity);

// Process entity addition/removal callbacks
virtual void processEntityAdditionCallbacks(DtReflectedEntity* ent);
virtual void processEntityRemovalCallbacks(DtReflectedEntity* ent);

protected:

// callback for the exercise connection
static void entityStateCb(DtEntityStatePdu* pdu, void* usr);

```

```
// calls doTimeouts
static void doTimeoutsPostDrainCb(void* usr);

protected:

    DtExerciseConn* myExConn;
    DtObjectIdHashList myEntityList;
    DtTime myTimeoutInterval;
    DtBoolean myTimeoutProcessingFlag;
    DtIntrusiveList myEntityAddedCallbacks;
    DtIntrusiveList myEntityRemovedCallbacks;
    DtBoolean mySmoothCapable;
};
#endif
```


dscvObjCbInfo.h

```

#ifdef DtHLA

#include "RTI.hh"
#include "StateMsg.h"
#include "hlaObj.h"

// class DtDiscoverObjectCbInfo:
//
// Instances of DtDiscoverObjectCbInfo are used for providing
// a callback when an object which meets the federate's subscription
// criteria is discovered.

typedef void (*DtDiscoverObjectCb)(DtHlaObject* obj, void* usr);

class DtDiscoverObjectCbInfo
{
public:

    // constructor
    DtDiscoverObjectCbInfo(RTI::ObjectClassHandle h,
        DtDiscoverObjectCb cb,
        void* usr);

    // destructor
    virtual ~DtDiscoverObjectCbInfo();

    // copy constructor
    DtDiscoverObjectCbInfo(
        const DtDiscoverObjectCbInfo& orig);

    // assignment operator
    DtDiscoverObjectCbInfo& operator=(
        const DtDiscoverObjectCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(const DtDiscoverObjectCbInfo* const other) const;

    // Check whether one callback info is equal to another.
    virtual int operator==(const DtDiscoverObjectCbInfo& other) const;

    // Get the object class handle
    virtual const RTI::ObjectClassHandle& objectClass() const;

    // Get the callback function.
    virtual const DtDiscoverObjectCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:

```

```
RTI::ObjectClassHandle myClass;  
DtDiscoverObjectCb myFunc;  
void* myUsr;  
};  
#endif
```

dStateMsg.h

```
#if !DtHLA

#include "pdu.h"

#include "chooseDefines.h"

// class DtStateMsg:
//
// Instances of DtStateMsg are used to represent state data
// communicated via the network in a DIS (as opposed to HLA) environment.

class DtStateMsg : public DtPdu
{
public:

    // default constructor
    DtStateMsg();

    // destructor
    virtual ~DtStateMsg();

    // copy constructor
    DtStateMsg(const DtStateMsg& orig);

    // assignment operator
    DtStateMsg& operator=(const DtStateMsg& orig);
};
#endif
```

elecEmitPdu.h

```
#if !DtHLA

#if DtDISVERS != 3

#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "packets/elecEmitP.h"
#define DtElectromagneticEmissionStaticSize 28

#define DtEMISSION_SUCCESS 0
#define DtBAD_SYS_NUM 1
#define DtBAD_BEAM_NUM 2
#define DtBAD_TARGET_NUM 3
#define DtBAD_SYS_ADDR 4
#define DtBAD_BEAM_ADDR 5
#define DtBAD_TARGET_ADDR 6
#define DtINCONSISTENT_SYS_LENGTHS_PDU_LENGTH 7
#define DtINCONSISTENT_BEAM_LENGTHS_SYS_LENGTH 8
#define DtINCONSISTENT_NUM_TARGETS_BEAM_LENGTH 9

class DtFundamentalParameterData
{
public:
    float frequency;
    float frequencyRange;
    float effectiveRadiatedPower;
    float pulseRepetitionFrequency;
    float pulseWidth;
    float beamAzCenter;
    float beamAzSweep;
    float beamElCenter;
    float beamElSweep;
    float beamSweepSync;
    void printData() const;
};

class DtTrackJamTarget
{
public:
    DtEntityIdentifier entityId;
    int emitterId;
    int beamId;
    void printData() const;
};

class DtElectromagneticEmissionPdu;
typedef void (*DtElectromagneticEmissionPduCb)(DtElectromagneticEmissionPdu
*pdu, void *usr);

class DtElectromagneticEmissionPdu : public DtPdu
{
```

```

public:

/***** STANDARD CONSTRUCTORS *****/

    DtElectromagneticEmissionPdu(DtBufferPtr buffer =
DtUSE_INTERNAL_BUFFER);
    DtElectromagneticEmissionPdu(const DtNetElectromagneticEmissionPdu
*initial,
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtElectromagneticEmissionPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
    DtElectromagneticEmissionPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &emittingEntityId() const;
    const DtEventID &eventId() const;
    DtEmissionPduType stateUpdateIndicator() const;
    int numSystems() const;

    /***** FOR A PARTICULAR SYSTEM *****/
    int systemDataWords(int sysNum, int *retcode = NULL) const;
    int numBeams(int sysNum, int *retcode = NULL) const;
    DtEmitterSysName systemName(int sysNum, int *retcode = NULL) const;
    DtEmissionFunc systemFunction(int sysNum, int *retcode = NULL) const;
    int emitterId(int sysNum, int *retcode = NULL) const;
    DtConstVector location(int sysNum, int *retcode = NULL) const;

    /***** FOR A PARTICULAR BEAM *****/
    int beamDataWords(int sysNum, int beamNum, int *retcode = NULL) const;
    int beamId(int sysNum, int beamNum, int *retcode = NULL) const;
    int beamParameterIndex(int sysNum, int beamNum,
int *retcode = NULL) const;
    const DtFundamentalParameterData
&parameterData(int sysNum, int beamNum, int *retcode = NULL) const;
    int beamFunction(int sysNum, int beamNum, int *retcode = NULL) const;
    int numTargetsInBeam(int sysNum, int beamNum,
int *retcode = NULL) const;
    DtTrackingJamming trackingAllTargets(int sysNum,
int beamNum,
int *retcode = NULL) const;
    unsigned long jammingModeSequence(int sysNum,
int beamNum,
int *retcode = NULL) const;
    const DtTrackJamTarget &trackJamTarget(int sysNum, int beamNum,
int targetNum,
int *retcode = NULL) const;

/***** MUTATOR FUNCTIONS *****/

    void setEmittingEntityId(const DtEntityIdentifier &id);

```

```
void setEventId(const DtEventID &id);
void setStateUpdateIndicator(DtEmissionPduType indicator);
void setNumSystems(int num);

/***** FOR A PARTICULAR SYSTEM *****/
void setNumBeams(int sysNum, int numBeams, int *retcode = NULL);
void setSystemName(int sysNum, DtEmitterSysName name,
    int *retcode = NULL);
void setSystemFunction(int sysNum, DtEmissionFunc func,
    int *retcode = NULL);
void setEmitterId(int sysNum, int id, int *retcode = NULL);
void setLocation(int sysNum, DtConstVector loc, int *retcode = NULL);

/***** FOR A PARTICULAR BEAM *****/
void setBeamId(int sysNum, int beamNum, int id, int *retcode = NULL);
void setBeamParameterIndex(int sysNum, int beamNum, int bpIndex,
    int *retcode = NULL);
void setParameterData(int sysNum, int beamNum,
    const DtFundamentalParameterData &data,
    int *retcode = NULL);
void setBeamFunction(int sysNum, int beamNum,
    int func, int *retcode = NULL);
void setNumTargetsInBeam(int sysNum, int beamNum,
    int num, int *retcode = NULL);
void setTrackingAllTargets(int sysNum, int beamNum,
    DtTrackingJamming trackJamAll,
    int *retcode = NULL);
void setJammingModeSequence(int sysNum, int beamNum,
    unsigned long seq, int *retcode = NULL);
void setTrackJamTarget(int sysNum, int beamNum, int targetNum,
    const DtTrackJamTarget &targ, int *retcode = NULL);

void printData() const;
protected:
    DtPduKind internalGetPduKind() const {return
DtElectromagneticEmissionPduKind;}
    DtEntityIdentifier cacheId;
    DtEventID cacheEvent;
    DtVector cacheLoc;
    DtFundamentalParameterData cacheData;
    DtTrackJamTarget cacheTarget;
    DtNetSystemData *emitterSystemAddr(int sysNum, int *retcode = NULL)
const;
    DtNetBeamData *emitterBeamAddr(int sysNum, int beamNum,
    int *retcode = NULL) const;
    DtNetEmitterTrackJam *targetAddr(int sysNum, int beamNum,
    int targetNum, int *retcode = NULL) const;
    int computedSystemDataLength(int sysNum) const;
    int computedBeamDataLength(int sysNum, int beamNum) const;
    void printSystem(int i) const;
    void printBeam(int i, int j) const;
    int addressInPacket(void *addr) const;
    int computeStatus() const;
};
```

```
#endif // DtDISVERS  
#endif // DtHLA
```

embedSystemSR.h

```
#include "disEnums.h"
#include "packets/transmitterP.h"
#include "hostStructs.h"
#include "stateRep.h"

// class DtEmbeddedSystemRepository:
//
// Instances of DtEmbeddedSystemRepository are

class DtEmbeddedSystemRepository : public DtStateRepository
{
public:

    // default constructor
    DtEmbeddedSystemRepository();

    // copy constructor
    DtEmbeddedSystemRepository(const DtEmbeddedSystemRepository& orig);

    // destructor
    virtual ~DtEmbeddedSystemRepository();

    // assignment operator
    DtEmbeddedSystemRepository& operator=(
        const DtEmbeddedSystemRepository& orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const;

    static DtEmbeddedSystemRepository* create();

    // Set/Get the entity identifier.
    virtual void setEntityId(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& entityId() const;

    // Set/Get the host identifier.
    virtual void setHostId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& hostId() const;

    // Set/Get the relative position.
    virtual void setRelativePosition(DtConstVector position);
    virtual DtConstVector relativePosition() const;

    virtual void printData() const;

    //embedded system class attributes
protected:
    DtEntityIdentifier        myEntityId;
    DtGlobalObjectDesignator  myHostId;
    DtVector                  myRelativePosition;
};
```


emissionPdu.h

```

#if !DtHLA

#if DtDISVERS == 3

#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "packets/emmissionP.h" /* sic */

#define DtEmissionStaticSize 28

#define DtEMISSION_SUCCESS 0
#define DtBAD_SYS_NUM 1
#define DtBAD_BEAM_NUM 2
#define DtBAD_TARGET_NUM 3
#define DtBAD_SYS_ADDR 4
#define DtBAD_BEAM_ADDR 5
#define DtBAD_TARGET_ADDR 6
#define DtINCONSISTENT_SYS_LENGTHS_PDU_LENGTH 7
#define DtINCONSISTENT_BEAM_LENGTHS_SYS_LENGTH 8
#define DtINCONSISTENT_NUM_TARGETS_BEAM_LENGTH 9

class DtFundamentalParameterData
{
public:
    float frequency;
    float frequencyRange;
    float effectiveRadiatedPower;
    float pulseRepetitionFrequency;
    float pulseWidth;
    float beamAzCenter;
    float beamAzSweep;
    float beamElCenter;
    float beamElSweep;
    float beamSweepSync;
    void printData() const;
};

class DtTrackJamTarget
{
public:
    DtEntityIdentifier entityId;
    int emitterId;
    int beamId;
    void printData() const;
};

class DtEmissionPdu;
typedef void (*DtEmissionPduCb)(DtEmissionPdu *pdu, void *usr);

class DtEmissionPdu : public DtPdu
{

```

```
public:

/***** STANDARD CONSTRUCTORS *****/

    DtEmissionPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEmissionPdu(const DtNetEmissionPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtEmissionPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtEmissionPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &emittingEntityId() const;
    const DtEventID &eventId() const;
    DtEmissionPduType stateUpdateIndicator() const;
    int numSystems() const;

/***** FOR A PARTICULAR SYSTEM *****/
    int systemDataWords(int sysNum, int *retcode = NULL) const;
    int numBeams(int sysNum, int *retcode = NULL) const;
    DtEmitterSysName systemName(int sysNum, int *retcode = NULL) const;
    DtEmissionFunc systemFunction(int sysNum, int *retcode = NULL) const;
    int emitterId(int sysNum, int *retcode = NULL) const;
    DtConstVector location(int sysNum, int *retcode = NULL) const;

/***** FOR A PARTICULAR BEAM *****/
    int beamDataWords(int sysNum, int beamNum, int *retcode = NULL) const;
    int beamId(int sysNum, int beamNum, int *retcode = NULL) const;
    int beamParameterIndex(int sysNum, int beamNum,
int *retcode = NULL) const;
    const DtFundamentalParameterData
&parameterData(int sysNum, int beamNum, int *retcode = NULL) const;
    int beamFunction(int sysNum, int beamNum, int *retcode = NULL) const;
    int numTargetsInBeam(int sysNum, int beamNum,
int *retcode = NULL) const;
    DtTrackingJamming trackingAllTargets(int sysNum,
int beamNum,
int *retcode = NULL) const;
    unsigned long jammingModeSequence(int sysNum,
int beamNum,
int *retcode = NULL) const;
    const DtTrackJamTarget &trackJamTarget(int sysNum, int beamNum,
int targetNum,
int *retcode = NULL) const;

/***** MUTATOR FUNCTIONS *****/

    void setEmittingEntityId(const DtEntityIdentifier &id);
    void setEventId(const DtEventID &id);
    void setStateUpdateIndicator(DtEmissionPduType indicator);
```

```

void setNumSystems(int num);

/***** FOR A PARTICULAR SYSTEM *****/
void setNumBeams(int sysNum, int numBeams, int *retcode = NULL);
void setSystemName(int sysNum, DtEmitterSysName name,
    int *retcode = NULL);
void setSystemFunction(int sysNum, DtEmissionFunc func,
    int *retcode = NULL);
void setEmitterId(int sysNum, int id, int *retcode = NULL);
void setLocation(int sysNum, DtConstVector loc, int *retcode = NULL);

/***** FOR A PARTICULAR BEAM *****/
void setBeamId(int sysNum, int beamNum, int id, int *retcode = NULL);
void setBeamParameterIndex(int sysNum, int beamNum, int bpIndex,
    int *retcode = NULL);
void setParameterData(int sysNum, int beamNum,
    const DtFundamentalParameterData &data,
    int *retcode = NULL);
void setBeamFunction(int sysNum, int beamNum,
    int func, int *retcode = NULL);
void setNumTargetsInBeam(int sysNum, int beamNum,
    int num, int *retcode = NULL);
void setTrackingAllTargets(int sysNum, int beamNum,
    DtTrackingJamming trackJamAll,
    int *retcode = NULL);
void setJammingModeSequence(int sysNum, int beamNum,
    unsigned long seq, int *retcode = NULL);
void setTrackJamTarget(int sysNum, int beamNum, int targetNum,
    const DtTrackJamTarget &targ, int *retcode = NULL);

void printData() const;
protected:
    DtPduKind internalGetPduKind() const {return DtEmissionPduKind;}
    DtEntityIdentifier cacheId;
    DtEventID cacheEvent;
    DtVector cacheLoc;
    DtFundamentalParameterData cacheData;
    DtTrackJamTarget cacheTarget;
    DtNetSystemData *emitterSystemAddr(int sysNum, int *retcode = NULL)
const;
    DtNetBeamData *emitterBeamAddr(int sysNum, int beamNum,
    int *retcode = NULL) const;
    DtNetEmitterTrackJam *targetAddr(int sysNum, int beamNum,
    int targetNum, int *retcode = NULL) const;
    int computedSystemDataLength(int sysNum) const;
    int computedBeamDataLength(int sysNum, int beamNum) const;
    void printSystem(int i) const;
    void printBeam(int i, int j) const;
    int addressInPacket(void *addr) const;
    int computeStatus() const;
};

#endif // DtDISVERS
#endif // !DtHLA

```

emitrSysCbInf.h

```
#if DtHLA || DtHLA_GATEWAY

#include "vllist.h"
#include "hlaDefines.h"

// Entity state callback function signature.
typedef void (*DtEmitterSystemCallbackFunction)(
    DtReflectedEmitterSystem* ent, void* userData);

#endif

// class DtEmitterSystemCallbackInfo:
//
// Instances of DtEmitterSystemCallbackInfo are

class DtEmitterSystemCallbackInfo : public DtListedObj
{
public:

    // default constructor
    DtEmitterSystemCallbackInfo();

    // constructor
    DtEmitterSystemCallbackInfo(
        DtEmitterSystemCallbackFunction cb,
        void* userData);

    // destructor
    virtual ~DtEmitterSystemCallbackInfo();

    // copy constructor
    DtEmitterSystemCallbackInfo(const DtEmitterSystemCallbackInfo& orig);

    // assignment operator
    DtEmitterSystemCallbackInfo& operator=
        (const DtEmitterSystemCallbackInfo& orig);

    // equality operators
    int operator==(const DtEmitterSystemCallbackInfo& orig) const;
    int operator!=(const DtEmitterSystemCallbackInfo& orig) const;

    // set/get the callback function
    virtual void setCallbackFunction(DtEmitterSystemCallbackFunction funct);
    virtual DtEmitterSystemCallbackFunction callbackFunction() const;

    // set/get the user data
    virtual void setUserData(void* data);
    virtual void* userData() const;

protected:
```

```
DtEmitterSystemCallbackFunction myCallbackFunction;  
void* myUserData;  
};
```

emitterSysDec.h

```
#if DtHLA

#ifndef DtEmitterSystemDecoder_H_
#define DtEmitterSystemDecoder_H_

#include "chooseDefines.h"
#include "baseEmbedDec.h"

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLARE_EMITTERSYS_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtEmitterSystemRepository, attrName)

// class DtEmitterSystemDecoder:
//
// Instances of DtEmitterSystemDecoder are

class DtEmitterSystemDecoder : public DtBaseEmbeddedSystemDecoder
{
public:

    // default constructor
    DtEmitterSystemDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtEmitterSystemDecoder();

    // copy constructor
    DtEmitterSystemDecoder(const DtEmitterSystemDecoder& orig);

    // assignment operator
    DtEmitterSystemDecoder& operator=(const DtEmitterSystemDecoder& orig);

public:

    // Create a DtBaseEntityDecoder. Caller is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
    DtDECLARE_EMITTERSYS_ATTR_DECODER(EmitterFunction);
```

```
DtDECLARE_EMITTERSYS_ATTR_DECODER(EmitterName);
DtDECLARE_EMITTERSYS_ATTR_DECODER(EmitterNumber);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_EMITTERSYS_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtEmitterSystemDecoder, DtEmitterSystemRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_EMITTERSYS_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif //DtEmitterSystemDecoder_H_
```

emitterSysEnc.h

```
#if DtHLA

#ifndef DtEmitterSystemEncoder_H_
#define DtEmitterSystemEncoder_H_

#include "chooseDefines.h"
#include "baseEmbedEnc.h"

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_EMITTERSYS_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtEmitterSystemRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_EMITTERSYS_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtEmitterSystemRepository, attrName)

// class DtEmitterSystemEncoder:
//
// Instances of DtEmitterSystemEncoder are

class DtEmitterSystemEncoder : public DtBaseEmbeddedSystemEncoder
{
public:

    // default constructor
    DtEmitterSystemEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtEmitterSystemEncoder();

    // copy constructor
    DtEmitterSystemEncoder(const DtEmitterSystemEncoder& orig);

    // assignment operator
    DtEmitterSystemEncoder& operator=(const DtEmitterSystemEncoder& orig);

public:

    // Create a DtBaseEntityEncoder. Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    DtDECLARE_EMITTERSYS_ATTR_CHECKER(EmitterFunction);
    DtDECLARE_EMITTERSYS_ATTR_CHECKER(EmitterName);
```

```

DtDECLARE_EMITTERSYS_ATTR_CHECKER(EmitterNumber);

DtDECLARE_EMITTERSYS_ATTR_ENCODER(EmitterFunction);
DtDECLARE_EMITTERSYS_ATTR_ENCODER(EmitterName);
DtDECLARE_EMITTERSYS_ATTR_ENCODER(EmitterNumber);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_EMITTERSYS_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtEmitterSystemEncoder, \
        DtEmitterSystemRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that check whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_EMITTERSYS_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtEmitterSystemEncoder, \
        DtEmitterSystemRepository, \
        attrName, inspector)

#endif //DtEmitterSystemEncoder_H_

```

emitterSysSR.h

```
#if DtHLA

#ifndef DtEmitterSystemRepository_H_
#define DtEmitterSystemRepository_H_

#include "embedSystemSR.h"
#include "emitterBeamSR.h"
#include "vllist.h"

//error codes for looking up individual beams
enum DtEmitterSysRetCode
{
    DtEmitterSuccess,
    DtBadBeamId,
    DtDuplicateBeamId
};

// class DtEmitterSystemRepository:
//
// Instances of DtEmitterSystemRepository are

class DtEmitterSystemRepository : public DtEmbeddedSystemRepository
{
public:

    // default constructor
    DtEmitterSystemRepository();

    // destructor
    virtual ~DtEmitterSystemRepository();

    // copy constructor
    DtEmitterSystemRepository(const DtEmitterSystemRepository& orig);

    // assignment operator
    DtEmitterSystemRepository& operator=(const DtEmitterSystemRepository&
orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const;

    static DtEmitterSystemRepository* create();

    // Set/Get the emitter function.
    virtual void setEmitterFunction(DtEmissionFunc function);
    virtual DtEmissionFunc emitterFunction() const;

    // Set/Get the emitter name.
    virtual void setEmitterName(DtEmitterSysName name);
    virtual DtEmitterSysName emitterName() const;
};

#endif
```

```

// Set/Get the emitter number.
virtual void setEmitterNumber(DtU8 number);
virtual DtU8 emitterNumber() const;

virtual void printData() const;

// Return a pointer to the first BeamSR with specified beamID
virtual DtEmitterBeamRepository* beamSRById(
    DtU8 beamId,
    DtEmitterSysRetCode* retcode = NULL);

// add/remove a new emitter beam within this system
// if add a beam with duplicate ID, retcode will be set
// to DtDuplicateBeam, and beamSRById will return
// the first beamSR with the given ID.
virtual DtEmitterBeamRepository* addBeam(
    DtU8 beamId = 0, DtBeamType type = DtRegularBeam,
    DtEmitterSysRetCode *retcode = NULL);
virtual DtEmitterSysRetCode removeBeam(DtU8 beamId);

// Returns number of beams
virtual int numBeams() const;

// Returns DtList of DtEmitterBeamStateRepositories
virtual DtList* beamList();

// Returns DtTrue if the give BSR is in this ESR's list of BSRs
DtBoolean isMember(DtEmitterBeamRepository* bsr) const;

protected:
    virtual void purgeBeamList();

protected:
    DtEmissionFunc          myEmitterFunction;
    DtEmitterSysName        myEmitterName;
    DtU8                   myEmitterNumber;

    //list of beams in this system.
    DtList                  myBeamList;
};
#endif //DtEmitterSystemRepository_H_

```

emittRBeamSR.h

```
#if DtHLA

#ifndef DtEmitterBeamRepository_H_
#define DtEmitterBeamRepository_H_

#include "stateRep.h"
#include "hostStructs.h"
#include "disEnums.h"
#include "vllist.h"
#include "objDesList.h"

enum DtBeamType
{
    DtRegularBeam = -1,
    DtTrackBeam = 1,
    DtJamBeam = 0
};

// class DtEmitterBeamRepository:
//
// Instances of DtEmitterBeamRepository are

class DtEmitterBeamRepository : public DtStateRepository
{
public:

    // default constructor
    DtEmitterBeamRepository();

    // destructor
    virtual ~DtEmitterBeamRepository();

    // copy constructor
    DtEmitterBeamRepository(const DtEmitterBeamRepository& orig);

    // assignment operator
    DtEmitterBeamRepository& operator=(
        const DtEmitterBeamRepository& orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const;

    static DtEmitterBeamRepository* create();

    // Set/Get Azimuth Center
    virtual void setAzimuthCenter(DtFloat32 center);
    virtual DtFloat32 azimuthCenter() const;

    // Set/Get Azimuth Sweep
    virtual void setAzimuthSweep(DtFloat32 sweep);
    virtual DtFloat32 azimuthSweep() const;
```

```
// Set/Get Beam Function
virtual void setBeamFunction(DtBeamFunc func);
virtual DtBeamFunc beamFunction() const;

// Set/Get Beam Id
virtual void setBeamId(DtU8 id);
virtual DtU8 beamId() const;

// Set/Get Beam Parameter Index
virtual void setBeamParameterIndex(DtU16 index);
virtual DtU16 beamParameterIndex() const;

// Set/Get Elevation Center
virtual void setElevationCenter(DtFloat32 center);
virtual DtFloat32 elevationCenter() const;

// Set/Get Elevation Sweep
virtual void setElevationSweep(DtFloat32 sweep);
virtual DtFloat32 elevationSweep() const;

// Set/Get Emitting System Id
virtual void setEmittingSystemId(const DtGlobalObjectDesignator& id);
virtual const DtGlobalObjectDesignator& emittingSystemId() const;

// Set/Get ERP
virtual void setERP(DtFloat32 erp);
virtual DtFloat32 ERP() const;

// Set/Get Entity Id
virtual void setEntityId(const DtEntityIdentifier& id);
virtual const DtEntityIdentifier& entityId() const;

// Set/Get Frequency
virtual void setFrequency(DtFloat32 freq);
virtual DtFloat32 frequency() const;

// Set/Get Frequency Range
virtual void setFrequencyRange(DtFloat32 freq);
virtual DtFloat32 frequencyRange() const;

// Set/Get PRF
virtual void setPRF(DtFloat32 prf);
virtual DtFloat32 PRF() const;

// Set/Get Pulse Width
virtual void setPulseWidth(DtFloat32 width);
virtual DtFloat32 pulseWidth() const;

// Set/Get Sweep Sync
virtual void setSweepSync(DtFloat32 sync);
virtual DtFloat32 sweepSync() const;

// Set/Get beam type - track, jam or neither.
virtual void setBeamType(DtBeamType flag);
```

```
virtual DtBeamType beamType() const;

// The next two accessors should only be called if the beam type
// is DtTrackBeam or DtJamBeam.

// Set/Get Jamming Mode Sequence
virtual void setJammingModeSequence(DtU32 sequence);
virtual DtU32 jammingModeSequence() const;

// Get set of track jam targets. Returns pointer to internal object.
// Do not delete. It will be deleted when the SR is deleted.
virtual const DtGlobalObjectDesignatorList& trackJamTargets() const;
virtual DtGlobalObjectDesignatorList* trackJamTargets();

virtual void printData() const;

protected:

    // Prints track/jam target information
    virtual void printTrackJam() const;

protected:
    // Common to emitter beams and track jam beams
    DtFloat32          myAzimuthCenter;
    DtFloat32          myAzimuthSweep;
    DtBeamFunc         myBeamFunction;
    DtU8               myBeamId;
    DtU16              myBeamParameterIndex;
    DtFloat32          myElevationCenter;
    DtFloat32          myElevationSweep;
    DtGlobalObjectDesignator myEmittingSystemId;
    DtFloat32          myERP;
    DtEntityIdentifier myEntityId;
    DtFloat32          myFrequency;
    DtFloat32          myFrequencyRange;
    DtFloat32          myPRF;
    DtFloat32          myPulseWidth;
    DtFloat32          mySweepSync;

    // For Track Jam Beams
    DtU32              myJammingModeSequence;
    DtGlobalObjectDesignatorList myTrackJamTargets;
    DtBeamType         myBeamType;
};
#endif //DtEmitterBeamRepository_H_
```

encFactory.h

```

#if DtHLA

#include "hStateEncoder.h"
#include "MachineTypes.h"

// class DtStateEncoderFactory:
//
// Instances of DtStateEncoderFactory are used to create encoders which
// know how to encode a state repository (or host representation)
// into an HLA state message (or network representation).

class DtStateEncoderFactory
{
public:

    // constructor
    DtStateEncoderFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtStateEncoderFactory();

    // copy constructor
    DtStateEncoderFactory(const DtStateEncoderFactory& orig);

    // assignment operator
    DtStateEncoderFactory& operator=(const DtStateEncoderFactory& orig);

    // Create an encoder for the specified object class.
    // Note: Caller is responsible for deleting.
    // Returns NULL if encoder creation function is not in the factory.
    virtual DtHlaStateEncoder* createEncoder(RTI::ObjectClassHandle hand);

    // Add a encoder prototype to the factory. The encoder prototype
    // will be cloned by createEncoder when called for the appropriate
    // handle. Caller deletes the encoder, but AFTER the encoder
    // factory no longer needs it. Returns any previous encoder which was
    // replaced by the add.
    virtual DtHlaStateEncoder* addEncoder(
        RTI::ObjectClassHandle hand,
        DtHlaStateEncoder* encoder);

    // This function is used to add an attribute encoder function
    // to several prototype encoders all at once - namely, those
    // associated with the object class denoted by hand, as well
    // as all of its subclasses. If applyToSubclasses is DtFalse,
    // then the function is only registered with the one encoder
    // prototype, making it equivalent to
    // encoder(hand)->addEncoder(hand, encoder));
    virtual void addAttributeEncoder(
        RTI::ObjectClassHandle classHand,
        RTI::AttributeHandle attrHand,

```

```
DtHlaStateEncoder::DtAttributeEncoder encoder,  
DtBoolean applyToSubclasses = DtTrue);  
  
// Same as above, but identify class by name  
virtual void addAttributeEncoder(  
    char* className,  
    RTI::AttributeHandle attrHand,  
    DtHlaStateEncoder::DtAttributeEncoder func,  
    DtBoolean applyToSubclasses = DtTrue);  
  
// Same as above, but identify attr by name  
virtual void addAttributeEncoder(  
    RTI::ObjectClassHandle classHand,  
    char* attrName,  
    DtHlaStateEncoder::DtAttributeEncoder func,  
    DtBoolean applyToSubclasses = DtTrue);  
  
// Same as above, but identify class and attr by name  
virtual void addAttributeEncoder(  
    char* className,  
    char* attrName,  
    DtHlaStateEncoder::DtAttributeEncoder func,  
    DtBoolean applyToSubclasses = DtTrue);  
  
// This function is used to add an attribute checker function  
// to several prototype encoders all at once - namely, those  
// associated with the object class denoted by hand, as well  
// as all of its subclasses. If applyToSubclasses is DtFalse,  
// then the function is only registered with the one encoder  
// prototype, making it equivalent to  
// encoder(hand)->addChecker(hand, encoder));  
virtual void addAttributeChecker(  
    RTI::ObjectClassHandle classHand,  
    RTI::AttributeHandle attrHand,  
    DtHlaStateEncoder::DtAttributeChecker encoder,  
    DtBoolean applyToSubclasses = DtTrue);  
  
// Same as above, but identify class by name  
virtual void addAttributeChecker(  
    char* className,  
    RTI::AttributeHandle attrHand,  
    DtHlaStateEncoder::DtAttributeChecker func,  
    DtBoolean applyToSubclasses = DtTrue);  
  
// Same as above, but identify attr by name  
virtual void addAttributeChecker(  
    RTI::ObjectClassHandle classHand,  
    char* attrName,  
    DtHlaStateEncoder::DtAttributeChecker func,  
    DtBoolean applyToSubclasses = DtTrue);  
  
// Same as above, but identify class and attr by name  
virtual void addAttributeChecker(  
    char* className,  
    char* attrName,
```



```
        DtHlaStateEncoder::DtAttributeChecker func,  
        DtBoolean applyToSubclasses = DtTrue);  
  
    // Returns the encoder prototype associated with handle by a previous  
    // call to addEncoder.  
    virtual DtHlaStateEncoder* encoder(RTI::ObjectClassHandle hand);  
  
protected:  
  
    // myProtos is an array of DtHlaObject::maxObjectClasses() items.  
    DtHlaStateEncoder** myProtos;  
    DtExerciseConn* myExConn;  
};  
#endif
```

entCbInfo.h

```
#ifndef DtEntityCallbackInfo_H_
#define DtEntityCallbackInfo_H_

#include "vllist.h"

#if DtHLA || DtHLA_GATEWAY
#include "hlaDefines.h"
#else
#include "disDefines.h"
#endif

class DtReflectedEntity;

// Entity state callback function signature.
typedef void (*DtEntityCallbackFunction)(
    DtReflectedEntity* ent, void* userData);

#include "chooseDefines.h"

// class DtEntityCallbackInfo:
//
// Instances of DtEntityCallbackInfo are used to contain the function
// and user data associated with a callback which delivers entity state
// information.

class DtEntityCallbackInfo : public DtListedObj
{
public:
    // default constructor
    DtEntityCallbackInfo();

    // constructor
    DtEntityCallbackInfo(DtEntityCallbackFunction cb, void* userData);

    // destructor
    virtual ~DtEntityCallbackInfo();

    // copy constructor
    DtEntityCallbackInfo(const DtEntityCallbackInfo& orig);

    // assignment operator
    DtEntityCallbackInfo& operator=(const DtEntityCallbackInfo& orig);

    // equality operators
    int operator==(const DtEntityCallbackInfo& orig) const;
    int operator!=(const DtEntityCallbackInfo& orig) const;

    // set/get the callback function
    virtual void setCallbackFunction(DtEntityCallbackFunction funct);
```

```
virtual DtEntityCallbackFunction callbackFunction() const;

// set/get the user data
virtual void setUserData(void* data);
virtual void* userData() const;

protected:

    DtEntityCallbackFunction myCallbackFunction;
    void* myUserData;

};

#endif
```

entitySR.h

```
#include "hostStructs.h"
#include "mathTypes.h"
#include "artPartList.h"
#include "attPartList.h"
#include "baseEntitySR.h"

// class DtEntityStateRepository:
//
// Instances of DtEntityStateRepository are used to maintain entity
// state. It provides a fat interface which covers various types of
// entities. As a result, individual fields may not apply depending
// on the type of entity.

class DtEntityStateRepository : public DtBaseEntityStateRepository
{
public:

    // default constructor
    DtEntityStateRepository();

    // destructor
    virtual ~DtEntityStateRepository();

    // copy constructor
    DtEntityStateRepository(const DtEntityStateRepository& orig);

    // assignment operator
    DtEntityStateRepository& operator=(const DtEntityStateRepository& orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const;

    // Set/Get the force type.
    virtual void setForceId(DtForceType force);
    virtual DtForceType forceId() const;

    // Set/Get the alternate entity type or guise.
    virtual void setGuise(const DtEntityType& type);
    virtual const DtEntityType& guise() const;

#if DtDISVERS != 3
    // Set/Get whether frozen or not.
    virtual void setFrozen(DtBoolean onOff);
    virtual DtBoolean frozen() const;
#endif

    // Set/Get the character set used for the marking.
    // Defaults to 1.
    virtual void setMarkingCharSet(int charSet);
    virtual int markingCharSet() const;
```

```

    // Set/Get the marking text.
    virtual void setMarkingText(const char* text);
    virtual const char* markingText() const;

#if DtDISVERS != 3
    // Set/Get whether entity's power plant is on or not.
    virtual void setPowerPlantOn(DtBoolean onOff);
    virtual DtBoolean powerPlantOn() const;
#endif

    // Set/Get the damage state.
    virtual void setDamageState(DtDamageState s);
    virtual DtDamageState damageState() const;

    // Set/Get whether engine smoke is on or not.
    virtual void setEngineSmokeOn(DtBoolean onOff);
    virtual DtBoolean engineSmokeOn() const;

    // Set/Get whether flames are present or not.
    virtual void setFlamesPresent(DtBoolean onOff);
    virtual DtBoolean flamesPresent() const;

    // Set/Get whether entity has fuel supplying capability.
    virtual void setHasFuelSupplyCap(DtBoolean onOff);
    virtual DtBoolean hasFuelSupplyCap() const;

    // Set/Get whether entity has recovery capability.
    virtual void setHasRecoveryCap(DtBoolean onOff);
    virtual DtBoolean hasRecoveryCap() const;

    // Set/Get whether entity has repair capability.
    virtual void setHasRepairCap(DtBoolean onOff);
    virtual DtBoolean hasRepairCap() const;

    // Set/Get the entity's hatch state.
    virtual void setHatchState(DtHatchState s);
    virtual DtHatchState hatchState() const;

    // Set/Get whether the entity is immobilized or not.
    virtual void setImmobilized(DtBoolean onOff);
    virtual DtBoolean immobilized() const;

    // Set/Get entity life form state.
    virtual void setLifeformState(DtLifeformState s);
    virtual DtLifeformState lifeformState() const;

    // Set/Get light state of entity.
    virtual void setLightState(DtLightState s);
    virtual DtLightState lightState() const;

    // Set/Get light state of entity. (Environment entities).
    virtual void setDensity(DtDensityType dens);
    virtual DtDensityType density() const;

    // Set/Get whether entity has ramp deployed or not.

```

```
virtual void setRampDeployed(DtBoolean onOff);
virtual DtBoolean rampDeployed() const;

// Set/Get whether a smoke plume is present or not.
virtual void setSmokePlumePresent(DtBoolean onOff);
virtual DtBoolean smokePlumePresent() const;

// Set/Get whether a tent is deployed or not.
virtual void setTentDeployed(DtBoolean onOff);
virtual DtBoolean tentDeployed() const;

// Set/Get trailing effect state.
virtual void setTrailState(DtTrailState s);
virtual DtTrailState trailState() const;

// Set/Get camouflage type.
virtual void setCamouflageType(DtCamouflageType type);
virtual DtCamouflageType camouflageType() const;

// Set/Get whether entity is fire power disabled or not.
virtual void setFirePowerDisabled(DtBoolean onOff);
virtual DtBoolean firePowerDisabled() const;

// Set/Get whether entity is concealed or not.
virtual void setIsConcealed(DtBoolean onOff);
virtual DtBoolean isConcealed() const;

// Set/Get whether entity has after burner on or not.
virtual void setAfterburnerOn(DtBoolean onOff);
virtual DtBoolean afterburnerOn() const;

// Set/Get whether entity has ammunition supply capability or not.
virtual void setHasAmmunitionSupplyCap(DtBoolean onOff);
virtual DtBoolean hasAmmunitionSupplyCap() const;

// Set/Get whether entity has launcher raised or not.
virtual void setLauncherRaised(DtBoolean onOff);
virtual DtBoolean launcherRaised() const;

// Set/Get whether launcher flash is present or not.
virtual void setLauncherFlashPresent(DtBoolean onOff);
virtual DtBoolean launcherFlashPresent() const;

// Set/Get primary weapon deployment state.
virtual void setPrimaryWeaponState(DtWeaponState s);
virtual DtWeaponState primaryWeaponState() const;

// Set/Get secondary weapon deployment state.
virtual void setSecondaryWeaponState(DtWeaponState s);
virtual DtWeaponState secondaryWeaponState() const;

// initialize articulated parts
virtual void initArtParts(const DtArtPartSpec* specList);

// get the articulated parts list
```

```

virtual const DtArtPartList* artPartList() const;
virtual DtArtPartList* artPartList();

// get the attached parts list
virtual const DtAttachedPartList* attPartList() const;
virtual DtAttachedPartList* attPartList();

#if DtHLA
protected:
#endif

// Set/Get the appearance field.
virtual void setAppearance(DtU32 app);
virtual DtU32 appearance() const;

// Set/Get an appearance attribute.
// attr is the appearance bitfield mask
// val is the hex number to be or'ed in
virtual void setAppearanceAttribute(const DtAppearAttribute attr,
    const DtAppearAttrVal val);
virtual DtAppearAttrVal appearanceAttribute(
    const DtAppearAttribute attr) const;

// Set/Get the capabilities field.
virtual void setCapabilities(DtU32 cap);
virtual DtU32 capabilities() const;

public:

// Print the state data.
virtual void printData() const;

public:

    static DtEntityStateRepository* create();

protected:

// Cast off const-ness for trivial internal change.
// Non-virtual since can't override on return type.
DtEntityStateRepository* self() const;

// Set the marking text.
// Non-virtual since meant to be used by constructor.
void internalSetMarkingText(const char* text);

protected:

DtArtPartList myArtPartList;
DtAttachedPartList myAttPartList;

DtForceType myForceId;
DtEntityType myGuise;
DtU32 myAppearance;
int myMarkingCharSet;

```

```
// myMarkingText is an array of theMaxMarkingSize.  
// String length is theMaxMarkingSize - 1 (doesn't include NULL).  
char* myMarkingText;  
DtU32 myCapabilities;  
  
protected:  
  
    static const int theMaxMarkingSize;  
  
};
```


EntityTypes.h

```

/*
 * We use enumerated types instead of manifest constants
 * so that cross-type assignment mistakes can be caught
 * at compile time.
 * -
 */

#include "packets/disVersion.h"

/*****
 * DtEntityKinds
 *****/

enum DtEntityKind
{
    DtEntityKindOther =0,
    DtPlatform =1,
    DtMunition =2,
    DtLifeForm =3,
    DtEnvironmental =4,
    DtCulturalFeature =5,
    DtSupply =6
};

/*****
 * Domains
 *****/

enum DtPlatformDomain
{
    DtPlatformDomainOther =0,
    DtPlatformDomainLand =1,
    DtPlatformDomainAir =2,
    DtPlatformDomainSurface=3,
    DtPlatformDomainSubsurface =4,
    DtPlatformDomainSpace =5
};

enum DtMunitionDomain
{
    DtMunitionDomainOther =0,
    DtAntiAir =1,
    DtAntiArmor =2,
    DtAntiGuidedMunition =3,
    DtAntiRadar =4,
    DtAntiSatellite =5,
    DtAntiShip =6,
    DtAntiSubmarine =7,
    DtAntiPersonnel = 8,
#ifdef DtDISVERS == 3
    DtBattlefieldSupport8 =8,
    DtBattlefieldSupport9 =9,
    DtStrategic =9,
    DtMiscellaneous =10,
    DtBomb = 12
#else
    DtBattlefieldSupport =9,
    DtStrategic =10
#endif
};

enum DtLifeFormDomain
{

```

```
DtLifeFormDomainOther =0,
DtLifeFormDomainLand =1,
DtLifeFormDomainAir =2,
DtLifeFormDomainSurface =3,
DtLifeFormDomainSubsurface =4,
DtLifeFormDomainSpace =5
};

enum DtEnvironmentalDomain
{
    DtEnvironmentalDomainOther =0,
    DtEnvironmentalDomainLand =1,
    DtEnvironmentalDomainAir =2,
    DtEnvironmentalDomainSurface =3,
    DtEnvironmentalDomainSubsurface =4,
    DtEnvironmentalDomainSpace =5
};

enum DtCulturalFeatureDomain
{
    DtCulturalFeatureDomainOther =0,
    DtCulturalFeatureDomainLand =1,
    DtCulturalFeatureDomainAir =2,
    DtCulturalFeatureDomainSurface =3,
    DtCulturalFeatureDomainSubsurface =4,
    DtCulturalFeatureDomainSpace =5
};

/*****
 * Country codes
 *****/

enum DtCountryCode
{
    DtCountryCodeOther =0,
    DtFrance = 71,
    DtIraq =102,
    DtUnionOfSovietSocialistRepublics = 222, /* Hard to keep current, isn't
it? */
    DtUnitedKingdom =224,
    DtUnitedStates =225
};

/*****
 * Platforms
 *****/

enum DtPlatformLand
{
    DtTank = 1,
    DtArmoredFightingVehicle =2,
    DtArmoredUtilityVehicle =3,
    DtSelfPropelledArtillery =4,
    DtTowedArtillery =5,
    DtSmallWheeledUtilityVehicle =6,
    DtLargeWheeledUtilityVehicle =7,
    DtSmallTrackedUtilityVehicle =8,
    DtLargeTrackedUtilityVehicle =9,
#if DtDISVERS == 3
    DtAmphibiousAssaultVehicle =50,
    DtMiscellaneousWeaponSystem =51,
#endif
    DtMortar =10
};

enum DtPlatformAir
{
```

```

    DtFighter =1,
    DtAttack =2,
    DtBomber =3,
    DtCargo =4,
    DtAntiSubmarinePatrol =5,
#if DtDISVERS == 3
    DtAttackHelicopter = 6,
    DtUtilityHelicopter =7,
    DtCargoHelicopter =50,
    DtObservationHelicopter =51,
    DtSeaGoingHelicopter =52,
    DtElectronicWarfare = 53,
    DtUnmannedAirVehicle = 54,
    DtSpyPlanes =54,
    DtSpecialOperations =55
#else
    DtAttackHelicopter = 20,
    DtUtilityHelicopter =21,
    DtSeaGoingHelicopter =22,
    DtElectronicWarfare = 6,
    DtUnmannedAirVehicle = 50,
    DtReconnaissance = 8,
    DtSpecialOperations =25
#endif
};

enum DtPlatformSurface
{
    DtCarrier = 1,
    DtCommandShip = 2,
    DtGuidedMissileCruiser = 3,
    DtGuidedMissileDestroyer = 4,
    DtDestroyer = 5,
    DtGuidedMissileFrigate = 6,
    DtPatrolCraft = 7,
    DtMineCounterMeasureShip = 8,
    DtDockLandingShip = 9,
    DtTankLandingShip = 10,
    DtLandingCraft = 11,
    DtFrigate = 50,
    DtBattleship = 51,
    DtHeavyCruiser = 52,
    DtDestroyerTender = 53,
    DtAmphibiousAssaultShip = 54,
    DtAmphibiousCargoShip = 55,
    DtAmphibiousTransportDock = 56,
    DtAmmunitionShip = 57,
    DtCombatStoresShip = 58
};

#if DtDISVERS == 3
enum DtPlatformSubsurface
{
    DtStrategicMissileSubmarine = 1,
    DtAttackSubmarine = 2,
    DtResearchSubmarine = 3
};
#endif

/*
 * Any Country
 */

enum DtPlatformLandAnySmallWheeledUtilityVehicle
{
    DtCivilianCar = 0
};

```

```
#if DtDISVERS == 3
enum DtPlatformLandAnyLargeWheeledUtilityVehicle
{
    DtTruck = 0,
    DtBus = 14
};
#endif

/*
 * US.
 */

enum DtPlatformLandUSTank
{
    #if DtDISVERS == 3
        DtVirtualPrototype = 5,
    #endif
    DtM1 = 1
};

enum DtPlatformLandUSArmoredFightingVehicle
{
    #if DtDISVERS == 3
        DtLOSAT = 3,
        DtM2 = 3,
        DtM3 = 4,
        DtM113 = 5
    #else
        DtM2 = 1,
        DtM3 = 1,
        DtM113 = 3
    #endif
};

#if DtDISVERS == 3
enum DtLOSATSpecific
{
    DtSpecificLOSAT = 1
};
#endif

enum DtPlatformLandUSArmoredUtilityVehicle
{
    DtM88 = 1
};

enum DtPlatformLandUSSelfPropelledArtillery
{
    DtM109 = 3
};

#if DtDISVERS == 3
enum DtPlatformLandUSSelfTowedArtillery
{
    DtPatriotLauncher = 5,
    DtThAADLauncher = 6
};
#endif

enum DtPlatformLandUSSmallWheeledUtilityVehicle
{
    DtUSCivilianCar = 0,
    #if DtDISVERS == 3
        DtM577 = 2,
    #endif
    DtHMMWV = 1
}
```

```

};

#if DtDISVERS == 3
enum DtPlatformLandUSLargeWheeledUtilityVehicle
{
    DtM977HEMTT = 2,
    DtM978HEMTT = 3,
    DtPatriotRadarTrailer = 12
};

enum DtPlatformLandUSMiscellaneousWeaponSystem
{
    Dt20mmVulcanAirDefenseSystem = 3 /* Alias FAADS */
};
#endif

enum DtPlatformAirUSFighter{
    DtF117 = 1,
    DtF16 = 3,
#if DtDISVERS == 3
    DtF15 = 7,
    DtF15E = 7,
    DtF18 = 15
#else
    DtF15 = 5,
    DtF15E = 5,
    DtF18 = 9
#endif
};

enum DtF16
{
    DtBasicF16 = 0,
    DtF16C = 3
};

enum DtPlatformAirUSAttack
{
    DtA10 = 4
};

enum DtPlatformAirUSCargo
{
    DtC130 = 1,
#if DtDISVERS == 3
    DtKC130 = 7,
    DtV22 = 8
#else
    DtKC130 = 1
#endif
};

enum DtPlatformAirUSAttackHelicopter
{
    DtAH64 = 1,
    DtAH1W = 2,
    DtRAH66 = 3
};

enum DtAH1WSpecific
{
    DtSpecificAH1W = 3
};
#endif
DtSpecificAH1W = 10
#endif
};

```

```
enum DtPlatformAirUSObservationHelicopter
{
    DtOH58 = 1
};

enum DtPlatformAirUSUnmannedAirVehicle
{
    DtUAV = 0
};

#if DtDISVERS == 3
enum DtPlatformAirUSElectronicWarfare
{
    DtE3 = 4,
    DtJSTARS = 6
};

enum DtPlatformAirUSSpyPlanes
{
    DtU2 = 3
};
#endif

enum DtPlatformAirUSSpecialOperations
{
    DtMH60 = 1,
    #if DtDISVERS == 3
    DtBiplane = 5,
    #endif
    DtMH53 = 3
};

enum DtPlatformSurfaceUSCarrier
{
    DtCV59 = 4
};

enum DtCV59Specific
{
    DtSpecificCV59 = 1
};

enum DtPlatformSurfaceUSGuidedMissileCruiser
{
    #if DtDISVERS == 3
    DtCG47AegisCruiser
    #else
    DtTiconderogaClass
    #endif
};

enum DtAegisCruiserSpecific
{
    DtTiconderoga = 1,
    DtYorktown = 2,
    DtVincennes = 3,
    DtValleyForge = 4,
    DtThomasSGates = 5
};

enum DtPlatformSurfaceUSGuidedMissileFrigate
{
    DtFFG7 = 1
};

enum DtFFG7Specific
```

```

{
    DtSpecificFFG7 = 1
};

enum DtPlatformSurfaceUSAmphibiousAssaultShip
{
    #if DtDISVERS == 3
        DtLHA = 1
    #else
        DtTarawaClass = 1
    #endif
};

enum DtLHASpecific
{
    DtTarawa = 1,
    DtSaipan = 2,
    DtBelleauWood = 3,
    DtNassau = 4,
    DtPeleliu = 5
};

enum DtPlatformSubsurfaceUSStrategicMissileSubmarine
{
    DtSBBNOhio = 1
};

/*
 * France.
 */

enum DtPlatformAirFranceUtilityHelicopter
{
    #if DtDISVERS == 3
        DtGazelle = 3
    #else
        DtGazelle = 6
    #endif
};

/*
 * UK
 */

enum DtPlatformLandUKTank
{
    #if DtDISVERS == 3
        DtChallengerMBT = 3
    #else
        DtChallengerMBT = 2
    #endif
};

/*
 * USSR.
 */

enum DtPlatformLandUSSRTank
{
    DtT72 = 2,
    #if DtDISVERS == 3
        DtT64 = 4
    #else
        DtT64 = 3
    #endif
};

```

```
enum DtT72Specific
{
    DtBasicT72= 1
};

enum DtPlatformLandUSSRArmoredFightingVehicle
{
    DtBMP1 = 1,
    DtBMP2 = 2,
    DtBRDM1 = 3,
    #if DtDISVERS == 3
        DtMCV80 = 3,
        DtACRV2 = 10,
    #endif
    DtMTLB = 7
};

#if DtDISVERS == 3
enum DtMTLBSpecific
{
    DtMTLBUTrackedVehicle = 17
};
#endif

enum DtPlatformLandUSSRSelfPropelledArtillery
{
    DtSS1ScudMissileLauncher = 10,
    DtZSU23_4 = 18,
    DtSA6Launcher = 19,
    DtSAMSite = 19,
    #if DtDISVERS == 3
        DtZSU30 = 23,
        DtSA10FlapLidRadar = 38,
    #else
        DtSA10FlapLidRadar = 42,
    #endif
    DtSA8LauncherRadar = 41
};

enum DtSA10Specific
{
    DtSpecificSA10 = 1
};

enum DtSA8Specific
{
    DtSpecificSA8 = 1
};

enum DtPlatformLandUSSRSelfTowedArtillery
{
    DtD20 = 3
};

#if DtDISVERS == 3
enum DtPlatformLandUSSRSmallWheeledUtilityVehicle
{
    DtSA2FanSongRadarTrailer = 8,
    DtSA3LowBlowRadarTrailer = 9
};
#endif

enum DtPlatformLandUSSRLargeWheeledUtilityVehicle
{
    DtZIL157 = 6,
    DtGAZ66 = 8,
    DtUral375 = 9
}
```



```

};

#if DtDISVERS == 3
enum DtPlatformLandUSSRSsmallTrackedUtilityVehicle
{
    DtSA6StraightFlushRadar = 10
};

enum DtPlatformLandUSSRLargeTrackedUtilityVehicle
{
    DtATTHHeavyTrackedVehicle = 5
};
#endif

enum DtPlatformAirUSSRAttackHelicopter
{
    DtMI28 = 1,
    DtMI24 = 2
};

enum DtPlatformAirUSSRFighter
{
    DtMiG29 = 2,
    DtMiG23 = 5
};

enum DtPlatformAirUSSRAttack
{
    DtSU25 = 8
};

enum DtPlatformAirUSSRBomber
{
    DtTU26 = 3,
    DtTU95 = 5
};

#if DtDISVERS == 3
enum DtPlatformAirUSSRElectronicWarfare
{
    DtEA3B = 2
};

enum DtEA3BSpecific
{
    DtSpecificEA3B = 2
};

enum DtPlatformSubsurfaceUSSRAttackSubmarine
{
    DtSSTangoClass = 20
};
#endif

/*****
 * Munitions.
 *****/

enum DtMunitionAntiAir
{
    DtMunitionGuided = 1,
    DtMunitionUnguided = 2
};

enum DtMunitionAntiArmor
{
    DtAntiArmorGuided = 1,

```

```
    DtAntiArmorUnguided = 2
};

enum DtMunitionAntiRadar
{
    DtAntiRadarGuided = 1,
    DtAntiRadarUnguided = 2
};

enum DtMunitionAntiShip
{
    DtAntiShipGuided = 1,
    DtAntiShipUnguided = 2
};

enum DtMunitionAntiSubmarine
{
    DtAntiSubmarineGuided = 1,
    DtAntiSubmarineUnguided = 2
};

enum DtMunitionAntiPersonnel
{
    DtAntiPersonnelGuided = 1,
    DtAntiPersonnelUnguided = 2
};

enum DtMunitionBattlefieldSupport
{
    DtBattlefieldSupportGuided = 1,
    DtBattlefieldSupportUnguided = 2
};

#if DtDISVERS == 3
enum DtMunitionBomb
{
    DtBombGuided = 1,
    DtBombUnguided = 2
};
#endif

/*
 * US.
 */

enum DtMunitionUSAntiAirGuided
{
    DtSidewinder = 1,
    DtAMRAAM = 2,
    DtHawk = 6,
    DtPhoenix = 8,
    DtRoland = 11,
    DtSeaSparrow = 12,
    DtSparrow = 13,
    DtStinger = 15,
    #if DtDISVERS == 3
    DtThAAD = 20,
    #endif
    DtPatriot = 16
};

#if DtDISVERS == 3
enum DtSidewinderSpecific
{
    DtAIM9 = 0,
    DtAIM9M = 1
};
#endif
```

```

enum DtMunitionUSAntiAirUnguided
{
    DtM60Cannon = 1,
    Dt20mmAntiAirUnguided = 1
};
#endif

enum DtMunitionUSAntiArmorGuided
{
    DtBGM71TOW = 1,
    DtHellfire = 3,
    #if DtDISVERS == 3
        DtLOSATMunition = 7,
        DtFOGM = 8,
    #endif
    DtMaverick = 4
};

#if DtDISVERS == 3
enum DtMaverickSpecific
{
    DtAGM65 = 0,
    DtAGM65D = 1
};
#endif

enum DtMunitionUSAntiArmorUnguided
{
    #if DtDISVERS == 3
        Dt25mm = 1,
        Dt105m = 3,
        Dt120mm = 4
    #else
        Dt25mm = 3,
        Dt105m = 10,
        Dt120mm = 13
    #endif
};

enum DtMunitionUSAntiRadarGuided
{
    DtAGM88HARM = 1
};

#if DtDISVERS == 3
enum DtMunitionUSAntiShipGuided
{
    DtHarpoon84A = 1,
    DtHarpoon84B = 2,
    DtHarpoon84C = 3,
    DtHarpoon84D = 4,
    DtMk48 = 6
};
#endif

enum DtMunitionUSAntiSubmarineGuided
{
    DtMk48AntiSubmarine = 2
};

enum DtMunitionUSAntiPersonnelUnguided
{
    Dt7_62mm = 2
};

enum DtMunitionUSBattlefieldSupportUnguided

```

```
{
    DtBattlefield20mm = 1,
    DtBattlefield30mm = 3,
    DtBattlefield105mm = 10,
#if DtDISVERS == 3
    Dt2_75inRocket = 19,
    DtUnguidedM60Cannon = 20,
#endif
    DtBattlefield120mm = 11
};

enum DtMunitionUSBattlefieldSupportGuided
{
    DtBGM109Tomahawk = 6
};

#if DtDISVERS == 3
enum DtTomahawkSpecific
{
    DtBasicTomahawk = 0,
    DtTomahawkBooster = 1,
    DtTomahawkWings = 2
};

enum DtMunitionUSBombUnguided
{
    DtCBU1 = 1,
    DtCBU2 = 2,
    DtCBU3 = 3,
    DtCBU4 = 4,
    DtCBU5 = 5,
    DtCBU6 = 6,
    DtCBU7 = 7,
    DtMk82 = 9,
    DtMk84 = 11
};
#endif

/*
 * USSR
 */

enum DtMunitionUSSRAntiAirGuided
{
    DtAA3Anan = 3,
    DtAA9Amos = 9,
#if DtDISVERS == 3
    DtSA2Guideline = 13,
    DtSA3Goa = 14,
    DtSA4Ganef = 15,
    DtSA6Gainful = 17,
    DtSA7Grail = 18,
    DtSA8Gecko = 19,
    DtSA9Gaskin = 20,
    DtSA10Grumble = 21
#else
    DtSA2Guideline = 14,
    DtSA3Goa = 15,
    DtSA4Ganef = 16,
    DtSA6Gainful = 18,
    DtSA7Grail = 19,
    DtSA8Gecko = 20,
    DtSA9Gaskin = 21,
    DtSA10Grumble = 22
#endif
};
```

```

#if DtDISVERS == 3
enum DtMunitionUSSRAntiAirUnguided
{
    DtZSU30AAA = 2
};

enum DtMunitionUSSRAntiArmorUnguided
{
    Dt30mm = 3
};

enum DtMunitionUSSRBattlefieldSupport
{
    DtSS1 = 10/* Alias SCUD */
};

enum DtMunitionUSSRBattlefieldSupportUnguided
{
    DtBattlefield25mmUSSR = 3,
    DtBattlefield120mUSSR = 15
};
#endif

enum DtMunitionUSSRAntiPersonnelUnguided
{
    Dt7_62mmUSSR = 2
};

#if DtDISVERS == 3
enum DtMunitionUSSRAntiSubmarineGuided
{
    DtAntiSubmarineE53 = 1
};

enum DtE53Specific
{
    DtET80A = 1,
    DtE5383 = 2,
    DtE5371 = 3
};
#endif

/*
 * UK
 */

enum DtMunitionUKAntiPersonnelUnguided
{
    Dt7_62mmUK = 1
};

enum DtMunitionUKBattlefieldSupportUnguided
{
    DtBattlefield105mmUK = 3
};

/*****
 * Life Forms.
 *****/

enum DtLifeFormLand
{
    DtDismountedInfantry = 1
};

enum DtNumberOfIndividualLifeForms
{

```

```
    DtFireTeam = 101,
    DtSquad = 102,
    DtPlatoon = 103,
    DtCompany = 104,
    DtBattalion = 105,
    DtRegiment = 106,
    DtDivision = 107,
    DtCorps = 108,
    DtArmy = 109
};

/*****
 * Environmental
 *****/

enum DtEnvironmentalAir
{
    DtEnvironmentalDomainAirOther = 0,
    DtSmoke = 1,
    DtFog = 2,
    DtCloud = 3,
    DtCloudWithRainFalling = 4,
    DtCloudWithSnowFalling = 5
};

/*****
 * Cultural Feature
 *****/

enum DtCulturalFeatureLand
{
    DtBridgeConcreteTwoLane = 1,
    DtBuildingOneStory = 7,
    DtBuildingTwoStory = 8,
    DtBuildingThreeStory = 9,
    DtBuildingFourStory = 10,
    DtDragonsTeeth = 20
};

/*****
 * Supply Types
 *****/

enum DtSupplyCategories
{
    DtSupplyOther = 0,
    DtSupplyFuels = 1,
    DtSupplyOils = 2,
    DtSupplyLubricants = 3,
    DtSupplyFood = 4,
    DtSupplySpareParts = 5,
    DtSupplyPersonnel = 6
};

#if DtDISVERS == 3
enum DtSupplyFuels
{
    DtSupplyFuelOther = 0,
    DtGasoline = 1,
    DtDieselFuel = 2,
    DtJP4 = 3,
    DtFuelOil = 4
};
#endif

/*****
 * Used by both Environmental and Cultural features
 *****/
```

```

*****/

enum DtSizeQualifier
{
    DtOther = 0,
    DtVerySmall = 20,
    DtSmall = 40,
    DtMedium = 60,
    DtLarge = 80,
    DtVeryLarge = 100/* You want fries with that? */
};

/*****
 * Some useful macros
 *****/

#define DtSameEntityKind(a,b) \
    ((a)->DtEntityKind == (b)->DtEntityKind && \
    (a)->DtDomain == (b)->DtDomain && \
    (a)->DtCountry == (b)->DtCountry && \
    (a)->DtCategory == (b)->DtCategory && \
    (a)->DtSubCategory == (b)->DtSubCategory && \
    (a)->DtSpecific == (b)->DtSpecific && \
    (a)->DtExtra == (b)->DtExtra)

#define DtIsLandPlatform(a) \
    ((a)->DtEntityKind == DtPlatform && \
    (a)->DtDomain == DtPlatformDomainLand)

#define DtIsLandLifeForm(a) \
    ((a)->DtEntityKind == DtLifeForm && \
    (a)->DtDomain == DtLifeFormDomainLand)

#define DtIsAirPlatform(a) \
    ((a)->DtEntityKind == DtPlatform && \
    (a)->DtDomain == DtPlatformDomainAir)

#define DtIsTank(a) \
    (DtIsLandPlatform(a) && (a)->DtCategory == DtTank)

#if DtDISVERS == 3
#define DtIsRotaryWing(a) \
    (DtIsAirPlatform(a) && \
    (a)->DtCategory > DtAntiSubmarinePatrol && \
    (a)->DtCategory < DtElectronicWarfare)

#define DtIsFixedWing(a) \
    (DtIsAirPlatform(a) && \
    ((a)->DtCategory < DtAttackHelicopter || \
    (a)->DtCategory > DtSeaGoingHelicopter))

#else

#define DtIsRotaryWing(a) \
    (DtIsAirPlatform(a) && \
    (a)->DtCategory >= 20 && \
    (a)->DtCategory < 30)

#define DtIsFixedWing(a) \
    (DtIsAirPlatform(a) && \
    (!DtIsRotaryWing(a)))

#endif

#if DtDISVERS == 3
#define antiPersonell 1
#else

```

```
#define antiPersonell 8
#endif

#define DtEntityIsAntiPersonnel(a) \
((a)->DtEntityKind == DtMunition) && ((a)->DtDomain == antiPersonell)
```


enum2string.h

```
#include "disEnums.h"

const char *DtEnumString(DtPduKind kind);
const char *DtAddEnumString(DtPduKind kind,
    const char *str, int warn = 0);

#define DtADD_ENUM_STRING(k) DtAddEnumString(k, #k + 2)
```

envDec.h

```
#if DtHLA

#include "baseEntDec.h"

// class DtEnvironmentEntityDecoder:
//
// Instances of DtEnvironmentEntityDecoder are used to decode an
// environmental entity state message into a state repository in an HLA
// (as opposed to DIS) environment.

class DtEnvironmentEntityDecoder : public DtBaseEntityDecoder
{
public:

    // constructor
    DtEnvironmentEntityDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtEnvironmentEntityDecoder();

    // copy constructor
    DtEnvironmentEntityDecoder(const DtEnvironmentEntityDecoder& orig);

    // assignment operator
    DtEnvironmentEntityDecoder& operator=(
        const DtEnvironmentEntityDecoder& orig);

public:

    // Create a DtEnvironmentEntityDecoder.
    // User is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
    DtDECLARE_ENTITY_ATTR_DECODER(Density);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_ENVENT_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtEnvironmentEntityDecoder, DtEntityStateRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_ENVENT_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ENVENT_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif
```

envEnc.h

```

#if DtHLA

#include "baseEntEnc.h"
#include "entitySR.h"

// class DtEnvironmentEntityEncoder:
//
// Instances of DtEnvironmentEntityEncoder are used to encode an
// environment entity's entity's state repository into a state message
// in an HLA environment.

class DtEnvironmentEntityEncoder : public DtBaseEntityEncoder
{
public:
    // constructor
    DtEnvironmentEntityEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtEnvironmentEntityEncoder();

    // copy constructor
    DtEnvironmentEntityEncoder(const DtEnvironmentEntityEncoder& orig);

    // assignment operator
    DtEnvironmentEntityEncoder& operator=(
        const DtEnvironmentEntityEncoder& orig);

public:
    // Create a DtEnvironmentEntityEncoder.
    // Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // static need<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_CHECKER(Density);

    // static decode<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_ENCODER(Density);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_ENVENT_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtEnvironmentEntityEncoder, \
        DtEntityStateRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that checker whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.

```

```
#define DtDEFINE_SIMPLE_ENVENT_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtEnvironmentEntityEncoder, \
        DtEntityStateRepository, \
        attrName, inspector)

#endif
```

enviroTypeRec.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"

// class DtEnvironmentTypeRecord:
//
// Instances of DtEnvironmentTypeRecord are used by the
// DtGriddedDataPdu for specifying environmental type.

class DtEnvironmentTypeRecord : public DtEntityType
{
public:
    // default constructor
    DtEnvironmentTypeRecord();

    // destructor
    virtual ~DtEnvironmentTypeRecord();

    // copy constructor
    DtEnvironmentTypeRecord(const DtEnvironmentTypeRecord& orig);

    // assignment operators
    DtEnvironmentTypeRecord& operator=(const DtEnvironmentTypeRecord& orig);
    DtEnvironmentTypeRecord& operator=(const DtNetEntityType& orig);

    // Set/Get the environmental class
    virtual void setClassOfEnv(int cls);
    virtual int classOfEnv() const;

protected:
    // Get/Set country. This field is used for the environmental class.
    virtual void setCountry(int country);
    virtual int country() const;
};
#endif
#endif
```

envProcPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"

#include "packets/envProcP.h"

// class DtEnvironmentalProcessPdu:
//
// Instances of DtEnvironmentalProcessPdu are used to send information about
// environmental effects and processes.

// DtEnvironmentalProcessPdu callback function signature.
typedef void (*DtEnvironmentalProcessPduCb)(
    DtEnvironmentalProcessPdu* pdu, void* usr);

class DtEnvironmentalProcessPdu : public DtPdu
{
public:
    // Constructors
    DtEnvironmentalProcessPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEnvironmentalProcessPdu(const DtNetEnvironmentalProcessPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEnvironmentalProcessPdu(const DtEnvironmentalProcessPdu& orig);

    // destructor
    virtual ~DtEnvironmentalProcessPdu();

    // Assignment operator
    DtEnvironmentalProcessPdu& operator=(const DtEnvironmentalProcessPdu&
orig);

    // Set/Get originating environmental process ID
    virtual void setEnvironmentalProcessID(const DtEntityIdentifier&
entity);
    virtual const DtEntityIdentifier& environmentalProcessID() const;

    // Set/Get environment type
    virtual void setEnvironmentType(const DtEntityType& type);
    virtual const DtEntityType& environmentType() const;

    // Set/Get the model type
    virtual void setModelType(DtEnvironmentalModelType value);
    virtual DtEnvironmentalModelType modelType() const;

    // Set/Get the environmental status (as an 8-bit value)
    virtual void setEnvironmentalStatus(DtU8 value);
    virtual DtU8 environmentalStatus() const;

    // Set/Get an environmental status bit
    virtual void setEnvironmentalStatusBit(
        DtEnvironmentalStatusBit bit, DtBoolean value);
    virtual DtBoolean environmentalStatusBit(
        DtEnvironmentalStatusBit bit) const;

    // Set/Get number of environment records
    virtual DtBoolean setNumberOfEnvironmentalRecords(DtU16 newNumberOf-
Records);
```

```

virtual DtU16 numberOfEnvironmentalRecords() const;

// Set/Get sequence number
virtual void setSequenceNumber(DtU16 value);
virtual DtU16 sequenceNumber() const;

// Set the size of an environmental record (in bits). Returns DtFalse if
// the size is invalid.
virtual DtBoolean setRecordBitSize(int recordIndex, int newRecordBit-
Size);

// Get a pointer to an environmental record. Returns DtFalse if the
record
// index is out of range.
virtual DtNetEnvironmentalRecord* recordPointer(int recordIndex);

// Print the data contained by this PDU
virtual void printData() const;

public:

// Create a DtEnvironmentalProcessPdu.
// Caller is responsible for deletion.
static DtPdu* create(const DtNetPacket* initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
DtEnvironmentalProcessPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
DtEnvironmentalProcessPduCb cb, void* usr);

protected:

// Returns the internal PDU kind
virtual DtPduKind internalGetPduKind() const;

// Casts the netPdu element to our network struct
// Non-virtual since you can't override based on return type
DtNetEnvironmentalProcessPdu* netPduStruct() const;

// Casts the 'this' pointer for changing elements in a const function
// Non virtual since you can't override based on return type
DtEnvironmentalProcessPdu* self() const;

// Returns a pointer to the first byte of variable length data
virtual char* var() const;

// Returns an offset in bytes from the beginning of the variable length
// data region to the beginning of a particular record.
virtual int recordOffset(int recordIndex) const;

protected:

// Size of PDU with zero data bytes.
static int theStaticSize;

protected:

// Temporary objects for reference returns
DtEntityIdentifier myEntityID;
DtEntityType myEntityType;

};
#endif
#endif

```

esPdu.h

```
#if !DtHLA

#include "hostStructs.h"
#include "pdu.h"
#include "packets/entityStateP.h"
#include "artPartList.h"
#include "mathTypes.h"
#include "disEnums.h"
#include "stateMsg.h"

#include "chooseDefines.h"

#define DtEntityStateStaticSize 144
#define DtINCREMENT -1

#define DtNO_SUCH_ART_PART 0
#define DtART_PART_SUCCESS 1

class DtEntityStatePdu;
typedef void (*DtEntityStatePduCb)(DtEntityStatePdu *pdu, void *usr);

class DtEntityStatePdu : public DtStateMsg
{
public:
/***** STANDARD CONSTRUCTORS *****/

    DtEntityStatePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEntityStatePdu(const DtNetEntityStatePdu *initial,
                    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtEntityStatePduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtEntityStatePduCb cb, void *usr);

/***** ADDITIONAL CONSTRUCTORS *****/
    DtEntityStatePdu(const DtEntityIdentifier &id,
const DtEntityType &type,
const DtEntityType &guise,
DtDeadReckonTypes algorithm,
DtForceType forceId,
const DtArtPartSpec *specList,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEntityStatePdu(const DtEntityIdentifier &id,
const DtEntityType &type,
const DtEntityType &guise,
DtDeadReckonTypes algorithm,
DtForceType forceId,
const DtArtPartSpec *artList,
int *stationList,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEntityStatePdu(const DtArtPartSpec *specList,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/
    const DtEntityIdentifier &id() const;
    DtForceType forceId() const;
    int numArtParams() const;
    int numAttachedParams() const;
    int numArtPlusAttachedParams() const;
    const DtEntityType &entityType() const;
```



```

    const DtEntityType      &guise() const;
    DtConstVector           velocity() const;
    DtConstVector           location() const;
    const DtTaitBryan       &orientation() const;
    DtU32                   appearance() const;
    DtAppearAttrVal
appearanceAttribute(const DtAppearAttribute &attr) const;
    DtDeadReckonTypes      algorithm() const;
    DtConstVector          acceleration() const;
    DtConstVector          rotationalVelocity() const;
    int                    markingCharSet() const;
    const char             *markingText() const;
    DtU32                  capabilities() const;
    const DtArtPartSpec    *artPartSpecs() const;

    // returns a DtArtPartList containing the parts described in this PDU.
    const DtArtPartList&   artPartList() const;

    int                    *attachedPartStations() const;
    int                    getArtParamVal(int paramIndex, DtFloat32 *value) const;
    int                    getArtParamVal(int partType, int param, DtFloat32 *value)
const;
    int                    getArtParamChange(int paramIndex, int *change) const;
    int                    getArtParamChange(int partType, int param, int *change) const;
    int                    attachedPartChange(int station) const;
    int                    getAttachedType(int station, DtEntityType *type) const;

/***** MUTATOR FUNCTIONS *****/

    void                    setId(const DtEntityIdentifier &id);
    void                    setForceId(DtForceType force);
    void                    setEntityType(const DtEntityType &type);
    void                    setGuise(const DtEntityType &type);
    void                    setVelocity(DtConstVector Velocity);
    void                    setLocation(DtConstVector Location);
    void                    setOrientation(const DtTaitBryan &Orient);
    void                    setAppearance(DtU32 app);
    void                    setAppearanceAttribute(const DtAppearAttribute attr,
const DtAppearAttrVal val);
    void                    setAlgorithm(DtDeadReckonTypes alg);
    void                    setAcceleration(DtConstVector Acceleration);
    void                    setRotationalVelocity(DtConstVector BodyRates);
    void                    setMarkingText(const char *text);
    void                    setMarkingCharSet(int charSet);
    void                    setCapabilities(DtU32 cap);
    int                    setArtParamVal(int partType, int param, DtFloat32 value,
int change = DtINCREMENT);
    int                    setArtParamVal(int paramIndex, DtFloat32 value,
int change = DtINCREMENT);
    void                    initArtAndAttachedParts(const DtArtPartSpec *artList,
const int* stationList);
    void                    initArtAndAttachedParts(const DtArtPartList &apList,
const int* stationList);
    void                    initArtParts(const DtArtPartList &apList);
    int                    setAttachedPartChange(int station, int change);
    int                    setAttachedPartType(int station, const DtEntityType &type);

    void                    printData() const;
protected:
    DtPduKind               internalGetPduKind() const {return DtEntityStatePduKind;}
    void                    setNumArtPlusAttachedParams(int num);
    DtNetArtPartRecord      *artPartRecord(int partType, int param) const;
    DtNetArtPartRecord      *attachedPartRecord(int station) const;
    void                    setArtParamVal(DtNetArtPartRecord *part, DtFloat32 value,
int change);
    float                   getArtParamVal(DtNetArtPartRecord *part) const;
    DtEntityIdentifier       myId;

```

```
DtEntityType type, theGuise;  
DtVector vel, loc, acc;  
DtTaitBryan ori;  
DtBodyRates rvel;  
char markingBuff[DtMaxEntityMarkingLength+1];  
  
// Does not store state.  
DtArtPartList myArtPartList;  
};  
#endif
```

esUpdatePdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"
#include "artPartList.h"
#include "attPartList.h"

#include "packets/esUpdateP.h"

// class DtEntityStateUpdatePdu:
//
// Instances of DtEntityStateUpdatePdu are used to send non-static informa-
// tion
// about a particular entity.

// DtEntityStateUpdatePdu callback function signature.
typedef void (*DtEntityStateUpdatePduCb)(
    DtEntityStateUpdatePdu* pdu, void* usr);

class DtEntityStateUpdatePdu : public DtPdu
{
public:
    // Constructors
    DtEntityStateUpdatePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEntityStateUpdatePdu(const DtNetEntityStateUpdatePdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEntityStateUpdatePdu(const DtEntityStateUpdatePdu& orig);

    // destructor
    virtual ~DtEntityStateUpdatePdu();

    // Create a DtEntityStateUpdatePdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtEntityStateUpdatePduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtEntityStateUpdatePduCb cb, void* usr);

    // Assignment operator
    DtEntityStateUpdatePdu& operator=(const DtEntityStateUpdatePdu& orig);

    // Set/Get entity ID
    virtual void setEntityID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& entityID() const;

    // Get the number of articulation parameter records (this field is
    // automatically set by the PDU when the articulated part list or
    attached
    // part list is set)
    virtual DtU8 numRecords() const;

    // Set/Get velocity
    virtual void setVelocity(DtConstVector velocity);

```

```
virtual DtConstVector velocity() const;

// Set/Get location
virtual void setLocation(DtConstVector coords);
virtual DtConstVector location() const;

// Set/Get orientation
virtual void setOrientation(const DtTaitBryan& orientation);
virtual const DtTaitBryan& orientation() const;

// Set/Get appearance (as a 32 bit integer)
virtual void setAppearance(DtU32 app);
virtual DtU32 appearance() const;

// Set/Get appearance by attribute
virtual void setAppearanceAttribute(
    const DtAppearAttribute attr, const DtAppearAttrVal val);
virtual DtAppearAttrVal appearanceAttribute(
    const DtAppearAttribute attr) const;

// The articulated parts and attached parts information are both
optional.
// Either or both can be set.

// Set/Get articulated parts parameters.
// Parameters are passed in the form of an articulated parts list.
// Returns DtFalse if the operation could not be performed.
virtual DtBoolean setArtPartParameters(const DtArtPartList& partList);
virtual DtBoolean getArtPartParameters(DtArtPartList& partList) const;

// Set/Get attached parts parameters.
// Parameters are passed in the form of an attached parts list.
// Returns DtFalse if the operation could not be performed.
virtual DtBoolean setAttPartParameters(const DtAttachedPartList&
partList);
virtual DtBoolean getAttPartParameters(DtAttachedPartList& partList)
const;

// Print the data contained by this PDU
virtual void printData() const;

protected:

// Returns the internal PDU kind
virtual DtPduKind internalGetPduKind() const;

// Casts the netPdu element to our network struct
virtual DtNetEntityStateUpdatePdu* netPduStruct() const;

// Casts the 'this' pointer for changing elements in a const function
// Non-virtual since you can't override based on return type
DtEntityStateUpdatePdu* self() const;

// Returns a pointer to the first byte of variable length data
virtual char* var() const;

// Set number of records
virtual void setNumRecords(DtU8 value);

// Returns the offset (in bytes) of a field in the variable data region.
virtual int artParmOffset() const;
virtual int attParmOffset() const;

protected:

// Size of PDU with zero parameter records.
static int theStaticSize;
```

```
protected:

    // Number of articulated part parameters for this PDU
    int myNumArtPartParams;

    // Number of attached part parameters for this PDU
    int myNumAttPartParams;

    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
    DtVector myVector;
    DtTaitBryan myEulerAngles;
};
#endif
#endif
```

Euler.h

```
#include "LibMatrix.h"

/* Vrlink (and DIS) Euler angles are defined as sucessive right-hand
 * rotations about
 * reference Z axis heading or psi
 * the rotated Y axispitch or theta
 * body X axisroll or phi
 */
* This is the standard sequence of rotations for vehicle design and
* simulation, and is known to some as the Tait-Bryan sequence.
*
* The body axis system for DIS is defined by
*X forward
*Y right
*Z down
*
* The reference axis system is defined by some "fixed" or "world"
* frame. When topographic-down is chosen as the reference frame,
* you get heading relative to north, and pitch and roll relative to
* the local horizon. See topoCoord.h.
*/

// Given Euler angles between a body and a reference frame, produce
// the transformation matrix that converts from the body frame to the
// reference frame (or visa-versa).
void DtEuler_to_BodyToRef(const DtTaitBryan &euler, DtDcmRef body2ref);
void DtEuler_to_RefToBody(const DtTaitBryan &euler, DtDcmRef ref2body);

// Given the transformation matrix that converts from a body frame to
// a reference frame (or visa-versa), produce the Euler angles between
// the body and the reference frames.
void DtBodyToRef_to_Euler(DtConstDcm body2ref, DtTaitBryan *euler);
void DtRefToBody_to_Euler(DtConstDcm ref2body, DtTaitBryan *euler);

// Given Euler angles with respect to reference frame "in", and the
// transformation matrix between frame "in" and frame "out", produce
// the Euler angles with repect to reference frame "out".
void DtEulerToEuler(const DtTaitBryan &in,
    DtConstDcm in2out,
    DtTaitBryan *out);
```

eventId.h

```

#include "NetStructs.h"

#if DtHLA

class DtEventId
{
public:
    // Construct a DtEventId from a character string consisting of
    // the event number followed by a colon, and then an object name.
    // For example: DtEventId event("36:Object1");
    DtEventId(const char* str);

    // Construct a DtEventId from a name and an event number.
    DtEventId(const DtGlobalObjectDesignator& des, int eventNum);

    // Construct a DtEventId from network representation.
    DtEventId(const DtNetEventId& netId);

    // Same as DtEventId("", 0);
    DtEventId();

    // Set from network representation/
    void setFromNet(const DtNetEventId& netId);

    // Convert to network representation. Returns ptr to internal memory.
    DtNetEventId* netRep() const;

    // equivalence operators.
    int operator==(const DtEventId &otherGuy) const;
    int operator!=(const DtEventId &otherGuy) const;

    // Set/Get event number component
    void setEventNum(int eventNum);
    int eventNum() const;

    // Set/Get object designator component
    const DtGlobalObjectDesignator& objectId() const;
    void setObjectId(const DtGlobalObjectDesignator& id);

    // Convert to string.
    char *string() const;

public:
    // Return a null DtEventId.
    static const DtEventId& nullId();

protected:
    // initialize
    void init(const DtGlobalObjectDesignator& des, int eventNum);

protected:
    DtString myObjName;
    int myEventNum;
};
#else
class DtEventId
{
public:
    DtEventId(const char* str);

```

```
DtEventId(int site, int host, int eventNum);
DtEventId(DtNetEventId netId);
DtEventId();

void init(int site, int host, int eventNum);

void setFromNet(DtNetEventId netId);
operator DtNetEventId () const;

int site() const;
int host() const;
int eventNum() const;

void setSite(int site);
void setHost(int host);
void setEventNum(int eventNum);

char *string() const;

int operator==(const DtEventId &otherGuy) const;
int operator!=(const DtEventId &otherGuy) const;

static const DtEventId& nullId();

public:
    // Public for backwards compatibility.
    DtSimulationAddress DtSimulator;
    DtU16 DtEvent;
};

#endif

typedef DtEventId DtEventID;
```


eventReptPdu.h

```

#if !DtHLA

#include "hostStructs.h"
#include "pduWithData.h"
#define DtEventReportStaticSize 40

class DtEventReportPdu;
typedef void (*DtEventReportPduCb)(DtEventReportPdu *pdu, void *usr);

class DtEventReportPdu : public DtPduWithData
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtEventReportPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEventReportPdu(const DtNetEventReportPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtEventReportPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtEventReportPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    DtEventType eventType() const;

/***** MUTATOR FUNCTIONS *****/

    void setEventType(DtEventType type);

    void printData() const;

protected:

    DtPduKind internalGetPduKind() const {return DtEventReportPduKind;}
    DtNetDataPduTail *dataSection() const;
};
#endif

```

eventReptRPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "pduWithData.h"
#include "exerciseConn.h"

// class DtEventReptRPdu:
//
// Instances of DtEventReptRPdu are used to send/receive
// reliable event report PDUs.

// DtEventReptRPdu callback function signature.
typedef void (*DtEventReptRPduCb)(DtEventReptRPdu* pdu, void* usr);

class DtEventReptRPdu : public DtPduWithData
{
public:

    // Constructors
    DtEventReptRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEventReptRPdu(const DtNetEventReportPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtEventReptRPdu(const DtEventReptRPdu& orig);

    // destructor
    virtual ~DtEventReptRPdu();

    // Assignment operator
    DtEventReptRPdu& operator=(const DtEventReptRPdu& orig);

    // Set/Get event type.
    virtual void setEventType(DtEventType type);
    virtual DtEventType eventType() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:

    // Create a DtEventReptRPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtEventReptRPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtEventReptRPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Returns the data portion of the net structure.
    virtual DtNetDataPduTail* dataSection() const;

protected:

    static int theStaticSize;
```

```
};  
#endif  
#endif
```

filename.h

```
#include "str.h"

// class DtFilename:
//
// Instances of DtFilename are DtStrings which convert filenames to the
// native format.
class DtFilename : public DtString
{
public:
    // default constructor
    DtFilename();

    // constructor
    DtFilename(const char* str);

    // destructor
    virtual ~DtFilename();

    // copy constructor
    DtFilename(const DtFilename& orig);

    // assignment operator
    DtFilename& operator=(const DtFilename& orig);

protected:
    // Replaces slashes. On unix, this replaces backslashes with forward
    // slashes. Vice versa on win32. Hail Mr Gates.
    // This is not virtual because we call this from the constructors and
    // destructor.
    void replaceSlashes();
};
```

firePdu.h

```

#ifndef DtHLA

#include "hostStructs.h"
#include "interaction.h"
#include "packets/fireP.h"
#include "mathTypes.h"
#define DtFireStaticSize 96

#include "chooseDefines.h"

class DtFirePdu;
typedef DtFirePdu DtFireInteraction;
typedef void (*DtFireInteractionCb)(DtFireInteraction *inter, void *usr);

class DtFirePdu : public DtInteraction
{
public:

    /***** STANDARD CONSTRUCTORS *****/

    DtFirePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtFirePdu(const DtNetFirePdu *initial,
              DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static DtPdu *create(const DtNetPacket *initial,
                        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static void addCallback(DtExerciseConn *conn,
                          DtFireInteractionCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
                              DtFireInteractionCb cb, void *usr);

    /***** ADDITIONAL CONSTRUCTORS *****/
    DtFirePdu(const DtEntityIdentifier &attacker,
              const DtEntityIdentifier &target,
              const DtEntityIdentifier &munit,
              const DtEventID &eventId,
              DtConstVector location,
              const DtBurstDescriptor &burst,
              DtConstVector linVelocity,
              const double range,
              DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    /***** INSPECTOR FUNCTIONS *****/

    const DtObjectId &attackerId() const;
    const DtObjectId &targetId() const;
    const DtObjectId &munitId() const;
    const DtEventID &eventId() const;
    #if DtDISVERS >= 5
        unsigned long fireMissionIndex() const;
    #endif
    DtConstVector location() const;
    DtConstVector velocity() const;
    double range() const;
    DtDetonatorFuze fuseType() const;
    const DtEntityType &munitType() const;
    int quantity() const;
    int rate() const;
    DtWarheadType warheadType() const;

    // NOT USED IN HLA
    const DtBurstDescriptor &burst() const;

```

```
/****** MUTATOR FUNCTIONS *****/

void setAttackerId(const DtObjectId &id);
void setTargetId(const DtObjectId &id);
void setMunitionId(const DtObjectId &id);
void setEventId(const DtEventID &id);
#if DtDISVERS >= 5
void setFireMissionIndex(unsigned long i);
#endif
void setLocation(DtConstVector location);
void setVelocity(DtConstVector velocity);
void setRange(double range);
void setFuseType(DtDetonatorFuze f);
void setMunitionType(const DtEntityType &t);
void setQuantity(int q);
void setRate(int r);
void setWarheadType(DtWarheadType t);

// NOT USED IN HLA
void setBurst(const DtBurstDescriptor &burst);

void printData() const;

protected:
DtPduKind internalGetPduKind() const {return DtFirePduKind;}
DtObjectId att;
DtObjectId tar;
DtObjectId mun;
DtEventID event;
DtVector loc;
DtVector vel;
DtBurstDescriptor theBurst;
DtEntityType myMunType;
};
#endif
```

fom.h

```

#if DtHLA

#include "chooseDefines.h"

#include "fedTarget.h"
#include "RTI.hh"

// #include "chooseDefines.h"

// class DtFom:
//
// An instances of DtFom is used to represent a particular FOM, as described
// by a FED file. Object classes are represented by DtObjClassDescs;
// Interaction classes by DtInterClassDescs.
// Being a DtFedTarget, a DtFom can be filled in by
// a DtFedReader. Alternatively, you can fill it in programatically,
// with addObjClass, addInterClass, addParameter and addAttribute. Once
// so initialized, you can query for a class descriptor associated with a
// class.

class DtFom : public DtFedTarget
{
public:

    // Constructor
    DtFom(DtExerciseConn* exConn);

    // Destructor
    virtual ~DtFom();

    // copy constructor
    DtFom(const DtFom& orig);

    // assignment operator
    DtFom& operator=(const DtFom& orig);

    // Define a class. Used when initializing from FED file.
    virtual void addInterClass(const char* name,
        const char* baseName);

    // Define a class. Used when initializing from FED file.
    virtual void addObjClass(const char* name,
        const char* baseName);

    // Define an parameter of a class.
    // Used when initializing from FED file.
    virtual void addParameter(const char* className,
        const char* paramName);

    // Define an parameter of a class.
    // Used when initializing from FED file.
    virtual void addAttribute(const char* className,
        const char* paramName);

    // Returns the class descriptor for the name
    virtual DtInterClassDesc* interClassByName(const char *name) const;

    // Returns the class descriptor for the name
    virtual DtObjClassDesc* objClassByName(const char *name) const;

    // Returns the class descriptor for the handle
    virtual DtInterClassDesc* interClassByHandle(
        RTI::InteractionClassHandle handle) const;

```

```
// Returns the class descriptor for the handle
virtual DtObjClassDesc* objClassByHandle(
    RTI::ObjectClassHandle handle) const;

// Returns the first, last and next object classes in the fom.
// Next returns NULL if current is the last class, or if current is NULL.
// These functions facilitate looping through classes like this:
// for (desc = firstObjClass; desc; desc = nextObjClass(desc))
virtual DtObjClassDesc* firstObjClass() const;
virtual DtObjClassDesc* lastObjClass() const;
virtual DtObjClassDesc* nextObjClass(DtObjClassDesc* current) const;

// Returns the first, last and next interaction class in the fom.
// Next returns NULL if current is the last class, or if current is NULL.
// These functions facilitate looping through classes like this:
// for (desc = firstInterClass; desc; desc = nextInterClass(desc))
virtual DtInterClassDesc* firstInterClass() const;
virtual DtInterClassDesc* lastInterClass() const;
virtual DtInterClassDesc* nextInterClass(DtInterClassDesc* current)
const;

// Returns our RTI ambassador
virtual DtVrIRtiAmbassador* rtiAmb();

// prints the contents of the FOM
virtual void print() const;

public:

    // Set/Get max number of object or interaction classes
    static void setMaxClasses(int num);
    static int maxClasses();

protected:

    DtExerciseConn* myExConn;
    DtInterClassDesc** myInterClasses;
    DtObjClassDesc** myObjClasses;

    int myLowestObj;
    int myHighestObj;
    int myLowestInter;
    int myHighestInter;

#if !DtrTI103
    char* myObjRootName;
    char* myInterRootName;
#endif

protected:

    static int theMaxClasses;
};
#include "exerciseConn.h"

#include "chooseDefines.h"

#endif
```


fomMapper.h

```

#if DtHLA

#include "RTI.hh"
#include "hostStructs.h"
#include "vllist.h"

#include "chooseDefines.h"

// class DtFomMapper:
//
// Instances of DtFomMapper are used to provide FOM agility in VR-Link
// by mapping VR-Link classes and functionality to FOM classes,
// attributes and parameters.

class DtFomMapper
{
public:
    // A function that chooses an object class from the FOM for
    // an entity.
    typedef char* (*DtEntityClassChooser)(
        DtExerciseConn* exConn, const DtEntityType& type);

    // A function that chooses an object class from the FOM
    // for an aggregate.
    typedef char* (*DtAggregateClassChooser)(
        DtExerciseConn* exConn, const DtEntityType& type);

    // A function that chooses an interaction class from the FOM based on
    // the contents of a derived DtInteraction object.
    typedef char* (*DtInteractionClassChooser)(
        DtExerciseConn* exConn, const DtInteraction& inter);

public:
    // default constructor
    DtFomMapper(DtExerciseConn* exConn);

    // destructor
    virtual ~DtFomMapper();

    // copy constructor
    DtFomMapper(const DtFomMapper& orig);

    // assignment operator
    DtFomMapper& operator=(const DtFomMapper& orig);

    // Set/Get the current factory that can be used to create
    // DtHlaStateDecoders
    virtual DtStateDecoderFactory* setStateDecoderFactory(
        DtStateDecoderFactory* fact);
    virtual DtStateDecoderFactory* stateDecoderFactory();

    // Set/Get the current factory that can be used to create
    // DtHlaStateEncoders
    virtual DtStateEncoderFactory* setStateEncoderFactory(
        DtStateEncoderFactory* fact);
    virtual DtStateEncoderFactory* stateEncoderFactory();

    // Set/Get the current factory that can be used to create
    // DtInteractionDecoders
    virtual DtInteractionDecoderFactory* setInteractionDecoderFactory(
        DtInteractionDecoderFactory* fact);

```

```
virtual DtInteractionDecoderFactory* interactionDecoderFactory();

// Set/Get the current factory that can be used to create
// DtInteractionEncoders
virtual DtInteractionEncoderFactory* setInteractionEncoderFactory(
    DtInteractionEncoderFactory* fact);
virtual DtInteractionEncoderFactory* interactionEncoderFactory();

// Set/Get the current factory that can be used to create
// DtHlaObjects
virtual DtHlaObjectFactory* setHlaObjectFactory(
    DtHlaObjectFactory* fact);
virtual DtHlaObjectFactory* hlaObjectFactory();

// Set/Get the current factory that can be used to create
// DtInteractions
virtual DtInteractionFactory* setInteractionFactory(
    DtInteractionFactory* fact);
virtual DtInteractionFactory* interactionFactory();

// Set the current object class choosing function for entities
// Returns the previously registered function.
virtual DtEntityClassChooser setEntityClassChooser(
    DtEntityClassChooser chooser);

// Chooses an object class to use when simulating an entity of
// type type.
virtual char* chooseEntityObjectClass(
    const DtEntityType& type) const;

// Set the current object class choosing function for aggregates
// Returns the previously registered function.
virtual DtAggregateClassChooser setAggregateClassChooser(
    DtAggregateClassChooser chooser);

// Chooses an object class to use when simulating an aggregate of
// type type.
virtual char* chooseAggregateObjectClass(
    const DtEntityType& type) const;

// Set the current interaction class choosing function.
// Returns the previously registered function.
virtual DtInteractionClassChooser setInteractionClassChooser(
    DtInteractionClassChooser chooser);

// Chooses an interaction class to use when sending an interaction.
virtual char* chooseInteractionClass(
    const DtInteraction& inter) const;

// Set/Get a list of RTI::ObjectClassHandles indicating which
// FOM classes should be interpreted as entities (i.e. subscribed
// to by the REL.
virtual void setEntityObjectClasses(DtList* classes);
virtual DtList* entityObjectClasses();

// Set/Get a list of RTI::ObjectClassHandles indicating which
// FOM classes should be interpreted as aggregates (i.e. subscribed
// to by the RAL.
virtual void setAggregateObjectClasses(DtList* classes);
virtual DtList* aggregateObjectClasses();

// Set/Get a list of RTI::InteractionClassHandles that can be decoded
// into the derived DtInteraction class whose name is name. These
// classes will be subscribed to, when you use a derived DtInteraction's
// addCallback function.
virtual void setInteractionClasses(const char* vlinkClassName,
    DtList* classes);
```

```
    virtual DtList* interactionClasses(const char* vrlinkClassName);

public:
    // A Creator function that returns a DtFomMapper configured with
    // no mappings at all.
    static DtFomMapper* create(DtExerciseConn* exConn);

protected:
    DtExerciseConn* myExConn;

    // Current factories and other objects
    DtStateDecoderFactory* myDecFact;
    DtStateEncoderFactory* myEncFact;
    DtInteractionDecoderFactory* myInterDecFact;
    DtInteractionEncoderFactory* myInterEncFact;
    DtHlaObjectFactory* myObjFact;
    DtInteractionFactory* myInterFact;

    // Lists of RTI::ObjectClassHandles.
    DtList* myEntityClassList;
    DtList* myAggClassList;

    // List of DtInterClassMappings
    DtList myInterClasses;

    // These are just function pointers.
    DtEntityClassChooser myEntityChooser;
    DtAggregateClassChooser myAggChooser;
    DtInteractionClassChooser myInterChooser;
};

#endif
```

gedBasFixWing.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/isGroupOfP.h"

// class DtBasicFixedWingAircraftRecord:
//
// Instances of DtBasicFixedWingAircraftRecord are used to contain the
// entity state and logistics information about an individual entity within
// a group of entities.

class DtBasicFixedWingAircraftRecord
{
public:
    // Constructors
    DtBasicFixedWingAircraftRecord();
    DtBasicFixedWingAircraftRecord(
        const DtBasicFixedWingAircraftRecord& orig);

    // Destructor
    virtual ~DtBasicFixedWingAircraftRecord();

    // Assignment operators
    DtBasicFixedWingAircraftRecord& operator=(
        const DtBasicFixedWingAircraftRecord& orig);
    DtBasicFixedWingAircraftRecord& operator=(
        const DtNetBasicFixedWingAircraftRecord& orig);

    // Set/Get the entity ID
    virtual void setEntityID(DtU16 value);
    virtual DtU16 entityID() const;

    // Set/Get the entity location X offset
    virtual void setX_Offset(DtInt16 value);
    virtual DtInt16 X_Offset() const;

    // Set/Get the entity location Y offset
    virtual void setY_Offset(DtInt16 value);
    virtual DtInt16 Y_Offset() const;

    // Set/Get the entity location Z offset
    virtual void setZ_Offset(DtInt16 value);
    virtual DtInt16 Z_Offset() const;

    // Set/Get appearance (as a 32 bit integer)
    virtual void setAppearance(DtU32 app);
    virtual DtU32 appearance() const;

    // Set/Get appearance by attribute
    virtual void setAppearanceAttribute(
        const DtAppearAttribute attr, const DtAppearAttrVal val);
    virtual DtAppearAttrVal appearanceAttribute(
        const DtAppearAttribute attr) const;

    // Set/Get the entity orientation psi
    virtual void setPsi(DtInt8 value);
    virtual DtInt8 psi() const;
```

```

// Set/Get the entity orientation theta
virtual void setTheta(DtInt8 value);
virtual DtInt8 theta() const;

// Set/Get the entity orientation phi
virtual void setPhi(DtInt8 value);
virtual DtInt8 phi() const;

// Set/Get the fuel status
virtual void setFuelStatus(DtInt8 value);
virtual DtInt8 fuelStatus() const;

// Set/Get the horizontal deviation
virtual void setHorizontalDeviation(DtInt8 value);
virtual DtInt8 horizontalDeviation() const;

// Set/Get the vertical deviation
virtual void setVerticalDeviation(DtInt8 value);
virtual DtInt8 verticalDeviation() const;

// Set/Get the movement speed
virtual void setMovementSpeed(DtInt16 value);
virtual DtInt16 movementSpeed() const;

// Print this record
virtual void print() const;

// Load/Unload data from a packet record
virtual void unloadBaseNetRecord(
    const DtNetBasicFixedWingAircraftRecord& record);
virtual void loadBaseNetRecord(
    DtNetBasicFixedWingAircraftRecord& record) const;

protected:

DtU16      myEntityID;
DtInt16    myX_Offset;
DtInt16    myY_Offset;
DtInt16    myZ_Offset;
DtU32      myAppearance;
DtInt8     myPsi;
DtInt8     myTheta;
DtInt8     myPhi;
DtU8       myFuelStatus;
DtU8       myHorizontalDeviation;
DtU8       myVerticalDeviation;
DtInt16    myMovementSpeed;

};
#endif
#endif

```

gedBasRotWing.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/isGroupOfP.h"

// class DtBasicRotorWingAircraftRecord:
//
// Instances of DtBasicRotorWingAircraftRecord are used to contain the
// entity state and logistics information about an individual entity within
// a group of entities.

class DtBasicRotorWingAircraftRecord
{
public:
    // Constructors
    DtBasicRotorWingAircraftRecord();
    DtBasicRotorWingAircraftRecord(
        const DtBasicRotorWingAircraftRecord& orig);

    // Destructor
    virtual ~DtBasicRotorWingAircraftRecord();

    // Assignment operators
    DtBasicRotorWingAircraftRecord& operator=(
        const DtBasicRotorWingAircraftRecord& orig);
    DtBasicRotorWingAircraftRecord& operator=(
        const DtNetBasicRotorWingAircraftRecord& orig);

    // Set/Get the entity ID
    virtual void setEntityID(DtU16 value);
    virtual DtU16 entityID() const;

    // Set/Get the entity location X offset
    virtual void setX_Offset(DtInt16 value);
    virtual DtInt16 X_Offset() const;

    // Set/Get the entity location Y offset
    virtual void setY_Offset(DtInt16 value);
    virtual DtInt16 Y_Offset() const;

    // Set/Get the entity location Z offset
    virtual void setZ_Offset(DtInt16 value);
    virtual DtInt16 Z_Offset() const;

    // Set/Get appearance (as a 32 bit integer)
    virtual void setAppearance(DtU32 app);
    virtual DtU32 appearance() const;

    // Set/Get appearance by attribute
    virtual void setAppearanceAttribute(
        const DtAppearAttribute attr, const DtAppearAttrVal val);
    virtual DtAppearAttrVal appearanceAttribute(
        const DtAppearAttribute attr) const;

    // Set/Get the entity orientation psi
    virtual void setPsi(DtInt8 value);
    virtual DtInt8 psi() const;
```

```

// Set/Get the entity orientation theta
virtual void setTheta(DtInt8 value);
virtual DtInt8 theta() const;

// Set/Get the entity orientation phi
virtual void setPhi(DtInt8 value);
virtual DtInt8 phi() const;

// Set/Get the fuel status
virtual void setFuelStatus(DtInt8 value);
virtual DtInt8 fuelStatus() const;

// Set/Get the horizontal deviation
virtual void setHorizontalDeviation(DtInt8 value);
virtual DtInt8 horizontalDeviation() const;

// Set/Get the vertical deviation
virtual void setVerticalDeviation(DtInt8 value);
virtual DtInt8 verticalDeviation() const;

// Set/Get the movement speed
virtual void setMovementSpeed(DtInt16 value);
virtual DtInt16 movementSpeed() const;

// Set/Get the turret azimuth
virtual void setTurretAzimuth(DtInt8 value);
virtual DtInt8 turretAzimuth() const;

// Set/Get the gun elevation
virtual void setGunElevation(DtInt8 value);
virtual DtInt8 gunElevation() const;

// Set/Get the turret slew rate
virtual void setTurretSlewRate(DtInt8 value);
virtual DtInt8 turretSlewRate() const;

// Set/Get the gun elevation rate
virtual void setGunElevationRate(DtInt8 value);
virtual DtInt8 gunElevationRate() const;

// Print this record
virtual void print() const;

// Load/Unload data from a packet record
virtual void unloadBaseNetRecord(
    const DtNetBasicRotorWingAircraftRecord& record);
virtual void loadBaseNetRecord(
    DtNetBasicRotorWingAircraftRecord& record) const;

```

protected:

```

DtU16    myEntityID;
DtInt16  myX_Offset;
DtInt16  myY_Offset;
DtInt16  myZ_Offset;
DtU32    myAppearance;
DtInt8   myPsi;
DtInt8   myTheta;
DtInt8   myPhi;
DtU8     myFuelStatus;
DtU8     myHorizontalDeviation;
DtU8     myVerticalDeviation;
DtInt16  myMovementSpeed;
DtInt8   myTurretAzimuth;
DtInt8   myGunElevation;
DtInt8   myTurretSlewRate;

```

```
    DtInt8    myGunElevationRate;  
};  
#endif  
#endif
```


gedBasSoldier.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/isGroupOfP.h"

// class DtBasicGroundCombatSoldierRecord:
//
// Instances of DtBasicGroundCombatSoldierRecord are used to contain the
// entity state and logistics information about an individual entity within
// a group of entities.

class DtBasicGroundCombatSoldierRecord
{
public:
    // Constructors
    DtBasicGroundCombatSoldierRecord();
    DtBasicGroundCombatSoldierRecord(
        const DtBasicGroundCombatSoldierRecord& orig);

    // Destructor
    virtual ~DtBasicGroundCombatSoldierRecord();

    // Assignment operators
    DtBasicGroundCombatSoldierRecord& operator=(
        const DtBasicGroundCombatSoldierRecord& orig);
    DtBasicGroundCombatSoldierRecord& operator=(
        const DtNetBasicGroundCombatSoldierRecord& orig);

    // Set/Get the entity ID
    virtual void setEntityID(DtU16 value);
    virtual DtU16 entityID() const;

    // Set/Get the entity location X offset
    virtual void setX_Offset(DtInt8 value);
    virtual DtInt8 X_Offset() const;

    // Set/Get the entity location Y offset
    virtual void setY_Offset(DtInt8 value);
    virtual DtInt8 Y_Offset() const;

    // Set/Get the entity location Z offset
    virtual void setZ_Offset(DtInt8 value);
    virtual DtInt8 Z_Offset() const;

    // Set/Get appearance (as a 32 bit integer)
    virtual void setAppearance(DtU32 app);
    virtual DtU32 appearance() const;

    // Set/Get appearance by attribute
    virtual void setAppearanceAttribute(
        const DtAppearAttribute attr, const DtAppearAttrVal val);
    virtual DtAppearAttrVal appearanceAttribute(
        const DtAppearAttribute attr) const;

    // Set/Get the entity orientation psi
    virtual void setPsi(DtInt8 value);
    virtual DtInt8 psi() const;

```

```
// Set/Get the entity orientation theta
virtual void setTheta(DtInt8 value);
virtual DtInt8 theta() const;

// Set/Get the entity orientation phi
virtual void setPhi(DtInt8 value);
virtual DtInt8 phi() const;

// Set/Get the entity speed
virtual void setEntitySpeed(DtInt8 value);
virtual DtInt8 entitySpeed() const;

// Set/Get the head azimuth
virtual void setHeadAzimuth(DtInt8 value);
virtual DtInt8 headAzimuth() const;

// Set/Get the head elevation
virtual void setHeadElevation(DtInt8 value);
virtual DtInt8 headElevation() const;

// Set/Get the head scan rate
virtual void setHeadScanRate(DtInt8 value);
virtual DtInt8 headScanRate() const;

// Set/Get the head elevation rate
virtual void setHeadElevationRate(DtInt8 value);
virtual DtInt8 headElevationRate() const;

// Print this record
virtual void print() const;

// Load/Unload data from a packet record
virtual void unloadBaseNetRecord(
    const DtNetBasicGroundCombatSoldierRecord& record);
virtual void loadBaseNetRecord(
    DtNetBasicGroundCombatSoldierRecord& record) const;

protected:

    DtU16    myEntityID;
    DtInt8   myX_Offset;
    DtInt8   myY_Offset;
    DtInt8   myZ_Offset;
    DtU32    myAppearance;
    DtInt8   myPsi;
    DtInt8   myTheta;
    DtInt8   myPhi;
    DtInt8   myEntitySpeed;
    DtInt8   myHeadAzimuth;
    DtInt8   myHeadElevation;
    DtInt8   myHeadScanRate;
    DtInt8   myHeadElevationRate;

};
#endif
#endif
```

gedBasVehicle.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/isGroupOfP.h"

// class DtBasicGroundCombatVehicleRecord:
//
// Instances of DtBasicGroundCombatVehicleRecord are used to contain the
// entity state and logistics information about an individual entity within
// a group of entities.

class DtBasicGroundCombatVehicleRecord
{
public:
    // Constructors
    DtBasicGroundCombatVehicleRecord();
    DtBasicGroundCombatVehicleRecord(
        const DtBasicGroundCombatVehicleRecord& orig);

    // Destructor
    virtual ~DtBasicGroundCombatVehicleRecord();

    // Assignment operators
    DtBasicGroundCombatVehicleRecord& operator=(
        const DtBasicGroundCombatVehicleRecord& orig);
    DtBasicGroundCombatVehicleRecord& operator=(
        const DtNetBasicGroundCombatVehicleRecord& orig);

    // Set/Get the entity ID
    virtual void setEntityID(DtU16 value);
    virtual DtU16 entityID() const;

    // Set/Get the entity location X offset
    virtual void setX_Offset(DtInt8 value);
    virtual DtInt8 X_Offset() const;

    // Set/Get the entity location Y offset
    virtual void setY_Offset(DtInt8 value);
    virtual DtInt8 Y_Offset() const;

    // Set/Get the entity location Z offset
    virtual void setZ_Offset(DtInt8 value);
    virtual DtInt8 Z_Offset() const;

    // Set/Get appearance (as a 32 bit integer)
    virtual void setAppearance(DtU32 app);
    virtual DtU32 appearance() const;

    // Set/Get appearance by attribute
    virtual void setAppearanceAttribute(
        const DtAppearAttribute attr, const DtAppearAttrVal val);
    virtual DtAppearAttrVal appearanceAttribute(
        const DtAppearAttribute attr) const;

    // Set/Get the entity orientation psi
    virtual void setPsi(DtInt8 value);
    virtual DtInt8 psi() const;

```

```
// Set/Get the entity orientation theta
virtual void setTheta(DtInt8 value);
virtual DtInt8 theta() const;

// Set/Get the entity orientation phi
virtual void setPhi(DtInt8 value);
virtual DtInt8 phi() const;

// Set/Get the entity speed
virtual void setEntitySpeed(DtInt8 value);
virtual DtInt8 entitySpeed() const;

// Set/Get the turret azimuth
virtual void setTurretAzimuth(DtInt8 value);
virtual DtInt8 turretAzimuth() const;

// Set/Get the gun elevation
virtual void setGunElevation(DtInt8 value);
virtual DtInt8 gunElevation() const;

// Set/Get the turret slew rate
virtual void setTurretSlewRate(DtInt8 value);
virtual DtInt8 turretSlewRate() const;

// Set/Get the gun elevation rate
virtual void setGunElevationRate(DtInt8 value);
virtual DtInt8 gunElevationRate() const;

// Print this record
virtual void print() const;

// Load/Unload data from a packet record
virtual void unloadBaseNetRecord(
    const DtNetBasicGroundCombatVehicleRecord& record);
virtual void loadBaseNetRecord(
    DtNetBasicGroundCombatVehicleRecord& record) const;

protected:

    DtU16    myEntityID;
    DtInt8   myX_Offset;
    DtInt8   myY_Offset;
    DtInt8   myZ_Offset;
    DtU32    myAppearance;
    DtInt8   myPsi;
    DtInt8   myTheta;
    DtInt8   myPhi;
    DtInt8   myEntitySpeed;
    DtInt8   myTurretAzimuth;
    DtInt8   myGunElevation;
    DtInt8   myTurretSlewRate;
    DtInt8   myGunElevationRate;
};
#endif
#endif
```

gedEnhFixWing.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "gedBasFixWing.h"
#include "packets/isGroupOfP.h"

// class DtEnhancedFixedWingAircraftRecord:
//
// Instances of DtEnhancedFixedWingAircraftRecord are used to contain the
// entity state and logistics information about an individual entity within
// a group of entities.

class DtEnhancedFixedWingAircraftRecord :
    public DtBasicFixedWingAircraftRecord
{
public:
    // Constructors
    DtEnhancedFixedWingAircraftRecord();
    DtEnhancedFixedWingAircraftRecord(
        const DtEnhancedFixedWingAircraftRecord& orig);

    // Destructor
    virtual ~DtEnhancedFixedWingAircraftRecord();

    // Assignment operators
    DtEnhancedFixedWingAircraftRecord& operator=(
        const DtEnhancedFixedWingAircraftRecord& orig);
    DtEnhancedFixedWingAircraftRecord& operator=(
        const DtNetEnhancedFixedWingAircraftRecord& orig);

    // Set/Get the fuel status
    virtual void setSupplementalFuelStatus(DtU8 value);
    virtual DtU8 supplementalFuelStatus() const;

    // Set/Get the air maintenance status
    virtual void setAirMaintenanceStatus(DtU8 value);
    virtual DtU8 airMaintenanceStatus() const;

    // Set/Get the primary ammunition
    virtual void setPrimaryAmmunition(DtU8 value);
    virtual DtU8 primaryAmmunition() const;

    // Set/Get the secondary ammunition
    virtual void setSecondaryAmmunition(DtU8 value);
    virtual DtU8 secondaryAmmunition() const;

    // Print this record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetEnhancedFixedWingAircraftRecord& record);
    virtual void loadNetRecord(
        DtNetEnhancedFixedWingAircraftRecord& record) const;

protected:

```

```
DtU8    mySupplementalFuelStatus;  
DtU8    myAirMaintenanceStatus;  
DtU8    myPrimaryAmmunition;  
DtU8    mySecondaryAmmunition;  
  
};  
  
#endif  
#endif  
#endif
```

gedEnhRotWing.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "gedBasRotWing.h"
#include "packets/isGroupOfP.h"

// class DtEnhancedRotorWingAircraftRecord:
//
// Instances of DtEnhancedRotorWingAircraftRecord are used to contain the
// entity state and logistics information about an individual entity within
// a group of entities.

class DtEnhancedRotorWingAircraftRecord :
    public DtBasicRotorWingAircraftRecord
{
public:
    // Constructors
    DtEnhancedRotorWingAircraftRecord();
    DtEnhancedRotorWingAircraftRecord(
        const DtEnhancedRotorWingAircraftRecord& orig);

    // Destructor
    virtual ~DtEnhancedRotorWingAircraftRecord();

    // Assignment operators
    DtEnhancedRotorWingAircraftRecord& operator=(
        const DtEnhancedRotorWingAircraftRecord& orig);
    DtEnhancedRotorWingAircraftRecord& operator=(
        const DtNetEnhancedRotorWingAircraftRecord& orig);

    // Set/Get the fuel status
    virtual void setSupplementalFuelStatus(DtU8 value);
    virtual DtU8 supplementalFuelStatus() const;

    // Set/Get the air maintenance status
    virtual void setAirMaintenanceStatus(DtU8 value);
    virtual DtU8 airMaintenanceStatus() const;

    // Set/Get the primary ammunition
    virtual void setPrimaryAmmunition(DtU8 value);
    virtual DtU8 primaryAmmunition() const;

    // Set/Get the secondary ammunition
    virtual void setSecondaryAmmunition(DtU8 value);
    virtual DtU8 secondaryAmmunition() const;

    // Print this record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetEnhancedRotorWingAircraftRecord& record);
    virtual void loadNetRecord(
        DtNetEnhancedRotorWingAircraftRecord& record) const;

protected:

```

```
DtU8    mySupplementalFuelStatus;  
DtU8    myAirMaintenanceStatus;  
DtU8    myPrimaryAmmunition;  
DtU8    mySecondaryAmmunition;  
  
};  
  
#endif  
#endif  
#endif
```


gedEnhSoldier.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "gedBasSoldier.h"
#include "packets/isGroupOfP.h"

// class DtEnhancedGroundCombatSoldierRecord:
//
// Instances of DtEnhancedGroundCombatSoldierRecord are used to contain the
// entity state and logistics information about an individual entity within
// a group of entities.

class DtEnhancedGroundCombatSoldierRecord :
public DtBasicGroundCombatSoldierRecord
{

public:

    // Constructors
    DtEnhancedGroundCombatSoldierRecord();
    DtEnhancedGroundCombatSoldierRecord(
        const DtEnhancedGroundCombatSoldierRecord& orig);

    // Destructor
    virtual ~DtEnhancedGroundCombatSoldierRecord();

    // Assignment operators
    DtEnhancedGroundCombatSoldierRecord& operator=(
        const DtEnhancedGroundCombatSoldierRecord& orig);
    DtEnhancedGroundCombatSoldierRecord& operator=(
        const DtNetEnhancedGroundCombatSoldierRecord& orig);

    // Set/Get the water status
    virtual void setWaterStatus(DtU8 value);
    virtual DtU8 waterStatus() const;

    // Set/Get the rest status
    virtual void setRestStatus(DtRestStatus value);
    virtual DtRestStatus restStatus() const;

    // Set/Get the primary ammunition
    virtual void setPrimaryAmmunition(DtU8 value);
    virtual DtU8 primaryAmmunition() const;

    // Set/Get the secondary ammunition
    virtual void setSecondaryAmmunition(DtU8 value);
    virtual DtU8 secondaryAmmunition() const;

    // Print this record
    virtual void print() const;

    // Load/Unload data from a packet record

```

```
virtual void unloadNetRecord(  
    const DtNetEnhancedGroundCombatSoldierRecord& record);  
virtual void loadNetRecord(  
    DtNetEnhancedGroundCombatSoldierRecord& record) const;  
  
protected:  
  
    DtU8    myWaterStatus;  
    DtU8    myRestStatus;  
    DtU8    myPrimaryAmmunition;  
    DtU8    mySecondaryAmmunition;  
  
};  
  
#endif  
#endif  
#endif
```

gedEnhVehicle.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "gedBasVehicle.h"
#include "packets/isGroupOfP.h"

// class DtEnhancedGroundCombatVehicleRecord:
//
// Instances of DtEnhancedGroundCombatVehicleRecord are used to contain the
// entity state and logistics information about an individual entity within
// a group of entities.

class DtEnhancedGroundCombatVehicleRecord :
    public DtBasicGroundCombatVehicleRecord
{
public:
    // Constructors
    DtEnhancedGroundCombatVehicleRecord();
    DtEnhancedGroundCombatVehicleRecord(
        const DtEnhancedGroundCombatVehicleRecord& orig);

    // Destructor
    virtual ~DtEnhancedGroundCombatVehicleRecord();

    // Assignment operators
    DtEnhancedGroundCombatVehicleRecord& operator=(
        const DtEnhancedGroundCombatVehicleRecord& orig);
    DtEnhancedGroundCombatVehicleRecord& operator=(
        const DtNetEnhancedGroundCombatVehicleRecord& orig);

    // Set/Get the fuel status
    virtual void setFuelStatus(DtU8 value);
    virtual DtU8 fuelStatus() const;

    // Set/Get the ground maintenance status
    virtual void setGroundMaintenanceStatus(DtU8 value);
    virtual DtU8 groundMaintenanceStatus() const;

    // Set/Get the primary ammunition
    virtual void setPrimaryAmmunition(DtU8 value);
    virtual DtU8 primaryAmmunition() const;

    // Set/Get the secondary ammunition
    virtual void setSecondaryAmmunition(DtU8 value);
    virtual DtU8 secondaryAmmunition() const;

    // Print this record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetEnhancedGroundCombatVehicleRecord& record);
    virtual void loadNetRecord(
        DtNetEnhancedGroundCombatVehicleRecord& record) const;

protected:

```

```
DtU8    myFuelStatus;  
DtU8    myGroundMaintenanceStatus;  
DtU8    myPrimaryAmmunition;  
DtU8    mySecondaryAmmunition;  
  
};  
  
#endif  
#endif  
#endif
```

gedLogVehicle.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/isGroupOfP.h"

// class DtGroundLogisticsVehicleRecord:
//
// Instances of DtGroundLogisticsVehicleRecord are used to contain the
// entity state and logistics information about an individual entity within
// a group of entities.

class DtGroundLogisticsVehicleRecord
{
public:
    // Constructors
    DtGroundLogisticsVehicleRecord();
    DtGroundLogisticsVehicleRecord(
        const DtGroundLogisticsVehicleRecord& orig);

    // Destructor
    virtual ~DtGroundLogisticsVehicleRecord();

    // Assignment operators
    DtGroundLogisticsVehicleRecord& operator=(
        const DtGroundLogisticsVehicleRecord& orig);
    DtGroundLogisticsVehicleRecord& operator=(
        const DtNetGroundLogisticsVehicleRecord& orig);

    // Set/Get the entity ID
    virtual void setEntityID(DtU16 value);
    virtual DtU16 entityID() const;

    // Set/Get the entity location X offset
    virtual void setX_Offset(DtInt8 value);
    virtual DtInt8 X_Offset() const;

    // Set/Get the entity location Y offset
    virtual void setY_Offset(DtInt8 value);
    virtual DtInt8 Y_Offset() const;

    // Set/Get the entity location Z offset
    virtual void setZ_Offset(DtInt8 value);
    virtual DtInt8 Z_Offset() const;

    // Set/Get appearance (as a 32 bit integer)
    virtual void setAppearance(DtU32 app);
    virtual DtU32 appearance() const;

    // Set/Get appearance by attribute
    virtual void setAppearanceAttribute(
        const DtAppearAttribute attr, const DtAppearAttrVal val);
    virtual DtAppearAttrVal appearanceAttribute(
        const DtAppearAttribute attr) const;

    // Set/Get the entity orientation psi
    virtual void setPsi(DtInt8 value);
    virtual DtInt8 psi() const;

```

```
// Set/Get the entity orientation theta
virtual void setTheta(DtInt8 value);
virtual DtInt8 theta() const;

// Set/Get the entity orientation phi
virtual void setPhi(DtInt8 value);
virtual DtInt8 phi() const;

// Set/Get the entity speed
virtual void setEntitySpeed(DtInt8 value);
virtual DtInt8 entitySpeed() const;

// Print this record
virtual void print() const;

// Load/Unload data from a packet record
virtual void unloadNetRecord(
    const DtNetGroundLogisticsVehicleRecord& record);
virtual void loadNetRecord(
    DtNetGroundLogisticsVehicleRecord& record) const;

protected:

    DtU16    myEntityID;
    DtInt8   myX_Offset;
    DtInt8   myY_Offset;
    DtInt8   myZ_Offset;
    DtU32    myAppearance;
    DtInt8   myPsi;
    DtInt8   myTheta;
    DtInt8   myPhi;
    DtInt8   myEntitySpeed;

};
#endif
#endif
```

geodCoord.h

```

#include "vlVector.h"
#include "mathTypes.h"

// class DtGeodeticCoord:
//
// Instances of DtGeodeticCoord are used to explicitly represent
// Geodetic coordinates.

class DtGeodeticCoord : public DtVector
{
    friend void DtGeocToGeod(DtConstVector gc, DtGeodeticCoord& gd);

public:
    // default constructor
    DtGeodeticCoord();

    // constructor
    DtGeodeticCoord(double lat, double lon, double alt);

    // destructor
    virtual ~DtGeodeticCoord();

    // copy constructor
    DtGeodeticCoord(const DtGeodeticCoord& orig);

    // assignment operator
    DtGeodeticCoord& operator=(const DtGeodeticCoord& orig);

    // Set from geocentric coordinate
    // (convert geocentric to geodetic).
    // Can also use assignment operator.
    virtual DtGeodeticCoord& setGeocentric(DtConstVector geocentric);

    // Get geocentric coordinate equivalent
    // (convert geodetic to geocentric).
    virtual void getGeocentric(DtVectorRef geocentric) const;

    // Get geocentric coordinate equivalent
    // (convert geodetic to geocentric).
    // Note that this is not returning a reference.
    virtual DtVector geocentric() const;

    // Set/Get the latitude.
    virtual void setLat(DtFloat64 lat);
    virtual DtFloat64 lat() const;

    // Set/Get the longitude.
    virtual void setLon(DtFloat64 lon);
    virtual DtFloat64 lon() const;

    // Set/Get the altitude.
    virtual void setAlt(DtFloat64 alt);
    virtual DtFloat64 alt() const;
};

// Conversions between geocentric and geodetic:
// Use of these functions is discouraged in favor of DtGeodeticCoord
// member functions which perform conversions.
void DtGeocToGeod(DtConstVector in, DtGeodeticCoord& out);
void DtGeodToGeoc(const DtGeodeticCoord& in, DtVectorRef out);

// Conversions between geocentric and geodetic coordinates depend
// on the particular MapDatum used. The default is DtWGS84, but you

```

```
// can set the current MapDatum using DtUseMapDatum declared below.

// Spheroid
typedef struct DtSpheroid
{
    DtFloat64 semiMajor;    // semimajor axis of the ellipsoid, in meters
    DtFloat64 semiMinor;    // semiminor axis of the ellipsoid, in meters
} DtSpheroid;

typedef struct DtMapDatum
{
    DtSpheroid spheroid;
    DtFloat64 datumShift[3]; // add this to wgs84 to get GCC in this datum
} DtMapDatum;

// Set and Get current map datum
// NOTE: map datum memory passed to DtUseMapDatum() must remain valid
// until next change of datum.

void DtUseMapDatum(const DtMapDatum* md);
const DtMapDatum* DtCurrentMapDatum();

// Here are a few of the most important spheroids; we should put all 13 in.
#define DtWGS84Spheroid      {6378137.0, 6356752.3142}
#define DtInternationalSpheroid {6378388.0, 6356911.946127}
#define DtClark1866Spheroid  {6378206.4, 6356583.8}

// ...and Mapping Datums

const DtMapDatum DtWGS84      = {DtWGS84Spheroid,      0.,  0.,  0.};
const DtMapDatum DtED50      = {DtInternationalSpheroid, -87., -98., -
121.};
const DtMapDatum DtNASANAD27 = {DtClark1866Spheroid,   -8., 161.,
179.};
const DtMapDatum DtNASBNAD27 = {DtClark1866Spheroid,   -8., 159.,
175.};
const DtMapDatum DtCONUSNAD27 = {DtClark1866Spheroid,   -8., 160.,
176.};

// inline functions
```


globalObjDes.h

```
// A DtGlobalObjectDesignator is an identifier that can be used to
// describe an object when communicating among simulations in DIS
// PDUs or HLA objects or interactions. In DIS, the Global Object
// Designator is the entity ID. In HLA, it is a DtString denoting an
// object name when using RTI 1.3, or an HLA ObjectID when using
// RTI 1.0.3.

#include "str.h"
#include "chooseDefines.h"

#if DtHLA
#else
typedef DtEntityIdentifier DtObjectId;
#endif

#if DtHLA
#if DtRTI103
typedef DtObjectId DtGlobalObjectDesignator;
#else
typedef DtString DtGlobalObjectDesignator;
#endif
#else
typedef DtObjectId DtGlobalObjectDesignator;
#endif
```

globldHLst.h

```
#include "hashlist.h"
#include "globalObjDes.h"

// class DtGlobalIdHashlist:
//
// Instances of DtGlobalIdHashlist are hashlists, where you can
// store associations between DtGlobalObjectDesignators and arbitrary
// pieces of data.

class DtGlobalIdHashlist
{
public:
    // default constructor. Pass DtTrue only if the data that you are
    listing
    // are instances of a class derived from DtListedObject.
    DtGlobalIdHashlist(DtBoolean intrusive = DtFalse);

    // destructor
    virtual ~DtGlobalIdHashlist();

    // In the event that an item with key already exists, then add
    // will replace the data and move it to the location in list.
    // If your data was allocated, do not add unless you know key is
    // unique, or a memory leak will result.

    // Add an association between key and data to the list
    virtual DtListItem* add(const DtGlobalObjectDesignator& key, void* data);

    // Remove the current association involving key from the list.
    // Returns the data that was associated with key.
    virtual void* remove(const DtGlobalObjectDesignator& key);

    // Lookup the data associated with key.
    virtual void* lookup(const DtGlobalObjectDesignator& key) const;
    virtual void* operator[](const DtGlobalObjectDesignator& key) const;

    // Get a DtList containing all of the void* data that you
    // have listed, regardless of keys.
    virtual const DtList* list() const;

protected:
    // copy constructor
    DtGlobalIdHashlist(const DtGlobalIdHashlist& orig);

    // assignment operator
    DtGlobalIdHashlist& operator=(const DtGlobalIdHashlist& orig);

protected:
    DtList* myList;
    DtHashlist* myHashlist;
};
typedef DtGlobalIdHashlist DtGlobalIdHashList;
```

gridAxisRecFix.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/gridDataP.h"

// class DtGridAxisRecordFixed:
//
// DtGridAxisRecordFixed is the fixed grid axis record used by the gridded
// data PDU class (DtGridDataPdu). It is also the base class for the
// variable grid axis record.

class DtGridAxisRecordFixed
{
public:
    // Constructors
    DtGridAxisRecordFixed();
    DtGridAxisRecordFixed(const DtGridAxisRecordFixed& orig);

    // Destructor
    virtual ~DtGridAxisRecordFixed();

    // Assignment operator
    DtGridAxisRecordFixed& operator=(const DtGridAxisRecordFixed& orig);
    DtGridAxisRecordFixed& operator=(const DtNetGridAxisRecordFixed& orig);

    // Set/Get the domain initial Xi
    virtual void setDomainInitialXi(const DtFloat64& initial);
    virtual const DtFloat64& domainInitialXi() const;

    // Set/Get the domain final Xi
    virtual void setDomainFinalXi(const DtFloat64& final);
    virtual const DtFloat64& domainFinalXi() const;

    // Set/Get the domain number of points
    virtual void setDomainNumPointsXi(DtU16 num);
    virtual DtU16 domainNumPointsXi() const;

    // Set/Get the interleaf factor
    virtual void setInterleafFactor(DtU8 factor);
    virtual DtU8 interleafFactor() const;

    // Get the axis type
    virtual DtGridAxisType axisType() const;

    // Set/Get the number of points. May return false if PDU size exceeds
    // the maximum size for a PDU.
    virtual DtBoolean setNumPointsXi(DtU16 num);
    virtual DtU16 numPointsXi() const;

    // Set/Get the initial index
    virtual void setInitialIndexXi(DtU16 index);
    virtual DtU16 initialIndexXi() const;

    // Set the total number of bytes required for the network structure.
    virtual int recordLength() const;

    // Print the sample type and data representation
    virtual void print() const;

```

```
// Load/unload data from a packet record
virtual void baseUnloadNetRecord(const DtNetGridAxisRecordFixed&
record);
virtual void baseLoadNetRecord(DtNetGridAxisRecordFixed& record) const;

public:

    // Get the grid axis record type from a network structure
    static DtGridAxisType axisType(const DtNetGridAxisRecordFixed* rec);

    // Get the grid axis record length from a network structure
    static int recordLength(const DtNetGridAxisRecordFixed* rec);

protected:

    // Constructor for derived classes
    DtGridAxisRecordFixed(DtGridAxisType axType);

protected:

    // Data members:
    DtFloat64          myDomainInitialXi;
    DtFloat64          myDomainFinalXi;
    DtU16              myDomainNumPointsXi;
    DtU8               myInterleafFactor;
    DtGridAxisType     myAxisType;
    DtU16              myNumPointsXi;
    DtU16              myInitialIndexXi;
};
#endif
#endif
```

gridAxiRecVar.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"
#include "exerciseConn.h"

#include "gridAxiRecFix.h"

// class DtGridAxisRecordVariable:
//
// Instances of DtGridAxisRecordVariable are used to contain the information
// in the grid axis variable spacing records used by the gridded data PDU.

class DtGridAxisRecordVariable : public DtGridAxisRecordFixed
{
public:
    // Constructors
    DtGridAxisRecordVariable();
    DtGridAxisRecordVariable(const DtGridAxisRecordVariable& orig);

    // Destructor
    virtual ~DtGridAxisRecordVariable();

    // Assignment operators
    DtGridAxisRecordVariable& operator=(
        const DtGridAxisRecordVariable& orig);
    DtGridAxisRecordVariable& operator=(
        const DtNetGridAxisRecordVariable& orig);

    // Get the total number of bytes required for the network structure.
    virtual int recordLength() const;

    // Set/Get the coordinate scale
    virtual void setCoordinateScaleXi(const DtFloat64& num);
    virtual const DtFloat64& coordinateScaleXi() const;

    // Set/Get the coordinate offset
    virtual void setCoordinateOffsetXi(const DtFloat64& num);
    virtual const DtFloat64& coordinateOffsetXi() const;

    // Set/Get the number of points.
    // If the number of points is changed, previous points are not preserved.
    virtual DtBoolean setNumPointsXi(DtU16 num);
    virtual DtU16 numPointsXi() const;

    // Set/Get an Xi value
    virtual void setXi(int index, DtU16 x);
    virtual DtU16 xi(int index) const;

    // Print this record
    virtual void print() const;

    // Load/unload data from a packet record
    virtual void unloadNetRecord(const DtNetGridAxisRecordVariable& record);
    virtual void loadNetRecord(DtNetGridAxisRecordVariable& record) const;

public:
    // Get the grid axis record length from a network structure

```

```
static int recordLength(DtNetGridAxisRecordVariable* rec);  
protected:  
    DtFloat64          myCoordinateScaleXi;  
    DtFloat64          myCoordinateOffsetXi;  
    DtU16*             myXi;  
};  
#endif  
#endif
```

gridDataPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"

#include "packets/gridDataP.h"

#include "gridAxiRecFix.h"
#include "gridAxiRecVar.h"

#include "gridDataRec.h"
#include "gridDataRecT0.h"
#include "gridDataRecT1.h"
#include "gridDataRecT2.h"

#include "enviroTypeRec.h"

// class DtGriddedDataPdu:
//
// Instances of DtGriddedDataPdu are used to communicate environmental data
// defined on a grid.

// DtGriddedDataPdu callback function signature.
typedef void (*DtGriddedDataPduCb)(DtGriddedDataPdu* pdu, void* usr);

class DtGriddedDataPdu : public DtPdu
{
public:
    // Constructors
    DtGriddedDataPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtGriddedDataPdu(const DtNetGriddedDataPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtGriddedDataPdu(const DtGriddedDataPdu& orig);

    // destructor
    virtual ~DtGriddedDataPdu();

    // Create a DtGriddedDataPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtGriddedDataPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtGriddedDataPduCb cb, void* usr);

    // Assignment operator
    DtGriddedDataPdu& operator=(const DtGriddedDataPdu& orig);

    // Set/Get environmental simulation application identifier
    virtual void setEnvSimAppID(const DtEntityIdentifier& app);
    virtual const DtEntityIdentifier& envSimAppID() const;

    // Set/Get field number
    virtual void setFieldNumber(DtU16 field);

```

```
virtual DtU16 fieldNumber() const;

// Set/Get the PDU index number
virtual void setPduNumber(DtU16 num);
virtual DtU16 pduNumber() const;

// Set/Get total number of PDUs used to transmit the environmental data
virtual void setPduTotal(DtU16 total);
virtual DtU16 pduTotal() const;

// Set/Get coordinate system
virtual void setCoordSystem(DtU16 sys);
virtual DtU16 coordSystem() const;

// Set/Get number of grid axes
virtual DtBoolean setNumGridAxes(DtU8 axes);
virtual DtU8 numGridAxes() const;

// Set/Get constant grid indicator
virtual void setConstantGrid(DtU8 constant);
virtual DtU8 constantGrid() const;

// Set/Get environment type
virtual void setEnvironmentType(const DtEnvironmentTypeRecord& type);
virtual const DtEnvironmentTypeRecord& environmentType() const;

// Set/Get orientation
virtual void setOrientation(const DtTaitBryan& orientation);
virtual const DtTaitBryan& orientation() const;

// Set/Get sample time. This is a 64-bit quantity. The argument
// index selects between the most significant 32-bit word (0) and least
// significant 32-bit word (1) of the complete 64-bit value.
virtual void setSampleTime(int index, DtU32 time);
virtual DtU32 sampleTime(int index) const;

// Set/Get total values.
virtual void setTotalValues(DtU32 total);
virtual DtU32 totalValues() const;

// Set/Get vector dimension. setVectorDim may return false if the PDU
// size exceeds the maximum allowable size for a PDU.
virtual DtBoolean setVectorDim(DtU8 dim);
virtual DtU8 vectorDim() const;

// Get the grid axes record type for a particular record
virtual DtGridAxisType gridAxisRecordType(int index) const;

// Set/Get the grid axis record type and values of record index to
// values specified by the initializer in the second argument. If index
// is out of bounds, or if the record type at the specified index does
// not match the parameter record type (for a get), then DtFalse is
// returned. If the space would exceed the maximum space permitted for
// a PDU (for a set) then DtFalse may be returned.
virtual DtBoolean setGridAxisRecord(int index,
    const DtGridAxisRecordFixed& init);
virtual DtBoolean getGridAxisRecord(int index,
    DtGridAxisRecordFixed& record) const;
virtual DtBoolean setGridAxisRecord(int index,
    const DtGridAxisRecordVariable& init);
virtual DtBoolean getGridAxisRecord(int index,
    DtGridAxisRecordVariable& record) const;

// Get the grid data record type for a particular record
virtual DtGridDataRecType gridDataRecordType(int index) const;

// Set/Get the grid data record type and values of record index to
```



```

// values specified by the initializer in the second argument. If index
// is out of bounds, or if the record type at the specified index does
// not match the parameter record type (for a get), then DtFalse is
// returned. If the space would exceed the maximum space permitted for
// a PDU (for a set) then DtFalse may be returned.
virtual DtBoolean setGridDataRecord(int index,
    const DtGridDataRecordType0& init);
virtual DtBoolean getGridDataRecord(int index,
    DtGridDataRecordType0& record) const;
virtual DtBoolean setGridDataRecord(int index,
    const DtGridDataRecordType1& init);
virtual DtBoolean getGridDataRecord(int index,
    DtGridDataRecordType1& record) const;
virtual DtBoolean setGridDataRecord(int index,
    const DtGridDataRecordType2& init);
virtual DtBoolean getGridDataRecord(int index,
    DtGridDataRecordType2& record) const;

// Print the data contained by this PDU
virtual void printData() const;

```

protected:

```

// Returns the internal PDU kind
virtual DtPduKind internalGetPduKind() const;

// Casts the netPdu element to our network struct
virtual DtNetGriddedDataPdu* netPduStruct() const;

// Returns a pointer to the first byte of variable length data
virtual char* var() const;

// Casts the 'this' pointer for changing elements in a const function
// Non-virtual since you can't override based on return type
DtGriddedDataPdu* self() const;

// Returns the length of a given grid axis record
virtual int gridAxisRecordLength(DtNetGridAxisRecordFixed* rec) const;

// Returns the length of a given grid data record
virtual int gridDataRecordLength(DtNetGridDataRecord* rec) const;

// Returns a buffer offset to the given grid axis record
virtual int gridAxisRecordOffset(int recordIndex) const;

// Returns a buffer offset to the given grid data record
virtual int gridDataRecordOffset(int recordIndex) const;

// Resizes a grid axis record
virtual DtBoolean resizeGridAxisRecord(DtNetGridAxisRecordFixed* rec,
    int newSize);

// Resizes a grid data record
virtual DtBoolean resizeGridDataRecord(DtNetGridDataRecord* rec,
    int newSize);

```

protected:

```

// Size of PDU with zero parameter records.
static int theStaticSize;

```

protected:

```

// Temporary objects for reference returns
DtEntityIdentifier myEntityID;
DtEnvironmentTypeRecord myEnvironmentType;
DtTaitBryan myEulerAngles;

```

```
};  
#endif  
#endif
```

gridDataRec.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/gridDataP.h"

// class DtGridDataRecord:
//
// DtGridDataRecord is an abstract base class for the grid data record
// classes which are used by the gridded data PDU class (DtGridDataPdu).

class DtGridDataRecord
{
public:
    // Destructor
    virtual ~DtGridDataRecord();

    // Set/Get the grid data record sample type
    virtual void setSampleType(DtU16 type);
    virtual DtU16 sampleType() const;

    // Get the grid data record representation type
    virtual DtGridDataRecType recordType() const;

    // Get the total number of bytes required for the network structure.
    virtual int recordLength() const = 0;

    // Print the sample type and data representation
    virtual void print() const;

    // Load/unload data from a packet record
    virtual void baseUnloadNetRecord(const DtNetGridDataRecord& record);
    virtual void baseLoadNetRecord(DtNetGridDataRecord& record) const;

public:
    // Get the grid data record representation type from a network structure
    static DtGridDataRecType recordType(DtNetGridDataRecord* rec);

protected:
    // Constructors (protected because this is an abstract base class)
    DtGridDataRecord();
    DtGridDataRecord(const DtGridDataRecord& orig);
    DtGridDataRecord(DtGridDataRecType recType);

    // Assignment operator
    DtGridDataRecord& operator=(const DtGridDataRecord& orig);

protected:
    // Data members:
    DtU16 mySampleType;
    DtGridDataRecType myRecordType;
};
#endif
#endif

```

gridDataRecT0.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/gridDataP.h"
#include "gridDataRec.h"

// class DtGridDataRecordType0:
//
// Instances of DtGridDataRecordType0 are used to contain the information
// in the grid data records (of representation type 0) used by the gridded
// data PDU class.

class DtGridDataRecordType0 : public DtGridDataRecord
{
public:

    // Constructors
    DtGridDataRecordType0();
    DtGridDataRecordType0(const DtGridDataRecordType0& orig);

    // Destructor
    virtual ~DtGridDataRecordType0();

    // Assignment operators
    DtGridDataRecordType0& operator=(
        const DtGridDataRecordType0& orig);
    DtGridDataRecordType0& operator=(
        const DtNetGridDataRecordType0& orig);

    // Get the total number of bytes required for the network structure.
    virtual int recordLength() const;

    // Set/Get the number of data bytes
    // When number of data bytes is changed, old values are not preserved
    virtual DtBoolean setNumDataBytes(DtU16 bytes);
    virtual DtU16 numDataBytes() const;

    // Set/Get a data byte
    virtual void setDataByte(int byteNum, DtU8 data);
    virtual DtU8 dataByte(int byteNum) const;

    // Set/Get multiple data bytes. Provide a pointer and data is
    // copied to and from that memory. Will copy numDataBytes() on get.
    virtual DtBoolean setData(const DtU8* data, int numBytes);
    virtual void getData(DtU8* buff) const;

    // Print this record
    virtual void print() const;

    // Load/unload data from a packet record
    virtual void unloadNetRecord(const DtNetGridDataRecordType0& record);
    virtual void loadNetRecord(DtNetGridDataRecordType0& record) const;

public:

    // Get the grid data record representation type from a network structure
    static int recordLength(DtNetGridDataRecordType0* rec);
```

protected:

```
    DtU16                myNumDataBytes;  
    DtU8*                myData;  
};  
#endif  
#endif
```

gridDataRecT1.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/gridDataP.h"
#include "gridDataRec.h"

// class DtGridDataRecordType1:
//
// Instances of DtGridDataRecordType1 are used to contain the information
// in the grid data records (of representation type 1) used by the gridded
// data PDU class.

class DtGridDataRecordType1 : public DtGridDataRecord
{
public:

    // Constructors
    DtGridDataRecordType1();
    DtGridDataRecordType1(const DtGridDataRecordType1& orig);

    // Destructor
    virtual ~DtGridDataRecordType1();

    // Assignment operators
    DtGridDataRecordType1& operator=(
        const DtGridDataRecordType1& orig);
    DtGridDataRecordType1& operator=(
        const DtNetGridDataRecordType1& orig);

    // Get the total number of bytes required for the network structure.
    virtual int recordLength() const;

    // Set/Get the field scale
    virtual void setFieldScale(DtFloat32 scale);
    virtual DtFloat32 fieldScale() const;

    // Set/Get the field offset
    virtual void setFieldOffset(DtFloat32 offset);
    virtual DtFloat32 fieldOffset() const;

    // Set/Get the number of data values
    // Old values are not preserved when number of values is changed
    virtual DtBoolean setNumDataValues(DtU16 values);
    virtual DtU16 numDataValues() const;

    // Set/Get a data value
    virtual void setDataValue(int index, DtInt16 data);
    virtual DtInt16 dataValue(int index) const;

    // Print this record
    virtual void print() const;

    // Load/unload data from a packet record
    virtual void unloadNetRecord(const DtNetGridDataRecordType1& record);
    virtual void loadNetRecord(DtNetGridDataRecordType1& record) const;

public:
```

```
// Get the grid data record representation type from a network structure
static int recordLength(DtNetGridDataRecordType1* rec);

protected:

    DtFloat32          myFieldScale;
    DtFloat32          myFieldOffset;
    DtU16              myNumDataValues;
    DtInt16*           myData;

};
#endif
#endif
```

gridDataRecT2.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/gridDataP.h"
#include "gridDataRec.h"

// class DtGridDataRecordType2:
//
// Instances of DtGridDataRecordType2 are used to contain the information
// in the grid data records (of representation type 2) used by the gridded
// data PDU class.

class DtGridDataRecordType2 : public DtGridDataRecord
{
public:
    // Constructors
    DtGridDataRecordType2();
    DtGridDataRecordType2(const DtGridDataRecordType2& orig);

    // Destructor
    virtual ~DtGridDataRecordType2();

    // Assignment operators
    DtGridDataRecordType2& operator=(
        const DtGridDataRecordType2& orig);
    DtGridDataRecordType2& operator=(
        const DtNetGridDataRecordType2& orig);

    // Get the total number of bytes required for the network structure.
    virtual int recordLength() const;

    // Set/Get the number of data values
    // Old values are not preserved when the number of values is changed
    virtual DtBoolean setNumDataValues(DtU16 values);
    virtual DtU16 numDataValues() const;

    // Set/Get a data value
    virtual void setDataValue(int index, DtFloat32 data);
    virtual DtFloat32 dataValue(int index) const;

    // Print this record
    virtual void print() const;

    // Load/unload data from a packet record
    virtual void unloadNetRecord(const DtNetGridDataRecordType2& record);
    virtual void loadNetRecord(DtNetGridDataRecordType2& record) const;

public:
    // Get the grid data record representation type from a network structure
    static int recordLength(DtNetGridDataRecordType2* rec);

protected:
    DtU16 myNumDataValues;
    DtFloat32* myData;
```



```
};  
#endif  
#endif
```

hAckDec.h

```
#if DtHLA

#include "interDecoder.h"

#include "chooseDefines.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_ACK_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtAcknowledgeInteraction, paramName)

#define DtDEFINE_SIMPLE_ACK_PARAM_DECODER( \
    paramName, netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER( \
        DtAcknowledgeDecoder, DtAcknowledgeInteraction, \
        paramName, netType, mutator)

#define DtDEFINE_SIMPLE_ACK_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
        DtAcknowledgeDecoder, DtAcknowledgeInteraction, \
        paramName, netType, mutator, castExpr)

// class DtAcknowledgeDecoder:
//
// Instances of DtAcknowledgeDecoder are used to decode Acknowledge
// Interactions from the network.

class DtAcknowledgeDecoder : public DtInteractionDecoder
{
public:

    // constructor
    DtAcknowledgeDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtAcknowledgeDecoder();

    // copy constructor
    DtAcknowledgeDecoder(const DtAcknowledgeDecoder& orig);

    // assignment operator
    DtAcknowledgeDecoder& operator=(const DtAcknowledgeDecoder& orig);

protected:

    // Individual parameter decoders

    DtDECLARE_ACK_PARAM_DECODER(OriginatingEntity);
    DtDECLARE_ACK_PARAM_DECODER(ReceivingEntity);
    DtDECLARE_ACK_PARAM_DECODER(RequestID);
    DtDECLARE_ACK_PARAM_DECODER(AcknowledgeFlag);
    DtDECLARE_ACK_PARAM_DECODER(ResponseFlag);
};
#endif
```

hAckEnc.h

```

#if DtHLA

#include "interEncoder.h"

#define DtDECLARE_ACK_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtAcknowledgeInteraction, paramName)

#define DtDEFINE_SIMPLE_ACK_PARAM_ENCODER(paramName, netType, inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtAcknowledgeEncoder, \
        DtAcknowledgeInteraction, paramName, netType, inspector)

// class DtAcknowledgeEncoder:
//
// DtAcknowledgeEncoder is a DtInteractionEncoder that registers
// parameter encoders for all of the ACK interaction parameters.

class DtAcknowledgeEncoder : public DtInteractionEncoder
{
public:
    DtAcknowledgeEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtAcknowledgeEncoder();

    // copy constructor
    DtAcknowledgeEncoder(const DtAcknowledgeEncoder& orig);

    // assignment operator
    DtAcknowledgeEncoder& operator=(const DtAcknowledgeEncoder& orig);

protected:
    // Individual parameter encoders

    DtDECLARE_ACK_PARAM_ENCODER(OriginatingEntity);
    DtDECLARE_ACK_PARAM_ENCODER(ReceivingEntity);
    DtDECLARE_ACK_PARAM_ENCODER(RequestID);
    DtDECLARE_ACK_PARAM_ENCODER(AcknowledgeFlag);
    DtDECLARE_ACK_PARAM_ENCODER(ResponseFlag);
};
#endif

```

hAckInter.h

```
#if DtHLA

#include "hIntWEncDec.h"
#include "hostStructs.h"
#include "mathTypes.h"

#include "chooseDefines.h"

// class DtAcknowledgeInteraction:
//
// Instances of DtAcknowledgeInteraction are used to represent
// Acknowledge Interactions in the RPR ROM.

// Acknowledge interaction callback function signature.
typedef void (*DtAcknowledgeInteractionCb) (DtAcknowledgeInteraction*
inter,
void* usr);

class DtAcknowledgeInteraction : public DtInteractionWithEncDec
{
public:

    // default constructor
    DtAcknowledgeInteraction();

    // destructor
    virtual ~DtAcknowledgeInteraction();

    // copy constructor
    DtAcknowledgeInteraction(const DtAcknowledgeInteraction& orig);

    // assignment operator
    DtAcknowledgeInteraction& operator=(const DtAcknowledgeInteraction&
orig);

    // Print the interaction's data.
    virtual void printData() const;

    // Set/Get sender ID
    virtual void setSenderId(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& senderId() const;

    // Set/Get receiver ID
    virtual void setReceiverId(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& receiverId() const;

    // Set/Get acknowledge flag
    virtual void setAcknowledgeFlag(DtAcknowledgeFlag flag);
    virtual DtAcknowledgeFlag acknowledgeFlag() const;

    // Set/Get response flag
    virtual void setResponseFlag(DtResponseFlag flag);
    virtual DtResponseFlag responseFlag() const;

    // Set/Get request ID
    virtual void setRequestId(int id);
    virtual int requestId() const;

    // Name of the interaction (Not nec. the ClassName from the FOM).
    virtual const char* name() const;

public:
```

```
// Create a DtAcknowledgeInteraction.
// Caller is responsible for deletion.
static DtInteraction* create();

// Add (register)/Remove (deregister) function to be called when
// interaction occurs.
static void addCallback(DtExerciseConn* conn,
    DtAcknowledgeInteractionCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtAcknowledgeInteractionCb cb, void* usr);

protected:
    DtEntityIdentifier myOrigEnt;
    DtEntityIdentifier myReceiveEnt;
    int myRequestId;
    DtAcknowledgeFlag myAckFlag;
    DtResponseFlag myRespFlag;
};
#endif
```

hActReqDec.h

```
#if DtHLA

#include "interDecoder.h"

#include "chooseDefines.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_ACTREQ_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtActionRequestInteraction, paramName)

#define DtDEFINE_SIMPLE_ACTREQ_PARAM_DECODER( \
    paramName, netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER( \
        DtActionRequestDecoder, DtActionRequestInteraction, \
        paramName, netType, mutator)

#define DtDEFINE_SIMPLE_ACTREQ_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
        DtActionRequestDecoder, DtActionRequestInteraction, \
        paramName, netType, mutator, castExpr)

// class DtActionRequestDecoder:
//
// Instances of DtActionRequestDecoder are used to decode ActionRequest
// Interactions from the network.

class DtActionRequestDecoder : public DtInteractionDecoder
{
public:

    // constructor
    DtActionRequestDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtActionRequestDecoder();

    // copy constructor
    DtActionRequestDecoder(const DtActionRequestDecoder& orig);

    // assignment operator
    DtActionRequestDecoder& operator=(const DtActionRequestDecoder& orig);

protected:

    // Individual parameter decoders

    DtDECLARE_ACTREQ_PARAM_DECODER(OriginatingEntity);
    DtDECLARE_ACTREQ_PARAM_DECODER(ReceivingEntity);
    DtDECLARE_ACTREQ_PARAM_DECODER(RequestID);
    DtDECLARE_ACTREQ_PARAM_DECODER(Action);
    DtDECLARE_ACTREQ_PARAM_DECODER(FixedDatums);
    DtDECLARE_ACTREQ_PARAM_DECODER(VariableDatumSet);
};
#endif
```

hActReqEnc.h

```

#if DtHLA

#include "interEncoder.h"

#define DtDECLARE_ACTREQ_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtActionRequestInteraction, paramName)

#define DtDEFINE_SIMPLE_ACTREQ_PARAM_ENCODER(paramName, netType, inspector) \
    \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtActionRequestEncoder, \
        DtActionRequestInteraction, paramName, netType, inspector)

// class DtActionRequestEncoder:
//
// DtActionRequestEncoder is a DtInteractionEncoder that registers
// parameter encoders for all of the ActionRequest interaction parameters.

class DtActionRequestEncoder : public DtInteractionEncoder
{
public:

    DtActionRequestEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtActionRequestEncoder();

    // copy constructor
    DtActionRequestEncoder(const DtActionRequestEncoder& orig);

    // assignment operator
    DtActionRequestEncoder& operator=(const DtActionRequestEncoder& orig);

protected:

    // Individual parameter encoders

    DtDECLARE_ACTREQ_PARAM_ENCODER(OriginatingEntity);
    DtDECLARE_ACTREQ_PARAM_ENCODER(ReceivingEntity);
    DtDECLARE_ACTREQ_PARAM_ENCODER(RequestID);
    DtDECLARE_ACTREQ_PARAM_ENCODER(Action);
    DtDECLARE_ACTREQ_PARAM_ENCODER(FixedDatums);
    DtDECLARE_ACTREQ_PARAM_ENCODER(VariableDatumSet);
};
#endif

```

hActReqInter.h

```
#if DtHLA

#include "hInterWData.h"
#include "hostStructs.h"
#include "mathTypes.h"

#include "chooseDefines.h"

// class DtActionRequestInteraction:
//
// Instances of DtActionRequestInteraction are used to represent
// ActionRequest Interactions in the RPR FOM. The variable and fixed
// datum records are accessed through DtInteractionWithData member
// functions.

// ActionRequest interaction callback function signature.
typedef void (*DtActionRequestInteractionCb)
    (DtActionRequestInteraction* inter, void* usr);

class DtActionRequestInteraction : public DtInteractionWithData
{
public:

    // default constructor
    DtActionRequestInteraction();

    // destructor
    virtual ~DtActionRequestInteraction();

    // copy constructor
    DtActionRequestInteraction(const DtActionRequestInteraction& orig);

    // assignment operator
    DtActionRequestInteraction& operator=(
        const DtActionRequestInteraction& orig);

    // Print the interaction's data.
    virtual void printData() const;

    // 'Set/Get sender ID
    virtual void setSenderId(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& senderId() const;

    // Set/Get receiver ID
    virtual void setReceiverId(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& receiverId() const;

    // Set/Get request ID
    virtual void setRequestId(int id);
    virtual int requestId() const;

    // Set/Get action ID
    virtual void setActionId(DtActionId id);
    virtual DtActionId actionId() const;

    // Name of the interaction (Not nec. the ClassName from the FOM).
    virtual const char* name() const;

public:

    // Create a DtActionRequestInteraction.
    // Caller is responsible for deletion.
    static DtInteraction* create();

};

#endif
```



```
// Add (register)/Remove (deregister) function to be called when
// interaction occurs.
static void addCallback(DtExerciseConn* conn,
    DtActionRequestInteractionCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtActionRequestInteractionCb cb, void* usr);

protected:
    DtEntityIdentifier myOrigEnt;
    DtEntityIdentifier myReceiveEnt;
    int myRequestId;
    DtActionId myActionId;
};
#endif
```

hAggPub.h

```
#if DtHLA

#include "hObjPub.h"

// class DtAggregatePublisher:
//
// Instances of DtAggregatePublisher are used for local or source simula-
// tions
// of aggregate entities, (i.e. these entities are not dependent on data
// being
// produced elsewhere on the network for their existence). These
// entities send data to the network.

class DtAggregatePublisher : public DtObjectPublisher
{
public:

    // Function prototype for functions that create
    // DtAggregateStateRepositories
    typedef DtAggregateStateRepository* (*DtStateRepCreator)();

public:

    // constructors
    DtAggregatePublisher(RTI::ObjectClassHandle h,
        DtExerciseConn* conn, const char* name = NULL);

    DtAggregatePublisher(const DtEntityType& type,
        DtExerciseConn* conn, const char* name = NULL);

    DtAggregatePublisher(const DtEntityType& type,
        DtExerciseConn *conn,
        DtDeadReckonTypes algorithm,
        DtForceType forceId,
        const char* name = NULL);

    // destructor
    virtual ~DtAggregatePublisher();

    // Returns a pointer to the AggregateStateRepository - through
    // which you can set or inspect current state information.
    virtual DtAggregateStateRepository* aggregateStateRep();

    // Synonym for aggregateStateRep()
    virtual DtAggregateStateRepository* asr();

public:

    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

protected:

    // copy constructor - not implemented
    DtAggregatePublisher(const DtAggregatePublisher& orig);

    // assignment operator - not implemented
    DtAggregatePublisher& operator=(const DtAggregatePublisher& orig);

    // Non-virtual since called from constructor. If you don't like
    // the default mapping between entity type and object class,
    // just pass the object class you want to the constructor.
    RTI::ObjectClassHandle objectClassHandleFromType(
```

```
        const DtEntityType &type);  
protected:  
    static DtStateRepCreator theStateRepCreator;  
};  
#endif
```

hAppRadSigDec.h

```
#if DtHLA

#include "chooseDefines.h"
#include "interDecoder.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtSignalInteraction, paramName)

#define DtDEFINE_SIMPLE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER(paramName, \
    \
    netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER(DtAppSpecificRadioSignalDecoder, \
    DtSignalInteraction, paramName, netType, mutator)

#define DtDEFINE_SIMPLE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER_WITH_CAST( \
    \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
    DtAppSpecificRadioSignalDecoder, DtSignalInteraction, \
    paramName, netType, mutator, castExpr)

// class DtAppSpecificRadioSignalDecoder:
//
// Instances of DtAppSpecificRadioSignalDecoder are

class DtAppSpecificRadioSignalDecoder : public DtInteractionDecoder
{
public:

    // default constructor
    DtAppSpecificRadioSignalDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtAppSpecificRadioSignalDecoder();

    // copy constructor
    DtAppSpecificRadioSignalDecoder(
        const DtAppSpecificRadioSignalDecoder& orig);

    // assignment operator
    DtAppSpecificRadioSignalDecoder& operator=(
        const DtAppSpecificRadioSignalDecoder& orig);

protected:

    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER(HostRadioID);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER(DataRate);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER(SignalDataLength);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER(SignalData);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER(TacticalDataLinkType);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER(TDLMessageCount);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_DECODER(UserProtocolID);
};
#endif // DtHLA
```

hAppRadSigEnc.h

```

#if DtHLA

#include "chooseDefines.h"
#include "interEncoder.h"

#define DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtSignalInteraction, paramName)

#define DtDEFINE_SIMPLE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_ENCODER(paramName, \
    \
    netType, inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtAppSpecificRadioSignalEncoder, \
    DtSignalInteraction, paramName, netType, inspector)

// class DtAppSpecificRadioSignalEncoder:
//
// Instances of DtAppSpecificRadioSignalEncoder are

class DtAppSpecificRadioSignalEncoder : public DtInteractionEncoder
{
public:

    // default constructor
    DtAppSpecificRadioSignalEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtAppSpecificRadioSignalEncoder();

    // copy constructor
    DtAppSpecificRadioSignalEncoder(
        const DtAppSpecificRadioSignalEncoder& orig);

    // assignment operator
    DtAppSpecificRadioSignalEncoder& operator=(
        const DtAppSpecificRadioSignalEncoder& orig);

protected:

    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_ENCODER(HostRadioID);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_ENCODER(DataRate);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_ENCODER(SignalDataLength);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_ENCODER(SignalData);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_ENCODER(TacticalDataLinkType);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_ENCODER(TDLMessageCount);
    DtDECLARE_APP_SPECIFIC_RADIO_SIGNAL_PARAM_ENCODER(UserProtocolID);
};
#endif //DtHLA

```

hashlist.h

```
#include "vllist.h"
#include "baseHashKey.h"
#include "hashItem.h"

// Include obsolete old version for backwards compatibility.
#include "oldHashList.h"

// class DtHashlist:
//
// A DtHashlist is used to store associations between keys and arbitrary
pieces
// of data passed as void*. Keys must be instances of a class derived
// from DtBaseHashKey (defined in baseHashKey.h). A subclass of DtHashKey
// defines a hashing function that hashes to an unsigned long, as well as a
// stringRep function that converts your key to a string representation.
This
// allows DtHashlist to compare two DtHashKeys without knowing what type
// of DtHashKey subclass they actually are. You can add, remove and look
// up data by its DtHashKey.

class DtHashlist
{
public:
    // default constructor
    DtHashlist(int numBuckets = theNumBucketsDefault);

    // default constructor
    DtHashlist(DtList* list, int numBuckets = theNumBucketsDefault);

    // destructor
    virtual ~DtHashlist();

    // In the event that an item with key already exists, then add
    // will replace the data and move it to the location in list.
    // If your data was allocated, do not add unless you know key is
    // unique, or a memory leak will result.

    // Add something to the list. The actual pointer is stored, so you
should
    // make sure the pointer remains valid until after you remove the data
    // from the hash list.
    virtual DtListItem* add(const DtBaseHashKey& key, void* data);

    // Remove something from the list. Returns the data that was
    // associated with the key, so that you can cast it to the right type
    // and delete it.
    virtual void* remove(const DtBaseHashKey& key);

    // Lookup something in the list, return data pointer.
    virtual void* lookup(const DtBaseHashKey& key) const;
    virtual void* operator[] (const DtBaseHashKey& key) const;

    // Returns a DtList of void*'s which are the data that you have
    // added to the hashlist (independent of hash keys).
    virtual const DtList* list() const;

protected:
    // copy constructor does not copy the list
    DtHashlist(const DtHashlist& orig);

    // assignment operator does not copy the list
```

```
DtHashlist& operator=(const DtHashlist& orig);

// Lookup something in the list, return DtListItem
virtual DtListItem* itemLookup(const DtBaseHashKey& key) const;

// Determine the bucket from the hashkey
virtual DtList& bucketFromHashKey(const DtBaseHashKey& key) const;

// Lookup an item in a bucket
virtual DtListItem* bucketItemLookup(const DtBaseHashKey& key) const;

protected:

    static int theNumBucketsDefault;

protected:

    int myNumBuckets;
    const DtBoolean myListIsMine;
    DtList* myList;
    DtList* myBuckets;
};
typedef DtHashlist DtHashList;
```

hCollideDec.h

```
#if DtHLA

#include "interDecoder.h"

#include "chooseDefines.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_COLLISION_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtCollisionInteraction, paramName)

#define DtDEFINE_SIMPLE_COLLISION_PARAM_DECODER(paramName, netType, \
    mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER(DtCollisionDecoder, DtCollisionInterac- \
    tion, \
                                paramName, netType, mutator)

#define DtDEFINE_SIMPLE_COLLISION_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
        DtCollisionDecoder, DtCollisionInteraction, \
        paramName, netType, mutator, castExpr)

// class DtCollisionDecoder:
//
// Instances of DtCollisionDecoder are used to decode Collision
// Interactions from the network.

class DtCollisionDecoder : public DtInteractionDecoder
{
public:

    // constructor
    // Adds collision parameter decoders.
    DtCollisionDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtCollisionDecoder();

    // copy constructor
    DtCollisionDecoder(const DtCollisionDecoder& orig);

    // assignment operator
    DtCollisionDecoder& operator=(const DtCollisionDecoder& orig);

protected:

    DtDECLARE_COLLISION_PARAM_DECODER(CollidingObjectID);
    #if !DtrPR016
        DtDECLARE_COLLISION_PARAM_DECODER(IssuingObjectMass);
        DtDECLARE_COLLISION_PARAM_DECODER(IssuingObjectVelocity);
    #endif
    DtDECLARE_COLLISION_PARAM_DECODER(CollisionType);
    DtDECLARE_COLLISION_PARAM_DECODER(CollisionLocation);
    DtDECLARE_COLLISION_PARAM_DECODER(EventID);
    DtDECLARE_COLLISION_PARAM_DECODER(IssuingObjectID);
    #if DtrPR016
        DtDECLARE_COLLISION_PARAM_DECODER(CollidingObjectMass);
        DtDECLARE_COLLISION_PARAM_DECODER(CollidingObjectVelocity);
    #endif
};
```



```
#endif
```

hCollideEnc.h

```
#if DtHLA

#include "interEncoder.h"

#define DtDECLARE_COLLISION_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtCollisionInteraction, paramName)

#define DtDEFINE_SIMPLE_COLLISION_PARAM_ENCODER(paramName, netType, \
    inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtCollisionEncoder, DtCollisionInterac- \
    tion, \
                                paramName, netType, \
                                inspector)

// class DtCollisionEncoder:
//
// DtCollisionEncoder is a DtInteractionEncoder that registers
// parameter encoders for all of the COLLISION interaction parameters.

class DtCollisionEncoder : public DtInteractionEncoder
{
public:

    DtCollisionEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtCollisionEncoder();

    // copy constructor
    DtCollisionEncoder(const DtCollisionEncoder& orig);

    // assignment operator
    DtCollisionEncoder& operator=(const DtCollisionEncoder& orig);

protected:

    DtDECLARE_COLLISION_PARAM_ENCODER(CollidingObjectID);
#if !DtrPR016
    DtDECLARE_COLLISION_PARAM_ENCODER(IssuingObjectMass);
    DtDECLARE_COLLISION_PARAM_ENCODER(IssuingObjectVelocity);
#endif
    DtDECLARE_COLLISION_PARAM_ENCODER(CollisionType);
    DtDECLARE_COLLISION_PARAM_ENCODER(CollisionLocation);
    DtDECLARE_COLLISION_PARAM_ENCODER(EventID);
    DtDECLARE_COLLISION_PARAM_ENCODER(IssuingObjectID);
#if DtrPR016
    DtDECLARE_COLLISION_PARAM_ENCODER(CollidingObjectMass);
    DtDECLARE_COLLISION_PARAM_ENCODER(CollidingObjectVelocity);
#endif
};
#endif
```

hCollideInter.h

```

#if DtHLA

#include "hIntWEncDec.h"
#include "hostStructs.h"
#include "mathTypes.h"

#include "chooseDefines.h"

// class DtCollisionInteraction:
//
// Instances of DtCollisionInteraction are used to represent
// Collision Interactions in the RPR FOM.

// Collision interaction callback function signature.
typedef void (*DtCollisionInteractionCb)
    (DtCollisionInteraction* inter, void* usr);

class DtCollisionInteraction : public DtInteractionWithEncDec
{
public:

    // default constructor
    DtCollisionInteraction();

    // destructor
    virtual ~DtCollisionInteraction();

    // copy constructor
    DtCollisionInteraction(const DtCollisionInteraction& orig);

    // assignment operator
    DtCollisionInteraction& operator=(const DtCollisionInteraction& orig);

    // Print the interaction's data.
    virtual void printData() const;

    // Set/Get the issuer identifier.
    virtual void setIssueId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& issueId() const;

    // Set/Get the collide identifier.
    virtual void setCollideId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& collideId() const;

    // Set/Get the collision event id.
    virtual void setEventId(const DtEventID& id);
    virtual const DtEventID& eventId() const;

    // Set/Get the collision type.
    virtual void setCollisionType(DtCollisionType t);
    virtual DtCollisionType collisionType() const;

    // Set/Get the velocity.
    virtual void setVelocity(DtConstVector veloc);
    virtual DtConstVector velocity() const;

    // Set/Get the mass involved.
    virtual void setMass(DtFloat32 mass);
    virtual DtFloat32 mass() const;

    // Set/Get the entity coordinates.
    virtual void setEntityCoords(DtConstVector coords);
    virtual DtConstVector entityCoords() const;

```

```
// Name of the interaction (Not nec. the ClassName from the FOM).
virtual const char* name() const;

public:

    // Create a DtCollisionInteraction.
    // Caller is responsible for deletion.
    static DtInteraction* create();

    // Add (register)/Remove (deregister) function to be called when
    // interaction occurs.
    static void addCallback(DtExerciseConn* conn,
        DtCollisionInteractionCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtCollisionInteractionCb cb, void* usr);

protected:

    DtGlobalObjectDesignator myIssueId;
    DtGlobalObjectDesignator myCollideId;
    DtEventID myEventId;
    DtCollisionType myCollisionType;
    DtVector myVelocity;
    DtFloat32 myMass;
    DtVector myEntityCoords;
};
#endif
```

hDbiRadSigDec.h

```

#if DtHLA

#include "chooseDefines.h"
#include "interDecoder.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtSignalInteraction, paramName)

#define DtDEFINE_SIMPLE_DATABASE_RADIO_SIGNAL_PARAM_DECODER(paramName, \
    netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER(DtDatabaseIndexRadioSignalDecoder, \
        DtSignalInteraction, paramName, netType, mutator)

#define DtDEFINE_SIMPLE_DATABASE_RADIO_SIGNAL_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
        DtDatabaseIndexRadioSignalDecoder, DtSignalInteraction, \
        paramName, netType, mutator, castExpr)

// class DtDatabaseIndexRadioSignalDecoder:
//
// Instances of DtDatabaseIndexRadioSignalDecoder are

class DtDatabaseIndexRadioSignalDecoder : public DtInteractionDecoder
{
public:
    // default constructor
    DtDatabaseIndexRadioSignalDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtDatabaseIndexRadioSignalDecoder();

    // copy constructor
    DtDatabaseIndexRadioSignalDecoder(
        const DtDatabaseIndexRadioSignalDecoder& orig);

    // assignment operator
    DtDatabaseIndexRadioSignalDecoder& operator=(
        const DtDatabaseIndexRadioSignalDecoder& orig);

protected:
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_DECODER(HostRadioID);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_DECODER(DatabaseIndex);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_DECODER(Duration);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_DECODER(StartOffset);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_DECODER(TacticalDataLinkType);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_DECODER(TDLMessageCount);
};
#endif // DtHLA

```

hDbiRadSigEnc.h

```
#if DtHLA

#include "chooseDefines.h"
#include "interEncoder.h"

#define DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtSignalInteraction, paramName)

#define DtDEFINE_SIMPLE_DATABASE_RADIO_SIGNAL_PARAM_ENCODER(paramName, \
    netType, inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtDatabaseIndexRadioSignalEncoder, \
    DtSignalInteraction, paramName, netType, inspector)

// class DtDatabaseIndexRadioSignalEncoder:
//
// Instances of DtDatabaseIndexRadioSignalEncoder are
class DtDatabaseIndexRadioSignalEncoder : public DtInteractionEncoder
{
public:
    // default constructor
    DtDatabaseIndexRadioSignalEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtDatabaseIndexRadioSignalEncoder();

    // copy constructor
    DtDatabaseIndexRadioSignalEncoder(
        const DtDatabaseIndexRadioSignalEncoder& orig);

    // assignment operator
    DtDatabaseIndexRadioSignalEncoder& operator=(
        const DtDatabaseIndexRadioSignalEncoder& orig);

protected:
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_ENCODER(HostRadioID);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_ENCODER(DatabaseIndex);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_ENCODER(Duration);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_ENCODER(StartOffset);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_ENCODER(TacticalDataLinkType);
    DtDECLARE_DATABASE_RADIO_SIGNAL_PARAM_ENCODER(TDLMessageCount);
};
#endif //DtHLA
```

hDesignatrPub.h

```

#if DtHLA

#include "hObjPub.h"

// class DtDesignatorPublisher:
//
// Instances of DtDesignatorPublisher are used for local or source simula-
// tions
// of designators, (i.e. these entities are not dependent on data being
// produced elsewhere on the network for their existence). These
// designators send data to the network.

class DtDesignatorPublisher : public DtObjectPublisher
{
public:

    // Function prototype for functions that create DtDesignatorRepositories
    typedef DtDesignatorRepository* (*DtStateRepCreator)();

public:

    // constructor
    DtDesignatorPublisher(RTI::ObjectClassHandle h,
        DtExerciseConn* conn, const char* name = NULL);

    // constructor
    DtDesignatorPublisher(DtExerciseConn* conn, const char* name = NULL);

    // destructor
    virtual ~DtDesignatorPublisher();

    // Returns a pointer to the DtDesignatorRepository - through
    // which you can set or inspect current state information.
    virtual DtDesignatorRepository* designatorStateRep();

    // Synonym for designatorStateRep()
    virtual DtDesignatorRepository* dsr();

    // Returns the object class handle to use for this class.
    virtual RTI::ObjectClassHandle classToUse() const;

public:

    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

protected:

    // copy constructor
    DtDesignatorPublisher(const DtDesignatorPublisher& orig);

    // copy constructor
    DtDesignatorPublisher& operator=
        (const DtDesignatorPublisher& orig);

protected:

    static DtStateRepCreator theStateRepCreator;
};
#endif //DtDesignatorPublisher_H_

```

hDetDec.h

```
#if DtHLA

#include "interDecoder.h"

#include "chooseDefines.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_DETONATION_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtDetonationInteraction, paramName)

#define DtDEFINE_SIMPLE_DETONATION_PARAM_DECODER( \
    paramName, netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER( \
        DtDetonationDecoder, DtDetonationInteraction, \
        paramName, netType, mutator)

#define DtDEFINE_SIMPLE_DETONATION_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
        DtDetonationDecoder, DtDetonationInteraction, \
        paramName, netType, mutator, castExpr)

// class DtDetonationDecoder:
//
// Instances of DtDetonationDecoder are used to decode Detonation
// Interactions from the network.

class DtDetonationDecoder : public DtInteractionDecoder
{
public:

    // constructor
    DtDetonationDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtDetonationDecoder();

    // copy constructor
    DtDetonationDecoder(const DtDetonationDecoder& orig);

    // assignment operator
    DtDetonationDecoder& operator=(const DtDetonationDecoder& orig);

protected:

    // Individual parameter decoders

#if !DtRPR016
    DtDECLARE_DETONATION_PARAM_DECODER(ArticulatedPartsArray);
#else
    DtDECLARE_DETONATION_PARAM_DECODER(ArticulatedParametersArray);
    DtDECLARE_DETONATION_PARAM_DECODER(ArticulatedParametersCount);
#endif
    DtDECLARE_DETONATION_PARAM_DECODER(DetonationLocation);
    DtDECLARE_DETONATION_PARAM_DECODER(DetonationResult);
    DtDECLARE_DETONATION_PARAM_DECODER(EventID);
    DtDECLARE_DETONATION_PARAM_DECODER(FiringObjectID);
    DtDECLARE_DETONATION_PARAM_DECODER(FinalVelocityVector);
    DtDECLARE_DETONATION_PARAM_DECODER(FuseType);
    DtDECLARE_DETONATION_PARAM_DECODER(MunitionObjectID);
```



```
DtDECLARE_DETONATION_PARAM_DECODER(MunitionType);
DtDECLARE_DETONATION_PARAM_DECODER(QuantityFired);
DtDECLARE_DETONATION_PARAM_DECODER(RateOfFire);
DtDECLARE_DETONATION_PARAM_DECODER(RelativeDetonationLocation);
DtDECLARE_DETONATION_PARAM_DECODER(TargetObjectID);
DtDECLARE_DETONATION_PARAM_DECODER(WarheadType);
};
#endif
```

hDetEnc.h

```
#if DtHLA

#include "interEncoder.h"

#define DtDECLARE_DETONATION_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtDetonationInteraction, paramName)

#define DtDEFINE_SIMPLE_DETONATION_PARAM_ENCODER(paramName, netType, \
    inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtDetonationEncoder, DtDetonationInterac- \
    tion, \
                                paramName, netType, \
                                inspector)

// class DtDetonationEncoder:
//
// DtDetonationEncoder is a DtInteractionEncoder that registers
// parameter encoders for all of the DETONATION interaction parameters.

class DtDetonationEncoder : public DtInteractionEncoder
{
public:

    DtDetonationEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtDetonationEncoder();

    // copy constructor
    DtDetonationEncoder(const DtDetonationEncoder& orig);

    // assignment operator
    DtDetonationEncoder& operator=(const DtDetonationEncoder& orig);

protected:

#if !DtrPR016
    DtDECLARE_DETONATION_PARAM_ENCODER(ArticulatedPartsArray);
#else
    DtDECLARE_DETONATION_PARAM_ENCODER(ArticulatedParametersArray);
    DtDECLARE_DETONATION_PARAM_ENCODER(ArticulatedParametersCount);
#endif
    DtDECLARE_DETONATION_PARAM_ENCODER(DetonationLocation);
    DtDECLARE_DETONATION_PARAM_ENCODER(DetonationResult);
    DtDECLARE_DETONATION_PARAM_ENCODER(EventID);
    DtDECLARE_DETONATION_PARAM_ENCODER(FiringObjectID);
    DtDECLARE_DETONATION_PARAM_ENCODER(FinalVelocityVector);
    DtDECLARE_DETONATION_PARAM_ENCODER(FuseType);
    DtDECLARE_DETONATION_PARAM_ENCODER(MunitionObjectID);
    DtDECLARE_DETONATION_PARAM_ENCODER(MunitionType);
    DtDECLARE_DETONATION_PARAM_ENCODER(QuantityFired);
    DtDECLARE_DETONATION_PARAM_ENCODER(RateOfFire);
    DtDECLARE_DETONATION_PARAM_ENCODER(RelativeDetonationLocation);
    DtDECLARE_DETONATION_PARAM_ENCODER(TargetObjectID);
    DtDECLARE_DETONATION_PARAM_ENCODER(WarheadType);
};
#endif
```

hDetInter.h

```

#if DtHLA

#include "hIntWEncDec.h"
#include "hostStructs.h"
#include "mathTypes.h"
#include "artPartList.h"

#include "chooseDefines.h"

// class DtDetonationInteraction:
//
// Instances of DtDetonationInteraction are used to represent
// Detonation Interactions in the RPR ROM.

// Detonation interaction callback function signature.
typedef void (*DtDetonationInteractionCb) (DtDetonationInteraction* inter,
void* usr);

class DtDetonationInteraction : public DtInteractionWithEncDec
{
public:

    // default constructor
    DtDetonationInteraction();

    // Constructor
    // Fills in common fields with data from Fire Interaction
    DtDetonationInteraction(const DtFireInteraction& fire);

    // destructor
    virtual ~DtDetonationInteraction();

    // copy constructor
    DtDetonationInteraction(const DtDetonationInteraction& orig);

    // assignment operator
    DtDetonationInteraction& operator=(const DtDetonationInteraction& orig);

    // Print the interaction's data.
    virtual void printData() const;

    // Set/Get the attacker identifier.
    virtual void setAttackerId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& attackerId() const;

    // Set/Get the target identifier.
    virtual void setTargetId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& targetId() const;

    // Set/Get the munition identifier.
    virtual void setMunitionId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& munitionId() const;

    // Set/Get the detonation event id.
    virtual void setEventId(const DtEventID& id);
    virtual const DtEventID& eventId() const;

    // Set/Get the velocity.
    virtual void setVelocity(DtConstVector velocity);
    virtual DtConstVector velocity() const;

    // Set/Get the location of the detonation in world coordinates.
    virtual void setWorldLocation(DtConstVector location);

```

```
virtual DtConstVector worldLocation() const;

// Set/Get the location of the detonation in entity coordinates.
virtual void setEntityLocation(DtConstVector location);
virtual DtConstVector entityLocation() const;

// Set/Get the detonation result.
virtual void setResult(DtDetonationResult result);
virtual DtDetonationResult result() const;

// Set/Get the detonator fuze type.
virtual void setFuseType(DtDetonatorFuze fuse);
virtual DtDetonatorFuze fuseType() const;

// Set/Get the munition type.
virtual void setMunitionType(const DtEntityType& munType);
virtual const DtEntityType& munitionType() const;

// Set/Get the quantity.
virtual void setQuantity(int quant);
virtual int quantity() const;

// Set/Get the rate.
virtual void setRate(int rate);
virtual int rate() const;

// Set/Get the warhead type.
virtual void setWarheadType(DtWarheadType type);
virtual DtWarheadType warheadType() const;

// Set/Get ArticulatedParts
virtual void initArtParts(const DtArtPartSpec* specList);
virtual const DtArtPartSpec* artPartSpecs() const;

// Get number of articulation parameters
virtual int numArtParams() const;

// Set/Get articulation parameter values by part type
// and parameter type.
virtual int setArtParamVal(int partType, int param, DtFloat32 value);
virtual int getArtParamVal(
    int partType, int param, DtFloat32* value) const;

// Get a pointer to an DtArtPartList containing the interaction's
// articulated part information.
virtual DtArtPartList* artPartList();
virtual const DtArtPartList* artPartList() const;

// Name of the interaction (Not nec. the ClassName from the FOM).
virtual const char* name() const;

public:

    // Create a DtDetonationInteraction.
    // Caller is responsible for deletion.
    static DtInteraction* create();

    // Add (register)/Remove (deregister) function to be called when
    // interaction occurs.
    static void addCallback(DtExerciseConn* conn,
        DtDetonationInteractionCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtDetonationInteractionCb cb, void* usr);

protected:

    DtVector myWorldLocation;
```

```
DtDetonationResult myResult;  
DtEventID myEventId;  
DtGlobalObjectDesignator myAttackerId;  
DtVector myVelocity;  
DtDetonatorFuze myFuseType;  
DtGlobalObjectDesignator myMunitionId;  
DtEntityType myMunitionType;  
int myQuantity;  
int myRate;  
DtVector myEntityLocation;  
DtGlobalObjectDesignator myTargetId;  
DtWarheadType myWarheadType;  
DtArtPartList myPartList;  
};  
#endif
```

hEmitrBeamPub.h

```
#if DtHLA

#ifndef DtEmitterBeamPublisher_H_
#define DtEmitterBeamPublisher_H_

#include "hObjPub.h"

// class DtEmitterBeamPublisher:
//
// Instances of DtEmitterBeamPublisher are used for local or source simulations
// of emitter beams, (i.e. these entities are not dependent on data being
// produced elsewhere on the network for their existence). These
// emitter beams send data to the network.

class DtEmitterBeamPublisher : public DtObjectPublisher
{
public:

    // Function prototype for functions that create DtBeamRepositories
    typedef DtEmitterBeamRepository* (*DtStateRepCreator)();

public:

    // constructor. NULL bsr means create your own.
    DtEmitterBeamPublisher(
        RTI::ObjectClassHandle h,
        DtExerciseConn* conn,
        DtEmitterBeamRepository* bsr = NULL,
        const char* name = NULL);

    // constructor. NULL bsr means create your own.
    DtEmitterBeamPublisher(
        DtExerciseConn* conn,
        DtEmitterBeamRepository* bsr = NULL,
        const char* name = NULL);

    // destructor
    virtual ~DtEmitterBeamPublisher();

    // Returns a pointer to the DtEmitterBeamRepository - through
    // which you can set or inspect current state information.
    virtual DtEmitterBeamRepository* emitrBeamRep();

    // Synonym for emitrBeamRep()
    virtual DtEmitterBeamRepository* bsr();

    // Returns the object class handle to use for this class.
    virtual RTI::ObjectClassHandle classToUse(
        DtEmitterBeamRepository* bsr) const;

public:

    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

protected:

    // copy constructor
    DtEmitterBeamPublisher(const DtEmitterBeamPublisher& orig);

    // assignment operator
    DtEmitterBeamPublisher& operator=
```

```
        (const DtEmitterBeamPublisher& orig);  
protected:  
    DtBoolean myOwnBsr;  
protected:  
    static DtStateRepCreator theStateRepCreator;  
};  
#endif    //DtEmitterBeamPublisher_H_
```

hEmittSysPub.h

```
#if DtHLA

#ifndef DtEmitterSystemPublisher_H_
#define DtEmitterSystemPublisher_H_

#include "hObjPub.h"

// class DtEmitterSystemPublisher:
//
// Instances of DtEmitterSystemPublisher are used for local or source
// simulations of emitter systems, (i.e. these entities are not dependent
// on data being produced elsewhere on the network for their existence).
// These emitter systems send data to the network.

class DtEmitterSystemPublisher : public DtObjectPublisher
{
public:

    // Function prototype for functions that create DtEmitterSystemReposito-
    // ries
    typedef DtEmitterSystemRepository* (*DtStateRepCreator)();

public:

    // constructor
    DtEmitterSystemPublisher(
        RTI::ObjectClassHandle h, DtExerciseConn* conn,
        const char* name = NULL);

    // constructor
    DtEmitterSystemPublisher(DtExerciseConn* conn,
        const char* name = NULL);

    // destructor
    virtual ~DtEmitterSystemPublisher();

    // Returns a pointer to the DtEmitterSystemRepository - through
    // which you can set or inspect current state information.
    virtual DtEmitterSystemRepository* emitterSystemRep();

    // Synonym for emitterSystemRep()
    virtual DtEmitterSystemRepository* esr();

    // Returns the object class handle to use for this class.
    virtual RTI::ObjectClassHandle classToUse() const;

    // syncBeamList and update beam objects as well as main system object.
    virtual void tick();

public:

    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

protected:

    // copy constructor
    DtEmitterSystemPublisher(const DtEmitterSystemPublisher& orig);

    // assignment operator
    DtEmitterSystemPublisher& operator=
        (const DtEmitterSystemPublisher& orig);
};

#endif
```



```
// Update myBeamPubs so that it has a publisher for all beamSRs
// in my systemSR, but no others. Called by tick() so that
// all beamPubs can be ticked after systemSR has been updated.
virtual void syncBeamList();

// Returns the publisher responsible for publishing the indicated BSR,
// NULL if there is none.
virtual DtEmitterBeamPublisher* publisherForBsr(
    DtEmitterBeamRepository* bsr);

// Return number of beam publishers
virtual int numBeams() const;

protected:

    // List of DtEmitterBeamPubs
    DtList myBeamPubs;

protected:

    static DtStateRepCreator theStateRepCreator;
};
#endif //DtEmitterSystemPublisher_H_
```

hEncARSigDec.h

```
#if DtHLA

#include "chooseDefines.h"
#include "interDecoder.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtSignalInteraction, paramName)

#define DtDEFINE_SIMPLE_ENC_AUDIO_RADIO_SIGNAL_PARAM_DECODER(paramName, \
    netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER(DtEncodedAudioRadioSignalDecoder, \
        DtSignalInteraction, paramName, netType, mutator)

#define DtDEFINE_SIMPLE_ENC_AUDIO_RADIO_SIGNAL_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
        DtEncodedAudioRadioSignalDecoder, DtSignalInteraction, \
        paramName, netType, mutator, castExpr)

// class DtEncodedAudioRadioSignalDecoder:
//
// Instances of DtEncodedAudioRadioSignalDecoder are

class DtEncodedAudioRadioSignalDecoder : public DtInteractionDecoder
{
public:
    // default constructor
    DtEncodedAudioRadioSignalDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtEncodedAudioRadioSignalDecoder();

    // copy constructor
    DtEncodedAudioRadioSignalDecoder(
        const DtEncodedAudioRadioSignalDecoder& orig);

    // assignment operator
    DtEncodedAudioRadioSignalDecoder& operator=(
        const DtEncodedAudioRadioSignalDecoder& orig);

protected:
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_DECODER(HostRadioID);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_DECODER(EncodingType);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_DECODER(SampleRate);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_DECODER(SampleCount);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_DECODER(SignalDataLength);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_DECODER(SignalData);
};
#endif // DtHLA
```

hEncARSigEnc.h

```

#if DtHLA

#include "chooseDefines.h"
#include "interEncoder.h"

#define DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtSignalInteraction, paramName)

#define DtDEFINE_SIMPLE_ENC_AUDIO_RADIO_SIGNAL_PARAM_ENCODER(paramName, \
    netType, inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtEncodedAudioRadioSignalEncoder, \
    DtSignalInteraction, paramName, netType, inspector)

// class DtEncodedAudioRadioSignalEncoder:
//
// Instances of DtEncodedAudioRadioSignalEncoder are
class DtEncodedAudioRadioSignalEncoder : public DtInteractionEncoder
{
public:

    // default constructor
    DtEncodedAudioRadioSignalEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtEncodedAudioRadioSignalEncoder();

    // copy constructor
    DtEncodedAudioRadioSignalEncoder(
        const DtEncodedAudioRadioSignalEncoder& orig);

    // assignment operator
    DtEncodedAudioRadioSignalEncoder& operator=(
        const DtEncodedAudioRadioSignalEncoder& orig);

protected:

    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_ENCODER(HostRadioID);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_ENCODER(EncodingType);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_ENCODER(SampleRate);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_ENCODER(SampleCount);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_ENCODER(SignalDataLength);
    DtDECLARE_ENC_AUDIO_RADIO_SIGNAL_PARAM_ENCODER(SignalData);
};
#endif // DtHLA

```

hEntityPub.h

```
#if DtHLA

#include "hObjPub.h"
#include "threshold.h"

#ifdef __GNUC__ // g++
// Note: Here due to bug in g++ when using class static member function
// as a default argument. Don't call these otherwise.
#ifdef __GNUC__ // g++
#else
#undef gGuiseSameAsType
#endif
#endif

#endif
```

hExerciseConn.h

```

#if DtHLA

#include "baseExConn.h"
#include "RTI.hh"
#include "vlTime.h"
#include "hostStructs.h"
#include "callbacks.h"
#include "tStampType.h"
#include "vrlRtiAmb.h"

#include "chooseDefines.h"

typedef DtVrlFederateAmbassador* (*DtFedAmbCreator)();
typedef DtVrlRtiAmbassador* (*DtRtiAmbCreator)();
typedef DtFomMapper* (*DtFomMapperCreator)(DtExerciseConn*);

// class DtExerciseConn:
//
// Instances of DtExerciseConn are used to provide an application's
// interface or connection to the network in an HLA environment (as
// opposed to DIS).

class DtExerciseConn : public DtBaseExerciseConn
{
public:
    // constructor
    DtExerciseConn(const char* execName, const char* federateName);

#if !DtRTI103
    // constructor
    DtExerciseConn(const char* execName, const char* federateName,
        const char* fedFileName);
#endif

    // destructor
    virtual ~DtExerciseConn();

    // Send a state message to the network.
    virtual void send(const DtStateMsg& msg);

    // Send an interaction to the network.
    virtual void send(const DtInteraction& msg);

    // Send a state message to the network which has been
    // time stamped with the current time.
    virtual void sendStamped(const DtStateMsg& msg);

    // Send an interaction to the network which has been
    // time stamped with the current time.
    virtual void sendStamped(const DtInteraction& msg);

    // Read and process messages from the network until either no more
    // messages, or for timeout seconds, whichever happens first.
    // Will read nReadPerTimeCheck messages at a time before checking
    // for timeout.
    // Returns number of incoming events which have been processed.
    virtual int drainInput(DtTime timeout=-1, int nReadPerTimeCheck=100);

#if DtRTI103
    // Get a unique id for use in registering objects with the federation.

```

```
virtual RTI::ObjectID nextId();
#endif

// Set objectID to use in the next event ID
virtual void setObjectIdForEventId(const DtGlobalObjectDesignator& id);

// Get a unique event ID
virtual const DtEventID& nextEventId();

// Get the RTI ambassador.
virtual DtVrIRtiAmbassador* rtiAmb();

// Get the federate ambassador.
virtual DtVrIFederateAmbassador* fedAmb();

// Get the FOM
virtual DtFom* fom();

// Get the interaction factory.
virtual DtInteractionFactory* interactionFactory();

// Get the object factory.
virtual DtHlaObjectFactory* objectFactory();

// Get the Fom Mapper
virtual DtFomMapper* fomMapper();

// Get the reflectedObjectManager
virtual DtReflectedHlaObjectManager* reflectedObjectManager();

// Get the localObjectManager
virtual DtLocalHlaObjectManager* localObjectManager();

// Set/Get time stamp type (DtTimeStampAbsolute or DtTimeStampRelative)
virtual void setTimeStampType(DtTimeStampType relOrAbs);
virtual DtTimeStampType timeStampType() const;

// Add an interaction callback which is called with the user data
// when the specified interaction is received.
virtual void addInteractionCallback(RTI::InteractionClassHandle h,
    DtReceiveInteractionCb cb,
    void* usr);

// Remove an interaction callback.
// Note: The user data is significant for identifying the callback
// and needs to be the same as when the callback was added.
virtual void removeInteractionCallback(RTI::InteractionClassHandle h,
    DtReceiveInteractionCb cb,
    void* usr);

// Add a discoverObject callback which is called with the user data
// when the specified interaction is received.
virtual void addDiscoverObjectCallback(RTI::ObjectClassHandle h,
    DtDiscoverObjectCb cb,
    void* usr);

// Remove a discoverObject callback.
// Note: The user data is significant for identifying the callback
// and needs to be the same as when the callback was added.
virtual void removeDiscoverObjectCallback(RTI::ObjectClassHandle h,
    DtDiscoverObjectCb cb,
    void* usr);

// Add a removeObject callback which is called with the user data
// when the specified interaction is received.
virtual void addRemoveObjectCallback(RTI::ObjectHandle id,
    DtRemoveObjectCb cb,
```

```

        void* usr);

// Remove a removeObject callback.
// Note: The user data is significant for identifying the callback
// and needs to be the same as when the callback was added.
virtual void removeObjectCallback(RTI::ObjectHandle id,
    DtRemoveObjectCb cb,
    void* usr);

// Set/Get flag that determines whether or not the DtExerciseConn
// destructor will attempt to destroy the federation execution.
// Default is true.
virtual void setDestroyFedExecFlag(DtBoolean flag);
virtual DtBoolean destroyFedExecFlag();

public:

// Set/get the function to be used to create a federate ambassador.
// Call before creating a DtExerciseConn if you want to use
// something other than a DtVrlFedAmbassador.
static void setFedAmbCreator(DtFedAmbCreator creator);
static DtFedAmbCreator fedAmbCreator();

// Set/get function to be used to create an RTI ambassador.
// Call before creating a DtExerciseConn if you want to use
// something other than a DtDirectRtiAmbassador.
static void setRtiAmbCreator(DtRtiAmbCreator creator);
static DtRtiAmbCreator rtiAmbCreator();

// Set/Get FomMapper creator - use to create a fomMapper. Call set
// before creating a DtExConn if you want it to use something other
// than the default RPR FOM mapper.
static void setFomMapperCreator(DtFomMapperCreator creator);
static DtFomMapperCreator fomMapperCreator();

// Returns a string indicating the VR-Link version number
static const char* vrlinkVersion();

protected:

// Note: The copy constructor and assignment operator are protected
// because it is not clear what these would mean.
// An execution name must be unique.
// These should not be called.

// copy constructor
DtExerciseConn(const DtExerciseConn& orig);

// assignment operator
DtExerciseConn& operator=(const DtExerciseConn& orig);

// Not virtual since called by ctor. NULL fedFileName means
// use execName.fed
void init(const char* execName, const char* federateName,
    const char* fedFileName);

// Not virtual since called by ctor. Initializes myFom from
// fed file. NULL fedFileName means use execName.fed
void initFom(const char* execName, const char* fedFileName);

// Return time to send with outgoing messages
// (default is real time if absolute timestamping, sim time
// otherwise.
virtual DtTime currentTimeForStamping();

// Create the RTI ambassador. Not virtual since called by ctor.
// Generates FatalError if unsuccessful.

```

```
void createRtiAmb();

// Create the federation execution. Not virtual since called by ctor.
// Generates FatalError if unsuccessful.
void createFedEx();

// Create the federate ambassador. Not virtual since called by ctor.
// Generates FatalError if unsuccessful.
void createFedAmb();

// Join the federation execution. Not virtual since called by ctor.
// Generates FatalError if unsuccessful.
void joinFedEx();

protected:

char* myExecName;
char* myFedFileName;
char* myFedName;
RTI::FederateHandle myFedHandle;
DtVrlRtiAmbassador* myRtiAmb;
DtVrlFederateAmbassador* myFedAmb;

DtFom* myFom;

DtReflectedHlaObjectManager* myReflectedObjectManager;
DtLocalHlaObjectManager* myLocalObjectManager;
DtInteractionManager* myInteractionManager;

DtFomMapper* myFomMapper;

DtEventID myCurrentEventId;
DtTimeStampType myRelOrAbs;

DtBoolean myDestroyFedExFlag;

protected:

static DtFedAmbCreator theFedAmbCreator;
static DtRtiAmbCreator theRtiAmbCreator;
static DtFomMapperCreator theFomMapperCreator;
};
#endif
```


hFireDec.h

```

#if DtHLA

#include "interDecoder.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_FIRE_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtFireInteraction, paramName)

#define DtDEFINE_SIMPLE_FIRE_PARAM_DECODER(paramName, netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER(DtFireDecoder, DtFireInteraction, \
                                   paramName, netType, mutator)

#define DtDEFINE_SIMPLE_FIRE_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
        DtFireDecoder, DtFireInteraction, \
        paramName, netType, mutator, castExpr)

// class DtFireDecoder:
//
// Instances of DtFireDecoder are used to decode Fire Interactions
// from the network.

class DtFireDecoder : public DtInteractionDecoder
{
public:
    // constructor
    DtFireDecoder(DtExerciseConn* exConn,
                  DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtFireDecoder();

    // copy constructor
    DtFireDecoder(const DtFireDecoder& orig);

    // assignment operator
    DtFireDecoder& operator=(const DtFireDecoder& orig);

protected:
    // Individual parameter decoders

    DtDECLARE_FIRE_PARAM_DECODER(EventID);
    DtDECLARE_FIRE_PARAM_DECODER(FireControlSolutionRange);
    DtDECLARE_FIRE_PARAM_DECODER(FireMissionIndex);
    DtDECLARE_FIRE_PARAM_DECODER(FiringLocation);
    DtDECLARE_FIRE_PARAM_DECODER(FiringObjectID);
    DtDECLARE_FIRE_PARAM_DECODER(FuseType);
    DtDECLARE_FIRE_PARAM_DECODER(InitialVelocityVector);
    DtDECLARE_FIRE_PARAM_DECODER(MunitionObjectID);
    DtDECLARE_FIRE_PARAM_DECODER(MunitionType);
    DtDECLARE_FIRE_PARAM_DECODER(QuantityFired);
    DtDECLARE_FIRE_PARAM_DECODER(RateOfFire);
    DtDECLARE_FIRE_PARAM_DECODER(TargetObjectID);
    DtDECLARE_FIRE_PARAM_DECODER(WarheadType);
};
#endif

```

hFireEnc.h

```
#if DtHLA

#include "interEncoder.h"

#define DtDECLARE_FIRE_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtFireInteraction, paramName)

#define DtDEFINE_SIMPLE_FIRE_PARAM_ENCODER(paramName, netType, inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtFireEncoder, DtFireInteraction, \
                                   paramName, netType, \
                                   inspector)

// class DtFireEncoder:
//
// DtFireEncoder is a DtInteractionEncoder that registers
// parameter encoders for all of the FIRE interaction parameters.

class DtFireEncoder : public DtInteractionEncoder
{
public:

    DtFireEncoder(DtExerciseConn* exConn,
                  DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtFireEncoder();

    // copy constructor
    DtFireEncoder(const DtFireEncoder& orig);

    // assignment operator
    DtFireEncoder& operator=(const DtFireEncoder& orig);

protected:

    DtDECLARE_FIRE_PARAM_ENCODER(EventID);
    DtDECLARE_FIRE_PARAM_ENCODER(FireControlSolutionRange);
    DtDECLARE_FIRE_PARAM_ENCODER(FireMissionIndex);
    DtDECLARE_FIRE_PARAM_ENCODER(FiringLocation);
    DtDECLARE_FIRE_PARAM_ENCODER(FiringObjectID);
    DtDECLARE_FIRE_PARAM_ENCODER(FuseType);
    DtDECLARE_FIRE_PARAM_ENCODER(InitialVelocityVector);
    DtDECLARE_FIRE_PARAM_ENCODER(MunitionObjectID);
    DtDECLARE_FIRE_PARAM_ENCODER(MunitionType);
    DtDECLARE_FIRE_PARAM_ENCODER(QuantityFired);
    DtDECLARE_FIRE_PARAM_ENCODER(RateOfFire);
    DtDECLARE_FIRE_PARAM_ENCODER(TargetObjectID);
    DtDECLARE_FIRE_PARAM_ENCODER(WarheadType);
};
#endif
```

hFireInter.h

```

#if DtHLA

#include "hIntWEncDec.h"
#include "hostStructs.h"
#include "mathTypes.h"

#include "chooseDefines.h"

// class DtFireInteraction:
//
// Instances of DtFireInteraction are used to represent
// Fire Interactions in the RPR FOM.

// Fire interaction callback function signature.
typedef void (*DtFireInteractionCb)(DtFireInteraction* inter, void* usr);

class DtFireInteraction : public DtInteractionWithEncDec
{
public:
    // default constructor
    DtFireInteraction();

    // destructor
    virtual ~DtFireInteraction();

    // copy constructor
    DtFireInteraction(const DtFireInteraction& orig);

    // assignment operator
    DtFireInteraction& operator=(const DtFireInteraction& orig);

    // Print the interaction's data.
    virtual void printData() const;

    // Set/Get the attacker identifier.
    virtual void setAttackerId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& attackerId() const;

    // Set/Get the target identifier.
    virtual void setTargetId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& targetId() const;

    // Set/Get the munition identifier.
    virtual void setMunitionId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& munitionId() const;

    // Set/Get the event id.
    virtual void setEventId(const DtEventID& id);
    virtual const DtEventID& eventId() const;

    // Set/Get the fire mission index.
    virtual void setFireMissionIndex(unsigned long i);
    virtual unsigned long fireMissionIndex() const;

    // Set/Get the location in world coordinates.
    virtual void setLocation(DtConstVector location);
    virtual DtConstVector location() const;

    // Set/Get the velocity.
    virtual void setVelocity(DtConstVector velocity);
    virtual DtConstVector velocity() const;

```

```
// Set/Get the range.
virtual void setRange(double range);
virtual double range() const;

// Set/Get the detonator fuze type.
virtual void setFuseType(DtDetonatorFuze f);
virtual DtDetonatorFuze fuseType() const;

// Set/Get the munition type.
virtual void setMunitionType(const DtEntityType& t);
virtual const DtEntityType& munitionType() const;

// Set/Get the quantity.
virtual void setQuantity(int q);
virtual int quantity() const;

// Set/Get the rate.
virtual void setRate(int r);
virtual int rate() const;

// Set/Get the warhead type.
virtual void setWarheadType(DtWarheadType t);
virtual DtWarheadType warheadType() const;

// Name of the interaction (Not nec. the ClassName from the FOM).
virtual const char* name() const;

public:

    // Create a DtFireInteraction.
    // Caller is responsible for deletion.
    static DtInteraction* create();

    // Add (register)/Remove (deregister) function to be called when
    // interaction occurs.
    static void addCallback(DtExerciseConn* conn,
        DtFireInteractionCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtFireInteractionCb cb, void* usr);

protected:

    DtGlobalObjectDesignator myAttackerId;
    DtGlobalObjectDesignator myTargetId;
    DtGlobalObjectDesignator myMunitionId;
    DtEventID myEventId;
    unsigned long myFireMissionIndex;
    DtVector myLocation;
    DtVector myVelocity;
    double myRange;
    DtDetonatorFuze myFuseType;
    DtEntityType myMunitionType;
    int myQuantity;
    int myRate;
    DtWarheadType myWarheadType;

protected:

    static char* theClassName;
};
#endif
```

hlInteraction.h

```

#if DtHLA

#include "RTI.hh"
#include "MachineTypes.h"
#include "recIntrCbInfo.h"
#include "vlTime.h"
#include "tStampType.h"
#include "str.h"

#include "chooseDefines.h"

// class DtInteraction:
//
// DtInteraction is an abstract base class for representing HLA
// interactions (or events) (as opposed to HLA objects).

class DtInteraction
{
public:

    // default constructor
    DtInteraction();

    // destructor
    virtual ~DtInteraction();

    // copy constructor
    DtInteraction(const DtInteraction& orig);

    // assignment operator
    DtInteraction& operator=(const DtInteraction& orig);

    // Get the parameter handle value pair set for the interaction.
    virtual const RTI::ParameterHandleValuePairSet& phvps() const = 0;

    // Set contents from a parameter handle value pair set
    virtual void setFromPhvps(
        const RTI::ParameterHandleValuePairSet& pvList) = 0;

    // Sets the interaction's ExerciseConn and class descriptor. If
    // classDesc is NULL, derived class' versions choose a reasonable
    default.
    // Without being told what ExConn or class descriptor to use,
    // many of the HLA-specific functions won't work properly.
    // However, it is usually not necessary for application
    // code to call this function, since it is called by
    // DtExerciseConn::send on the outgoing side, and by the
    // DtInteractionFactory on the incoming side.
    virtual void setExConn(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc = NULL);

    // Get the interaction class handle. Returns 0 if no classDesc.
    virtual RTI::InteractionClassHandle interactionClassHandle() const;

    // Get the interaction class name. Returns NULL if no classDesc.
    virtual char* interactionClassName() const;

    // Get the number of parameters. Returns 0 if no classDesc.
    virtual int numParameters() const;

    // Returns DtTrue if the interaction class is needed by the federation
    // DtFalse otherwise. Returns DtFalse if no classDesc.
    virtual DtBoolean neededByFederation() const;

```

```
// Print the interaction's host rep data, including header
virtual void print() const;

// Print the header info
virtual void printHeader() const;

// Print the interaction's host rep data.
virtual void printData() const;

// Print the header followed by the interaction's parameters.
virtual void printParams(DtBoolean withHex) const;

// Name of the interaction (Not nec. the ClassName from the FOM).
virtual const char* name() const = 0;

// Get the current exercise connection
virtual DtExerciseConn* exerciseConn() const;

// Get the current class descriptor we're using for this
// interaction.
virtual DtInterClassDesc* classDesc() const;

// Set/Get the user-defined tag.
virtual void setTag(const DtString& tag);
virtual const DtString& tag() const;

// Set the time of validity of the message, and whether this
// represents relative or absolute time. Time is in seconds, but only
// time modulo an hour is stored.
virtual void setTimeStamp(DtTime time,
    DtTimeStampType type = DtTimeStampRelative);

// Return the time of validity of the message (in seconds past an hour)
// as set by setTimeStamp.
virtual DtTime timeStamp() const;

// Return the type of the time stamp.
virtual DtTimeStampType timeStampType() const;

// Since timestamp only represents number of seconds past the hour,
// guessTimeValid chooses the hour that would make the timestamp
// closest to reference time, and returns a time that is timestamp
// seconds past that hour.
virtual DtTime guessTimeValid(DtTime referenceTime) const;

public:

    // Set/Get flag that tells us whether we're using the RPR FOM
    // convention that the UserDefinedTag in updates and interactions
    // represents a DIS-style timestamp.
    static void setTagIsAsciiTime(DtBoolean val);
    static DtBoolean tagIsAsciiTime();

    // Register/Deregister callback function cb with conn, on
    // all RTI::InteractionClassHandles contained in classes.
    static void addCallback(DtExerciseConn* conn,
        DtList* classes, DtReceiveInteractionCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtList* classes, DtReceiveInteractionCb cb, void* usr);

    // Returns maximum supported number of interaction classes.
    static int maxInteractionClasses();

protected:

    // Returns the FOM interaction class name to use.
```

```

// Default implementation is to ask the fomMapper.
virtual char* interactionClassToUse(
    DtExerciseConn* exConn) const;

// Returns a pointer to a private pvList that can be used as
// a return value by phvps. It's guaranteed to be able to fit
// at least numParameters() parameters.
virtual RTI::ParameterHandleValuePairSet*
    DtInteraction::privPvList() const;

// Get rid of constness on the "this" pointer in order to access
// non-const functions from const member functions (such as simple
// accessors). This should be used sparingly since it bypasses
// the const protection.
// Note: This is essentially like an operator and is
//       non-virtual since can't override based on return type.
//       Allows derived classes to also define this member.
DtInteraction* self() const;

protected:

    DtString myTag;
    DtInterClassDesc* myClassDesc;
    DtExerciseConn* myExConn;
    DtTime myTime;
    DtTimeStampType myTimeType;

protected:

    static DtBoolean theTagIsAsciiTime;
    static RTI::ParameterHandleValuePairSet* thePvList;
    static int thePvListMaxSize;
    static const int theMaxInteractionClasses;
};
#include "intClassDesc.h"

#endif

```

hlInterFactory.h

```
#if DtHLA

#include "vllist.h"
#include "RTI.hh"
#include "MachineTypes.h"

#include "chooseDefines.h"

// class DtInteractionFactory:
//
// Instances of DtInteractionFactory are used to create derived
// DtInteractions based on Interaction Class.

class DtInteractionFactory
{
public:
    // Interaction creation function signature.
    typedef DtInteraction* (*DtInteractionCreator)();

public:
    // constructor
    DtInteractionFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtInteractionFactory();

    // copy constructor
    DtInteractionFactory(const DtInteractionFactory& orig);

    // assignment operator
    DtInteractionFactory& operator=(const DtInteractionFactory& orig);

    // Create the specified interaction.
    // Note: Caller is responsible for deleting.
    // Returns NULL if interaction creation function is not in the factory.
    virtual DtInteraction* createInteraction(
        RTI::InteractionClassHandle handle,
        const RTI::ParameterHandleValuePairSet& pairSet);

    // Add an interaction creation function to the factory.
    // The function must return a new'ed interaction object which will
    // be deleted by the user.
    // Returns any previous interaction creation function which was
    // replaced by the add.
    virtual DtInteractionCreator addCreator(
        RTI::InteractionClassHandle handle,
        DtInteractionCreator creator);

protected:
    // myCreators is an array of DtInteraction::theMaxInteractionClasses
    // items.
    DtInteractionCreator* myCreators;
    DtExerciseConn* myExConn;
};
#endif
```


hInterWData.h

```

#if DtHLA

#include "hIntWEncDec.h"
#include "hostStructs.h"
#include "mathTypes.h"
#include "datumRecSet.h"

#include "chooseDefines.h"

// class DtInteractionWithData:
//
// Instances of DtInteractionWithData are used to represent
// RPR FOM Interactions that use fixed and var datum records.

class DtInteractionWithData : public DtInteractionWithEncDec
{
public:
    // default constructor
    DtInteractionWithData();

    // destructor
    virtual ~DtInteractionWithData();

    // copy constructor
    DtInteractionWithData(const DtInteractionWithData& orig);

    // assignment operator
    DtInteractionWithData& operator=(
        const DtInteractionWithData& orig);

    // Print the interaction's data.
    virtual void printData() const;

    virtual int numFixedFields() const;

    // returns number of variable length fields
    virtual int numVarFields() const;

    // Returns the datum param found in the index'th variable
    // datum record. Index is between 1 and numRecords. Returns
    // DtDatumParam(0) if index is out of bounds.
    virtual DtDatumParam varDatumId(int index) const;

    // Returns the datum param found in the index'th fixed
    // datum record. Index is between 1 and numRecords. Returns
    // DtDatumParam(0) if index is out of bounds.
    virtual DtDatumParam fixedDatumId(int index) const;

    // Returns the datum param found in the index'th fixed or variable
    // datum record, depending on whether fixedOrVar is DtFixed or DtVar.
    // Index is between 1 and numRecords. Returns
    // DtDatumParam(0) if index is out of bounds.
    virtual DtDatumParam datumId(DtDatumRecordType fixedOrVar, int index)
const;

    // Returns the number of bits of data contained in the value
    // fields of the index'th fixed or variable datum record,
    // depending on whether fixedOrVar is DtFixed or DtVar. If DtFixed,
    // the answer is always 32.
    virtual int datumSizeBits(DtDatumRecordType fixedOrVar, int index) const;

    // Returns the number of bytes of data contained in the value

```

```
// fields of the index'th fixed or variable datum record,
// depending on whether fixedOrVar is DtFixed or DtVar. If DtFixed,
// the answer is always 4.
virtual int datumSizeBytes(DtDatumRecordType fixedOrVar, int index)
const;

// Looks through all fixed and variable records for one containing
// param as the parameter ID. If it finds one, fills supplied fixedOrVar
// pointer with DtFixed or DtVar, and index with the index to indicate
// which record was found, and fills retcode with DtDatumValid. If param
// was not found, retcode is filled with DtDatumBadId, and index and
// fixedOrVar are invalid.
virtual void getIndex(DtDatumParam param, DtDatumRecordType* fixedOrVar,
    int* index, DtDatumRetCode* retcode = NULL) const;

// ***** INSPECTING DATUM VALUES BY INDEX *****
// If you know the data type of the value contained in a particular
// record, use one of the following to return it. fixedOrVar (either
DtFixed
// or DtVar) and index (between 1 and numRecords) indicate which record
// you are inspecting. datumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It returns a pointer to a character string containing the
// exact bytes in the network representation that represents the value
// being inspected. The supplied retcode pointer will be filled with
// DtDatumValid if the value was successfully inspected, and DtDatumBa-
dLength
// if either the index was invalid, or the size of the data indicated that
// it could not be interpreted as the chosen data type.

    virtual DtFloat32 datumValFloat32(DtDatumRecordType fixedOrVar, int
index,
        DtDatumRetCode* retcode = NULL) const;
    virtual DtFloat64 datumValFloat64(DtDatumRecordType fixedOrVar, int
index,
        DtDatumRetCode* retcode = NULL) const;
    virtual unsigned long datumValUnsigned32(DtDatumRecordType fixedOrVar,
int index, DtDatumRetCode* retcode = NULL) const;
    virtual long datumValInt32(DtDatumRecordType fixedOrVar, int index,
DtDatumRetCode* retcode = NULL) const;
    virtual const char* datumValByteArray(DtDatumRecordType fixedOrVar,
int index) const;

// ***** INSPECTING DATUM VALUES BY PARAM ID *****
// If you know the data type of the value contained in a particular
// record, use one of the following to return it. The particular record
// inspected will be the first one containing param as the datum ID.
// datumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It returns a pointer to a character string containing the
// exact bytes in the network representation that represents the value
// being inspected. The supplied retcode pointer will be filled with
// DtDatumValid if the value was successfully inspected, and DtDatumBa-
dLength
// if either param was not found, or the size of the data indicated that
// it could not be interpreted as the chosen data type.

    virtual DtFloat32 datumValFloat32(DtDatumParam param,
        DtDatumRetCode* retcode = NULL) const;
    virtual DtFloat64 datumValFloat64(DtDatumParam param,
        DtDatumRetCode* retcode = NULL) const;
    virtual unsigned long datumValUnsigned32(DtDatumParam param,
        DtDatumRetCode* retcode = NULL) const;
    virtual long datumValInt32(DtDatumParam param,
        DtDatumRetCode* retcode = NULL) const;
    virtual const char* datumValByteArray(DtDatumParam param,
        DtDatumRetCode* retcode = NULL) const;
```

```
// ***** MUTATOR FUNCTIONS *****

// Initialize to reflect the fixed parameters
// and variable parameters and sizes indicated. fixedParams is a NULL-
// terminated array of DtDatumParams. specList is an array of DtVarDa-
// tumSpecs
// terminated with the DtNullVarDatumSpec constant. This call can be
made
// instead of calls to setNumVarDatumRecords, setNumFixedDatumRecord,
// and repeated calls to setDatumParam and setVarDataBytes.
virtual void initDataSection(const DtDatumParam* fixedParams,
    const DtVarDatumSpec* specList);

// Sets the number of fixed or variable fields to newNum. Previously
// existing fixed fields with indices greater than newNum are deleted.
// If newNum exceeds the number of existing fields, trailing new fields
// are created with zeros as datum params, sizes (for var only) and
values.
virtual void setNumFixedFields(int newNum);
virtual void setNumVarFields(int newNum);

// Set the datum ID in the index'th fixed or variable datum record to
// param, depending on whether fixedOrVar is DtFixed or DtVar. Supplied
// retcode pointer is filled with DtDatumValid if successful, or DtDatum-
// BadIndex
// if the index is not valid.
virtual void setDatumParam(DtDatumRecordType fixedOrVar, int index, DtDa-
// tumParam param,
    DtDatumRetCode* retcode = NULL);

// Allocates length bytes for the index'th variable datum record.
// Should be called before using one of the setDatumVal functions
// to actually set the data. Supplied retcode pointer is set to
// DtDatumValid if successful, or DtDatumBadIndex if the index
// is not valid.
virtual void setVarDataBytes(int index, int length, DtDatumRetCode*
retcode = NULL);

// ***** SETTING DATA VALUES BY INDEX *****
// Use one of the following to set the data value in a fixed or variable
datum
// record, depending on the data type of your data. fixedOrVar (either
DtFixed
// or DtVar) and index (between 1 and numRecords) indicate which record
// you are modifying. setDatumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It takes a pointer to a character string containing the
// exact bytes that should be used in the network representation.
// The supplied retcode pointer will be filled with
// DtDatumValid if the value was successfully set, DtDatumBadIndex if the
// index is invalid, and DtDatumBadLength if the size of the data indicated
that
// it could not be set from as the chosen data type.

virtual void setDatumValFloat64(DtDatumRecordType fixedOrVar, int index,
DtFloat64 val,
    DtDatumRetCode* retcode = NULL);
virtual void setDatumValFloat32(DtDatumRecordType fixedOrVar, int index,
DtFloat32 val,
    DtDatumRetCode* retcode = NULL);
virtual void setDatumValInt32(DtDatumRecordType fixedOrVar, int index,
long val,
    DtDatumRetCode* retcode = NULL);
virtual void setDatumValUnsigned32(DtDatumRecordType fixedOrVar, int
index,
    unsigned long val, DtDatumRetCode* retcode = NULL);
```

```
    virtual void setDatumValByteArray(DtDatumRecordType fixedOrVar, int
index,
    const char* val, DtDatumRetCode* retcode = NULL);

// ***** SETTING DATA VALUES BY PARAM ID *****
// Use one of the following to set the data value in a fixed or variable
datum
// record, depending on the data type of your data. The particular record
modified
// will be the first one containing param as its parameter ID.
// setDatumValBytesArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It takes a pointer to a character string containing the
// exact bytes that should be used in the network representation.
// The supplied retcode pointer will be filled with
// DtDatumValid if the value was successfully set, DtDatumBadId if param
// is invalid, and DtDatumBadLength if the size of the data indicated that
// it could not be set from the chosen data type.

    virtual void setDatumValFloat64(DtDatumParam param, DtFloat64 val,
DtDatumRetCode* retcode = NULL);
    virtual void setDatumValFloat32(DtDatumParam param, DtFloat32 val,
DtDatumRetCode* retcode = NULL);
    virtual void setDatumValInt32(DtDatumParam param, long val,
DtDatumRetCode* retcode = NULL);
    virtual void setDatumValUnsigned32(DtDatumParam param, unsigned long val,
DtDatumRetCode* retcode = NULL);
    virtual void setDatumValByteArray(DtDatumParam param, const char* val,
DtDatumRetCode* retcode = NULL);

// Return the DtDatumRecordSet which an InteractionWithData
// uses to store its fixed and var records. In general,
// the wrapper functions above should be used rather than going directly
// through the DtDatumRecordSet, which is necessary only when doing
// encoding and decoding of network representations.
    virtual DtDatumRecordSet* datumRecSet();
    virtual const DtDatumRecordSet* datumRecSet() const;

protected:

    DtDatumRecordSet myRecSet;
};
#endif
```

hIntWEncDec.h

```

#if DtHLA

#include "RTI.hh"
#include "hInteraction.h"

#include "chooseDefines.h"

// class DtInteractionWithEncDec:
//
// DtInteractionWithEncDec is a DtInteraction that uses an encoder and
// decoder to implement the virtual functions phvps and setFromPhvps.

class DtInteractionWithEncDec : public DtInteraction
{
public:
    // default constructor
    DtInteractionWithEncDec();

    // destructor
    virtual ~DtInteractionWithEncDec();

    // copy constructor
    DtInteractionWithEncDec(const DtInteractionWithEncDec& orig);

    // assignment operator
    DtInteractionWithEncDec& operator=(const DtInteractionWithEncDec& orig);

    // Sets the interaction's ExerciseConn and class descriptor. If
    // classDesc is NULL, a default is chosen by calling
    // interactionClassToUse(). Without being told what ExConn or class
    // descriptor to use, many of the HLA-specific functions won't
    // work properly. However, it is usually not necessary for application
    // code to call this function, since it is called by
    // DtExerciseConn::send on the outgoing side, and by the
    // DtInteractionFactory on the incoming side.
    virtual void setExConn(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc = NULL);

    // Get the parameter handle value pair set for the interaction.
    virtual const RTI::ParameterHandleValuePairSet& phvps() const;

    // Set contents from a parameter handle value pair set
    virtual void setFromPhvps(
        const RTI::ParameterHandleValuePairSet& pvList);

    // Specify an encoder, decoder, and interaction data repository
    // to use in encoding and decoding. Called by setExConn (using
    // encoder and decoder obtained through encoderToUse() and
    // decoderToUse()) so call AFTER setExConn if you want to
    // override defaults.
    virtual void useThese(DtInteractionEncoder* enc,
        DtInteractionDecoder* dec,
        DtInteractionRep* data);

    // Return the encoder we are using
    virtual DtInteractionEncoder* encoder() const;

    // Return the decoder we are using
    virtual DtInteractionDecoder* decoder() const;

protected:

```

```
// Returns the encoder to use for decoding the interaction.
// Default implementation is to ask the fomMapper.
virtual DtInteractionDecoder* decoderToUse(
    DtExerciseConn* exConn) const;

// Returns the decoder to use for decoding the interaction.
// Default implementation is to ask the fomMapper.
virtual DtInteractionEncoder* encoderToUse(
    DtExerciseConn* exConn) const;

protected:
    DtInteractionDecoder* myDecoder;
    DtInteractionEncoder* myEncoder;
    DtInteractionRep* myData;
};
#endif
```

hlaObj.h

```

#if DtHLA

#include "RTI.hh"
#include "MachineTypes.h"
#include "remObjCbInfo.h"
#include "vlTime.h"
#include "objRefCbInfo.h"
#include "vlprint.h"
#include "objClassDesc.h"

#include "chooseDefines.h"

// class DtHlaObject:
//
// Instances of DtHlaObject are used to represent HLA objects (as
// opposed to interactions) which are communicated across the network.

class DtHlaObject
{
    // DtReflectedHlaObjectManager is a friend
    // so that it can manage these objects.
    friend class DtReflectedHlaObjectManager;

public:

#if DtRTI103
    // constructor
    DtHlaObject(DtExerciseConn* conn, RTI::ObjectHandle id,
                DtObjClassDesc* classDesc);
#else // constructor
    DtHlaObject(DtExerciseConn* conn, RTI::ObjectHandle id,
                const char* name,
                DtObjClassDesc* classDesc);

    // constructor
    DtHlaObject(DtExerciseConn* conn, const char* name,
                DtObjClassDesc* classDesc);
#endif

    // destructor
    virtual ~DtHlaObject();

    // Get the object's identifier.
    virtual const RTI::ObjectHandle& objectId() const;

#if !DtRTI103
    // Get the object's identifier.
    virtual const char* name() const;
#endif

    // Get the object's class handle.
    virtual RTI::ObjectClassHandle objectClassHandle() const;

    // Get the number of attributes associated with this object.
    virtual int numAttributes() const;

    // Get the attribute handle set.
    virtual const RTI::AttributeHandleSet& attrHandleSet();

    // Get the exercise connection for this object.
    virtual DtExerciseConn* exerciseConn() const;

    //

```

```
// Local Object specific functions.
//

// Do the initialization required to begin publishing this object.
virtual void setupUpdating();

// Update (publish) this object on the network.
virtual void update() = 0;

// Do the deinitialization required to stop publishing this object.
virtual void cleanupFromUpdating();

// Set yourself up so that on the next tick, you send an update
// for the appropriate attributes. (Can't send them within this
// function since this is called from within an RTI service).
virtual void processProvideUpdate(
    const RTI::AttributeHandleSet& theAttributes);

// returns the union of all sets of attributes requested since the
// last call to clearRequestedAttributes.
virtual const RTI::AttributeHandleSet& requestedAttributes();

// clears the set of requested attributes. Should be called after
// request has been serviced.
virtual void clearRequestedAttributes();

// returns the set of attributes that are currently needed by the
// federation, i.e. those that have been specified in startUpdates
// calls, but not those that have been specified in stopUpdates.
virtual const RTI::AttributeHandleSet& attributesNeededByFederation();

// returns DtTrue if the class is needed by the federation,
// i.e. we've received a startRegistration or startUpdates
// service call from the RTI for the object's class.
DtBoolean classNeededByFederation();

// ensures that the class published, and that start and stopUpdates
// callbacks are registered.
virtual void setupClassForUpdating();

//
// Remote Object specific functions.
//

// Do the initialization required to handle reflection of object.
virtual void setupReflecting();

// Calls reflect, and any postReflect callbacks
// Override reflect rather than this function
// to change the way an HLA object reflects a stateMsg.
virtual void callReflect(const DtStateMsg& stateMsg);

// Do the deinitialization required to stop handling reflection of
// object.
virtual void cleanupFromReflecting();

// Request an update of the object's attributes
virtual void requestUpdate(RTI::AttributeHandleSet* attrs = NULL);

// Add (register) function to be called when object is removed.
virtual void addRemoveObjectCb(DtRemoveObjectCb cb,
    void* usr);

// Remove (deregister) function to be called when object is removed.
virtual void removeRemoveObjectCb(DtRemoveObjectCb cb,
    void* usr);
```

```

// Add (register) function to be called after each reflectAttrValues
virtual void addPostReflectCb(DtObjectReflectAttributeValuesCb cb,
    void* usr);

// Remove (deregister) function to be called after each reflectAttrValues
virtual void removePostReflectCb(DtObjectReflectAttributeValuesCb cb,
    void* usr);

virtual DtTime lastSimTimeUpdateReceived() const;

// Prints contents of a state msg. Base class prints only header info.
virtual void printMsg(const DtStateMsg& msg) const;

public:

    // Get maximum number of object class handles supported
    static int maxObjectClasses();

protected:

    // copy constructor
    DtHlaObject(const DtHlaObject& orig);

    // assignment operator
    DtHlaObject& operator=(const DtHlaObject& orig);

    // Process notification of removal of the object from the network.
    // Call back all the remove object callbacks.
    virtual void processRemoveObject();

    // Handle reflection of object (receipt of state message) from the
network.
    // called by callReflect. Override this function rather than callReflect
    // to change the way an HLA object reflects a stateMsg.
    virtual void reflect(const DtStateMsg& stateMsg) = 0;

    // Remove all callbacks from the remove callback list.
    // Non-virtual since meant to be called from dtor.
    void wipeRemoveList();

    // Virtual version of above
    virtual void clearRemoveList();

    // Remove all callbacks from the reflect callback list.
    // Non-virtual since meant to be called from dtor.
    void wipeReflectList();

    // Copy another HLA object's remove callback list into this
    // object's remove callback list.
    // Non-virtual since meant to be called from ctor.
    void copyRemoveList(const DtHlaObject& orig);

    // Set the time an update was last received or sent.
    virtual void setLastSimTimeUpdateReceived(DtTime time);

    // apply callbacks after a state msg is reflected
    virtual void applyPostReflectCallbacks(const DtStateMsg& stateMsg);

#if !DtRTI103
    // Callback for handling turnUpdatesOnForObjectInstance RTI call.
    virtual void processUpdatesOn(
        RTI::ObjectHandle hand,
        const RTI::AttributeHandleSet& set);

    // Callback for handling turnUpdatesOffForObjectInstance RTI call.
    virtual void processUpdatesOff(
        RTI::ObjectHandle hand,

```

```
        const RTI::AttributeHandleSet& set);
#endif

protected:

    // Callback for handling state messages for the HLA object.
    // Casts hlaObj to DtHlaObject* and calls reflect().
    static void reflectAttributeValuesCb(const DtStateMsg& stateMsg,
        void* hlaObj);

#if !DtRTI103
    // Callback for handling turnUpdatesOnForObjectInstance RTI call.
    static void DtHlaObject::updatesOnCb(
        RTI::ObjectHandle hand,
        const RTI::AttributeHandleSet& attributeSet,
        void* usr);

    // Callback for handling turnUpdatesOffForObjectInstance RTI call.
    static void DtHlaObject::updatesOffCb(
        RTI::ObjectHandle hand,
        const RTI::AttributeHandleSet& attributeSet,
        void* usr);
#endif

protected:

#if !DtRTI103
    char* myName;
    // Needed attributes is a per-object concept in RTI1.3
    RTI::AttributeHandleSet* myNeededAttrs;
#endif

    DtExerciseConn* myExConn;
    RTI::ObjectHandle myObjectId;
    DtBoolean myReflecting;
    DtBoolean myUpdating;
    DtList myRemoveCbList;
    DtTime myLastSimTimeUpdateReceived;
    DtList myReflectCbList;
    DtObjClassDesc* myClassDesc;
    RTI::AttributeHandleSet* myRequestedAttributes;

protected:

    static const int theMaxObjectClasses;
};
#if !DtRTI103
#endif
#endif
```

hlaObjFact.h

```

#if DtHLA

#include "hlaObj.h"

#include "chooseDefines.h"

// class DtHlaObjectFactory:
//
// Instances of DtHlaObjectFactory are used to create HLA objects.

class DtHlaObjectFactory
{
public:

#if DtRTI103
    // HLA object creation function signature.
    typedef DtHlaObject* (*DtHlaObjectCreator)
        (DtExerciseConn* conn, RTI::ObjectHandle id, DtObjClassDesc* cDesc);
#else
    // HLA object creation function signature.
    typedef DtHlaObject* (*DtHlaObjectCreator)
        (DtExerciseConn* conn, RTI::ObjectHandle id,
         const char* name, DtObjClassDesc* cDesc);
#endif

public:

    // constructor
    DtHlaObjectFactory(DtExerciseConn* conn);

    // destructor
    virtual ~DtHlaObjectFactory();

    // copy constructor
    DtHlaObjectFactory(const DtHlaObjectFactory& orig);

    // assignment operator
    DtHlaObjectFactory& operator=(const DtHlaObjectFactory& orig);

#if DtRTI103
    // Create the specified object.
    // Note: Caller is responsible for deleting.
    // Returns NULL if object creation function is not in the factory.
    virtual DtHlaObject* createHlaObject(
        RTI::ObjectClassHandle handle,
        RTI::ObjectHandle id);
#else
    // Create the specified object.
    // Note: Caller is responsible for deleting.
    // Returns NULL if object creation function is not in the factory.
    virtual DtHlaObject* createHlaObject(
        RTI::ObjectClassHandle handle,
        RTI::ObjectHandle id,
        const char* name);
#endif

    // Add an object creation function to the factory.
    // The function must return a new'ed object which will
    // be deleted by the user.
    // Returns any previous object creation function which was
    // replaced by the add.
    virtual DtHlaObjectCreator addCreator(RTI::ObjectClassHandle handle,
        DtHlaObjectCreator creator);

```

```
protected:
    // myCreators is an array of DtHlaObject::theMaxObjectClasses items.
    DtHlaObjectCreator* myCreators;
    DtExerciseConn* myConn;
};
#endif
```

hlaObjWithSR.h

```

#if DtHLA

#include "hlaObj.h"

#include "chooseDefines.h"

// class DtHlaObjectWithStateRep:
//
// Instances of DtHlaObjectWithStateRep are used to represent an
// HLA object via a state representation.

class DtHlaObjectWithStateRep : public DtHlaObject
{
public:

#if DtRTI103
    // constructor
    DtHlaObjectWithStateRep(DtExerciseConn* conn, RTI::ObjectHandle id,
        DtObjClassDesc* classDesc);
#else
    // constructor
    DtHlaObjectWithStateRep(DtExerciseConn* conn, RTI::ObjectHandle id,
        const char* name,
        DtObjClassDesc* classDesc);
    // constructor
    DtHlaObjectWithStateRep(DtExerciseConn* conn, const char* name,
        DtObjClassDesc* classDesc);
#endif

    // destructor
    virtual ~DtHlaObjectWithStateRep();

    // copy constructor
    DtHlaObjectWithStateRep(const DtHlaObjectWithStateRep& orig);

    // assignment operator
    DtHlaObjectWithStateRep& operator=(const DtHlaObjectWithStateRep& orig);
    //
    // Local Entity specific functions.
    //

    // Set the state repositories (local representation and remote
    // reflection), and the encoder and decoder which are used to
    // send and receive HLA state messages from/into the repository.
    virtual void useTheseForUpdating(DtStateRepository* stateRep,
        DtStateRepository* asSeenByRemote,
        DtHlaStateEncoder* encoder,
        DtHlaStateDecoder* decoder);

    // Send representation (remote reflection) onto the network.
    virtual void update();

    //
    // Remote Entity specific functions.
    //

    // Set the state repository for holding a remote reflection
    // and the decoder which is used to receive HLA state messages
    // into the repository.
    virtual void useTheseForReflecting(DtStateRepository* stateRep,
        DtHlaStateDecoder* decoder);

    // Receive representation (remote reflection) from the network.

```

```
    virtual void reflect(const DtStateMsg& stateMsg);

    // Prints contents of a state msg. Base class prints only header info.
    virtual void printMsg(const DtStateMsg& msg) const;

public:
    #if DtRTI103
        // Creator function for this class
        static DtHlaObject* create(DtExerciseConn* conn,
            RTI::ObjectHandle id,
            DtObjClassDesc* classDesc);
    #else
        // Creator function for this class
        static DtHlaObject* create(DtExerciseConn* conn,
            RTI::ObjectHandle id,
            const char* name,
            DtObjClassDesc* classDesc);
    #endif

protected:
    DtHlaStateDecoder* myDecoder;
    DtHlaStateEncoder* myEncoder;
    DtStateRepository* myStateRep;
    DtStateRepository* myAsSeenByRemote;
};
    #if DtRTI103
    #else
    #endif
#endif
```

hLcDec.h

```

#if DtHLA

#include "interDecoder.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_LGR_CONTROL_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtLgrControlInteraction, paramName)

// class DtLgrControlDecoder:
//
// Instances of DtLgrControlDecoder are used to decode LgrControl Interac-
// tions
// from the network.

class DtLgrControlDecoder : public DtInteractionDecoder
{
public:
    // constructor
    DtLgrControlDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtLgrControlDecoder();

    // copy constructor
    DtLgrControlDecoder(const DtLgrControlDecoder& orig);

    // assignment operator
    DtLgrControlDecoder& operator=(const DtLgrControlDecoder& orig);

protected:
    // Individual parameter decoders

    DtDECLARE_LGR_CONTROL_PARAM_DECODER(LgrControlPdu);
};
#endif

```

hLcEnc.h

```
#if DtHLA

#include "interEncoder.h"

#define DtDECLARE_LGR_CONTROL_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtLgrControlInteraction, paramName)

#define DtDEFINE_SIMPLE_LGR_CONTROL_PARAM_ENCODER(paramName, netType, \
    inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtLgrControlEncoder, DtLgrControlInterac- \
    tion, \
    paramName, netType, inspector)

// class DtLgrControlEncoder:
//
// DtLgrControlEncoder is a DtInteractionEncoder that registers
// parameter encoders for all of the LGR_CONTROL interaction parameters.

class DtLgrControlEncoder : public DtInteractionEncoder
{
public:

    // add parameter encoder "encodeLgrControlPdu"
    DtLgrControlEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtLgrControlEncoder();

    // copy constructor
    DtLgrControlEncoder(const DtLgrControlEncoder& orig);

    // assignment operator
    DtLgrControlEncoder& operator=(const DtLgrControlEncoder& orig);

protected:

    // Individual parameter encoders

    DtDECLARE_LGR_CONTROL_PARAM_ENCODER(LgrControlPdu);
};
#endif
```


hLclnter.h

```

#ifdef DtHLA

#include "chooseDefines.h"
#include "hIntWEncDec.h"
#include "lcPdu.h"

// class DtLgrControlInteraction:
//
// Instances of DtLgrControlInteraction are used to represent
// LgrControl Interactions in HLA.

// LgrControl interaction callback function signature.
typedef void (*DtLgrControlInteractionCb)(DtLgrControlInteraction* inter,
    void* usr);

class DtLgrControlInteraction : public DtInteractionWithEncDec
{
public:
    // default constructor
    DtLgrControlInteraction();

    // destructor
    virtual ~DtLgrControlInteraction();

    // copy constructor
    DtLgrControlInteraction(const DtLgrControlInteraction& orig);

    // assignment operator
    DtLgrControlInteraction& operator=(const DtLgrControlInteraction& orig);

    // Print the interaction's data.
    virtual void printData() const;

    // Get/Set a logger control PDU
    virtual const DtLgrControlPdu* lcPdu() const;
    virtual void setLcPdu(const DtLgrControlPdu& pdu);
    virtual void setFromNetPacket(const DtNetPacket* pack);

    // Name of the interaction
    virtual const char* name() const;

public:
    // Create a DtLgrControlInteraction.
    // Caller is responsible for deletion.
    static DtInteraction* create();

    // Add (register)/Remove (deregister) function to be called when
    // interaction occurs.
    static void addCallback(DtExerciseConn* conn,
        DtLgrControlInteractionCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtLgrControlInteractionCb cb, void* usr);

protected:
    DtLgrControlPdu* myLcPdu;
};
#endif

```

hObjPub.h

```
#if DtHLA

#ifndef DtObjectPublisher_H_
#define DtObjectPublisher_H_

#include "chooseDefines.h"
#include "RTI.hh"
#include "hostStructs.h"
#include "hlaObjWithSR.h"

// class DtObjectPublisher:
//
// Instances of DtObjectPublisher are used for local or source simulations
// of objects, (i.e. these objects are not dependent on data being
// produced elsewhere on the network for their existence). These
// object send data to the network.

class DtObjectPublisher
{
public:

    // Constructor. Object is not fully constructed until init is called.
    DtObjectPublisher(DtExerciseConn* conn, const char* name = NULL);

    // Destructor
    virtual ~DtObjectPublisher();

    // Returns a pointer to the DtStateRepository - through
    // which you can set or inspect current state information.
    virtual DtStateRepository* stateRep();

    // Synonym for stateRep()
    virtual DtStateRepository* sr();

    // Returns a pointer to the Decoder being used to decode
    // state messages that we send back into the asSeenByRemote
    // SR.
    virtual DtHlaStateDecoder* decoder();

    // Returns a pointer to the Encoder being used to encode
    // current state into an outgoing message.
    virtual DtHlaStateEncoder* encoder();

    // Get the object's hlaObject
    virtual DtHlaObject* hlaObject();

    // Get the object's local identifier.
    virtual const DtObjectId& id() const;

    // Get the object's local identifier.
    virtual const DtObjectId& objectId() const;

    // Get the object's name.
    virtual const DtString& name() const;

    // Get the HLA object's global object designator -
    // the object name in RTI1.3.3, but the objectId in RTI1.0.3.
    virtual const DtGlobalObjectDesignator& globalId() const;

    // Tick (update) this object.
    virtual void tick();

protected:
```

```
// copy constructor
DtObjectPublisher(const DtObjectPublisher& orig);

// assignment operator
DtObjectPublisher& operator= (const DtObjectPublisher& orig);

// Perform constructor common initialization.
// Non-virtual since meant to be called from constructors of derived
// classes. If encoder or decoder are NULL, the FOM Mapper is consulted.
// This class assumes ownership of the objects passed to init. Do
// not delete them.
void init(
    RTI::ObjectClassHandle hand,
    DtStateRepository* stateRep,
    DtStateRepository* asSeenByRemote,
    DtHlaStateEncoder* encoder = NULL,
    DtHlaStateDecoder* decoder = NULL);

protected:

    DtExerciseConn* myExConn;
    DtHlaObjectWithStateRep* myHlaObj;
    DtStateRepository* myStateRep;
    DtStateRepository* myAsSeenByRemote;
    DtHlaStateEncoder* myEncoder;
    DtHlaStateDecoder* myDecoder;
    DtObjectId myObjectId;
    DtString myName;
};
#endif //DtObjectPublisher_H_
```

hostStructs.h

```
#include "chooseDefines.h"
#include "MachineTypes.h"
#include "NetStructs.h"
#include "disEnums.h"
#include <string.h>
#include <stdio.h>
#include "globalObjDes.h"

#if DtHLA
#include "RTI.hh"
#endif

#if DtHLA && DtRTI103
#define ObjectHandle ObjectID
#endif

typedef DtU16 DtExerciseId;

enum
{
    DtkObjectIsUnknown = -1,
    DtkObjectIsDis      = 0,
    DtkObjectIsHla      = 1
};

/*for backwards compatibility */
typedef struct DtSimulationAddress
{
    DtU16 DtSite;
    DtU16 DtHost;
} DtSimulationAddress;

#include "eventId.h"

#if DtHLA

class DtFederateId
{
public:
    DtFederateId();
    DtFederateId(const char* str);
    DtFederateId(int site, int app);
    DtFederateId(const DtNetFederateId& netId);

    const DtFederateId& operator=(const DtFederateId &netId);
    const DtNetFederateId& operator=(const DtNetFederateId &netId);

    int operator==(const DtFederateId &otherGuy) const;
    int operator!=(const DtFederateId &otherGuy) const;

    void init(int site, int app);

    operator DtNetFederateId () const;

    int siteId() const;
    void setSiteId(int siteId);

    int applicationId() const;
    void setApplicationId(int id);

    char *string() const;

    static const DtFederateId& nullId();
};
```

```

protected:
    int mySiteId;
    int myAppId;
};

#endif // DtHLA

#if !i386
const DtU16 DtDeferredEntityId = ((1<<16)-1);
#else
const DtU16 DtDeferredEntityId = 0xff; /* CJW: THIS IS A BUG */
#endif

class DtEntityIdentifier
{
public:
    DtEntityIdentifier(const char* str);
    DtEntityIdentifier(int site, int host, int entityNum);
    DtEntityIdentifier(DtNetEntityIdentifier netId);
    DtEntityIdentifier();

    void init(int site, int host, int entityNum);

    void setFromNet(DtNetEntityIdentifier netId)
    {
        init(netId.DtSimulator.DtSite, netId.DtSimulator.DtHost,
            netId.DtEntity);
    }

    operator DtNetEntityIdentifier () const
    {
        DtNetEntityIdentifier id;

        id.DtSimulator.DtSite = site();
        id.DtSimulator.DtHost = host();
        id.DtEntity = entityNum();
        return(id);
    }

    int site() const {return (int) DtSimulator.DtSite;}
    int host() const {return (int) DtSimulator.DtHost;}
    int entityNum() const {return (int) DtEntity;}

    void setSite(int site) {DtSimulator.DtSite = site;}
    void setHost(int host) {DtSimulator.DtHost = host;}
    void setEntityNum(int entityNum) {DtEntity = entityNum;}

    int sameHost(const DtEntityIdentifier &otherGuy) const;
    int sameEntity(const DtEntityIdentifier &otherGuy) const;
    int matchPattern(const DtEntityIdentifier &pat) const;

    int operator==(const DtEntityIdentifier &otherGuy) const{return sameEntity(otherGuy);}
    int operator!=(const DtEntityIdentifier &otherGuy) const{if(sameEntity(otherGuy)){return 0;}else{return 1;}}

    char *string() const;

    static const DtEntityIdentifier& nullId();

public:
    DtSimulationAddress DtSimulator;
    DtU16 DtEntity;
};

```

```
#if !DtHLA
typedef DtEntityIdentifier DtObjectId;
#else
class DtObjectId
{
public:

    DtObjectId();
    DtObjectId(RTI::ObjectHandle id);
    DtObjectId(const char* str);
    DtObjectId& operator=(const DtObjectId& orig);
    DtObjectId(const DtObjectId& orig);

#if DT_LITTLE_ENDIAN
    // On big endian machines, this is the same
    // as DtObjectId(RTI::ObjectHandle id); defined above.
    DtObjectId(DtNetObjectId netId);
#endif // DT_LITTLE_ENDIAN
    operator RTI::ObjectHandle() const;

    char *string() const;
    static const DtObjectId& nullId();

protected:

    RTI::ObjectHandle    myId;
};
#endif

#define DtAny -1

class DtEntityType
{
public:
    DtEntityType(const char* p);
    DtEntityType(int kind, int domain, int country, int category,
        int subCategory, int specific, int extra);
    DtEntityType();
    DtEntityType(DtNetEntityType netType);
    void setFromNet(DtNetEntityType netType)
    {
        init(netType.DtEntityKind, netType.DtDomain, netType.DtCountry,
            netType.DtCategory, netType.DtSubCategory, netType.DtSpecific,
            netType.DtExtra);
    }

    operator DtNetEntityType() const;

    void init(int kind, int domain, int country, int category,
        int subCategory, int specific, int extra);
    char *string() const;

    static const DtEntityType& nullType();

public:
    int DtEntityKind; /* this conflicts with enum of same name, so must
        be before funcs that use it. Leave for awhile
        back-compat for awhile. */
    int DtDomain;
    int DtCountry;
    int DtCategory;
    int DtSubCategory;
    int DtSpecific;
    int DtExtra;

public:
```

```

/* int entityKind() const {return(DtEntityKind);} */
int kind() const {return(DtEntityKind);}
int domain() const {return(DtDomain);}
int country() const {return(DtCountry);}
int category() const {return(DtCategory);}
int subCategory() const {return(DtSubCategory);}
int specific() const {return(DtSpecific);}
int extra() const {return(DtExtra);}

/* void setEntityKind(int kind){DtEntityKind = kind;} */
void setKind(int kind){DtEntityKind = kind;}
void setDomain(int domain){DtDomain = domain;}
void setCountry(int country){DtCountry = country;}
void setCategory(int category){DtCategory = category;}
void setSubCategory(int subCategory){DtSubCategory = subCategory;}
void setSpecific(int specific){DtSpecific = specific;}
void setExtra(int extra){DtExtra = extra;}

/* the funcs used to take longs, so for back-compat: */
void setKind(long kind){DtEntityKind = (int)kind;}
void setDomain(long domain){DtDomain = (int)domain;}
void setCountry(long country){DtCountry = (int)country;}
void setCategory(long category){DtCategory = (int)category;}
void setSubCategory(long subCategory){DtSubCategory = (int)subCategory;}
void setSpecific(long specific){DtSpecific = (int)specific;}
void setExtra(long extra){DtExtra = (int)extra;}

DtBoolean matchPattern(const DtEntityType &pat) const;
DtBoolean DtTypeMatchPattern(const DtEntityType &pat) const
{ return matchPattern(pat); }

/* == returns 1 if they are equal, 0 if they are not equal */
int match(const DtEntityType &type) const
{return operator==(type);}
int operator==(const DtEntityType &type) const;

/* != returns 0 if they are equal, nonzero if they are not equal */
int operator!=(const DtEntityType &type) const;
};

class DtRadioEntityType : public DtEntityType
{
public:
    DtRadioEntityType();
    DtRadioEntityType(const char* p);
    DtRadioEntityType(int kind, int domain, int country, int category,
        int nomenclatureVersion, int nomenclature);
    DtRadioEntityType(DtNetRadioEntityType netType);
    void setFromNet(DtNetRadioEntityType netType);

    int nomenclatureVersion() const;
    int nomenclature() const;

    void setNomenclatureVersion(int version);
    void setNomenclature(int nom);

    char *string() const;

    static const DtRadioEntityType& nullType();

    operator DtNetRadioEntityType() const;
};

class DtBurstDescriptor
{
public:

```

```
DtBurstDescriptor()
{
    DtEntityType type;
    init(&type, DtWarheadOther, DtFuzeOther, 0, 0);
}

DtBurstDescriptor(DtNetBurstDescriptor *netBurst)
{
    DtEntityType type(netBurst->DtMunition);
    init(&type,
        DtWarheadType((int)netBurst->DtWarhead),
        DtDetonatorFuze((int)netBurst->DtFuze),
        (int) netBurst->DtQuantity,
        (int) netBurst->DtRate);
}

DtBurstDescriptor(DtEntityType *type, DtWarheadType warhead,
    DtDetonatorFuze fuze, DtU16 quantity, DtU16 rate)
{
    init(type, warhead, fuze, quantity, rate);
}

operator DtNetBurstDescriptor () const
{
    DtNetBurstDescriptor b;
    b.DtMunition = DtMunition;
    b.DtWarhead = DtWarhead;
    b.DtFuze = DtFuze;
    b.DtQuantity = DtQuantity;
    b.DtRate = DtRate;
    return(b);
}

void init(DtEntityType *type, DtWarheadType warhead,
    DtDetonatorFuze fuze, int quantity, int rate)
{
    DtMunition = *type;
    DtWarhead = warhead;
    DtFuze = fuze;
    DtQuantity = quantity;
    DtRate = rate;
}

public:
    DtEntityType    DtMunition;
    DtWarheadType   DtWarhead;
    DtDetonatorFuze DtFuze;
    int             DtQuantity;
    int             DtRate;
};
```


hRadioRcvrPub.h

```

#if DtHLA

#ifndef DtRadioReceiverPublisher_H_
#define DtRadioReceiverPublisher_H_

#include "hObjPub.h"

// class DtRadioReceiverPublisher:
//
// Instances of DtRadioReceiverPublisher are used for local or source
// simulations of radio receivers, (i.e. these entities are not dependent
// on data being produced elsewhere on the network for their existence).
// These radio receivers send data to the network.

class DtRadioReceiverPublisher : public DtObjectPublisher
{
public:
    // Function prototype for functions that create
    // DtRadioReceiverRepositories
    typedef DtRadioReceiverRepository* (*DtStateRepCreator)();

public:
    // constructor
    DtRadioReceiverPublisher(RTI::ObjectClassHandle h,
        DtExerciseConn* conn, const char* name = NULL);

    // constructor
    DtRadioReceiverPublisher(DtExerciseConn* conn, const char* name = NULL);

    // destructor
    virtual ~DtRadioReceiverPublisher();

    // Returns a pointer to the DtRadioReceiverRepository - through
    // which you can set or inspect current state information.
    virtual DtRadioReceiverRepository* receiverRep();

    // Synonym for receiverRep()
    virtual DtRadioReceiverRepository* rsr();

    // Returns the object class handle to use for this class.
    virtual RTI::ObjectClassHandle classToUse() const;

public:
    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

protected:
    // copy constructor
    DtRadioReceiverPublisher(const DtRadioReceiverPublisher& orig);

    // copy constructor
    DtRadioReceiverPublisher& operator=
        (const DtRadioReceiverPublisher& orig);

protected:
    static DtStateRepCreator theStateRepCreator;
};
#endif //DtRadioReceiverPublisher_H_

```

hRadioXmitPub.h

```
#if DtHLA

#ifndef DtRadioTransmitterPublisher_H_
#define DtRadioTransmitterPublisher_H_

#include "hObjPub.h"

// class DtRadioTransmitterPublisher:
//
// Instances of DtRadioTransmitterPublisher are used for local or source
// simulations of radio transmitters, (i.e. these entities are not dependent
// on data being produced elsewhere on the network for their existence).
// These radio transmitters send data to the network.

class DtRadioTransmitterPublisher : public DtObjectPublisher
{
public:

    // Function prototype for functions that create
    // DtRadioTransmitterRepositories
    typedef DtRadioTransmitterRepository* (*DtStateRepCreator)();

public:

    // constructor
    DtRadioTransmitterPublisher(RTI::ObjectClassHandle h,
        DtExerciseConn* conn, const char* name = NULL);

    // constructor
    DtRadioTransmitterPublisher(DtExerciseConn* conn, const char* name =
NULL);

    // destructor
    virtual ~DtRadioTransmitterPublisher();

    // Returns a pointer to the DtRadioTransmitterRepository - through
    // which you can set or inspect current state information.
    virtual DtRadioTransmitterRepository* radioXmitRep();

    // Synonym for radioXmitRep()
    virtual DtRadioTransmitterRepository* tsr();

    // Returns the object class handle to use for this class.
    virtual RTI::ObjectClassHandle classToUse() const;

public:

    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

protected:

    // copy constructor
    DtRadioTransmitterPublisher(const DtRadioTransmitterPublisher& orig);

    // copy constructor
    DtRadioTransmitterPublisher& operator=
        (const DtRadioTransmitterPublisher& orig);

protected:

    static DtStateRepCreator theStateRepCreator;
};
```

```

#endif //DtRadioTransmitterPublisher_H_
hRawBinRSDec.h

#if DtHLA

#include "chooseDefines.h"
#include "interDecoder.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_RAW_BIN_RADIO_SIGNAL_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtSignalInteraction, paramName)

#define DtDEFINE_SIMPLE_RAW_BIN_RADIO_SIGNAL_PARAM_DECODER(paramName, \
    netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER(DtRawBinaryRadioSignalDecoder, \
    DtSignalInteraction, paramName, netType, mutator)

#define DtDEFINE_SIMPLE_RAW_BIN_RADIO_SIGNAL_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
    DtRawBinaryRadioSignalDecoder, DtSignalInteraction, \
    paramName, netType, mutator, castExpr)

// class DtRawBinaryRadioSignalDecoder:
//
// Instances of DtRawBinaryRadioSignalDecoder are

class DtRawBinaryRadioSignalDecoder : public DtInteractionDecoder
{
public:
    // default constructor
    DtRawBinaryRadioSignalDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtRawBinaryRadioSignalDecoder();

    // copy constructor
    DtRawBinaryRadioSignalDecoder(
        const DtRawBinaryRadioSignalDecoder& orig);

    // assignment operator
    DtRawBinaryRadioSignalDecoder& operator=(
        const DtRawBinaryRadioSignalDecoder& orig);

protected:
    DtDECLARE_RAW_BIN_RADIO_SIGNAL_PARAM_DECODER(HostRadioID);
    DtDECLARE_RAW_BIN_RADIO_SIGNAL_PARAM_DECODER(DataRate);
    DtDECLARE_RAW_BIN_RADIO_SIGNAL_PARAM_DECODER(SignalDataLength);
    DtDECLARE_RAW_BIN_RADIO_SIGNAL_PARAM_DECODER(SignalData);
    DtDECLARE_RAW_BIN_RADIO_SIGNAL_PARAM_DECODER(TacticalDataLinkType);
    DtDECLARE_RAW_BIN_RADIO_SIGNAL_PARAM_DECODER(TDLMessageCount);
};
#endif // DtHLA

```

hRawBinRSEnc.h

```
#if DtHLA

#include "chooseDefines.h"
#include "interEncoder.h"

#define DtDECLARE_RAW_BINARY_RADIO_SIGNAL_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtSignalInteraction, paramName)

#define DtDEFINE_SIMPLE_RAW_BINARY_RADIO_SIGNAL_PARAM_ENCODER(paramName, \
    netType, inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtRawBinaryRadioSignalEncoder, \
    DtSignalInteraction, paramName, netType, inspector)

// class DtRawBinaryRadioSignalEncoder:
//
// Instances of DtRawBinaryRadioSignalEncoder are
class DtRawBinaryRadioSignalEncoder : public DtInteractionEncoder
{
public:
    // default constructor
    DtRawBinaryRadioSignalEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtRawBinaryRadioSignalEncoder();

    // copy constructor
    DtRawBinaryRadioSignalEncoder(const DtRawBinaryRadioSignalEncoder&
orig);

    // assignment operator
    DtRawBinaryRadioSignalEncoder& operator=(
        const DtRawBinaryRadioSignalEncoder& orig);

protected:

    DtDECLARE_RAW_BINARY_RADIO_SIGNAL_PARAM_ENCODER(HostRadioID);
    DtDECLARE_RAW_BINARY_RADIO_SIGNAL_PARAM_ENCODER(DataRate);
    DtDECLARE_RAW_BINARY_RADIO_SIGNAL_PARAM_ENCODER(SignalDataLength);
    DtDECLARE_RAW_BINARY_RADIO_SIGNAL_PARAM_ENCODER(SignalData);
    DtDECLARE_RAW_BINARY_RADIO_SIGNAL_PARAM_ENCODER(TacticalDataLinkType);
    DtDECLARE_RAW_BINARY_RADIO_SIGNAL_PARAM_ENCODER(TDLMessageCount);
};
#endif // DtHLA
```

hRefBeamList.h

```

#if DtHLA

#include "hRefObjList.h"
#include "hReflBeam.h"

// class DtReflectedEmitterBeamList:
//
// Instances of DtReflectedEmitterBeamList manage the list of
// of remote EmitterBeams in the exercise.

class DtReflectedEmitterBeamList : public DtReflectedObjectList
{
public:
    // Constructor.
    DtReflectedEmitterBeamList(DtExerciseConn* exConn);

    // Constructor - only subscribe to the RTI::ObjectClassHandles
    // in handleList
    DtReflectedEmitterBeamList(
        DtExerciseConn* exConn, const DtList *handleList);

    // destructor
    virtual ~DtReflectedEmitterBeamList();

    // Look up beam by identifier.
    virtual DtReflectedEmitterBeam* lookup(RTI::ObjectHandle id);

#if !DtRTI103
    // Look up beam by name.
    virtual DtReflectedEmitterBeam* lookup(
        const DtGlobalObjectDesignator& objName);
#endif

    // Get the first/last beam in the list.
    virtual DtReflectedEmitterBeam* first() const;
    virtual DtReflectedEmitterBeam* last() const;

    // EmitterBeam added callback registration/deregistration
    virtual void addEmitterBeamAdditionCallback(
        DtEmitterBeamCallbackFunction cb, void* userData);
    virtual void removeEmitterBeamAdditionCallback(
        DtEmitterBeamCallbackFunction cb, void* userData);

    // EmitterBeam removed callback registration/deregistration
    virtual void addEmitterBeamRemovalCallback(
        DtEmitterBeamCallbackFunction cb, void* userData);
    virtual void removeEmitterBeamRemovalCallback(
        DtEmitterBeamCallbackFunction cb, void* userData);

protected:
    // copy constructor
    DtReflectedEmitterBeamList(const DtReflectedEmitterBeamList& orig);

    // assignment operator
    DtReflectedEmitterBeamList& operator=(
        const DtReflectedEmitterBeamList& orig);

    // Override of virtual function. Here, we create a DtReflectedEmitter-
    Beam.
    virtual DtReflectedObject* newReflectedObject(DtHlaObject* obj) const;

```

```
// Called from processDiscoverObject() after a remote beam has
// been added to the list. Default implementation prints a message.
// Subclasses can override this function to add functionality on
// addition. obj will be an instance of a class derived from
// DtReflectedObj.
virtual void objectAdded(DtReflectedObject* obj);

// Returns a DtList of RTI::ObjectClassHandles that represent the list
// of object classes to which this list should subscribe.
virtual const DtList* dfltHandleList();

protected:
    DtList myDfltHandleList;
};
#endif    // DtHLA
```

hRefDesigList.h

```

#if DtHLA

#include "hRefObjList.h"
#include "hRefDesigntr.h"

// class DtReflectedDesignatorList:
//
// Instances of DtReflectedDesignatorList manage the list of
// of remote Designators in the exercise.

class DtReflectedDesignatorList : public DtReflectedObjectList
{
public:

    // Constructor - obtains the set of object classes to subscribe
    // to from the exConn's fomMapper.
    DtReflectedDesignatorList(DtExerciseConn* exConn);

    // Constructor - only subscribe to the RTI::ObjectClassHandles
    // in handleList
    DtReflectedDesignatorList(
        DtExerciseConn* exConn, const DtList *handleList);

    // destructor
    virtual ~DtReflectedDesignatorList();

    // Look up designator by identifier.
    virtual DtReflectedDesignator* lookup(RTI::ObjectHandle id);

#if !DtRTI103
    // Look up designator by name.
    virtual DtReflectedDesignator* lookup(
        const DtGlobalObjectDesignator& objName);
#endif

    // Get the first/last designator in the list.
    virtual DtReflectedDesignator* first() const;
    virtual DtReflectedDesignator* last() const;

    // Designator added callback registration/deregistration
    virtual void addDesignatorAdditionCallback(
        DtDesignatorCallbackFunction cb, void* userData);
    virtual void removeDesignatorAdditionCallback(
        DtDesignatorCallbackFunction cb, void* userData);

    // Designator removed callback registration/deregistration
    virtual void addDesignatorRemovalCallback(
        DtDesignatorCallbackFunction cb, void* userData);
    virtual void removeDesignatorRemovalCallback(
        DtDesignatorCallbackFunction cb, void* userData);

protected:

    // copy constructor
    DtReflectedDesignatorList(const DtReflectedDesignatorList& orig);

    // assignment operator
    DtReflectedDesignatorList& operator=(const DtReflectedDesignatorList&
orig);

    // Override of virtual function. Here, we create a DtReflectedDesig-
    // nator.
    virtual DtReflectedObject* newReflectedObject(DtHlaObject* obj) const;

```

```
// Called from processDiscoverObject() after a remote designator has
// been added to the list. Default implementation prints a message.
// Subclasses can override this function to add functionality on
// addition. obj will be an instance of a class derived from DtReflect-
edObj.
virtual void objectAdded(DtReflectedObject* obj);

// Returns a DtList of RTI::ObjectClassHandles that represent the list
// of object classes to which this list should subscribe.
virtual const DtList* dfltHandleList();

protected:
    DtList myDfltHandleList;
};
#endif // DtHLA
```


hRefEmSysList.h

```

#if DtHLA

#include "hRefObjList.h"
#include "hRefEmitrSys.h"
#include "hRefBeamList.h"

// class DtReflectedEmitterSystemList:
//
// Instances of DtReflectedEmitterSystemList manage the list of
// of remote EmitterSystems in the exercise.

class DtReflectedEmitterSystemList : public DtReflectedObjectList
{
public:
    // Constructor
    DtReflectedEmitterSystemList(DtExerciseConn* exConn);

    // Constructor - only subscribe to the RTI::ObjectClassHandles
    // in handleList
    DtReflectedEmitterSystemList(DtExerciseConn* exConn,
        const DtList *handleList);

    // destructor
    virtual ~DtReflectedEmitterSystemList();

    // Look up emitter system by identifier.
    virtual DtReflectedEmitterSystem* lookup(RTI::ObjectHandle id);

#if !DtRTI103
    // Look up emitter system by name.
    virtual DtReflectedEmitterSystem* lookup(
        const DtGlobalObjectDesignator& objName);
#endif

#ifdef
    // Get the first/last emitter system in the list.
    virtual DtReflectedEmitterSystem* first() const;
    virtual DtReflectedEmitterSystem* last() const;

    // EmitterSystem added callback registration/deregistration
    // Called when an emitter system is added due to discovery of
    // a system OR beam.
    virtual void addEmitterSystemAdditionCallback(
        DtEmitterSystemCallbackFunction cb, void* userData);
    virtual void removeEmitterSystemAdditionCallback(
        DtEmitterSystemCallbackFunction cb, void* userData);

    // EmitterSystem removed callback registration/deregistration
    // Called when an emitter system is removed due to discovery of
    // a system OR beam.
    virtual void addEmitterSystemRemovalCallback(
        DtEmitterSystemCallbackFunction cb, void* userData);
    virtual void removeEmitterSystemRemovalCallback(
        DtEmitterSystemCallbackFunction cb, void* userData);

protected:
    // copy constructor
    DtReflectedEmitterSystemList(const DtReflectedEmitterSystemList& orig);

    // assignment operator

```

```
DtReflectedEmitterSystemList& operator=
    (const DtReflectedEmitterSystemList& orig);

// Process the discovery of a remote emitter system on the network.
// Can't use base version, since we need to account for systems that
// are already in the list by virtue of discovered beams.
virtual void processDiscoverObject(DtHlaObject* obj);

// Process the removal of a remote emitter system from the network.
// Removes the remote emitter system from this list.
// Can't use base version, since we need to keep systems that
// still have live beams.
virtual void processRemoveObject(DtHlaObject* obj);

// Called by similar callback functions registered with myBeamList
// on beam additions/removals. Addition function adds the beam to the
// set of orphan beams (since we don't know what system he's part of),
// the removal function removes the beam's beamSR from his system's
// systemSR.
virtual void processEmitterBeamAddition(DtReflectedEmitterBeam* beam);
virtual void processEmitterBeamRemoval(DtReflectedEmitterBeam* beam);

// Check if any orphanBeams have non-NULL emittingSystemId. If so,
// add them to their appropriate system (creating one if it doesn't
// exist), and delete them from orphan list.
virtual void resolveOrphans();

// Override of virtual function. Here, we create a DtReflectedEmitter-
System.
virtual DtReflectedObject* newReflectedObject(DtHlaObject* obj) const;

// Create a remote emitter system object for the HLA object.
// Caller is responsible for deletion.
virtual DtReflectedEmitterSystem* newReflectedEmitterSystem(
    const DtGlobalObjectDesignator& des) const;

// Called from processDiscoverObject() after a remote transmitter has
// been added to the list. Default implementation prints a message.
// Subclasses can override this function to add functionality on
// addition. obj will be an instance of a class derived from DtReflect-
edObj.
virtual void objectAdded(DtReflectedObject* obj);

// Returns whether or not the object should be timed out.
// Assumes that timeout processing is enabled. Can't use base version
// since we don't want to time out systems with live beams.
virtual DtBoolean shouldBeTimedOut(DtReflectedObject* object);

// Returns a DtList of RTI::ObjectClassHandles that represent the list
// of object classes to which this list should subscribe.
virtual const DtList* dfltHandleList();

protected:

// Callback functions registered with myBeamList on beam additions, and
// removals. Calls processEmitterBeamAddition/Removal.
static void emitterBeamAdditionCb(
    DtReflectedEmitterBeam* beam, void* usr);
static void emitterBeamRemovalCb(
    DtReflectedEmitterBeam* beam, void* usr);

// Callback that calls resolveOrphans.
static void resolveOrphansPostDrainCb(void *usr);

protected:

    DtList myDfltHandleList;
```

```
    DtList myOrphanBeams;  
    DtReflectedEmitterBeamList myBeamList;  
};  
#endif    //DtHLA
```

hRefEntHList.h

```
#if DtHLA

#include "hashlist.h"

// class DtReflectedEntityHashlist:
//
// Instances of DtReflectedEntityHashlist are used to maintain a hashable
// list of remote entities.

class DtReflectedEntityHashlist : public DtIntrusiveHashlist
{
public:
    // default constructor
    DtReflectedEntityHashlist();

    // destructor
    virtual ~DtReflectedEntityHashlist();

    // copy constructor
    DtReflectedEntityHashlist(const DtReflectedEntityHashlist& orig);

    // assignment operator
    DtReflectedEntityHashlist& operator=(const DtReflectedEntityHashlist&
orig);
    // Hash the key to a bucket.
    virtual unsigned hash(DtHashKey k) const;

protected:
    static const int theNumBuckets;
};
#endif
```

hRefIAggList.h

```

#if DtHLA

#include "hRefObjList.h"
#include "hReflectedAgg.h"

// class DtReflectedAggregateList:
//
// Instances of DtReflectedAggregateList are used to manage remote aggregates
// from the network.

class DtReflectedAggregateList : public DtReflectedObjectList
{
public:
    // Constructor - obtains the set of object classes to subscribe
    // to from the exConn's fomMapper.
    DtReflectedAggregateList(DtExerciseConn* exConn,
        DtBoolean smoothCapable = DtFalse);

    // Constructor - only subscribe to the RTI::ObjectClassHandles
    // in handleList
    DtReflectedAggregateList(DtExerciseConn* exConn,
        const DtList *handleList, DtBoolean smoothCapable = DtFalse);

    // destructor
    virtual ~DtReflectedAggregateList();

    // Look up aggregate by identifier.
    virtual DtReflectedAggregate* lookup(RTI::ObjectHandle id);

#if !DtRTI103
    // Look up aggregate by name.
    virtual DtReflectedAggregate* lookup(
        const DtGlobalObjectDesignator& objName);
#endif

    // Get the first/last entity in the list.
    virtual DtReflectedAggregate* first() const;
    virtual DtReflectedAggregate* last() const;

    // Aggregate added callback registration/deregistration
    virtual void addAggregateAdditionCallback(
        DtAggregateCallbackFunction cb, void* userData);
    virtual void removeAggregateAdditionCallback(
        DtAggregateCallbackFunction cb, void* userData);

    // Aggregate removed callback registration/deregistration
    virtual void addAggregateRemovalCallback(
        DtAggregateCallbackFunction cb, void* userData);
    virtual void removeAggregateRemovalCallback(
        DtAggregateCallbackFunction cb, void* userData);

protected:
    // copy constructor
    DtReflectedAggregateList(const DtReflectedAggregateList& orig);

    // assignment operator
    DtReflectedAggregateList& operator=(const DtReflectedAggregateList&
orig);

```

```
// Asks the FOM Mapper for the DtList of RTI::ObjectClassHandles to use.
virtual const DtList* dfltHandleList();

// Override of virtual function. Calls newReflectedAggregate.
virtual DtReflectedObject* newReflectedObject(DtHlaObject* obj) const;

// Create a remote aggregate object for the HLA object.
// Caller is responsible for deletion.
virtual DtReflectedAggregate* newReflectedAggregate(
    DtHlaObject* obj) const;

// Virtual function override.
// Calls removeAndDelete(DtReflectedAggregate*....)
virtual void removeAndDelete(DtReflectedObject* object, char* msg);

// Default calls down to base class version, but can be overridden.
virtual void removeAndDelete(
    DtReflectedAggregate* aggregate, char* msg = NULL);

// Called from processDiscoverObject() after a remote transmitter has
// been added to the list. Calls aggregateAdded.
virtual void objectAdded(DtReflectedObject* obj);

// Called by objectAdded (which is called from processDiscoverObject())
// after a remote aggregate has been added to the list. Default
// implementation prints a message. Subclasses can override this func-
tion
// to add functionality on addition.
virtual void aggregateAdded(DtReflectedAggregate* ent);

protected:
    DtBoolean mySmoothCapable;
};
#endif
```

hReflBeam.h

```

#if DtHLA

#ifndef DtReflectedEmitterBeam_H_
#define DtReflectedEmitterBeam_H_

#include "emitttrBeamSR.h"
#include "hReflectedObj.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtEmitterBeamCallbackFunction)(
    DtReflectedEmitterBeam* obj, void* userData);

// class DtReflectedEmitterBeam:
//
// Instances of DtReflectedEmitterBeam represent EmitterBeam
// object simulated by remote applications. State data can be viewed
// through it's state repository.

class DtReflectedEmitterBeam : public DtReflectedObject
{
public:

    // Function prototype for functions that create DtEmitterBeamRepositories
    typedef DtEmitterBeamRepository* (*DtStateRepCreator)();

public:

    // constructor
    DtReflectedEmitterBeam(DtHlaObject* obj, DtExerciseConn* conn);

    // destructor
    virtual ~DtReflectedEmitterBeam();

    // Get the beam's state repository.
    virtual DtEmitterBeamRepository* beamRep() const;

    // Synonym for beamRep()
    virtual DtEmitterBeamRepository* bsr() const;

    // Set beam state repository. Only necessary when you want to
    // use something other than the default, which is created internally.
    virtual void setBsr(DtEmitterBeamRepository* bsr);

    // Get the next remote beam.
    virtual DtReflectedEmitterBeam* next() const;

    // Get the previous remote beam.
    virtual DtReflectedEmitterBeam* prev() const;

    // Get the next remote beam.
    // Will cycle back to the first remote beam.
    virtual DtReflectedEmitterBeam* wrapNext() const;

    // Get the previous remote beam.
    // Will cycle back to the last remote beam.
    virtual DtReflectedEmitterBeam* wrapPrev() const;

    // beam state callback registration/deregistration
    virtual void addPostUpdateCallback(
        DtEmitterBeamCallbackFunction cb, void* userData);

    virtual void removePostUpdateCallback(
        DtEmitterBeamCallbackFunction cb, void* userData);

```

```
public:
    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);
protected:
    // copy constructor
    DtReflectedEmitterBeam(const DtReflectedEmitterBeam& orig);
    // assignment operator
    DtReflectedEmitterBeam& operator=(
        const DtReflectedEmitterBeam& orig);
protected:
    DtEmitterBeamRepository* myConstructedBsr;
protected:
    static DtStateRepCreator theStateRepCreator;
};
#endif //DtReflectedEmitterBeam_H_
```


hReflDesigntr.h

```

#ifdef DtHLA

#include "designatorSR.h"
#include "hReflectedObj.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtDesignatorCallbackFunction)(
    DtReflectedDesignator* obj, void* userData);

// class DtReflectedDesignator:
//
// Instances of DtReflectedDesignator are

class DtReflectedDesignator : public DtReflectedObject
{
public:

    // Function prototype for functions that create DtDesignatorRepositories
    typedef DtDesignatorRepository* (*DtStateRepCreator)();

public:

    // constructor
    DtReflectedDesignator(DtHlaObject* obj, DtExerciseConn* conn);

    // destructor
    virtual ~DtReflectedDesignator();

    // Get the designator's state repository.
    virtual DtDesignatorRepository* designatorRep() const;

    // Synonym for designatorRep()
    virtual DtDesignatorRepository* dsr() const;

    // Get the next remote designator.
    virtual DtReflectedDesignator* next() const;

    // Get the previous remote designator.
    virtual DtReflectedDesignator* prev() const;

    // Get the next remote designator.
    // Will cycle back to the first remote designator.
    virtual DtReflectedDesignator* wrapNext() const;

    // Get the previous remote designator.
    // Will cycle back to the last remote designator.
    virtual DtReflectedDesignator* wrapPrev() const;

    // designator state callback registration/deregistration
    virtual void addPostUpdateCallback(
        DtDesignatorCallbackFunction cb, void* userData);

    virtual void removePostUpdateCallback(
        DtDesignatorCallbackFunction cb, void* userData);

public:

    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

protected:

    // copy constructor

```

```
DtReflectedDesignator(const DtReflectedDesignator& orig);  
    // assignment operator  
    DtReflectedDesignator& operator=(const DtReflectedDesignator& orig);  
protected:  
    static DtStateRepCreator theStateRepCreator;  
};  
#endif    //DtHLA
```

hReflectedAgg.h

```

#if DtHLA

#include "aggregateSR.h"
#include "hReflectedObj.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtAggregateCallbackFunction)(
    DtReflectedAggregate* obj, void* userData);

// class DtReflectedAggregate:
//
// Instances of DtReflectedAggregate are used to represent a simulation
// aggregate reflected across a network (i.e. this is not an full
// simulation running on this machine).

class DtReflectedAggregate : public DtReflectedObject
{
public:
    // Function prototype for functions that create DtAggregateStateRepositories
    typedef DtAggregateStateRepository* (*DtStateRepCreator)();

public:
    // constructor
    DtReflectedAggregate(DtHlaObject* obj, DtExerciseConn* conn);

    // destructor
    virtual ~DtReflectedAggregate();

    // Returns a pointer to the AggregateStateRepository - through
    // which you can inspect current state information.
    virtual DtAggregateStateRepository* aggregateStateRep() const;

    // Synonym for aggregateStateRep()
    virtual DtAggregateStateRepository* asr() const;

    // Get the next remote aggregate.
    virtual DtReflectedAggregate* next() const;

    // Get the previous remote aggregate.
    virtual DtReflectedAggregate* prev() const;

    // Get the next remote aggregate.
    // Will cycle back to the first remote aggregate.
    virtual DtReflectedAggregate* wrapNext() const;

    // Get the previous remote aggregate.
    // Will cycle back to the last remote aggregate.
    virtual DtReflectedAggregate* wrapPrev() const;

    // Aggregate state callback registration/deregistration
    virtual void addPostUpdateCallback(
        DtAggregateCallbackFunction cb, void* userData);
    virtual void removePostUpdateCallback(
        DtAggregateCallbackFunction cb, void* userData);

public:
    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

```

```
public:
    // Get the aggregate HLA object's id
    // For backwards compatibility. Use id() instead.
    virtual RTI::ObjectHandle aggregateObjectId() const;

protected:
    // copy constructor
    DtReflectedAggregate(const DtReflectedAggregate& orig);

    // assignment operator
    DtReflectedAggregate& operator=(const DtReflectedAggregate& orig);

protected:
    static DtStateRepCreator theStateRepCreator;
};
#endif // DtHLA
```

hReflectedEnt.h

```

#if DtHLA

#include "entitySR.h"
#include "hReflectedObj.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtEntityCallbackFunction)(
    DtReflectedEntity* obj, void* userData);

// class DtReflectedEntity:
//
// Instances of DtReflectedEntity are used to represent a simulation
// entity reflected across a network (i.e. this is not an full
// simulation running on this machine).

class DtReflectedEntity : public DtReflectedObject
{
public:

    // Function prototype for functions that create DtEntityStateRepositories
    typedef DtEntityStateRepository* (*DtStateRepCreator)();

public:

    // constructor
    DtReflectedEntity(DtHlaObject* obj, DtExerciseConn* conn);

    // destructor
    virtual ~DtReflectedEntity();

    // Returns a pointer to the EntityStateRepository - through
    // which you can inspect current state information.
    virtual DtEntityStateRepository* entityStateRep() const;

    // Synonym for entityStateRep()
    virtual DtEntityStateRepository* esr() const;

    // Get the next remote entity.
    virtual DtReflectedEntity* next() const;

    // Get the previous remote entity.
    virtual DtReflectedEntity* prev() const;

    // Get the next remote entity.
    // Will cycle back to the first remote entity.
    virtual DtReflectedEntity* wrapNext() const;

    // Get the previous remote entity.
    // Will cycle back to the last remote entity.
    virtual DtReflectedEntity* wrapPrev() const;

    // Entity state callback registration/deregistration
    virtual void addPostUpdateCallback(
        DtEntityCallbackFunction cb, void* userData);
    virtual void removePostUpdateCallback(
        DtEntityCallbackFunction cb, void* userData);

public:

    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

public:

```

```
// For backwards compatibility. Use DtStateRepCreator instead.
typedef DtStateRepCreator DtEsrCreator;

// For backwards compatibility. Use id() instead.
virtual RTI::ObjectHandle entityObjectId() const;

// For backwards compatibility. Use setStateRepCreator instead.
static DtEsrCreator setEsrCreator(DtEsrCreator creator);

protected:

    // copy constructor
    DtReflectedEntity(const DtReflectedEntity& orig);

    // assignment operator
    DtReflectedEntity& operator=(const DtReflectedEntity& orig);

protected:

    static DtStateRepCreator theStateRepCreator;
};
#endif // DtHLA
```

hReflectedObj.h

```

#if DtHLA

#include "chooseDefines.h"
#include "vllist.h"
#include "vlTime.h"
#include "globalObjDes.h"
#include "hlaObjWithSR.h"
#include "refObjCbInfo.h"
#include "hostStructs.h"
#include "str.h"

// class DtReflectedObject:
//
// Instances of DtReflectedObject are used to represent a simulation
// entity reflected across a network (i.e. this is not an full
// simulation running on this machine).

class DtReflectedObject : public DtListedObj
{
public:
    // Function prototype for functions that create DtStateRepositories
    typedef DtStateRepository* (*DtBaseStateRepCreator)();

public:
    // constructor
    DtReflectedObject(DtHlaObject* obj,
        DtExerciseConn* conn, DtBaseStateRepCreator creator);

    // constructor
    DtReflectedObject(const DtObjectId& id,
        const DtGlobalObjectDesignator& des,
        DtExerciseConn* conn, DtBaseStateRepCreator creator);

    // destructor
    virtual ~DtReflectedObject();

    // Returns a pointer to the StateRepository - through
    // which you can inspect current state information.
    virtual DtStateRepository* stateRep() const;

    // Synonym for stateRep()
    virtual DtStateRepository* sr() const;

    // Returns a pointer to the Decoder being used to decode
    // state messages into our SR.
    virtual DtHlaStateDecoder* decoder();

    // Get the entity's hlaObject
    virtual DtHlaObject* hlaObject() const;

    // Get the HLA object's id
    virtual const DtObjectId& objectId() const;

    // Get the HLA object's id (local ID).
    virtual const DtObjectId& id() const;

    // Get the HLA object name
    virtual const DtString& name() const;

    // Get the HLA object's global object designator -

```

```
// the object name in RTI1.3.3, but the objectId in RTI1.0.3.
virtual const DtGlobalObjectDesignator& globalId() const;

// Returns the value of simulation time at the time
// an update was last received for the entity.
virtual DtTime lastSimTimeUpdated() const;

// Object state callback registration/deregistration
virtual void baseAddPostUpdateCallback(
    DtReflectedObjectCallbackFunction cb, void* userData);
virtual void baseRemovePostUpdateCallback(
    DtReflectedObjectCallbackFunction cb, void* userData);

// Get the next remote object.
virtual DtReflectedObject* nextObj() const;

// Get the previous remote object.
virtual DtReflectedObject* prevObj() const;

// Get the next remote object.
// Will cycle back to the first remote object.
virtual DtReflectedObject* wrapNextObj() const;

// Get the previous remote object.
// Will cycle back to the last remote object.
virtual DtReflectedObject* wrapPrevObj() const;

// Sets self up to manage obj.
virtual void setHlaObject(DtHlaObject* obj);

protected:

    // copy constructor
    DtReflectedObject(const DtReflectedObject& orig);

    // assignment operator
    DtReflectedObject& operator=(const DtReflectedObject& orig);

    // Request that the owner of this entity send an attribute update
    virtual void requestUpdate();

    // process the post update callbacks
    virtual void processPostUpdateCallbacks();

    // Ends managing of current HLA object.
    virtual void detachHlaObject();

protected:

    // calls requestUpdate
    static void requestUpdatePostDrainCb(void* usr);

    // callback from HlaObject to process post update callbacks
    static void processPostUpdateCallbacksCb(
        const DtStateMsg& stateMsg,
        DtHlaObject* obj,
        void* usr);

protected:

    DtExerciseConn* myConn;
    DtHlaObjectWithStateRep* myHlaObj;
    DtStateRepository* myStateRep;
    DtHlaStateDecoder* myDecoder;
    DtBoolean myRequestCbRegistered;
    DtIntrusiveList myStateUpdateCallbacks;
    DtObjectId myId;
```



```
    DtString myName;  
};  
#endif
```

hRefObjList.h

```
#if DtHLA

#include "chooseDefines.h"
#include "objIdHashLst.h"
#include "RTI.hh"
#include "MachineTypes.h"
#include "vlTime.h"
#include "globalObjDes.h"
#include "refObjCbInfo.h"

// class DtReflectedObjectList:
//
// Instances of DtReflectedObjectList are list of instances of classes
// derived from DtReflectedObject. A DtReflectedObjectList manages the
// comings
// and goings of its reflected objects based on discoveries and removes
// received from the RTI.

class DtReflectedObjectList
{
public:

    // Constructor
    DtReflectedObjectList(DtExerciseConn* exConn);

    // destructor
    virtual ~DtReflectedObjectList();

    // Look up object by identifier.
    virtual DtReflectedObject* lookupObj(RTI::ObjectHandle id);

    // Look up object by global ID (which is the object name in RTI1.3.3).
    virtual DtReflectedObject* lookupObj(
        const DtGlobalObjectDesignator& objName);

    // Get the first/last object in the list.
    virtual DtReflectedObject* firstObj() const;
    virtual DtReflectedObject* lastObj() const;

    // Get number of objects
    virtual int count() const;

    // Get my exerciseConn.
    virtual DtExerciseConn *exerciseConn() const;

    // Register callbacks with exercise connection so that we get
    // notified when an object of this class is discovered. Automatically
    // called for all object classes passed to the constructor, or for
    // the default set of objects classes if the ctor without a handleList
    // is used. But if you want to handle your own subscription and
    // update requests, pass an empty handleSet to the constructor,
    // use DtObjectClassDesc's registerInterest and requestUpdates,
    // then use this function to tell the ROL to manage the objects that
    // get discovered.
    virtual void manageObjectClass(RTI::ObjectClassHandle handle);

    // Have the application act as though we had never discovered the
    // object. We call the RTI's localDelete service, and delete the
    // DtHlaObject, which causes any DtReflectedObjects associated with
    // id (in this list and in any others) to be removed. The object
    // might be re-discovered later.
    virtual void removeObject(const DtObjectId& id);

}
```

```

// Object added callback registration/deregistration
virtual void addObjectAdditionCallback(
    DtReflectedObjectCallbackFunction cb, void* userData);
virtual void removeObjectAdditionCallback(
    DtReflectedObjectCallbackFunction cb, void* userData);

// Object removed callback registration/deregistration
virtual void addObjectRemovalCallback(
    DtReflectedObjectCallbackFunction cb, void* userData);
virtual void removeObjectRemovalCallback(
    DtReflectedObjectCallbackFunction cb, void* userData);

// Print the SRs of all objects
virtual void print();

// Set whether or not timeout processing occurs.
// Defaults to off.
virtual void setTimeoutProcessing(DtBoolean onOff);

// Set/Get the remote object timeout period.
virtual void setTimeoutInterval(DtTime time);
virtual DtTime timeoutInterval();

public:

    // Set/Get flag that determines whether we request updates
    // from all objects of the object classes we subscribe to.
    // Default is true.
    static void setRequestClassUpdateFlag(DtBoolean onOff);
    static DtBoolean requestClassUpdateFlag();

    // Set/Get flag that determines whether we request updates
    // from an object as we discover it. Default is true.
    static void setRequestObjectUpdateFlag(DtBoolean onOff);
    static DtBoolean requestObjectUpdateFlag();

protected:

    // copy constructor
    DtReflectedObjectList(const DtReflectedObjectList& orig);

    // assignment operator
    DtReflectedObjectList& operator=(const DtReflectedObjectList& orig);

    // Process the discovery of a remote object on the network.
    // Adds the remote object to this list.
    virtual void processDiscoverObject(DtHlaObject* obj);

    // Process the removal of a remote object from the network.
    // Removes the remote object from this list.
    virtual void processRemoveObject(DtHlaObject* obj);

    // Handle the receipt of attributes for a remote object which
    // was timed out. Reinstates the remote object in this list.
    virtual void processTimedOutObjectReflect(const DtStateMsg& msg,
        DtHlaObject* obj);

    // Remove any remote objects which have not been updated during
    // the timeout period. Gets called automatically by a postDrainCallback
    // registered by the constructor. Users should not need to call this.
    virtual void doTimeouts();

    // Create a remote object object for the HLA object.
    // Caller is responsible for deletion.
    virtual DtReflectedObject* newReflectedObject(DtHlaObject* obj) const =
0;

```

```
// Remove the remote object from the list if found and delete it.
// Prints a message on deletion if specified.
virtual void removeAndDelete(DtReflectedObject* obj,
    char* msg = NULL);

// Called from processDiscoverObject() after a remote object has
// been added to the list.
// Defaults to do nothing.
// Subclasses can override this function to add functionality on
// addition.
virtual void objectAdded(DtReflectedObject* object);

// Returns whether or not the object should be timed out.
// Assumes that timeout processing is enabled.
virtual DtBoolean shouldBeTimedOut(DtReflectedObject* object);

// Times out object by removing it from the remote object list and
// deleting it. Manages callbacks to reinstate remote object
// upon receiving attribute updates for the timed out object.
virtual void timeOutObject(DtReflectedObject* object);

// Manages callbacks and data used for reinstating a remote object
// which was timed out.
virtual void unTimeOutObject(DtHlaObject* obj);

// Register interest in this list of RTI::ObjectClassHandles.
virtual void init(const DtList* handleList);

// Empty the object list and the timeout list.
// Non-virtual since meant to be called from destructor.
void emptyLists();

// Process object addition/removal callbacks
virtual void processObjectAdditionCallbacks(
    DtReflectedObject* object);
virtual void processObjectRemovalCallbacks(
    DtReflectedObject* object);

protected:

    // Cast the void* to a DtReflectedObjectList* and calls
    // processDiscoverObject().
    static void discoverObjectCb(DtHlaObject* hlaObj, void* rel);

    // Cast the void* to a DtReflectedObjectList* and calls
    // processRemoveObject().
    static void removeObjectCb(DtHlaObject* hlaObj, void* rel);

    // Cast the void* to a DtReflectedObjectList* and calls
    // processTimedOutObjectReflect().
    static void timedOutObjectReflectCb(const DtStateMsg& msg,
        DtHlaObject* obj,
        void* usr);

    // calls doTimeouts
    static void doTimeoutsPostDrainCb(void* usr);

protected:

    DtExerciseConn* myExConn;
    DtObjectIdHashList myObjectList;
    DtList myTimedOutObjectList;
    DtList myHandleList;
    DtIntrusiveList myObjectAddedCallbacks;
    DtIntrusiveList myObjectRemovedCallbacks;
    DtTime myTimeoutInterval;
    DtBoolean myTimeoutProcessingFlag;
```

```
protected:
    static DtBoolean theRequestClassUpdateFlag;
    static DtBoolean theRequestObjectUpdateFlag;
};
#endif    // DtHLA
```

hRefLEmitrSys.h

```
#if DtHLA

#include "emitterSysSR.h"
#include "hReflectedObj.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtEmitterSystemCallbackFunction)(
    DtReflectedEmitterSystem* obj, void* userData);

// class DtReflectedEmitterSystem:
//
// Instances of DtReflectedEmitterSystem represent EmitterSystem
// object simulated by remote applications. State data can be viewed
// through it's state repository.

class DtReflectedEmitterSystem : public DtReflectedObject
{
public:
    // Function prototype for functions that create DtEmitterSystemRepository
    typedef DtEmitterSystemRepository* (*DtStateRepCreator)();

public:
    // constructor
    DtReflectedEmitterSystem(DtHlaObject* obj, DtExerciseConn* conn);

    // Constructor used to construct a placeholder system when we
    // discover a beam whose system hasn't been discovered yet.
    DtReflectedEmitterSystem(const DtObjectId& id,
        const DtGlobalObjectDesignator& des,
        DtExerciseConn* conn);

    // destructor
    virtual ~DtReflectedEmitterSystem();

    // Get the emitter system's state repository.
    virtual DtEmitterSystemRepository* emitterSysRep() const;

    // Synonym for emitterSysRep()
    virtual DtEmitterSystemRepository* esr() const;

    // Get the next remote emitterSystem.
    virtual DtReflectedEmitterSystem* next() const;

    // Get the previous remote emitterSystem.
    virtual DtReflectedEmitterSystem* prev() const;

    // Get the next remote emitterSystem.
    // Will cycle back to the first remote emitterSystem.
    virtual DtReflectedEmitterSystem* wrapNext() const;

    // Get the previous remote emitterSystem.
    // Will cycle back to the last remote emitterSystem.
    virtual DtReflectedEmitterSystem* wrapPrev() const;

    // emitterSystem state callback registration/deregistration
    virtual void addPostUpdateCallback(
        DtEmitterSystemCallbackFunction cb, void* userData);

    virtual void removePostUpdateCallback(
        DtEmitterSystemCallbackFunction cb, void* userData);

};

#endif
```

```
public:
    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

protected:
    // copy constructor
    DtReflectedEmitterSystem(const DtReflectedEmitterSystem& orig);

    // assignment operator
    DtReflectedEmitterSystem& operator=(
        const DtReflectedEmitterSystem& orig);

protected:
    static DtStateRepCreator theStateRepCreator;
};
#endif //DtReflectedEmitterSystem_H_
```

hRefEntList.h

```
#if DtHLA

#include "hRefObjList.h"
#include "hReflectedEnt.h"

// class DtReflectedEntityList:
//
// Instances of DtReflectedEntityList are used to manage remote entities
// from the network.

class DtReflectedEntityList : public DtReflectedObjectList
{
public:

    // Constructor - obtains the set of object classes to subscribe
    // to from the exConn's fomMapper.
    DtReflectedEntityList(DtExerciseConn* exConn,
        DtBoolean smoothCapable = DtFalse);

    // Constructor - only subscribe to the RTI::ObjectClassHandles
    // in handleList
    DtReflectedEntityList(DtExerciseConn* exConn,
        const DtList *handleList, DtBoolean smoothCapable = DtFalse);

    // destructor
    virtual ~DtReflectedEntityList();

    // Look up entity by identifier.
    virtual DtReflectedEntity* lookup(RTI::ObjectHandle id);

#if !DtRTI103

    // Look up entity by name.
    virtual DtReflectedEntity* lookup(const DtGlobalObjectDesignator&
objName);

#endif

    // Get the first/last entity in the list.
    virtual DtReflectedEntity* first() const;
    virtual DtReflectedEntity* last() const;

    // Entity added callback registration/deregistration
    virtual void addEntityAdditionCallback(
        DtEntityCallbackFunction cb, void* userData);
    virtual void removeEntityAdditionCallback(
        DtEntityCallbackFunction cb, void* userData);

    // Entity removed callback registration/deregistration
    virtual void addEntityRemovalCallback(
        DtEntityCallbackFunction cb, void* userData);
    virtual void removeEntityRemovalCallback(
        DtEntityCallbackFunction cb, void* userData);

protected:

    // copy constructor
    DtReflectedEntityList(const DtReflectedEntityList& orig);

    // assignment operator
    DtReflectedEntityList& operator=(const DtReflectedEntityList& orig);

    // Asks the FOM Mapper for the DtList of RTI::ObjectClassHandles to use.
```

```

virtual const DtList* dfltHandleList();

// Override of virtual function. Calls newReflectedEntity.
virtual DtReflectedObject* newReflectedObject(DtHlaObject* obj) const;

// Create a remote entity object for the HLA object.
// Caller is responsible for deletion.
virtual DtReflectedEntity* newReflectedEntity(DtHlaObject* obj) const;

// Virtual function override.
// Calls removeAndDelete(DtReflectedEntity*....)
virtual void removeAndDelete(DtReflectedObject* object, char* msg);

// Default calls down to base class version, but can be overridden.
virtual void removeAndDelete(DtReflectedEntity* entity, char* msg =
NULL);

// Called from processDiscoverObject() after a remote transmitter has
// been added to the list. Calls entityAdded.
virtual void objectAdded(DtReflectedObject* obj);

// Called by objectAdded (which is called from processDiscoverObject())
// after a remote entity has been added to the list. Default
// implementation prints a message. Subclasses can override this func-
tion
// to add functionality on addition.
virtual void entityAdded(DtReflectedEntity* ent);

protected:
    DtBoolean mySmoothCapable;
};
#endif

```

hRefRadRecvr.h

```
#if DtHLA

#ifndef DtReflectedRadioReceiver_H_
#define DtReflectedRadioReceiver_H_

#include "radioRecvrSR.h"
#include "hReflectedObj.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtRadioReceiverCallbackFunction)(
    DtReflectedRadioReceiver* obj, void* userData);

// class DtReflectedRadioReceiver:
//
// Instances of DtReflectedRadioReceiver represent RadioReceiver
// object simulated by remote applications. State data can be viewed
// through it's state repository.

class DtReflectedRadioReceiver : public DtReflectedObject
{
public:
    // Function prototype for functions that create DtRadioReceiverRepository
    typedef DtRadioReceiverRepository* (*DtStateRepCreator)();

public:
    // constructor
    DtReflectedRadioReceiver(DtHlaObject* obj, DtExerciseConn* conn);

    // destructor
    virtual ~DtReflectedRadioReceiver();

    // Get the receiver's state repository.
    virtual DtRadioReceiverRepository* receiverRep() const;

    // Synonym for receiverRep()
    virtual DtRadioReceiverRepository* rsr() const;

    // Get the next remote receiver.
    virtual DtReflectedRadioReceiver* next() const;

    // Get the previous remote receiver.
    virtual DtReflectedRadioReceiver* prev() const;

    // Get the next remote receiver.
    // Will cycle back to the first remote receiver.
    virtual DtReflectedRadioReceiver* wrapNext() const;

    // Get the previous remote receiver.
    // Will cycle back to the last remote receiver.
    virtual DtReflectedRadioReceiver* wrapPrev() const;

    // receiver state callback registration/deregistration
    virtual void addPostUpdateCallback(
        DtRadioReceiverCallbackFunction cb, void* userData);

    virtual void removePostUpdateCallback(
        DtRadioReceiverCallbackFunction cb, void* userData);

public:
    // Give us a function to be used for creating SRs.
```

```
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);  
protected:  
    // copy constructor  
    DtReflectedRadioReceiver(const DtReflectedRadioReceiver& orig);  
    // assignment operator  
    DtReflectedRadioReceiver& operator=(  
        const DtReflectedRadioReceiver& orig);  
protected:  
    static DtStateRepCreator theStateRepCreator;  
};  
#endif    //DtReflectedRadioReceiver_H_
```

hRefRadXmitr.h

```
#if DtHLA

#include "radioXmitSR.h"
#include "hReflectedObj.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtRadioTransmitterCallbackFunction)(
    DtReflectedRadioTransmitter* obj, void* userData);

// class DtReflectedRadioTransmitter:
//
// Instances of DtReflectedRadioTransmitter represent RadioTransmitter
// object simulated by remote applications. State data can be viewed
// through it's state repository.

class DtReflectedRadioTransmitter : public DtReflectedObject
{
public:
    // Function prototype for functions that create DtRadioTransmitterRepos-
    // itories
    typedef DtRadioTransmitterRepository* (*DtStateRepCreator)();

public:
    // constructor
    DtReflectedRadioTransmitter(DtHlaObject* obj, DtExerciseConn* conn);

    // destructor
    virtual ~DtReflectedRadioTransmitter();

    // Get the transmitter's state repository.
    virtual DtRadioTransmitterRepository* transmitterRep() const;

    // Synonym for transmitterRep()
    virtual DtRadioTransmitterRepository* tsr() const;

    // Get the next remote transmitter.
    virtual DtReflectedRadioTransmitter* next() const;

    // Get the previous remote transmitter.
    virtual DtReflectedRadioTransmitter* prev() const;

    // Get the next remote transmitter.
    // Will cycle back to the first remote transmitter.
    virtual DtReflectedRadioTransmitter* wrapNext() const;

    // Get the previous remote transmitter.
    // Will cycle back to the last remote transmitter.
    virtual DtReflectedRadioTransmitter* wrapPrev() const;

    // transmitter state callback registration/deregistration
    virtual void addPostUpdateCallback(
        DtRadioTransmitterCallbackFunction cb, void* userData);

    virtual void removePostUpdateCallback(
        DtRadioTransmitterCallbackFunction cb, void* userData);

public:
    // Give us a function to be used for creating SRs.
    static DtStateRepCreator setStateRepCreator(DtStateRepCreator creator);

};

#endif
```

```
protected:
    // copy constructor
    DtReflectedRadioTransmitter(const DtReflectedRadioTransmitter& orig);

    // assignment operator
    DtReflectedRadioTransmitter& operator=(
        const DtReflectedRadioTransmitter& orig);
protected:
    static DtStateRepCreator theStateRepCreator;
};
#endif    //DtReflectedRadioTransmitter_H_
```

hRefRadRvList.h

```
#if DtHLA

#include "hRefObjList.h"
#include "hRefRadRecvr.h"

// class DtReflectedRadioReceiverList:
//
// Instances of DtReflectedRadioReceiverList manage the list of
// of remote RadioReceivers in the exercise.

class DtReflectedRadioReceiverList : public DtReflectedObjectList
{
public:
    // Constructor
    DtReflectedRadioReceiverList(DtExerciseConn* exConn);

    // Constructor - only subscribe to the RTI::ObjectClassHandles
    // in handleList
    DtReflectedRadioReceiverList(
        DtExerciseConn* exConn, const DtList *handleList);

    // destructor
    virtual ~DtReflectedRadioReceiverList();

    // Look up receiver by identifier.
    virtual DtReflectedRadioReceiver* lookup(RTI::ObjectHandle id);

#if !DtRTI103
    // Look up receiver by name.
    virtual DtReflectedRadioReceiver* lookup(
        const DtGlobalObjectDesignator& objName);
#endif

    // Get the first/last receiver in the list.
    virtual DtReflectedRadioReceiver* first() const;
    virtual DtReflectedRadioReceiver* last() const;

    // RadioReceiver added callback registration/deregistration
    virtual void addRadioReceiverAdditionCallback(
        DtRadioReceiverCallbackFunction cb, void* userData);
    virtual void removeRadioReceiverAdditionCallback(
        DtRadioReceiverCallbackFunction cb, void* userData);

    // RadioReceiver removed callback registration/deregistration
    virtual void addRadioReceiverRemovalCallback(
        DtRadioReceiverCallbackFunction cb, void* userData);
    virtual void removeRadioReceiverRemovalCallback(
        DtRadioReceiverCallbackFunction cb, void* userData);

protected:
    // copy constructor
    DtReflectedRadioReceiverList(const DtReflectedRadioReceiverList& orig);

    // assignment operator
    DtReflectedRadioReceiverList& operator=(
        const DtReflectedRadioReceiverList& orig);

    // Override of virtual function.
    // Here, we create a DtReflectedRadioReceiver.
    virtual DtReflectedObject* newReflectedObject(DtHlaObject* obj) const;
```

```
// Called from processDiscoverObject() after a remote receiver has
// been added to the list. Default implementation prints a message.
// Subclasses can override this function to add functionality on
// addition. obj will be an instance of a class derived from
// DtReflectedObj.
virtual void objectAdded(DtReflectedObject* obj);

// Returns a DtList of RTI::ObjectClassHandles that represent the list
// of object classes to which this list should subscribe.
virtual const DtList* dfltHandleList();

protected:
    DtList myDfltHandleList;
};
#endif //DtHLA
```

hRefRadXmList.h

```
#if DtHLA

#include "hRefObjList.h"
#include "hRefRadXmitr.h"

// class DtReflectedRadioTransmitterList:
//
// Instances of DtReflectedRadioTransmitterList manage the list of
// of remote RadioTransmitters in the exercise.

class DtReflectedRadioTransmitterList : public DtReflectedObjectList
{
public:
    // Constructor.
    DtReflectedRadioTransmitterList(DtExerciseConn* exConn);

    // Constructor - only subscribe to the RTI::ObjectClassHandles
    // in handleList
    DtReflectedRadioTransmitterList(
        DtExerciseConn* exConn, const DtList *handleList);

    // destructor
    virtual ~DtReflectedRadioTransmitterList();

    // Look up transmitter by identifier.
    virtual DtReflectedRadioTransmitter* lookup(RTI::ObjectHandle id);

#if !DtrTI103
    // Look up transmitter by name.
    virtual DtReflectedRadioTransmitter* lookup(
        const DtGlobalObjectDesignator& objName);
#endif

    // Get the first/last transmitter in the list.
    virtual DtReflectedRadioTransmitter* first() const;
    virtual DtReflectedRadioTransmitter* last() const;

    // RadioTransmitter added callback registration/deregistration
    virtual void addRadioTransmitterAdditionCallback(
        DtRadioTransmitterCallbackFunction cb, void* userData);
    virtual void removeRadioTransmitterAdditionCallback(
        DtRadioTransmitterCallbackFunction cb, void* userData);

    // RadioTransmitter removed callback registration/deregistration
    virtual void addRadioTransmitterRemovalCallback(
        DtRadioTransmitterCallbackFunction cb, void* userData);
    virtual void removeRadioTransmitterRemovalCallback(
        DtRadioTransmitterCallbackFunction cb, void* userData);

protected:
    // copy constructor
    DtReflectedRadioTransmitterList(
        const DtReflectedRadioTransmitterList& orig);

    // assignment operator
    DtReflectedRadioTransmitterList& operator=(
        const DtReflectedRadioTransmitterList& orig);

    // Override of virtual function. Here, we create a
    // DtReflectedRadioTransmitter.
    virtual DtReflectedObject* newReflectedObject(DtHlaObject* obj) const;
}
```



```
// Called from processDiscoverObject() after a remote transmitter has
// been added to the list. Default implementation prints a message.
// Subclasses can override this function to add functionality on
// addition. obj will be an instance of a class derived from
// DtReflectedObj.
virtual void objectAdded(DtReflectedObject* obj);

// Returns a DtList of RTI::ObjectClassHandles that represent the list
// of object classes to which this list should subscribe.
virtual const DtList* dfltHandleList();

protected:
    DtList myDfltHandleList;
};
#endif //DtHLA
```

hSignalInter.h

```
#if DtHLA

#include "chooseDefines.h"
#include "hIntWEncDec.h"
#include "hostStructs.h"

// class DtSignalInteraction:
//
// Instances of DtSignalInteraction are

// Collision interaction callback function signature.
typedef void (*DtSignalInteractionCb)
    (DtSignalInteraction* inter, void* usr);

class DtSignalInteraction : public DtInteractionWithEncDec
{
public:

    // default constructor
    DtSignalInteraction();

    // destructor
    virtual ~DtSignalInteraction();

    // copy constructor
    DtSignalInteraction(const DtSignalInteraction& orig);

    // assignment operator
    DtSignalInteraction& operator=(const DtSignalInteraction& orig);

    // Print the interaction's data.
    virtual void printData() const;

    // Set/Get Host radio id
    virtual void setRadioId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& radioId() const;

    // Set/Get data rate
    virtual void setSampleRate(DtU32 rate);
    virtual DtU32 sampleRate() const;

    // Set/Get sample count
    virtual void setNumSamples(DtU16 count);
    virtual DtU16 numSamples() const;

    // Set/Get signal data length in bytes or bits.
    virtual void setNumDataBytes(DtU16 len);
    virtual DtU16 numDataBytes() const;
    virtual void setNumDataBits(DtU16 len);
    virtual DtU16 numDataBits() const;

    // Set/Get signal data
    virtual void setData(const void* data, int numBytes);
    virtual const void* data() const;

    // Set/Get the byteNum'th data byte
    virtual DtU8 dataByte(int byteNum) const;
    virtual void setDataByte(int byteNum, DtU8 data);

    // Set/Get tactical data link
    virtual void setTdlType(DtTacticalDataLinkType link);
    virtual DtTacticalDataLinkType tdlType() const;

};

#endif
```

```

// Set/Get encoding type
virtual void setEncodingType(DtEncodingType encoding);
virtual DtEncodingType encodingType() const;

// Set/Get encoding class
virtual DtEncodingClass encodingClass() const;
virtual void setEncodingClass(DtEncodingClass encodingClass);

// Set/Get TDL message count
virtual void setTdlMessageCount(DtU16 count);
virtual DtU16 tdlMessageCount() const;

// Set/Get user protocol id
virtual void setProtocolId(DtUserProtocol id);
virtual DtUserProtocol protocolId() const;

// Set/Get database index
virtual void setDatabaseIndex(DtU32 index);
virtual DtU32 databaseIndex() const;

// Set/Get duration
virtual void setDuration(DtU32 duration);
virtual DtU32 duration() const;

// Set/Get start offset
virtual void setStartOffset(DtU32 offset);
virtual DtU32 startOffset() const;

// Name of the interaction (Not nec. the ClassName from the FOM).
virtual const char* name() const;

public:

// Create a DtSignalInteraction.
// Caller is responsible for deletion.
static DtInteraction* create();

// Add (register)/Remove (deregister) function to be called when
// interaction occurs.
static void addCallback(DtExerciseConn* conn,
    DtSignalInteractionCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtSignalInteractionCb cb, void* usr);

protected:

DtGlobalObjectDesignator    myHostRadioId;
DtU16                      mySignalDataLength; // bits
DtU8*                      mySignalData;
DtTacticalDataLinkType     myTacticalDataLink;
DtU16                      myTDLMessageCount;
DtUserProtocol             myUserProtocolId;
DtU32                      myDatabaseIndex;
DtU32                      myDuration;
DtU32                      myStartOffset;
DtEncodingType             myEncodingType;
DtEncodingClass            myEncodingClass;
DtU32                      mySampleRate;
DtU16                      mySampleCount;
};
#endif //DtHLA

```

hStartDec.h

```
#if DtHLA

#include "interDecoder.h"

#include "chooseDefines.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_START_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtStartResumeInteraction, paramName)

#define DtDEFINE_SIMPLE_START_PARAM_DECODER( \
    paramName, netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER( \
        DtStartResumeDecoder, DtStartResumeInteraction, \
        paramName, netType, mutator)

#define DtDEFINE_SIMPLE_START_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
        DtStartResumeDecoder, DtStartResumeInteraction, \
        paramName, netType, mutator, castExpr)

// class DtStartResumeDecoder:
//
// Instances of DtStartResumeDecoder are used to decode StartResume
// Interactions from the network.

class DtStartResumeDecoder : public DtInteractionDecoder
{
public:

    // constructor
    DtStartResumeDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtStartResumeDecoder();

    // copy constructor
    DtStartResumeDecoder(const DtStartResumeDecoder& orig);

    // assignment operator
    DtStartResumeDecoder& operator=(const DtStartResumeDecoder& orig);

protected:

    // Individual parameter decoders

    DtDECLARE_START_PARAM_DECODER(OriginatingEntity);
    DtDECLARE_START_PARAM_DECODER(ReceivingEntity);
    DtDECLARE_START_PARAM_DECODER(RequestID);
    DtDECLARE_START_PARAM_DECODER(RealWorldTime);
    DtDECLARE_START_PARAM_DECODER(SimulationTime);
};
#endif
```

hStartEnc.h

```

#if DtHLA

#include "interEncoder.h"

#define DtDECLARE_START_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtStartResumeInteraction, paramName)

#define DtDEFINE_SIMPLE_START_PARAM_ENCODER(paramName, netType, inspector) \
    \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtStartResumeEncoder, \
        DtStartResumeInteraction, paramName, netType, inspector)

// class DtStartResumeEncoder:
//
// DtStartResumeEncoder is a DtInteractionEncoder that registers
// parameter encoders for all of the START/RESUME interaction parameters.

class DtStartResumeEncoder : public DtInteractionEncoder
{
public:

    DtStartResumeEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtStartResumeEncoder();

    // copy constructor
    DtStartResumeEncoder(const DtStartResumeEncoder& orig);

    // assignment operator
    DtStartResumeEncoder& operator=(const DtStartResumeEncoder& orig);

protected:

    // Individual parameter encoders

    DtDECLARE_START_PARAM_ENCODER(OriginatingEntity);
    DtDECLARE_START_PARAM_ENCODER(ReceivingEntity);
    DtDECLARE_START_PARAM_ENCODER(RequestID);
    DtDECLARE_START_PARAM_ENCODER(RealWorldTime);
    DtDECLARE_START_PARAM_ENCODER(SimulationTime);
};
#endif

```

hStartInter.h

```
#if DtHLA

#include "hIntWEncDec.h"
#include "hostStructs.h"
#include "mathTypes.h"

#include "chooseDefines.h"

// class DtStartResumeInteraction:
//
// Instances of DtStartResumeInteraction are used to represent
// StartResume Interactions in the RPR ROM.

// StartResume interaction callback function signature.
typedef void (*DtStartResumeInteractionCb) (DtStartResumeInteraction*
inter,
void* usr);

class DtStartResumeInteraction : public DtInteractionWithEncDec
{
public:

    // default constructor
    DtStartResumeInteraction();

    // destructor
    virtual ~DtStartResumeInteraction();

    // copy constructor
    DtStartResumeInteraction(const DtStartResumeInteraction& orig);

    // assignment operator
    DtStartResumeInteraction& operator=(const DtStartResumeInteraction&
orig);

    // Print the interaction's data.
    virtual void printData() const;

    // Set/Get sender ID
    virtual void setSenderId(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& senderId() const;

    // Set/Get receiver ID
    virtual void setReceiverId(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& receiverId() const;

    // Set/Get real world time
    virtual void setRealWorldTime(const DtTime time);
    virtual DtTime realWorldTime() const;

    // Set/Get simulation time
    virtual void setSimulationTime(const DtTime time);
    virtual DtTime simulationTime() const;

    // Set/Get request ID
    virtual void setRequestId(int id);
    virtual int requestId() const;

    // Name of the interaction (Not nec. the ClassName from the FOM).
    virtual const char* name() const;

public:
```

```
// Create a DtStartResumeInteraction.
// Caller is responsible for deletion.
static DtInteraction* create();

// Add (register)/Remove (deregister) function to be called when
// interaction occurs.
static void addCallback(DtExerciseConn* conn,
    DtStartResumeInteractionCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtStartResumeInteractionCb cb, void* usr);

protected:
    DtEntityIdentifier myOrigEnt;
    DtEntityIdentifier myReceiveEnt;
    int myRequestId;
    DtTime myRealTime;
    DtTime mySimTime;
};
#endif
```

hStateDecoder.h

```
#if DtHLA

#include "RTI.hh"
#include "vlprint.h"

#include "chooseDefines.h"

// class DtHlaStateDecoder:
//
// DtHlaStateDecoder is used to provide decoding of a state message
// into a state repository in an HLA (as opposed to DIS) environment.

class DtHlaStateDecoder
{
public:

    // constructor
    DtHlaStateDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtHlaStateDecoder();

    // copy constructor
    DtHlaStateDecoder(const DtHlaStateDecoder& orig);

    // assignment operator
    DtHlaStateDecoder& operator=(const DtHlaStateDecoder& orig);

    // Attribute decoding function declaration.
    typedef void (*DtAttributeDecoder) (
        DtStateRepository* stateRep,
        const RTI::AttributeHandleValuePairSet& avlist,
        int pairSetIndex);

    // Add/Remove a decoding function for the attribute specified
    // by handle.
    // Returns the decoding function which is replaced by the add/remove.
    // Note: This is non-virtual since it is typically called by
    // constructors.
    DtAttributeDecoder addDecoder(RTI::AttributeHandle h,
        DtAttributeDecoder func);
    DtAttributeDecoder removeDecoder(RTI::AttributeHandle h);

    // Add/Remove a decoding function for the attribute specified
    // by name.
    // Returns the decoding function which is replaced by the add/remove.
    // Note: This is non-virtual since it is typically called by
    // constructors.
    DtAttributeDecoder addDecoder(char* name,
        DtAttributeDecoder func);
    DtAttributeDecoder removeDecoder(char* name);

    // Get the attribute decoder for the specified handle.
    virtual DtAttributeDecoder decoder(int handle) const;

    // Decode the state message into the state repository.
    virtual void decode(const DtStateMsg& msg,
        DtStateRepository* stateRep) const;

    // Get the object's class handle.
    virtual RTI::ObjectClassHandle objectClassHandle() const;

};

#endif
```



```

    // Returns a new-ed clone of this object
    virtual DtHlaStateDecoder* clone() const;

public:
    // Returns a pointer to a large static buffer that can be used to
    // copy data obtained from a phvps.
    static char* decodingBuffer();

protected:
    // Decode the attribute at the specified index in the attribute
    // handle pair set.
    virtual void decodeAttribute(DtStateRepository* stateRep,
        const RTI::AttributeHandleValuePairSet& attrs,
        int pairSetIndex) const;

protected:
    DtExerciseConn* myExConn;
    DtObjClassDesc* myClassDesc;

    DtVrLRtiAmbassador* myRtiAmb;
    RTI::ObjectClassHandle myHandle;

    // myDecoderList is an array of theMaxAttributesPerObject items.
    DtAttributeDecoder* myDecoderList;

protected:
    static const unsigned int theMaxAttributesPerObject;
    static char theDecBuffer[2000];
};
// This macro is used to add attribute decoder functions
// (functions that decode individual attributes) to a DtHlaStateDecoder
#define DtADD_ATTR_DECODER(attrName) \
    addDecoder(#attrName, (DtAttributeDecoder) decode##attrName);

// This macro is used to declare attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes) to a DtHlaStateDecoder
#define DtDECLARE_ATTR_DECODER(stateRepClass, attrName) \
    static void decode##attrName(stateRepClass* stateRep, \
        const RTI::AttributeHandleValuePairSet& attrs, int pairSetIndex)

// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
    decClass, stateRepClass, attrName, netType, mutator, castExpr) \
void decClass::decode##attrName( \
    stateRepClass* stateRep, \
    const RTI::AttributeHandleValuePairSet& attrs, \
    int pairSetIndex) \
{ \
    RTI::ULong length = 0; \
    attrs.getValue(pairSetIndex, DtHlaStateDecoder::decodingBuffer(), \
length); \
    netType* netVal = (netType*) DtHlaStateDecoder::decodingBuffer(); \
    if (length > sizeof(netType)) \
    { \
        DtWarn("Size of value decoded from RTI update for attribute %s \
(%d)\n", \
        #attrName, length); \
    } \
    mutator(stateRep, netVal, castExpr); \
}

```

```
        DtWarn(" is larger than size of %s (%d)\n", #netType,
sizeof(netType)); \
    } \
    stateRep->mutator(castExpr* netVal); \
}

#define DtDEFINE_SIMPLE_ATTR_DECODER( \
    decClass, stateRepClass, attrName, netType, mutator) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST(decClass, stateRepClass, \
        attrName, netType, mutator, (netType))

#endif
```

hStateEncoder.h

```

#if DtHLA

#include "RTI.hh"
#include "MachineTypes.h"

#include "chooseDefines.h"

// class DtHlaStateEncoder:
//
// DtHlaStateEncoder is used to provide encoding of a state repository
// into a state message in an HLA (as opposed to DIS) environment.

class DtHlaStateEncoder
{
public:
    // constructor
    DtHlaStateEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtHlaStateEncoder();

    // copy constructor
    DtHlaStateEncoder(const DtHlaStateEncoder& orig);

    // assignment operator
    DtHlaStateEncoder& operator=(const DtHlaStateEncoder& orig);

    // Attribute encoding function declaration.
    typedef void (*DtAttributeEncoder)(
        const DtStateRepository& stateRep,
        RTI::AttributeHandleValuePairSet* avList,
        RTI::AttributeHandle handle);

    // Attribute checking function declaration.
    typedef DtBoolean (*DtAttributeChecker)(
        const DtStateRepository& stateRep,
        const DtStateRepository& asSeenByRemote);

    // Add/Remove an encoding function for the attribute specified
    // by handle.
    // Returns the encoding function which is replaced by the add/remove.
    // Note: This is non-virtual since it is typically called by
    // constructors.
    DtAttributeEncoder addEncoder(RTI::AttributeHandle hand,
        DtAttributeEncoder func);
    DtAttributeEncoder removeEncoder(RTI::AttributeHandle hand);

    // Add/Remove an encoding function for the attribute specified
    // by name.
    // Returns the encoding function which is replaced by the add/remove.
    // Note: This is non-virtual since it is typically called by
    // constructors.
    DtAttributeEncoder addEncoder(char* name,
        DtAttributeEncoder func);
    DtAttributeEncoder removeEncoder(char* name);

    // Add/Remove a checker function for the attribute specified
    // by handle.
    // Returns the checker function which is replaced by the add/remove.
    // Note: This is non-virtual since it is typically called by

```

```
//      constructors.
DtAttributeChecker addChecker(RTI::AttributeHandle hand,
    DtAttributeChecker func);
DtAttributeChecker removeChecker(RTI::AttributeHandle hand);

// Add/Remove a checker function for the attribute specified
// by name.
// Returns the checker function which is replaced by the add/remove.
// Note: This is non-virtual since it is typically called by
//      constructors.
DtAttributeChecker addChecker(char* name,
    DtAttributeChecker func);
DtAttributeChecker removeChecker(char* name);

// Get the attribute encoder for the specified handle.
virtual DtAttributeEncoder encoder(int handle) const;

// Get the attribute checker for the specified handle.
virtual DtAttributeChecker checker(int handle) const;

// Encode necessary values from the state repository.
// The second state repository represents what
// is seen by remote simulators and is used to determine when
// things need to be sent in the state message. includeThese
// is a set of attributes that should be included even if their
// thresholds have not been exceeded.
virtual void encode(const DtStateRepository& stateRep,
    const DtStateRepository& asSeenByRemote,
    const RTI::AttributeHandleSet& neededAttrs,
    const RTI::AttributeHandleSet& includeThese,
    DtStateMsg* msg);

// Get the object's class handle.
virtual RTI::ObjectClassHandle objectClassHandle();

// Returns a new-ed clone of this object
virtual DtHlaStateEncoder* clone() const;

public:

    // A checker function that always returns 1, meaning that the
    // attribute for which it is being used will be deemed
    // necessary to send on every tick.
    static DtBoolean trueChecker(const DtStateRepository& stateRep,
        const DtStateRepository& asSeenByRemote);

    // A checker function that always returns 0, meaning that the
    // attribute for which it is being used will never be deemed
    // necessary to send.
    static DtBoolean falseChecker(const DtStateRepository& stateRep,
        const DtStateRepository& asSeenByRemote);

protected:

    DtExerciseConn* myExConn;
    DtObjClassDesc* myClassDesc;

    DtVrIRtiAmbassador* myRtiAmb;
    RTI::InteractionClassHandle myHandle;

    // myEncoderList and myCheckerList are arrays
    // of theMaxParametersPerInteraction items.
    DtAttributeEncoder* myEncoderList;
    DtAttributeChecker* myCheckerList;

protected:
```

```

    static const int theMaxAttributesPerObject;
};
// This macro is used to add attribute encoder functions
// (functions that encode individual attributes) to a DtHlaStateEncoder
#define DtADD_ATTR_ENCODER(attrName) \
    addEncoder(#attrName, (DtAttributeEncoder) encode##attrName)

// This macro is used to add attribute checker functions
// (functions that determine whether particular attributes need to
// be included in an outgoing state upDtHlp
#define DtADD_ATTR_CHECKER(attrName) \
    adChecker(#attrName, (DtAttributeNC#a trName)

// This macro is used to declare attribute encoder fu
// ambers of aodeivmd (DtHlaStateEnc)13.4(odrs that encode)13.4( )TJ1~T*1~[ (//iu )13.4ndi
static void attrName( const)13.4 statRdpClass& rmp,
    ttatts,
    ttattHandlee)

// This is used to declare static member functions
#define DtDECLARED_ATTR(attrName)

    static DtBoole annemd##(attrName( cot)13.4 s statRdpClass& statRdp,
    ttss& asSeenByReam

// This macro is used to define simple attribute functions
// ambers of aodeivmd (DtHlaStateEnc)13.4(odrs that encode)13.4( )TJ1~T*1~[ (//iu )13.4ndi
// ciovernt te value rmtun(d byt te fi)13.4(spjecor to anem)13.4tType,
// au to te a
#define DtDEFINE_SIMPLE_ATTRIBUTE(attrName, intType, ispjecore) \
    attrName( const statRdpClass& -)-134(rm)13.4(p,)- \
    RTmValuePairSde*n atts,
    RTm (attHandlee) \)TJ1~T*1~({) \ nem
    nemter\
    #a ts-> adR(attHandhe, char*e)&ndetntTt

// ambers of aodeivmd (DtHlaStateEnc)13.4(odrs that checd )13.4 whethe
// ad articular attributo isout ofod ae. / Tnesimpldd
// isto jusat omplaerate values8 rturind byt te two SatatRdps •
// iu
#define DtDEFINE_SIMPLE_ATTRIBUTE(attrName, ispjecore) \
    an enClass::nemd##(attrName( cot)13.4 s statRdpClass& statRdp,
    cio s statRdpClass&

    remDBoole ae)m
    spjecore());) \

```

hStateMsg.h

```
#if DtHLA

#include "RTI.hh"

#include "chooseDefines.h"
#include "vlTime.h"
#include "tStampType.h"
#include "str.h"

// class DtStateMsg:
//
// Instances of DtStateMsg are used to represent state data
// communicated via the network in an HLA (as opposed to DIS) environment.

#if DtRTI103
#define ObjectHandle ObjectID
#endif

class DtStateMsg
{
public:
    // constructor
    DtStateMsg(RTI::ObjectHandle id,
               const RTI::AttributeHandleValuePairSet& attrs);

    // constructor
    DtStateMsg(RTI::ObjectHandle id,
               RTI::AttributeHandleValuePairSet* attrs);

    // destructor
    virtual ~DtStateMsg();

    // copy constructor
    DtStateMsg(const DtStateMsg& orig);

    // assignment operator
    DtStateMsg& operator=(const DtStateMsg& orig);

    // Get the attribute handle value pair set. (Const version)
    virtual const RTI::AttributeHandleValuePairSet& ahvps() const;

    // Get the attribute handle value pair set. (Non-const version)
    virtual RTI::AttributeHandleValuePairSet* ahvps();

    // Get the object identifier.
    virtual RTI::ObjectHandle objectId() const;

    // Prints prints header and attributes with values
    virtual void print(DtExerciseConn* exConn,
                      RTI::ObjectClassHandle h) const;

    // Uses the object to print host representation of data
    virtual void print(DtHlaObject *obj) const;

    // prints the header information
    virtual void printHeader(DtExerciseConn* exConn,
                             RTI::ObjectClassHandle classHandle) const;

    // Print the attributes (with hex values if withHex is 1)
    virtual void DtStateMsg::printAttrs(DtExerciseConn* exConn,
                                         RTI::ObjectClassHandle classHandle, int withHex) const;
```

```

// Set/Get the user-defined tag.
virtual void setTag(const DtString& tag);
virtual const DtString& tag() const;

// Set the time of validity of the message, and whether this
// represents relative or absolute time. Time is in seconds, but only
// time modulo an hour is stored.
virtual void setTimeStamp(DtTime time,
    DtTimeStampType type = DtTimeStampRelative);

// Return the time of validity of the message (in seconds past an hour)
// as set by setTimeStamp.
virtual DtTime timeStamp() const;

// Return the type of the time stamp.
virtual DtTimeStampType timeStampType() const;

// Since timestamp only represents number of seconds past the hour,
// guessTimeValid chooses the hour that would make the timestamp
// closest to reference time, and returns a time that is timestamp
// seconds past that hour.
virtual DtTime guessTimeValid(DtTime referenceTime) const;

public:

    // Set/Get flag that tells us whether we're using the RPR FOM
    // convention that the UserDefinedTag in updates and interactions
    // represents a DIS-style timestamp.
    static void setTagIsAsciiTime(DtBoolean val);
    static DtBoolean tagIsAsciiTime();

protected:

    DtString myTag;
    RTI::AttributeHandleValuePairSet* myAvList;
    RTI::ObjectHandle myId;
    DtTime myTime;
    DtTimeStampType myTimeType;

protected:

    static DtBoolean theTagIsAsciiTime;
};
#endif

```

hStopDec.h

```
#if DtHLA

#include "interDecoder.h"

#include "chooseDefines.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_STOP_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtStopFreezeInteraction, paramName)

#define DtDEFINE_SIMPLE_STOP_PARAM_DECODER( \
    paramName, netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER( \
        DtStopFreezeDecoder, DtStopFreezeInteraction, \
        paramName, netType, mutator)

#define DtDEFINE_SIMPLE_STOP_PARAM_DECODER_WITH_CAST( \
    paramName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST( \
        DtStopFreezeDecoder, DtStopFreezeInteraction, \
        paramName, netType, mutator, castExpr)

// class DtStopFreezeDecoder:
//
// Instances of DtStopFreezeDecoder are used to decode StopFreeze
// Interactions from the network.

class DtStopFreezeDecoder : public DtInteractionDecoder
{
public:

    // constructor
    DtStopFreezeDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtStopFreezeDecoder();

    // copy constructor
    DtStopFreezeDecoder(const DtStopFreezeDecoder& orig);

    // assignment operator
    DtStopFreezeDecoder& operator=(const DtStopFreezeDecoder& orig);

protected:

    // Individual parameter decoders

    DtDECLARE_STOP_PARAM_DECODER(OriginatingEntity);
    DtDECLARE_STOP_PARAM_DECODER(ReceivingEntity);
    DtDECLARE_STOP_PARAM_DECODER(RequestID);
    DtDECLARE_STOP_PARAM_DECODER(RealWorldTime);
    DtDECLARE_STOP_PARAM_DECODER(Reason);
    DtDECLARE_STOP_PARAM_DECODER(ReflectValues);
    DtDECLARE_STOP_PARAM_DECODER(RunInternalSimulationClock);
    DtDECLARE_STOP_PARAM_DECODER(UpdateAttributes);
};
#endif
```


hStopEnc.h

```

#if DtHLA

#include "interEncoder.h"

#define DtDECLARE_STOP_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtStopFreezeInteraction, paramName)

#define DtDEFINE_SIMPLE_STOP_PARAM_ENCODER(paramName, netType, inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtStopFreezeEncoder, \
        DtStopFreezeInteraction, paramName, netType, inspector)

// class DtStopFreezeEncoder:
//
// DtStopFreezeEncoder is a DtInteractionEncoder that registers
// parameter encoders for all of the STOP/FREEZE interaction parameters.

class DtStopFreezeEncoder : public DtInteractionEncoder
{
public:
    DtStopFreezeEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtStopFreezeEncoder();

    // copy constructor
    DtStopFreezeEncoder(const DtStopFreezeEncoder& orig);

    // assignment operator
    DtStopFreezeEncoder& operator=(const DtStopFreezeEncoder& orig);

protected:
    // Individual parameter encoders

    DtDECLARE_STOP_PARAM_ENCODER(OriginatingEntity);
    DtDECLARE_STOP_PARAM_ENCODER(ReceivingEntity);
    DtDECLARE_STOP_PARAM_ENCODER(RequestID);
    DtDECLARE_STOP_PARAM_ENCODER(RealWorldTime);
    DtDECLARE_STOP_PARAM_ENCODER(Reason);
    DtDECLARE_STOP_PARAM_ENCODER(ReflectValues);
    DtDECLARE_STOP_PARAM_ENCODER(RunInternalSimulationClock);
    DtDECLARE_STOP_PARAM_ENCODER(UpdateAttributes);
};
#endif

```

hStopInter.h

```
#if DtHLA

#include "hIntWEncDec.h"
#include "hostStructs.h"
#include "mathTypes.h"

#include "chooseDefines.h"

// class DtStopFreezeInteraction:
//
// Instances of DtStopFreezeInteraction are used to represent
// StopFreeze Interactions in the RPR ROM.

// StopFreeze interaction callback function signature.
typedef void (*DtStopFreezeInteractionCb) (DtStopFreezeInteraction* inter,
void* usr);

class DtStopFreezeInteraction : public DtInteractionWithEncDec
{
public:

    // default constructor
    DtStopFreezeInteraction();

    // destructor
    virtual ~DtStopFreezeInteraction();

    // copy constructor
    DtStopFreezeInteraction(const DtStopFreezeInteraction& orig);

    // assignment operator
    DtStopFreezeInteraction& operator=(const DtStopFreezeInteraction& orig);

    // Print the interaction's data.
    virtual void printData() const;

    // Set/Get sender ID
    virtual void setSenderId(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& senderId() const;

    // Set/Get receiver ID
    virtual void setReceiverId(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& receiverId() const;

    // Set/Get real world time
    virtual void setRealWorldTime(const DtTime time);
    virtual DtTime realWorldTime() const;

    // Set/Get request ID
    virtual void setRequestId(int id);
    virtual int requestId() const;

    // Set/Get stopReason
    virtual DtStopReason reason() const;
    virtual void setReason(DtStopReason reason);

    // Set/Get reflectWhenFrozen flag
    virtual DtBoolean reflectWhenFrozen() const;
    virtual void setReflectWhenFrozen(DtBoolean flag);

    // Set/Get runClockWhenFrozen flag
    virtual DtBoolean runClockWhenFrozen() const;
    virtual void setRunClockWhenFrozen(DtBoolean flag);

};

#endif
```

```
// SetGet updateWhenFrozen flag
virtual DtBoolean updateWhenFrozen() const;
virtual void setUpdateWhenFrozen(DtBoolean flag);

// Name of the interaction (Not nec. the ClassName from the FOM).
virtual const char* name() const;

public:

    // Create a DtStopFreezeInteraction.
    // Caller is responsible for deletion.
    static DtInteraction* create();

    // Add (register)/Remove (deregister) function to be called when
    // interaction occurs.
    static void addCallback(DtExerciseConn* conn,
        DtStopFreezeInteractionCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtStopFreezeInteractionCb cb, void* usr);

protected:

    DtEntityIdentifier myOrigEnt;
    DtEntityIdentifier myReceiveEnt;
    int myRequestId;
    DtTime myRealTime;
    DtStopReason myReason;
    DtBoolean myReflect;
    DtBoolean myUpdate;
    DtBoolean myRunClock;
};
#endif
```

hUnknownInter.h

```
#if DtHLA

#include "hInteraction.h"
#include "hostStructs.h"
#include "exerciseConn.h"

#include "chooseDefines.h"

// class DtUnknownInteraction:
//
// Instances of DtUnknownInteraction are used to represent
// Interactions for which there is no derived DtInteraction.

// Unknown interaction callback function signature.
typedef void (*DtUnknownInteractionCb)
    (DtUnknownInteraction* inter, void* usr);

class DtUnknownInteraction : public DtInteraction
{
public:

    // default constructor
    DtUnknownInteraction();

    // constructor
    DtUnknownInteraction(RTI::InteractionClassHandle handle,
        const RTI::ParameterHandleValuePairSet& pvList);

    // destructor
    virtual ~DtUnknownInteraction();

    // copy constructor
    DtUnknownInteraction(const DtUnknownInteraction& orig);

    // assignment operator
    DtUnknownInteraction& operator=(const DtUnknownInteraction& orig);

    // Get the parameter handle value pair set for the interaction.
    virtual const RTI::ParameterHandleValuePairSet& phvps() const;

    // Decode into this object from parameter value list.
    virtual void setFromPhvps(
        const RTI::ParameterHandleValuePairSet& pvList);

    // Sets the interaction's ExerciseConn and class descriptor. If
    // classDesc is NULL, we use the classDesc associated with the
    // handle on which the interaction was created.
    // Without being told what ExConn or class descriptor to use,
    // many of the HLA-specific functions won't work properly.
    virtual void setExConn(DtExerciseConn* exConn, DtInterClassDesc* class-
Desc);

    // Just prints the parameters
    virtual void printData() const;

    // Name of the interaction
    virtual const char* name() const;

public:

    // Create a DtUnknownInteraction.
    // Caller is responsible for deletion.
    static DtInteraction* create();


```

```
// Add (register)/Remove (deregister) function to be called when
// interaction with specified class handle occurs.
static void addCallback(RTI::InteractionClassHandle handle,
    DtExerciseConn* conn,
    DtUnknownInteractionCb cb, void* usr);
static void removeCallback(RTI::InteractionClassHandle handle,
    DtExerciseConn* conn,
    DtUnknownInteractionCb cb, void* usr);

protected:

    // Decode into this object from parameter value list.
    // Non-virtual since meant to be called from ctor.
    void internalSetFromPhvps(
        const RTI::ParameterHandleValuePairSet& pvList);

protected:

    RTI::ParameterHandleValuePairSet* myParams;
    RTI::InteractionClassHandle myHandle;
};
#endif
```

hVcDec.h

```
#if DtHLA

#include "interDecoder.h"

// This can be used to declare parameter processor functions
// which are static members of a class.

#define DtDECLARE_VIEW_CONTROL_PARAM_DECODER(paramName) \
    DtDECLARE_PARAM_DECODER(DtViewControlInteraction, paramName)

// class DtViewControlDecoder:
//
// Instances of DtViewControlDecoder are used to decode ViewControl Interac-
// tions
// from the network.

class DtViewControlDecoder : public DtInteractionDecoder
{
public:
    // constructor
    DtViewControlDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtViewControlDecoder();

    // copy constructor
    DtViewControlDecoder(const DtViewControlDecoder& orig);

    // assignment operator
    DtViewControlDecoder& operator=(const DtViewControlDecoder& orig);

protected:
    // Individual parameter decoders

    DtDECLARE_VIEW_CONTROL_PARAM_DECODER(ViewControlPdu);
};
#endif
```

hVcEnc.h

```

#if DtHLA

#include "interEncoder.h"
#include "chooseDefines.h"

#define DtDECLARE_VIEW_CONTROL_PARAM_ENCODER(paramName) \
    DtDECLARE_PARAM_ENCODER(DtViewControlInteraction, paramName)

#define DtDEFINE_SIMPLE_VIEW_CONTROL_PARAM_ENCODER(paramName, netType, \
    inspector) \
    DtDEFINE_SIMPLE_PARAM_ENCODER(DtViewControlEncoder, DtViewControlInter- \
    action, \
                                paramName, netType, \
                                inspector)

// class DtViewControlEncoder:
//
// DtViewControlEncoder is a DtInteractionEncoder that registers
// parameter encoders for all of the VIEW_CONTROL interaction parameters.
class DtViewControlEncoder : public DtInteractionEncoder
{
public:
    DtViewControlEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtViewControlEncoder();

    // copy constructor
    DtViewControlEncoder(const DtViewControlEncoder& orig);

    // assignment operator
    DtViewControlEncoder& operator=(const DtViewControlEncoder& orig);

protected:
    // Individual parameter encoders
    DtDECLARE_VIEW_CONTROL_PARAM_ENCODER(ViewControlPdu);
};
#endif

```

hVcInter.h

```
#if DtHLA

#include "hIntWEncDec.h"
#include "vcPdu.h"

#include "chooseDefines.h"

// class DtViewControlInteraction:
//
// Instances of DtViewControlInteraction are used to represent
// ViewControl Interactions in the RPR FOM.

// ViewControl interaction callback function signature.
typedef void (*DtViewControlInteractionCb)(DtViewControlInteraction* inter,
void* usr);

class DtViewControlInteraction : public DtInteractionWithEncDec
{
public:

    // default constructor
    DtViewControlInteraction();

    // destructor
    virtual ~DtViewControlInteraction();

    // copy constructor
    DtViewControlInteraction(const DtViewControlInteraction& orig);

    // assignment operator
    DtViewControlInteraction& operator=(const DtViewControlInteraction&
orig);

    // Print the interaction's data.
    virtual void printData() const;

    // Get/Set a view control PDU
    virtual const DtViewControlPdu* vcPdu() const;
    virtual void setVcPdu(const DtViewControlPdu& pdu);
    virtual void setFromNetPacket(const DtNetPacket* pack);

    // Name of the interaction
    virtual const char* name() const;

public:

    // Create a DtViewControlInteraction.
    // Caller is responsible for deletion.
    static DtInteraction* create();

    // Add (register)/Remove (deregister) function to be called when
    // interaction occurs.
    static void addCallback(DtExerciseConn* conn,
DtViewControlInteractionCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
DtViewControlInteractionCb cb, void* usr);

protected:

    DtViewControlPdu* myVcPdu;
};
#endif
```


iffAtcNavBeam.h

```

#if !DtHLA
#if DtDISVERS >= 5

#ifndef DtIffAtcNav aidsBeamData_H_
#define DtIffAtcNav aidsBeamData_H_

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/iffAtcNavP.h"

// class DtIffAtcNav aidsBeamData:
//
// DtIffAtcNav aidsBeamData is a class which contains the antenna data
// for the IFF/ATC/NAVAIDS PDU.

class DtIffAtcNav aidsBeamData
{
public:

    // Constructors
    DtIffAtcNav aidsBeamData();
    DtIffAtcNav aidsBeamData(const DtIffAtcNav aidsBeamData& orig);

    // Assignment operator
    DtIffAtcNav aidsBeamData& operator=(
        const DtIffAtcNav aidsBeamData& orig);
    DtIffAtcNav aidsBeamData& operator=(
        const DtNetIffAtcNav aidsBeamData& orig);

    // Destructor
    virtual ~DtIffAtcNav aidsBeamData();

    // Set/Get the Azimuth Center
    virtual void setAzimuthCenter(DtFloat32 value);
    virtual DtFloat32 azimuthCenter() const;

    // Set/Get the Azimuth Sweep
    virtual void setAzimuthSweep(DtFloat32 value);
    virtual DtFloat32 azimuthSweep() const;

    // Set/Get the Elevation Center
    virtual void setElevationCenter(DtFloat32 value);
    virtual DtFloat32 elevationCenter() const;

    // Set/Get the Elevation Sweep
    virtual void setElevationSweep(DtFloat32 value);
    virtual DtFloat32 elevationSweep() const;

    // Set/Get the Sweep Sync
    virtual void setSweepSync(DtFloat32 value);
    virtual DtFloat32 sweepSync() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(const DtNetIffAtcNav aidsBeamData& record);
    virtual void loadNetRecord(DtNetIffAtcNav aidsBeamData& record) const;

protected:

```

```
// Data members:
DtFloat32      myAzimuthCenter;
DtFloat32      myAzimuthSweep;
DtFloat32      myElevationCenter;
DtFloat32      myElevationSweep;
DtFloat32      mySweepSync;

};
#endif
#endif
```

iffAtcNavLayr.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/iffAtcNavP.h"

// class DtIffAtcNavAidsLayerHeader:
//
// DtIffAtcNavAidsLayerHeader is a class which contains the layer header
// information for a layer of the IFF/ATC/NAVAIDS PDU. These records are
// maintained automatically by the PDU class as layers are created and
// deleted from the PDU.

class DtIffAtcNavAidsLayerHeader
{
public:
    // Constructors
    DtIffAtcNavAidsLayerHeader();
    DtIffAtcNavAidsLayerHeader(const DtIffAtcNavAidsLayerHeader& orig);

    // Assignment operator
    DtIffAtcNavAidsLayerHeader& operator=(
        const DtIffAtcNavAidsLayerHeader& orig);
    DtIffAtcNavAidsLayerHeader& operator=(
        const DtNetIffAtcNavAidsLayerHeader& orig);

    // Destructor
    virtual ~DtIffAtcNavAidsLayerHeader();

    // Set/Get the layer number
    virtual void setLayerNumber(DtU8 num);
    virtual DtU8 layerNumber() const;

    // Set/Get the layer specific information
    virtual void setLayerInfo(DtIffAtcNavAidsLayerSpecificInfo info);
    virtual DtIffAtcNavAidsLayerSpecificInfo layerInfo() const;

    // Set/Get the layer length
    virtual void setLayerLength(DtU16 length);
    virtual DtU16 layerLength() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(const DtNetIffAtcNavAidsLayerHeader&
record);
    virtual void loadNetRecord(DtNetIffAtcNavAidsLayerHeader& record) const;

protected:
    // Data members:
    DtU8 myLayerNumber;
    DtU8 myLayerInfo;
    DtU16 myLayerLength;
};
#endif
#endif

```

iffAtcNavOper.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/iffAtcNavP.h"

// class DtIffAtcNavAidsOperational:
//
// DtIffAtcNavAidsOperational is a class which contains the fundamental
// operational data for the IFF/ATC/NAVAIDS PDU.

class DtIffAtcNavAidsOperational
{
public:
    // Constructors
    DtIffAtcNavAidsOperational();
    DtIffAtcNavAidsOperational(const DtIffAtcNavAidsOperational& orig);

    // Assignment operator
    DtIffAtcNavAidsOperational& operator=(
        const DtIffAtcNavAidsOperational& orig);
    DtIffAtcNavAidsOperational& operator=(
        const DtNetIffAtcNavAidsOperational& orig);

    // Destructor
    virtual ~DtIffAtcNavAidsOperational();

    // Set/Get the system status
    virtual void setSystemStatus(DtU8 status);
    virtual DtU8 systemStatus() const;

    // Set/Get the system status by bit
    virtual void setSystemStatusBit(DtIffAtcNavAidsStatusBit bit,
        DtBoolean value);
    virtual DtBoolean systemStatusBit(
        DtIffAtcNavAidsStatusBit bit) const;

    // Set/Get alternate parameter 4
    virtual void setAltParameter4(DtIffAtcNavAidsAltParm4 mod);
    virtual DtIffAtcNavAidsAltParm4 altParameter4() const;

    // Set/Get the information layers field (each layer has a corresponding
    // bit in this field which indicates the presence of that layer)
    virtual void setInformationLayers(DtU8 layers);
    virtual DtU8 informationLayers() const;

    // Set/Query an information layer field for a particular layer
    virtual void setLayer(DtU8 layer, DtBoolean value);
    virtual DtBoolean layerPresent(int layer) const;

    // Set/Get modifier
    virtual void setModifier(DtIffAtcNavAidsModifier mod);
    virtual DtIffAtcNavAidsModifier modifier() const;

    // The following functions allow you to set/get the fundamental opera-
    tional
    // parameters. These parameters may be stored in different formats
    // depending on the system type field. These functions do not check
    // to make sure that the appropriate format is being used.
```

```
// Set/Get a parameter (unsigned 16-bit values)
virtual void setParameter(int index, DtU16 value);
virtual DtU16 parameter(int index) const;

// Set/Get parameter code elements. These are stored in 3-bit integers
// which occupy bits 0-14 of the parameter.
virtual void setParameterCode(int index, int code, int value);
virtual int parameterCode(int index, int code) const;

// Set/Get parameter code bit
virtual void setParameterBit(int index,
    DtIffAtcNavAidsParameterBit bit, DtBoolean value);
virtual DtBoolean parameterBit(int index,
    DtIffAtcNavAidsParameterBit bit) const;

// Print the information contained in the record
virtual void print() const;

// Load/Unload data from a packet record
virtual void unloadNetRecord(const DtNetIffAtcNavAidsOperational&
record);
virtual void loadNetRecord(DtNetIffAtcNavAidsOperational& record) const;

protected:

    // Data members:
    DtU8          myStatus;
    DtU8          myAltParameter4;
    DtU8          myLayers;
    DtU8          myModifier;
    DtU16         myParameter[6];
};
#endif
#endif
```

iffAtcNavOpr2.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/iffAtcNavP.h"

// class DtIffAtcNavAidsSecondOperational:
//
// DtIffAtcNavAidsSecondOperational is a class which contains the secondary
// operational data for a layer of the IFF/ATC/NAVAIDS PDU.

class DtIffAtcNavAidsSecondOperational
{
public:
    // Constructors
    DtIffAtcNavAidsSecondOperational();
    DtIffAtcNavAidsSecondOperational(
        const DtIffAtcNavAidsSecondOperational& orig);

    // Assignment operators
    DtIffAtcNavAidsSecondOperational& operator=(
        const DtIffAtcNavAidsSecondOperational& orig);
    DtIffAtcNavAidsSecondOperational& operator=(
        const DtNetIffAtcNavAidsSecondOperational& orig);

    // Destructor
    virtual ~DtIffAtcNavAidsSecondOperational();

    // Set/Get parameter 1
    virtual void setParameter1(DtIffAtcNavAidsParm1 value);
    virtual DtIffAtcNavAidsParm1 parameter1() const;

    // Set/Get parameter 2
    virtual void setParameter2(DtIffAtcNavAidsParm2 value);
    virtual DtIffAtcNavAidsParm2 parameter2() const;

    // Get the number of parameter sets
    virtual void setNumParameterSets(DtU16);
    virtual DtU16 numParameterSets() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetIffAtcNavAidsSecondOperational& record);
    virtual void loadNetRecord(
        DtNetIffAtcNavAidsSecondOperational& record) const;

protected:
    // Data members:
    DtU8 myParameter1;
    DtU8 myParameter2;
    DtU16 myNumParameterSets;
};
#endif
#endif
```

iffAtcNavParm.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/iffAtcNavP.h"

// class DtIffAtcNavAidsParameter:
//
// DtIffAtcNavAidsParameter is a class which contains the parameter data
// for the IFF/ATC/NAVAIDS PDU.

class DtIffAtcNavAidsParameter
{
public:
    // Constructors
    DtIffAtcNavAidsParameter();
    DtIffAtcNavAidsParameter(const DtIffAtcNavAidsParameter& orig);

    // Assignment operator
    DtIffAtcNavAidsParameter& operator=(
        const DtIffAtcNavAidsParameter& orig);
    DtIffAtcNavAidsParameter& operator=(
        const DtNetIffAtcNavAidsParameter& orig);

    // Destructor
    virtual ~DtIffAtcNavAidsParameter();

    // Set/Get the ERP
    virtual void setERP(DtFloat32 value);
    virtual DtFloat32 ERP() const;

    // Set/Get the frequency
    virtual void setFrequency(DtFloat32 value);
    virtual DtFloat32 frequency() const;

    // Set/Get the PgRF
    virtual void setPgRF(DtFloat32 value);
    virtual DtFloat32 PgRF() const;

    // Set/Get the pulse width
    virtual void setPulseWidth(DtFloat32 value);
    virtual DtFloat32 pulseWidth() const;

    // Set/Get the burst length
    virtual void setBurstLength(DtFloat32 value);
    virtual DtFloat32 burstLength() const;

    // Set/Get the applicable modes
    virtual void setApplicableModes(DtIffAtcNavAidsApplicableModes value);
    virtual DtIffAtcNavAidsApplicableModes applicableModes() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(const DtNetIffAtcNavAidsParameter& record);
    virtual void loadNetRecord(DtNetIffAtcNavAidsParameter& record) const;

protected:

```

```
// Data members:
DtFloat32      myERP;
DtFloat32      myFrequency;
DtFloat32      myPgRF;
DtFloat32      myPulsewidth;
DtFloat32      myBurstLength;
DtU8           myApplicableModes;

};
#endif
#endif
```


iffAtcNavPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"
#include "mathTypes.h"

#include "packets/iffAtcNavP.h"

#include "iffAtcNavBeam.h"
#include "iffAtcNavLayr.h"
#include "iffAtcNavOper.h"
#include "iffAtcNavOpr2.h"
#include "iffAtcNavParm.h"
#include "iffAtcNavSyID.h"

// class DtIffAtcNavAidsPdu:
//
// Instances of DtIffAtcNavAidsPdu are used to send information about simu-
// lated
// radio transponders and interrogators.

// DtIffAtcNavAidsPdu callback function signature.
typedef void (*DtIffAtcNavAidsPduCb)(DtIffAtcNavAidsPdu* pdu, void* usr);

class DtIffAtcNavAidsPdu : public DtPdu
{
public:
    // Constructors
    DtIffAtcNavAidsPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIffAtcNavAidsPdu(const DtNetIffAtcNavAidsPduLayer1* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIffAtcNavAidsPdu(const DtIffAtcNavAidsPdu& orig);

    // destructor
    virtual ~DtIffAtcNavAidsPdu();

    // Create a DtIffAtcNavAidsPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtIffAtcNavAidsPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtIffAtcNavAidsPduCb cb, void* usr);

    // Assignment operator
    DtIffAtcNavAidsPdu& operator=(const DtIffAtcNavAidsPdu& orig);

    // Set/Get emitting entity identifier
    virtual void setEmittingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& emittingEntityID() const;

    // Set/Get event identifier
    virtual void setEventID(const DtEventID& event);
    virtual const DtEventID& eventID() const;

```

```
// Set/Get location
virtual void setLocation(DtConstVector coords);
virtual DtConstVector location() const;

// Set/Get system ID
virtual void setSystemID(const DtIffAtcNavAidsSystemID& id);
virtual const DtIffAtcNavAidsSystemID& systemID() const;

// Set/Get fundamental operational data
virtual void setOperational(const DtIffAtcNavAidsOperational& oper);
virtual const DtIffAtcNavAidsOperational& operational() const;

// Query the layers field of the fundamental operational data record.
virtual DtBoolean layerPresent(int layer) const;

// Enable/Disable layer 2. This method is also used to select the
// number of fundamental parameter data records for the second layer.
virtual DtBoolean enableLayer2(DtU8 layerSpecificInfo, int numRecords);
virtual DtBoolean disableLayer2();

// Note: the following accessor functions are for records in layer 2.
// If layer 2 is not enabled, the sets will do nothing and the gets will
// return undefined results.

// Get layer header for layer 2
// This record is set automatically by the PDU class
virtual const DtIffAtcNavAidsLayerHeader& layer2Header() const;

// Set/Get beam data
virtual void setBeamData(const DtIffAtcNavAidsBeamData& beamData);
virtual const DtIffAtcNavAidsBeamData& beamData() const;

// Set/Get secondary operational parameters
virtual void setSecondaryOperational(
    const DtIffAtcNavAidsSecondOperational& oper);
virtual const DtIffAtcNavAidsSecondOperational&
    secondaryOperational() const;

// Get the number of fundamental parameter data records from the
secondary
// operational parameters record
virtual DtU16 numParameterSets() const;

// Set/Get the fundamental parameter data records for the second layer.
virtual DtBoolean setParmData(int parmIndex,
    const DtIffAtcNavAidsParameter& init);
virtual DtBoolean getParmData(int parmIndex,
    DtIffAtcNavAidsParameter& record) const;

// Print the data contained by this PDU
virtual void printData() const;

protected:

// Returns the internal PDU kind
virtual DtPduKind internalGetPduKind() const;

// Casts the netPdu element to our network struct
virtual DtNetIffAtcNavAidsPduLayer1* netPduStruct() const;

// Casts the netPdu element to our network struct
virtual DtNetIffAtcNavAidsPduLayer2* netPduStructL2() const;

// Casts the 'this' pointer for changing elements in a const function
// Non virtual since you can't override based on return type
DtIffAtcNavAidsPdu* self() const;
```

```
// Returns a pointer to the first parameter record (after static data).
virtual DtNetIffAtcNavAidsParameter* parameter() const;

protected:

    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier          myEntityID;
    DtEventID                  myEventID;
    DtVector                   myEntCoords;
    DtIffAtcNavAidsSystemID    mySystemID;
    DtIffAtcNavAidsOperational myOperational;
    DtIffAtcNavAidsLayerHeader myLayerHeader;
    DtIffAtcNavAidsBeamData    myBeamData;
    DtIffAtcNavAidsSecondOperational mySecondOperational;
    DtIffAtcNavAidsParameter   myParameter;

};
#endif
#endif
```

iffAtcNavSysID.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/iffAtcNavP.h"

// class DtIffAtcNavAidsSystemID:
//
// DtIffAtcNavAidsSystemID is a class which contains information identifying
// the emitting system in the IFF/ATC/NAVAIDS PDU.

class DtIffAtcNavAidsSystemID
{
public:

    // Constructors
    DtIffAtcNavAidsSystemID();
    DtIffAtcNavAidsSystemID(const DtIffAtcNavAidsSystemID& orig);

    // Assignment operator
    DtIffAtcNavAidsSystemID& operator=(const DtIffAtcNavAidsSystemID& orig);
    DtIffAtcNavAidsSystemID& operator=(const DtNetIffAtcNavAidsSystemID&
orig);

    // Destructor
    virtual ~DtIffAtcNavAidsSystemID();

    // Set/Get the system type
    virtual void setSystemType(DtIffAtcNavAidsSysType type);
    virtual DtIffAtcNavAidsSysType systemType() const;

    // Set/Get the system name
    virtual void setSystemName(DtIffAtcNavAidsSysName name);
    virtual DtIffAtcNavAidsSysName systemName() const;

    // Set/Get the system mode
    virtual void setSystemMode(DtIffAtcNavAidsSysMode mode);
    virtual DtIffAtcNavAidsSysMode systemMode() const;

    // Set/Get the system change/options
    virtual void setChangeOptions(DtU8 opts);
    virtual DtU8 changeOptions() const;

    // Set/Get the system change/option by bit value
    virtual void setChangeOptionsBit(DtIffAtcNavAidsChangeOptionsBit bit,
DtBoolean value);
    virtual DtBoolean changeOptionsBit(
DtIffAtcNavAidsChangeOptionsBit bit) const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(const DtNetIffAtcNavAidsSystemID& record);
    virtual void loadNetRecord(DtNetIffAtcNavAidsSystemID& record) const;

protected:

    // Data members:
```

```
DtIfAtcNavAidsSysType    myType;  
DtIfAtcNavAidsSysName    myName;  
DtIfAtcNavAidsSysMode    myMode;  
DtU8                     myChangeOptions;  
  
};  
#endif  
#endif
```

intClassDesc.h

```
#if DtHLA

#include "RTI.hh"
#include "MachineTypes.h"

#include "chooseDefines.h"

// class DtInterClassDesc:
//
// Instances of DtInterClassDesc are used to represent
// HLA Object and Interaction classes

class DtInterClassDesc
{
public:
    // constructor
    DtInterClassDesc(const char* className,
        DtInterClassDesc* base, RTI::InteractionClassHandle handle,
        DtExerciseConn* exConn);

    // destructor
    virtual ~DtInterClassDesc();

    // copy constructor
    DtInterClassDesc(const DtInterClassDesc& orig);

    // assignment operator
    DtInterClassDesc& operator=(const DtInterClassDesc& orig);

    // Add a parameter (Used when reading FED file).
    virtual void addParam(const char* paramName);

    // Get the object's class handle.
    virtual RTI::InteractionClassHandle handle() const;

    // Get the object's class name
    virtual char* name() const;

    // Return a pointer to the base class's descriptor, NULL if none
    virtual const DtInterClassDesc* baseClass() const;

    // Returns 1 if the class is of class hand or a subclass
    // of hand, 0 otherwise
    virtual int isOfClass(RTI::InteractionClassHandle hand) const;

    // Get the number of parameters associated with this object.
    virtual int numParameters() const;

    // Get the set of valid parameter handles for this class. Includes
    // parameters inherited from base classes.
    virtual const RTI::AttributeHandleSet& paramHandleSet() const;

    // Get the set of non-inherited parameter handles for this class.
    virtual const RTI::AttributeHandleSet& ownParamHandles() const;

    // Ensures that the class is published, and that start and stopUpdates
    // callbacks are registered.
    virtual void setupClassForSending();

    // Subscribe to the class
    virtual void registerInterest();
};

#endif
```

```

    // Unsubscribe to the class
    virtual void deregisterInterest();

    // Get my RTI ambassador
    virtual DtVrlRtiAmbassador* rtiAmb();

    // Returns DtTrue if the interaction class is needed by the federation
    // false otherwise.
    virtual DtBoolean neededByFederation() const;

public:
    // Set/Get maximum number of allowable parameters for an interaction
    static void setMaxParameters(int num);
    static int maxParameters();

protected:
    // Callback for handling startInters RTI callback
    static void startIntersCb(RTI::InteractionClassHandle hand,
        void* usr);

    // Callback for handling stopInters RTI callback
    static void stopIntersCb(RTI::InteractionClassHandle hand,
        void* usr);

    // Callback for handling startInters RTI callback
    virtual void processStartInters(RTI::InteractionClassHandle hand);

    // Callback for handling stopInters RTI callback
    virtual void processStopInters(RTI::InteractionClassHandle hand);

protected:
    char* myName;
    RTI::InteractionClassHandle myHandle;
    RTI::AttributeHandleSet* myParams;
    RTI::AttributeHandleSet* myOwnParams;
    DtInterClassDesc* myBase;
    DtExerciseConn* myExConn;
    DtBoolean myPublishedFlag;
    DtBoolean myNeededFlag;

protected:
    static int theMaxParams;
};
#endif

```

intClassMap.h

```
// class DtInterClassMapping:
//
// Instances of DtInterClassMapping are mappings between names of VR-Link
// derived DtInteraction classes, and lists of names of FOM Interaction
// classes.

class DtInterClassMapping
{
public:
    // default constructor. vrlinkClassName is the name of
    // a VR-Link derived DtInteraction class. classes is a list
    // of RTI::InteractionClassHandles.
    DtInterClassMapping(const char* vrlinkClassName, DtList* classes);

    // destructor
    virtual ~DtInterClassMapping();

    // If a copy constructor is implemented, the assignment operator should
    // also be implemented. Should generally implement both of them anyway
to // avoid memory problems due to inappropriate compiler provided defaults.
    // Access varies so need to put proper access keyword above these
    // declarations.

    // copy constructor
    DtInterClassMapping(const DtInterClassMapping& orig);

    // assignment operator
    DtInterClassMapping& operator=(const DtInterClassMapping& orig);

    // Returns the name of the VR-Link derived DtInteraction class.
    const char* vrlinkClassName() const;

    // Returns the list of RTI::InteractionClassHandles that can
    // be associated with the VR-Link class.
    DtList* interClassNameList();

protected:
    const char* myVrlClassName;
    DtList* myList;
};
```


intDecFact.h

```

#if DtHLA

#include "interDecoder.h"
#include "MachineTypes.h"

// class DtInteractionDecoderFactory:
//
// Instances of DtInteractionDecoderFactory are used to create decoders
// which
// know how to decode an HLA interaction message (or network representation)
// into an interaction repository (or host representation).

class DtInteractionDecoderFactory
{
public:
    // constructor
    DtInteractionDecoderFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtInteractionDecoderFactory();

    // copy constructor
    // Copies across added decoder creation functions.
    DtInteractionDecoderFactory(const DtInteractionDecoderFactory& orig);

    // assignment operator
    // Copies across added decoder creation functions.
    DtInteractionDecoderFactory& operator=(
        const DtInteractionDecoderFactory& orig);

    // Create a decoder for the specified interaction class.
    // Note: Caller is responsible for deleting.
    // Returns NULL if decoder creation function is not in the factory.
    virtual DtInteractionDecoder* createDecoder(
        RTI::InteractionClassHandle hand);

    // Add a decoder prototype to the factory. The decoder prototype
    // will be cloned by createDecoder when called for the appropriate
    // handle. Caller deletes the decoder, but AFTER the decoder
    // factory no longer needs it. Returns any previous decoder which
    // was replaced by the add.
    virtual DtInteractionDecoder* addDecoder(
        RTI::InteractionClassHandle hand,
        DtInteractionDecoder* decoder);

    // This function is used to add an parameter decoder function
    // to several prototype decoders all at once - namely, those
    // associated with the interaction class denoted by hand, as well
    // as all of its subclasses. If applyToSubclasses is DtFalse,
    // then the function is only registered with the one decoder
    // prototype, making it equivalent to
    // decoder(hand)->addDecoder(hand, decoder));
    virtual void addParameterDecoder(
        RTI::InteractionClassHandle classHand,
        RTI::ParameterHandle paramHand,
        DtInteractionDecoder::DtParameterDecoder decoder,
        DtBoolean applyToSubclasses = DtTrue);

    // Same as above, but identify class by name
    void addParameterDecoder(
        char* className,
        RTI::ParameterHandle paramHand,

```

```
        DtInteractionDecoder::DtParameterDecoder func,  
        DtBoolean applyToSubclasses = DtTrue);  
  
// Same as above, but identify param by name  
void addParameterDecoder(  
    RTI::InteractionClassHandle classHand,  
    char* paramName,  
    DtInteractionDecoder::DtParameterDecoder func,  
    DtBoolean applyToSubclasses = DtTrue);  
  
// Same as above, but identify class and param by name  
void addParameterDecoder(  
    char* className,  
    char* paramName,  
    DtInteractionDecoder::DtParameterDecoder func,  
    DtBoolean applyToSubclasses = DtTrue);  
  
// Returns the decoder prototype associated with handle by a previous  
// call to addDecoder.  
virtual DtInteractionDecoder* decoder(RTI::InteractionClassHandle hand);  
  
protected:  
  
    // myProtos is an array of DtInteraction::maxInteractionClasses() items.  
    DtInteractionDecoder** myProtos;  
    DtExerciseConn* myExConn;  
};  
#endif
```

intEncFact.h

```

#if DtHLA

#include "interEncoder.h"
#include "MachineTypes.h"

// class DtInteractionEncoderFactory:
//
// Instances of DtInteractionEncoderFactory are used to create encoders
// which know how to encode an interaction repository (or host
// representation) into an HLA interaction message (or network
// representation).

class DtInteractionEncoderFactory
{
public:
    // constructor
    DtInteractionEncoderFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtInteractionEncoderFactory();

    // copy constructor
    DtInteractionEncoderFactory(const DtInteractionEncoderFactory& orig);

    // assignment operator
    DtInteractionEncoderFactory& operator=(
        const DtInteractionEncoderFactory& orig);

    // Create an encoder for the specified interaction class.
    // Note: Caller is responsible for deleting.
    // Returns NULL if encoder creation function is not in the factory.
    virtual DtInteractionEncoder* createEncoder(
        RTI::InteractionClassHandle hand);

    // Add a encoder prototype to the factory. The encoder prototype
    // will be cloned by createEncoder when called for the appropriate
    // handle. Caller deletes the encoder, but AFTER the encoder
    // factory no longer needs it. Returns any previous encoder which was
    // replaced by the add.
    virtual DtInteractionEncoder* addEncoder(
        RTI::InteractionClassHandle hand,
        DtInteractionEncoder* encoder);

    // This function is used to add an parameter encoder function
    // to several prototype encoders all at once - namely, those
    // associated with the interaction class denoted by hand, as well
    // as all of its subclasses. If applyToSubclasses is DtFalse,
    // then the function is only registered with the one encoder
    // prototype, making it equivalent to
    // encoder(hand)->addEncoder(hand, encoder));
    virtual void addParameterEncoder(
        RTI::InteractionClassHandle classHand,
        RTI::ParameterHandle paramHand,
        DtInteractionEncoder::DtParameterEncoder encoder,
        DtBoolean applyToSubclasses = DtTrue);

    // Same as above, but identify class by name
    void addParameterEncoder(
        char* className,
        RTI::ParameterHandle paramHand,
        DtInteractionEncoder::DtParameterEncoder func,
        DtBoolean applyToSubclasses = DtTrue);
}

```

```
// Same as above, but identify param by name
void addParameterEncoder(
    RTI::InteractionClassHandle classHand,
    char* paramName,
    DtInteractionEncoder::DtParameterEncoder func,
    DtBoolean applyToSubclasses = DtTrue);

// Same as above, but identify class and param by name
void addParameterEncoder(
    char* className,
    char* paramName,
    DtInteractionEncoder::DtParameterEncoder func,
    DtBoolean applyToSubclasses = DtTrue);

// Returns the encoder prototype associated with handle by a previous
// call to addEncoder.
virtual DtInteractionEncoder* encoder(RTI::InteractionClassHandle hand);

protected:

    // myProtos is an array of DtInteraction::maxInteractionClasses() items.
    DtInteractionEncoder** myProtos;
    DtExerciseConn* myExConn;
};
#endif
```

intercmCtlPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"

#include "packets/intercmCtlP.h"

#include "intercmParRec.h"
#include "intercmEntDst.h"
#include "intercmGrpAsg.h"
#include "intercmGrpDst.h"

// class DtIntercomControlPdu:
//
// Instances of DtIntercomControlPdu are used to communicate state
// information for simulated intercom devices.

// DtIntercomControlPdu callback function signature.
typedef void (*DtIntercomControlPduCb)(DtIntercomControlPdu* pdu, void*
usr);

class DtIntercomControlPdu : public DtPdu
{
public:
    // Constructors
    DtIntercomControlPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIntercomControlPdu(const DtNetIntercomControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIntercomControlPdu(const DtIntercomControlPdu& orig);

    // destructor
    virtual ~DtIntercomControlPdu();

    // Assignment operator
    DtIntercomControlPdu& operator=(const DtIntercomControlPdu& orig);

    // Set/Get intercom control type
    virtual void setControlType(DtIntercomCtlType controlType);
    virtual DtIntercomCtlType controlType() const;

    // Set/Get intercom communications channel class
    virtual void setCommClass(DtIntercomCommClass commClass);
    virtual DtIntercomCommClass commClass() const;

    // Set/Get intercom communications channel type
    virtual void setCommType(DtIntercomCommType commType);
    virtual DtIntercomCommType commType() const;

    // Set/Get source entity identifier
    virtual void setSourceEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& sourceEntityID() const;

    // Set/Get source communications device ID
    virtual void setSourceDeviceID(DtU16 deviceID);
    virtual DtU16 sourceDeviceID() const;

    // Set/Get source line ID
    virtual void setSourceLineID(DtU8 lineID);

```

```
virtual DtU8 sourceLineID() const;

// Set/Get transmit priority
virtual void setTransmitPriority(DtU8 transmitPriority);
virtual DtU8 transmitPriority() const;

// Set/Get transmit line state
virtual void setTransmitLineState(DtIntercomTransLineState lineState);
virtual DtIntercomTransLineState transmitLineState() const;

// Set/Get intercom command type
virtual void setCmdType(DtIntercomCmdType cmdType);
virtual DtIntercomCmdType cmdType() const;

// Set/Get master entity identifier
virtual void setMasterEntityID(const DtEntityIdentifier& entity);
virtual const DtEntityIdentifier& masterEntityID() const;

// Set/Get master communications device ID
virtual void setMasterDeviceID(DtU16 deviceID);
virtual DtU16 masterDeviceID() const;

// Set/Get master channel ID
virtual void setMasterChannelID(DtU16 channelID);
virtual DtU16 masterChannelID() const;

// Get the parameters length. This field is automatically changed
// by the functions: setNumParmRecords(), and setParmRecord()
virtual DtU32 parmRecordsLength() const;

// Set/Get number of parameter records (this is not a PDU field).
// If the required space would exceed the maximum space permitted for
// a PDU, then DtFalse may be returned.
virtual DtBoolean setNumParmRecords(int numRecords);
virtual int numParmRecords() const;

// Get the record type for parameter record parmIndex. If parmIndex is
// out of range or the parameter is invalid, then DtFalse is returned.
virtual DtBoolean getRecordType(int parmIndex,
    DtIntercomCtlRecType& recordType) const;

// Set/Get the parameter record type and values of record parmIndex to
// values specified by the initializer in the second argument. If
parmIndex
// is out of bounds, or if the record type does not match the function
// (for a get), then DtFalse is returned. If the space would exceed the
// maximum space permitted for a PDU (for a set) then DtFalse may be
// returned.
virtual DtBoolean setParmRecord(int parmIndex,
    const DtIntercomEntityDestRec& init);
virtual DtBoolean getParmRecord(int parmIndex,
    DtIntercomEntityDestRec& record) const;
virtual DtBoolean setParmRecord(int parmIndex,
    const DtIntercomGroupDestRec& init);
virtual DtBoolean getParmRecord(int parmIndex,
    DtIntercomGroupDestRec& record) const;
virtual DtBoolean setParmRecord(int parmIndex,
    const DtIntercomGroupAsgmtRec& init);
virtual DtBoolean getParmRecord(int parmIndex,
    DtIntercomGroupAsgmtRec& record) const;

// Print the data contained by this PDU
virtual void printData() const;

public:

    // Create a DtIntercomControlPdu.
```

```

// Caller is responsible for deletion.
static DtPdu* create(const DtNetPacket* initial,
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtIntercomControlPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtIntercomControlPduCb cb, void* usr);

protected:

// Returns the internal PDU kind
virtual DtPduKind internalGetPduKind() const;

// Casts the netPdu element to our network struct
virtual DtNetIntercomControlPdu* netPduStruct() const;

// Casts the 'this' pointer for changing elements in a const function
// Non-virtual since you can't override based on return type
DtIntercomControlPdu* self() const;

// Returns a pointer to the first byte of variable length data
virtual char* var() const;

// Returns the offset (in bytes) of the given parameter record
virtual int recordOffset(int recordIndex) const;

// Resize a given parameter record to the size indicated
virtual void resizeRecord(DtNetIntercomParmRec* rec, int newSize);

protected:

// Size of PDU with zero parameter records.
static int theStaticSize;

protected:

// Number of parameter records in this PDU
int myNumParmRecords;

// Temporary object for reference returns
DtEntityIdentifier myEntityID;
};
#endif
#endif

```

intercmEntDst.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/intercmCtlP.h"
#include "intercmParRec.h"

// class DtIntercomEntityDestRec:
//
// Instances of DtIntercomEntityDestRec are used to contain the information
// in the entity destination parameter records used by the intercom control
// PDU class.

class DtIntercomEntityDestRec : public DtIntercomParmRec
{
public:
    // Constructors
    DtIntercomEntityDestRec();
    DtIntercomEntityDestRec(const DtIntercomEntityDestRec& orig);

    // Destructor
    virtual ~DtIntercomEntityDestRec();

    // Assignment operators
    DtIntercomEntityDestRec& operator=(
        const DtIntercomEntityDestRec& orig);
    DtIntercomEntityDestRec& operator=(
        const DtNetIntercomEntityDestRec& orig);

    // Set/Get the destination entity identifier
    virtual void setDestEntityID(const DtEntityIdentifier& entityID);
    virtual const DtEntityIdentifier& destEntityID() const;

    // Set/Get the destination device ID
    virtual void setDestDeviceID(DtU16 deviceID);
    virtual DtU16 destDeviceID() const;

    // Set/Get the destination line ID
    virtual void setDestLineID(DtU8 lineID);
    virtual DtU8 destLineID() const;

    // Set/Get the destination priority
    virtual void setDestPriority(DtU8 priority);
    virtual DtU8 destPriority() const;

    // Set/Get the destination line state command
    virtual void setDestLineStateCmd(DtIntercomDestLineStateCmd cmd);
    virtual DtIntercomDestLineStateCmd destLineStateCmd() const;

    // Print this record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(const DtNetIntercomEntityDestRec& record);
    virtual void loadNetRecord(DtNetIntercomEntityDestRec& record) const;

protected:
```



```
DtEntityIdentifier      myDestEntityID;  
DtU16                  myDestDeviceID;  
DtU8                   myDestLineID;  
DtU8                   myDestPriority;  
DtIntercomDestLineStateCmd myDestLineStateCmd;  
};  
#endif  
#endif
```

intercmGrpAsg.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/intercmCtlP.h"
#include "intercmParRec.h"

// class DtIntercomGroupAsgmtRec:
//
// Instances of DtIntercomGroupAsgmtRec are used to contain the information
// in the group assignment parameter record used by the intercom control
// PDU class.

class DtIntercomGroupAsgmtRec : public DtIntercomParmRec
{
public:
    // Constructors
    DtIntercomGroupAsgmtRec();
    DtIntercomGroupAsgmtRec(const DtIntercomGroupAsgmtRec& orig);

    // Destructor
    virtual ~DtIntercomGroupAsgmtRec();

    // Assignment operators
    DtIntercomGroupAsgmtRec& operator=(
        const DtIntercomGroupAsgmtRec& orig);
    DtIntercomGroupAsgmtRec& operator=(
        const DtNetIntercomGroupAsgmtRec& orig);

    // Set/Get the group bit field
    virtual void setGroupBitField(DtU32 bitField);
    virtual DtU32 groupBitField() const;

    // Set/Get the destination entity identifier
    virtual void setDestEntityID(const DtEntityIdentifier& entityID);
    virtual const DtEntityIdentifier& destEntityID() const;

    // Set/Get the destination equipment ID
    virtual void setDestEquipID(DtU16 equipID);
    virtual DtU16 destEquipID() const;

    // Set/Get the destination line ID
    virtual void setDestLineID(DtU8 lineID);
    virtual DtU8 destLineID() const;

    // Print this record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(const DtNetIntercomGroupAsgmtRec& record);
    virtual void loadNetRecord(DtNetIntercomGroupAsgmtRec& record) const;

protected:
    DtU32                myGroupBitField;
    DtEntityIdentifier    myDestEntityID;
    DtU16                myDestEquipID;
    DtU8                 myDestLineID;
```

```
};  
#endif  
#endif
```

intercmGrpDst.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"

#include "packets/intercmCtlP.h"
#include "intercmParRec.h"

// class DtIntercomGroupDestRec:
//
// Instances of DtIntercomGroupDestRec are used to contain the information
// in the group destination parameter record used by the intercom control
// PDU class.

class DtIntercomGroupDestRec : public DtIntercomParmRec
{
public:
    // Constructors
    DtIntercomGroupDestRec();
    DtIntercomGroupDestRec(const DtIntercomGroupDestRec& orig);

    // Destructor
    virtual ~DtIntercomGroupDestRec();

    // Assignment operators
    DtIntercomGroupDestRec& operator=(
        const DtIntercomGroupDestRec& orig);
    DtIntercomGroupDestRec& operator=(
        const DtNetIntercomGroupDestRec& orig);

    // Set/Get the group bit field
    virtual void setDestGroupBitField(DtU32 bitField);
    virtual DtU32 destGroupBitField() const;

    // Set/Get the destination priority
    virtual void setDestPriority(DtU8 priority);
    virtual DtU8 destPriority() const;

    // Set/Get the destination line state command
    virtual void setDestLineStateCmd(DtIntercomDestLineStateCmd cmd);
    virtual DtIntercomDestLineStateCmd destLineStateCmd() const;

    // Print this record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(const DtNetIntercomGroupDestRec& record);
    virtual void loadNetRecord(DtNetIntercomGroupDestRec& record) const;

protected:
    DtU32                myDestGroupBitField;
    DtU8                 myDestPriority;
    DtIntercomDestLineStateCmd myDestLineStateCmd;
};
#endif
#endif
```

intercmParRec.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

// class DtIntercomParmRec:
//
// DtIntercomParmRec is an abstract base class for the parameter record
// classes which are used by the intercom control PDU (DtIntercomCon-
// trolPdu).

class DtIntercomParmRec
{
public:
    // Destructor
    virtual ~DtIntercomParmRec();

    // Get the record type
    virtual DtIntercomCtlRecType recordType() const;

    // Get the record length
    virtual DtU16 recordLength() const;

    // Print the record type and length
    virtual void print() const;

protected:
    // Constructors
    DtIntercomParmRec();
    DtIntercomParmRec(const DtIntercomParmRec& orig);
    DtIntercomParmRec(DtIntercomCtlRecType recordType,
        int recordLength);

    // Assignment operator
    DtIntercomParmRec& operator=(const DtIntercomParmRec& orig);

protected:
    // Data members:
    DtIntercomCtlRecType    myRecordType;
    DtU16                  myRecordLength;
};
#endif
#endif

```

intercmSigPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"

#include "packets/intercmSigP.h"

// class DtIntercomSignalPdu:
//
// Instances of DtIntercomSignalPdu are used to send simulated intercom
// signal data.

// DtIntercomSignalPdu callback function signature.
typedef void (*DtIntercomSignalPduCb)(DtIntercomSignalPdu* pdu, void* usr);

class DtIntercomSignalPdu : public DtPdu
{
public:
    // Constructors
    DtIntercomSignalPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIntercomSignalPdu(const DtNetIntercomSignalPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIntercomSignalPdu(const DtIntercomSignalPdu& orig);

    // destructor
    virtual ~DtIntercomSignalPdu();

    // Assignment operator
    DtIntercomSignalPdu& operator=(const DtIntercomSignalPdu& orig);

    // Set/get transmitting entity identifier
    virtual void setTransEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& transEntityID() const;

    // Set/get intercom ID
    virtual void setDeviceID(DtU16 intercomID);
    virtual DtU16 deviceID() const;

    // Set/get intercom encoding class
    virtual void setEncodingClass(DtEncodingClass encClass);
    virtual DtEncodingClass encodingClass() const;

    // Set/get intercom encoding type
    virtual void setEncodingType(DtU16 encType);
    virtual DtU16 encodingType() const;

    // Set/get intercom TDL type
    virtual void setTDL_Type(DtU16 TDL_Type);
    virtual DtU16 TDL_Type() const;

    // Set/get sample rate
    virtual void setSampleRate(DtU32 rate);
    virtual DtU32 sampleRate() const;

    // Set/get data length in bits
    virtual void setNumDataBits(DtU16 lengthInBits);
    virtual DtU16 numDataBits() const;
```

```

    // Set/get data length in bytes
    virtual void setNumDataBytes(DtU16 lengthInBytes);
    virtual DtU16 numDataBytes() const;

    // Set/get number of samples
    virtual void setNumSamples(DtU16 samples);
    virtual DtU16 numSamples() const;

    // Set/get a data byte
    virtual void setDataByte(int byteNum, DtU8 data);
    virtual DtU8 dataByte(int byteNum) const;

    // Set/get multiple data bytes. Provide a pointer and data is
    // copied to and from that memory. Check numDataBytes() to make sure
that
    // you provide enough room in the buffer.
    virtual void setData(const DtU8* data, int numBytes);
    virtual void getData(DtU8* buff) const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:
    // Create a DtIntercomSignalPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtIntercomSignalPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtIntercomSignalPduCb cb, void* usr);

protected:
    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetIntercomSignalPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non virtual since you can't override based on return type
    DtIntercomSignalPdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

protected:
    // Size of PDU with zero data bytes.
    static int theStaticSize;

protected:
    // Temporary object for reference returns
    DtEntityIdentifier myEntityID;
};
#endif
#endif

```

interDecoder.h

```
#if DtHLA

#include "RTI.hh"
#include "vlprint.h"

#include "chooseDefines.h"

// DtInteractionRep is not a real class. It is a placeholder denoting
// any class that is a container for an interaction's values. For example,
// our derived DtInteraction classes currently are DtInteractionReps.
// class DtInteractionDecoder:
//
//
// Instances of DtInteractionDecoder are used to decode interactions
// from the network.

class DtInteractionDecoder
{
public:

    // Parameter decoder function signature.
    typedef void (*DtParameterDecoder)(DtInteractionRep* inter,
        const RTI::ParameterHandleValuePairSet& pvlist,
        int i);

public:

    // constructor
    DtInteractionDecoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtInteractionDecoder();

    // copy constructor
    DtInteractionDecoder(const DtInteractionDecoder& orig);

    // assignment operator
    DtInteractionDecoder& operator=(const DtInteractionDecoder& orig);

    // Add/Remove a decoding function for the parameter specified
    // by handle.
    // Returns the decoding function which is replaced by the add/remove.
    // Note: This is non-virtual since it is typically called by
    // constructors.
    DtParameterDecoder addDecoder(RTI::ParameterHandle handle,
        DtParameterDecoder proc);

    // Add/Remove a decoding function for the parameter specified
    // by name.
    // Returns the decoding function which is replaced by the add/remove.
    // Note: This is non-virtual since it is typically called by
    // constructors.
    DtParameterDecoder addDecoder(char* name,
        DtParameterDecoder proc);

    // Get the parameter decoder for the specified handle.
    virtual DtParameterDecoder decoder(int handle) const;

    // Decode the interaction.
    virtual void decode(DtInteractionRep* interaction,
        const RTI::ParameterHandleValuePairSet& pvList);

    // Get the interaction's class handle.
```



```

virtual RTI::InteractionClassHandle interactionClassHandle();

// Returns a new-ed clone of this object
virtual DtInteractionDecoder* clone() const;

public:

    // Returns a pointer to a large static buffer that can be used to
    // copy data obtained from a phvps.
    static char* decodingBuffer();

protected:

    // Process (decode) the specified parameter.
    virtual void decodeParameter(DtInteractionRep* inter,
        const RTI::ParameterHandleValuePairSet& pvList, int i);

protected:

    DtExerciseConn* myExConn;
    DtInterClassDesc* myClassDesc;

    DtVrlRtiAmbassador* myRtiAmb;
    RTI::InteractionClassHandle myClassHandle;

    // myList is an array of theMaxParametersPerInteraction items.
    DtParameterDecoder* myList;

protected:

    static const int theMaxParametersPerInteraction;
    static char theDecBuffer[2000];
};
// This can be used to add a parameter decoder function from a
// routine.
#define DtADD_PARAM_DECODER(attrName) \
    addDecoder(#attrName, (DtParameterDecoder) decode##attrName)

// This can be used to declare attributeDecoder functions
// which are static members of a class.
#define DtDECLARE_PARAM_DECODER(RepT, paramName) \
    static void decode##paramName(RepT* inter, \
        const RTI::ParameterHandleValuePairSet& params, int pairSetIndex)

#define DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST(decClass, interClass, \
    paramName, netType, mutator, castExpr) \
void decClass::decode##paramName(interClass* inter, \
    const RTI::ParameterHandleValuePairSet& params, \
    int index) \
{ \
    RTI::ULong length = 0; \
    params.getValue(index, DtInteractionDecoder::decodingBuffer(), length); \
    \
    netType* netVal = (netType*) DtInteractionDecoder::decodingBuffer(); \
    if (length > sizeof(netType)) \
    { \
        DtWarn("Size of value decoded for parameter %s (%d)\n", \
            #paramName, length); \
        DtWarn(" is larger than size of %s (%d)\n", #netType, \
            sizeof(netType)); \
    } \
    inter->mutator(castExpr *netVal); \
}

#define DtDEFINE_SIMPLE_PARAM_DECODER(decClass, interClass, \
    paramName, netType, mutator) \
    DtDEFINE_SIMPLE_PARAM_DECODER_WITH_CAST(decClass, interClass, \

```

```
        paramName, netType, mutator, (netType))  
#endif
```

interEncoder.h

```

#if DtHLA

#include "RTI.hh"

#include "chooseDefines.h"

// DtInteractionRep is not a real class. It is a placeholder denoting
// any class that is a container for an interaction's values. For example,
// our derived DtInteraction classes currently are DtInteractionReps.
// class DtInteractionEncoder:
//
//
// DtInteractionEncoder is used to provide encoding of an
// interaction into an RTI::ParameterHandleValuePairSet in an HLA
// (as opposed to DIS) environment.

class DtInteractionEncoder
{
public:

    // constructor
    DtInteractionEncoder(DtExerciseConn* exConn,
        DtInterClassDesc* classDesc);

    // destructor
    virtual ~DtInteractionEncoder();

    // copy constructor
    DtInteractionEncoder(const DtInteractionEncoder& orig);

    // assignment operator
    DtInteractionEncoder& operator=(const DtInteractionEncoder& orig);

    // Attribute encoding function declaration.
    typedef void (*DtParameterEncoder)
        (const DtInteractionRep& inter,
         RTI::ParameterHandleValuePairSet* avlist,
         RTI::ParameterHandle handle);

    // Add a parameter encoder function by parameter handle.
    // Returns any previous parameter encoder function which was
    // replaced by the add.
    virtual DtParameterEncoder addEncoder(RTI::ParameterHandle h,
        DtParameterEncoder func);

    // Add a parameter encoder function by parameter name.
    // Returns any previous parameter encoder function which was
    // replaced by the add.
    virtual DtParameterEncoder addEncoder(char* name,
        DtParameterEncoder proc);

    // Get the parameter encoder for the specified handle.
    virtual DtParameterEncoder encoder(int handle) const;

    // Encode necessary values from the protocol-independent interaction.
    virtual void encode(const DtInteractionRep& inter,
        RTI::ParameterHandleValuePairSet* pvList);

    // Get the object's class handle.
    virtual RTI::InteractionClassHandle interClassHandle();

    // Returns a new-ed clone of this object
    virtual DtInteractionEncoder* clone() const;

```

protected:

```
DtExerciseConn* myExConn;
DtInterClassDesc* myClassDesc;

DtVrIRtiAmbassador* myRtiAmb;
RTI::InteractionClassHandle myHandle;

// myList is an array of theMaxParametersPerInteraction items.
DtParameterEncoder* myList;
```

protected:

```
    static const int theMaxParametersPerInteraction;
};
#define DtADD_PARAM_ENCODER(attrName) \
    addEncoder(#attrName, (DtParameterEncoder) encode##attrName)

#define DtDECLARE_PARAM_ENCODER(interClass, paramName) \
    static void encode##paramName(const interClass& inter, \
        RTI::ParameterHandleValuePairSet* params, \
        RTI::ParameterHandle paramHandle)

#define DtDEFINE_SIMPLE_PARAM_ENCODER(encClass, interClass, \
        paramName, netType, \
        inspector) \
void encClass::encode##paramName(const interClass& inter, \
    RTI::ParameterHandleValuePairSet* params, \
    RTI::ParameterHandle paramHandle) \
{ \
    netType netVal; \
    netVal = inter.inspector(); \
    params->add(paramHandle, (char*) &netVal, sizeof(netType)); \
}

#endif
```

interMgr.h

```

#ifdef DtHLA

#include "hInterFactory.h"
#include "callbacks.h"

// class DtInteractionManager:
//
// Instances of DtInteractionManager are used to manage the creation
// and processing of HLA interactions.

class DtInteractionManager
{
public:
    // constructor
    DtInteractionManager(DtInteractionFactory* factory);

    // destructor
    virtual ~DtInteractionManager();

    // copy constructor
    // Care needs to be taken since this copies the callbacks.
    // This means the manager will have a pointer to usr data supplied
    // to the other manager which the original owner may not be aware
    // of or anticipate.
    DtInteractionManager(const DtInteractionManager& orig);

    // assignment operator
    // Care needs to be taken since this copies the callbacks.
    // This means the manager will have a pointer to usr data supplied
    // to the other manager which the original owner may not be aware
    // of or anticipate.
    DtInteractionManager& operator=(const DtInteractionManager& orig);

    // Add/remove a callback to be called when an interaction is
    // received from another federate.
    virtual void addInteractionCallback(RTI::InteractionClassHandle h,
        DtReceiveInteractionCb cb,
        void* usr);
    virtual void removeInteractionCallback(RTI::InteractionClassHandle h,
        DtReceiveInteractionCb cb,
        void* usr);

    // Create an interaction object, call the registered callbacks,
    // then delete the interaction object.
    virtual void processInteraction(RTI::InteractionClassHandle h,
        const RTI::ParameterHandleValuePairSet& p,
        const char* tag = NULL);

protected:
    // Copy callback list.
    // Non-virtual since meant to be called from constructor.
    void copyCbList(const DtInteractionManager& orig);

    // Empty callback list.
    // Non-virtual since meant to be called from destructor.
    void emptyCbList();

protected:
    DtInteractionFactory* myFactory;
    // myCbList is an array of DtInteraction::maxInteractionClasses() items.

```

```
    DtList* myCbList;  
};  
#endif
```

intHashKey.h

```
#include "baseHashKey.h"

// class DtIntHashKey:
//
// Instances of DtIntHashKey are used as a hash key made from an integer.
class DtIntHashKey : public DtBaseHashKey
{
public:
    // default constructor
    DtIntHashKey(unsigned long num);

    // destructor
    virtual ~DtIntHashKey();

    // copy constructor
    DtIntHashKey(const DtIntHashKey& orig);

    // assignment operator
    DtIntHashKey& operator=(const DtIntHashKey& orig);

    // make a copy of this hash key
    virtual DtBaseHashKey* clone() const;

protected:
    // get the string representation of this key
    virtual const DtString& stringRep() const;

    // compute the hashed value for this key
    virtual DtU32 computeHashValue();

protected:
    unsigned long myNum;
    DtString myStr;
};
```

isGroupOfPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/isGroupOfP.h"

#include "gedBasFixWing.h"
#include "gedBasRotWing.h"
#include "gedBasSoldier.h"
#include "gedBasVehicle.h"
#include "gedEnhFixWing.h"
#include "gedEnhRotWing.h"
#include "gedEnhSoldier.h"
#include "gedEnhVehicle.h"
#include "gedLogVehicle.h"

// class DtIsGroupOfPdu:
//
// Instances of DtIsGroupOfPdu are used to communicate information about a
// particular group of entities (grouped together for the purposes of
// network
// bandwidth reduction).

// DtIsGroupOfPdu callback function signature.
typedef void (*DtIsGroupOfPduCb)(DtIsGroupOfPdu* pdu, void* usr);

class DtIsGroupOfPdu : public DtPdu
{
public:
    // Constructors
    DtIsGroupOfPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIsGroupOfPdu(const DtNetIsGroupOfPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIsGroupOfPdu(const DtIsGroupOfPdu& orig);

    // destructor
    virtual ~DtIsGroupOfPdu();

    // Assignment operator
    DtIsGroupOfPdu& operator=(const DtIsGroupOfPdu& orig);

    // Set/Get group entity ID
    virtual void setGroupEntityID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& groupEntityID() const;

    // Set/Get the grouped entity category (setCategory will also set the
    // number
    // of entities to zero)
    virtual void setCategory(DtGroupedEntityCategory value);
    virtual DtGroupedEntityCategory category() const;

    // Set/Get the number of entities
    virtual DtBoolean setNumEntities(DtU8 num);
    virtual DtU8 numEntities() const;

    // Set/Get the reference latitude
```



```

virtual void setLatitude(DtFloat64 value);
virtual DtFloat64 latitude() const;

// Set/Get the reference longitude
virtual void setLongitude(DtFloat64 value);
virtual DtFloat64 longitude() const;

// Set/Get a BasicFixedWingAircraftRecord. Returns DtFalse if the
// index is out of range, or if the entity category does not match.
virtual DtBoolean setBasicFixedWingAircraftRecord(int index,
    const DtBasicFixedWingAircraftRecord& init);
virtual DtBoolean getBasicFixedWingAircraftRecord(int index,
    DtBasicFixedWingAircraftRecord& record) const;

// Set/Get an EnhancedFixedWingAircraftRecord. Returns DtFalse if the
// index is out of range, or if the entity category does not match.
virtual DtBoolean setEnhancedFixedWingAircraftRecord(int index,
    const DtEnhancedFixedWingAircraftRecord& init);
virtual DtBoolean getEnhancedFixedWingAircraftRecord(int index,
    DtEnhancedFixedWingAircraftRecord& record) const;

// Set/Get a BasicRotorWingAircraftRecord. Returns DtFalse if the
// index is out of range, or if the entity category does not match.
virtual DtBoolean setBasicRotorWingAircraftRecord(int index,
    const DtBasicRotorWingAircraftRecord& init);
virtual DtBoolean getBasicRotorWingAircraftRecord(int index,
    DtBasicRotorWingAircraftRecord& record) const;

// Set/Get an EnhancedRotorWingAircraftRecord. Returns DtFalse if the
// index is out of range, or if the entity category does not match.
virtual DtBoolean setEnhancedRotorWingAircraftRecord(int index,
    const DtEnhancedRotorWingAircraftRecord& init);
virtual DtBoolean getEnhancedRotorWingAircraftRecord(int index,
    DtEnhancedRotorWingAircraftRecord& record) const;

// Set/Get a BasicGroundCombatSoldierRecord. Returns DtFalse if the
// index is out of range, or if the entity category does not match.
virtual DtBoolean setBasicGroundCombatSoldierRecord(int index,
    const DtBasicGroundCombatSoldierRecord& init);
virtual DtBoolean getBasicGroundCombatSoldierRecord(int index,
    DtBasicGroundCombatSoldierRecord& record) const;

// Set/Get an EnhancedGroundCombatSoldierRecord. Returns DtFalse if the
// index is out of range, or if the entity category does not match.
virtual DtBoolean setEnhancedGroundCombatSoldierRecord(int index,
    const DtEnhancedGroundCombatSoldierRecord& init);
virtual DtBoolean getEnhancedGroundCombatSoldierRecord(int index,
    DtEnhancedGroundCombatSoldierRecord& record) const;

// Set/Get a BasicGroundCombatVehicleRecord. Returns DtFalse if the
// index is out of range, or if the entity category does not match.
virtual DtBoolean setBasicGroundCombatVehicleRecord(int index,
    const DtBasicGroundCombatVehicleRecord& init);
virtual DtBoolean getBasicGroundCombatVehicleRecord(int index,
    DtBasicGroundCombatVehicleRecord& record) const;

// Set/Get an EnhancedGroundCombatVehicleRecord. Returns DtFalse if the
// index is out of range, or if the entity category does not match.
virtual DtBoolean setEnhancedGroundCombatVehicleRecord(int index,
    const DtEnhancedGroundCombatVehicleRecord& init);
virtual DtBoolean getEnhancedGroundCombatVehicleRecord(int index,
    DtEnhancedGroundCombatVehicleRecord& record) const;

// Set/Get a GroundLogisticsVehicleRecord. Returns DtFalse if the
// index is out of range, or if the entity category does not match.
virtual DtBoolean setGroundLogisticsVehicleRecord(int index,
    const DtGroundLogisticsVehicleRecord& init);

```

```
virtual DtBoolean getGroundLogisticsVehicleRecord(int index,
    DtGroundLogisticsVehicleRecord& record) const;

// Print the data contained by this PDU
virtual void printData() const;

public:
    // Create a DtIsGroupOfPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtIsGroupOfPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtIsGroupOfPduCb cb, void* usr);

protected:
    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetIsGroupOfPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non-virtual since you can't override based on return type
    DtIsGroupOfPdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

    // Returns the record size for the grouped entity category
    virtual int recordSize() const;

protected:
    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:
    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
};
#endif
#endif
```

isPartOfPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"

#include "mathTypes.h"

#include "packets/isPartOfP.h"

// class DtIsPartOfPdu:
//
// Instances of DtIsPartOfPdu are used to join two simulation entities to
// form a single entity.

// DtIsPartOfPdu callback function signature.
typedef void (*DtIsPartOfPduCb)(DtIsPartOfPdu* pdu, void* usr);

class DtIsPartOfPdu : public DtPdu
{
public:
    // Constructors
    DtIsPartOfPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIsPartOfPdu(const DtNetIsPartOfPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtIsPartOfPdu(const DtIsPartOfPdu& orig);

    // destructor
    virtual ~DtIsPartOfPdu();

    // Assignment operator
    DtIsPartOfPdu& operator=(const DtIsPartOfPdu& orig);

    // Set/Get originating entity identifier
    virtual void setOriginatingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& originatingEntityID() const;

    // Set/Get receiving entity identifier
    virtual void setReceivingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& receivingEntityID() const;

    // Set/Get nature
    virtual void setNature(DtNature& value);
    virtual DtNature nature() const;

    // Set/Get position
    virtual void setPosition(DtPosition& value);
    virtual DtPosition position() const;

    // Set/Get part location
    virtual void setLocation(DtConstVector coords);
    virtual DtConstVector location() const;

    // Set/Get station name
    virtual void setStationName(DtStationName value);
    virtual DtStationName stationName() const;

    // Set/Get Station number
    virtual void setStationNumber(DtU16 value);

```

```
virtual DtU16 stationNumber() const;

// Set/Get Part Entity Type
virtual void setEntityType(const DtEntityType& type);
virtual const DtEntityType& entityType() const;

// Print the data contained by this PDU
virtual void printData() const;

public:

    // Create a DtIsPartOfPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtIsPartOfPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtIsPartOfPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    // Non-virtual since you can't override based on return type
    virtual DtNetIsPartOfPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non virtual since you can't override based on return type
    DtIsPartOfPdu* self() const;

protected:

    // Size of PDU with zero data bytes.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
    DtVector myVector;
    DtEntityType myEntityType;

};
#endif
#endif
```

laserPdu.h

```

#if !DtHLA

#if DtDISVERS == 3

#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "packets/laserP.h"
#define DtLaserStaticSize 72

class DtLaserPdu;
typedef void (*DtLaserPduCb)(DtLaserPdu *pdu, void *usr);

class DtLaserPdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtLaserPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLaserPdu(const DtNetLaserPdu *initial,
               DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtLaserPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtLaserPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &lasingEntityId() const;
    DtLaserCodeName codeName() const;
    const DtEntityIdentifier &lasedEntityId() const;
    DtLaserCode laserCode() const;
    float power() const;
    float wavelength() const;
    DtConstVector lasedSpotRelative() const;
    DtConstVector lasedSpotWorld() const;

/***** MUTATOR FUNCTIONS *****/

    void setLasingEntityId(const DtEntityIdentifier &id);
    void setCodeName(DtLaserCodeName name);
    void setLasedEntityId(const DtEntityIdentifier &id);
    void setLaserCode(DtLaserCode code);
    void setPower(float power);
    void setWavelength(float wavelength);
    void setLasedSpotRelative(DtConstVector loc);
    void setLasedSpotWorld(DtConstVector loc);

    void printData() const;
protected:
    DtPduKind internalGetPduKind() const {return DtLaserPduKind;}
    DtEntityIdentifier theId;
    DtVector loc;
};
#endif // DtDISVERS
#endif // !DtHLA

```

lcEnd.h

```
#include "lcPdu.h"

// class DtLgrEndPdu:
//
// Instances of DtLgrEndPdu are to tell remote logger to skip to end
// of file during playback.

class DtLgrEndPdu : public DtLgrControlPdu
{
public:

    // default constructor
    DtLgrEndPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrEndPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrEndPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator&
receiver,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrEndPdu();

    // copy constructor
    DtLgrEndPdu(const DtLgrEndPdu& orig);

    // assignment operator
    DtLgrEndPdu& operator=(const DtLgrEndPdu& orig);
};
```

lcEofAction.h

```

#include "lcPdu.h"

// class DtLgrEofActionPdu:
//
// Instances of DtLgrEofActionPdu are to tell remote logger that
// it should pause, wrap, or stop at the end of playback.

class DtLgrEofActionPdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrEofActionPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrEofActionPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrEofActionPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver,
        DtLgrMode mode, DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrEofActionPdu();

    // copy constructor
    DtLgrEofActionPdu(const DtLgrEofActionPdu& orig);

    // assignment operator
    DtLgrEofActionPdu& operator=(const DtLgrEofActionPdu& orig);

    // Set/Get value for what to do when Logger reaches EOF
    //
    virtual DtLgrMode action();

    // mutator functions
    virtual void setAction( DtLgrMode cmd );

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrIntParams* params();

private:
    // parameter
    DtLgrMode myAction;
};

```

lcHeartbeat.h

```
#include "lcPdu.h"

// class DtLgrHeartbeatPdu:
//
// Instances of DtLgrHeartbeatPdu are to tell remote logger to
// send packet for each playback entity at least every HEARTBEAT
// seconds. Important when in slow-motion.

class DtLgrHeartbeatPdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrHeartbeatPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrHeartbeatPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrHeartbeatPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver, DtReal hbeat,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrHeartbeatPdu();

    // copy constructor
    DtLgrHeartbeatPdu(const DtLgrHeartbeatPdu& orig);

    // assignment operator
    DtLgrHeartbeatPdu& operator=(const DtLgrHeartbeatPdu& orig);

    // set/get the heartbeat. see class description.
    virtual void setHeartbeat( DtReal val);
    virtual DtReal heartbeat();

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrRealParams* params();

private:
    // parameter
    DtReal myHeartbeat;
};
```


lcInter.h

```
// This file provides protocol independence for LgrControl Interactions.  
  
#if DtHLA  
#include "hLcInter.h"  
#endif  
  
#if !DtHLA  
#include "lcPdu.h"  
#endif
```

lcPause.h

```
#include "lcPdu.h"

// class DtLgrPausePdu:
//
// Instances of DtLgrPausePdu are to tell remote logger to pause

class DtLgrPausePdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrPausePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrPausePdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrPausePdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrPausePdu();

    // copy constructor
    DtLgrPausePdu(const DtLgrPausePdu& orig);

    // assignment operator
    DtLgrPausePdu& operator=(const DtLgrPausePdu& orig);
};
```

lcPdu.h

```

#include "chooseDefines.h"
#include "MachineTypes.h"
#include "LibMatrix.h"
#include "printUtil.h"
#include "parseEnum.h"
#include "exerciseConn.h"
#include "hostStructs.h"
#include "pdu.h"
#include "packets/lgrControlP.h"
#include "mathTypes.h"
#include "interaction.h"

#if !DtHLA

class DtExerciseConn;
class DtLgrControlPdu;

typedef DtLgrControlPdu DtLgrControlInteraction;
typedef void (*DtLgrControlInteractionCb)
    (DtLgrControlInteraction* inter, void* usr);

#endif

// class DtLgrControlPdu:
//
// Instances of DtLgrControlPdu are a family of messgae used to control
// a remote MaK-Logger over the network. See lc*.h

#if DtHLA
class DtLgrControlPdu : public DtPdu
#else
class DtLgrControlPdu : public DtInteraction
#endif
{
public:

    // default constructor
    DtLgrControlPdu();

    //Create a PDU representation from a network packet.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    //Create a PDU from DtLgrCommand;
    static DtLgrControlPdu* create(DtLgrCommand command);

#if !DtHLA
    //DIS -- install/remove handler to call when receiving a LC packet.
    static void addCallback(DtExerciseConn* conn, DtLgrControlInteractionCb
cb,
        void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtLgrControlInteractionCb cb, void* usr);
#endif

    // destructor
    virtual ~DtLgrControlPdu();

    // copy constructor
    DtLgrControlPdu(const DtLgrControlPdu& orig);

    // assignment operator
    DtLgrControlPdu& operator=(const DtLgrControlPdu& orig);

```

```
// set/get Object Identifier of intended recipient (logger)
virtual void setReceiverId(const DtGlobalObjectDesignator& id);
const DtGlobalObjectDesignator& receiverId() const;

// set/get Object Identifier of sender, if applicable.
virtual void setSenderId(const DtGlobalObjectDesignator& id);
virtual const DtGlobalObjectDesignator& senderId() const;

// get message-type for proper casting to a subclass.
DtLgrCommand lgrCommand() const;

//print contents
virtual void printData() const;

protected:
    // identity for use by toolkit.
    DtPduKind internalGetPduKind() const;

    // set is done by way of constructing the proper subclass.
    void setLgrCommand(DtLgrCommand com);

private:
    // parameter
    DtGlobalObjectDesignator mySenderId;
    DtGlobalObjectDesignator myReceiverId;
};

inline DtPduKind DtLgrControlPdu::internalGetPduKind() const
{
    return DtLgrControlPduKind;
}

//General operation modes
enum DtLgrMode
{
    DtLGR_PLAY,
    DtLGR_PAUSE,
    DtLGR_RECORD,
    DtLGR_STOP,
    DtLGR_EMPTY,
    DtLGR_LOOP
};

//Play forwards or backwards when in LGR_PLAY mode
enum DtLgrDirection
{
    DtLGR_STOPPED    = 0,
    DtLGR_NOPKT      = 0,
    DtLGR_FORWARDS   = 1,
    DtLGR_BACKWARDS  = -1
};

//Location status
enum DtLgrFilePosition
{
    DtLGR_NOFILE,
    DtLGR_START,
    DtLGR_MIDDLE,
    DtLGR_END
};

enum DtLgrStyle
{
    DtLGR_PRISTINE,
    DtLGR_RVA
};
```

```
//Receiving application may filter lcPdu based on this start-up flag.  
enum DtLgrProcess  
{  
    DtLGR_NONE,  
    DtLGR_MINE,  
    DtLGR_ALL  
};  
  
#endif
```

lcPlay.h

```
#include "lcPdu.h"

// class DtLgrPlayPdu:
//
// Instances of DtLgrPlayPdu are used to tell remote logger to begin
// recording

class DtLgrPlayPdu : public DtLgrControlPdu
{
public:

    // default constructor
    DtLgrPlayPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrPlayPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrPlayPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator&
receiver,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrPlayPdu();

    // copy constructor
    DtLgrPlayPdu(const DtLgrPlayPdu& orig);

    // assignment operator
    DtLgrPlayPdu& operator=(const DtLgrPlayPdu& orig);
};
```

lcPlayAction.h

```

#include "lcPdu.h"

// class DtLgrPlayActionPdu:
//
// Instances of DtLgrPlayActionPdu are to tell remote logger
// what action to take initially when set to playback mode.

class DtLgrPlayActionPdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrPlayActionPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrPlayActionPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrPlayActionPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDes-
ignator& receiver,
        DtLgrMode mode, DtBufferPtr buffer =DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrPlayActionPdu();

    // copy constructor
    DtLgrPlayActionPdu(const DtLgrPlayActionPdu& orig);

    // assignment operator
    DtLgrPlayActionPdu& operator=(const DtLgrPlayActionPdu& orig);

    // set/get action to take place when logger switched to playback mode.
    virtual void setAction( DtLgrMode cmd );
    virtual DtLgrMode action();

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrIntParams* params();

private:
    // parameter
    DtLgrMode myAction;
};

```

lcPlayFile.h

```
#include "lcPdu.h"

// class DtLgrPbFilePdu:
//
// Instances of DtLgrPbFilePdu are used for switching to play-mode and
// defining
// which file to read from when sending traffic into the simulation.

class DtLgrPbFilePdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrPbFilePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrPbFilePdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrPbFilePdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver, char* filename,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrPbFilePdu();

    // copy constructor
    DtLgrPbFilePdu(const DtLgrPbFilePdu& orig);

    // assignment operator
    DtLgrPbFilePdu& operator=(const DtLgrPbFilePdu& orig);

    // set/get file to open for playback. (Path relative to logger)
    virtual void setFilename(const char* path);
    virtual const char* filename();

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrStringParams* params();

    // diagnostic output
    virtual void printData() const;

private:
    // parameter
    char* myFilename;
};
```


lcQuit.h

```
#include "lcPdu.h"

// class DtLgrQuitPdu:
//
// Instances of DtLgrQuitPdu are messages telling a remote logger to quit
class DtLgrQuitPdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrQuitPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrQuitPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrQuitPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator&
receiver,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrQuitPdu();

    // copy constructor
    DtLgrQuitPdu(const DtLgrQuitPdu& orig);

    // assignment operator
    DtLgrQuitPdu& operator=(const DtLgrQuitPdu& orig);
};
```

lcRecAction.h

```
#include "lcPdu.h"

// class DtLgrRecordActionPdu:
//
// Instances of DtLgrRecordActionPdu are to tell remote logger to
// stop or begin recording when entering record-mode.

class DtLgrRecordActionPdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrRecordActionPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrRecordActionPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrRecordActionPdu(DtGlobalObjectDesignator& sender, DtGlobalObject-
Designator& receiver,
        DtLgrMode mode, DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrRecordActionPdu();

    // copy constructor
    DtLgrRecordActionPdu(const DtLgrRecordActionPdu& orig);

    // assignment operator
    DtLgrRecordActionPdu& operator=(const DtLgrRecordActionPdu& orig);

    // set/get action to take when switching to record-mode
    virtual void setAction( DtLgrMode cmd );
    virtual DtLgrMode action();

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrIntParams* params();

private:
    // parameter
    DtLgrMode myAction;
};
```

lcRecFile.h

```

#include "lcPdu.h"

// class DtLgrRecFilePdu:
//
// Instances of DtLgrRecFilePdu are used to switching remote logger to
// recording-mode and to set the file into which to store captured
// simulation traffic

class DtLgrRecFilePdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrRecFilePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrRecFilePdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrRecFilePdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver,
        char* filename, DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrRecFilePdu();

    // copy constructor
    DtLgrRecFilePdu(const DtLgrRecFilePdu& orig);

    // assignment operator
    DtLgrRecFilePdu& operator=(const DtLgrRecFilePdu& orig);

    // set/get the name of file into which traffic will be recorded
    virtual void setFilename(const char* path);
    virtual const char* filename();

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrStringParams* params();

    // diagnostic output
    virtual void printData() const;

private:
    // parameter
    char* myFilename;
};

```

lcRecord.h

```
#include "lcPdu.h"

// class DtLgrRecordPdu:
//
// Instances of DtLgrRecordPdu are to tell remote logger to begin recording
class DtLgrRecordPdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrRecordPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrRecordPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrRecordPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrRecordPdu();

    // copy constructor
    DtLgrRecordPdu(const DtLgrRecordPdu& orig);

    // assignment operator
    DtLgrRecordPdu& operator=(const DtLgrRecordPdu& orig);
};
```

lcSetTime.h

```

#include "lcPdu.h"

// class DtLgrSetTimePdu:
//
// Instances of DtLgrSetTimePdu are to tell remote logger to
// go to set position to HH:MM:SS

class DtLgrSetTimePdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrSetTimePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrSetTimePdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrSetTimePdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver,
        DtTime tm, DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrSetTimePdu();

    // copy constructor
    DtLgrSetTimePdu(const DtLgrSetTimePdu& orig);

    // assignment operator
    DtLgrSetTimePdu& operator=(const DtLgrSetTimePdu& orig);

    // set/get time to jump to within logger playback file
    virtual void setTime(const DtTime time);
    virtual DtTime time();

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrTimeParams* params();

private:
    // parameter
    DtTime myTime;
};

```

lcSkipTime.h

```
#include "lcPdu.h"

// class DtLgrSkipTimePdu:
//
// Instances of DtLgrSetTimePdu are to tell remote logger to
// forward or backward HH:MM:SS

class DtLgrSkipTimePdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrSkipTimePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrSkipTimePdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrSkipTimePdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver,
        DtTime tm, DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrSkipTimePdu();

    // copy constructor
    DtLgrSkipTimePdu(const DtLgrSkipTimePdu& orig);

    // assignment operator
    DtLgrSkipTimePdu& operator=(const DtLgrSkipTimePdu& orig);

    // set/get amount of time to skip forward or back in playback file
    virtual void setTime(DtTime dt);
    virtual DtTime time();

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrTimeParams* params();

private:
    // parameter
    DtTime myTime;
};
```

lcStart.h

```
#include "lcPdu.h"

// class DtLgrStartPdu:
//
// Instances of DtLgrStartPdu are used to tell remote logger to start
// recording

class DtLgrStartPdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrStartPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrStartPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrStartPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrStartPdu();

    // copy constructor
    DtLgrStartPdu(const DtLgrStartPdu& orig);

    // assignment operator
    DtLgrStartPdu& operator=(const DtLgrStartPdu& orig);
};
```

lcStop.h

```
#include "lcPdu.h"

// class DtLgrStopPdu:
//
// Instances of DtLgrStopPdu are to tell remote logger to stop

class DtLgrStopPdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrStopPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrStopPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrStopPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator&
receiver,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrStopPdu();

    // copy constructor
    DtLgrStopPdu(const DtLgrStopPdu& orig);

    // assignment operator
    DtLgrStopPdu& operator=(const DtLgrStopPdu& orig);
};
```


lcTimeScale.h

```

#include "lcPdu.h"

// class DtLgrTimeScalePdu:
//
// Instances of DtLgrTimeScalePdu are to tell remote logger to
// slow down or speed up playback. 1.0 is normal speed

class DtLgrTimeScalePdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrTimeScalePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrTimeScalePdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrTimeScalePdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver, double scale,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrTimeScalePdu();

    // copy constructor
    DtLgrTimeScalePdu(const DtLgrTimeScalePdu& orig);

    // assignment operator
    DtLgrTimeScalePdu& operator=(const DtLgrTimeScalePdu& orig);

    // set/get playback speed
    virtual void setTimeScale( DtReal time);
    virtual double timeScale();

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrRealParams* params();

private:
    // parameter
    DtReal myScale;
};

```

lcTimeout.h

```
#include "lcPdu.h"

// class DtLgrTimeoutPdu:
//
// Instances of DtLgrTimeoutPdu are to tell remote logger how long to
// generate
// interim packets for entity before assuming no more packets in file.
// Useful during slow-motion playback.

class DtLgrTimeoutPdu : public DtLgrControlPdu
{
public:
    // default constructor
    DtLgrTimeoutPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct from network representation
    DtLgrTimeoutPdu(const DtNetLgrControlPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // construct and prepare for send
    DtLgrTimeoutPdu(DtGlobalObjectDesignator& sender, DtGlobalObjectDesignator& receiver,
        DtReal tmount, DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // destructor
    virtual ~DtLgrTimeoutPdu();

    // copy constructor
    DtLgrTimeoutPdu(const DtLgrTimeoutPdu& orig);

    // assignment operator
    DtLgrTimeoutPdu& operator=(const DtLgrTimeoutPdu& orig);

    // set/get timeout. see class description
    virtual void setTimeout( DtReal val);
    virtual DtReal timeout();

    // return correctly casted parameters pointer; see lgrControlP.h
    virtual DtLgrRealParams* params();

private:
    // parameter
    DtReal myTimeout;
};
```

leAppearPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/leAppearP.h"

#include "leEntityID.h"

// class DtLeAppearancePdu:
//
// Instances of DtLeAppearancePdu are used to send appearance information
// for live entities.

// DtLeAppearancePdu callback function signature.
typedef void (*DtLeAppearancePduCb)(DtLeAppearancePdu* pdu, void* usr);

class DtLeAppearancePdu : public DtPdu
{
public:
    // Constructors
    DtLeAppearancePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeAppearancePdu(const DtNetLeAppearancePdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeAppearancePdu(const DtLeAppearancePdu& orig);

    // destructor
    virtual ~DtLeAppearancePdu();

    // Assignment operator
    DtLeAppearancePdu& operator=(const DtLeAppearancePdu& orig);

    // Set/Get the sub-protocol number
    virtual void setSubProtocol(DtSubProtocol value);
    virtual DtSubProtocol subProtocol() const;

    // Set/Get entity ID
    virtual void setEntityID(const DtLeEntityID& id);
    virtual const DtLeEntityID& entityID() const;

    // Get the first appearance flag field (this field is automatically
    // maintained by the PDU as other PDU fields are set)
    virtual DtU8 appearanceField1() const;

    // Get the value for a specific appearance flag from the first
    // appearance field
    virtual DtBoolean appearanceField1Flag(DtLE_AppearanceField1Flag flag)
const;

    // The following accessor functions refer to records which are optional.
    // The set methods create the record in the PDU if it does not already
    // exist. The get methods return DtFalse if the record does not exist.

    // Get the second appearance flag field (this field is automatically
    // maintained by the PDU as other PDU fields are set)
    virtual DtBoolean getAppearanceField2(DtU8& value) const;

```

```
// Get the value for a specific appearance flag from the second
// appearance field
virtual DtBoolean getAppearanceField2Flag(DtLE_AppearanceField2Flag
flag,
DtBoolean& value) const;

// Set/Get the force ID
virtual void setForceID(DtForceType force);
virtual DtBoolean getForceID(DtForceType& force) const;

// Set/Get entity type
virtual void setEntityType(const DtEntityType& type);
virtual DtBoolean getEntityType(DtEntityType& type) const;

// Set/Get alternative entity type
virtual void setAlternativeEntityType(const DtEntityType& type);
virtual DtBoolean getAlternativeEntityType(DtEntityType& type) const;

// Set/Get marking character set
virtual void setMarkingCharacterSet(DtCharacterSet value);
virtual DtBoolean getMarkingCharacterSet(DtCharacterSet& value) const;

// Set/Get marking text (copies 11 bytes)
virtual void setMarkingText(DtU8* dataPtr);
virtual DtBoolean getMarkingText(DtU8* dataPtr) const;

// Set/Get capabilities (as a 32-bit integer)
virtual void setCapabilities(DtU32 value);
virtual DtBoolean getCapabilities(DtU32& value) const;

// Set/Get a particular capability (as a Boolean)
virtual void setCapability(DtCapability flag, DtBoolean value);
virtual DtBoolean getCapability(DtCapability flag, DtBoolean& value)
const;

// Set/Get visual appearance (as a 32 bit integer)
virtual void setVisualAppearance(DtU32 value);
virtual DtBoolean getVisualAppearance(DtU32& value) const;

// Set/Get appearance by attribute
virtual void setVisualAppearanceAttribute(
DtAppearAttribute attr, DtAppearAttrVal val);
virtual DtBoolean getVisualAppearanceAttribute(
DtAppearAttribute attr, DtAppearAttrVal& val);

// Set/Get IR appearance (as a 32 bit integer)
virtual void setIR_Appearance(DtU32 value);
virtual DtBoolean getIR_Appearance(DtU32& value) const;

// Set/Get EM appearance (as a 32 bit integer)
virtual void setEM_Appearance(DtU32 value);
virtual DtBoolean getEM_Appearance(DtU32& value) const;

// Set/Get audio appearance (as a 32 bit integer)
virtual void setAudioAppearance(DtU32 value);
virtual DtBoolean getAudioAppearance(DtU32& value) const;

// Print the data contained by this PDU
virtual void printData() const;

// Print the header (override to include the new sub-protocol field)
virtual void printHeader() const;

public:

// Create a DtLeAppearPdu.
```

```

    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtLeAppearancePduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtLeAppearancePduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetLeAppearancePdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non-virtual since you can't override based on return type
    DtLeAppearancePdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

    // Set appearance field 1
    virtual void setAppearanceField1(DtU8 value);

    // Set a flag of appearance field 1
    virtual void setAppearanceField1Flag(DtLE_AppearanceField1Flag flag);

    // Clear a flag and all following flags in appearance field 1
    virtual void clearField1Flags(DtLE_AppearanceField1Flag flag);

    // Set appearance field 2
    virtual void setAppearanceField2(DtU8 value);

    // Set a flag of appearance field 2
    virtual void setAppearanceField2Flag(DtLE_AppearanceField2Flag flag);

    // Returns the offset (in bytes) of a field in the variable data region.
    virtual int appearanceField1Offset() const;
    virtual int appearanceField2Offset() const;
    virtual int forceID_Offset() const;
    virtual int entityTypeOffset() const;
    virtual int alternativeEntityTypeOffset() const;
    virtual int entityMarkingOffset() const;
    virtual int capabilitiesOffset() const;
    virtual int visualAppearanceOffset() const;
    virtual int IR_AppearanceOffset() const;
    virtual int EM_AppearanceOffset() const;
    virtual int audioAppearanceOffset() const;

protected:

    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtLeEntityID myEntityID;

};
#endif
#endif

```

leArtPartsPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"
#include "artPartList.h"
#include "attPartList.h"

#include "packets/leArtPartsP.h"

#include "leEntityID.h"

// class DtLeArticulatedPartsPdu:
//
// Instances of DtLeArticulatedPartsPdu are used to send information about
// articulated and attached parts for live entities.

// DtLeArticulatedPartsPdu callback function signature.
typedef void (*DtLeArticulatedPartsPduCb)(
    DtLeArticulatedPartsPdu* pdu, void* usr);

class DtLeArticulatedPartsPdu : public DtPdu
{
public:
    // Constructors
    DtLeArticulatedPartsPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeArticulatedPartsPdu(const DtNetLeArticulatedPartsPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeArticulatedPartsPdu(const DtLeArticulatedPartsPdu& orig);

    // destructor
    virtual ~DtLeArticulatedPartsPdu();

    // Assignment operator
    DtLeArticulatedPartsPdu& operator=(const DtLeArticulatedPartsPdu& orig);

    // Set/Get the sub-protocol number
    virtual void setSubProtocol(DtSubProtocol value);
    virtual DtSubProtocol subProtocol() const;

    // Set/Get entity ID
    virtual void setEntityID(const DtLeEntityID& id);
    virtual const DtLeEntityID& entityID() const;

    // Get the number of articulation parameter records (this field is
    // automatically set by the PDU when the articulated part list or
    attached
    // part list is set)
    virtual DtU8 numRecords() const;

    // The articulated parts and attached parts information are both
    optional.
    // Either or both can be set.

    // Set/Get articulated parts parameters. Parameters are passed in the
    form
    // of an articulated parts list. Returns DtFalse if the operation could
    // not be performed.
    virtual DtBoolean setArtPartParameters(const DtArtPartList& partList);
```

```

    virtual DtBoolean getArtPartParameters(DtArtPartList& partList) const;

    // Set/Get attached parts parameters. Parameters are passed in the form
    // of an attached parts list. Returns DtFalse if the operation could
    // not be performed.
    virtual DtBoolean setAttPartParameters(const DtAttachedPartList&
partList);
    virtual DtBoolean getAttPartParameters(DtAttachedPartList& partList)
const;

    // Print the data contained by this PDU
    virtual void printData() const;

    // Print the header (override to include the new sub-protocol field)
    virtual void printHeader() const;

public:
    // Create a DtLeArticulatedPartsPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtLeArticulatedPartsPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtLeArticulatedPartsPduCb cb, void* usr);

protected:
    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetLeArticulatedPartsPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non-virtual since you can't override based on return type
    DtLeArticulatedPartsPdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

    // Set number of records
    virtual void setNumRecords(DtU8 value);

    // Returns the offset (in bytes) of a field in the variable data region.
    virtual int numRecordsOffset() const;
    virtual int artParmOffset() const;
    virtual int attParmOffset() const;

protected:
    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:
    // Number of articulated part parameters for this PDU
    int myNumArtPartParams;

    // Number of attached part parameters for this PDU
    int myNumAttPartParams;

    // Temporary objects for reference returns

```

```
DtLeEntityID myEntityID;  
};  
#endif  
#endif
```


leDeadReck.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/leTspiP.h"

// class DtLeDeadReckoningParameters:
//
// DtLeDeadReckoningParameters is a class which contains the entity
// dead reckoning parameter information for the live entity PDUs.

class DtLeDeadReckoningParameters
{
public:
    // Constructors
    DtLeDeadReckoningParameters();
    DtLeDeadReckoningParameters(const DtLeDeadReckoningParameters& orig);

    // Assignment operator
    DtLeDeadReckoningParameters& operator=(
        const DtLeDeadReckoningParameters& orig);
    DtLeDeadReckoningParameters& operator=(
        const DtNetLeDeadReckoningParameters& orig);

    // Destructor
    virtual ~DtLeDeadReckoningParameters();

    // Set/Get dead reckoning algorithm
    virtual void setDeadReckoningAlgorithm(DtDeadReckonTypes value);
    virtual DtDeadReckonTypes deadReckoningAlgorithm() const;

    // Set/Get entity linear acceleration. This PDU field is only 24-bits
wide.
    virtual void setLinearAcceleration(DtU32 value);
    virtual DtU32 linearAcceleration() const;

    // Set/Get entity angular velocity. This PDU field is only 24-bits wide.
    virtual void setAngularVelocity(DtU32 value);
    virtual DtU32 angularVelocity() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetLeDeadReckoningParameters& record);
    virtual void loadNetRecord(
        DtNetLeDeadReckoningParameters& record) const;

protected:
    // Data members:
    DtDeadReckonTypes    myDeadReckoningAlgorithm;
    DtU32                myLinearAcceleration;
    DtU32                myAngularVelocity;
};
#endif
#endif

```

leDetonPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/leDetonP.h"

#include "leEntityID.h"
#include "leLocWorld.h"
#include "leLocEntity.h"
#include "leEntityVel.h"
#include "leEntityOri.h"

// class DtLeDetonationPdu:

// Instances of DtLeDetonationPdu are used to send firing information for
// live
// entities.

// DtLeDetonationPdu callback function signature.
typedef void (*DtLeDetonationPduCb)(DtLeDetonationPdu* pdu, void* usr);

class DtLeDetonationPdu : public DtPdu
{
public:
    // Constructors
    DtLeDetonationPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeDetonationPdu(const DtNetLeDetonationPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeDetonationPdu(const DtLeDetonationPdu& orig);

    // destructor
    virtual ~DtLeDetonationPdu();

    // Assignment operator
    DtLeDetonationPdu& operator=(const DtLeDetonationPdu& orig);

    // Set/Get the sub-protocol number
    virtual void setSubProtocol(DtSubProtocol value);
    virtual DtSubProtocol subProtocol() const;

    // Set/Get firing entity ID
    virtual void setFiringEntityID(const DtLeEntityID& id);
    virtual const DtLeEntityID& firingEntityID() const;

    // Get flags field 1 (this field is automatically maintained by the
    // PDU as other PDU fields are set)
    virtual DtU8 detonFlagsField() const;

    // Get flags field 2 (this field is automatically maintained by the
    // PDU as other PDU fields are set). Returns DtFalse if this field is
    // not present.
    virtual DtBoolean getDeton2FlagsField(DtU8& field) const;

    // Get the value for a specific flag from the first flags field
    virtual DtBoolean detonFlag(DtLE_DetonFieldFlag flag) const;
```

```

// Get the value for a specific flag from the second flags field.
// Returns DtFalse if the field is not present in the PDU.
virtual DtBoolean getDeton2Flag(
    DtLE_DetonField2Flag flag, DtBoolean& value) const;

// Some of following accessor functions refer to records which are
optional.
// The set methods create the record in the PDU if it does not already
// exist. The get methods return DtFalse if the record does not exist.

// Set/Get target entity ID (optional)
virtual void setTargetEntityID(const DtLeEntityID& id);
virtual DtBoolean getTargetEntityID(DtLeEntityID& id) const;

// Set/Get munition ID site (optional)
virtual void setMunitionID_Site(DtU8 value);
virtual DtBoolean getMunitionID_Site(DtU8& value) const;

// Set/Get munition ID application (optional)
virtual void setMunitionID_Application(DtU8 value);
virtual DtBoolean getMunitionID_Application(DtU8& value) const;

// Set/Get munition ID entity (optional)
virtual void setMunitionID_Entity(DtU16 value);
virtual DtBoolean getMunitionID_Entity(DtU16& value) const;

// Set/Get event ID site (optional)
virtual void setEventID_Site(DtU8 value);
virtual DtBoolean getEventID_Site(DtU8& value) const;

// Set/Get event ID application (optional)
virtual void setEventID_Application(DtU8 value);
virtual DtBoolean getEventID_Application(DtU8& value) const;

// Set/Get event ID event
virtual void setEventID_Event(DtU16 value);
virtual DtBoolean getEventID_Event(DtU16& value) const;

// Set/Get entity location (world coordinates). In this PDU, entity
// location is stored in world coordinates or entity coordinates but not
// both. Get returns DtFalse if the coordinates are stored in entity
// coordinates.
virtual void setLocationWorld(
    const DtLeEntityLocationWorldCoordinates& value);
virtual DtBoolean getLocationWorld(
    DtLeEntityLocationWorldCoordinates& value) const;

// Set/Get entity location (entity coordinates). In this PDU, entity
// location is stored in world coordinates or entity coordinates but not
// both. Get returns DtFalse if the coordinates are stored in world
// coordinates.
virtual void setLocationEntity(
    const DtLeEntityLocationEntityCoordinates& value);
virtual DtBoolean getLocationEntity(
    DtLeEntityLocationEntityCoordinates& value) const;

// Set/Get entity linear velocity
virtual void setVelocity(const DtLeEntityLinearVelocity& value);
virtual const DtLeEntityLinearVelocity& velocity() const;

// Set/Get entity orientation
virtual void setOrientation(const DtLeEntityOrientation& value);
virtual DtBoolean getOrientation(DtLeEntityOrientation& value) const;

// Set/Get burst munition type
virtual void setBurstMunitionType(DtEntityType& type);
virtual const DtEntityType& burstMunitionType() const;

```

```
// Set/Get burst warhead (optional)
virtual void setBurstWarhead(DtU16 value);
virtual DtBoolean getBurstWarhead(DtU16& value) const;

// Set/Get burst fuse (optional)
virtual void setBurstFuse(DtU16 value);
virtual DtBoolean getBurstFuse(DtU16& value) const;

// Set/Get burst quantity (optional)
virtual void setBurstQuantity(DtU16 value);
virtual DtBoolean getBurstQuantity(DtU16& value) const;

// Set/Get burst rate (optional)
virtual void setBurstRate(DtU16 value);
virtual DtBoolean getBurstRate(DtU16& value) const;

// Set/Get detonation result
virtual void setDetonationResult(DtU8 value);
virtual DtU8 detonationResult() const;

// Print the data contained by this PDU
virtual void printData() const;

// Print the header (override to include the new sub-protocol field)
virtual void printHeader() const;

public:

    // Create a DtLeDetonationPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtLeDetonationPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtLeDetonationPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetLeDetonationPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non-virtual since you can't override based on return type
    DtLeDetonationPdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

    // Set first detonation flags field
    virtual void setDetonFlagsField(DtU8 value);

    // Set second detonation flags field
    virtual void setDeton2FlagsField(DtU8 value);

    // Set first detonation flag
    virtual void setDetonFlag(DtLE_DetonFieldFlag flag);

    // Set second detonation flag
    virtual void setDeton2Flag(DtLE_DetonField2Flag flag);
```

```

// Returns the offset (in bytes) of a field from variable data region.
// Some fields have subfields which are optional, so there are multiple
virtual int detonFlagFieldOffset() const;
virtual int deton2FlagFieldOffset() const;
virtual int targetEntityID_Offset() const;
virtual int munitionID_Offset1() const;
virtual int munitionID_Offset2() const;
virtual int eventID_Offset1() const;
virtual int eventID_Offset2() const;
virtual int locationWorldOffset() const;
virtual int velocityOffset() const;
virtual int munitionOrientationOffset() const;
virtual int burstOffset1() const;
virtual int burstOffset2() const;
virtual int burstOffset3() const;
virtual int locationEntityOffset() const;
virtual int resultOffset() const;

protected:

    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtLeEntityID myEntityID;
    DtLeEntityLinearVelocity myVelocity;
    DtEntityType myEntityType;
};
#endif
#endif

```

leEntityID.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/leTspiP.h"

// class DtLeEntityID:
//
// DtLeEntityID is a class which contains the entity identification
// information for the live entity PDUs.

class DtLeEntityID
{
public:
    // Constructors
    DtLeEntityID();
    DtLeEntityID(const DtLeEntityID& orig);

    // Assignment operator
    DtLeEntityID& operator=(const DtLeEntityID& orig);
    DtLeEntityID& operator=(const DtNetLeEntityID& orig);

    // Destructor
    virtual ~DtLeEntityID();

    // Set/Get site
    virtual void setSite(DtU8 value);
    virtual DtU8 site() const;

    // Set/Get application
    virtual void setApplication(DtU8 value);
    virtual DtU8 application() const;

    // Set/Get entity
    virtual void setEntity(DtU16 value);
    virtual DtU16 entity() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(const DtNetLeEntityID& record);
    virtual void loadNetRecord(DtNetLeEntityID& record) const;

protected:
    // Data members:
    DtU8 mySite;
    DtU8 myApplication;
    DtU16 myEntity;
};
#endif
#endif
```

leEntityOri.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/leTspiP.h"

// class DtLeEntityOrientation:
//
// DtLeEntityOrientation is a class which contains the entity
// orientation information for the live entity PDUs. This record uses
// 8-bit values for the TSPI PDU and 16-bit values for the detonation
// PDU.

class DtLeEntityOrientation
{
public:
    // Constructors
    DtLeEntityOrientation();
    DtLeEntityOrientation(const DtLeEntityOrientation& orig);

    // Assignment operator
    DtLeEntityOrientation& operator=(
        const DtLeEntityOrientation& orig);
    DtLeEntityOrientation& operator=(
        const DtNetLeEntityOrientation& orig);

    // Destructor
    virtual ~DtLeEntityOrientation();

    // Set/Get Psi
    virtual void setPsi(DtU16 value);
    virtual DtU16 psi() const;

    // Set/Get Theta
    virtual void setTheta(DtU16 value);
    virtual DtU16 theta() const;

    // Set/Get Phi
    virtual void setPhi(DtU16 value);
    virtual DtU16 phi() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record (for 8-bit fields)
    virtual void unloadNetRecord(
        const DtNetLeEntityOrientation& record);
    virtual void loadNetRecord(
        DtNetLeEntityOrientation& record) const;

    // Load/Unload data from a packet record (for 16-bit fields)
    virtual void unloadNetRecord16(
        const DtNetLeEntityOrientation16& record);
    virtual void loadNetRecord16(
        DtNetLeEntityOrientation16& record) const;

protected:
    // Data members:

```

```
DtU16          myPsi;  
DtU16          myTheta;  
DtU16          myPhi;  
};  
#endif  
#endif
```


leEntityVel.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/leTspiP.h"

// class DtLeEntityLinearVelocity:
//
// DtLeEntityLinearVelocity is a class which contains the entity linear
// velocity information for the live entity PDUs.

class DtLeEntityLinearVelocity
{
public:
    // Constructors
    DtLeEntityLinearVelocity();
    DtLeEntityLinearVelocity(const DtLeEntityLinearVelocity& orig);

    // Assignment operator
    DtLeEntityLinearVelocity& operator=(
        const DtLeEntityLinearVelocity& orig);
    DtLeEntityLinearVelocity& operator=(
        const DtNetLeEntityLinearVelocity& orig);

    // Destructor
    virtual ~DtLeEntityLinearVelocity();

    // Set/Get X
    virtual void setX(DtU16 value);
    virtual DtU16 x() const;

    // Set/Get Y
    virtual void setY(DtU16 value);
    virtual DtU16 y() const;

    // Set/Get Z
    virtual void setZ(DtU16 value);
    virtual DtU16 z() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetLeEntityLinearVelocity& record);
    virtual void loadNetRecord(
        DtNetLeEntityLinearVelocity& record) const;

protected:
    // Data members:
    DtU16 myX;
    DtU16 myY;
    DtU16 myZ;

};
#endif
#endif

```

leFirePdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/leFireP.h"

#include "leDeadReck.h"
#include "leEntityID.h"
#include "leLocWorld.h"
#include "leEntityVel.h"

// class DtLeFirePdu:

// Instances of DtLeFirePdu are used to send firing information for live
// entities.

// DtLeFirePdu callback function signature.
typedef void (*DtLeFirePduCb)(DtLeFirePdu* pdu, void* usr);

class DtLeFirePdu : public DtPdu
{
public:
    // Constructors
    DtLeFirePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeFirePdu(const DtNetLeFirePdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeFirePdu(const DtLeFirePdu& orig);

    // destructor
    virtual ~DtLeFirePdu();

    // Assignment operator
    DtLeFirePdu& operator=(const DtLeFirePdu& orig);

    // Set/Get the sub-protocol number
    virtual void setSubProtocol(DtSubProtocol value);
    virtual DtSubProtocol subProtocol() const;

    // Set/Get firing entity ID
    virtual void setFiringEntityID(const DtLeEntityID& id);
    virtual const DtLeEntityID& firingEntityID() const;

    // Get flags field (this field is automatically maintained by the
    // PDU as other PDU fields are set)
    virtual DtU8 fireFlagsField() const;

    // Get the value for a specific flag from the flags field
    virtual DtBoolean fireFlag(DtLE_FireFieldFlag flag) const;

    // Some of following accessor functions refer to records which are
    // optional.
    // The set methods create the record in the PDU if it does not already
    // exist. The get methods return DtFalse if the record does not exist.

    // Set/Get target entity ID (optional)
    virtual void setTargetEntityID(const DtLeEntityID& id);
```

```

virtual DtBoolean getTargetEntityID(DtLeEntityID& id) const;

// Set/Get munition ID site (optional)
virtual void setMunitionID_Site(DtU8 value);
virtual DtBoolean getMunitionID_Site(DtU8& value) const;

// Set/Get munition ID application (optional)
virtual void setMunitionID_Application(DtU8 value);
virtual DtBoolean getMunitionID_Application(DtU8& value) const;

// Set/Get munition ID entity (optional)
virtual void setMunitionID_Entity(DtU16 value);
virtual DtBoolean getMunitionID_Entity(DtU16& value) const;

// Set/Get event ID site (optional)
virtual void setEventID_Site(DtU8 value);
virtual DtBoolean getEventID_Site(DtU8& value) const;

// Set/Get event ID application (optional)
virtual void setEventID_Application(DtU8 value);
virtual DtBoolean getEventID_Application(DtU8& value) const;

// Set/Get event ID event
virtual void setEventID_Event(DtU16 value);
virtual DtU16 eventID_Event() const;

// Set/Get entity location (optional)
virtual void setLocation(const DtLeEntityLocationWorldCoordinates&
value);
virtual DtBoolean getLocation(
    DtLeEntityLocationWorldCoordinates& value) const;

// Set/Get burst munition type
virtual void setBurstMunitionType(DtEntityType& type);
virtual const DtEntityType& burstMunitionType() const;

// Set/Get burst warhead (optional)
virtual void setBurstWarhead(DtU16 value);
virtual DtBoolean getBurstWarhead(DtU16& value) const;

// Set/Get burst fuse (optional)
virtual void setBurstFuse(DtU16 value);
virtual DtBoolean getBurstFuse(DtU16& value) const;

// Set/Get burst quantity (optional)
virtual void setBurstQuantity(DtU16 value);
virtual DtBoolean getBurstQuantity(DtU16& value) const;

// Set/Get burst rate (optional)
virtual void setBurstRate(DtU16 value);
virtual DtBoolean getBurstRate(DtU16& value) const;

// Set/Get entity linear velocity
virtual void setVelocity(const DtLeEntityLinearVelocity& value);
virtual const DtLeEntityLinearVelocity& velocity() const;

// Set/Get range
virtual void setRange(DtU16 value);
virtual DtU16 range() const;

// Print the data contained by this PDU
virtual void printData() const;

// Print the header (override to include the new sub-protocol field)
virtual void printHeader() const;

public:

```

```
// Create a DtLeFirePdu.
// Caller is responsible for deletion.
static DtPdu* create(const DtNetPacket* initial,
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtLeFirePduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtLeFirePduCb cb, void* usr);

protected:

// Returns the internal PDU kind
virtual DtPduKind internalGetPduKind() const;

// Casts the netPdu element to our network struct
virtual DtNetLeFirePdu* netPduStruct() const;

// Casts the 'this' pointer for changing elements in a const function
// Non-virtual since you can't override based on return type
DtLeFirePdu* self() const;

// Returns a pointer to the first byte of variable length data
virtual char* var() const;

// Set Fire Flags
virtual void setFireFlagsField(DtU8 value);

// Set a flag of the flag field
virtual void setFireFlag(DtLE_FireFieldFlag flag);

// Clear the flag and all following flags in the fire flags field
virtual void clearFireFlags(DtLE_FireFieldFlag flag);

// Returns the offset (in bytes) of a field from variable data region.
// Some fields have subfields which are optional, so there are multiple
virtual int fireFlagFieldOffset() const;
virtual int targetEntityID_Offset() const;
virtual int munitionID_Offset1() const;
virtual int munitionID_Offset2() const;
virtual int eventID_Offset1() const;
virtual int eventID_Offset2() const;
virtual int locationOffset() const;
virtual int burstOffset1() const;
virtual int burstOffset2() const;
virtual int burstOffset3() const;
virtual int velocityOffset() const;
virtual int rangeOffset() const;

protected:

// Size of PDU with zero parameter records.
static int theStaticSize;

protected:

// Temporary objects for reference returns
DtLeEntityID myEntityID;
DtLeEntityLinearVelocity myVelocity;
DtEntityType myEntityType;
};
#endif
#endif
```

leLocEntity.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/leTspiP.h"

// class DtLeEntityLocationEntityCoordinates:
//
// DtLeEntityLocationEntityCoordinates is a class which contains the entity
// location information (in entity coordinates) for the live entity PDUs.

class DtLeEntityLocationEntityCoordinates
{
public:
    // Constructors
    DtLeEntityLocationEntityCoordinates();
    DtLeEntityLocationEntityCoordinates(
        const DtLeEntityLocationEntityCoordinates& orig);

    // Assignment operator
    DtLeEntityLocationEntityCoordinates& operator=(
        const DtLeEntityLocationEntityCoordinates& orig);
    DtLeEntityLocationEntityCoordinates& operator=(
        const DtNetLeEntityLocationEntityCoordinates& orig);

    // Destructor
    virtual ~DtLeEntityLocationEntityCoordinates();

    // Set/Get X
    virtual void setX(DtU16 value);
    virtual DtU16 x() const;

    // Set/Get Y
    virtual void setY(DtU16 value);
    virtual DtU16 y() const;

    // Set/Get Z
    virtual void setZ(DtU16 value);
    virtual DtU16 z() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetLeEntityLocationEntityCoordinates& record);
    virtual void loadNetRecord(
        DtNetLeEntityLocationEntityCoordinates& record) const;

protected:
    // Data members:
    DtU16 myX;
    DtU16 myY;
    DtU16 myZ;
};
#endif
#endif

```

leLocWorld.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/leTspiP.h"

// class DtLeEntityLocationWorldCoordinates:
//
// DtLeEntityLocationWorldCoordinates is a class which contains the entity
// location information for the live entity PDUs.

class DtLeEntityLocationWorldCoordinates
{
public:
    // Constructors
    DtLeEntityLocationWorldCoordinates();
    DtLeEntityLocationWorldCoordinates(
        const DtLeEntityLocationWorldCoordinates& orig);

    // Assignment operator
    DtLeEntityLocationWorldCoordinates& operator=(
        const DtLeEntityLocationWorldCoordinates& orig);
    DtLeEntityLocationWorldCoordinates& operator=(
        const DtNetLeEntityLocationWorldCoordinates& orig);

    // Destructor
    virtual ~DtLeEntityLocationWorldCoordinates();

    // Set/Get the reference point
    virtual void setReferencePoint(DtU16 value);
    virtual DtU16 referencePoint() const;

    // Set/Get delta X
    virtual void setDeltaX(DtU16 value);
    virtual DtU16 deltaX() const;

    // Set/Get delta Y
    virtual void setDeltaY(DtU16 value);
    virtual DtU16 deltaY() const;

    // Set/Get delta Z
    virtual void setDeltaZ(DtU16 value);
    virtual DtU16 deltaZ() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetLeEntityLocationWorldCoordinates& record);
    virtual void loadNetRecord(
        DtNetLeEntityLocationWorldCoordinates& record) const;

protected:
    // Data members:
    DtU16 myReferencePoint;
    DtU16 myDeltaX;
    DtU16 myDeltaY;
```

```
    DtU16                myDeltaZ;  
};  
#endif  
#endif
```

leOriError.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/leTspiP.h"

// class DtLeOrientationError:
//
// DtLeOrientationError is a class which contains the entity
// orientation error information for the live entity PDUs.

class DtLeOrientationError
{
public:
    // Constructors
    DtLeOrientationError();
    DtLeOrientationError(const DtLeOrientationError& orig);

    // Assignment operator
    DtLeOrientationError& operator=(
        const DtLeOrientationError& orig);
    DtLeOrientationError& operator=(
        const DtNetLeOrientationError& orig);

    // Destructor
    virtual ~DtLeOrientationError();

    // Set/Get azimuth error
    virtual void setAzimuthError(DtU16 value);
    virtual DtU16 azimuthError() const;

    // Set/Get elevation error
    virtual void setElevationError(DtU16 value);
    virtual DtU16 elevationError() const;

    // Set/Get rotation error
    virtual void setRotationError(DtU16 value);
    virtual DtU16 rotationError() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetLeOrientationError& record);
    virtual void loadNetRecord(
        DtNetLeOrientationError& record) const;

protected:
    // Data members:
    DtU16 myAzimuthError;
    DtU16 myElevationError;
    DtU16 myRotationError;
};
#endif
#endif
```


lePosError.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/leTspiP.h"

// class DtLePositionError:
//
// DtLePositionError is a class which contains the entity location
// information for the live entity PDUs.

class DtLePositionError
{
public:
    // Constructors
    DtLePositionError();
    DtLePositionError(const DtLePositionError& orig);

    // Assignment operator
    DtLePositionError& operator=(
        const DtLePositionError& orig);
    DtLePositionError& operator=(
        const DtNetLePositionError& orig);

    // Destructor
    virtual ~DtLePositionError();

    // Set/Get horizontal error
    virtual void setHorizontalError(DtU16 value);
    virtual DtU16 horizontalError() const;

    // Set/Get vertical error
    virtual void setVerticalError(DtU16 value);
    virtual DtU16 verticalError() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetLePositionError& record);
    virtual void loadNetRecord(
        DtNetLePositionError& record) const;

protected:
    // Data members:
    DtU16 myHorizontalError;
    DtU16 myVerticalError;
};
#endif
#endif

```

leTspiPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/leTspiP.h"

#include "leDeadReck.h"
#include "leEntityID.h"
#include "leLocWorld.h"
#include "leEntityOri.h"
#include "leEntityVel.h"
#include "leOriError.h"
#include "lePosError.h"

// class DtLeTspiPdu:
//
// Instances of DtLeTspiPdu are used to send state information for live
// entities.

// DtLeTspiPdu callback function signature.
typedef void (*DtLeTspiPduCb)(DtLeTspiPdu* pdu, void* usr);

class DtLeTspiPdu : public DtPdu
{
public:
    // Constructors
    DtLeTspiPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeTspiPdu(const DtNetLeTspiPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLeTspiPdu(const DtLeTspiPdu& orig);

    // destructor
    virtual ~DtLeTspiPdu();

    // Assignment operator
    DtLeTspiPdu& operator=(const DtLeTspiPdu& orig);

    // Set/Get the sub-protocol number
    virtual void setSubProtocol(DtSubProtocol value);
    virtual DtSubProtocol subProtocol() const;

    // Set/Get entity ID
    virtual void setEntityID(const DtLeEntityID& id);
    virtual const DtLeEntityID& entityID() const;

    // Get TSPI Flag field (this field is automatically maintained by the
    // PDU as other PDU fields are set)
    virtual DtU8 tspiField() const;

    // Get the value for a specific TSPI flag from the flag field
    virtual DtBoolean tspiFlag(DtLE_TSPI_FieldFlag flag) const;

    // Set/Get entity location
    virtual void setLocation(const DtLeEntityLocationWorldCoordinates&
value);
    virtual const DtLeEntityLocationWorldCoordinates& location() const;
```

```

// The following accessor functions refer to records which are optional.
// The set methods create the record in the PDU if it does not already
// exist. The get methods return DtFalse if the record does not exist.

// Set/Get entity linear velocity
virtual void setVelocity(const DtLeEntityLinearVelocity& value);
virtual DtBoolean getVelocity(DtLeEntityLinearVelocity& value) const;

// Set/Get entity orientation
virtual void setOrientation(const DtLeEntityOrientation& value);
virtual DtBoolean getOrientation(DtLeEntityOrientation& value) const;

// Set/Get position error
virtual void setPositionError(const DtLePositionError& value);
virtual DtBoolean getPositionError(DtLePositionError& value) const;

// Set/Get orientation error
virtual void setOrientationError(const DtLeOrientationError& value);
virtual DtBoolean getOrientationError(DtLeOrientationError& value)
const;

// Set/Get dead reckoning parameters
virtual void setDeadReckoningParameters(
    const DtLeDeadReckoningParameters& value);
virtual DtBoolean getDeadReckoningParameters(
    DtLeDeadReckoningParameters& value) const;

// Set/Get measured speed
virtual void setMeasuredSpeed(DtU16 value);
virtual DtBoolean getMeasuredSpeed(DtU16& value) const;

// Set/Get the system specific data length
virtual DtBoolean setSystemSpecificDataLength(DtU8 value);
virtual DtU8 systemSpecificDataLength() const;

// Set/Get the system specific data (will copy systemSpecificDataLength()
// bytes on set/get). Returns DtFalse if the length is zero.
virtual DtBoolean setSystemSpecificData(DtU8* dataPtr);
virtual DtBoolean getSystemSpecificData(DtU8* dataPtr);

// Print the data contained by this PDU
virtual void printData() const;

// Print the header (override to include the new sub-protocol field)
virtual void printHeader() const;

public:
    // Create a DtLeTspiPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtLeTspiPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtLeTspiPduCb cb, void* usr);

protected:
    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct

```

```
virtual DtNetLeTspiPdu* netPduStruct() const;

// Casts the 'this' pointer for changing elements in a const function
// Non-virtual since you can't override based on return type
DtLeTspiPdu* self() const;

// Returns a pointer to the first byte of variable length data
virtual char* var() const;

// Set TSPI Flags
virtual void setTspiField(DtU8 value);

// Set a flag of the TSPI field
virtual void setTspiFlag(DtLE_TSPI_FieldFlag flag);

// Clear the flag and all following flags in the TSPI field
virtual void clearTspiFlags(DtLE_TSPI_FieldFlag flag);

// Returns the offset (in bytes) of a field from variable data region.
virtual int tspiFlagOffset() const;
virtual int locationOffset() const;
virtual int velocityOffset() const;
virtual int orientationOffset() const;
virtual int positionErrorOffset() const;
virtual int orientationErrorOffset() const;
virtual int deadReckoningParametersOffset() const;
virtual int measuredSpeedOffset() const;
virtual int systemSpecificOffset() const;

protected:

    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtLeEntityID myEntityID;
    DtLeEntityLocationWorldCoordinates myLocation;
};
#endif
#endif
```

LibMatrix.h

```

#include <math.h>

#include "MachineTypes.h"
#include "mathTypes.h"

#define DtRad2Deg(x)      (((x) / M_PI) * 180.0)
#define DtDeg2Rad(x)      (((x) * M_PI) / 180.0)

/* Returns the absolute value of x.      */
#define DtAbs(x)          (((x) >= 0) ? (x) : -(x))

/* Returns the sign of x.      */
#define DtSgn(x)          (((x) >= 0) ? 1 : -1)

/* Returns x if greater than zero.      */
#define DtPos(x)          (((x) >= 0) ? (x) : 0)

/* Returns x if x is less than zero.      */
#define DtNeg(x)          (((x) >= 0) ? 0 : (x))

/* Returns the square of x.      */
#define DtSquare(x)       ((x) * (x))

/* A selection of constants useful as tolerance and threshold levels, etc. */
#define DtEPico            1.0E-12
#define DtENano            1.0E-09
#define DtEMicro          1.0E-06
#define DtEMilli          1.0E-03
#define DtECenti          1.0E-02
#define DtEDeci           1.0E-01
#define DtEHalf           5.0E-01
#define DtENineTenths     9.0E-01
#define DtEOne            1.0
#define DtEDeca           1.0E01
#define DtEHecto          1.0E02
#define DtEKilo           1.0E03
#define DtEMega           1.0E06
#define DtEGiga           1.0E09

/* A macro to make use of these tolerances. */
#define DtEq(x, y, e)      ((DtAbs((x) - (y)) <= (e)) ? DtTrue : DtFalse)

/*
 * Vector routines.
 */

#define DtSetVec(v, x, y, z) v[DtX]=x;v[DtY]=y;v[DtZ]=z;

voidDtVecInit(DtVectorRef v);/* Initialize to all zeros. */
voidDtVecCopy(DtConstVector from, DtVectorRef to);
voidDtVecNormalize(DtConstVector v, DtVectorRef result);
intDtVecCheck(DtConstVector v);/* Is this vector normalized? */
voidDtVecScale(DtConstVector v, DtReal scale_factor, DtVectorRef result);
voidDtVecNeg(DtConstVector v1, DtVectorRef result); /* result = -vec */
voidDtVecAdd(DtConstVector v1, DtConstVector v2, DtVectorRef result);
voidDtVecSub(DtConstVector v1, DtConstVector v2, DtVectorRef result); /*
result = v1 - v2 */
DtRealDtVecCosProd(DtConstVector v1, DtConstVector v2); /* cos of angle
between vecs */
DtRealDtVecDotProd(DtConstVector v1, DtConstVector v2);
voidDtVecDump(const char *str, DtConstVector v);

```

```
voidDtVecCrossProd(DtConstVector v1, DtConstVector v2, DtVectorRef result);
/* sweep v1 to v2 */
DtReal DtVecBoxProd(DtConstVector v1, DtConstVector v2, DtConstVector v3);
/* sweep v1 to v2 to v3*/

/*
 * Direction Cosine Matrix routines.
 */

voidDtDcmInit(DtDcmRef m);/* Initialize to all zeros. */
voidDtDcmIdentInit(DtDcmRef m);
intDtDcmCheck(DtConstDcm m);/* Is this matrix orthonormal? */
voidDtDcmFixMatrix(DtDcmRef mat, int axis); /* make orthonormal, pick most
important. axis */
voidDtDcmCopy(DtConstDcm src, DtDcmRef dest);
DtRealDtDcmDeterminant(DtConstDcm m);
voidDtDcmAdd(DtConstDcm m1, DtConstDcm m2, DtDcmRef result); /* result = m1
+ m2 */
voidDtDcmSub(DtConstDcm m1, DtConstDcm m2, DtDcmRef result); /* result = m1
- m2 */
voidDtDcmDcmMul(DtConstDcm m1, DtConstDcm m2, DtDcmRef result); /* result =
m1 x m2 */
voidDtDcmTranspose(DtConstDcm m, DtDcmRef result);
voidDtDcmAdjugate(DtConstDcm m, DtDcmRef result);
voidDtDcmInverse(DtConstDcm m, DtDcmRef result);
voidDtDcmScale(DtConstDcm mat, DtReal scale_factor, DtDcmRef result);
voidDtDcmDump(const char *str, DtConstDcm mat);

/* Rotation is counterclockwise when viewed a positive axis. */
voidDtDcmRotInit(DtDcmRef m, DtReal theta, int rot_axis);
voidDtDcmRotInit2(DtDcmRef m, DtReal sin_theta, int rot_axis); /* $$$ only
pos rots? */
voidDtDcmTrigInit(DtDcmRef m, DtReal sin_theta, DtReal cos_theta, int
rot_axis);
voidDtDcmForm(DtReal *axis, DtReal angle, DtDcmRef C, DtReal *sin_ang,
DtReal *cos_ang);
voidDtDcmLevelInit (DtDcmRef m, DtConstVector v); /* y axis given, x axis
parallel to ground. */

/*
 * Combined vector/matrix routines.
 */

voidDtDcmVecMul(DtConstDcm m, DtConstVector v, DtVectorRef result);
voidDtVecDcmMul(DtConstVector v, DtConstDcm m, DtVectorRef result);

/*
 * Euler parameter routines.
 */

typedef DtReal DtEParm[4];

extern voidDtElrCopy (DtEParm from, DtEParm to);
extern voidDtElrElrCat (DtEParm A_e_B0, DtEParm B0_e_B, DtEParm A_e_B);
extern voidDtElrForm (DtEParm axis, DtReal angle, DtEParm result);
extern voidDtElrIdent (DtEParm e);
extern voidDtElrToMat (DtEParm A_e_B, DtDcmRef A_c_B);
extern voidDtElrTranspose (DtEParm e, DtEParm result);
extern voidDtDcmToElr (DtConstDcm C, DtEParm e);
extern voidDtVecElrTransform (DtEParm v_A, DtEParm A_to_B, DtEParm v_B);

/*
 * Quaternion routines.
 */

typedef struct
{
```

```

    DtReal DtQ0;
    DtReal DtQ1;
    DtReal DtQ2;
    DtReal DtQ3;
} DtQuaternion;

voidDtQuatQuatMul(DtQuaternion *qresult, DtQuaternion *DtQ1, DtQuaternion
 *DtQ2);
voidDtDcmFromQuaternion(DtDcmRef m, DtQuaternion *q);
/* voidDtQuatScale(DtQuaternion *qresult, DtQuaternion *q, DtReal delta_t);
 */
voidDtQuaternionDump(DtQuaternion *q);

voidDtQuaternionFromTaitBryan(DtQuaternion *q, const DtTaitBryan &tb);
voidDtBodyToMat(DtBodyRates av, DtConstDcm HcW, DtDcmRef cDot);
voidDtTbToMat(const DtTaitBryan &tb, DtDcmRef HcW);

/*
 * Periodic modulus
 */

/* For x with period ABS(incl-excl),
 * return DtModPerLo(x, incl, excl) if incl < excl
 * return DtModPerHi(x, excl, incl) if incl > excl */
double DtModPer(double x, double incl, double excl);

/* For x with period per, return 0 <= x < per */
double DtModPerZero(double x, double per);

/* For x with period (max-min), return min <= x < max */
double DtModPerLo(double x, double min, double max);

/* For x with period (max-min), return min < x <= max */
double DtModPerHi(double x, double min, double max);

/* directed rotation rates */

DtDirectedRotRate DtDirectRotRate(DtConstVector pqr);
void DtMakeRotationDcm(DtDirectedRotRate drr, double dt, DtDcmRef m);

/* Returns the trace of the rotation matrix, R, between between
   matrices A and B. If A is a DCM between frame body and world, and
   B is that same frame rotated through angle theta about an arbitrary
   axis, then trace of R [where B = A * R, or R = inv(A) * B] is
   2 * cos(theta) + 1. */
double DtDcmDcmRotTrace(DtConstDcm a, DtConstDcm b);

double DtDcmDcmRotAngle(DtConstDcm a, DtConstDcm b);

void DtDcmDcmRotAxis(
    DtConstDcm a2ref, DtConstDcm b2ref, DtVectorRef n /* a2b */);

/* Generic DcmDcm Multiply routine */

enum generic_mm_multflag
{
    TRANSPPOSE_NONE = 0,
    TRANSPPOSE_M1 = 1,
    TRANSPPOSE_M2 = 2,
    TRANSPPOSE_M1_AND_M2 = 3
};
DtReal DtMM_Mult(DtConstDcm m1, DtConstDcm m2, DtDcmRef
result, generic_mm_multflag flag=TRANSPPOSE_NONE);

/* Generic Vector Sum :Ans=A+s*B*/
#define ADD 1.0
#define SUBTRACT -1.0

```

```
void DtSum(DtConstVector a,DtConstVector b,DtReal s,DtVectorRef ans);

/* generic initialize matrix routines */
#define ZERO 0
#define IDENTITY 1;
void DtM_Init(DtConstDcm input,DtDcmRef result,int flag);

void DtSetSym(DtDcmRef v,DtReal aa,DtReal ab,DtReal ac,DtReal bb,DtReal
bc,DtReal cc);

void DtVV_Product(DtConstVector a,DtConstVector b, DtDcmRef ans);

void DtCreateCross(DtConstVector r,DtDcmRef ans);

void DtM(DtConstVector v,DtDcmRef m);

void DtRotate(DtDcmRef ans,DtConstVector direction,double theta);

double DtVecInfNorm(DtConstVector v1);

/* Unobvious Constants */
typedef DtReal DtThreeIndex[3][3][3];
typedef DtReal (*DtThreeIndexPtr)[3][3];

extern DtThreeIndexDtLevi_Civita;

/* additional Routines */
double DtDistance(DtConstVector p1, DtConstVector p2); /* ||p1-p2||_{2} */
double DtDistSqr(DtConstVector p1, DtConstVector p2);

double DtMax(double x,double y);
double DtMin(double x,double y);
double DtLimit(double x,double hi ,double lo);

class DtCoordTranslation
{
public:
    DtCoordTranslation();
    DtCoordTranslation(DtConstVector toOriginInFromCoord);

    void setOffset(DtConstVector toOriginInFromCoord);
    void setByInverse(DtCoordTranslation &to2from);

    void getOffset(DtVectorRef toOriginInFromCoord);
    void coordTrans(DtConstVector from, DtVectorRef to);

private:
    DtVector offset; /* "to" origin in base coords */
};

class DtCoordTransform
{
public:
    DtCoordTransform();
    DtCoordTransform(DtConstVector toOriginInFromCoord, DtConstDcm from2to);
    void setOffset(DtConstVector toOriginInFromCoord);
    void setRotation(DtConstDcm from2to);
    void setByInverse(const DtCoordTransform &to2from);

    void getOffset(DtVectorRef toOriginInFromCoord) const;
    void getOrientation(DtDcmRef from_to) const;

    void coordTrans(DtConstVector from, DtVectorRef to) const;
    void vecTrans(DtConstVector from, DtVectorRef to) const;
    void eulerTrans(const DtTaitBryan &from, DtTaitBryan *to) const;

    /* these were protected, but it is often more efficient to set these
    by passing them to another function */
};
```



```
    DtVector offset; /* "to" origin in base coords */  
    DtDcm    rotation;  
};
```

linObjStPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/linObjStP.h"

#include "linSegParam.h"
#include "objectType.h"

// class DtLinearObjectStatePdu:
//
// Instances of DtLinearObjectStatePdu are used to communicate information
// about the addition/modification of a synthetic object which is geometri-
// cally
// anchored to the terrain with one point.

// DtLinearObjectStatePdu callback function signature.
typedef void (*DtLinearObjectStatePduCb)(
    DtLinearObjectStatePdu* pdu, void* usr);

class DtLinearObjectStatePdu : public DtPdu
{
public:
    // Constructors
    DtLinearObjectStatePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLinearObjectStatePdu(const DtNetLinearObjectStatePdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtLinearObjectStatePdu(const DtLinearObjectStatePdu& orig);

    // destructor
    virtual ~DtLinearObjectStatePdu();

    // Assignment operator
    DtLinearObjectStatePdu& operator=(const DtLinearObjectStatePdu& orig);

    // Set/Get object ID
    virtual void setObjectID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& objectID() const;

    // Set/Get referenced object ID
    virtual void setReferencedObjectID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& referencedObjectID() const;

    // Set/Get update number
    virtual void setUpdateNumber(DtU16 field);
    virtual DtU16 updateNumber() const;

    // Set/Get the force ID
    virtual void setForceID(DtForceType force);
    virtual DtForceType forceID() const;

    // Set/Get the number of segments
    virtual DtBoolean setNumSegments(DtU8 num);
    virtual DtU8 numSegments() const;

    // Set/Get the requester ID
```

```

virtual void setRequesterID(const DtSimulationAddress& type);
virtual const DtSimulationAddress& requesterID() const;

// Set/Get the receiving ID
virtual void setReceivingID(const DtSimulationAddress& type);
virtual const DtSimulationAddress& receivingID() const;

// Set/Get the object type
virtual void setObjectType(const DtObjectType& type);
virtual const DtObjectType& objectType() const;

// Set/Get a linear segment parameter record. Returns DtFalse if the
// index is out of range.
virtual DtBoolean setLinearSegmentParameter(int index,
    const DtLinearSegmentParameter& init);
virtual DtBoolean getLinearSegmentParameter(int index,
    DtLinearSegmentParameter& record) const;

// Print the data contained by this PDU
virtual void printData() const;

public:
    // Create a DtLinearObjectStatePdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtLinearObjectStatePduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtLinearObjectStatePduCb cb, void* usr);

protected:
    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetLinearObjectStatePdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non-virtual since you can't override based on return type
    DtLinearObjectStatePdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

protected:
    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:
    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
    DtObjectType myObjectType;
    DtSimulationAddress mySimulationAddress;
};
#endif
#endif

```

linSegParam.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "mathTypes.h"
#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/linObjStP.h"

// class DtLinearSegmentParameter:
//
// DtLinearSegmentParameter is a class which contains the linear segment
// parameters for the linear object state PDU.

class DtLinearSegmentParameter
{
public:

    // Constructors
    DtLinearSegmentParameter();
    DtLinearSegmentParameter(const DtLinearSegmentParameter& orig);

    // Assignment operator
    DtLinearSegmentParameter& operator=(
        const DtLinearSegmentParameter& orig);
    DtLinearSegmentParameter& operator=(
        const DtNetLinearSegmentParameter& orig);

    // Destructor
    virtual ~DtLinearSegmentParameter();

    // Set/Get the segment number
    virtual void setSegmentNumber(DtU8 value);
    virtual DtU8 segmentNumber() const;

    // Set/Get the modifications (by 8-bit integer)
    virtual void setModifications(DtU8 value);
    virtual DtU8 modifications() const;

    // Set/Get the modifications (by field)
    virtual void setModificationsField(
        DtModificationField field, DtBoolean value);
    virtual DtBoolean modificationsField(
        DtModificationField field) const;

    // Set/Get appearance, the lower 16 bits (as an integer)
    virtual void setAppearance(DtU16 app);
    virtual DtU16 appearance() const;

    // Set/Get appearance by attribute, the lower 16 bits
    virtual void setAppearanceAttribute(
        DtAppearAttribute attr, DtAppearAttrVal val);
    virtual DtAppearAttrVal appearanceAttribute(
        DtAppearAttribute attr) const;

    // Set/Get specific appearance, the upper 32 bits (as a 32 bit integer)
    virtual void setSpecificAppearance(DtU32 app);
    virtual DtU32 specificAppearance() const;

    // Set/Get specific appearance by attribute, the upper 32 bits
    virtual void setSpecificAppearanceAttribute(
        DtAppearAttribute attr, DtAppearAttrVal val);
```

```

virtual DtAppearAttrVal specificAppearanceAttribute(
    DtAppearAttribute attr) const;

// Set/Get location
virtual void setLocation(DtConstVector coords);
virtual DtConstVector location() const;

// Set/Get orientation
virtual void setOrientation(const DtTaitBryan& orientation);
virtual const DtTaitBryan& orientation() const;

// Set/Get the segment length
virtual void setSegmentLength(DtU16 value);
virtual DtU16 segmentLength() const;

// Set/Get the segment width
virtual void setSegmentWidth(DtU16 value);
virtual DtU16 segmentWidth() const;

// Set/Get the segment height
virtual void setSegmentHeight(DtU16 value);
virtual DtU16 segmentHeight() const;

// Set/Get the segment depth
virtual void setSegmentDepth(DtU16 value);
virtual DtU16 segmentDepth() const;

// Print the information contained in the record
virtual void print() const;

// Load/Unload data from a packet record
virtual void unloadNetRecord(
    const DtNetLinearSegmentParameter& record);
virtual void loadNetRecord(
    DtNetLinearSegmentParameter& record) const;

protected:

    // Data members:
    DtU8          mySegmentNumber;
    DtU8          myModifications;
    DtU32         mySpecificAppearance;
    DtU16         myAppearance;
    DtVector      myLocation;
    DtTaitBryan   myOrientation;
    DtU16         mySegmentLength;
    DtU16         mySegmentWidth;
    DtU16         mySegmentHeight;
    DtU16         mySegmentDepth;
};
#endif
#endif

```

locObjMgr.h

```
#if DtHLA

#include "hlaObj.h"
#include "objIdHashLst.h"

// class DtLocalHlaObjectManager:
//
// Instances of DtLocalHlaObjectManager are used to manage
// local HLA objects (objects which are being published by this
// application). This manager handles providing updates to the network.

class DtLocalHlaObjectManager
{
public:

    // default constructor
    DtLocalHlaObjectManager();

    // destructor
    virtual ~DtLocalHlaObjectManager();

    // Get the list of all HLA objects.
    virtual const DtList& allHlaObjects() const;

    // Get the HLA object based on its identifier.
    virtual DtHlaObject* hlaObject(RTI::ObjectHandle id) const;

    // process RTI's provideUpdate service.
    virtual void processProvideUpdate(RTI::ObjectHandle id,
        const RTI::AttributeHandleSet& theAttributes);

    // add the object to the list of local objects. This enables it to
    // get RTI callbacks destined for it.
    virtual void manage(DtHlaObject* obj);

    // remove the object from the list of local objects. Indicates
    // that the object is no longer being published.
    virtual void unManage(DtHlaObject* obj);

protected:

    // Note: The copy constructor and assignment operator are protected
    // because there should only be one object manager per exercise
    // connection.
    // These should not be called.

    // copy constructor
    DtLocalHlaObjectManager(const DtLocalHlaObjectManager& orig);

    // assignment operator
    DtLocalHlaObjectManager& operator=(const DtLocalHlaObjectManager& orig);

protected:

    DtObjectIdHashlist myObjectList;
};
#endif
```

MachineTypes.h

```

/*****
 *
Name:..... machine_types.h

Function:..... Contains all the structures and types needed for
net_types.h
*****/
*/
#ifndef _MachineTypes_h
#define _MachineTypes_h

#include "vlconfig.h"

/* defined(__cplusplus), not __cplusplus,
 * because __SUNPRO_CC defines it to be empty */
#if DT_HAVE_STD_H && defined(__cplusplus)
#include <std.h>
#endif

#include <sys/types.h>

typedef unsigned charDtU8;
typedef unsigned shortDtU16;

#if DT_HAVE_64BIT_LONGS
typedef unsigned intDtU32;
typedef intDtInt32;
#else
typedef unsigned longDtU32;
typedef long intDtInt32;
#endif

typedef charDtInt8;
typedef short intDtInt16;
typedef floatDtFloat32;
typedef doubleDtFloat64;
typedef DtFloat64DtReal; /* For convenience */

enum DtBoolean { DtFalse, DtTrue } ;

/* Windows requires extra work to export from DLLs */
#if DT_NEED_DLL_DECLARATIONS

#if DT_DLL_MTL
#undef DT_DLL_MTL
#define DT_DLL_MTL __declspec( dllexport )

#elif DT_DLL_MATRIX
#undef DT_DLL_MATRIX
#define DT_DLL_MATRIX __declspec( dllexport )

#elif DT_DLL_VL
#undef DT_DLL_VL
#define DT_DLL_VL __declspec( dllexport )

#elif DT_DLL_VLUTIL
#undef DT_DLL_VLUTIL
#define DT_DLL_VLUTIL __declspec( dllexport )
#endif

#ifndef DT_DLL_MTL
#define DT_DLL_MTL __declspec( dllimport )
#endif
#endif

```

```
#ifndef DT_DLL_MATRIX
#define DT_DLL_MATRIX    __declspec( dllimport )
#endif

#ifndef DT_DLL_VL
#define DT_DLL_VL        __declspec( dllimport )
#endif

#ifndef DT_DLL_VLUTIL
#define DT_DLL_VLUTIL    __declspec( dllimport )
#endif

/* else /* Default -- Unix or Win32 static */

#define DT_DLL_MTL
#define DT_DLL_MATRIX
#define DT_DLL_VL
#define DT_DLL_VLUTIL
#endif

#include "vlInit.h" // This should be last before endif in file.
```


mathTypes.h

```
#include <math.h>
#ifndef SQR
#define SQR(x) ((x) * (x))
#endif
#ifndef MIN
#define MIN(x, y) ((x)<(y) ? (x) : (y))
#endif
#ifndef MAX
#define MAX(x, y) ((x)>(y) ? (x) : (y))
#endif
#ifndef LIMIT
#define LIMIT(low, x, high) MAX(MIN(x, high), low)
#endif

// For DtVector
#include "vlVector.h"

// For DtDcm
#include "dcm.h"

// For DtTaitBryan
#include "taitBryan.h"

// Angular rotation about body X, Y, Z axes
typedef DtVector DtBodyRates;

// For DtDirectedRotRate and DtBodyRates
#include "directRotRate.h"
```

messagePdu.h

```
#if !DtHLA

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"

#if DtDISVERS <= 4

#define DtMessageStaticSize 32

class DtMessagePdu;
typedef void (*DtMessagePduCb)(DtMessagePdu *pdu, void *usr);

class DtMessagePdu : public DtSimanPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtMessagePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMessagePdu(const DtNetMessagePdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMessagePdu(const DtEntityIdentifier &sender,
const DtEntityIdentifier &receiver,
const char *message,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtMessagePduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtMessagePduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    int numVarFields() const;
    DtDatumParam varDatumId(int index) const;
    int varDatumSizeBits(int index) const;
    int varDatumSizeBytes(int index) const;
    int paramIndex(DtDatumParam param, int *retcode = NULL) const;

/***** INSPECTING DATUM VALUES BY INDEX *****/

    DtFloat32 datumValFloat32(int index, int *retcode = NULL) const;
    DtFloat64 datumValFloat64(int index, int *retcode = NULL) const;
    unsigned long datumValUnsigned32(int index, int *retcode = NULL) const;
    long datumValInt32(int index, int *retcode = NULL) const;
    const char* datumValByteArray(int index) const;

/***** INSPECTING DATUM VALUES BY PARAM ID *****/

    DtFloat32 datumValFloat32(DtDatumParam param,
int *retcode = NULL) const;
    DtFloat64 datumValFloat64(DtDatumParam param,
int *retcode = NULL) const;
    unsigned long datumValUnsigned32(DtDatumParam param,
int *retcode = NULL) const;
    long datumValInt32(DtDatumParam param,
int *retcode = NULL) const;
    const char* datumValByteArray(DtDatumParam param,
int *retcode = NULL) const;


```

```

/***** MUTATOR FUNCTIONS *****/

void initDataSection(const DtVarDatumSpec *specList);
void setNumVarFields(int newNum);
void setDatumParam(int index, DtDatumParam param, int *retcode = NULL);
void setVarDataBytes(int index, int length, int *retcode = NULL);
void setMessage(const char *message, int size);

/***** SETTING DATA VALUES BY INDEX *****/

void setDatumValFloat64(int index, DtFloat64 val, int *retcode = NULL);
void setDatumValFloat32(int index, DtFloat32 val, int *retcode = NULL);
void setDatumValInt32(int index, long val, int *retcode = NULL);
void setDatumValUnsigned32(int index, unsigned long val,
    int *retcode = NULL);
void setDatumValByteArray(int index, const char *val,
    int *retcode = NULL);

/***** SETTING DATA VALUES BY PARAM ID *****/

void setDatumValFloat64(DtDatumParam param, DtFloat64 val,
    int *retcode = NULL);
void setDatumValFloat32(DtDatumParam param, DtFloat32 val,
    int *retcode = NULL);
void setDatumValInt32(DtDatumParam param, long val,
    int *retcode = NULL);
void setDatumValUnsigned32(DtDatumParam param, unsigned long val,
    int *retcode = NULL);
void setDatumValByteArray(DtDatumParam param, const char *val,
    int *retcode = NULL);

void printData() const;

protected:

    DtPduKind internalGetPduKind() const {return DtMessagePduKind;}
    DtNetVarDatum *nthVarDataRecord(int index) const;
    void *datumValPtr(int index) const;
};
#endif // DtDISVERS
#endif // !DtHLA

```

milEntDec.h

```
#if DtHLA

#include "physEntDec.h"

// class DtMilitaryEntityDecoder:
//
// Instances of DtMilitaryEntityDecoder are used to decode a
// military entity state message into a state repository in an HLA
// (as opposed to DIS) environment.

class DtMilitaryEntityDecoder : public DtPhysicalEntityDecoder
{
public:
    // constructor
    DtMilitaryEntityDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtMilitaryEntityDecoder();

    // copy constructor
    DtMilitaryEntityDecoder(const DtMilitaryEntityDecoder& orig);

    // assignment operator
    DtMilitaryEntityDecoder& operator=(const DtMilitaryEntityDecoder& orig);

public:
    // Create a DtMilitaryEntityDecoder.
    // User is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    // add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
    DtDECLARE_ENTITY_ATTR_DECODER(AlternateEntityType);
    DtDECLARE_ENTITY_ATTR_DECODER(CamouflageType);
    DtDECLARE_ENTITY_ATTR_DECODER(FirePowerDisabled);
    DtDECLARE_ENTITY_ATTR_DECODER(ForceID);
    DtDECLARE_ENTITY_ATTR_DECODER(IsConcealed);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_MIL_ENTITY_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtMilitaryEntityDecoder, DtEntityStateRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_MIL_ENTITY_ATTR_DECODER( \
    attrName, netType, inspector) \
```

```
DtDEFINE_SIMPLE_MIL_ENTITY_ATTR_DECODER_WITH_CAST( \
    attrName, netType, inspector, (netType))
#endif
```

milEntEnc.h

```
#if DtHLA

#include "physEntEnc.h"
#include "entitySR.h"

// class DtMilitaryEntityEncoder:
//
// Instances of DtMilitaryEntityEncoder are used to encode a
// military entity's state repository into a state message in an
// HLA environment.

class DtMilitaryEntityEncoder : public DtPhysicalEntityEncoder
{
public:

    // constructor
    DtMilitaryEntityEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtMilitaryEntityEncoder();

    // copy constructor
    DtMilitaryEntityEncoder(const DtMilitaryEntityEncoder& orig);

    // assignment operator
    DtMilitaryEntityEncoder& operator=(const DtMilitaryEntityEncoder& orig);

public:

    // Create a DtMilitaryEntityEncoder.
    // Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // static need<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_CHECKER(AlternateEntityType);
    DtDECLARE_ENTITY_ATTR_CHECKER(CamouflageType);
    DtDECLARE_ENTITY_ATTR_CHECKER(FirePowerDisabled);
    DtDECLARE_ENTITY_ATTR_CHECKER(ForceID);
    DtDECLARE_ENTITY_ATTR_CHECKER(IsConcealed);

    // static decode<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_ENCODER(AlternateEntityType);
    DtDECLARE_ENTITY_ATTR_ENCODER(CamouflageType);
    DtDECLARE_ENTITY_ATTR_ENCODER(FirePowerDisabled);
    DtDECLARE_ENTITY_ATTR_ENCODER(ForceID);
    DtDECLARE_ENTITY_ATTR_ENCODER(IsConcealed);
};

// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_MIL_ENTITY_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtMilitaryEntityEncoder, \
        DtEntityStateRepository, \
        attrName, netType, inspector)
```

```
// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that checker whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_MIL_ENTITY_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtMilitaryEntityEncoder, \
        DtEntityStateRepository, \
        attrName, inspector)

#endif
```

milPlatDec.h

```
#if DtHLA

#include "milEntDec.h"

// class DtMilitaryPlatformEntityDecoder:
//
// Instances of DtMilitaryPlatformEntityDecoder are used to decode a
// military platform entity state message into a state repository in an HLA
// (as opposed to DIS) environment.

class DtMilitaryPlatformEntityDecoder : public DtMilitaryEntityDecoder
{
public:
    // constructor
    DtMilitaryPlatformEntityDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtMilitaryPlatformEntityDecoder();

    // copy constructor
    DtMilitaryPlatformEntityDecoder(
        const DtMilitaryPlatformEntityDecoder& orig);

    // assignment operator
    DtMilitaryPlatformEntityDecoder& operator=(
        const DtMilitaryPlatformEntityDecoder& orig);

public:
    // Create a DtMilitaryPlatformEntityDecoder.
    // User is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
    DtDECLARE_ENTITY_ATTR_DECODER(AfterburnerOn);
    DtDECLARE_ENTITY_ATTR_DECODER(HasAmmunitionSupplyCap);
    DtDECLARE_ENTITY_ATTR_DECODER(LauncherRaised);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_MIL_PLAT_ENTITY_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtMilitaryPlatformEntityDecoder, DtEntityStateRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_MIL_PLAT_ENTITY_ATTR_DECODER( \
    attrName, netType, inspector) \
```



```
DtDEFINE_SIMPLE_MIL_PLAT_ENTITY_ATTR_DECODER_WITH_CAST( \
    attrName, netType, inspector, (netType))
#endif
```

milPlatEnc.h

```
#if DtHLA

#include "milEntEnc.h"
#include "entitySR.h"

// class DtMilitaryPlatformEntityEncoder:
//
// Instances of DtMilitaryPlatformEntityEncoder are used to encode a
// military platform entity's state repository into a state message in an
// HLA environment.

class DtMilitaryPlatformEntityEncoder : public DtMilitaryEntityEncoder
{
public:

    // constructor
    DtMilitaryPlatformEntityEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtMilitaryPlatformEntityEncoder();

    // copy constructor
    DtMilitaryPlatformEntityEncoder(
        const DtMilitaryPlatformEntityEncoder& orig);

    // assignment operator
    DtMilitaryPlatformEntityEncoder& operator=(
        const DtMilitaryPlatformEntityEncoder& orig);

public:

    // Create a DtMilitaryPlatformEntityEncoder.
    // Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // static need<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_CHECKER(AfterburnerOn);
    DtDECLARE_ENTITY_ATTR_CHECKER(HasAmmunitionSupplyCap);
    DtDECLARE_ENTITY_ATTR_CHECKER(LauncherRaised);

    // static decode<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_ENCODER(AfterburnerOn);
    DtDECLARE_ENTITY_ATTR_ENCODER(HasAmmunitionSupplyCap);
    DtDECLARE_ENTITY_ATTR_ENCODER(LauncherRaised);
};

// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_MIL_PLAT_ENTITY_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtMilitaryPlatformEntityEncoder, \
        DtEntityStateRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
```

```
// (members of a derived DtHlaStateEncoder that checker whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_MIL_PLAT_ENTITY_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtMilitaryPlatformEntityEncoder, \
        DtEntityStateRepository, \
        attrName, inspector)

#endif
```

mineDataPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/mineDataP.h"

// class DtMinefieldDataPdu:
//
// Instances of DtMinefieldDataPdu are used to communicate minefield data
// information.

// DtMinefieldDataPdu callback function signature.
typedef void (*DtMinefieldDataPduCb)(DtMinefieldDataPdu* pdu, void* usr);

class DtMinefieldDataPdu : public DtPdu
{
public:

    // Constructors
    DtMinefieldDataPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMinefieldDataPdu(const DtNetMinefieldDataPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMinefieldDataPdu(const DtMinefieldDataPdu& orig);

    // destructor
    virtual ~DtMinefieldDataPdu();

    // Assignment operator
    DtMinefieldDataPdu& operator=(const DtMinefieldDataPdu& orig);

    // Set/Get minefield ID
    virtual void setMinefieldID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& minefieldID() const;

    // Set/Get requesting entity ID
    virtual void setRequestingEntityID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& requestingEntityID() const;

    // Set/Get minefield sequence number
    virtual void setMinefieldSequenceNumber(DtU16 field);
    virtual DtU16 minefieldSequenceNumber() const;

    // Set/Get the request ID
    virtual void setRequestID(DtU8 value);
    virtual DtU8 requestID() const;

    // Set/Get the PDU sequence number
    virtual void setPDU_SequenceNumber(DtU8 value);
    virtual DtU8 PDU_SequenceNumber() const;

    // Set/Get the number of PDUs
    virtual void setNumberOfPdus(DtU8 value);
    virtual DtU8 numberOfPdus() const;

    // Set/Get the number of mines (this call erases any data which was set
    // in the variable length fields)
    virtual DtBoolean setNumberOfMines(DtU8 value);
```

```

virtual DtU8 numberOfMines() const;

// Set/Get the number of sensor types (this call erases any data which
// was set in the variable length fields)
virtual DtBoolean setNumberOfSensorTypes(DtU8 value);
virtual DtU8 numberOfSensorTypes() const;

// Set/Get the data filter as a 32-bit value (this call erases any data
// which was set in the variable length fields)
virtual void setDataFilter(DtU32 value);
virtual DtU32 dataFilter() const;

// Set/Get a data filter bit (this call erases any data which was
// set in the variable length fields)
virtual void setDataFilterBit(
    DtMinefieldDataFilterField field, DtBoolean value);
virtual DtBoolean dataFilterBit(DtMinefieldDataFilterField field) const;

// Set/Get the mine type record.
virtual void setMineType(const DtEntityType& value);
virtual const DtEntityType& mineType() const;

// Set/Get a sensor type category. Returns DtFalse if the index is out
// of range.
virtual DtBoolean setSensorTypeCategory(
    int sensorIndex, DtSensorTypeCategory value);
virtual DtBoolean getSensorTypeCategory(
    int sensorIndex, DtSensorTypeCategory& value) const;

// Set/Get a sensor type sub-category. Returns DtFalse if the index is
out
// of range.
virtual DtBoolean setSensorTypeSubcategory(
    int sensorIndex, DtU16 value);
virtual DtBoolean getSensorTypeSubcategory(
    int sensorIndex, DtU16& value) const;

// Set/Get a mine location. Returns DtFalse if the index is out of
range.
virtual DtBoolean setMineLocation(int mineIndex, DtConstVector value);
virtual DtBoolean getMineLocation(int mineIndex, DtVectorRef value)
const;

// Set/Get a ground burial depth offset. Returns DtFalse if the index
// is out of range or if an appropriate flag bit is not set.
virtual DtBoolean setGroundBurialDepthOffset(
    int mineIndex, DtFloat32 value);
virtual DtBoolean getGroundBurialDepthOffset(
    int mineIndex, DtFloat32& value) const;

// Set/Get a water burial depth offset. Returns DtFalse if the index is
// out of range or if the appropriate flag bit is not set.
virtual DtBoolean setWaterBurialDepthOffset(
    int mineIndex, DtFloat32 value);
virtual DtBoolean getWaterBurialDepthOffset(
    int mineIndex, DtFloat32& value) const;

// Set/Get a snow burial depth offset. Returns DtFalse if the index is
out
// of range or if the appropriate flag bit is not set.
virtual DtBoolean setSnowBurialDepthOffset(
    int mineIndex, DtFloat32 value);
virtual DtBoolean getSnowBurialDepthOffset(
    int mineIndex, DtFloat32& value) const;

// Set/Get mine orientation. Returns DtFalse if the index is out of
range

```

```
// or if the appropriate flag bit is not set.
virtual DtBoolean setMineOrientation(
    int mineIndex, const DtTaitBryan& value);
virtual DtBoolean getMineOrientation(
    int mineIndex, DtTaitBryan& value) const;

// Set/Get a thermal contrast value. Returns DtFalse if the index
// is out of range or if the appropriate flag bit is not set.
virtual DtBoolean setThermalContrast(int mineIndex, DtFloat32 value);
virtual DtBoolean getThermalContrast(int mineIndex, DtFloat32& value)
const;

// Set/Get a reflectance value. Returns DtFalse if the index
// is out of range or if the appropriate flag bit is not set.
virtual DtBoolean setReflectance(int mineIndex, DtFloat32 value);
virtual DtBoolean getReflectance(int mineIndex, DtFloat32& value) const;

// Set/Get a mine emplacement time. Returns DtFalse if the index
// is out of range or if the appropriate flag bit is not set.
virtual DtBoolean setMineEmplacementTime(int mineIndex, DtTime value);
virtual DtBoolean getMineEmplacementTime(int mineIndex, DtTime& value)
const;

// Set/Get a mine entity ID. Returns DtFalse if the index is out of
range.
virtual DtBoolean setMineEntityID(int mineIndex, DtU16 value);
virtual DtBoolean getMineEntityID(int mineIndex, DtU16& value) const;

// Set/Get primary fusing. Returns DtFalse if the index is out of range
// or if the appropriate flag bit is not set.
virtual DtBoolean setPrimaryFusing(int mineIndex,
    DtMinefieldFusing value);
virtual DtBoolean getPrimaryFusing(int mineIndex,
    DtMinefieldFusing& value) const;

// Set/Get secondary fusing. Returns DtFalse if the index is out of
range
// or if the appropriate flag bit is not set.
virtual DtBoolean setSecondaryFusing(int mineIndex,
    DtMinefieldFusing value);
virtual DtBoolean getSecondaryFusing(int mineIndex,
    DtMinefieldFusing& value) const;

// Set/Get anti-handling device. Returns DtFalse if the index is out of
// range or if the appropriate flag bit is not set.
virtual DtBoolean setAntiHandlingDevice(int mineIndex,
    DtMinefieldAntiHandlingDevice value);
virtual DtBoolean getAntiHandlingDevice(int mineIndex,
    DtMinefieldAntiHandlingDevice& value) const;

// Set/Get scalar detection coefficient. Returns DtFalse if the index
// is out of range or if the appropriate flag bit is not set.
virtual DtBoolean setScalarDetectionCoefficient(
    int sensorIndex, int mineIndex, DtU8 value);
virtual DtBoolean getScalarDetectionCoefficient(
    int sensorIndex, int mineIndex, DtU8& value) const;

// Set/Get minefield algae. Returns DtFalse if the index
// is out of range or if the appropriate flag bit is not set.
virtual DtBoolean setMinefieldAlgae(
    int mineIndex, DtMinefieldAlgae value);
virtual DtBoolean getMinefieldAlgae(
    int mineIndex, DtMinefieldAlgae& value) const;

// Set/Get paint scheme. Returns DtFalse if the index
// is out of range or if the appropriate flag bit is not set.
virtual DtBoolean setPaintScheme(
```

```

        int mineIndex, DtMinefieldPaintScheme value);
virtual DtBoolean getPaintScheme(
    int mineIndex, DtMinefieldPaintScheme& value) const;

// Set/Get the number of trip/detonation wires. Returns DtFalse if the
// index is out of range or if the appropriate flag bit is not set.
virtual DtBoolean setNumTripWires(int mineIndex, DtU8 num);
virtual DtBoolean getNumTripWires(int mineIndex, DtU8& num) const;

// Set/Get the number of vertices. Returns DtFalse if the index is out
// of range or if the appropriate flag bit is not set.
virtual DtBoolean setNumVertices(
    int mineIndex, int wireIndex, DtU8 num);
virtual DtBoolean getNumVertices(
    int mineIndex, int wireIndex, DtU8& num) const;

// Set/Get a vertex. Returns DtFalse if the index is out of range or if
// the appropriate flag bit is not set.
virtual DtBoolean setVertex(int mineIndex, int wireIndex, int vertex-
Index,
    DtConstVector value);
virtual DtBoolean getVertex(int mineIndex, int wireIndex, int vertex-
Index,
    DtVectorRef value) const;

// Print the data contained by this PDU
virtual void printData() const;

public:

    // Create a DtMinefieldDataPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtMinefieldDataPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtMinefieldDataPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetMinefieldDataPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non-virtual since you can't override based on return type
    DtMinefieldDataPdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

    // Returns a buffer offset to the sensor type records
    virtual int sensorTypeRecordOffset(int sensorIndex) const;

    // Returns a buffer offset to the padding following sensor type informa-
tion
    virtual int sensorTypePaddingOffset() const;

    // Returns the number of padding bytes following sensor type information
    virtual int sensorTypePaddingBytes() const;

```

```
// Returns a buffer offset to the mine location records
virtual int mineLocationRecordOffset(int mineIndex) const;

// Returns a buffer offset to the ground burial depth records
virtual int groundBurialDepthRecordOffset(int mineIndex) const;

// Returns a buffer offset to the water burial depth records
virtual int waterBurialDepthRecordOffset(int mineIndex) const;

// Returns a buffer offset to the snow burial depth records
virtual int snowBurialDepthRecordOffset(int mineIndex) const;

// Returns a buffer offset to the mine orientation records
virtual int mineOrientationRecordOffset(int mineIndex) const;

// Returns a buffer offset to the thermal contrast records
virtual int thermalContrastRecordOffset(int mineIndex) const;

// Returns a buffer offset to the reflectance records
virtual int reflectanceRecordOffset(int mineIndex) const;

// Returns a buffer offset to the mine emplacement time records
virtual int mineEmplacementTimeRecordOffset(int mineIndex) const;

// Returns a buffer offset to the mine entity ID records
virtual int mineEntityID_RecordOffset(int mineIndex) const;

// Returns a buffer offset to the fusing records
virtual int fusingRecordOffset(int mineIndex) const;

// Returns a buffer offset to the scalar detection coefficient records
virtual int scalarDetectionCoefficientRecordOffset(
    int sensorIndex, int mineIndex) const;

// Returns a buffer offset to the paint scheme records
virtual int paintSchemeRecordOffset(int mineIndex) const;

// Returns a buffer offset to the padding following paint scheme records
virtual int paintSchemePaddingOffset() const;

// Returns the number of padding bytes following paint scheme records
virtual int paintSchemePaddingBytes() const;

// Returns a buffer offset to the number of wires records
virtual int numberOfTripWiresRecordOffset(int mineIndex) const;

// Returns a buffer offset to the padding following number of wires
records
virtual int numberOfTripWiresPaddingOffset() const;

// Returns the number of padding bytes following number of wires records
virtual int numberOfTripWiresPaddingBytes() const;

// Returns the length of the variable data section in the PDU after the
// variable data has been reset. For each mine, the number of trip wires
// has been set to zero so there is no variable length data following.
virtual int variableDataLengthAfterReset() const;

// Returns a buffer offset to the number of vertices records
virtual int numberOfVerticesRecordOffset(
    int mineIndex, int wireIndex) const;

// Returns a buffer offset to the padding following number of
// vertices records
virtual int numberOfVerticesPaddingOffset() const;
```



```
    // Returns the number of padding bytes following number of vertices
records
    virtual int numberOfVerticesPaddingBytes() const;

    // Returns a buffer offset to the vertex records
    virtual int vertexRecordOffset(int mineIndex, int wireIndex,
        int vertexIndex) const;

    // Returns the length of the variable data section in the PDU
    virtual int variableDataLength() const;

protected:

    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
    DtEntityType myEntityType;

};
#endif
#endif
```

minePerimeter.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/mineStateP.h"

// class DtMinefieldPerimeterRecord:
//
// DtMinefieldPerimeterRecord is a class which contains coordinates for the
// minefield perimeter points. This record is used in the minefield state
// and minefield query PDUs.

class DtMinefieldPerimeterRecord
{
public:

    // Constructors
    DtMinefieldPerimeterRecord();
    DtMinefieldPerimeterRecord(const DtMinefieldPerimeterRecord& orig);

    // Assignment operator
    DtMinefieldPerimeterRecord& operator=(
        const DtMinefieldPerimeterRecord& orig);
    DtMinefieldPerimeterRecord& operator=(
        const DtNetMinefieldPerimeterRecord& orig);

    // Destructor
    virtual ~DtMinefieldPerimeterRecord();

    // Set/Get the X Component
    virtual void setX(DtFloat32 value);
    virtual DtFloat32 x() const;

    // Set/Get the Y Component
    virtual void setY(DtFloat32 value);
    virtual DtFloat32 y() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetMinefieldPerimeterRecord& record);
    virtual void loadNetRecord(
        DtNetMinefieldPerimeterRecord& record) const;

protected:

    // Data members:
    DtFloat32          myX;
    DtFloat32          myY;

};
#endif
#endif
```

mineQueryPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"

#include "packets/mineQueryP.h"

#include "minePerimeter.h"

// class DtMinefieldQueryPdu:
//
// Instances of DtMinefieldQueryPdu are used to communicate minefield state
// information.

// DtMinefieldQueryPdu callback function signature.
typedef void (*DtMinefieldQueryPduCb)(DtMinefieldQueryPdu* pdu, void* usr);

class DtMinefieldQueryPdu : public DtPdu
{
public:
    // Constructors
    DtMinefieldQueryPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMinefieldQueryPdu(const DtNetMinefieldQueryPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMinefieldQueryPdu(const DtMinefieldQueryPdu& orig);

    // destructor
    virtual ~DtMinefieldQueryPdu();

    // Assignment operator
    DtMinefieldQueryPdu& operator=(const DtMinefieldQueryPdu& orig);

    // Set/Get minefield ID
    virtual void setMinefieldID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& minefieldID() const;

    // Set/Get requesting entity ID
    virtual void setRequestingEntityID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& requestingEntityID() const;

    // Set/Get request ID
    virtual void setRequestID(DtU8 id);
    virtual DtU8 requestID() const;

    // Set/Get the number of perimeter points
    virtual DtBoolean setNumPerimeterPoints(DtU8 num);
    virtual DtU8 numPerimeterPoints() const;

    // Set/Get the number of sensor types
    virtual DtBoolean setNumSensorTypes(DtU8 num);
    virtual DtU8 numSensorTypes() const;

    // Set/Get the data filter (as a 32-bit integer)
    virtual void setDataFilter(DtU32 value);
    virtual DtU32 dataFilter() const;

    // Set/Get data filter attributes (DtTrue indicates that the data for
    that

```

```
// field is requested, DtFalse indicates that the data for that field is
// not requested.
virtual void setDataFilterField(
    DtMinefieldDataFilterField field, DtBoolean value);
virtual DtBoolean getDataFilterField(DtMinefieldDataFilterField field);

// Set/Get requested mine type
virtual void setRequestedMineType(const DtEntityType& type);
virtual const DtEntityType& requestedMineType() const;

// Set/Get a perimeter point coordinate record. Returns DtFalse if the
// index is out of range.
virtual DtBoolean setPerimeterPoint(int index,
    const DtMinefieldPerimeterRecord& init);
virtual DtBoolean getPerimeterPoint(int index,
    DtMinefieldPerimeterRecord& record) const;

// Get the sensor type category for a record. Returns DtFalse if the
// index is out of range. The category is set by setSensorTypeRecord.
virtual DtBoolean getSensorTypeCategory(int index,
    DtSensorTypeCategory& value);

// Set a sensor type subcategory for a record. The correct category is
// automatically set by this function according to type of the enumera-
tion
// in the argument list. DtFalse is returned if the index is out of
range.
virtual DtBoolean setSensorTypeSubcategory(int index,
    DtOpticalSensorTypeSubcategory value);
virtual DtBoolean setSensorTypeSubcategory(int index,
    DtFLIR_SensorTypeSubcategory value);
virtual DtBoolean setSensorTypeSubcategory(int index,
    DtRADAR_SensorTypeSubcategory value);
virtual DtBoolean setSensorTypeSubcategory(int index,
    DtMagneticSensorTypeSubcategory value);
virtual DtBoolean setSensorTypeSubcategory(int index,
    DtLaserSensorTypeSubcategory value);
virtual DtBoolean setSensorTypeSubcategory(int index,
    DtSONARSensorTypeSubcategory value);
virtual DtBoolean setSensorTypeSubcategory(int index,
    DtPhysicalSensorTypeSubcategory value);
virtual DtBoolean setSensorTypeSubcategory(int index,
    DtMultispectralSensorTypeSubcategory value);

// Get a sensor type subcategory for a record. DtFalse is returned if
the
// index is out of range or if the category of the record does not match
// the category indicated by the type of the enumeration in the argument
// list.
virtual DtBoolean getSensorTypeSubcategory(int index,
    DtOpticalSensorTypeSubcategory& value);
virtual DtBoolean getSensorTypeSubcategory(int index,
    DtFLIR_SensorTypeSubcategory& value);
virtual DtBoolean getSensorTypeSubcategory(int index,
    DtRADAR_SensorTypeSubcategory& value);
virtual DtBoolean getSensorTypeSubcategory(int index,
    DtMagneticSensorTypeSubcategory& value);
virtual DtBoolean getSensorTypeSubcategory(int index,
    DtLaserSensorTypeSubcategory& value);
virtual DtBoolean getSensorTypeSubcategory(int index,
    DtSONARSensorTypeSubcategory& value);
virtual DtBoolean getSensorTypeSubcategory(int index,
    DtPhysicalSensorTypeSubcategory& value);
virtual DtBoolean getSensorTypeSubcategory(int index,
    DtMultispectralSensorTypeSubcategory& value);

// Print the data contained by this PDU
```

```

    virtual void printData() const;

public:
    // Create a DtMinefieldQueryPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtMinefieldQueryPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtMinefieldQueryPduCb cb, void* usr);

protected:
    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetMinefieldQueryPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non-virtual since you can't override based on return type
    DtMinefieldQueryPdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

    // Returns a buffer offset to a perimeter point record
    virtual int perimeterPointRecordOffset(int recordIndex) const;

    // Returns a buffer offset to a sensor type record
    virtual int sensorTypeRecordOffset(int recordIndex) const;

    // Set/Get the sensor type. (As a 16-bit integer)
    virtual void setSensorType(int index, DtU16 value);
    virtual DtU16 sensorType(int index) const;

protected:
    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:
    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
    DtEntityType myEntityType;
    DtMinefieldPerimeterRecord myPerimeterRecord;
};
#endif
#endif

```

mineRsNackPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"

#include "packets/mineRsNackP.h"

// class DtMinefieldResponseNackPdu:
//
// Instances of DtMinefieldResponseNackPdu are used to request a
// retransmission of minefield data PDUs.

// DtMinefieldResponseNackPdu callback function signature.
typedef void (*DtMinefieldResponseNackPduCb)
    (DtMinefieldResponseNackPdu* pdu, void* usr);

class DtMinefieldResponseNackPdu : public DtPdu
{
public:

    // Constructors
    DtMinefieldResponseNackPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMinefieldResponseNackPdu(const DtNetMinefieldResponseNackPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMinefieldResponseNackPdu(const DtMinefieldResponseNackPdu& orig);

    // destructor
    virtual ~DtMinefieldResponseNackPdu();

    // Assignment operator
    DtMinefieldResponseNackPdu& operator=(
        const DtMinefieldResponseNackPdu& orig);

    // Set/Get minefield ID
    virtual void setMinefieldID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& minefieldID() const;

    // Set/Get requesting entity ID
    virtual void setRequestingEntityID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& requestingEntityID() const;

    // Set/Get request ID
    virtual void setRequestID(DtU8 id);
    virtual DtU8 requestID() const;

    // Set/Get the number of missing PDUs
    virtual DtBoolean setNumMissingPDUs(DtU8 num);
    virtual DtU8 numMissingPDUs() const;

    // Set/Get a PDU sequence number. Returns DtFalse if the
    // index is out of range.
    virtual DtBoolean setPDU_SequenceNumber(int index, DtU8 num);
    virtual DtBoolean getPDU_SequenceNumber(int index, DtU8& num) const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:
```

```

// Create a DtMinefieldResponseNackPdu.
// Caller is responsible for deletion.
static DtPdu* create(const DtNetPacket* initial,
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtMinefieldResponseNackPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtMinefieldResponseNackPduCb cb, void* usr);

protected:

// Returns the internal PDU kind
virtual DtPduKind internalGetPduKind() const;

// Casts the netPdu element to our network struct
virtual DtNetMinefieldResponseNackPdu* netPduStruct() const;

// Returns a pointer to the first byte of variable length data
virtual char* var() const;

// Casts the 'this' pointer for changing elements in a const function
// Non-virtual since you can't override based on return type
DtMinefieldResponseNackPdu* self() const;

protected:

// Size of PDU with zero parameter records.
static int theStaticSize;

protected:

// Temporary objects for reference returns
DtEntityIdentifier myEntityID;
};
#endif
#endif

```

mineStatePdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/mineStateP.h"

#include "minePerimeter.h"

// class DtMinefieldStatePdu:
//
// Instances of DtMinefieldStatePdu are used to communicate minefield state
// information.

// DtMinefieldStatePdu callback function signature.
typedef void (*DtMinefieldStatePduCb)(DtMinefieldStatePdu* pdu, void* usr);

class DtMinefieldStatePdu : public DtPdu
{
public:
    // Constructors
    DtMinefieldStatePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMinefieldStatePdu(const DtNetMinefieldStatePdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMinefieldStatePdu(const DtMinefieldStatePdu& orig);

    // destructor
    virtual ~DtMinefieldStatePdu();

    // Assignment operator
    DtMinefieldStatePdu& operator=(const DtMinefieldStatePdu& orig);

    // Set/Get minefield ID
    virtual void setMinefieldID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& minefieldID() const;

    // Set/Get minefield sequence number
    virtual void setSequenceNumber(DtU16 field);
    virtual DtU16 sequenceNumber() const;

    // Set/Get the force ID
    virtual void setForceID(DtForceType force);
    virtual DtForceType forceID() const;

    // Set/Get the number of perimeter points
    virtual DtBoolean setNumPerimeterPoints(DtU8 num);
    virtual DtU8 numPerimeterPoints() const;

    // Set/Get minefield type
    virtual void setMinefieldType(const DtEntityType& type);
    virtual const DtEntityType& minefieldType() const;

    // Set/Get the number of mine types
    virtual DtBoolean setNumMineTypes(DtU16 num);
    virtual DtU16 numMineTypes() const;

    // Set/Get location of minefield
```



```

virtual void setLocation(DtConstVector coords);
virtual DtConstVector location() const;

// Set/Get orientation
virtual void setOrientation(const DtTaitBryan& orientation);
virtual const DtTaitBryan& orientation() const;

// Set/Get appearance (as a 16 bit integer)
virtual void setAppearance(DtU16 app);
virtual DtU16 appearance() const;

// Set/Get appearance by attribute
virtual void setAppearanceAttribute(
    DtAppearAttribute attr, DtAppearAttrVal val);
virtual DtAppearAttrVal appearanceAttribute(
    DtAppearAttribute attr) const;

// Set/Get which network protocol mode is being used in the
// minefield transmission.
virtual void setMinefieldProtocolMode(DtMinefieldProtocol mode);
virtual DtMinefieldProtocol minefieldProtocolMode() const;

// Set/Get a perimeter point coordinate record, returns DtFalse if the
// index is out of range.
virtual DtBoolean setPerimeterPoint(int index,
    const DtMinefieldPerimeterRecord& init);
virtual DtBoolean getPerimeterPoint(int index,
    DtMinefieldPerimeterRecord& record) const;

// Set/Get a mine type record. Returns DtFalse if the index is
// out of range.
virtual DtBoolean setMineType(int index,
    const DtEntityType& init);
virtual DtBoolean getMineType(int index,
    DtEntityType& record) const;

// Print the data contained by this PDU
virtual void printData() const;

public:

    // Create a DtMinefieldStatePdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtMinefieldStatePduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtMinefieldStatePduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetMinefieldStatePdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non-virtual since you can't override based on return type
    DtMinefieldStatePdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

```

```
// Returns a buffer offset to a perimeter point record
virtual int perimeterPointRecordOffset(int recordIndex) const;

// Returns a buffer offset to a mine type record
virtual int mineTypeRecordOffset(int recordIndex) const;

protected:

    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
    DtEntityType myEntityType;
    DtTaitBryan myEulerAngles;
    DtVector myVector;
    DtMinefieldPerimeterRecord myPerimeterRecord;

};
#endif
#endif
```

munEntDec.h

```

#if DtHLA

#include "milEntDec.h"

// class DtMunitionEntityDecoder:
//
// Instances of DtMunitionEntityDecoder are used to decode a
// munition entity state message into a state repository in an HLA
// (as opposed to DIS) environment.

class DtMunitionEntityDecoder : public DtMilitaryEntityDecoder
{
public:
    // constructor
    DtMunitionEntityDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtMunitionEntityDecoder();

    // copy constructor
    DtMunitionEntityDecoder(const DtMunitionEntityDecoder& orig);

    // assignment operator
    DtMunitionEntityDecoder& operator=(const DtMunitionEntityDecoder& orig);

public:
    // Create a DtMunitionEntityDecoder.
    // User is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
    DtDECLARE_ENTITY_ATTR_DECODER(LauncherFlashPresent);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_MUN_ENTITY_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtMunitionEntityDecoder, DtEntityStateRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_MUN_ENTITY_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_MUN_ENTITY_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif

```

munEntEnc.h

```
#if DtHLA

#include "milEntEnc.h"
#include "entitySR.h"

// class DtMunitionEntityEncoder:
//
// Instances of DtMunitionEntityEncoder are used to encode a
// munition entity's state repository into a state message in an
// HLA environment.

class DtMunitionEntityEncoder : public DtMilitaryEntityEncoder
{
public:

    // constructor
    DtMunitionEntityEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtMunitionEntityEncoder();

    // copy constructor
    DtMunitionEntityEncoder(const DtMunitionEntityEncoder& orig);

    // assignment operator
    DtMunitionEntityEncoder& operator=(const DtMunitionEntityEncoder& orig);

public:

    // Create a DtMunitionEntityEncoder.
    // Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:

    // static need<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_CHECKER(LauncherFlashPresent);

    // static decode<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_ENCODER(LauncherFlashPresent);
};

// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_MUN_ENTITY_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtMunitionEntityEncoder, \
        DtEntityStateRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that checker whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_MUN_ENTITY_ATTR_CHECKER( \
    attrName, inspector) \
```

```
DtDEFINE_SIMPLE_ATTR_CHECKER(DtMunitionEntityEncoder, \  
    DtEntityStateRepository, \  
    attrName, inspector)  
#endif
```

netBoolean.h

```
#if DtHLA

#include "NetTypes.h"
#include "MachineTypes.h"

// class DtNetBoolean:
//
// Instances of DtNetBoolean are network representations of booleans.
// The only thing that's required is that they can be cast to DtBooleans.
// This class has NO virtual functions since it is a NetStruct and we
// rely on its size and byte layout.

#if !DtrPR016

typedef DtNetU8 DtNetBoolean;

#else

#if !DT_LITTLE_ENDIAN

typedef DtNetU32 DtNetBoolean;

#else

class DtNetBoolean
{
public:

    // default constructor
    DtNetBoolean();

    // destructor
    ~DtNetBoolean();

    // If a copy constructor is implemented, the assignment operator should
    // also be implemented. Should generally implement both of them anyway
    to
    // avoid memory problems due to inappropriate compiler provided defaults.
    // Access varies so need to put proper access keyword above these
    // declarations.

    // copy constructor
    DtNetBoolean(const DtNetBoolean& orig);

    // assignment operator
    DtNetBoolean& operator=(const DtNetBoolean& orig);

    // from host constructor
    DtNetBoolean(DtBoolean val);

    // from host assignment operator
    DtNetBoolean& operator=(DtBoolean val);

    // to host cast operator
    DtNetBoolean& operator DtBoolean() const;

protected:

    DtNetU32 myRep;
};
#endif
#endif
#endif
```

NetSocket.h

```

/*
 *                               class NetSocket
 *
 * NetSocket provides several overloaded constructors that allow the user
 * to specify UDP port, device, and destination address in any combination
 * except port and destination address together.
 *
 * In addition, each constructor has an optional flags argument, which
 * may be set to DtRead (read only), DtWrite (write only) or both
 * (default, DtRead | DtWrite).
 *
 * The default destination address is broadcast.
 * The default network interface device is the first appropriate one
 * located by the constructor.
 */

#ifndef _NetSocket_h
#define _NetSocket_h

#include "MachineTypes.h" /* needed for <sys/types.h> or relatives */
#include "NetTypes.h"

#if !WIN32
extern "C" {
#include <netinet/in.h>
}
#endif

#if __osf__
#include <sys/param.h> /* needed for stuff that should be netinet/in.h */
#endif

#ifdef _GT_
#include "IpUdp.h"

#include <sys/socket.h> /* need for if.h */
#include <net/if.h> /* need for if_ether.h */
#include <net/if_arp.h> /* need for if_ether.h */
#include <netinet/if_ether.h> /* need for struct ether_header */
#endif

#define DtRead((DtU8) 0x01)
#define DtWrite((DtU8) 0x02)
#define DtReadWrite (DtRead | DtWrite)

/* setting this bit in the flags constructor argument will cause the
   NetSocket to ignore all traffic (incl. broadcast when dest_addr is
   multicast and point-to-point) besides that to dest_addr. */
#define DtBindToDestAddr ((DtU8) 0x08)

#define DtQuiet ((DtU8) 0x10)

enum DtInitErrorAction {
    DtInitErrorFatal,
    DtInitErrorWarn,
    DtInitErrorIgnore
};

#include "Socket.h"
#include "vlutil.h"

#ifdef NULL
#undef NULL
#define NULL 0

```

```
#endif

/* *****

***** */

#ifdef _GT_

typedef struct gtQueueElement
{
    int datalen;
    caddr_t data;
} GtQueueElement;

#endif

class DtNetSocket : public DtSocket
{
public:

    DtNetSocket(int port=DtDEFAULT_PORT, DtU8 flags=DtReadWrite);

    /* No default for flags since 3rd arg is needed to disambiguate.
     * Note a real loss, however, since any point-to-point NetSocket
     * should probably be write-only anyway.
     */
    DtNetSocket(int port, DtInetAddr destAddr, DtU8 flags);

    /* This constructor is obsolete, and will eventually disappear
     * Use DtInitNetSocket(port, dest_addr->s_addr) instead */
    DtNetSocket(int port, const struct in_addr *dest_addr,
DtU8 flags=DtReadWrite)
    {
DtInitNetSocket (port, dest_addr->s_addr, flags, NULL);
    }

    DtNetSocket(char* dummy, int sockfd, DtU8 flags = DtReadWrite);

#ifdef _GT_
    /* It only makes sense to set both destAddr and device on the GT.
     * Under UNIX (with IP sockets), the device is implied by the destAddr
     * It also only makes sense to speciify from_addr on the _GT_. On UNIX
     * boxes, the from IP address gets filled in automagically.
     */
    DtNetSocket(int port, DtInetAddr fromAddr, DtInetAddr destAddr,
const char *device, DtU8 flags = DtReadWrite);
#endif

    virtual ~DtNetSocket();

    int sendTo(caddr_t packet, size_t size, DtInetAddr addr);

    int DtRecv(caddr_t buff, size_t);
    int DtRecv(caddr_t *buffptr, int *status=NULL);
    int DtReportPktsRcvd(void);
    DtInetAddr lastSourceInetAddr() {
return ntohl(sourceSockAddr.sin_addr.s_addr);
    }

    void DtBecomeNonBlocking(void);
    void setUnbundling(int onOff);
    int isUnbundling();
    int dataLeftInCurrentBundle();

    int readFd() { return ((RW_flags & DtRead) ? skt_num : -1); }
    int writeFd() { return ((RW_flags & DtWrite) ? skt_num : -1); }
```



```

    int isOk()    { return ok; }

    static DtInitErrorAction initErrorAction;
#ifdef PCFTP || WIN32
    static DtU32 DtInetBroadcastMask;
    void DtGetBroadcastMask();
#endif

    int udpPort;      /* must be kept around for GT */

inline int listenToMulticastGroup(DtInetAddr multicastAddr) {
    return PrivateListenToMulticastGroup(multicastAddr, 0); }

    int dropMulticastGroup(DtInetAddr multicastAddr);

int setMcastTtl(int ttl);

#ifdef _GT_
    int DtGtPacketDequeue(caddr_t);
    void DtGtPacketEnqueue(char *);
#endif

protected:
    void DtInitNetSocket(int port, DtInetAddr dest_addr,
DtU8 rw_flags, const char *device);

    int recvWithStatus(caddr_t *buffptr, int *status);
    int recvFromNet(caddr_t *buffptr);
    int recvWithUnbundling(caddr_t *buffptr);
    void *nextPdu;
    int sizeLeft;
    int unbundling;
    int m_bExternalSocket;
    DtInetAddr m_devAddr;

private:
    DtNetFloat64 pktBuff8[DtMaxPacketSize/sizeof(DtNetFloat64)];
    DtNetFloat64 unbundleBuff8[DtMaxPacketSize/sizeof(DtNetFloat64)];

    int init_iodev(const char *);
    int init_write(DtInetAddr *dest);
    int init_read(DtInetAddr dest /* may become the read addr */);
    int PrivateListenToMulticastGroup(DtInetAddr multicastAddr, int
on_init=0);
    void findNextPdu();
    unsigned int currentHeaderMatchesFirstInBundle();

    DtU8 RW_flags;
    int boundToDestAddr;
    DtInetAddr destInetAddr;
    int firstInBundle;
    int firstExId;
    int firstProtocol;

    int ok;
    int skt_num; /* UNIX socket number. */
    struct sockaddr_in destSockAddr;
    struct sockaddr_in sourceSockAddr;

#ifdef _GT_
    char outBuf[2048]; /* does DIS have a defined maximum? */
        // No, but Ethernet does, and that's the only
        // thing the GT100 will ever work with

    int DtGtSend(caddr_t, size_t);
    int DtGtRecv(caddr_t bufp, size_t bufsize);
    GtQueueElement *PacketQueueOut = 0;

```

```
GtQueueElement *PacketQueueIn = 0;

/* These point into the above structure after INIT */
struct ether_header *ethernetHdr;
struct ip *ipHdr;
struct udphdr *udpHdr;
DtInetAddr          from_addr = 0;
#endif

};
/* The functionality of the older constructor
 * DtNetSocket(const char *device, int port)
 * has been replaced by the call
 * DtNetSocket(int port, DtInetBroadcastOfDevice(char *device));
 */
/* example device names are "ec0" and "le0" */
```

NetStructs.h

```

#include "chooseDefines.h"
#include "vlTime.h"
#include "NetTypes.h"
#include "netTimeStamp.h"
#include "netBoolean.h"
#include "globalObjDes.h"

// For backwards compatibility
#include "oldTStamp.h"

#ifdef __cplusplus
#include "mathTypes.h"
#endif

typedef DtNetU8      DtNet8Bits;
typedef DtNetU8      DtNet16Bits[2];
typedef DtNetU8      DtNet32Bits[4];
typedef DtNetU8      DtNet64Bits[8];

#ifdef __cplusplus

#if DT_LITTLE_ENDIAN
class DtEventId;
#endif

class DtNetEntityMarking
{
public:
    DtNetU8      DtCharacterSet;
#define DtMaxEntityMarkingLength 11
    DtNetU8      DtText[DtMaxEntityMarkingLength];
};

class DtNetEntityType
{
public:
    DtNetU8DtEntityKind;
    DtNetU8DtDomain;
    DtNetU16DtCountry;
    DtNetU8DtCategory;
    DtNetU8DtSubCategory;
    DtNetU8DtSpecific;
    DtNetU8DtExtra;

#if DT_LITTLE_ENDIAN
    DtEntityType operator = (DtEntityType);
    DtNetEntityType operator = (DtNetEntityType);
#endif
};

class DtNetRadioEntityType
{
public:
    DtNetU8      kind;
    DtNetU8      domain;
    DtNetU16     country;
    DtNetU8      category;
    DtNetU8      nomenclatureVersion;
    DtNetU16     nomenclature;

#if DT_LITTLE_ENDIAN
    DtRadioEntityType operator = (DtRadioEntityType);
    DtNetRadioEntityType operator = (DtNetRadioEntityType);
#endif
}

```

```
};

class DtNetBurstDescriptor
{
public:
    DtNetEntityType DtMunition;
    DtNetU16DtWarhead;
    DtNetU16DtFuze;
    DtNetU16 DtQuantity;
    DtNetU16 DtRate;

#ifdef DT_LITTLE_ENDIAN
    DtBurstDescriptor operator = (DtBurstDescriptor);
    DtNetBurstDescriptor operator = (DtNetBurstDescriptor);
#endif
};

class DtNetSimulationAddress
{
public:
    DtNetU16DtSite;
    DtNetU16 DtHost;

#ifdef DT_LITTLE_ENDIAN
    DtSimulationAddress operator = (DtSimulationAddress);
    DtNetSimulationAddress operator = (DtNetSimulationAddress);
#endif
};

class DtNetEntityIdentifier
{
public:
    DtNetSimulationAddressDtSimulator;
    DtNetU16DtEntity;

#ifdef DT_LITTLE_ENDIAN
    DtEntityIdentifier operator = (DtEntityIdentifier);
    DtNetEntityIdentifier operator = (DtNetEntityIdentifier);
#endif
};

#ifdef DtHLA
typedef DtNetU32 DtNetObjectId;
#else
typedef DtNetObjectId DtNetGlobalObjectDesignator;
#endif

// This class is basically a hack so that the simple encoders and
// decoders will work for DtGlobalObjectDesignators. You can cast
// a char* to a DtNetGlobalObjectDesignator* (and vice versa), and
// you can assign a char* to a DtNetGlobalObjectDesignator. It probably
// shouldn't be used for anything else.
#define DtMAX_OBJECT_NAME_LENGTH 64
class DtNetGlobalObjectDesignator
{
public:
    DtNetGlobalObjectDesignator& operator=(
        const DtGlobalObjectDesignator& des);
    operator DtGlobalObjectDesignator() const;

protected:
    char myRep[DtMAX_OBJECT_NAME_LENGTH];
};

#ifdef DtRTI103
#else // !DtHLA

typedef DtNetEntityIdentifier DtNetObjectId;
typedef DtNetEntityIdentifier DtNetGlobalObjectDesignator;
```

```

#endif // !DtHLA

#if DtHLA

class DtNetFederateId
{
public:
    DtNetU16 site;
    DtNetU16 app;

#if DT_LITTLE_ENDIAN
    DtFederateId operator = (DtFederateId);
    DtNetFederateId operator = (DtNetFederateId);
#endif
};

class DtNetEventId
{
public:
    DtNetEventId();

    DtNetU16 eventNum;
    DtNetGlobalObjectDesignator objectId;

protected:
    // Copy ctor and assignment operator not implemented,
    // since we don't want to change
    // size when necessary. (NetTypes are sometimes used in
    // structures, where we count on their size.)
    DtNetEventId(const DtNetEventId& orig);
    DtNetEventId operator= (const DtNetEventId& orig);
};

#endif // DtHLA

#if !DtHLA

class DtNetEventId
{
public:
    DtNetSimulationAddress    DtSimulator;
    DtNetU16                  DtEvent;

#if DT_LITTLE_ENDIAN
    DtEventId operator = (DtEventId);
    DtNetEventId operator = (DtNetEventId);
#endif
};

#endif // !DtHLA

#if DT_LITTLE_ENDIAN
class DtNet64Vector
{
public:
    DtNet64Vector()
    {rep[0] = rep[1] = rep[2] = 0.0;}
    DtNet64Vector(const DtVector &vec)
    {rep[0] = vec[0]; rep[1] = vec[1]; rep[2] = vec[2];}
    DtVector operator = (const DtVector &vec)
    {rep[0] = vec[0]; rep[1] = vec[1]; rep[2] = vec[2]; return vec;}
    operator DtVector() const {return DtVector(rep[0], rep[1], rep[2]);}
    DtNetFloat64 &operator[](int i) { return rep[i];}
    DtNetFloat64 operator[](int i) const { return rep[i];}
private:

```

```
    DtNetFloat64 rep[3];
};
#else
typedef DtVector DtNet64Vector;
#endif

class DtNet32Vector
{
public:
    DtNet32Vector()
    {rep[0] = rep[1] = rep[2] = (float)0.0;}
    DtNet32Vector(const DtVector &vec)
    {rep[0] = (float)vec[0]; rep[1] = (float)vec[1]; rep[2] =
(float)vec[2];}
    DtVector operator = (const DtVector &vec)
    {rep[0] = (float)vec[0]; rep[1] = (float)vec[1]; rep[2] = (float)vec[2];
return vec;}
    operator DtVector() const {return DtVector(rep[0], rep[1], rep[2]);}
    DtNetFloat32 &operator[](int i) { return rep[i];}
    DtNetFloat32 operator[](int i) const { return rep[i];}
private:
    DtNetFloat32 rep[3];
};

class DtNetEulerAngles
{
public:
    DtNetEulerAngles() { psi = theta = phi = (float)0.0;}
    DtNetEulerAngles(const DtTaitBryan &tait)
    {psi = (DtFloat32)tait.psi(); theta = (DtFloat32)tait.theta(); phi =
(DtFloat32)tait.phi();}
    DtTaitBryan operator = (const DtTaitBryan &tait)
    {psi = (DtFloat32)tait.psi(); theta = (DtFloat32)tait.theta(); phi =
(DtFloat32)tait.phi(); return tait;}
    operator DtTaitBryan() const {return DtTaitBryan(psi, theta, phi);}
    DtNetFloat32 psi; /* Counter clockwise about z */
    DtNetFloat32 theta; /* Counter clockwise about y */
    DtNetFloat32 phi; /* Counter clockwise about x */
};

#elif !DT_LITTLE_ENDIAN

/*****
 * On Big Endian Machines, C can include these definitions:
 *****/

typedef struct DtNetRadioEntityType
{
    DtNetU8kind;
    DtNetU8domain;
    DtNetU16country;
    DtNetU8category;
    DtNetU8    nomenclatureVersion;
    DtNetU16nomenclature;
} DtNetRadioEntityType;

typedef struct DtNetEntityType
{
    DtNetU8DtEntityKind;
    DtNetU8DtDomain;
    DtNetU16DtCountry;
    DtNetU8DtCategory;
    DtNetU8DtSubCategory;
    DtNetU8DtSpecific;
    DtNetU8DtExtra;
} DtNetEntityType;
```

```

typedef struct DtNetBurstDescriptor
{
    DtNetEntityType DtMunition;
    DtNetU16DtWarhead;
    DtNetU16DtFuze;
    DtNetU16 DtQuantity;
    DtNetU16 DtRate;
} DtNetBurstDescriptor;

typedef struct DtNetSimulationAddress
{
    DtNetU16DtSite;
    DtNetU16 DtHost;
} DtNetSimulationAddress;

typedef struct DtNetEntityIdentifier
{
    DtNetSimulationAddressDtSimulator;
    DtNetU16DtEntity;
} DtNetEntityIdentifier;

typedef struct DtNetEventId
{
    DtNetSimulationAddressDtSimulator;
    DtNetU16 DtEvent;
} DtNetEventId;

typedef struct DtNetEulerAngles
{
    DtNetFloat32    psi;/* Counter clockwise about z */
    DtNetFloat32    theta;/* Counter clockwise about y */
    DtNetFloat32    phi;/* Counter clockwise about x */
} DtNetEulerAngles;

typedef DtNetFloat32 DtNet32Vector[3];
typedef DtNetFloat64 DtNet64Vector[3];

#ifdef
#error Must have C++ to use NetStructs.h on little endian machine.
#endif

/*****
These structures don't need to be C++ classes even on little endian
machines.
*****/

#ifdef DtHLA

#ifdef DtRPR016

typedef struct DtNetArtParamRecord
{
    DtNetEnum      partType;
    DtNetU8        change;
    DtNetU16       partAttachedTo;
    DtNetU32       paramType;
    DtNetFloat32   value;
} DtNetArtParamRecord;

#else // !DtRPR016

typedef struct DtNetArtParamRecord
{
#ifdef DtDISVERS != 3
    DtNetU8        artOrAttached;
    DtNetU8        DtChange;
#else

```

```
        DtNetU16      DtChange;
    #endif
        DtNetU16      DtAttachedTo;
        DtNetU32      DtType;
        DtNet64Bits   DtValue;
    } DtNetArtParamRecord;

    #endif // DtRPR016

    #endif // DtHLA

    #if !DtHLA

    typedef struct DtNetArtParamRecord
    {
    #if DtDISVERS != 3
        DtNetU8      artOrAttached;
        DtNetU8      DtChange;
    #else
        DtNetU16     DtChange;
    #endif
        DtNetU16     DtAttachedTo;
        DtNetU32     DtType;
        DtNet64Bits  DtValue;
    } DtNetArtParamRecord;

    #endif // !DtHLA

    #if DtHLA

    typedef DtNetArtParamRecord DtArticulatedParts;
    typedef DtNetArtParamRecord DtNetArtPartRecord;

    #endif // DtHLA

    #if !DtHLA

    typedef DtNetArtParamRecord DtArticulatedParts;
    typedef DtNetArtParamRecord DtNetArtPartRecord;

    #endif // !DtHLA

    typedef struct DtNetAttachedPart
    {
    #if DtDISVERS != 3
        DtNetU8      myArtOrAttached;
        DtNetU8      myChange;
    #else
        DtNetU16     myChange;
    #endif
        DtNetU16     myAttachedTo;
        DtNetU32     myStation;
        DtNetEntityType myType;
    } DtNetAttachedPart;

    typedef struct DtNetAngularVelocity
    {
        DtNetFloat32 wx;
        DtNetFloat32 wy;
        DtNetFloat32 wz;
    } DtNetAngularVelocity;

    typedef struct DtNetLinearVelocity
    {
        DtNetFloat32 DtX;
        DtNetFloat32 DtY;
        DtNetFloat32 DtZ;
```



```
} DtNetLinearVelocity;

typedef struct DtNetWorldCoordinates
{
    DtNetFloat64 DtX;
    DtNetFloat64 DtY;
    DtNetFloat64 DtZ;
} DtNetWorldCoordinates;

typedef struct DtNetEntityCoordinates
{
    DtNetFloat32 DtX;
    DtNetFloat32 DtY;
    DtNetFloat32 DtZ;
} DtNetEntityCoordinates;

typedef struct DtNetTopoUpCoord
{
    DtNetFloat32 East;
    DtNetFloat32 North;
    DtNetFloat32 Up;
} DtNetTopoUpCoord;

typedef struct
{
    DtNetInt32hour;
    DtNetU32hourFrac;
} DtNetClockTime;

#ifdef __cplusplus

DtTime DtClockTime2AbsTime(const DtNetClockTime *clock);
void DtAbsTime2ClockTime(DtTime absTime, DtNetClockTime *clock);

#endif
```

netTimeStamp.h

```
#include "vlTime.h"
#include "tStampType.h"
#include "NetTypes.h"

// Host representation of a timeStamp
typedef DtU32 DtTimeStamp;

// class DtNetTimeStamp:
//
// Instances of DtNetTimeStamp are network representations of a
// DIS time stamp. This class has NO VIRTUAL FUNCTIONS, since it
// is a DtNetType - whose exact byte layout is relied on by
// PDU classes, etc.

class DtNetTimeStamp
{
public:
    // default constructor - creates an instance with a time of zero
    // and timestamp type of relative.
    DtNetTimeStamp();

    // Creates an instance with the time set to time (specified in
    // seconds) modulo an hour, and the timestamp type set to type.
    DtNetTimeStamp(DtTime time,
        DtTimeStampType type = DtTimeStampRelative);

    // destructor
    ~DtNetTimeStamp();

    // copy constructor
    DtNetTimeStamp(const DtNetTimeStamp& orig);

    // assignment operator
    DtNetTimeStamp& operator=(const DtNetTimeStamp& orig);

    // Cast to a DtU32 representing the whole timestamp
    // Useful mainly for backwards compatibility for applications
    // that used to cast DtNetTimeStamp to DtU32 before accessing it.
    operator DtU32 () const;

    // Get the time contained in the timestamp (in seconds).
    // The value returned will be between 0.0 and 3600.0 seconds.
    DtTime time() const;

    // Get the time contained in the timestamp to time % an hour.
    // time is specified in seconds.
    void setTime(DtTime time);

    // Set/Get timestamp type
    void setType(DtTimeStampType type);
    DtTimeStampType type() const;

protected:
    DtNetU32 myRep;

protected:
    static double theHourFracPerHour;
    static double theHourPerHourFrac;
};
```

NetTypes.h

```

#ifndef _NetTypes
#define _NetTypes

#include "vlconfig.h"

#include "MachineTypes.h"

typedef DtU8DtNetU8;
typedef DtInt8DtNetInt8;

#if ! DT_LITTLE_ENDIAN

/*****
 * Big Endian Machines.
 *****/

typedef DtU16DtNetU16;
typedef DtU32DtNetU32;
typedef DtInt16DtNetInt16;
typedef DtInt32DtNetInt32;
typedef DtFloat32DtNetFloat32;
typedef DtFloat64DtNetFloat64;

#else

/*****
 *
 * Little Endian (i.e., Brain-dead) Machines.
 *
 * For example, the Intel 80x86 series.
 *****/

/*
 * Network-order unsigned 16-bit quantity.
 */

class DtNetU16
{
public:
    DtNetU8msbyte; /* Most Significant Byte */
    DtNetU8lsbyte; /* Least Significant Byte */

    DtNetU16();
    DtNetU16(DtU16);
    DtNetU16(const DtNetU16&);
    DtU16 operator = (DtU16);
    operator DtU16() const;
};

/*
 * Network-order signed 16-bit quantity.
 */

class DtNetInt16
{
public:
    DtNetInt8msbyte;
    DtNetInt8lsbyte;

    DtNetInt16();
    DtNetInt16(DtInt16);

```

```
DtNetInt16(const DtNetInt16&);
DtInt16 operator = (DtInt16);
operator DtInt16() const;
};

/*
 * Network-order unsigned 32-bit quantity.
 */

class DtNetU32
{
public:

    DtNetU16msword; /* Most Significant Word */
    DtNetU16lsword; /* Least Significant Word */

    DtNetU32();
    DtNetU32(DtU32);
    DtNetU32(const DtNetU32&);
    DtU32 operator = (DtU32);
    operator DtU32() const;
};

/*
 * Network-order signed 32-bit quantity.
 */

class DtNetInt32
{
public:

    DtNetInt16msword; /* Most Significant Word */
    DtNetInt16lsword; /* Least Significant Word */

    DtNetInt32();
    DtNetInt32(DtInt32);
    DtNetInt32(const DtNetInt32&);
    DtInt32 operator = (DtInt32);
    operator DtInt32() const;
};

/*
 * Network-order 32-bit IEEE-754 floating-point quantity.
 */

class DtNetFloat32
{
public:

    DtNetU32bits;

    DtNetFloat32();
    DtNetFloat32(DtFloat32);
    DtNetFloat32(const DtNetFloat32&);
    DtFloat32 operator = (DtFloat32);
    operator DtFloat32() const;
};

/*
 * Network-order 64-bit IEEE-754 floating-point quantity.
 */

class DtNetFloat64
{
public:

    DtNetU32mslong; /* Most Significant Long */
```

```
DtNetU32lslong; /* Least Significant Least */

DtNetFloat64();
DtNetFloat64(DtFloat64);
DtNetFloat64(const DtNetFloat64&);
DtFloat64 operator = (DtFloat64);
operator DtFloat64() const;
};

#endif

/*****
 * You should probably use either of the two above schemes.
 *****/
```

objClassDesc.h

```
#if DtHLA

#include "RTI.hh"
#include "MachineTypes.h"

#include "chooseDefines.h"

// class DtObjClassDesc:
//
// Instances of DtObjClassDesc are used to represent
// HLA Object classes

class DtObjClassDesc
{
public:
    // Constructor. Adds the base class' attributes to our attribute set.
    DtObjClassDesc(const char* className,
        DtObjClassDesc* base,
        RTI::ObjectClassHandle handle,
        DtExerciseConn* exConn);

    // destructor
    virtual ~DtObjClassDesc();

    // Add an attribute (Used when reading FED file).
    virtual void addAttr(const char* attrName);

    // Get the object's class handle.
    virtual RTI::ObjectClassHandle handle() const;

    // Get the object's class handle.
    virtual char* name() const;

    // Return a pointer to the base class's descriptor, NULL if none
    virtual const DtObjClassDesc* baseClass() const;

    // Returns 1 if the class is of class hand or a subclass of hand,
    // 0 otherwise
    virtual int isOfClass(RTI::ObjectClassHandle hand) const;

    // Get the number of attributes associated with this object,
    // including privilege to delete (usually handle 0).
    virtual int numAttributes() const;

    // Get the set of valid attribute handles for this class. Includes
    // attributes inherited from base classes.
    virtual const RTI::AttributeHandleSet& attrHandleSet() const;

    // Get the set of non-inherited attribute handles for this class.
    virtual const RTI::AttributeHandleSet& ownAttrHandles() const;

#if DtRTI103
    // returns the set of attributes that are currently needed by the
    // federation, i.e. those that have been specified in startUpdates
    // calls, but not those that have been specified in stopUpdates.
    virtual const RTI::AttributeHandleSet& attributesNeededByFederation();
#else
    // returns DtTrue if the class is currently needed by the
    // federation, i.e. we've received a startRegistrationForObjectClass
    // call for the class.
    virtual DtBoolean classNeededByFederation();
#endif
#endif
```

```

// Ensures that the class published, and that start and stopUpdates
// callbacks are registered. If hSet is specified,
// publish those attributes, otherwise publish all of the class's attrs.
virtual void setupClassForUpdating();
virtual void setupClassForUpdating(const RTI::AttributeHandleSet& hSet);

// Subscribe to the class. If hSet is specified,
// ask for those attributes, otherwise ask for all of the class's attrs.
virtual void registerInterest(const RTI::AttributeHandleSet& hSet);
virtual void registerInterest();

// Unsubscribe to the class
virtual void deregisterInterest();

// Request updates of all objects of this class. If hSet is specified,
// ask for those attributes, otherwise ask for all of the class's attrs.
virtual void requestUpdates();
virtual void requestUpdates(const RTI::AttributeHandleSet& hSet);

// Get my RTI ambassador
virtual DtVrlRtiAmbassador* rtiAmb();

// Get my federate ambassador
virtual DtVrlFederateAmbassador* fedAmb();

public:
    static int maxAttributes();
    static void setMaxAttributes(int num);

protected:
#if DtRTI103
    // Callback for handling startUpdates RTI callback
    static void startUpdatesCb(RTI::ObjectClassHandle h,
        const RTI::AttributeHandleSet& attributeSet,
        void* usr);

    // Callback for handling stopUpdates RTI callback
    static void stopUpdatesCb(RTI::ObjectClassHandle h,
        const RTI::AttributeHandleSet& attributeSet,
        void* usr);
#else
    // Callback for handling startRegistration RTI callback
    static void startRegCb(RTI::ObjectClassHandle h,
        void* usr);

    // Callback for handling stopRegistration RTI callback
    static void stopRegCb(RTI::ObjectClassHandle h,
        void* usr);
#endif

protected:
#if DtRTI103
    // Callback for handling startUpdates RTI callback
    virtual void processStartUpdates(RTI::ObjectClassHandle h,
        const RTI::AttributeHandleSet& attributeSet);

    // Callback for handling stopUpdates RTI callback
    virtual void processStopUpdates(RTI::ObjectClassHandle h,
        const RTI::AttributeHandleSet& attributeSet);
#else
    // Callback for handling startRegistration RTI callback

```

```
    virtual void processStartReg(RTI::ObjectClassHandle h);

    // Callback for handling stopRegistration RTI callback
    virtual void processStopReg(RTI::ObjectClassHandle h);
#endif

protected:

    char* myName;
    RTI::ObjectClassHandle myHandle;
    RTI::AttributeHandleSet* myAttrs;
    RTI::AttributeHandleSet* myOwnAttrs;
#if DtRTI103
    RTI::AttributeHandleSet* myNeededAttrs;
#else
    DtBoolean myClassNeeded;
#endif
    DtObjClassDesc* myBase;
    DtExerciseConn* myExConn;
    DtBoolean myPublishedFlag;

protected:

    static int theMaxAttrs;
};
#if DtRTI103
#else
#endif
#endif
```


objDesList.h

```

// class DtGlobalObjectDesignatorList:
//
// Instances of DtGlobalObjectDesignatorList are list of globally
// significant object identifiers.

#include "hostStructs.h"

class DtGlobalObjectDesignatorList
{
public:
    // default constructor
    DtGlobalObjectDesignatorList();

    // destructor
    virtual ~DtGlobalObjectDesignatorList();

    // copy constructor
    DtGlobalObjectDesignatorList(const DtGlobalObjectDesignatorList& orig);

    // assignment operator
    DtGlobalObjectDesignatorList& operator=(
        const DtGlobalObjectDesignatorList& orig);

    // equivlance operators
    int operator==(const DtGlobalObjectDesignatorList& other) const;
    int operator!=(const DtGlobalObjectDesignatorList& other) const;

    // Set/Get number of objects. Setting to a number lower than
    // the current number will remove trailing objects from the list.
    virtual void setNumObjects(int num);
    virtual int numObjects() const;

    // Set index'th object designator.
    virtual DtBoolean setObject(int index,
        const DtGlobalObjectDesignator& des);

    // Get index'th object designator. Set valid to DtFalse if bad index,
    // DtTrue otherwise.
    virtual const DtGlobalObjectDesignator& object(
        int index, DtBoolean* valid = NULL) const;

    // Print data.
    virtual void print() const;

    // Set from network representation.
    virtual void setFromNet(const void* netRep, int length);

    // Returns network representation, fills size with size of netRep.
    // Return value is pointer to internal object with same lifetime as this.
    // Do not delete.
    virtual const void* netRep(int* size = NULL) const;

    // Returns size of network representation.
    virtual int netSize() const;

protected:
    int myNum;
    DtGlobalObjectDesignator* myObjs;
};

```

objectType.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/objectTypeP.h"

// class DtObjectType:
//
// DtObjectType is a class which contains information identifying
// the type of a synthetic environment object.

class DtObjectType
{
public:

    // Constructors
    DtObjectType();
    DtObjectType(const DtObjectType& orig);

    // Assignment operator
    DtObjectType& operator=(const DtObjectType& orig);
    DtObjectType& operator=(const DtNetObjectType& orig);

    // Destructor
    virtual ~DtObjectType();

    // Set/Get the domain
    virtual void setDomain(DtU8 value);
    virtual DtU8 domain() const;

    // Set/Get the kind
    virtual void setKind(DtU8 value);
    virtual DtU8 kind() const;

    // Set/Get the category
    virtual void setCategory(DtU8 value);
    virtual DtU8 category() const;

    // Set/Get the subcategory
    virtual void setSubcategory(DtU8 value);
    virtual DtU8 subcategory() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(const DtNetObjectType& record);
    virtual void loadNetRecord(DtNetObjectType& record) const;

protected:

    // Data members:
    DtU8          myDomain;
    DtU8          myKind;
    DtU8          myCategory;
    DtU8          mySubcategory;
};
#endif
#endif
```

objIdHashKey.h

```
#include "baseHashKey.h"
#include "hostStructs.h"

// class DtObjectIdHashKey:
//
// Instances of DtObjectIdHashKey are hash keys made from DtObjectIds.
// They can be used with DtBaseHashList to store arbitrary pieces of
// data hashed by DtObjectId.

class DtObjectIdHashKey : public DtBaseHashKey
{
public:
    // default constructor
    DtObjectIdHashKey(const DtObjectId& id);

    // destructor
    virtual ~DtObjectIdHashKey();

    // copy constructor
    DtObjectIdHashKey(const DtObjectIdHashKey& orig);

    // assignment operator
    DtObjectIdHashKey& operator=(const DtObjectIdHashKey& orig);

    // make a copy of this hash key
    virtual DtBaseHashKey* clone() const;

protected:
    // get the string representation of this key
    virtual const DtString& stringRep() const;

    // compute the hashed value for this key
    virtual DtU32 computeHashValue();

protected:
    DtObjectId myId;
    DtString myStr;
};
```

objIdHashLst.h

```
#include "hashlist.h"
#include "hostStructs.h"

// class DtObjectIdHashlist:
//
// Instances of DtObjectIdHashlist are hashlists, where you can
// store associations between DtObjectIds and arbitrary pieces of data.

class DtObjectIdHashlist
{
public:
    // default constructor. Pass DtTrue only if the data that you are
    listing
    // are instances of a class derived from DtListedObject.
    DtObjectIdHashlist(DtBoolean intrusive = DtFalse);

    // destructor
    virtual ~DtObjectIdHashlist();

    // In the event that an item with key already exists, then add
    // will replace the data and move it to the location in list.
    // If your data was allocated, do not add unless you know key is
    // unique, or a memory leak will result.

    // Add an association between key and data to the list
    virtual DtListItem* add(const DtObjectId& key, void* data);

    // Remove the current association involving key from the list.
    // Returns the data that was associated with key.
    virtual void* remove(const DtObjectId& key);

    // Lookup the data associated with key.
    virtual void* lookup(const DtObjectId& key) const;
    virtual void* operator[](const DtObjectId& key) const;

    // Get a DtList containing all of the void* data that you
    // have listed, regardless of keys.
    virtual const DtList* list() const;

protected:
    // copy constructor
    DtObjectIdHashlist(const DtObjectIdHashlist& orig);

    // assignment operator
    DtObjectIdHashlist& operator=(const DtObjectIdHashlist& orig);

protected:
    DtList* myList;
    DtHashlist* myHashlist;
};
typedef DtObjectIdHashlist DtObjectIdHashList;
```

objRefCbInfo.h

```

#if DtHLA

#include "RTI.hh"
#include "stateMsg.h"

#include "chooseDefines.h"

// Object reflect attribute values callback signature.
typedef void (*DtObjectReflectAttributeValuesCb)(
    const DtStateMsg& stateMsg,
    DtHlaObject* obj,
    void* usr);

// class DtObjectReflectAttributeValuesCbInfo:
//
// Instances of DtReflectAttributeValuesCbInfo are used for providing
// a callback when attributes change for an object in the federation.

class DtObjectReflectAttributeValuesCbInfo
{
public:
    // constructor
    DtObjectReflectAttributeValuesCbInfo(
        DtObjectReflectAttributeValuesCb cb,
        void* usr);

    // destructor
    virtual ~DtObjectReflectAttributeValuesCbInfo();

    // copy constructor
    DtObjectReflectAttributeValuesCbInfo(
        const DtObjectReflectAttributeValuesCbInfo& orig);

    // assignment operator
    DtObjectReflectAttributeValuesCbInfo& operator=(
        const DtObjectReflectAttributeValuesCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(
        const DtObjectReflectAttributeValuesCbInfo* const other) const;

    // Check whether one callback info is equal to another.
    virtual int operator==(
        const DtObjectReflectAttributeValuesCbInfo& other) const;

    // Get the callback function.
    virtual const DtObjectReflectAttributeValuesCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:
    DtObjectReflectAttributeValuesCb myFunc;
    void* myUsr;
};
#endif

```

pairSock.h

```
#include "Socket.h"

class DtSocketPairSocket : public DtSocket
{
public:
    DtSocketPairSocket();
    DtSocketPairSocket(int sockNum);

    int sendTo(caddr_t packet, size_t size, DtInetAddr addr);
    int recvWithStatus(caddr_t buffptr, size_t size,
        int *status=NULL);
    int DtRecv(caddr_t *buffptr, int *status);
    int DtRecv(caddr_t buff, size_t size);
    void DtBecomeNonBlocking(void);

    int otherFd() { return ofd; }
    int readFd() { return myFd; }
    int writeFd() { return myFd; }

private:
    int myFd;
    int ofd;
    char buffer[1500];
};
```

pdu.h

```

#include <stdlib.h>
#include "packets/pduHeader.h"
#include "disEnums.h"
#include "tStampType.h"
#include "baseMsg.h"

#define DtSTATUS_OK 0
#define DtSTATUS_NOT_OK 1

extern int DtProtocolVersionToSend;
extern int DtProtocolVersionToRecvMin;
extern int DtProtocolVersionToRecvMax;

class DtPdu : public DtBaseMsg
{
public:
    // Destructor
    virtual ~DtPdu();

    // Copy constructor
    DtPdu(const DtPdu &pdu,
          DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Assignment operator
    DtPdu &operator=(const DtPdu &pdu);

    // Older function that returns a pointer to the network representation.
    // The newer DtBaseMsg::netRep should generally be used instead.
    virtual void* packet() const;

    // Get PDU status. DtSTATUS_OK means it's good. Other values
    // indicate a bad PDU. Return codes are derived PDU-dependent.
    virtual int status() const;

    // Set/Get protocol version from PDU header.
    virtual void setVersion(int vers);
    virtual int protocolVersion() const;

    // Set/Get exerciseID in PDU header.
    virtual void setExerciseId(int id);
    virtual int exerciseId() const;

    // Returns PDU kind from PDU header
    virtual DtPduKind kind() const;

#ifdef DtDISVERS != 3
    // Returns protocol family from PDU header.
    virtual DtProtocolFamily protocolFamily() const;
#endif

    // Set the PDU header's time stamp to reflect time and type.
    // Time is in seconds, but only the time modulo an hour
    // actually gets into the time stamp.
    virtual void setTimeStamp(DtTime time,
                             DtTimeStampType type = DtTimeStampRelative);

    // Return the time (in seconds past an hour) encoded in the
    // PDU header's time stamp.
    virtual DtTime timeStamp() const;

    // Return the type of the time stamp, as indicated in PDU header.
    virtual DtTimeStampType timeStampType() const;

```

```
// Since timestamp only represents number of seconds past the hour,
// guessTimeValid chooses the hour that would make the timestamp
// closest to reference time, and returns a time that is timestamp
// seconds past that hour.
virtual DtTime guessTimeValid(DtTime referenceTime) const;

// Print the contents of the PDU by calling printHeader
// and printData.
virtual void print() const;

// Print the PDU's header.
virtual void printHeader() const;

// Print the PDU's data (not including header).
// Base DtPdu version just prints hex.
virtual void printData() const;

protected:

    // Constructor.
    // PDU is not fully constructed until initPdu is called.
    DtPdu();

    // Sets the PDU's status to okStatus.
    virtual void setStatus(int okStatus);

    // Returns a pointer to the PDU's header.
    virtual DtNetPduHeader* netHeader() const;

    // Must be called after default ctor before PDU is completely
    // constructed. Fills in PDU header, and initializes DtBaseMsg.
    virtual void initPdu(int size, DtBufferPtr buffer);

    // Called by from-net-rep ctor. Initializes DtBaseMsg.
    virtual void initPdu(const void* initial, DtBufferPtr buffer);

    // Returns the PDU kind associated with the derived PDU class.
    virtual DtPduKind internalGetPduKind() const = 0;

#if DtDISVERS != 3
    // Maps PDU kind to ProtocolFamily.
    virtual DtProtocolFamily kindToFamily(DtPduKind kind) const;
#endif

    // Called by changeSize. Sets the size in the header to size.
    virtual void sizeChanged(int size);

    // zeros out all bytes after the PDU header
    virtual void clearData();

    // Initializes PDU header. Called by initPdu.
    virtual void initHeader(DtPduKind kind, int size);

protected:

    int myOKStatus;
};
#define netPdu packet()

#if DtDISVERS != 3
#endif
```


pduCbInfo.h

```

#if !DtHLA

#include "pduCbMgr.h"

// class DtPduCallbackInfo:
//
// Instances of DtPduCallbackInfo are used by DtPduCallbackManager
// to keep track of PDU callback functions.

class DtPduCallbackInfo
{
public:
    // constructor
    DtPduCallbackInfo(DtPduCallbackFcn func, void* usr);

    // destructor
    virtual ~DtPduCallbackInfo();

    // copy constructor
    DtPduCallbackInfo(const DtPduCallbackInfo& orig);

    // assignment operator
    DtPduCallbackInfo& operator=(const DtPduCallbackInfo& orig);

    // Get whether one callback info equals another.
    virtual int match(const DtPduCallbackInfo& other) const;

    // Get the callback function.
    virtual DtPduCallbackFcn func() const;

    // Get the user data.
    virtual void* usr() const;

protected:
    DtPduCallbackFcn myFunc;
    void* myUsr;
};
#endif

```

pduCbMgr.h

```
#if !DtHLA

#include "vllist.h"
#include "pdu.h"

// PDU callback function signature.
typedef void (*DtPduCallbackFcn)(DtPdu* pdu, void* usr);

// class DtPduCallbackManager:
//
// Instances of DtPduCallbackManager are used to manage callbacks
// on the PDU kinds for the exercise connection.

class DtPduCallbackManager
{
public:
    // default constructor
    DtPduCallbackManager();

    // destructor
    virtual ~DtPduCallbackManager();

    // copy constructor
    DtPduCallbackManager(const DtPduCallbackManager& orig);

    // assignment operator
    DtPduCallbackManager& operator=(const DtPduCallbackManager& orig);

    // Add/Remove a function to be called when a particular PDU kind
    // is received.
    virtual void addPduCallback(DtPduKind k,
        DtPduCallbackFcn fcn, void* usr);
    virtual void removePduCallback(DtPduKind k,
        DtPduCallbackFcn fcn, void* usr);

    // Call the callbacks registered for the PDU's kind.
    virtual void processPdu(const DtPdu& pdu) const;

    // Get the number of callbacks registered for the PDU kind.
    virtual int numCallbacks(DtPduKind k) const;

protected:
    // Add/Remove callback info for a particular PDU kind.
    virtual void addPduCallback(DtPduKind k, DtPduCallbackInfo* info);
    virtual void removePduCallback(DtPduKind k, const DtPduCallbackInfo&
info);

    // Empty the list.
    // Non-virtual since called in dtor.
    void emptyList();

    // Copy the list.
    // Non-virtual since called in ctor.
    void copyList(const DtPduCallbackManager& orig);

protected:
#define DtMAX_PDU_KINDS 255
    DtList myCbList[DtMAX_PDU_KINDS];
};
#endif
```

pduDatRecSet.h

```

#if !DtHLA

#include "datumRecSet.h"

// An instance of this class is a DtDatumRecordSet that uses a part of
// a DtPduWithData as its network representation. Normally, a
// DtDatumRecord manages the memory associated with its network representa-
// tion
// through its DtBaseMsg base class. However, since a DtPduWithData's
// netRep memory is managed by that DtPduWithData, DtPduDatumRecordSets must
// defer ITS memory management responsibilities to the DtPduWithData.
// This is accomplished by overriding changeSize, addBytes and deleteBytes
// here, with versions that call the DtPduWithData's versions of the func-
// tions.
// No one should need to use this class, except for DtPduWithData.

class DtPduDatumRecordSet : public DtDatumRecordSet
{
public:
    // Constructor
    DtPduDatumRecordSet(DtPduWithData* pdu);

    // Destructor
    virtual ~DtPduDatumRecordSet();

    // Copy ctor
    DtPduDatumRecordSet(const DtPduDatumRecordSet& orig);

    // Assignment operator
    DtPduDatumRecordSet& operator=(const DtPduDatumRecordSet& orig);

protected:
    // Changes the size of the datumRecSet, by asking my PDU to change
    // size, then reinitializing to use the right piece of the PDU's
    // buffer.
    virtual void changeSize(int size);

    // Inserts numBytes zeroed-out bytes at the indicated offset
    // into the net rep, by asking my PDU to insert bytes, then
    // reinitializing to use the right piece of the PDU's buffer.
    virtual void insertBytes(int offset, int numBytes);

    // Inserts numBytes zeroed-out bytes at the indicated addr
    // within the net rep, by asking my PDU to insert bytes, then
    // reinitializing to use the right piece of the PDU's buffer.
    virtual void insertBytes(void* addr, int numBytes);

    // Deletes numBytes bytes at the indicated offset in the net rep,
    // by asking my PDU to insert bytes, then reinitializing to use
    // the right piece of the PDU's buffer.
    virtual void deleteBytes(int offset, int numBytes);

    // Deletes numBytes bytes at the indicated addr within the net rep,
    // by asking my PDU to insert bytes, then reinitializing to use
    // the right piece of the PDU's buffer.
    virtual void deleteBytes(void* addr, int numBytes);

protected:
    DtPduWithData* myPdu;
};
#endif

```

pduFactory.h

```
#if !DtHLA

#include "pdu.h"

// Pdu creation function signature.
typedef DtPdu* (*DtPduCreator)(const DtNetPduHeader* packet,
    DtBufferPtr buffer);

// class DtPduFactory:
//
// Instances of DtPduFactory are used to create PDUs (protocol data
// units) or packets, typically for DIS (as opposed to HLA).

class DtPduFactory
{
public:
    // default constructor
    DtPduFactory();

    // destructor
    virtual ~DtPduFactory();

    // copy constructor
    DtPduFactory(const DtPduFactory& orig);

    // assignment operator
    DtPduFactory& operator=(const DtPduFactory& orig);

    // Create the specified pdu.
    // Note: Caller is responsible for deleting.
    // Returns NULL if pdu creation function is not in the factory.
    virtual DtPdu* createPdu(const DtNetPacket* packet,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add a pdu creation function to the factory.
    // The function must return a new'ed pdu object which will
    // be deleted by the user.
    // Returns any previous pdu creation function which was
    // replaced by the add.
    virtual DtPduCreator addCreator(DtPduKind kind, DtPduCreator creator);
    // Returns whether packet is appropriate DIS version.

    virtual int packetGood(const DtNetPacket* packet) const;

public:
    static int dfltPacketGood(const DtNetPacket* packet);

protected:
    // myCreators is an array of theMaxPduKinds items.
    DtPduCreator* myCreators;

protected:
    static const int theMaxPduKinds;
};
#endif
```

pduWithData.h

```

#if !DtHLA

#include "hostStructs.h"
#include "simanPdu.h"
#include "datumIdType.h"
#include "datumRecSet.h"

#define DtDataStaticSize 40

class DtPduWithData : public DtSimanPdu
{
public:
    // Constructor
    DtPduWithData();

    // returns number of fixed fields
    virtual int numFixedFields() const;

    // returns number of variable length fields
    virtual int numVarFields() const;

    // Returns the datum param found in the index'th variable
    // datum record. Index is between 1 and numRecords. Returns
    // DtDatumParam(0) if index is out of bounds.
    virtual DtDatumParam varDatumId(int index) const;

    // Returns the datum param found in the index'th fixed
    // datum record. Index is between 1 and numRecords. Returns
    // DtDatumParam(0) if index is out of bounds.
    virtual DtDatumParam fixedDatumId(int index) const;

    // Returns the datum param found in the index'th fixed or variable
    // datum record, depending on whether fixedOrVar is DtFIXED or DtVAR.
    // Index is between 1 and numRecords. Returns
    // DtDatumParam(0) if index is out of bounds.
    virtual DtDatumParam datumId(int fixedOrVar, int index) const;

    // Returns the number of bits of data contained in the value
    // fields of the index'th fixed or variable datum record,
    // depending on whether fixedOrVar is DtFIXED or DtVAR. If DtFIXED,
    // the answer is always 32.
    virtual int datumSizeBits(int fixedOrVar, int index) const;

    // Returns the number of bytes of data contained in the value
    // fields of the index'th fixed or variable datum record,
    // depending on whether fixedOrVar is DtFIXED or DtVAR. If DtFIXED,
    // the answer is always 4.
    virtual int datumSizeBytes(int fixedOrVar, int index) const;

    // Looks through all fixed and variable records for one containing
    // param as the parameter ID. If it finds one, fills supplied fixedOrVar
    // pointer with DtFIXED or DtVAR, and index with the index to indicate
    // which record was found, and fills retcode with DtDATUM_VALID. If
param
    // was not found, retcode is filled with DtDATUM_BAD_ID, and index and
    // fixedOrVar are invalid.
    virtual void getIndex(DtDatumParam param, int *fixedOrVar, int *index,
        int *retcode = NULL) const;

    // ***** INSPECTING DATUM VALUES BY INDEX *****
    // If you know the data type of the value contained in a particular

```

```
// record, use one of the following to return it. fixedOrVar (either
DtFIXED
// or DtVAR) and index (between 1 and numRecords) indicate which record
// you are inspecting. datumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It returns a pointer to a character string containing the
// exact bytes in the PDU's network representation that represent the value
// being inspected. The supplied retcode pointer will be filled with
// DtDATUM_VALID if the value was successfully inspected, and
DtDATUM_BAD_LENGTH
// if either the index was invalid, or the size of the data indicated that
// it could not be interpreted as the chosen data type.

virtual DtFloat32 datumValFloat32(int fixedOrVar, int index,
    int *retcode = NULL) const;
virtual DtFloat64 datumValFloat64(int fixedOrVar, int index,
    int *retcode = NULL) const;
virtual unsigned long datumValUnsigned32(int fixedOrVar, int index,
    int *retcode = NULL) const;
virtual long datumValInt32(int fixedOrVar, int index,
    int *retcode = NULL) const;
virtual const char* datumValByteArray(int fixedOrVar, int index) const;

// ***** INSPECTING DATUM VALUES BY PARAM ID *****
// If you know the data type of the value contained in a particular
// record, use one of the following to return it. The particular record
// inspected will be the first one containing param as the datum ID.
// datumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It returns a pointer to a character string containing the
// exact bytes in the PDU's network representation that represent the value
// being inspected. The supplied retcode pointer will be filled with
// DtDATUM_VALID if the value was successfully inspected, and
DtDATUM_BAD_LENGTH
// if either param was not found, or the size of the data indicated that
// it could not be interpreted as the chosen data type.

virtual DtFloat32 datumValFloat32(DtDatumParam param,
    int *retcode = NULL) const;
virtual DtFloat64 datumValFloat64(DtDatumParam param,
    int *retcode = NULL) const;
virtual unsigned long datumValUnsigned32(DtDatumParam param,
    int *retcode = NULL) const;
virtual long datumValInt32(DtDatumParam param,
    int *retcode = NULL) const;
virtual const char* datumValByteArray(DtDatumParam param,
    int *retcode = NULL) const;

// ***** MUTATOR FUNCTIONS *****

// Initialize the data section of the PDU to reflect the fixed parameters
// and variable parameters and sizes indicated. fixedParams is a NULL-
// terminated array of DtDatumParams. specList is an array of DtVarDa-
tumSpecs
// terminated with the DtNullVarDatumSpec constant. This call can be
made
// instead of calls to setNumVarDatumRecords, setNumFixedDatumRecord,
// and repeated calls to setDatumParam and setVarDataBytes.
virtual void initDataSection(const DtDatumParam *fixedParams,
    const DtVarDatumSpec *specList);

// Sets the number of fixed or variable fields to newNum. Previously
// existing fixed fields with indices greater than newNum are deleted.
// If newNum exceeds the number of existing fields, trailing new fields
// are created with zeros as datum params, sizes (for var only) and
values.
virtual void setNumFixedFields(int newNum);
```

```

virtual void setNumVarFields(int newNum);

// Set the datum ID in the index'th fixed or variable datum record to
// param, depending on whether fixedOrVar is DtFIXED or DtVAR. Supplied
// retcode pointer is filled with DtDATUM_VALID if successful, or
DtDATUM_
// BAD_INDEX if the index is not valid.
virtual void setDatumParam(int fixedOrVar, int index, DtDatumParam param,
    int *retcode = NULL);

// Allocates length bytes for the index'th variable datum record.
// Should be called before using one of the setDatumVal functions
// to actually set the data. Supplied retcode pointer is set to
// DtDATUM_VALID if successful, or DtDATUM_BAD_INDEX if the index
// is not valid.
virtual void setVarDataBytes(int index, int length, int *retcode = NULL);

// ***** SETTING DATA VALUES BY INDEX *****
// Use one of the following to set the data value in a fixed or variable
datum
// record, depending on the data type of your data. fixedOrVar (either
DtFIXED
// or DtVAR) and index (between 1 and numRecords) indicate which record
// you are modifying. setDatumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It takes a pointer to a character string containing the
// exact bytes that should be used in the PDU's network representation.
// The supplied retcode pointer will be filled with
// DtDATUM_VALID if the value was successfully set, DtDATUM_BAD_INDEX if the
// index is invalid, and DtDATUM_BAD_LENGTH if the size of the data indi-
cated that
// it could not be set from as the chosen data type.

virtual void setDatumValFloat64(int fixedOrVar, int index, DtFloat64 val,
    int *retcode = NULL);
virtual void setDatumValFloat32(int fixedOrVar, int index, DtFloat32 val,
    int *retcode = NULL);
virtual void setDatumValInt32(int fixedOrVar, int index, long val,
    int *retcode = NULL);
virtual void setDatumValUnsigned32(int fixedOrVar, int index, unsigned
long val,
    int *retcode = NULL);
virtual void setDatumValByteArray(int fixedOrVar, int index, const char
*val,
    int *retcode = NULL);

// ***** SETTING DATA VALUES BY PARAM ID *****
// Use one of the following to set the data value in a fixed or variable
datum
// record, depending on the data type of your data. The particular record
modified
// will be the first one containing param as its parameter ID.
// setDatumValByteArray is used when the data type is
// a character string, or something other than a long, unsigned long, float
// or double. It takes a pointer to a character string containing the
// exact bytes that should be used in the PDU's network representation.
// The supplied retcode pointer will be filled with
// DtDATUM_VALID if the value was successfully set, DtDATUM_BAD_ID if param
// is invalid, and DtDATUM_BAD_LENGTH if the size of the data indicated that
// it could not be set from the chosen data type.

virtual void setDatumValFloat64(DtDatumParam param, DtFloat64 val,
    int *retcode = NULL);
virtual void setDatumValFloat32(DtDatumParam param, DtFloat32 val,
    int *retcode = NULL);
virtual void setDatumValInt32(DtDatumParam param, long val,
    int *retcode = NULL);

```

```
virtual void setDatumValUnsigned32(DtDatumParam param, unsigned long val,
    int *retcode = NULL);
virtual void setDatumValByteArray(DtDatumParam param, const char *val,
    int *retcode = NULL);

// Implements DtPdu's virtual function to print the PDU's data
virtual void printData() const;

// Helper function that prints only the contents of the data records
virtual void printDataSection() const;

protected:

    friend class DtPduDatumRecordSet;

    // Returns our datum record set helper object.
    virtual DtDatumRecordSet* datumRecSet();
    // Const version of above
    virtual const DtDatumRecordSet* datumRecSet() const;

    // Returns a pointer to the data section of the PDU - the part that
    // starts with the numberOfFixedFields field.
    virtual DtNetDataPduTail *dataSection() const = 0;

    // Return a pointer to the index'th fixed data record in the
    // network representation. Index is between 1 and numRecords.
    virtual DtNetFixDatum *nthFixedDataRecord(int index) const;

    // Return a pointer to the index'th variable data record in the
    // network representation. Index is between 1 and numRecords.
    virtual DtNetVarDatum *nthVarDataRecord(int index) const;

    // Returns a pointer to the index'th fixed or variable record in the
    // network representation, depending on whether fixedOrVar is DtFIXED
    // or DtVAR.
    virtual void *datumValPtr(int fixedOrVar, int index) const;

    // Destructor
    virtual ~DtPduWithData();

protected:

    DtDatumRecordSet* myRecSet;
};
#endif
```


physEntDec.h

```

#if DtHLA

#include "baseEntDec.h"

// class DtPhysicalEntityDecoder:
//
// Instances of DtPhysicalEntityDecoder are used to decode a
// physical entity state message into a state repository in an HLA
// (as opposed to DIS) environment.

class DtPhysicalEntityDecoder : public DtBaseEntityDecoder
{
public:
    // constructor
    DtPhysicalEntityDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtPhysicalEntityDecoder();

    // copy constructor
    DtPhysicalEntityDecoder(const DtPhysicalEntityDecoder& orig);

    // assignment operator
    DtPhysicalEntityDecoder& operator=(const DtPhysicalEntityDecoder& orig);

public:
    // Create a DtPhysicalEntityDecoder.
    // User is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

protected:
    // static decode<AttributeName> member functions.
    DtDECLARE_ENTITY_ATTR_DECODER(ArticulatedParametersArray);
    #if DtRPR016
    DtDECLARE_ENTITY_ATTR_DECODER(ArticulatedParametersCount);
    #endif
    DtDECLARE_ENTITY_ATTR_DECODER(DamageState);
    DtDECLARE_ENTITY_ATTR_DECODER(EngineSmokeOn);
    DtDECLARE_ENTITY_ATTR_DECODER(FlamesPresent);
    DtDECLARE_ENTITY_ATTR_DECODER(HasFuelSupplyCap);
    DtDECLARE_ENTITY_ATTR_DECODER(HasRecoveryCap);
    DtDECLARE_ENTITY_ATTR_DECODER(HasRepairCap);
    DtDECLARE_ENTITY_ATTR_DECODER(HatchState);
    DtDECLARE_ENTITY_ATTR_DECODER(Immobilized);
    DtDECLARE_ENTITY_ATTR_DECODER(LifeformState);
    DtDECLARE_ENTITY_ATTR_DECODER(LightsState);
    DtDECLARE_ENTITY_ATTR_DECODER(Marking);
    DtDECLARE_ENTITY_ATTR_DECODER(PowerPlantOn);
    DtDECLARE_ENTITY_ATTR_DECODER(RampDeployed);

```

```
DtDECLARE_ENTITY_ATTR_DECODER(SmokePlumePresent);
DtDECLARE_ENTITY_ATTR_DECODER(TentDeployed);
DtDECLARE_ENTITY_ATTR_DECODER(TrailState);

// called by the decoder for the ArticulatedParametersArray attribute.
static void decodeArtParts(int numParams,
    DtNetArtParamRecord* netParams,
    DtArtPartList* apList);

// called by the decoder for the ArticulatedParametersArray attribute.
static void decodeAttParts(int numParams,
    DtNetArtParamRecord* netParams,
    DtAttachedPartList* attList);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_PHYS_ENTITY_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtPhysicalEntityDecoder, DtEntityStateRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_PHYS_ENTITY_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_PHYS_ENTITY_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif
```

physEntEnc.h

```

#if DtHLA

#include "baseEntEnc.h"
#include "entitySR.h"

// class DtPhysicalEntityEncoder:
//
// Instances of DtPhysicalEntityEncoder are used to encode a
// physical entity's state repository into a state message in an
// HLA environment.

class DtPhysicalEntityEncoder : public DtBaseEntityEncoder
{
public:
    // constructor
    DtPhysicalEntityEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtPhysicalEntityEncoder();

    // copy constructor
    DtPhysicalEntityEncoder(const DtPhysicalEntityEncoder& orig);

    // assignment operator
    DtPhysicalEntityEncoder& operator=(const DtPhysicalEntityEncoder& orig);

public:
    // Create a DtPhysicalEntityEncoder.
    // Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // static need<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_CHECKER(ArticulatedParametersArray);
#if DtRPR016
    DtDECLARE_ENTITY_ATTR_CHECKER(ArticulatedParametersCount);
#endif
    DtDECLARE_ENTITY_ATTR_CHECKER(DamageState);
    DtDECLARE_ENTITY_ATTR_CHECKER(EngineSmokeOn);
    DtDECLARE_ENTITY_ATTR_CHECKER(FlamesPresent);
    DtDECLARE_ENTITY_ATTR_CHECKER(HasFuelSupplyCap);
    DtDECLARE_ENTITY_ATTR_CHECKER(HasRecoveryCap);
    DtDECLARE_ENTITY_ATTR_CHECKER(HasRepairCap);
    DtDECLARE_ENTITY_ATTR_CHECKER(HatchState);
    DtDECLARE_ENTITY_ATTR_CHECKER(Immobilized);
    DtDECLARE_ENTITY_ATTR_CHECKER(LifeformState);
    DtDECLARE_ENTITY_ATTR_CHECKER(LightsState);
    DtDECLARE_ENTITY_ATTR_CHECKER(Marking);
    DtDECLARE_ENTITY_ATTR_CHECKER(PowerPlantOn);
    DtDECLARE_ENTITY_ATTR_CHECKER(RampDeployed);
    DtDECLARE_ENTITY_ATTR_CHECKER(SmokePlumePresent);
    DtDECLARE_ENTITY_ATTR_CHECKER(TentDeployed);
    DtDECLARE_ENTITY_ATTR_CHECKER(TrailState);

    // static decode<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_ENCODER(ArticulatedParametersArray);

```

```
#if DtrPR016
    DtDECLARE_ENTITY_ATTR_ENCODER(ArticulatedParametersCount);
#endif
    DtDECLARE_ENTITY_ATTR_ENCODER(DamageState);
    DtDECLARE_ENTITY_ATTR_ENCODER(EngineSmokeOn);
    DtDECLARE_ENTITY_ATTR_ENCODER(FlamesPresent);
    DtDECLARE_ENTITY_ATTR_ENCODER(HasFuelSupplyCap);
    DtDECLARE_ENTITY_ATTR_ENCODER(HasRecoveryCap);
    DtDECLARE_ENTITY_ATTR_ENCODER(HasRepairCap);
    DtDECLARE_ENTITY_ATTR_ENCODER(HatchState);
    DtDECLARE_ENTITY_ATTR_ENCODER(Immobilized);
    DtDECLARE_ENTITY_ATTR_ENCODER(LifeformState);
    DtDECLARE_ENTITY_ATTR_ENCODER(LightsState);
    DtDECLARE_ENTITY_ATTR_ENCODER(Marking);
    DtDECLARE_ENTITY_ATTR_ENCODER(PowerPlantOn);
    DtDECLARE_ENTITY_ATTR_ENCODER(RampDeployed);
    DtDECLARE_ENTITY_ATTR_ENCODER(SmokePlumePresent);
    DtDECLARE_ENTITY_ATTR_ENCODER(TentDeployed);
    DtDECLARE_ENTITY_ATTR_ENCODER(TrailState);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_PHYS_ENTITY_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtPhysicalEntityEncoder, \
        DtEntityStateRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that checker whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_PHYS_ENTITY_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtPhysicalEntityEncoder, \
        DtEntityStateRepository, \
        attrName, inspector)

#endif
```

pointObjStPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "exerciseConn.h"

#include "packets/pointObjStP.h"

#include "objectType.h"

// class DtPointObjectStatePdu:
//
// Instances of DtPointObjectStatePdu are used to state information for
// synthetic environment objects.

// DtPointObjectStatePdu callback function signature.
typedef void (*DtPointObjectStatePduCb)(DtPointObjectStatePdu* pdu, void*
usr);

class DtPointObjectStatePdu : public DtPdu
{
public:
    // Constructors
    DtPointObjectStatePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtPointObjectStatePdu(const DtNetPointObjectStatePdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtPointObjectStatePdu(const DtPointObjectStatePdu& orig);

    // destructor
    virtual ~DtPointObjectStatePdu();

    // Assignment operator
    DtPointObjectStatePdu& operator=(const DtPointObjectStatePdu& orig);

    // Set/Get object ID
    virtual void setObjectID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& objectID() const;

    // Set/Get referenced object ID
    virtual void setReferencedObjectID(const DtEntityIdentifier& id);
    virtual const DtEntityIdentifier& referencedObjectID() const;

    // Set/Get update number
    virtual void setUpdateNumber(DtU16 field);
    virtual DtU16 updateNumber() const;

    // Set/Get the force ID
    virtual void setForceID(DtForceType force);
    virtual DtForceType forceID() const;

    // Set/Get the modifications (by 8-bit integer)
    virtual void setModifications(DtU8 value);
    virtual DtU8 modifications() const;

    // Set/Get the modifications (by field)
    virtual void setModificationsField(
        DtModificationField field, DtBoolean value);
    virtual DtBoolean modificationsField(

```

```
        DtModificationField field) const;

// Set/Get the object type
virtual void setObjectType(const DtObjectType& type);
virtual const DtObjectType& objectType() const;

// Set/Get location
virtual void setLocation(DtConstVector coords);
virtual DtConstVector location() const;

// Set/Get orientation
virtual void setOrientation(const DtTaitBryan& orientation);
virtual const DtTaitBryan& orientation() const;

// Set/Get appearance, the lower 16 bits (as an integer)
virtual void setAppearance(DtU16 app);
virtual DtU16 appearance() const;

// Set/Get appearance by attribute for the lower 16 bits
virtual void setAppearanceAttribute(
    DtAppearAttribute attr, DtAppearAttrVal val);
virtual DtAppearAttrVal appearanceAttribute(
    DtAppearAttribute attr) const;

// Set/Get specific appearance, the upper 32 bits (as a 32 bit integer)
virtual void setSpecificAppearance(DtU32 app);
virtual DtU32 specificAppearance() const;

// Set/Get specific appearance by attribute, the upper 32 bits
virtual void setSpecificAppearanceAttribute(
    const DtAppearAttribute attr, const DtAppearAttrVal val);
virtual DtAppearAttrVal specificAppearanceAttribute(
    const DtAppearAttribute attr) const;

// Set/Get the requester ID
virtual void setRequesterID(const DtSimulationAddress& type);
virtual const DtSimulationAddress& requesterID() const;

// Set/Get the receiving ID
virtual void setReceivingID(const DtSimulationAddress& type);
virtual const DtSimulationAddress& receivingID() const;

// Print the data contained by this PDU
virtual void printData() const;

public:

    // Create a DtPointObjectStatePdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtPointObjectStatePduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtPointObjectStatePduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetPointObjectStatePdu* netPduStruct() const;
```

```
// Casts the 'this' pointer for changing elements in a const function
// Non-virtual since you can't override based on return type
DtPointObjectStatePdu* self() const;

protected:

    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
    DtObjectType myObjectType;
    DtVector myVector;
    DtTaitBryan myEulerAngles;
    DtSimulationAddress mySimulationAddress;
};
#endif
#endif
```

printUtil.h

```
#include "MachineTypes.h"

void DtDumpInteger(const char *label, long i);
void DtDumpFloat(const char *label, double f);
void DtDumpString(const char *label, const char *string);
void DtDumpEnum(const char *label, const char *enumStr, int enumNum);
void DtDumpHexBytes(const char *label, const void *bytes, int size);
void DtDumpHexWord(const char *label, DtU32 word);
void DtDumpBoolean(const char *label, DtBoolean val, int onlyIfTrue = 0);
```


ProcControl.h

```
#include "vlTime.h"

void DtSleep(DtTime sleep_period);

/* polls line-buffered standard input
 * returns NULL if no input
 * returns internal buffer of line if have input
 * exits on ^D
 */
char *DtPollInputLine(void);

#ifdef PCFTP
/* the following is used as the first argument to the PC version
of select. It cannot take a 0 like unix versions. I should be able
to get this number programatically. */

extern DtMaximumSocketNumber;
#endif
```

queryPdu.h

```
#if !DtHLA

#include "hostStructs.h"
#include "simanPdu.h"

#if DtDISVERS != 3
#define DtQueryStaticSize 40
#else
#define DtQueryStaticSize 44
#endif

class DtDataQueryPdu;
typedef void (*DtDataQueryPduCb)(DtDataQueryPdu *pdu, void *usr);

class DtDataQueryPdu : public DtSimanPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtDataQueryPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtDataQueryPdu(const DtNetDataQueryPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtDataQueryPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtDataQueryPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    unsigned long requestId() const;
    DtTime timeInterval() const;
    unsigned long numFixedFields() const;
    unsigned long numVarFields() const;
    DtDatumParam fixedDatumId(int index) const;
    DtDatumParam varDatumId(int index) const;

/***** MUTATOR FUNCTIONS *****/

    void setRequestId(unsigned long id);
    void setTimeInterval(DtTime seconds);
    int setFixedDatumId(int index, DtDatumParam id);
    int setVarDatumId(int index, DtDatumParam id);
    void initDatumIds(unsigned long numFixedFields,
        unsigned long numVarFields);

    void printData() const;

protected:
    DtPduKind internalGetPduKind() const {return DtDataQueryPduKind;}
};
#endif
```

radXmitCbInfo.h

```

#if DtHLA || DtHLA_GATEWAY

#include "hlaDefines.h"
#include "vllist.h"

// Entity state callback function signature.
typedef void (*DtRadioXmitterCallbackFunction)(
    DtReflectedRadioTransmitter* ent, void* userData);

#endif

// class DtRadioTransmitterCallbackInfo:
//
// Instances of DtRadioTransmitterCallbackInfo are
class DtRadioTransmitterCallbackInfo : public DtListedObj
{
public:
    // default constructor
    DtRadioTransmitterCallbackInfo();

    // constructor
    DtRadioTransmitterCallbackInfo(
        DtRadioXmitterCallbackFunction cb,
        void* userData);

    // destructor
    virtual ~DtRadioTransmitterCallbackInfo();

    // copy constructor
    DtRadioTransmitterCallbackInfo(
        const DtRadioTransmitterCallbackInfo& orig);

    // assignment operator
    DtRadioTransmitterCallbackInfo& operator=
        (const DtRadioTransmitterCallbackInfo& orig);

    // equality operators
    int operator==(const DtRadioTransmitterCallbackInfo& orig) const;
    int operator!=(const DtRadioTransmitterCallbackInfo& orig) const;

    // set/get the callback function
    virtual void setCallbackFunction(DtRadioXmitterCallbackFunction funct);
    virtual DtRadioXmitterCallbackFunction callbackFunction() const;

    // set/get the user data
    virtual void setUserData(void* data);
    virtual void* userData() const;

protected:
    DtRadioXmitterCallbackFunction myCallbackFunction;
    void* myUserData;
};

```

radioRecvrDec.h

```
#if DtHLA

#ifndef DtRadioReceiverDecoder_H_
#define DtRadioReceiverDecoder_H_

#include "chooseDefines.h"
#include "baseEmbedDec.h"

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLARE_RADIOECVR_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtRadioReceiverRepository, attrName)

// class DtRadioReceiverDecoder:
//
// Instances of DtRadioReceiverDecoder are

class DtRadioReceiverDecoder : public DtBaseEmbeddedSystemDecoder
{
public:
    // default constructor
    DtRadioReceiverDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtRadioReceiverDecoder();

    // copy constructor
    DtRadioReceiverDecoder(const DtRadioReceiverDecoder& orig);

    // assignment operator
    DtRadioReceiverDecoder& operator=(const DtRadioReceiverDecoder& orig);

public:
    // Create a DtBaseEntityDecoder. Caller is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.

    DtDECLARE_RADIOECVR_ATTR_DECODER(RadioNumber);
    DtDECLARE_RADIOECVR_ATTR_DECODER(ReceivedPower);
    DtDECLARE_RADIOECVR_ATTR_DECODER(ReceivedTransmitter);
    DtDECLARE_RADIOECVR_ATTR_DECODER(ReceiverState);
};

// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_RADIOECVR_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
```

```
DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
    DtRadioReceiverDecoder, DtRadioReceiverRepository, \
    attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_RADIOECVR_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_RADIOECVR_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif    //DtRadioReceiverDecoder_H_
```

radioRecvrEnc.h

```
#if DtHLA

#ifndef DtRadioReceiverEncoder_H_
#define DtRadioReceiverEncoder_H_

#include "chooseDefines.h"
#include "baseEmbedEnc.h"

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_RADIOECVR_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtRadioReceiverRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_RADIOECVR_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtRadioReceiverRepository, attrName)

// class DtRadioReceiverEncoder:
//
// Instances of DtRadioReceiverEncoder are

class DtRadioReceiverEncoder : public DtBaseEmbeddedSystemEncoder
{
public:
    // default constructor
    DtRadioReceiverEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtRadioReceiverEncoder();

    // copy constructor
    DtRadioReceiverEncoder(const DtRadioReceiverEncoder& orig);

    // assignment operator
    DtRadioReceiverEncoder& operator=(const DtRadioReceiverEncoder& orig);

public:
    // Create a DtBaseEntityEncoder. Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    DtDECLARE_RADIOECVR_ATTR_CHECKER(RadioNumber);
    DtDECLARE_RADIOECVR_ATTR_CHECKER(ReceivedPower);
    DtDECLARE_RADIOECVR_ATTR_CHECKER(ReceivedTransmitter);
    DtDECLARE_RADIOECVR_ATTR_CHECKER(ReceiverState);

    DtDECLARE_RADIOECVR_ATTR_ENCODER(RadioNumber);
    DtDECLARE_RADIOECVR_ATTR_ENCODER(ReceivedPower);
    DtDECLARE_RADIOECVR_ATTR_ENCODER(ReceivedTransmitter);
    DtDECLARE_RADIOECVR_ATTR_ENCODER(ReceiverState);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
```

```
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_RADIOECVR_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtRadioReceiverEncoder, \
        DtRadioReceiverRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that check whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_RADIOECVR_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtRadioReceiverEncoder, \
        DtRadioReceiverRepository, \
        attrName, inspector)

#endif    //DtRadioReceiverEncoder_H_
```

radioRecvrSR.h

```
#include "embedSystemSR.h"
#include "chooseDefines.h"

// class DtRadioReceiverRepository:
//
// Instances of DtRadioReceiverRepository are

class DtRadioReceiverRepository : public DtEmbeddedSystemRepository
{
public:
    // default constructor
    DtRadioReceiverRepository();

    // destructor
    virtual ~DtRadioReceiverRepository();

    // copy constructor
    DtRadioReceiverRepository(const DtRadioReceiverRepository& orig);

    // assignment operator
    DtRadioReceiverRepository& operator=(const DtRadioReceiverRepository&
orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const;

    static DtRadioReceiverRepository* create();

    // Set/Get the Receiver radio number.
    virtual void setRadioId(unsigned int number);
    virtual unsigned int radioId() const;

    // Set/Get the received power.
    virtual void setReceivedPower(DtFloat32 power);
    virtual DtFloat32 receivedPower() const;

    // Set/Get the received transmitter.
    virtual void setTransmitRadioId(const DtGlobalObjectDesignator& id);
    virtual const DtGlobalObjectDesignator& transmitRadioId() const;

    // Set/Get the receiver state.
    virtual void setReceiverState(DtRadioReceiverState state);
    virtual DtRadioReceiverState receiverState() const;

    virtual void printData() const;

protected:
    int myReceiverRadioNumber;
    DtFloat32 myReceivedPower;
    DtGlobalObjectDesignator myTransmitId;
    DtRadioReceiverState myReceiverState;
};
```


radioXmitDec.h

```

#if DtHLA

#ifndef DtRadioTransmitterDecoder_H_
#define DtRadioTransmitterDecoder_H_

#include "chooseDefines.h"
#include "baseEmbedDec.h"

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLARE_RADIOXMIT_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtRadioTransmitterRepository, attrName)

// class DtRadioTransmitterDecoder:
//
// Instances of DtRadioTransmitterDecoder are

class DtRadioTransmitterDecoder : public DtBaseEmbeddedSystemDecoder
{
public:
    // default constructor
    DtRadioTransmitterDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtRadioTransmitterDecoder();

    // copy constructor
    DtRadioTransmitterDecoder(const DtRadioTransmitterDecoder& orig);

    // assignment operator
    DtRadioTransmitterDecoder& operator=
        (const DtRadioTransmitterDecoder& orig);

public:
    // Create a DtBaseEntityDecoder. Caller is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    // add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
    DtDECLARE_RADIOXMIT_ATTR_DECODER(AntennaPattern);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(AntennaPatternType);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(CryptoKeyID);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(CryptoSystem);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(DetailedModulationType);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(Frequency);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(FrequencyBandwidth);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(FrequencyHopInUse);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(InputSource);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(ModulationParameters);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(MajorModulationType);
    DtDECLARE_RADIOXMIT_ATTR_DECODER(ModulationSystemType);

```

```
DtDECLARE_RADIOXMIT_ATTR_DECODER(Power);
DtDECLARE_RADIOXMIT_ATTR_DECODER(PsuedoNoiseSpectrumInUse);
DtDECLARE_RADIOXMIT_ATTR_DECODER(RadioNumber);
DtDECLARE_RADIOXMIT_ATTR_DECODER(RadioType);
DtDECLARE_RADIOXMIT_ATTR_DECODER(TimeHopInUse);
DtDECLARE_RADIOXMIT_ATTR_DECODER(TransmitState);
DtDECLARE_RADIOXMIT_ATTR_DECODER(WorldLocation);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_RADIOXMIT_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtRadioTransmitterDecoder, DtRadioTransmitterRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_RADIOXMIT_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_RADIOXMIT_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif //DtRadioTransmitterDecoder_H_
```

radioXmitEnc.h

```

#if DtHLA

#ifndef DtRadioTransmitterEncoder_H_
#define DtRadioTransmitterEncoder_H_

#include "chooseDefines.h"
#include "baseEmbedEnc.h"

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_RADIOXMIT_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtRadioTransmitterRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_RADIOXMIT_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtRadioTransmitterRepository, attrName)

// class DtRadioTransmitterEncoder:
//
// Instances of DtRadioTransmitterEncoder are

class DtRadioTransmitterEncoder : public DtBaseEmbeddedSystemEncoder
{
public:
    // default constructor
    DtRadioTransmitterEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtRadioTransmitterEncoder();

    // copy constructor
    DtRadioTransmitterEncoder(const DtRadioTransmitterEncoder& orig);

    // assignment operator
    DtRadioTransmitterEncoder& operator=(const DtRadioTransmitterEncoder&
orig);

public:
    // Create a DtBaseEntityEncoder. Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(AntennaPattern);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(AntennaPatternType);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(CryptoKeyID);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(CryptoSystem);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(DetailedModulationType);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(Frequency);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(FrequencyBandwidth);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(FrequencyHopInUse);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(InputSource);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(ModulationParameters);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(MajorModulationType);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(ModulationSystemType);
    DtDECLARE_RADIOXMIT_ATTR_CHECKER(Power);

```

```
DtDECLARE_RADIOXMIT_ATTR_CHECKER(PsuedoNoiseSpectrumInUse);
DtDECLARE_RADIOXMIT_ATTR_CHECKER(RadioNumber);
DtDECLARE_RADIOXMIT_ATTR_CHECKER(RadioType);
DtDECLARE_RADIOXMIT_ATTR_CHECKER(TimeHopInUse);
DtDECLARE_RADIOXMIT_ATTR_CHECKER(TransmitState);
DtDECLARE_RADIOXMIT_ATTR_CHECKER(WorldLocation);

DtDECLARE_RADIOXMIT_ATTR_ENCODER(AntennaPattern);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(AntennaPatternType);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(CryptoKeyID);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(CryptoSystem);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(DetailedModulationType);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(Frequency);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(FrequencyBandwidth);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(FrequencyHopInUse);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(InputSource);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(ModulationParameters);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(MajorModulationType);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(ModulationSystemType);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(Power);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(PsuedoNoiseSpectrumInUse);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(RadioNumber);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(RadioType);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(TimeHopInUse);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(TransmitState);
DtDECLARE_RADIOXMIT_ATTR_ENCODER(WorldLocation);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_RADIOXMIT_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtRadioTransmitterEncoder, \
        DtRadioTransmitterRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that check whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_RADIOXMIT_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtRadioTransmitterEncoder, \
        DtRadioTransmitterRepository, \
        attrName, inspector)

#endif //DtRadioTransmitterEncoder_H_
```

radioXmitSR.h

```

#include "chooseDefines.h"
#include "embedSystemSR.h"
#include "antPatRec.h"

enum DtLongWordPortion
{
    DtUpperWord,
    DtLowerWord
};

// class DtRadioTransmitterRepository:
//
// Instances of DtRadioTransmitterRepository are

class DtRadioTransmitterRepository : public DtEmbeddedSystemRepository
{
public:
    // default constructor
    DtRadioTransmitterRepository();

    // destructor
    virtual ~DtRadioTransmitterRepository();

    // copy constructor
    DtRadioTransmitterRepository(const DtRadioTransmitterRepository& orig);

    // assignment operator
    DtRadioTransmitterRepository& operator=(
        const DtRadioTransmitterRepository& orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const;

    // Set/Get the antenna pattern.
    virtual void setAntennaPattern(const DtAntennaPatternRecord& rec);
    virtual const DtAntennaPatternRecord& antennaPattern() const;

    // Set/Get the antenna pattern type.
    virtual void setAntennaPatternType(DtAntennaPatternType pattern);
    virtual DtAntennaPatternType antennaPatternType() const;

    // Set/Get the crypto key id.
    virtual void setCryptoKeyId(DtU16 key);
    virtual DtU16 cryptoKeyId() const;

    // Set/Get the crypto system.
    virtual void setCryptoSystem(DtRadioCryptoSystem system);
    virtual DtRadioCryptoSystem cryptoSystem() const;

    // Set/Get the detailed modulation.
    virtual void setDetailedModulationType(DtDetailedModulationType modula-
tion);
    virtual DtDetailedModulationType detailedModulationType() const;

    // Set the frequency, either as upper and lower word, or
    // as full 64 bit integer.
    virtual void setFrequency(unsigned long upper, unsigned long lower);
    virtual void setFrequency(unsigned long freq);

    // Get either the upper or lower word of the 64 bit frequency.
    virtual unsigned long frequency(
        DtLongWordPortion upperLower = DtLowerWord) const;

```

```
// Set/Get the frequency bandwidth.
virtual void setFrequencyBandwidth(DtFloat32 bandwidth);
virtual DtFloat32 frequencyBandwidth() const;

// Set/Get the frequency hop in use.
virtual void setFrequencyHopInUse(DtBoolean hopInUse);
virtual DtBoolean frequencyHopInUse() const;

// Set/Get the input source.
virtual void setInputSource(DtRadioInputSource source);
virtual DtRadioInputSource inputSource() const;

// Set/Get the modulation parameters' network representation
virtual void setModulationParams(
    const void *networkRepresentation, DtU8 size);
virtual const void *modulationParamsAddr() const;
virtual DtU8 modulationParamsLength() const;

// Set/Get the major modulation type.
virtual void setMajorModulationType(DtRadioMajorModulation modulation);
virtual DtRadioMajorModulation majorModulationType() const;

// Set/Get the modulation system.
virtual void setModulationSystemType(DtRadioSystem system);
virtual DtRadioSystem modulationSystemType() const;

// Set/Get the power.
virtual void setPower(DtFloat32 power);
virtual DtFloat32 power() const;

// Set/Get the Psuedo Noise Spectrum In Use.
virtual void setPsuedoNoiseSpectrumInUse(DtBoolean inUse);
virtual DtBoolean psuedoNoiseSpectrumInUse() const;

// Set/Get the Transmitter radio number.
virtual void setRadioId(unsigned int number);
virtual unsigned int radioId() const;

// Set/Get the radio entity type.
virtual void setRadioEntityType(const DtRadioEntityType& type);
virtual const DtRadioEntityType& radioEntityType() const;

// Set/Get the Time Hop In Use.
virtual void setTimeHopInUse(DtBoolean inUse);
virtual DtBoolean timeHopInUse() const;

// Set/Get the transmit state.
virtual void setTransmitState(DtRadioTransmitState state);
virtual DtRadioTransmitState transmitState() const;

// Set/Get the world location
virtual void setWorldAntennaLoc(DtConstVector location);
virtual DtConstVector worldAntennaLoc() const;

// Set/Get the relative location (wrapper around EmbedSysSR
// member func, for compatibility with DIS transmitter PDU.
virtual void setRelativeAntennaLoc(DtConstVector location);
virtual DtConstVector relativeAntennaLoc() const;

virtual void printData() const;

public:
    static DtRadioTransmitterRepository* create();

protected:
```

```
DtAntennaPatternType      myAntennaPatternType;
DtU16                     myCryptoKeyId;
DtRadioCryptoSystem       myCryptoSystem;
DtDetailedModulationType  myDetailedModulation;
unsigned long             myFreqHi;
unsigned long             myFreqLo;
DtFloat32                myFrequencyBandwidth;
DtBoolean                myFrequencyHopInUse;
DtRadioInputSource       myInputSource;
char*                    myModulationParamsAddr;
DtU8                     myModulationParamsLength;
DtAntennaPatternRecord*  myAntPat;
DtRadioMajorModulation    myMajorModulationType;
DtRadioSystem            myModulationSystem;
DtFloat32                myPower;
DtBoolean                myPsuedoNoiseSpectrumInUse;
DtU16                    myTransmitterRadioNumber;
DtRadioEntityType        myRadioEntityType;
DtBoolean                myTimeHopInUse;
DtRadioTransmitState     myTransmitState;
DtVector                 myWorldLocation;
};
```

radRcvrCbInfo.h

```
#if DtHLA || DtHLA_GATEWAY

#include "hlaDefines.h"
#include "vllist.h"

// Entity state callback function signature.
typedef void (*DtRadioRcvrCallbackFunction)(
    DtReflectedRadioReceiver* ent, void* userData);

#endif

// class DtRadioReceiverCallbackInfo:
//
// Instances of DtRadioReceiverCallbackInfo are
class DtRadioReceiverCallbackInfo : public DtListedObj
{
public:
    // default constructor
    DtRadioReceiverCallbackInfo();

    // constructor
    DtRadioReceiverCallbackInfo(
        DtRadioRcvrCallbackFunction cb,
        void* userData);

    // destructor
    virtual ~DtRadioReceiverCallbackInfo();

    // copy constructor
    DtRadioReceiverCallbackInfo(const DtRadioReceiverCallbackInfo& orig);

    // assignment operator
    DtRadioReceiverCallbackInfo& operator=
        (const DtRadioReceiverCallbackInfo& orig);

    // equality operators
    int operator==(const DtRadioReceiverCallbackInfo& orig) const;
    int operator!=(const DtRadioReceiverCallbackInfo& orig) const;

    // set/get the callback function
    virtual void setCallbackFunction(DtRadioRcvrCallbackFunction func);
    virtual DtRadioRcvrCallbackFunction callbackFunction() const;

    // set/get the user data
    virtual void setUserData(void* data);
    virtual void* userData() const;

protected:
    DtRadioRcvrCallbackFunction myCallbackFunction;
    void* myUserData;
};
```


receivePdu.h

```

#ifndef !DtHLA

#include "hostStructs.h"
#include "pdu.h"
#include "packets/transmitterP.h"
#define DtReceiverStaticSize 36

class DtReceiverPdu;
typedef void (*DtReceiverPduCb)(DtReceiverPdu *pdu, void *usr);

class DtReceiverPdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtReceiverPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtReceiverPdu(const DtNetReceiverPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtReceiverPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtReceiverPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &entityId() const;
    unsigned int radioId() const;
    DtRadioReceiverState receiverState() const;
    float receivedPower() const;
    const DtEntityIdentifier &transmitEntityId() const;
    unsigned int transmitRadioId() const;

/***** MUTATOR FUNCTIONS *****/

    void setEntityId(const DtEntityIdentifier &id);
    void setRadioId(unsigned int id);
    void setReceiverState(DtRadioReceiverState state);
    void setReceivedPower(float power);
    void setTransmitEntityId(const DtEntityIdentifier &id);
    void setTransmitRadioId(unsigned int id);

    void printData() const;
protected:
    DtPduKind internalGetPduKind() const {return DtReceiverPduKind;}
    DtEntityIdentifier theId;
    DtEntityIdentifier transId;
};
#endif // !DtHLA

```

recIntrCbInfo.h

```
#if DtHLA

#include "RTI.hh"

#include "chooseDefines.h"

// class DtReceiveInteractionCbInfo:
//
// Instances of DtReceiveInteractionCbInfo are used for providing
// a callback when an interaction is received from another federate.

typedef void (*DtReceiveInteractionCb)(DtInteraction* inter, void* usr);

class DtReceiveInteractionCbInfo
{
public:
    // constructor
    DtReceiveInteractionCbInfo(RTI::InteractionClassHandle h,
        DtReceiveInteractionCb cb,
        void* usr);

    // destructor
    virtual ~DtReceiveInteractionCbInfo();

    // copy constructor
    DtReceiveInteractionCbInfo(
        const DtReceiveInteractionCbInfo& orig);

    // assignment operator
    DtReceiveInteractionCbInfo& operator=(
        const DtReceiveInteractionCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(const DtReceiveInteractionCbInfo* const other) const;

    // Check whether one callback info is equal to another.
    virtual int operator==(const DtReceiveInteractionCbInfo& other) const;

    // Get the interaction class handle.
    virtual const RTI::InteractionClassHandle& interactionClass() const;

    // Get the callback function.
    virtual const DtReceiveInteractionCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:
    RTI::InteractionClassHandle myClass;
    DtReceiveInteractionCb myFunc;
    void* myUsr;
};
#endif
```

recordPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "recordSetPdu.h"
#include "exerciseConn.h"

#include "packets/recordP.h"

// class DtRecordPdu:
//
// Instances of DtRecordPdu are used to send responses to the Record Query
// PDU.

// DtRecordPdu callback function signature.
typedef void (*DtRecordPduCb)(DtRecordPdu* pdu, void* usr);

class DtRecordPdu : public DtRecordSetPdu
{
public:
    // Constructors
    DtRecordPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRecordPdu(const DtNetRecordPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRecordPdu(const DtRecordPdu& orig);

    // destructor
    virtual ~DtRecordPdu();

    // Assignment operator
    DtRecordPdu& operator=(const DtRecordPdu& orig);

    // Set/Get originating entity identifier
    virtual void setOriginatingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& originatingEntityID() const;

    // Set/Get receiving entity identifier
    virtual void setReceivingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& receivingEntityID() const;

    // Set/Get request ID
    virtual void setRequestID(DtU32 id);
    virtual DtU32 requestID() const;

    // Set/Get required reliability service
    virtual void setReqRelService(enum DtRequiredReliabilityService
service);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Set/Get the event type
    virtual void setEventType(DtEventType type);
    virtual DtEventType eventType() const;

    // Set/Get the response serial number
    virtual void setResponseSerialNumber(DtU32 num);
    virtual DtU32 responseSerialNumber() const;

    // Print the data contained by this PDU
    virtual void printData() const;

```

```
public:
    // Create a DtRecordPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtRecordPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtRecordPduCb cb, void* usr);

protected:
    // Returns a pointer to the beginning of the record set data struct.
    virtual DtNetRecordSetPdu* netRecordSetPduStruct() const;

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    // Non-virtual since you can't override based on return type
    virtual DtNetRecordPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non virtual since you can't override based on return type
    DtRecordPdu* self() const;

protected:
    // Size of PDU with zero data bytes.
    static int theStaticSize;

protected:
    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;
};
#endif
#endif
```

recordSet.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "hostStructs.h"
#include "datumIdType.h"

#include "packets/recordSetP.h"

// class DtRecordSet:
//
// Instances of DtRecordSet are used to contain the record sets used
// by the PDU classes which inherit from the record set PDU base class.

class DtRecordSet
{
public:
    // Constructors
    DtRecordSet();
    DtRecordSet(const DtRecordSet& orig);

    // Destructor
    virtual ~DtRecordSet();

    // Assignment operators
    DtRecordSet& operator=(const DtRecordSet& orig);
    DtRecordSet& operator=(const DtNetRecordSet& orig);

    // Set/Get the datum ID.
    virtual DtBoolean setDatumID(DtDatumParam value);
    virtual DtDatumParam datumID() const;

    // Set/Get the serial number.
    virtual void setSerialNumber(DtU32 value);
    virtual DtU32 serialNumber() const;

    // Set/Get the record length in bits.
    // setRecordBitLength() also resets the number of records to zero, and
    // may return DtFalse (with no action) if the length is invalid.
    virtual DtBoolean setRecordBitLength(DtU16 bitLength);
    virtual DtU16 recordBitLength() const;

    // Set/Get the record length in bytes.
    // setRecordLength() also resets the number of records to zero, and
    // may return DtFalse (with no action) if the length is invalid.
    virtual DtBoolean setRecordByteLength(int byteLength);
    virtual int recordByteLength() const;

    // Set/Get the number of of records in this set
    // When the number of records is changed, old records are not preserved
    virtual DtBoolean setRecordCount(DtU16 bytes);
    virtual DtU16 recordCount() const;

    // Set/Get multiple data bytes. Provide a pointer and data is copied to
    // and from that memory. Will copy recordByteLength() bytes on set/get.
    // These functions return DtFalse if the record index is out of range.
    virtual DtBoolean setDataBytes(int recordIndex, const void* data);
    virtual DtBoolean getDataBytes(int recordIndex, void* buff) const;

    // Set/Get a data byte. These functions return DtFalse if the record

```

```
// index or byte index is out of range.
virtual DtBoolean setDataByte(
    int recordIndex, DtU8 value, int byteIndex = 0);
virtual DtBoolean getDataByte(
    int recordIndex, DtU8& value, int byteIndex = 0) const;

// Set/Get data in signed 16-bit integer format.
// These functions return DtFalse if the record length is not 16-bits,
// or if the record index is out of range.
virtual DtBoolean setDataInt16(int recordIndex, DtInt16& value);
virtual DtBoolean getDataInt16(int recordIndex, DtInt16& value) const;

// Set/Get data in unsigned 16-bit integer format.
// These functions return DtFalse if the record length is not 16-bits,
// or if the record index is out of range.
virtual DtBoolean setDataU16(int recordIndex, DtU16& value);
virtual DtBoolean getDataU16(int recordIndex, DtU16& value) const;

// Set/Get data in signed 32-bit integer format.
// These functions return DtFalse if the record length is not 32-bits,
// or if the record index is out of range.
virtual DtBoolean setDataInt32(int recordIndex, DtInt32& value);
virtual DtBoolean getDataInt32(int recordIndex, DtInt32& value) const;

// Set/Get data in unsigned 32-bit integer format.
// These functions return DtFalse if the record length is not 32-bits,
// or if the record index is out of range.
virtual DtBoolean setDataU32(int recordIndex, DtU32& value);
virtual DtBoolean getDataU32(int recordIndex, DtU32& value) const;

// Set/Get data in 32-bit floating point format.
// These functions return DtFalse if the record length is not 32-bits,
// or if the record index is out of range.
virtual DtBoolean setDataFloat32(int recordIndex, DtFloat32& value);
virtual DtBoolean getDataFloat32(int recordIndex, DtFloat32& value)
const;

// Set/Get data in 64-bit floating point format.
// These functions return DtFalse if the record length is not 64-bits,
// or if the record index is out of range.
virtual DtBoolean setDataFloat64(int recordIndex, DtFloat64& value);
virtual DtBoolean getDataFloat64(int recordIndex, DtFloat64& value)
const;

// Get the total number of bytes required for the network representation
// of the record set.
virtual int recordSetLength() const;

// Print this record
virtual void print() const;

// Load/unload data from a packet record
virtual void unloadNetRecordSet(const DtNetRecordSet& record);
virtual void loadNetRecordSet(DtNetRecordSet& record) const;

protected:

    DtDatumParam          myDatumID;
    DtU32                 mySerialNumber;
    DtU16                 myRecordLength;
    DtU16                 myRecordCount;

    char*                 myData;
};
#endif
#endif
```

recordSetPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "packets/recordSetP.h"

#include "recordSet.h"

// class DtRecordSetPdu:
//
// DtRecordSetPdu is an abstract base class used by PDUs which require the
// record set functionality.

class DtRecordSetPdu : public DtPdu
{
public:
    // destructor
    virtual ~DtRecordSetPdu();

    // Assignment operator
    DtRecordSetPdu& operator=(const DtRecordSetPdu& orig);

    // Set/Get the number of record sets
    virtual DtBoolean setNumRecordSets(DtU32 numSets);
    virtual DtU32 numRecordSets() const;

    // Set/Get a record set. If the index is out of bounds, then DtFalse
    // is returned. If the space would exceed the maximum space permitted
    // for a PDU (for a set) then DtFalse may be returned.
    virtual DtBoolean setRecordSet(int index, const DtRecordSet& recSet);
    virtual DtBoolean getRecordSet(int index, DtRecordSet& recSet) const;

    // Print the data contained by this PDU
    virtual void printData() const;

protected:
    // Constructors (protected because this is an abstract base class)
    DtRecordSetPdu();
    DtRecordSetPdu(const DtRecordSetPdu& orig);

    // This function checks the data records for consistency between the
    // actual size of the PDU and the size suggested by the PDU. It also
    // tries to fix any problems that are detected. This function is
    // intended to be called in the PDU constructor that accepts network
    // data immediately following initpdu.
    // Non-virtual since meant to be called from ctor.
    DtBoolean validateRecordSets(DtNetRecordSetPdu& netStruct);

    // Returns a pointer to the beginning of the record set data struct.
    // This must be implemented by the derived class because the location
    // of the record sets within the PDU can vary.
    virtual DtNetRecordSetPdu* netRecordSetPduStruct() const = 0;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

    // Returns the length of data used by the record sets beginning at
    // the location specified by netRecordSetPduStruct(). This can be used

```

```
// by the derived class if any data follows the record set data.
virtual int netRecordSetLength() const;

// Returns a buffer offset to the given record set.
virtual int recordSetOffset(int recordSetIndex) const;

// Casts the 'this' pointer for changing elements in a const function
// Non virtual since you can't override based on return type
DtRecordSetPdu* self() const;

protected:

// These are used by recordSetOffset() to save the value and index of
// the last offset which was computed in case it can be used again.
int mySaveRecCount;
int mySaveBuffCount;

};
#endif
#endif
```


recQueryPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"

#include "packets/recQueryP.h"

// class DtRecordQueryPdu:
//
// Instances of DtRecordQueryPdu are used to request data records from an
// entity.

// DtRecordQueryPdu callback function signature.
typedef void (*DtRecordQueryPduCb)(DtRecordQueryPdu* pdu, void* usr);

class DtRecordQueryPdu : public DtPdu
{
public:
    // Constructors
    DtRecordQueryPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRecordQueryPdu(const DtNetRecordQueryPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRecordQueryPdu(const DtRecordQueryPdu& orig);

    // destructor
    virtual ~DtRecordQueryPdu();

    // Assignment operator
    DtRecordQueryPdu& operator=(const DtRecordQueryPdu& orig);

    // Set/Get originating entity identifier
    virtual void setOriginatingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& originatingEntityID() const;

    // Set/Get receiving entity identifier
    virtual void setReceivingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& receivingEntityID() const;

    // Set/Get request ID
    virtual void setRequestID(DtU32 id);
    virtual DtU32 requestID() const;

    // Set/Get required reliability service
    virtual void setReqRelService(enum DtRequiredReliabilityService
service);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Set/Get the event type
    virtual void setEventType(DtEventType type);
    virtual DtEventType eventType() const;

    // Set/Get the time
    virtual void setTime(DtU32 value);
    virtual DtU32 time() const;

    // Set/Get the number of records to be queried for. Set may return
DtFalse
    // if the number of records is too large.

```

```
virtual DtBoolean setNumRecords(DtU32 num);
virtual DtU32 numRecords() const;

// Set/Get a record ID. These methods return DtFalse if the index is out
// of range.
virtual DtBoolean setRecordID(int index, DtDatumParam value);
virtual DtBoolean getRecordID(int index, DtDatumParam& value) const;

// Print the data contained by this PDU
virtual void printData() const;

public:

    // Create a DtRecordQueryPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtRecordQueryPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtRecordQueryPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    // Non-virtual since you can't override based on return type
    virtual DtNetRecordQueryPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non virtual since you can't override based on return type
    DtRecordQueryPdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

protected:

    // Size of PDU with zero data bytes.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;

};
#endif
#endif
```

refAttrCbInfo.h

```

#if DtHLA

#include "refAttrCbList.h"
#include "RTI.hh"

#if DtRTI103
#define ObjectHandle ObjectID
#endif

// class DtReflectAttributeValuesCbInfo:
//
// Instances of DtReflectAttributeValuesCbInfo are used for providing
// a callback when attributes change for an object in federation.
// This class is a helper class to DtReflectAttributeValuesCbList.

class DtReflectAttributeValuesCbInfo
{
public:
    // constructor
    DtReflectAttributeValuesCbInfo(RTI::ObjectHandle id,
        DtReflectAttributeValuesCb cb,
        void* usr);

    // destructor
    virtual ~DtReflectAttributeValuesCbInfo();

    // copy constructor
    DtReflectAttributeValuesCbInfo(
        const DtReflectAttributeValuesCbInfo& orig);

    // assignment operator
    DtReflectAttributeValuesCbInfo& operator=(
        const DtReflectAttributeValuesCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(const DtReflectAttributeValuesCbInfo* const other)
const;

    // Check whether one callback info is equal to another.
    virtual int operator==(const DtReflectAttributeValuesCbInfo& other)
const;

    // Get the object id.
    virtual const RTI::ObjectHandle& objectId() const;

    // Get the callback function.
    virtual const DtReflectAttributeValuesCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:
    RTI::ObjectHandle myId;
    DtReflectAttributeValuesCb myFunc;
    void* myUsr;
};
#endif

```

refAttrCbList.h

```
#if DtHLA

#include "vllist.h"
#include "RTI.hh"

#include "chooseDefines.h"

#if DtRTI103
#define ObjectHandle ObjectID
#endif

// class DtReflectAttributeValuesCbList:
//
// Instances of DtReflectAttributeValuesCbList are used to provide
// a list of callbacks (DtReflectAttributeValuesCbInfo) which are
// called when attributes change for an object in a federation.

typedef void (*DtReflectAttributeValuesCb)(const DtStateMsg& stateMsg,
void* usr);

class DtReflectAttributeValuesCbList
{
public:

    // default constructor
    DtReflectAttributeValuesCbList();

    // destructor
    virtual ~DtReflectAttributeValuesCbList();

    // copy constructor
    DtReflectAttributeValuesCbList(
        const DtReflectAttributeValuesCbList& orig);

    // assignment operator
    DtReflectAttributeValuesCbList& operator=(
        const DtReflectAttributeValuesCbList& orig);

    // Add/Remove a reflect attribute callback to/from the list.
    // Returns whether add/remove took place.
    virtual int add(RTI::ObjectHandle id, DtReflectAttributeValuesCb cb,
        void* usr);
    virtual int remove(RTI::ObjectHandle id, DtReflectAttributeValuesCb cb,
        void* usr);

    // Remove all callbacks from the list.
    virtual void clear();

    // State messages are for a specific object id.
    // Callbacks are for a particular object id.
    // Call any callbacks in the list for the object id specified in
    // the state message.
    virtual void apply(const DtStateMsg &msg);

protected:

    // Remove all callbacks from the list.
    // Non-virtual since meant to be called from dtor, otherwise
    // should call clear().
    void dtorClear();

    // Copy another callback list into this object's callback list.
    // Non-virtual since meant to be called from ctor.
```

```
    void ctorCopyList(const DtReflectAttributeValuesCbList& orig);  
protected:  
    DtList myList;  
};  
#endif
```

reflAggList.h

```
#if DtDISVERS >= 5

// This file provides protocol independence for remote aggregate lists.

#if DtHLA
#include "hReflAggList.h"
#else
#include "dReflAggList.h"
#endif

#endif
```

reflectedAgg.h

```
#if DtDISVERS >= 5

// This file provides protocol independence for reflected aggregates.

#if DtHLA
#include "hReflectedAgg.h"
#else
#include "dReflectedAgg.h"
#endif

#endif
```

refObjCbInfo.h

```
#include "chooseDefines.h"
#include "vllist.h"

// Callback function that takes a DtReflectedObject and user data
typedef void (*DtReflectedObjectCallbackFunction)(
    DtReflectedObject* obj, void* userData);

// class DtReflectedObjectCallbackInfo:
//
// Instances of DtReflectedObjectCallbackInfo are used to contain the func-
// tion
// and user data associated with a DtReflectedObject callback,
class DtReflectedObjectCallbackInfo : public DtListedObj
{
public:
    // default constructor
    DtReflectedObjectCallbackInfo();

    // constructor
    DtReflectedObjectCallbackInfo(
        DtReflectedObjectCallbackFunction cb, void* userData);

    // destructor
    virtual ~DtReflectedObjectCallbackInfo();

    // copy constructor
    DtReflectedObjectCallbackInfo(const DtReflectedObjectCallbackInfo&
orig);

    // assignment operator
    DtReflectedObjectCallbackInfo& operator=(
        const DtReflectedObjectCallbackInfo& orig);

    // equality operators
    int operator==(const DtReflectedObjectCallbackInfo& orig) const;
    int operator!=(const DtReflectedObjectCallbackInfo& orig) const;

    // set/get the callback function
    virtual void setCallbackFunction(DtReflectedObjectCallbackFunction
funct);
    virtual DtReflectedObjectCallbackFunction callbackFunction() const;

    // set/get the user data
    virtual void setUserData(void* data);
    virtual void* userData() const;

protected:
    DtReflectedObjectCallbackFunction myCallbackFunction;
    void* myUserData;
};
```


refObjMgr.h

```

#if DtHLA

#include "callbacks.h"
#include "objIdHashLst.h"
#include "hlaObjFact.h"

// class DtReflectedHlaObjectManager:
//
// Instances of DtReflectedHlaObjectManager are used to manage the creation
// and
// destruction of HLA objects from the network.

class DtReflectedHlaObjectManager
{
public:
    // constructor
    DtReflectedHlaObjectManager(DtHlaObjectFactory* factory);

    // destructor
    virtual ~DtReflectedHlaObjectManager();

    // Add/remove a callback to be called when an object which meets the
    // federate's subscription criteria is discovered.
    virtual void addDiscoverObjectCallback(RTI::ObjectClassHandle handle,
        DtDiscoverObjectCb cb,
        void* usr);
    virtual void removeDiscoverObjectCallback(RTI::ObjectClassHandle handle,
        DtDiscoverObjectCb cb,
        void* usr);

    // Add/remove a callback to be called when an object has been
    // removed from the federation or no longer meets the interest
    // criteria of the federate.
    virtual void addRemoveObjectCallback(RTI::ObjectHandle id,
        DtRemoveObjectCb cb,
        void* usr);
    virtual void removeRemoveObjectCallback(RTI::ObjectHandle id,
        DtRemoveObjectCb cb,
        void* usr);

#if DtRTI103
    // Process the discovery of an object.
    // Creates the object, sets it up for reflecting, and
    // calls discover callbacks.
    virtual void processDiscoverObject(RTI::ObjectHandle id,
        RTI::ObjectClassHandle handle);
#else
    // Process the discovery of an object.
    // Creates the object, sets it up for reflecting, and
    // calls discover callbacks.
    virtual void processDiscoverObject(RTI::ObjectHandle id,
        RTI::ObjectClassHandle handle, const char* name);
#endif

    // Process the removal of an object.
    // Cleans up the object from reflecting.
    virtual void processRemoveObject(RTI::ObjectHandle id);

    // Process an update message.
    virtual void processReflectAttributeValues(const DtStateMsg &msg);

    // Get the list of all HLA objects.

```

```
virtual const DtList& allHlaObjects() const;

// Get the HLA object based on its identifier.
virtual DtHlaObject* hlaObject(RTI::ObjectHandle id) const;

protected:

// Call DtHlaObject::processRemoveObject().
// Provides friendship towards DtHlaObject to derived classes.
virtual void objectProcessRemoveObject(DtHlaObject* hlaObj);

// Call DtHlaObject::callReflect().
// Provides friendship towards DtHlaObject to derived classes.
virtual void objectProcessReflect(DtHlaObject* hlaObj,
    const DtStateMsg &msg);

protected:

// Note: The copy constructor and assignment operator are protected
// because there should only be one object manager per exercise
// connection.
// These should not be called.

// copy constructor
DtReflectedHlaObjectManager(const DtReflectedHlaObjectManager& orig);

// assignment operator
DtReflectedHlaObjectManager& operator=(
    const DtReflectedHlaObjectManager& orig);

protected:

// myDiscoverCbList is an array of DtHlaObject::theMaxObjectClasses
// items.
DtList* myDiscoverCbList;
DtHlaObjectFactory* myFactory;
DtObjectIdHashlist myObjectList;
};
#endif
```

regCtlCbInfo.h

```

#if DtHLA

#if !DtRTI103

#include "regCtlCbList.h"
#include "RTI.hh"
#include "stateMsg.h"

// class DtRegControlCbInfo:
//
// Instances of DtRegControlCbInfo are used for providing
// a callback when the RTI update control services are invoked.
// This class is a helper class to DtRegControlCbList.

class DtRegControlCbInfo
{
public:
    // constructor
    DtRegControlCbInfo(RTI::ObjectClassHandle h,
        DtRegControlCb cb,
        void* usr);

    // destructor
    virtual ~DtRegControlCbInfo();

    // copy constructor
    DtRegControlCbInfo(
        const DtRegControlCbInfo& orig);

    // assignment operator
    DtRegControlCbInfo& operator=(
        const DtRegControlCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(const DtRegControlCbInfo* const other) const;

    // Check whether one callback info is equal to another.
    virtual int operator==(const DtRegControlCbInfo& other) const;

    // Get the object class handle.
    virtual const RTI::ObjectClassHandle& objectClass() const;

    // Get the callback function.
    virtual const DtRegControlCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:
    RTI::ObjectClassHandle myClass;
    DtRegControlCb myFunc;
    void* myUsr;
};
typedef DtRegControlCbInfo DtStartRegsCbInfo;
typedef DtRegControlCbInfo DtStopRegsCbInfo;

#endif
#endif

```

regCtlCbList.h

```
#if DtHLA

#if !DtRTI103

#include "vllist.h"
#include "RTI.hh"

// class DtRegControlCbList:
//
// Instances of DtRegControlCbList are used to provide
// a list of callbacks (DtRegControlCbInfo) which are
// invoked when a registration control advisory is called by RTI.

typedef void (*DtRegControlCb)(
    RTI::ObjectClassHandle objectClass,
    void* usr);

typedef DtRegControlCb DtStartRegCb;
typedef DtRegControlCb DtStopRegCb;

class DtRegControlCbList
{
public:

    // default constructor
    DtRegControlCbList();

    // destructor
    virtual ~DtRegControlCbList();

    // copy constructor
    DtRegControlCbList(
        const DtRegControlCbList& orig);

    // assignment operator
    DtRegControlCbList& operator=(
        const DtRegControlCbList& orig);

    // Add/Remove a start updates callback to/from the list.
    // Returns whether add/remove took place.
    virtual int add(RTI::ObjectClassHandle h, DtRegControlCb cb,
        void* usr);
    virtual int remove(RTI::ObjectClassHandle h, DtRegControlCb cb,
        void* usr);

    // Remove all callbacks from the list.
    virtual void clear();

    // Callbacks are for a particular object class handle.
    // Call any callbacks in the list for the object class handle.
    virtual void apply(RTI::ObjectClassHandle objectClass);

protected:

    // Remove all callbacks from the list.
    // Non-virtual since meant to be called from dtor, otherwise
    // should call clear().
    void dtorClear();

    // Copy another callback list into this object's callback list.
    // Non-virtual since meant to be called from ctor.
    void ctorCopyList(const DtRegControlCbList& orig);

}
```

```
protected:
    DtList myList;
};
typedef DtRegControlCbList DtStartRegCbList;
typedef DtRegControlCbList DtStopRegCbList;
#endif
#endif
```

remObjCbInfo.h

```
#if DtHLA

#include "RTI.hh"
#include "stateMsg.h"
#include "vllist.h"

// class DtRemoveObjectCbInfo:
//
// Instances of DtRemoveObjectCbInfo are used for providing
// a callback when an object has been removed from the federation or
// no longer meets the interest criteria of the federate.
// This class is a helper class to DtRemoveObjectCbList.

typedef void (*DtRemoveObjectCb)(DtHlaObject* obj, void* usr);

class DtRemoveObjectCbInfo
{
public:

    // constructor
    DtRemoveObjectCbInfo(DtRemoveObjectCb cb, void* usr);

    // destructor
    virtual ~DtRemoveObjectCbInfo();

    // copy constructor
    DtRemoveObjectCbInfo(
        const DtRemoveObjectCbInfo& orig);

    // assignment operator
    DtRemoveObjectCbInfo& operator=(
        const DtRemoveObjectCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(const DtRemoveObjectCbInfo* const other) const;

    // Check whether one callback info is equal to another.
    virtual int operator==(const DtRemoveObjectCbInfo& other) const;

    // Get the callback function.
    virtual const DtRemoveObjectCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:

    DtRemoveObjectCb myFunc;
    void* myUsr;
};
#endif
```

removeEntRPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "simanPdu.h"
#include "packets/removeEntRP.h"
#include "exerciseConn.h"

// class DtRemoveEntityRPdu:
//
// Instances of DtRemoveEntityRPdu are used to send/receive
// reliable remove entity PDUs.

// DtRemoveEntityRPdu callback function signature.
typedef void (*DtRemoveEntityRPduCb)(DtRemoveEntityRPdu* pdu, void* usr);

class DtRemoveEntityRPdu : public DtSimanPdu
{
public:
    // Constructors
    DtRemoveEntityRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRemoveEntityRPdu(const DtNetRemoveEntityRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRemoveEntityRPdu(const DtEntityIdentifier& to,
        const DtEntityIdentifier& from,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRemoveEntityRPdu(const DtRemoveEntityRPdu& orig);

    // destructor
    virtual ~DtRemoveEntityRPdu();

    // Assignment operator
    DtRemoveEntityRPdu& operator=(const DtRemoveEntityRPdu& orig);

    // Set/Get the required reliability service.
    virtual void setReqRelService(DtRequiredReliabilityService value);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Set/Get the request id.
    virtual void setRequestId(int id);
    virtual int requestId() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:
    // Create a DtRemoveEntityRPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtRemoveEntityRPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtRemoveEntityRPduCb cb, void* usr);

protected:
    // Returns the internal PDU kind

```

```
    virtual DtPduKind internalGetPduKind() const;
protected:
    static int theStaticSize;
};
#endif
#endif
```


removePdu.h

```

#if !DtHLA

#include "hostStructs.h"
#include "simanPdu.h"

#define DtRemoveStaticSize 28

class DtRemoveEntityPdu;
typedef void (*DtRemoveEntityPduCb)(DtRemoveEntityPdu *pdu, void *usr);

class DtRemoveEntityPdu : public DtSimanPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtRemoveEntityPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRemoveEntityPdu(const DtNetRemoveEntityPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** ADDITIONAL CONSTRUCTORS *****/

    DtRemoveEntityPdu(const DtEntityIdentifier &to,
        const DtEntityIdentifier &from,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtRemoveEntityPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtRemoveEntityPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

#if DtDISVERS != 3
    int requestId() const;
#endif

/***** MUTATOR FUNCTIONS *****/

#if DtDISVERS != 3
    void setRequestId(int id);
    void printData() const;
#endif

protected:
    DtPduKind internalGetPduKind() const {return DtRemoveEntityPduKind;}
};
#endif

```

repairPdu.h

```
#if !DtHLA

#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "packets/logisticsP.h"

#define DtRepairResponseStaticSize 28

class DtRepairResponsePdu;
typedef void (*DtRepairResponsePduCb)(DtRepairResponsePdu *pdu, void *usr);

class DtRepairResponsePdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtRepairResponsePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRepairResponsePdu(const DtNetRepairResponsePdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtRepairResponsePduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtRepairResponsePduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &receivingId() const;
    const DtEntityIdentifier &repairingId() const;
    DtRepairResult repairResult() const;

/***** MUTATOR FUNCTIONS *****/

    void setReceivingId(const DtEntityIdentifier &id);
    void setRepairingId(const DtEntityIdentifier &id);
    void setRepairResult(DtRepairResult result);

    void printData() const;
protected:
    DtPduKind internalGetPduKind() const {return DtRepairResponsePduKind;}
    DtEntityIdentifier theId;
};
/***** REPAIR COMPLETE PDU *****/
/***** REPAIR COMPLETE PDU *****/

#define DtRepairCompleteStaticSize 28

typedef void (*DtRepairCompletePduCb)(DtRepairCompletePdu *pdu, void *usr);

class DtRepairCompletePdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtRepairCompletePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtRepairCompletePdu(const DtNetRepairCompletePdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
```

```
static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
static void addCallback(DtExerciseConn *conn,
DtRepairCompletePduCb cb, void *usr);
static void removeCallback(DtExerciseConn *conn,
DtRepairCompletePduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

const DtEntityIdentifier &receivingId() const;
const DtEntityIdentifier &repairingId() const;
DtRepairType repairType() const;

/***** MUTATOR FUNCTIONS *****/

void setReceivingId(const DtEntityIdentifier &id);
void setRepairingId(const DtEntityIdentifier &id);
void setRepairType(DtRepairType type);

void printData() const;
protected:
DtPduKind internalGetPduKind() const {return DtRepairCompletePduKind;}
DtEntityIdentifier theId;
};

#endif
```

resupplyPdu.h

```
#if !DtHLA

#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "packets/logisticsP.h"

#define DtResupplyCancelStaticSize 24

class DtResupplyCancelPdu;
typedef void (*DtResupplyCancelPduCb)(DtResupplyCancelPdu *pdu, void *usr);

class DtResupplyCancelPdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtResupplyCancelPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtResupplyCancelPdu(const DtNetResupplyCancelPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtResupplyCancelPdu(const DtEntityIdentifier &receiveId,
const DtEntityIdentifier &supplyId,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtResupplyCancelPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtResupplyCancelPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &receivingId() const;
    const DtEntityIdentifier &supplyingId() const;

/***** MUTATOR FUNCTIONS *****/

    void setReceivingId(const DtEntityIdentifier &id);
    void setSupplyingId(const DtEntityIdentifier &id);

    void printData() const;
protected:
    DtPduKind internalGetPduKind() const {return DtResupplyCancelPduKind;}
    DtEntityIdentifier theId;
};
/***** *****/
/***** RESUPPLY OFFER PDU *****/
/***** *****/

#define DtResupplyOfferStaticSize 28
#define DtSUPPLY_VALID 1
#define DtSUPPLY_BAD_INDEX 0

typedef void (*DtResupplyOfferPduCb)(DtResupplyOfferPdu *pdu, void *usr);

class DtResupplyOfferPdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/
```

```

    DtResupplyOfferPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtResupplyOfferPdu(const DtNetResupplyOfferPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** ADDITIONAL CONSTRUCTORS *****/

    DtResupplyOfferPdu(const DtEntityIdentifier &receiver,
        const DtEntityIdentifier &supplier,
        const DtEntityType &supply,
        float quantity,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtResupplyOfferPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtResupplyOfferPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &receivingId() const;
    const DtEntityIdentifier &supplyingId() const;
    int numSupplyTypes() const;
    const DtEntityType &nthSupply(int supplyIndex,
        float *quantity = NULL,
        int *retCode = NULL) const;
    float supplyQuantity(const DtEntityType &supply) const;

/***** MUTATOR FUNCTIONS *****/

    void setReceivingId(const DtEntityIdentifier &requester);
    void setSupplyingId(const DtEntityIdentifier &servicer);
    void setNumSupplyTypes(int numTypes);
    void setNthSupply(int supplyIndex,
        const DtEntityType &supply,
        float quantity,
        int *retCode = NULL);

    void printData() const;
protected:
    DtPduKind internalGetPduKind() const {return DtResupplyOfferPduKind;}
    int indexInBounds(int ind) const;
    DtEntityIdentifier theId;
    DtEntityType supType;
};

/***** RESUPPLY RECEIVED PDU *****/

#define DtResupplyReceivedStaticSize 28

typedef void (*DtResupplyReceivedPduCb)(DtResupplyReceivedPdu *pdu, void
*usr);

class DtResupplyReceivedPdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtResupplyReceivedPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtResupplyReceivedPdu(const DtNetResupplyReceivedPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** ADDITIONAL CONSTRUCTORS *****/

```

```
DtResupplyReceivedPdu(const DtEntityIdentifier &receiver,
    const DtEntityIdentifier &supplier,
    const DtEntityType &supply,
    float quantity,
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtResupplyReceivedPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtResupplyReceivedPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &receivingId() const;
    const DtEntityIdentifier &supplyingId() const;
    int numSupplyTypes() const;
    const DtEntityType &nthSupply(int supplyIndex,
float *quantity = NULL,
int *retCode = NULL) const;
    float supplyQuantity(const DtEntityType &supply) const;

/***** MUTATOR FUNCTIONS *****/

    void      setReceivingId(const DtEntityIdentifier &requester);
    void      setSupplyingId(const DtEntityIdentifier &servicer);
    void      setNumSupplyTypes(int numTypes);
    void      setNthSupply(int supplyIndex,
const DtEntityType &supply,
float quantity,
int *retCode = NULL);

    void printData() const;

protected:
    DtPduKind internalGetPduKind() const {return DtResupplyReceivedPdu-
Kind;}
    int indexInBounds(int ind) const;
    DtEntityIdentifier theId;
    DtEntityType supType;
};

#endif
```

rprDecFact.h

```
#if DtHLA
#include "decFactory.h"

// class DtRprStateDecoderFactory:
//
// Instances of DtRprStateDecoderFactory are used to create a Real-time
// Platform Reference (RPR) Federation Object Model (FOM) decoder.

class DtRprStateDecoderFactory : public DtStateDecoderFactory
{
public:
    // constructor
    DtRprStateDecoderFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtRprStateDecoderFactory();

    // copy constructor
    DtRprStateDecoderFactory(const DtRprStateDecoderFactory& orig);

    // assignment operator
    DtRprStateDecoderFactory& operator=(const DtRprStateDecoderFactory&
orig);
};
#endif
```

rprEncFact.h

```
#if DtHLA

#include "encFactory.h"

// class DtRprStateEncoderFactory:
//
// Instances of DtRprStateEncoderFactory are used to create encoders which
// know how to encode state repositories for RPR (Real-time Platform
// Reference) FOM specific objects into an HLA state message.

class DtRprStateEncoderFactory : public DtStateEncoderFactory
{
public:

    // constructor
    // Adds RPR FOM specific encoders.
    DtRprStateEncoderFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtRprStateEncoderFactory();

    // copy constructor
    DtRprStateEncoderFactory(const DtRprStateEncoderFactory& orig);

    // assignment operator
    DtRprStateEncoderFactory& operator=(
        const DtRprStateEncoderFactory& orig);
};
#endif
```


rprFomMap.h

```

#ifdef DtHLA
#include "fomMapper.h"
#include "chooseDefines.h"

// class DtRprFomMapper:
//
// Instances of DtRprFomMapper are DtFomMappers configured to map between
// VR-Link classes and functionality to RPR FOM classes,
// attributes and parameters.

class DtRprFomMapper : public DtFomMapper
{
public:
    // default constructor
    DtRprFomMapper(DtExerciseConn* exConn);

    // destructor
    virtual ~DtRprFomMapper();

public:
    // Creator function - returns new instance
    static DtFomMapper* create(DtExerciseConn* exConn);

protected:
    // copy constructor
    DtRprFomMapper(const DtRprFomMapper& orig);

    // assignment operator
    DtRprFomMapper& operator=(const DtRprFomMapper& orig);

    // The functions below are non-virtual since they are called by
    // the constructor.

    // Returns the default state decoder factory
    DtStateDecoderFactory* dfltStateDecoderFactory();

    // Returns the default state encoder factory
    DtStateEncoderFactory* dfltStateEncoderFactory();

    // Returns the default interaction decoder factory
    DtInteractionDecoderFactory* dfltInteractionDecoderFactory();

    // Returns the default state interaction factory
    DtInteractionEncoderFactory* dfltInteractionEncoderFactory();

    // Returns the default HlaObject factory
    DtHlaObjectFactory* dfltHlaObjectFactory();

    // Returns the default Interaction factory
    DtInteractionFactory* dfltInteractionFactory();

    // Returns default list of RTI::ObjectClassHandles that
    // indicate the set of objects to be interpreted as entities.
    DtList* dfltEntityObjectClasses();

    // Returns default list of RTI::ObjectClassHandles that
    // indicate the set of objects to be interpreted as aggregates.
    DtList* dfltAggregateObjectClasses();

```

```
// Set myInterClasses to contain default mappings.
void initInterClasses();

protected:

// The default object class chooser function.
static char* dfltEntityClassChooser(
    DtExerciseConn* exConn, const DtEntityType& type);

// The default object class chooser function.
static char* dfltAggClassChooser(
    DtExerciseConn* exConn, const DtEntityType& type);

// The default object class chooser function.
static char* dfltInterClassChooser(
    DtExerciseConn* exConn, const DtInteraction& inter);

protected:

// Default factories and other objects
DtStateDecoderFactory* myDfltDecFact;
DtStateEncoderFactory* myDfltEncFact;
DtInteractionDecoderFactory* myDfltInterDecFact;
DtInteractionEncoderFactory* myDfltInterEncFact;
DtHlaObjectFactory* myDfltObjFact;
DtInteractionFactory* myDfltInterFact;
DtList myDfltEntityClassList;
DtList myDfltAggClassList;

// Lists of Interaction Class Handles.
DtList myFireClasses;
DtList myDetClasses;
DtList myCollClasses;
DtList myAckClasses;
DtList myStartClasses;
DtList myStopClasses;
DtList myActReqClasses;
DtList mySignalClasses;
DtList myVcClasses;
DtList myLcClasses;
};
#endif
```

rprlDecFact.h

```
#if DtHLA

#include "intDecFact.h"

// class DtRprInteractionDecoderFactory:
//
// Instances of DtRprInteractionDecoderFactory are
// DtInteractionDecoderFactories configured with the VR-Link's
// default set of Encoders.

class DtRprInteractionDecoderFactory : public DtInteractionDecoderFactory
{
public:
    // constructor
    DtRprInteractionDecoderFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtRprInteractionDecoderFactory();

    // copy constructor
    DtRprInteractionDecoderFactory(
        const DtRprInteractionDecoderFactory& orig);

    // assignment operator
    DtRprInteractionDecoderFactory& operator=(
        const DtRprInteractionDecoderFactory& orig);
};
#endif
```

rprlEncFact.h

```
#if DtHLA

#include "intDecFact.h"

// class DtRprInteractionDecoderFactory:
//
// Instances of DtRprInteractionDecoderFactory are
// DtInteractionDecoderFactories configured with the VR-Link's
// default set of Encoders.

class DtRprInteractionDecoderFactory : public DtInteractionDecoderFactory
{
public:
    // constructor
    DtRprInteractionDecoderFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtRprInteractionDecoderFactory();

    // copy constructor
    DtRprInteractionDecoderFactory(
        const DtRprInteractionDecoderFactory& orig);

    // assignment operator
    DtRprInteractionDecoderFactory& operator=(
        const DtRprInteractionDecoderFactory& orig);
};
#endif
```

rprInterFact.h

```
#if DtHLA

#include "hInterFactory.h"

// class DtRprInteractionFactory:
//
// Instances of DtRprInteractionFactory are used to create RPR
// (Real-time Platform Reference) FOM interactions.

class DtRprInteractionFactory : public DtInteractionFactory
{
public:
    // constructor
    // Adds creators for the RPR FOM interactions.
    DtRprInteractionFactory(DtExerciseConn* exConn);

    // destructor
    virtual ~DtRprInteractionFactory();

    // copy constructor
    DtRprInteractionFactory(const DtRprInteractionFactory& orig);

    // assignment operator
    DtRprInteractionFactory& operator=(const DtRprInteractionFactory& orig);
};
#endif
```

rprObjFact.h

```
#if DtHLA

#include "hlaObjFact.h"
#include "chooseDefines.h"

// class DtRprHlaObjectFactory:
//
// Instances of DtRprHlaObjectFactory are used to create RPR (Real-time
// Platform Reference) FOM HLA objects.

class DtRprHlaObjectFactory : public DtHlaObjectFactory
{
public:

    // constructor
    DtRprHlaObjectFactory(DtExerciseConn* conn);

    // destructor
    virtual ~DtRprHlaObjectFactory();

    // copy constructor
    DtRprHlaObjectFactory(const DtRprHlaObjectFactory& orig);

    // assignment operator
    DtRprHlaObjectFactory& operator=(const DtRprHlaObjectFactory& orig);
};
#endif
```

script.h

```
#include "vllist.h"
#include "vcPdu.h"

typedef struct
{
    DtTime t;
    DtViewControlPdu *pdu;
} StealthInstruction;

/* A DtScript is a DtList of StealthInstructions */
class DtScript : public DtList
{
public:
    DtScript(char *fname);
    DtScript() {}
    ~DtScript();
    void addInstruction(DtTime time, DtViewerCommand command,
char *args);
    void addInstruction(DtTime time, DtViewControlPdu *pdu);
    void clear();
};
```

servReqPdu.h

```
#if !DthLA

#ifndef DtServiceRequestPdu_H_
#define DtServiceRequestPdu_H_

#include "pdu.h"
#include "packets/logisticsP.h"
#include "hostStructs.h"

#define DtServiceRequestStaticSize 28
#define DtSUPPLY_VALID 1
#define DtSUPPLY_BAD_INDEX 0

class DtServiceRequestPdu;
typedef void (*DtServiceRequestPduCb)(DtServiceRequestPdu *pdu, void *usr);

class DtServiceRequestPdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtServiceRequestPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtServiceRequestPdu(const DtNetServiceRequestPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtServiceRequestPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtServiceRequestPduCb cb, void *usr);

/***** ADDITIONAL CONSTRUCTORS *****/

    DtServiceRequestPdu(const DtEntityIdentifier &requester,
const DtEntityIdentifier &servicer,

#ifdef _DOS
/* This is some screwed up stuff. This is a hack that will go down in
infamy. This is how the world ends, not with a hack, but with a
compiler bug. Someday, children, all the weirdness will be gone, and
this ifdef can be taken out. */
DtServiceType serviceType = (DtServiceType) 2,
#else
DtServiceType serviceType = DtServiceTypeRepair,
#endif
DtBufferPtr another = DtUSE_INTERNAL_BUFFER);

    DtServiceRequestPdu(const DtEntityIdentifier &requester,
const DtEntityIdentifier &servicer,
const DtEntityType &supply,
float quantity,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &requestingId() const;
    const DtEntityIdentifier &servicingId() const;
    DtServiceType serviceType() const;
    int numSupplyTypes() const;
    const DtEntityType &nthSupply(int supplyIndex,
```



```
float *quantity = NULL,
int *retCode = NULL) const;
    float supplyQuantity(const DtEntityType &supply) const;

/***** MUTATOR FUNCTIONS *****/

    void      setRequestingId(const DtEntityIdentifier &requester);
    void      setServicingId(const DtEntityIdentifier &servicer);
    void      setServiceType(DtServiceType service);
    void      setNumSupplyTypes(int numTypes);
    void      setNthSupply(int supplyIndex,
const DtEntityType &supply,
float quantity,
int *retCode = NULL);

    void printData() const;
protected:
    DtPduKind internalGetPduKind() const {return DtServiceRequestPduKind;}
    int indexInBounds(int ind) const;
    DtEntityIdentifier reqId;
    DtEntityIdentifier servId;
    DtEntityType supType;
};
#endif
```

setDataPdu.h

```
#if !DtHLA

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"

#define DtSetDataStaticSize 40

class DtSetDataPdu;
typedef void (*DtSetDataPduCb)(DtSetDataPdu *pdu, void *usr);

class DtSetDataPdu : public DtPduWithData
{
public:
/***** STANDARD CONSTRUCTORS *****/

    DtSetDataPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtSetDataPdu(const DtNetSetDataPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtSetDataPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtSetDataPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    unsigned long requestId() const;

/***** MUTATOR FUNCTIONS *****/

    void setRequestId(unsigned long id);

    void printData() const;

protected:

    DtPduKind internalGetPduKind() const {return DtSetDataPduKind;}
    DtNetDataPduTail *dataSection() const;
};
#endif
```

setDataRPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "pduWithData.h"
#include "datumIdType.h"
#include "exerciseConn.h"
#include "packets/setDataRP.h"

// class DtSetDataRPdu:
//
// Instances of DtSetDataRPdu are used to send/receive
// reliable set data PDUs.

// DtSetDataRPdu callback function signature.
typedef void (*DtSetDataRPduCb)(DtSetDataRPdu* pdu, void* usr);

class DtSetDataRPdu : public DtPduWithData
{
public:

    // Constructors
    DtSetDataRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtSetDataRPdu(const DtNetSetDataRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtSetDataRPdu(const DtSetDataRPdu& orig);

    // destructor
    virtual ~DtSetDataRPdu();

    // Assignment operator
    DtSetDataRPdu& operator=(const DtSetDataRPdu& orig);

    // Set/Get the request id.
    virtual void setRequestId(unsigned long id);
    virtual unsigned long requestId() const;

    // Set/Get the required reliability service.
    virtual void setReqRelService(DtRequiredReliabilityService value);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:

    // Create a DtSetDataRPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtSetDataRPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtSetDataRPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

```

```
// Returns the data portion of the net structure.
virtual DtNetDataPduTail* dataSection() const;

protected:
    static int theStaticSize;
};
#endif
#endif
```

setRecordPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "recordSetPdu.h"
#include "exerciseConn.h"

#include "packets/setRecordP.h"

// class DtSetRecordPdu:
//
// Instances of DtSetRecordPdu are used to initialize or change internal
// parameter values using record sets.

// DtSetRecordPdu callback function signature.
typedef void (*DtSetRecordPduCb)(DtSetRecordPdu* pdu, void* usr);

class DtSetRecordPdu : public DtRecordSetPdu
{
public:
    // Constructors
    DtSetRecordPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtSetRecordPdu(const DtNetSetRecordPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtSetRecordPdu(const DtSetRecordPdu& orig);

    // destructor
    virtual ~DtSetRecordPdu();

    // Assignment operator
    DtSetRecordPdu& operator=(const DtSetRecordPdu& orig);

    // Set/Get originating entity identifier
    virtual void setOriginatingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& originatingEntityID() const;

    // Set/Get receiving entity identifier
    virtual void setReceivingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& receivingEntityID() const;

    // Set/Get request ID
    virtual void setRequestID(DtU32 id);
    virtual DtU32 requestID() const;

    // Set/Get required reliability service
    virtual void setReqRelService(enum DtRequiredReliabilityService
service);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:
    // Create a DtSetRecordPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when

```

```
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtSetRecordPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtSetRecordPduCb cb, void* usr);

protected:

    // Returns a pointer to the beginning of the record set data struct.
    virtual DtNetRecordSetPdu* netRecordSetPduStruct() const;

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    // Non-virtual since you can't override based on return type
    virtual DtNetSetRecordPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non virtual since you can't override based on return type
    DtSetRecordPdu* self() const;

protected:

    // Size of PDU with zero data bytes.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;

};
#endif
#endif
```

signalPdu.h

```

#if !DtHLA

#include "hostStructs.h"
#include "pdu.h"
#include "packets/signalP.h"
#include "interaction.h"
#define DtSignalStaticSize 32

class DtSignalPdu;
typedef DtSignalPdu DtSignalInteraction;
typedef void (*DtSignalInteractionCb)(DtSignalInteraction *pdu, void *usr);

class DtSignalPdu : public DtInteraction
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtSignalPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtSignalPdu(const DtNetSignalPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtSignalInteractionCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtSignalInteractionCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    const DtEntityIdentifier &entityId() const;
    unsigned int radioId() const;
    DtEncodingClass encodingClass() const;
    unsigned int encodingType() const;
    long sampleRate() const;
    int numDataBits() const;
    int numDataBytes() const;
    int numSamples() const;
    unsigned char dataByte(int byteNum) const;
    void getData(unsigned char *buff) const;
#if DtDISVERS != 3
    int tdlType() const;
#endif

/***** MUTATOR FUNCTIONS *****/

    void setEntityId(const DtEntityIdentifier &id);
    void setRadioId(unsigned int id);
    void setEncodingClass(DtEncodingClass encodingClass);
    void setEncodingType(unsigned int codeType);
    void setSampleRate(long rate);
    void setNumDataBits(int lengthInBits);
    void setNumDataBytes(int lengthInBytes);
    void setNumSamples(int numSamples);
    void setDataByte(int byteNum, unsigned char data);
    void setData(unsigned char *data, int numBytes);
#if DtDISVERS != 3
    void setTdlType(int type);
#endif

    void printData() const;
protected:

```

```
        DtPduKind internalGetPduKind() const {return DtSignalPduKind;}
        DtEntityIdentifier theId;
    };
#endif // !DtHLA
```


simanPdu.h

```
#if !DtHLA

#include "pdu.h"
#include "packets/simanP.h"
#include "hostStructs.h"
#include "interaction.h"

class DtSimanPdu : public DtInteraction
{
public:

/***** INSPECTOR FUNCTIONS *****/

    virtual const DtEntityIdentifier &senderId() const;
    virtual const DtEntityIdentifier &receiverId() const;

/***** MUTATOR FUNCTIONS *****/

    virtual void setSenderId(const DtEntityIdentifier &id);
    virtual void setReceiverId(const DtEntityIdentifier &id);

    virtual void printData() const;

protected:
    DtEntityIdentifier cache;
};
#endif
```

simplIntWED.h

```
#if DtHLA

#include "RTI.hh"
#include "hIntWEncDec.h"

// class DtSimpleInteractionWithEncDec:
//
// DtSimpleInteractionWithEncDec is a DtInteractionWithEncDec
// for which there is a direct mapping to a single InteractionClass
// in the FOM.

class DtSimpleInteractionWithEncDec : public DtInteractionWithEncDec
{
public:

    // default constructor
    DtSimpleInteractionWithEncDec();

    // destructor
    virtual ~DtSimpleInteractionWithEncDec();

    // copy constructor
    DtSimpleInteractionWithEncDec(const DtSimpleInteractionWithEncDec&
orig);

    // assignment operator
    DtSimpleInteractionWithEncDec& operator=(const DtSimpleInteractionWith-
EncDec& orig);

    // Sets the interaction's ExerciseConn and classDesc. If classDesc is
NULL,
    // uses the descriptor associated with the class name returned by class-
ToUse.
    // Also calls useThese, passing encoderToUse(), decoderToUse() and this.
    // (Without being told what class descriptor to use to represent the
    // interaction, many of the HLA-specific functions won't work properly.)
    virtual void setExConn(DtExerciseConn* exConn, DtInterClassDesc* class-
Desc);

    // Returns the FOM interaction class name to use for a particular
    // derived DtSimpleInteraction.
    virtual RTI::InteractionClassName classToUse() const = 0;

    // Returns the encoder to use for decoding the interaction
    // Result is passed to useThese by setExConn
    virtual DtInteractionDecoder* decoderToUse() const = 0;

    // Returns the decoder to use for decoding the interaction
    // Result is passed to useThese by setExConn
    virtual DtInteractionEncoder* encoderToUse() const = 0;
};
#endif
```

smoother.h

```

#include "deadReckoner.h"
#include "smoothPosWDR.h"
#include "smoothOriWDR.h"

// class DtSmoother:
//
// Instances of DtSmoother are used to provide approximated
// position and orientation information for entities.
// Position and orientation information is approximated over time
// from the last set absolute (non-approximated) information.
// When a new position is set, the current position moves smoothly
// to reflect the new information, rather than jumping to the new
// absolute position.

class DtSmoother : public DtDeadReckoner
{
public:
    // constructor
    DtSmoother();

    // destructor
    virtual ~DtSmoother();

    // copy constructor
    DtSmoother(const DtSmoother& orig);

    // assignment operator
    DtSmoother& operator=(const DtSmoother& orig);

    // Create a copy of this object. Memory is allocated and it is the
    // responsibility of the caller to free the memory via delete.
    virtual DtApproximator* clone() const;

    // Set absolute location.
    virtual void setLocation(DtConstVector loc,
        DtTime t = DtGetSimTime());
    // Get the approximated location at the specified time.
    virtual DtConstVector approxLocation(
        DtTime t = DtGetSimTime()) const;

    // Set absolute velocity.
    virtual void setVelocity(DtConstVector vel,
        DtTime t = DtGetSimTime());
    // Get the approximated velocity at the specified time.
    virtual DtConstVector approxVelocity(
        DtTime t = DtGetSimTime()) const;

    // Set absolute orientation.
    virtual void setOrientation(const DtTaitBryan& ori,
        DtTime t = DtGetSimTime());
    // Get approximated orientation (euler angles) at the specified time.
    virtual const DtTaitBryan& approxEuler(
        DtTime t = DtGetSimTime()) const;

    // Set/Get absolute acceleration.
    virtual void setAcceleration(DtConstVector acc,
        DtTime t = DtGetSimTime());
    virtual DtConstVector acceleration(
        DtTime t = DtGetSimTime()) const;

    // Set/Get absolute rotational velocity.
    virtual void setRotationalVelocity(DtConstVector vel,

```

```
DtTime t = DtGetSimTime());
virtual DtConstVector rotationalVelocity(
    DtTime t = DtGetSimTime()) const;

// Get the approximated data body to geocentric coordinate matrix.
virtual DtConstDcm approxBodyToGeoc(
    DtTime t = DtGetSimTime()) const;

// Set/Get the individual smoothing switch. In order for smoothing
// to be enabled, the "master" smoothing-enabled switch must be on, AND
// the individual smoothing switch (accessed through these functions),
// must be on. Individual smoothing switch defaults to on.
virtual void setSmoothEnable(DtBoolean on);
virtual DtBoolean smoothEnable() const;

// Set/Get smoothing period - the amount of time over
// which smoothing takes place. Pass a value of -1 to indicate
// use of the default smoothing period.
virtual void setSmoothPeriod(DtTime smoothPeriod);
virtual DtTime smoothPeriod() const;

public:

    // Set/Get the default smoothing period
    static void setDfltSmoothPeriod(DtTime smoothPeriod);
    static DtTime dfltSmoothPeriod();

    // Set/Get the value of the "master" smoothing-enabled switch.
    // Master smoothing switch defaults to on.
    static void setMasterSmoothEnable(DtBoolean onOff);
    static DtBoolean masterSmoothEnable();

    static DtTime theDfltSmoothPeriod;

protected:

    // provide inputs to the actual smoothing algorithm
    virtual void feedSmoothing();

    // Get rid of constness on the "this" pointer in order to access
    // non-const functions from const member functions (such as simple
    // accessors). This should be used sparingly since it bypasses
    // the const protection.
    // Note: This is essentially like an operator and is
    //       non-virtual since can't override based on return type.
    //       Allows derived classes to also define this member.
    DtSmoother* self() const;

protected:

    DtSmoothPosWithDr mySpos;
    DtSmoothOriWithDr mySorient;
};
```

smoothOriWDR.h

```
#include "smoothUtil.h"
// class DtSmoothOriWithDr:
//
// Instances of DtSmoothOriWithDr are

class DtSmoothOriWithDr : public DtSmoothOrient
{
public:
    // default constructor
    DtSmoothOriWithDr(DtDeadReckoner* dr, DtTime smoothT);

    // destructor
    virtual ~DtSmoothOriWithDr();

protected:
    // copy constructor
    DtSmoothOriWithDr(const DtSmoothOriWithDr& orig);

    // assignment operator
    DtSmoothOriWithDr& operator=(const DtSmoothOriWithDr& orig);

    // The default action is to dead-reckon
    virtual void dfltAction(DtTime simTime,
        void *usr, DtDcmRef os);
};
```

smoothPosWDR.h

```
#include "smoothUtil.h"

// class DtSmoothPosWithDr:
//
// Instances of DtSmoothPosWithDr are

class DtSmoothPosWithDr : public DtSmoothPos
{
public:
    // default constructor
    DtSmoothPosWithDr(DtDeadReckoner* dr, DtTime smoothT);

    // destructor
    virtual ~DtSmoothPosWithDr();

protected:
    // copy constructor
    DtSmoothPosWithDr(const DtSmoothPosWithDr& orig);

    // assignment operator
    DtSmoothPosWithDr& operator=(const DtSmoothPosWithDr& orig);

    // The default action is to dead-reckon
    virtual void dfltAction(DtTime simTime, void* usr,
        DtVectorRef vs, DtVectorRef ps);
};
```

smoothUtil.h

```

#include "mathTypes.h"
#include "vlTime.h"
#include "vlprint.h"

// ----- POSITION -----

class DtSmoothPos
{
public:
    // Constructors. smoothT is the initial smoothPeriod. Use a value of
    // -1 to indicate use of the default period.
    DtSmoothPos(void *usr, DtTime smoothT,
        DtConstVector pos, DtConstVector vel, DtConstVector acc);
    /* WARNING: not fully constructed until input() & tick() are called */
    DtSmoothPos(void *usr, DtTime smoothT);

    virtual ~DtSmoothPos() { };

    void input(DtConstVector pos, DtConstVector vel, DtConstVector acc,
        DtTime simTime, int freeze=0);
    void tick(DtTime simTime);

    DtConstVector position() const { return ps; }
    DtConstVector velocity() const { return vs; }

    // Set/Get the individual smoothing switch. In order for smoothing
    // to be enabled, the "master" smoothing-enabled switch must be on, AND
    // the individual smoothing switch (accessed through these functions),
    // must be on.
    virtual void setSmoothEnable(int on);
    virtual int smoothEnable() const;

    // Set/Get smoothPeriod. If set to -1, smooth period reverts to the
    // current default smooth period.
    virtual void setSmoothPeriod(DtTime t);
    virtual DtTime smoothPeriod() const;

public:
    // Set/Get default smooth period
    static void setDfltSmoothPeriod(DtTime t);
    static DtTime dfltSmoothPeriod();

    // Set/Get the value of the "master" smoothing-enabled switch.
    static void setMasterSmoothEnable(DtBoolean onOff);
    static DtBoolean masterSmoothEnable();

protected:
    DtSmoothPos(const DtSmoothPos&);

    virtual void dfltAction(DtTime simTime, void *usr,
        DtVectorRef vs, DtVectorRef ps) = 0;
    void init(void *u, DtTime t);
    void setT1data();

protected:
    int frozen;

    DtTime t0, t1, t2, lastT;

```

```
DtTime myLocalTau;
DtTime myLocalHtau;
DtTime myLocalTauInv;
DtTime* myTau;
DtTime* myHtau;
DtTime* myTauInv;

DtVector ps, vs, p0, v0, dp, dv, p1, v1, a1, a2, ap;
int myEnable;
int needT1data, needT2data;
void *usr;

protected:

    static DtTime theDfltTau;
    static DtTime theDfltHtau;
    static DtTime theDfltTauInv;
    static DtBoolean theMasterSmoothEnable;
};
// ----- ORIENTATION -----

class DtSmoothOrient
{
public:
    // Constructors. smoothT is the initial smoothPeriod. Use a value of
    // -1 to indicate use of the default period.
    DtSmoothOrient(void *usr, DtTime smoothT,
        DtConstDcm orient, const DtDirectedRotRate &drr);
    /* WARNING: not fully constructed until input() & tick() are called */
    DtSmoothOrient(void *usr, DtTime smoothT);

    void input(DtConstDcm orient, const DtDirectedRotRate &drr,
        DtTime simTime, int freeze=0);
    void tick(DtTime simTime);

    DtConstDcm orientation() const { return os; }

    // Set/Get the individual smoothing switch. In order for smoothing
    // to be enabled, the "master" smoothing-enabled switch must be on, AND
    // the individual smoothing switch (accessed through these functions),
    // must be on.
    virtual void setSmoothEnable(int on);
    virtual int smoothEnable() const;

    // Set/Get smoothPeriod. If set to -1, smooth period reverts to the
    // current default smooth period.
    virtual void setSmoothPeriod(DtTime t);
    virtual DtTime smoothPeriod() const;

public:
    // Set/Get default smooth period
    static void setDfltSmoothPeriod(DtTime t);
    static DtTime dfltSmoothPeriod();

    // Set/Get the value of the "master" smoothing-enabled switch.
    static void setMasterSmoothEnable(DtBoolean onOff);
    static DtBoolean masterSmoothEnable();

protected:

    virtual void dfltAction(DtTime simTime, void *usr, DtDcmRef os) = 0;
    void init(void *u, DtTime t);
    void setT1data();

protected:
```



```
int frozen;
DtTime t0, t2, lastT;
DtTime* myTau;
DtTime myLocalTau;
DtDcm os, o0;
DtVector ws;
int myEnable;
void *usr;

protected:

    static DtTime theDfltTau;
    static DtBoolean theMasterSmoothEnable;
};
```

smUtil.h

```
#if !DtHLA

#include "hostStructs.h"
#include "ackPdu.h"

/* options for DtAcknowledgePredicate */
class DtAckPredOpt {
public:
    DtAckPredOpt() :
        negAckRetVal(0),
        verboseToOther(0),
        verboseFromOther(0),
        verboseOtherType(0),
        verboseNegAck(1),
        verbosePosAck(0) { }

    DtEntityIdentifier to;
    DtEntityIdentifier from;
    DtAcknowledgeFlag ackFlag;
    int negAckRetVal;
    char verboseToOther;
    char verboseFromOther;
    char verboseOtherType;
    char verboseNegAck;
    char verbosePosAck;
};

/* for use with Connection::DtReadUntil */
int DtAcknowledgePredicate(DtPdu *pdu, void *arg);

/* Return 1 if to, from, and ackFlag match between pkt and arg and
 * positive response. Return negAckRetVal if match w/ neg Reponse.
 * Return 0 otherwise.
 * If (from.DtEntity == DtDeferredEntityId and hosts match), then
 * from.DtEntity will be set to from.DtEntity in PDU.
 */
int DtAcknowledgePredicate(DtAcknowledgePdu *pdu, DtAckPredOpt *opt);

#endif
```

Socket.h

```

/* DtSocket is a misnomer
   This provides net-level Packet IO
   Sometimes, it is implemented with sockets, sometimes not */

#ifndef _Socket_h
#define _Socket_h

#include "vlconfig.h"

#if DT_HAVE_STD_H
#include <std.h>
#endif
#include <stdio.h>

#include <sys/types.h>

#include "vlutil.h"
#include "vlList.h"

/*
 * IP + UDP headers are 20 + 8.
 * Ethernet MTU is 1500 (or 1492 for 802.3 encapsulation),
 * thus PDU limits will typically be 1472 or 1464
 * FDDI MTU is 4352 (PDU limit 4324)
 * IEEE 802.5 token ring MTU is 4464 (PDU limit of 4436)
 */
#define DtMaxPacketSize 4440 /* multiple of 8 bytes */
#define DtDEFAULT_PORT 3000
#define DtUSE_DEFAULT_IP_ADDR (DtInetAddr(0))

#define DtSOCKET_PACKET 1
#define DtSOCKET_NO_PACKET 0
#define DtSOCKET_SERVER_MSG -1
#define DtSOCKET_FILTERED_PACKET -2

enum DtTestOperator { Dto_EQ, Dto_NEQ } ;

enum DtFilterType { DtFtInetAddr, DtFtNetU8 };

class DtFilterTest
{
public:
    DtFilterTest() { disableTest();}
    virtual ~DtFilterTest() {}

    void disableTest() { setTest(Dto_NEQ, 0, NULL);}
    void setTest(DtTestOperator op, DtU32 value) { setTest(op, 1, &value);}
    void extendTest(DtU32 value) { extendTest (1, &value);}
    void reduceTest(DtU32 value) { reduceTest (1, &value);}

    virtual void setOperator(DtTestOperator op);
    virtual void setTest(DtTestOperator op, int n, DtU32 *values);
    virtual void extendTest(int n, DtU32 *values);
    virtual void reduceTest(int n, DtU32 *values);

    virtual DtBoolean evalTest(char *buf, int offset, DtFilterType type);

protected:
    virtual void clearList();

```

```
DtTestOperator testOp;
DtList values;
};

class DtSocket
{
public:

    DtSocket();

    virtual ~DtSocket() {}

    virtual int DtSend(caddr_t packet, size_t size,
        DtInetAddr addr = DtUSE_DEFAULT_IP_ADDR);
    virtual int sendTo(caddr_t packet, size_t size,
        DtInetAddr addr = DtUSE_DEFAULT_IP_ADDR) = 0;

    /* Depending on the implementation in derived DtSockets, one of the
     * forms of DtRecv may be faster than the other.
     *
     * NOTE: if a derived class complains "... hides virtual DtRecv() ...",
     * make sure you have BOTH overloads of DtRecv declared and defined
     * in the subclass.
     *
     * DtRecv has an optional status argument that could more elegantly be
     * implemented as different return codes, but this would break older
code.
     */
    virtual int DtRecv(caddr_t *buffptr, int *status=NULL) = 0;
    virtual int DtRecv(caddr_t buff, size_t) = 0;
    /* for those that don't care about length: */
    caddr_t DtRecv() { caddr_t bp; DtRecv(&bp); return bp; }

    int flush();
    void setBundling(int maxSize);
    int isBundling() {return maxBundleSize;}
    virtual void setUnbundling(int) {};

    virtual DtInetAddr lastSourceInetAddr() { return 0L; }
    virtual int DtReportPktsRcvd(void) { return (pkts_rcvd); }
    virtual void DtBecomeNonBlocking(void) {};
    virtual void notifyMe() {};

    virtual int listenToMulticastGroup(DtInetAddr) {return 0;}
    virtual int dropMulticastGroup(DtInetAddr) {return 0;}
    virtual int setMcastTtl(int) {return 0;}

    virtual DtFilterTest *exerciseId()      { return & version; }
    virtual DtFilterTest *protocolVersion() { return & exercise; }
    virtual DtFilterTest *pduKind()         { return & kind; }
    virtual DtFilterTest *ipAddr()          { return & addr; }

    int blocking;
    int pkts_rcvd;

    /*
     * not all DtSocket have associated file descriptors
     * those that do not will have readFd()/writeFd() return -1
     */
    virtual int readFd() { return -1; }
    virtual int writeFd() { return -1; }
    virtual int isOk() { return 1; }

protected:
```

```
    virtual DtBoolean accept(char *buf, DtInetAddr iaddr) {
return (protocolVersion()->evalTest(buf, 0, DtFtNetU8) &&
        exerciseId()->evalTest      (buf, 1, DtFtNetU8) &&
        pduKind()->evalTest          (buf, 2, DtFtNetU8) &&
        ipAddr()->evalTest            ((char *)&iaddr, 0, DtFtInetAddr))?

        DtTrue : DtFalse; }
    char outBuff[DtMaxPacketSize];
    int outSize;
    int bundling;
    int maxBundleSize;

private:
    DtFilterTest version;
    DtFilterTest exercise;
    DtFilterTest kind;
    DtFilterTest addr;
};

DtSocket *DtNewDfltSocket(int port, DtInetAddr destAddr = 0);

#endif
```

soldierDec.h

```
#if DtHLA

#include "milEntDec.h"

// class DtSoldierDecoder:
//
// Instances of DtSoldierDecoder are used to decode a
// soldier entity state message into a state repository in an HLA
// (as opposed to DIS) environment.

class DtSoldierDecoder : public DtMilitaryEntityDecoder
{
public:
    // constructor
    DtSoldierDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtSoldierDecoder();

    // copy constructor
    DtSoldierDecoder(const DtSoldierDecoder& orig);

    // assignment operator
    DtSoldierDecoder& operator=(const DtSoldierDecoder& orig);

public:
    // Create a DtSoldierDecoder.
    // User is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
    DtDECLARE_ENTITY_ATTR_DECODER(PrimaryWeaponState);
    DtDECLARE_ENTITY_ATTR_DECODER(SecondaryWeaponState);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_SOLDIER_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtSoldierDecoder, DtEntityStateRepository, \
        attrName, netType, mutator, castExpr)

#define DtDEFINE_SIMPLE_SOLDIER_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_SOLDIER_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))

#endif
```

soldierEnc.h

```

#if DtHLA

#include "milEntEnc.h"
#include "entitySR.h"

// class DtSoldierEncoder:
//
// Instances of DtSoldierEncoder are used to encode a soldier entity's
// state repository into a state message in an HLA environment.

class DtSoldierEncoder : public DtMilitaryEntityEncoder
{
public:
    // constructor
    DtSoldierEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtSoldierEncoder();

    // copy constructor
    DtSoldierEncoder(const DtSoldierEncoder& orig);

    // assignment operator
    DtSoldierEncoder& operator=(const DtSoldierEncoder& orig);

public:
    // Create a DtSoldierEncoder.
    // Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // static need<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_CHECKER(PrimaryWeaponState);
    DtDECLARE_ENTITY_ATTR_CHECKER(SecondaryWeaponState);

    // static decode<Attribute> member functions.
    DtDECLARE_ENTITY_ATTR_ENCODER(PrimaryWeaponState);
    DtDECLARE_ENTITY_ATTR_ENCODER(SecondaryWeaponState);
};
// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_SOLDIER_ATTR_ENCODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_ATTR_ENCODER(DtSoldierEncoder, \
        DtEntityStateRepository, \
        attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that checker whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_SOLDIER_ATTR_CHECKER( \

```

```
    attrName, inspector) \
DtDEFINE_SIMPLE_ATTR_CHECKER(DtSoldierEncoder, \
    DtEntityStateRepository, \
    attrName, inspector)
#endif
```


startPdu.h

```

#if !DtHLA

#include "hostStructs.h"
#include "simanPdu.h"

#define DtStartStaticSize 44

class DtStartResumePdu;
typedef void (*DtStartResumePduCb)(DtStartResumePdu *pdu, void *usr);

class DtStartResumePdu : public DtSimanPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtStartResumePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtStartResumePdu(const DtNetStartResumePdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** ADDITIONAL CONSTRUCTORS *****/

    DtStartResumePdu(const DtEntityIdentifier &to,
        const DtEntityIdentifier &from,
        const DtTime realTime,
        const DtTime simTime,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtStartResumePduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtStartResumePduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    DtTime realWorldTime() const;
    DtTime simulationTime() const;
#if DtDISVERS != 3
    int requestId() const;
#endif

/***** MUTATOR FUNCTIONS *****/

    void setRealWorldTime(const DtTime time);
    void setSimulationTime(const DtTime time);
#if DtDISVERS != 3
    void setRequestId(int id);
#endif

    void printData() const;

protected:
    DtPduKind internalGetPduKind() const {return DtStartResumePduKind;}
};
#endif

```

startResRPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "hostStructs.h"
#include "simanPdu.h"
#include "packets/startResRP.h"
#include "exerciseConn.h"

// class DtStartResumeRPdu:
//
// Instances of DtStartResumeRPdu are used to send/receive
// reliable start/resume PDUs.

typedef void (*DtStartResumeRPduCb)(DtStartResumeRPdu* pdu, void* usr);

class DtStartResumeRPdu : public DtSimanPdu
{
public:

    // Constructors
    DtStartResumeRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtStartResumeRPdu(const DtNetStartResumeRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtStartResumeRPdu(const DtEntityIdentifier& to,
        const DtEntityIdentifier& from,
        const DtTime realTime,
        const DtTime simTime,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtStartResumeRPdu(const DtStartResumeRPdu& orig);

    // destructor
    virtual ~DtStartResumeRPdu();

    // Assignment operator
    DtStartResumeRPdu& operator=(const DtStartResumeRPdu& orig);

    // Set/Get real world time.
    virtual void setRealWorldTime(const DtTime time);
    virtual DtTime realWorldTime() const;

    // Set/Get simulation time.
    virtual void setSimulationTime(const DtTime time);
    virtual DtTime simulationTime() const;

    // Set/Get the required reliability service.
    virtual void setReqRelService(DtRequiredReliabilityService value);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Set/Get the request id.
    virtual void setRequestId(int id);
    virtual int requestId() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:

    // Create a DtStartResumeRPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
```

```
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtStartResumeRPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtStartResumeRPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

protected:

    static int theStaticSize;

};
#endif
#endif
```

stateRep.h

```
#include "chooseDefines.h"

// class DtStateRepository:
//
// DtStateRepository is an abstract base class for maintaining state
// for an object. These are used to decode data into the repository
// from the network and encode data from the repository to the network.

class DtStateRepository
{
public:

    // destructor
    virtual ~DtStateRepository();

    // assignment operator
    DtStateRepository& operator=(const DtStateRepository& orig);

    // return an empty object of the same type as this
    virtual DtStateRepository* clone() const = 0;

    virtual void printData() const = 0;

protected:

    // Constructors are protected since this is an abstract base class.

    // default constructor
    DtStateRepository();

    // copy constructor
    DtStateRepository(const DtStateRepository& orig);
};
```

stIntCbInfo.h

```

#if DtHLA

#include "stIntCbList.h"
#include "RTI.hh"
#include "stateMsg.h"

// class DtStartStopInterCbInfo:
//
// Instances of DtStartStopInterCbInfo are used for providing
// a callback when the RTI's start or stopInteractionGeneration
// services are invoked.
// This class is a helper class to DtStartStopInterCbList.

class DtStartStopInterCbInfo
{
public:
    // constructor
    DtStartStopInterCbInfo(
        RTI::InteractionClassHandle hand,
        DtStartStopInterCb cb,
        void* usr);

    // destructor
    virtual ~DtStartStopInterCbInfo();

    // copy constructor
    DtStartStopInterCbInfo(
        const DtStartStopInterCbInfo& orig);

    // assignment operator
    DtStartStopInterCbInfo& operator=(
        const DtStartStopInterCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(const DtStartStopInterCbInfo* const other) const;

    // Check whether one callback info is equal to another.
    virtual int operator==(const DtStartStopInterCbInfo& other) const;

    // Get the inter class handle.
    virtual const RTI::InteractionClassHandle& interClass() const;

    // Get the callback function.
    virtual const DtStartStopInterCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:
    RTI::InteractionClassHandle myClass;
    DtStartStopInterCb myFunc;
    void* myUsr;
};
#endif

```

stIntCbList.h

```
#if DtHLA

#include "vList.h"
#include "RTI.hh"

// class DtStartStopInterCbList:
//
// Instances of DtStartStopInterCbList are used to provide
// a list of callbacks (DtStartStopInterCbInfo) which are
// called when the RTI's start or stopInteractionGeneration
// services are invoked.

typedef void (*DtStartStopInterCb)(
    RTI::InteractionClassHandle intClass,
    void* usr);

class DtStartStopInterCbList
{
public:

    // default constructor
    DtStartStopInterCbList();

    // destructor
    virtual ~DtStartStopInterCbList();

    // copy constructor
    DtStartStopInterCbList(
        const DtStartStopInterCbList& orig);

    // assignment operator
    DtStartStopInterCbList& operator=(
        const DtStartStopInterCbList& orig);

    // Add/Remove a start updates callback to/from the list.
    // Returns whether add/remove took place.
    virtual int add(RTI::InteractionClassHandle hand,
        DtStartStopInterCb cb,
        void* usr);
    virtual int remove(RTI::InteractionClassHandle hand,
        DtStartStopInterCb cb,
        void* usr);

    // Remove all callbacks from the list.
    virtual void clear();

    // Callbacks are for a particular inter class handle.
    // Call any callbacks in the list for the inter class handle
    // for the start updates inter.
    virtual void apply(RTI::InteractionClassHandle interClass);

protected:

    // Remove all callbacks from the list.
    // Non-virtual since meant to be called from dtor, otherwise
    // should call clear().
    void dtorClear();

    // Copy another callback list into this inter's callback list.
    // Non-virtual since meant to be called from ctor.
    void ctorCopyList(const DtStartStopInterCbList& orig);

protected:
```

```
    DtList myList;  
};  
#endif
```

stopFrzRPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "disEnums.h"
#include "hostStructs.h"
#include "simanPdu.h"
#include "packets/stopFrzRP.h"
#include "exerciseConn.h"

// class DtStopFreezeRPdu:
//
// Instances of DtStopFreezeRPdu are used to send/receive
// reliable stop/freeze PDUs.

// DtStopFreezeRPdu callback function signature.
typedef void (*DtStopFreezeRPduCb)(DtStopFreezeRPdu* pdu, void* usr);

class DtStopFreezeRPdu : public DtSimanPdu
{
public:

    // Constructors
    DtStopFreezeRPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtStopFreezeRPdu(const DtNetStopFreezeRPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtStopFreezeRPdu(const DtEntityIdentifier& to,
        const DtEntityIdentifier& from,
        const DtTime realTime,
        DtStopReason reason,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtStopFreezeRPdu(const DtStopFreezeRPdu& orig);

    // destructor
    virtual ~DtStopFreezeRPdu();

    // Assignment operator
    DtStopFreezeRPdu& operator=(const DtStopFreezeRPdu& orig);

    // Set/Get real world time.
    virtual void setRealWorldTime(const DtTime time);
    virtual DtTime realWorldTime() const;

    // Set/Get the stop reason.
    virtual void setReason(DtStopReason reason);
    virtual DtStopReason reason() const;

    // Set/Get the frozen behavior.
    virtual void setFrozenBehavior(int behavior);
    virtual int frozenBehavior() const;

    // Set/Get the required reliability service.
    virtual void setReqRelService(DtRequiredReliabilityService value);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Set/Get the request id.
    virtual void setRequestId(int id);
    virtual int requestId() const;

    // Print the data contained by this PDU
    virtual void printData() const;

public:
```



```
// Create a DtStopFreezeRPdu.  
// Caller is responsible for deletion.  
static DtPdu* create(const DtNetPacket* initial,  
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);  
  
// Add (register)/Remove (deregister) function to be called when  
// pdu is received.  
static void addCallback(DtExerciseConn* conn,  
    DtStopFreezeRPduCb cb, void* usr);  
static void removeCallback(DtExerciseConn* conn,  
    DtStopFreezeRPduCb cb, void* usr);  
  
protected:  
  
    // Returns the internal PDU kind  
    virtual DtPduKind internalGetPduKind() const;  
  
protected:  
  
    static int theStaticSize;  
  
};  
#endif  
#endif
```

stopPdu.h

```
#if !DtHLA

#include "disEnums.h"
#include "hostStructs.h"
#include "simanPdu.h"

#define DtStopStaticSize 40

class DtStopFreezePdu;
typedef void (*DtStopFreezePduCb)(DtStopFreezePdu *pdu, void *usr);

class DtStopFreezePdu : public DtSimanPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtStopFreezePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtStopFreezePdu(const DtNetStopFreezePdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** ADDITIONAL CONSTRUCTORS *****/

    DtStopFreezePdu(const DtEntityIdentifier &to,
        const DtEntityIdentifier &from,
        const DtTime realTime,
        DtStopReason reason,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
        DtStopFreezePduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtStopFreezePduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/

    DtTime realWorldTime() const;
    DtStopReason reason() const;
#if DtDISVERS != 3
    int frozenBehavior() const;
    int requestId() const;
#endif

/***** MUTATOR FUNCTIONS *****/

    void setRealWorldTime(const DtTime time);
    void setReason(DtStopReason reason);
#if DtDISVERS != 3
    void setFrozenBehavior(int behavior);
    void setRequestId(int id);
#endif

    void printData() const;

protected:
    DtPduKind internalGetPduKind() const {return DtStopFreezePduKind;}
};
#endif
```

stopUpdCbInfo.h

```

#ifdef DtHLA

#include "stopUpdCbList.h"
#include "RTI.hh"
#include "stateMsg.h"

// class DtStopUpdatesCbInfo:
//
// Instances of DtStopUpdatesCbInfo are used for providing
// a callback when an object stops updating.
// This class is a helper class to DtStopUpdatesCbList.

class DtStopUpdatesCbInfo
{
public:
    // constructor
    DtStopUpdatesCbInfo(RTI::ObjectClassHandle h,
        DtStopUpdatesCb cb,
        void* usr);

    // destructor
    virtual ~DtStopUpdatesCbInfo();

    // copy constructor
    DtStopUpdatesCbInfo(
        const DtStopUpdatesCbInfo& orig);

    // assignment operator
    DtStopUpdatesCbInfo& operator=(
        const DtStopUpdatesCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(const DtStopUpdatesCbInfo* const other) const;

    // Check whether one callback info is equal to another.
    virtual int operator==(const DtStopUpdatesCbInfo& other) const;

    // Get the object class handle.
    virtual const RTI::ObjectClassHandle& objectClass() const;

    // Get the callback function.
    virtual const DtStopUpdatesCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:
    RTI::ObjectClassHandle myClass;
    DtStopUpdatesCb myFunc;
    void* myUsr;
};
#endif

```

stopUpdCbList.h

```
#if DtHLA

#include "vllist.h"
#include "RTI.hh"

// class DtStopUpdatesCbList:
//
// Instances of DtStopUpdatesCbList are used to provide
// a list of callbacks (DtStopUpdatesCbInfo) which are
// called when an object starts updating.

typedef void (*DtStopUpdatesCb)(RTI::ObjectClassHandle objectClass,
    const RTI::AttributeHandleSet& attributeSet,
    void* usr);

class DtStopUpdatesCbList
{
public:

    // default constructor
    DtStopUpdatesCbList();

    // destructor
    virtual ~DtStopUpdatesCbList();

    // copy constructor
    DtStopUpdatesCbList(
        const DtStopUpdatesCbList& orig);

    // assignment operator
    DtStopUpdatesCbList& operator=(
        const DtStopUpdatesCbList& orig);

    // Add/Remove a start updates callback to/from the list.
    // Returns whether add/remove took place.
    virtual int add(RTI::ObjectClassHandle h, DtStopUpdatesCb cb,
        void* usr);
    virtual int remove(RTI::ObjectClassHandle h, DtStopUpdatesCb cb,
        void* usr);

    // Remove all callbacks from the list.
    virtual void clear();

    // Callbacks are for a particular object class handle.
    // Call any callbacks in the list for the object class handle
    // for the start updates object.
    virtual void apply(RTI::ObjectClassHandle objectClass,
        const RTI::AttributeHandleSet& hSet);

protected:

    // Remove all callbacks from the list.
    // Non-virtual since meant to be called from dtor, otherwise
    // should call clear().
    void internalClear();

    // Copy another callback list into this object's callback list.
    // Non-virtual since meant to be called from ctor.
    void internalCopyList(const DtStopUpdatesCbList& orig);

protected:

    DtList myList;
```

```
};  
#endif
```

str.h

```
#include "MachineTypes.h"
#include "iostream.h"

// class DtString:
//
// Instances of DtString are

class DtString
{
    friend ostream& operator<<(ostream& os, const DtString& str);
    friend istream& operator>>(istream& is, DtString& str);

public:

    // default constructor
    DtString();

    DtString(const char* str);

    // destructor
    virtual ~DtString();

    // copy constructor
    DtString(const DtString& orig);

    // assignment operators
    DtString& operator=(const DtString& orig);
    DtString& operator=(const char* orig);

    // cast operator
    virtual operator const char*() const;

    // comparison operators
    virtual DtBoolean operator==(const DtString& str) const;
    virtual DtBoolean operator==(const char* str) const;
    virtual DtBoolean operator<(const DtString& str) const;
    virtual DtBoolean operator<(const char* str) const;
    virtual DtBoolean operator<=(const DtString& str) const;
    virtual DtBoolean operator<=(const char* str) const;
    virtual DtBoolean operator>(const DtString& str) const;
    virtual DtBoolean operator>(const char* str) const;
    virtual DtBoolean operator>=(const DtString& str) const;
    virtual DtBoolean operator>=(const char* str) const;
    virtual DtBoolean operator!=(const DtString& str) const;
    virtual DtBoolean operator!=(const char* str) const;

    // concatenation operators
    virtual DtString& operator+(const DtString& str);
    virtual DtString& operator+=(const DtString& str);

    // index operator
    virtual const char& operator[](int index) const;
    virtual char& operator[](int index);

    // truncates everything before or after the character, including the
    // character, respectively
    virtual void snipFront(const char ch);
    virtual void snipBack(const char ch);

    // returns the number of occurrences of the character
    virtual int count(const char ch) const;

    // return the length of the string
```

```
virtual int length() const;

// returns the char* internal representation of the string
const char* string() const;

protected:

    // deletes the string, and sets the myString pointer to NULL
    // non-virtual because only called by destructor, and non-virtual members
    // myString is set to NULL, so it should be allocated some memory if
    // operators are to be used on the string
    void clear();

    // Set/Get for myString - char pointer that holds the string
    // non-virtual because only called by constructor
    void setString(const char* str);

protected:

    char* myString;

    static const int theMaxInsertionStringSize;
};
```

strHashKey.h

```
#include "baseHashKey.h"

// class DtStringHashKey:
//
// Instances of DtStringHashKey are used as a hash key made from a string.
class DtStringHashKey : public DtBaseHashKey
{
public:
    // default constructor
    DtStringHashKey(const DtString& str);

    // alternate constructor
    DtStringHashKey(const char* str);

    // destructor
    virtual ~DtStringHashKey();

    // copy constructor
    DtStringHashKey(const DtStringHashKey& orig);

    // assignment operator
    DtStringHashKey& operator=(const DtStringHashKey& orig);

    // make a copy of this hash key
    virtual DtBaseHashKey* clone() const;

protected:
    // get the string representation of this key
    virtual const DtString& stringRep() const;

    // compute the hashed value for this key
    virtual DtU32 computeHashValue();

protected:
    DtString myString;
};
```


strHashLst.h

```
#include "hashlist.h"
#include "str.h"

// class DtStringHashlist:
//
// Instances of DtStringHashlist are used to make hashlists keyed off
// of strings.

class DtStringHashlist : public DtIntrusiveHashlist
{
public:
    // default constructor
    DtStringHashlist();

    // destructor
    virtual ~DtStringHashlist();

    // get the hashed key value
    virtual unsigned int hash(DtHashKey k) const;

    // compare two hashkeys
    virtual int isEqual(DtHashKey key1, DtHashKey key2) const;

protected:
    // copy constructor
    DtStringHashlist(const DtStringHashlist& orig);

    // assignment operator
    DtStringHashlist& operator=(const DtStringHashlist& orig);
};
```

strtUpdCbInfo.h

```
#if DtHLA

#if DtRTI103

#include "strtUpdCbList.h"
#include "RTI.hh"
#include "stateMsg.h"

// class DtUpdateControlCbInfo:
//
// Instances of DtUpdateControlCbInfo are used for providing
// a callback when the RTI update control services are invoked.
// This class is a helper class to DtUpdateControlCbList.

class DtUpdateControlCbInfo
{
public:
    // constructor
    DtUpdateControlCbInfo(RTI::ObjectClassHandle h,
        DtUpdateControlCb cb,
        void* usr);

    // destructor
    virtual ~DtUpdateControlCbInfo();

    // copy constructor
    DtUpdateControlCbInfo(
        const DtUpdateControlCbInfo& orig);

    // assignment operator
    DtUpdateControlCbInfo& operator=(
        const DtUpdateControlCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(const DtUpdateControlCbInfo* const other) const;

    // Check whether one callback info is equal to another.
    virtual int operator==(const DtUpdateControlCbInfo& other) const;

    // Get the object class handle.
    virtual const RTI::ObjectClassHandle& objectClass() const;

    // Get the callback function.
    virtual const DtUpdateControlCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:
    RTI::ObjectClassHandle myClass;
    DtUpdateControlCb myFunc;
    void* myUsr;
};
typedef DtUpdateControlCbInfo DtStartUpdatesCbInfo;
typedef DtUpdateControlCbInfo DtStopUpdatesCbInfo;

#endif
#endif
```

strtUpdCbList.h

```

#if DtHLA

#if DtRTI103

#include "vllist.h"
#include "RTI.hh"

// class DtUpdateControlCbList:
//
// Instances of DtUpdateControlCbList are used to provide
// a list of callbacks (DtUpdateControlCbInfo) which are
// called when an object starts updating.

typedef void (*DtUpdateControlCb)(
    RTI::ObjectClassHandle objectClass,
    const RTI::AttributeHandleSet& attributeSet,
    void* usr);

typedef DtUpdateControlCb DtStartUpdatesCb;
typedef DtUpdateControlCb DtStopUpdatesCb;

class DtUpdateControlCbList
{
public:
    // default constructor
    DtUpdateControlCbList();

    // destructor
    virtual ~DtUpdateControlCbList();

    // copy constructor
    DtUpdateControlCbList(
        const DtUpdateControlCbList& orig);

    // assignment operator
    DtUpdateControlCbList& operator=(
        const DtUpdateControlCbList& orig);

    // Add/Remove a start updates callback to/from the list.
    // Returns whether add/remove took place.
    virtual int add(RTI::ObjectClassHandle h, DtUpdateControlCb cb,
        void* usr);
    virtual int remove(RTI::ObjectClassHandle h, DtUpdateControlCb cb,
        void* usr);

    // Remove all callbacks from the list.
    virtual void clear();

    // Callbacks are for a particular object class handle.
    // Call any callbacks in the list for the object class handle.
    virtual void apply(RTI::ObjectClassHandle objectClass,
        const RTI::AttributeHandleSet& hSet);

protected:
    // Remove all callbacks from the list.
    // Non-virtual since meant to be called from dtor, otherwise
    // should call clear().
    void dtorClear();

    // Copy another callback list into this object's callback list.
    // Non-virtual since meant to be called from ctor.

```

```
    void ctorCopyList(const DtUpdateControlCbList& orig);
protected:
    DtList myList;
};
typedef DtUpdateControlCbList DtStartUpdatesCbList;
typedef DtUpdateControlCbList DtStopUpdatesCbList;
#endif
#endif
```

supEmisEsNozz.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/supEmisEsP.h"

// class DtVectoringNozzleSystem:
//
// DtVectoringNozzleSystem is a class which contains the vectoring nozzle
// system data for the supplemental emissions/entity state PDU.

class DtVectoringNozzleSystem
{
public:
    // Constructors
    DtVectoringNozzleSystem();
    DtVectoringNozzleSystem(const DtVectoringNozzleSystem& orig);

    // Assignment operator
    DtVectoringNozzleSystem& operator=(
        const DtVectoringNozzleSystem& orig);
    DtVectoringNozzleSystem& operator=(
        const DtNetVectoringNozzleSystem& orig);

    // Destructor
    virtual ~DtVectoringNozzleSystem();

    // Set/Get the horizontal deflection
    virtual void setHorDeflection(DtFloat32 value);
    virtual DtFloat32 horDeflection() const;

    // Set/Get the vertical deflection
    virtual void setVertDeflection(DtFloat32 value);
    virtual DtFloat32 vertDeflection() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetVectoringNozzleSystem& record);
    virtual void loadNetRecord(
        DtNetVectoringNozzleSystem& record) const;

protected:
    // Data members:
    DtFloat32                myHorizontalDeflection;
    DtFloat32                myVerticalDeflection;
};
#endif
#endif

```

supEmisEsPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"

#include "packets/supEmisEsP.h"

#include "supEmisEsNozz.h"
#include "supEmisEsProp.h"

// class DtSuppEmisEntStatePdu:
//
// Instances of DtSuppEmisEntStatePdu are used to send information about
// simulated supplemental emissions.

// DtSuppEmisEntStatePdu callback function signature.
typedef void (*DtSuppEmisEntStatePduCb)(DtSuppEmisEntStatePdu* pdu, void*
usr);

class DtSuppEmisEntStatePdu : public DtPdu
{
public:
    // Constructors
    DtSuppEmisEntStatePdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtSuppEmisEntStatePdu(const DtNetSuppEmisEntStatePdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtSuppEmisEntStatePdu(const DtSuppEmisEntStatePdu& orig);

    // destructor
    virtual ~DtSuppEmisEntStatePdu();

    // Assignment operator
    DtSuppEmisEntStatePdu& operator=(const DtSuppEmisEntStatePdu& orig);

    // Set/Get originating entity identifier
    virtual void setOriginatingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& originatingEntityID() const;

    // Set/Get infra-red signature representation index
    virtual void setInfraRedSigRepIndex(DtU16 value);
    virtual DtU16 infraRedSigRepIndex() const;

    // Set/Get acoustic signature representation index
    virtual void setAcousticSigRepIndex(DtU16 value);
    virtual DtU16 acousticSigRepIndex() const;

    // Set/Get radar signature representation index
    virtual void setRadarSigRepIndex(DtU16 value);
    virtual DtU16 radarSigRepIndex() const;

    // Set/Get number of propulsion systems
    virtual DtBoolean setNumPropulsionSystems(DtU16 value);
    virtual DtU16 numPropulsionSystems() const;

    // Set/Get number of nozzle systems
    virtual DtBoolean setNumNozzleSystems(DtU16 value);
    virtual DtU16 numNozzleSystems() const;
```

```

// Set/Get a propulsion system data record.
// Returns false if the record does not exist.
virtual DtBoolean setPropulsionSystem(int index,
    const DtPropulsionSystem& record);
virtual DtBoolean getPropulsionSystem(int index,
    DtPropulsionSystem& record) const;

// Set/Get a vectoring nozzle system data record.
// Returns false if the record does not exist.
virtual DtBoolean setNozzleSystem(int index,
    const DtVectoringNozzleSystem& record);
virtual DtBoolean getNozzleSystem(int index,
    DtVectoringNozzleSystem& record) const;

// Print the data contained by this PDU
virtual void printData() const;

public:
    // Create a DtSuppEmisEntStatePdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtSuppEmisEntStatePduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtSuppEmisEntStatePduCb cb, void* usr);

protected:
    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetSuppEmisEntStatePdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non virtual since you can't override based on return type
    DtSuppEmisEntStatePdu* self() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

    // Returns a buffer offset to the propulsion system data record
    virtual int propulsionSystemRecordOffset(int recordIndex) const;

    // Returns a buffer offset to the given nozzle system data record
    virtual int nozzleSystemRecordOffset(int recordIndex) const;

protected:
    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:
    // Temporary objects for reference returns
    DtEntityIdentifier      myEntityID;
};
#endif
#endif

```

supEmisEsProp.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/supEmisEsP.h"

// class DtPropulsionSystem:
//
// DtPropulsionSystem is a class which contains the propulsion system data
// for the supplemental emissions/entity state PDU.

class DtPropulsionSystem
{
public:
    // Constructors
    DtPropulsionSystem();
    DtPropulsionSystem(const DtPropulsionSystem& orig);

    // Assignment operator
    DtPropulsionSystem& operator=(
        const DtPropulsionSystem& orig);
    DtPropulsionSystem& operator=(
        const DtNetPropulsionSystem& orig);

    // Destructor
    virtual ~DtPropulsionSystem();

    // Set/Get the power setting
    virtual void setPowerSetting(DtFloat32 value);
    virtual DtFloat32 powerSetting() const;

    // Set/Get the engine RPM
    virtual void setEngineRPM(DtFloat32 value);
    virtual DtFloat32 engineRPM() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetPropulsionSystem& record);
    virtual void loadNetRecord(
        DtNetPropulsionSystem& record) const;

protected:
    // Data members:
    DtFloat32                myPowerSetting;
    DtFloat32                myEngineRPM;
};
#endif
#endif
```


taitBryan.h

```
#include "MachineTypes.h"

class DtTaitBryan
{
public:
    DtTaitBryan();
    DtTaitBryan(double a, double b, double c);

    DtTaitBryan(const DtTaitBryan &tait);
    DtTaitBryan &operator=(const DtTaitBryan &tait);

    // Get whether another tait bryan equals or does not equal this one.
    int operator==(const DtTaitBryan& other) const;
    int operator!=(const DtTaitBryan& other) const;

    const char *string() const;

    DtFloat64 psi() const;
    DtFloat64 theta() const;
    DtFloat64 phi() const;

    void setPsi(DtFloat64 psi);
    void setTheta(DtFloat64 theta);
    void setPhi(DtFloat64 phi);

protected:
    DtFloat64 myPsi; /* about reference z */
    DtFloat64 myTheta; /* about intermediate y */
    DtFloat64 myPhi; /* about body x */
};
// Inline functions
```

thresholder.h

```
#include "MachineTypes.h"
#include "mathTypes.h"
#include "artPartThresh.h"

// class DtThresholder:
//
// Instances of DtThresholder store thresholding information used to
// determine what attributes of an entity need to be updated.

class DtThresholder
{
public:
    // default constructor
    DtThresholder();

    // destructor
    virtual ~DtThresholder();

    // copy constructor
    DtThresholder(const DtThresholder& orig);

    // assignment operator
    DtThresholder& operator=(const DtThresholder& orig);

    // Set/Get rotation threshold in radians.
    // Negative number means use default.
    virtual void setRotationThreshold(DtReal thresh);
    virtual double rotationThreshold() const;

    // Returns the "RotationTrace" threshold
    virtual double rotationTraceThreshold() const;

    // Set/Get translation threshold in meters.
    // Negative number means use default.
    virtual void setTranslationThreshold(DtReal thresh);
    virtual double translationThreshold() const;

    // Set/Get aggregate dimension threshold. Thresh is a number between
    // 0 and 1, indicating the percentage change that an aggregate dimension
    // can undergo before causing an update to be generated.
    // Negative number means use default.
    // Applies only to aggregate entities.
    virtual void setAggDimensionThreshold(DtReal thresh);
    virtual double aggDimensionThreshold() const;

    // Returns a pointer to an object that maintains thresholds
    // for articulated parts.
    virtual DtArtPartThresholder* artPartThresholder();
    virtual const DtArtPartThresholder& artPartThresholder() const;

public:
    // Set/Get default translation threshold in radians.
    static void setDfltTranslationThreshold(DtReal thresh);
    static double dfltTranslationThreshold();

    // Set/Get default rotation threshold in radians.
    static void setDfltRotationThreshold(DtReal thresh);
    static double dfltRotationThreshold();

    // Returns the "RotationTrace" threshold
    static double dfltRotationTraceThreshold();
```

```
// Set/Get aggregate dimension threshold. Thresh is a number between
// 0 and 1, indicating the percentage change that an aggregate dimension
// can undergo before causing an update to be generated.
// Applies only to aggregate entities.
static void setDfltAggDimensionThreshold(DtReal thresh);
static double dfltAggDimensionThreshold();

protected:

    double myTransThresh;
    double myRotThresh;
    double myTraceRotThresh;
    double myAggDimThresh;
    DtBoolean myUsingDfltTrans;
    DtBoolean myUsingDfltRot;
    DtBoolean myUsingDfltAggDim;

    DtArtPartThresholder myArtPartThresh;

protected:

    static double theDfltTransThresh;
    static double theDfltRotThresh;
    static double theDfltTraceRotThresh;
    static double theDfltAggDimThresh;
};
```

topoCoord.h

```
#include "geodCoord.h"
#include "LibMatrix.h"

// Any point on the globe can be used to specify the origin of a
// specific topographic coordinate system. In VR-Link, a topographic
// frame is defined with the positive X axis pointing north, the
// positive Y axis pointing east, and the positive Z axis pointing down.

// From latitude and longitude, we can produce the rotation matrices
// that rotate between geocentric and topographic-down orientations.
// VR-Link uses post-multiplication for its rotation matrices. For
// example, the matrix produced by DtLatLon_to_TopoToGeoc can be used
// to rotate a topographic vector to a geocentric vector like this:
//      DtDcmVecMul(topoToGeoc, Vtopo, Vgeoc);
void DtLatLon_to_TopoToGeoc(const DtGeodeticCoord& g, DtDcmRef topoToGeoc);

// The matrix produced by DtLatLon_to_GeocToTopo can be used
// to rotate a geocentric vector to a topographic vector like this:
//      DtDcmVecMul(geocToTopo, Vgeoc, Vtopo);
void DtLatLon_to_GeocToTopo(const DtGeodeticCoord& g, DtDcmRef geocToTopo);

// This function can initialize a DtCoordTransform so that it can provide
// transformations between the geocentric coordinate system, and a
// particular topographic coordinate system defined by latitude and
// longitude.
void DtGeocToTopoTransform(double latitude, double longitude,
    DtCoordTransform* geoc2topo);
```

topoView.h

```
#include "cartView.h"

// class DtTopoView:
//
// Instances of DtTopoView are used to create a Topographic
// coordinate view of the geocentric coordinates in an
// Entity State Repository.

class DtTopoView : public DtCartesianView
{
public:
    // constructor
    DtTopoView(DtBaseEntityStateRepository* esr,
               double latitude, double longitude);

    // destructor
    virtual ~DtTopoView();

    // copy constructor
    DtTopoView(const DtTopoView& orig);

    // assignment operator
    DtTopoView& operator=(const DtTopoView& orig);
};
```

transCtlPdu.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "recordSetPdu.h"
#include "exerciseConn.h"

#include "packets/transCtlP.h"

// class DtTransferControlRequestPdu:
//
// Instances of DtTransferControlRequestPdu are used to initiate the dynamic
// allocation and control of simulated entities.

// DtTransferControlRequestPdu callback function signature.
typedef void (*DtTransferControlRequestPduCb)
    (DtTransferControlRequestPdu* pdu, void* usr);

class DtTransferControlRequestPdu : public DtRecordSetPdu
{
public:
    // Constructors
    DtTransferControlRequestPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtTransferControlRequestPdu(const DtNetTransferControlRequestPdu*
initial,
    DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtTransferControlRequestPdu(const DtTransferControlRequestPdu& orig);

    // destructor
    virtual ~DtTransferControlRequestPdu();

    // Assignment operator
    DtTransferControlRequestPdu& operator=(
        const DtTransferControlRequestPdu& orig);

    // Set/Get originating entity identifier
    virtual void setOriginatingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& originatingEntityID() const;

    // Set/Get receiving entity identifier
    virtual void setReceivingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& receivingEntityID() const;

    // Set/Get request ID
    virtual void setRequestID(DtU32 id);
    virtual DtU32 requestID() const;

    // Set/Get required reliability service
    virtual void setReqRelService(enum DtRequiredReliabilityService
service);
    virtual DtRequiredReliabilityService reqRelService() const;

    // Set/Get transfer type
    virtual void setTransferType(DtTransferType type);
    virtual DtTransferType transferType() const;

    // Set/Get transferred entity identifier
    virtual void setTransferredEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& transferredEntityID() const;
```

```

    // Print the data contained by this PDU
    virtual void printData() const;

public:
    // Create a DtTransferControlRequestPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    // Add (register)/Remove (deregister) function to be called when
    // pdu is received.
    static void addCallback(DtExerciseConn* conn,
        DtTransferControlRequestPduCb cb, void* usr);
    static void removeCallback(DtExerciseConn* conn,
        DtTransferControlRequestPduCb cb, void* usr);

protected:
    // Returns a pointer to the beginning of the record set data struct.
    virtual DtNetRecordSetPdu* netRecordSetPduStruct() const;

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    // Non-virtual since you can't override based on return type
    virtual DtNetTransferControlRequestPdu* netPduStruct() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non virtual since you can't override based on return type
    DtTransferControlRequestPdu* self() const;

protected:
    // Size of PDU with zero data bytes.
    static int theStaticSize;

protected:
    // Temporary objects for reference returns
    DtEntityIdentifier myEntityID;

};
#endif

```

transmitPdu.h

```
#if !DtHLA

#include "hostStructs.h"
#include "pdu.h"
#include "mathTypes.h"
#include "packets/transmitterP.h"
#include "beamAntRec.h"

#define DtTransmitterStaticSize 104

class DtTransmitterPdu;
typedef void (*DtTransmitterPduCb)(DtTransmitterPdu *pdu, void *usr);

#define DtFREQ_UPPER 0
#define DtFREQ_LOWER 1

class DtModulationType
{
public:
    DtModulationType(int spectrum = 0,
        DtRadioMajorModulation major =
        DtRadioMajorModOther,
        int detail = 0,
        DtRadioSystem sys = DtRadioSystemOther)
: Spectrum(spectrum), Major(major), Detail(detail), Sys(sys) {}

    unsigned int spectrum() const {return Spectrum;}
    DtRadioMajorModulation majorModType() const {return Major;}
    int detail() const {return Detail;}
    DtRadioSystem systemType() const {return Sys;}

    void setSpectrum(unsigned int s) {Spectrum = s;}
    void setMajor(DtRadioMajorModulation m) {Major = m;}
    void setDetail(int d) {Detail = d;}
    void setSystemType(DtRadioSystem s) {Sys = s;}

    void printData() const;

private:
    unsigned int Spectrum;
    DtRadioMajorModulation Major;
    int Detail;
    DtRadioSystem Sys;
};

class DtTransmitterPdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtTransmitterPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtTransmitterPdu(const DtNetTransmitterPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtExerciseConn *conn,
DtTransmitterPduCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
DtTransmitterPduCb cb, void *usr);

/***** INSPECTOR FUNCTIONS *****/
```



```

    const DtEntityIdentifier &entityId() const;
    unsigned int radioId() const;
#if DtDISVERS >= 5
    const DtRadioEntityType &radioEntityType() const;
#else
    const DtEntityType &radioEntityType() const;
#endif
    DtRadioTransmitState transmitState() const;
    DtRadioInputSource inputSource() const;
    DtConstVector worldAntennaLoc() const;
    DtConstVector relativeAntennaLoc() const;
    DtAntennaPatternType antennaPatternType() const;
    int antennaParamsLength() const;
    unsigned long frequency(int upperLower = DtFREQ_LOWER) const;
    float bandwidth() const;
    float power() const;
    const DtModulationType &modulationType() const;
    DtRadioCryptoSystem cryptoSystem() const;
    int cryptoKeyId() const;
    int modulationParamsLength() const;
    void getNetModulationParams(void *destination) const;
    void getNetAntennaParams(void *destination) const;
    const DtBeamAntennaPatternRecord &antennaParams() const;

/***** MUTATOR FUNCTIONS *****/

    void setEntityId(const DtEntityIdentifier &id);
    void setRadioId(unsigned int id);
#if DtDISVERS >= 5
    void setRadioEntityType(const DtRadioEntityType &type);
#else
    void setRadioEntityType(const DtEntityType &type);
#endif
    void setTransmitState(DtRadioTransmitState state);
    void setInputSource(DtRadioInputSource source);
    void setWorldAntennaLoc(DtConstVector loc);
    void setRelativeAntennaLoc(DtConstVector loc);
    void setAntennaPatternType(DtAntennaPatternType type);
    void setFrequency(unsigned long upper, unsigned long lower);
    void setFrequency(unsigned long freq);
    void setBandwidth(float bw);
    void setPower(float power);
    void setModulationType(const DtModulationType &type);
    void setCryptoSystem(DtRadioCryptoSystem sys);
    void setCryptoKeyId(int key);
    void setModulationParams(void *networkRepresentation, int size);
    void setAntennaPatternParams(void *networkRepresentation, int size);
    void setAntennaPatternParams(const DtBeamAntennaPatternRecord &rec);

    void printData() const;
protected:
    DtPduKind internalGetPduKind() const {return DtTransmitterPduKind;}
    DtEntityIdentifier theId;
#if DtDISVERS >= 5
    DtRadioEntityType theType;
#else
    DtEntityType theType;
#endif
    DtVector wLoc;
    DtVector rLoc;
    DtModulationType modType;
    DtBeamAntennaPatternRecord pat;
    void setModulationParamsLength(int length);
    void setAntennaParamsLength(int length);
    void *antennaParamsAddr() const;
    void *modulationParamsAddr() const;
    virtual void printModulationParams() const;
    virtual void printAntennaParams() const;

```

```
};  
#endif // !DtHLA
```

trkJamBeamDec.h

```

#if DtHLA

#ifndef DtTrackJamBeamDecoder_H_
#define DtTrackJamBeamDecoder_H_

#include "chooseDefines.h"
#include "baseEmitttrDec.h"

// This is used to declare attribute decoding functions
// which are members of the class.
#define DtDECLARE_TRKJAMBEAM_ATTR_DECODER(attrName) \
    DtDECLARE_ATTR_DECODER(DtEmitterBeamRepository, attrName)

// class DtTrackJamBeamDecoder:
//
// Instances of DtTrackJamBeamDecoder are

class DtTrackJamBeamDecoder : public DtBaseEmitterBeamDecoder
{
public:
    // default constructor
    DtTrackJamBeamDecoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtTrackJamBeamDecoder();

    // copy constructor
    DtTrackJamBeamDecoder(const DtTrackJamBeamDecoder& orig);

    // assignment operator
    DtTrackJamBeamDecoder& operator=(const DtTrackJamBeamDecoder& orig);

public:
    // Create a DtBaseEntityDecoder. Caller is responsible for deletion.
    static DtHlaStateDecoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    // Add decoding functions for the attributes.
    // Note: Non-virtual since meant to be called from constructors to
    //       add decoders.
    void addMyDecoders();

    // static decode<AttributeName> member functions.
    DtDECLARE_TRKJAMBEAM_ATTR_DECODER(JammingModeSequence);
    DtDECLARE_TRKJAMBEAM_ATTR_DECODER(TrackJamObjectIDs);
    DtDECLARE_TRKJAMBEAM_ATTR_DECODER(TrackOrJammer);
};
// These macros are used to define simple attribute decoder functions
// (members of a derived DtHlaStateDecoder that decode
// individual attributes). The simple decoding is just to
// convert the value pulled out of the ahvps to a host type using
// the cast expression castExpr, and pass it to the mutator function.
#define DtDEFINE_SIMPLE_BASE_TRKJAMBEAM_ATTR_DECODER_WITH_CAST( \
    attrName, netType, mutator, castExpr) \
    DtDEFINE_SIMPLE_ATTR_DECODER_WITH_CAST( \
        DtTrackJamBeamDecoder, DtEmitterBeamRepository, \

```

```
        attrName, netType, mutator, castExpr)
#define DtDEFINE_SIMPLE_BASE_TRKJAMBEAM_ATTR_DECODER( \
    attrName, netType, inspector) \
    DtDEFINE_SIMPLE_BASE_TRKJAMBEAM_ATTR_DECODER_WITH_CAST( \
        attrName, netType, inspector, (netType))
#endif    //DtTrackJamBeamDecoder_H_
```

trkJamBeamEnc.h

```

#if DtHLA

#ifndef DtTrackJamBeamEncoder_H_
#define DtTrackJamBeamEncoder_H_

#include "chooseDefines.h"
#include "baseEmitttrEnc.h"

// This is used to declare static member functions for determining
// whether or not an attribute needs to be encoded.
#define DtDECLARE_BASE_TRKJAMBEAM_ATTR_CHECKER(attrName) \
    DtDECLARE_ATTR_CHECKER(DtEmitterBeamRepository, attrName)

// This is used to declare attribute encoding functions
// which are members of the class.
#define DtDECLARE_BASE_TRKJAMBEAM_ATTR_ENCODER(attrName) \
    DtDECLARE_ATTR_ENCODER(DtEmitterBeamRepository, attrName)

// class DtTrackJamBeamEncoder:
//
// Instances of DtTrackJamBeamEncoder are

class DtTrackJamBeamEncoder : public DtBaseEmitterBeamEncoder
{
public:
    // default constructor
    DtTrackJamBeamEncoder(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

    // destructor
    virtual ~DtTrackJamBeamEncoder();

    // copy constructor
    DtTrackJamBeamEncoder(const DtTrackJamBeamEncoder& orig);

    // assignment operator
    DtTrackJamBeamEncoder& operator=(const DtTrackJamBeamEncoder& orig);

public:
    // Create a DtBaseEntityEncoder. Caller is responsible for deletion.
    static DtHlaStateEncoder* create(
        DtExerciseConn* exConn,
        DtObjClassDesc* classDesc);

protected:
    DtDECLARE_BASE_TRKJAMBEAM_ATTR_ENCODER(JammingModeSequence);
    DtDECLARE_BASE_TRKJAMBEAM_ATTR_ENCODER(TrackJamObjectIDs);
    DtDECLARE_BASE_TRKJAMBEAM_ATTR_ENCODER(TrackOrJammer);

    DtDECLARE_BASE_TRKJAMBEAM_ATTR_CHECKER(JammingModeSequence);
    DtDECLARE_BASE_TRKJAMBEAM_ATTR_CHECKER(TrackJamObjectIDs);
    DtDECLARE_BASE_TRKJAMBEAM_ATTR_CHECKER(TrackOrJammer);
};

// This macro is used to define simple attribute encoder functions
// (members of a derived DtHlaStateEncoder that encode
// individual attributes). The simple encoding is just to implicitly
// convert the value returned by the inspector to a netType,
// and add it to the ahvps.
#define DtDEFINE_SIMPLE_BASE_TRKJAMBEAM_ATTR_ENCODER( \
    attrName, netType, inspector) \

```

```
DtDEFINE_SIMPLE_ATTR_ENCODER(DtTrackJamBeamEncoder, \
    DtEmitterBeamRepository, \
    attrName, netType, inspector)

// This macro is used to define simple attribute checker functions
// (members of a derived DtHlaStateEncoder that check whether
// a particular attribute is out of date. The simple implementation
// is to just compare the values returned by the two StateReps'
// inspector functions.
#define DtDEFINE_SIMPLE_BASE_TRKJAMBEAM_ATTR_CHECKER( \
    attrName, inspector) \
    DtDEFINE_SIMPLE_ATTR_CHECKER(DtTrackJamBeamEncoder, \
        DtEmitterBeamRepository, \
        attrName, inspector)

#endif    //DtTrackJamBeamEncoder_H_
```

tStampType.h

```
enum DtTimeStampType
{
    DtTimeStampRelative,
    DtTimeStampAbsolute
};
```

undwaAcBeam.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/undwaAcP.h"

// class DtUnderwaterAcousticBeam:
//
// DtUnderwaterAcousticBeam is a class which contains the beam fundamental
// data parameters for the underwater acoustics PDU

class DtUnderwaterAcousticBeam
{
public:
    // Constructors
    DtUnderwaterAcousticBeam();
    DtUnderwaterAcousticBeam(
        const DtUnderwaterAcousticBeam& orig);

    // Assignment operator
    DtUnderwaterAcousticBeam& operator=(
        const DtUnderwaterAcousticBeam& orig);
    DtUnderwaterAcousticBeam& operator=(
        const DtNetUnderwaterAcousticBeam& orig);

    // Destructor
    virtual ~DtUnderwaterAcousticBeam();

    // Set/Get the beam data length
    virtual void setBeamDataLength(DtU8 length);
    virtual DtU8 beamDataLength() const;

    // Set/Get the beam ID number
    virtual void setBeamID_Number(DtU8 num);
    virtual DtU8 beamID_Number() const;

    // Set/Get the active emission parameter index
    virtual void setActiveEmissionParmIndex(
        DtUnderwaterAcousticActiveEmissionParmIndex ind);
    virtual DtUnderwaterAcousticActiveEmissionParmIndex
        activeEmissionParmIndex() const;

    // Set/Get the acoustic scan pattern
    virtual void setScanPattern(DtUnderwaterAcousticScanPattern value);
    virtual DtUnderwaterAcousticScanPattern scanPattern() const;

    // Set/Get the beam center azimuth
    virtual void setBeamCenterAzimuth(DtFloat32 value);
    virtual DtFloat32 beamCenterAzimuth() const;

    // Set/Get the azimuthal beamwidth
    virtual void setAzimuthalBeamwidth(DtFloat32 value);
    virtual DtFloat32 azimuthalBeamwidth() const;

    // Set/Get the beam center D/E
    virtual void setBeamCenterD_E(DtFloat32 value);
    virtual DtFloat32 beamCenterD_E() const;

    // Set/Get the D/E beamwidth
```



```
virtual void setD_E_Beamwidth(DtFloat32 value);
virtual DtFloat32 D_E_Beamwidth() const;

// Print the information contained in the record
virtual void print() const;

// Load/unload data from a packet record
virtual void unloadNetRecord(
    const DtNetUnderwaterAcousticBeam& record);
virtual void loadNetRecord(
    DtNetUnderwaterAcousticBeam& record) const;

protected:

    // Data members:
    DtU8                myBeamDataLength;
    DtU8                myBeamID_Number;

    DtUnderwaterAcousticActiveEmissionParmIndex
        myActiveEmissionParmIndex;

    DtUnderwaterAcousticScanPattern myScanPattern;

    DtFloat32           myBeamCenterAzimuth;
    DtFloat32           myAzimuthalBeamwidth;
    DtFloat32           myBeamCenterD_E;
    DtFloat32           myD_E_Beamwidth;
};
#endif
#endif
```

undwaAcEmitSys.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"
#include "mathTypes.h"

#include "packets/undwaAcP.h"

// class DtUnderwaterAcousticEmitterSystem:
//
// DtUnderwaterAcousticEmitterSystem is a class which contains information
// describing the acoustic emitter systems in an underwater acoustics PDU.

class DtUnderwaterAcousticEmitterSystem
{
public:

    // Constructors
    DtUnderwaterAcousticEmitterSystem();
    DtUnderwaterAcousticEmitterSystem(
        const DtUnderwaterAcousticEmitterSystem& orig);

    // Assignment operator
    DtUnderwaterAcousticEmitterSystem& operator=(
        const DtUnderwaterAcousticEmitterSystem& orig);
    DtUnderwaterAcousticEmitterSystem& operator=(
        const DtNetUnderwaterAcousticEmitterSystem& orig);

    // Destructor
    virtual ~DtUnderwaterAcousticEmitterSystem();

    // Set/Get the data length
    virtual void setDataLength(DtU8 value);
    virtual DtU8 dataLength() const;

    // Set/Get the number of beams
    virtual void setNumberOfBeams(DtU8 value);
    virtual DtU8 numberOfBeams() const;

    // Set/Get the system name
    virtual void setAcousticName(DtUnderwaterAcousticSystemName name);
    virtual DtUnderwaterAcousticSystemName acousticName() const;

    // Set/Get the function
    virtual void setFunction(DtUnderwaterAcousticFunction function);
    virtual DtUnderwaterAcousticFunction function() const;

    // Set/Get the acoustic ID number
    virtual void setAcousticID(DtU8 id);
    virtual DtU8 acousticID() const;

    // Set/Get the location
    virtual void setLocation(DtConstVector coords);
    virtual DtConstVector location() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetUnderwaterAcousticEmitterSystem& record);
```

```
virtual void loadNetRecord(  
    DtNetUnderwaterAcousticEmitterSystem& record) const;  
  
protected:  
  
    // Data members:  
    DtU8 myDataLength;  
    DtU8 myNumberOfBeams;  
    DtUnderwaterAcousticSystemName myAcousticName;  
    DtUnderwaterAcousticFunction myFunction;  
    DtU8 myAcousticID;  
    DtVector myLocation;  
  
};  
#endif  
#endif
```

undwaAcPasAct.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/undwaAcP.h"

// class DtUnderwaterAcousticAddtnlPassiveAct:
//
// DtUnderwaterAcousticAddtnlPassiveAct is a class which contains the
// additional passive activity information for the underwater acoustics PDU

class DtUnderwaterAcousticAddtnlPassiveAct
{
public:
    // Constructors
    DtUnderwaterAcousticAddtnlPassiveAct();
    DtUnderwaterAcousticAddtnlPassiveAct(
        const DtUnderwaterAcousticAddtnlPassiveAct& orig);

    // Assignment operator
    DtUnderwaterAcousticAddtnlPassiveAct& operator=(
        const DtUnderwaterAcousticAddtnlPassiveAct& orig);
    DtUnderwaterAcousticAddtnlPassiveAct& operator=(
        const DtNetUnderwaterAcousticAddtnlPassiveAct& orig);

    // Destructor
    virtual ~DtUnderwaterAcousticAddtnlPassiveAct();

    // Set/Get the parameter index
    virtual void setParmIndex(
        DtUnderwaterAcousticAdditionalPassiveActivityParmIndex value);
    virtual DtUnderwaterAcousticAdditionalPassiveActivityParmIndex
        parmIndex() const;

    // Set/Get the APA value
    virtual void setAPA(DtInt16 value);
    virtual DtInt16 APA() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
        const DtNetUnderwaterAcousticAddtnlPassiveAct& record);
    virtual void loadNetRecord(
        DtNetUnderwaterAcousticAddtnlPassiveAct& record) const;

protected:
    // Data members:
    DtUnderwaterAcousticPassiveParmIndex myParmIndex;
    DtU16 myAPA;
};
#endif
#endif
```

undwaAcPdu.h

```

#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "hostStructs.h"
#include "pdu.h"
#include "exerciseConn.h"

#include "mathTypes.h"

#include "packets/undwaAcP.h"

#include "undwaAcBeam.h"
#include "undwaAcEmitSys.h"
#include "undwaAcPasAct.h"
#include "undwaAcShaft.h"

#include "chooseDefines.h"

// class DtUnderwaterAcousticPdu:
//
// Instances of DtUnderwaterAcousticPdu are used to send information about
// simulated underwater acoustic emissions.

// DtUnderwaterAcousticPdu callback function signature.
typedef void (*DtUnderwaterAcousticPduCb)
    (DtUnderwaterAcousticPdu* pdu, void* usr);

class DtUnderwaterAcousticPdu : public DtPdu
{
public:
    // Constructors
    DtUnderwaterAcousticPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtUnderwaterAcousticPdu(const DtNetUnderwaterAcousticPdu* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtUnderwaterAcousticPdu(const DtUnderwaterAcousticPdu& orig);

    // destructor
    virtual ~DtUnderwaterAcousticPdu();

    // Assignment operator
    DtUnderwaterAcousticPdu& operator=(const DtUnderwaterAcousticPdu& orig);

    // Set/Get emitting entity identifier
    virtual void setEmittingEntityID(const DtEntityIdentifier& entity);
    virtual const DtEntityIdentifier& emittingEntityID() const;

    // Set/Get event identifier
    virtual void setEventID(const DtEventID& event);
    virtual const DtEventID& eventID() const;

    // Set/Get state/change update indicator
    virtual void setStateChangeUpdate(DtUnderwaterAcousticUpdate upd);
    virtual DtUnderwaterAcousticUpdate stateChangeUpdate() const;

    // Set/Get passive parameter index
    virtual void setPassiveParameterIndex(
        DtUnderwaterAcousticPassiveParmIndex ind);
    virtual DtUnderwaterAcousticPassiveParmIndex
        passiveParameterIndex() const;

```

```
// Set/Get propulsion plant configuration
virtual void setPropulsionPlantConfig(
    DtUnderwaterAcousticPropulsionPlantConfig config);
virtual DtUnderwaterAcousticPropulsionPlantConfig
    propulsionPlantConfig() const;

// Set/Get number of shafts
// May return DtFalse if PDU size would exceed the maximum PDU size
virtual DtBoolean setNumShafts(DtU8 num);
virtual DtU8 numShafts() const;

// Set/Get number of additional passive activities
// May return DtFalse if PDU size would exceed the maximum PDU size
virtual DtBoolean setNumAdditionalPassiveAct(DtU8 num);
virtual DtU8 numAdditionalPassiveAct() const;

// Set/Get number of emitter systems
// May return DtFalse if PDU size would exceed the maximum PDU size
virtual DtBoolean setNumEmitterSystems(DtU8 num);
virtual DtU8 numEmitterSystems() const;

// Set/Get number of beam data records for a particular emitter system.
// May return DtFalse if PDU size would exceed the maximum PDU size
// or if the emitter system number is invalid.
virtual DtBoolean setNumBeams(int emitterSystemIndex, DtU8 num);
virtual DtU8 numBeams(int emitterSystemIndex) const;

// Set/Get a shaft speed data record.
// Returns false if the record does not exist.
virtual DtBoolean setShaftSpeed(int index,
    const DtUnderwaterAcousticShaftSpeed& record);
virtual DtBoolean getShaftSpeed(int index,
    DtUnderwaterAcousticShaftSpeed& record) const;

// Set/Get an additional passive activity data record.
// Returns false if the record does not exist.
virtual DtBoolean setAddtnlPassiveAct(int index,
    const DtUnderwaterAcousticAddtnlPassiveAct& record);
virtual DtBoolean getAddtnlPassiveAct(int index,
    DtUnderwaterAcousticAddtnlPassiveAct& record) const;

// Set/Get an underwater acoustic emitter system.
// Returns false if the record does not exist.
// Data length field is automatically set by the PDU class.
virtual DtBoolean setEmitterSystem(int index,
    const DtUnderwaterAcousticEmitterSystem& record);
virtual DtBoolean getEmitterSystem(int index,
    DtUnderwaterAcousticEmitterSystem& record) const;

// Set/Get beam fundamental data parameters for a beam.
// Returns false if the record does not exist.
// Data length field is automatically set by the PDU class.
virtual DtBoolean setBeam(int emitterSystemIndex, int beamIndex,
    const DtUnderwaterAcousticBeam& record);
virtual DtBoolean getBeam(int emitterSystemIndex, int beamIndex,
    DtUnderwaterAcousticBeam& record) const;

// Print the data contained by this PDU
virtual void printData() const;

public:

    // Create a DtUnderwaterAcousticPdu.
    // Caller is responsible for deletion.
    static DtPdu* create(const DtNetPacket* initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
```

```

// Add (register)/Remove (deregister) function to be called when
// pdu is received.
static void addCallback(DtExerciseConn* conn,
    DtUnderwaterAcousticPduCb cb, void* usr);
static void removeCallback(DtExerciseConn* conn,
    DtUnderwaterAcousticPduCb cb, void* usr);

protected:

    // Returns the internal PDU kind
    virtual DtPduKind internalGetPduKind() const;

    // Casts the netPdu element to our network struct
    virtual DtNetUnderwaterAcousticPdu* netPduStruct() const;

    // Returns a pointer to the first byte of variable length data
    virtual char* var() const;

    // Casts the 'this' pointer for changing elements in a const function
    // Non virtual since you can't override based on return type
    DtUnderwaterAcousticPdu* self() const;

    // Returns a buffer offset to the given shaft speed record
    virtual int shaftSpeedRecordOffset(int recordIndex) const;

    // Returns a buffer offset to the given additional passive activity
record
    virtual int passiveActRecordOffset(int recordIndex) const;

    // Returns a buffer offset to the given acoustic emitter system
    virtual int emitterSystemRecordOffset(int recordIndex) const;

    // Returns a buffer offset to the given beam fundamental data parameter
of
    // a particular emitter system.
    virtual int beamDataRecordOffset(
        int emitterSystemIndex, int beamIndex) const;

protected:

    // Size of PDU with zero parameter records.
    static int theStaticSize;

protected:

    // Temporary objects for reference returns
    DtEntityIdentifier          myEntityID;
    DtEventID                  myEventID;

};
#endif
#endif

```

undwaAcShaft.h

```
#if !DtHLA
#if DtDISVERS >= 5

#include "MachineTypes.h"
#include "disEnums.h"

#include "packets/undwaAcP.h"

// class DtUnderwaterAcousticShaftSpeed:
//
// DtUnderwaterAcousticShaftSpeed is a class which contains the shaft speed
// information for the underwater acoustics PDU.

class DtUnderwaterAcousticShaftSpeed
{
public:
    // Constructors
    DtUnderwaterAcousticShaftSpeed();
    DtUnderwaterAcousticShaftSpeed(const DtUnderwaterAcousticShaftSpeed&
orig);

    // Assignment operator
    DtUnderwaterAcousticShaftSpeed& operator=(
const DtUnderwaterAcousticShaftSpeed& orig);
    DtUnderwaterAcousticShaftSpeed& operator=(
const DtNetUnderwaterAcousticShaftSpeed& orig);

    // Destructor
    virtual ~DtUnderwaterAcousticShaftSpeed();

    // Set/Get the current shaft RPM
    virtual void setCurrentShaftRPM(DtU16 value);
    virtual DtU16 currentShaftRPM() const;

    // Set/Get the ordered shaft RPM
    virtual void setOrderedShaftRPM(DtU16 value);
    virtual DtU16 orderedShaftRPM() const;

    // Set/Get the shaft RPM rate of change
    virtual void setShaftRPM_RateOfChange(DtU32 value);
    virtual DtU32 shaftRPM_RateOfChange() const;

    // Print the information contained in the record
    virtual void print() const;

    // Load/Unload data from a packet record
    virtual void unloadNetRecord(
const DtNetUnderwaterAcousticShaftSpeed& record);
    virtual void loadNetRecord(
DtNetUnderwaterAcousticShaftSpeed& record) const;

protected:

    // Data members:
    DtU16 myCurrentShaftRPM;
    DtU16 myOrderedShaftRPM;
    DtU32 myShaftRPM_RateOfChange;
};
#endif
#endif
```


unknownHObj.h

```

#if DtHLA

#include "hlaObj.h"

// class DtUnknownHlaObject:
//
// Instances of DtUnknownHlaObject are use to represent HLA Objects for
// which
// no more specific derived DtHlaObject exists.

class DtUnknownHlaObject : public DtHlaObject
{
public:

#if DtRTI103
    // constructor
    DtUnknownHlaObject(
        DtExerciseConn* exConn,
        RTI::ObjectHandle id,
        DtObjClassDesc* classDesc);
#else
    // constructor
    DtUnknownHlaObject(
        DtExerciseConn* exConn,
        RTI::ObjectHandle id,
        const char* name,
        DtObjClassDesc* classDesc);

    // constructor
    DtUnknownHlaObject(
        DtExerciseConn* exConn,
        const char* name,
        DtObjClassDesc* classDesc);
#endif

    // destructor
    virtual ~DtUnknownHlaObject();

    // copy constructor
    DtUnknownHlaObject(const DtUnknownHlaObject& orig);

    // assignment operator
    DtUnknownHlaObject& operator=(const DtUnknownHlaObject& orig);

    // Does nothing. Use the version of update that takes an ahvps
    // to send data for a DtUnknownHlaObject.
    virtual void update();

    // Sends the ahvps through our DtExerciseConn. Does not look
    // at the contents of ahvps. It is the caller's responsibility to
    // insure that all attributes in the message are in the set of
    // attributes needed by the federation (obtained using
    // attributesNeededByFederation). It is also the responsibility of the
    // user of DtUnknownHlaObject to insure that requested attributes
    // (obtained via requestedAttributes) get updated in a timely manner.
    // clearRequestedAttributes should be called after each time requested
    // attributes are updated.
    virtual void update(const RTI::AttributeHandleValuePairSet& ahvps);

protected:

    // Does nothing. You can access information from incoming
    // update messages by using DtHlaObject::addPostReflectCb to

```

```
// register a function that will be called whenever an update
// msg is received.
virtual void reflect(const DtStateMsg& stateMsg);
};
#endif
```

unknownPdu.h

```

#if !DtHLA

#include "pdu.h"
#define DtUnknownStaticSize 12

class DtUnknownPdu;
typedef void (*DtUnknownPduCb)(DtUnknownPdu *pdu, void *usr);

class DtUnknownPdu : public DtPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtUnknownPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtUnknownPdu(const DtNetPduHeader *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    static DtPdu *create(const DtNetPacket *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    static void addCallback(DtPduKind kind, DtExerciseConn *conn,
DtUnknownPduCb cb, void *usr);
    static void removeCallback(DtPduKind kind, DtExerciseConn *conn,
DtUnknownPduCb cb, void *usr);

/***** MUTATOR FUNCTIONS *****/

    void setKind(DtPduKind kind);

protected:
    DtPduKind internalGetPduKind() const;
};
#endif

```

updCtlCbInfo.h

```
#if DtHLA

#if !DtRTI103

#include "updCtlCbList.h"
#include "RTI.hh"
#include "stateMsg.h"

// class DtUpdateControlCbInfo:
//
// Instances of DtUpdateControlCbInfo are used for providing
// a callback when the RTI update control services are invoked.
// This class is a helper class to DtUpdateControlCbList.

class DtUpdateControlCbInfo
{
public:

    // constructor
    DtUpdateControlCbInfo(RTI::ObjectHandle hand,
        DtUpdateControlCb cb,
        void* usr);

    // destructor
    virtual ~DtUpdateControlCbInfo();

    // copy constructor
    DtUpdateControlCbInfo(
        const DtUpdateControlCbInfo& orig);

    // assignment operator
    DtUpdateControlCbInfo& operator=(
        const DtUpdateControlCbInfo& orig);

    // Check whether one callback info is equal to another.
    virtual int match(const DtUpdateControlCbInfo* const other) const;

    // Check whether one callback info is equal to another.
    virtual int operator==(const DtUpdateControlCbInfo& other) const;

    // Get the object handle.
    virtual const RTI::ObjectHandle& objectHandle() const;

    // Get the callback function.
    virtual const DtUpdateControlCb& cbFunction() const;

    // Get the user data.
    virtual void* userData() const;

protected:

    RTI::ObjectHandle myHand;
    DtUpdateControlCb myFunc;
    void* myUsr;
};

typedef DtUpdateControlCbInfo DtUpdatesOnCbInfo;
typedef DtUpdateControlCbInfo DtUpdatesOnCbInfo;

#endif
#endif
```

updCtlCbList.h

```

#if DtHLA

#if !DtRTI103

#include "vllist.h"
#include "RTI.hh"

// class DtUpdateControlCbList:
//
// Instances of DtUpdateControlCbList are used to provide
// a list of callbacks (DtUpdateControlCbInfo) which are
// called when an object starts updating.

typedef void (*DtUpdateControlCb)(
    RTI::ObjectHandle handle,
    const RTI::AttributeHandleSet& attributeSet,
    void* usr);

typedef DtUpdateControlCb DtUpdatesOnCb;
typedef DtUpdateControlCb DtUpdatesOffCb;

class DtUpdateControlCbList
{
public:
    // default constructor
    DtUpdateControlCbList();

    // destructor
    virtual ~DtUpdateControlCbList();

    // copy constructor
    DtUpdateControlCbList(
        const DtUpdateControlCbList& orig);

    // assignment operator
    DtUpdateControlCbList& operator=(
        const DtUpdateControlCbList& orig);

    // Add/Remove a start updates callback to/from the list.
    // Returns whether add/remove took place.
    virtual int add(RTI::ObjectHandle hand, DtUpdateControlCb cb,
        void* usr);
    virtual int remove(RTI::ObjectHandle hand, DtUpdateControlCb cb,
        void* usr);

    // Remove all callbacks from the list.
    virtual void clear();

    // Callbacks are for a particular object class handle.
    // Call any callbacks in the list for the object class handle.
    virtual void apply(RTI::ObjectHandle handle,
        const RTI::AttributeHandleSet& hSet);

protected:
    // Remove all callbacks from the list.
    // Non-virtual since meant to be called from dtor, otherwise
    // should call clear().
    void dtorClear();

    // Copy another callback list into this object's callback list.
    // Non-virtual since meant to be called from ctor.

```

```
    void ctorCopyList(const DtUpdateControlCbList& orig);
protected:
    DtList myList;
};
typedef DtUpdateControlCbList DtUpdatesOnCbList;
typedef DtUpdateControlCbList DtUpdatesOffCbList;
#endif
#endif
```

utmCoord.h

```

#include "vlVector.h"
#include "geodCoord.h"

// class DtUtmCoord:
//
// Instances of DtUtmCoord are used to represent UTM coordinates.
// Universal Transverse Mercator projection coordinate with respect to
// reference set by DtUtmInit.

class DtUtmCoord : public DtVector
{
    friend void DtGeodToUtm(const DtGeodeticCoord& gd, DtUtmCoord& utm);
    friend void DtGeocToUtm(DtConstVector gd, DtUtmCoord& utm);

public:
    //
    // Note: For coordinate conversions, DtUtmInit must be called first.
    //

    // default constructor
    DtUtmCoord();

    // constructor
    DtUtmCoord(double east, double north, double up);

    // constructor, construct UTM from geodetic
    DtUtmCoord(const DtGeodeticCoord& geodetic);

    // destructor
    virtual ~DtUtmCoord();

    // copy constructor
    DtUtmCoord(const DtUtmCoord& orig);

    // assignment operator
    DtUtmCoord& operator=(const DtUtmCoord& orig);

    // assignment operator, convert geodetic to UTM.
    DtUtmCoord& operator=(const DtGeodeticCoord& geodetic);

    // Set from geodetic coordinate
    // (convert geodetic to UTM).
    // Can also use assignment operator.
    virtual DtUtmCoord& setGeodetic(const DtGeodeticCoord& geodetic);

    // Get geodetic coordinate equivalent
    // (convert UTM to geodetic).
    virtual void getGeodetic(DtGeodeticCoord& geodetic) const;

    // Get geodetic coordinate equivalent
    // (convert UTM to geodetic).
    // Note that this is not returning a reference.
    virtual DtGeodeticCoord geodetic() const;

    // Set from geocentric coordinate
    // (convert geocentric to UTM).
    // Can also use assignment operator.
    virtual DtUtmCoord& setGeocentric(DtConstVector geocentric);

    // Get geocentric coordinate equivalent
    // (convert UTM to geocentric).
    virtual void getGeocentric(DtVectorRef geocentric) const;

```

```
// Get geocentric coordinate equivalent
// (convert UTM to geocentric).
// Note that this is not returning a reference.
virtual DtVector geocentric() const;

// Set/Get east component.
virtual void setEast(DtFloat64 east);
virtual DtFloat64 east() const;

// Set/Get north component.
virtual void setNorth(DtFloat64 north);
virtual DtFloat64 north() const;

// Set/Get up component.
virtual void setUp(DtFloat64 up);
virtual DtFloat64 up() const;

// Get the heading that northing is off of north
virtual double gridDeclination() const;
};
// Coordinate conversions. DtUtmInit must be called first.

// Use of these functions is discouraged in favor of DtUtmCoord
// member functions which perform conversions.
void DtGeodToUtm(const DtGeodeticCoord& in, DtUtmCoord& out);
void DtUtmToGeod(const DtUtmCoord& utm, DtGeodeticCoord& out);

// Use of these functions is discouraged in favor of DtUtmCoord
// member functions which perform conversions.
void DtGeocToUtm(DtConstVector in, DtUtmCoord& out);
void DtUtmToGeoc(const DtUtmCoord& in, DtVectorRef out);

DtConstDcm DtNedToEnuDcm();
DtConstDcm DtEnuToNedDcm();

// Use of these functions is discouraged in favor of DtUtmCoord
// member functions which perform conversions.
double DtUtmGridDeclination(const DtUtmCoord& utm);
double DtUtmGridDeclination(double lat, double lon);

void DtGridDeclinationDcm(double gridDecl, DtDcmRef utm2ned);

enum DtPolarDirection { DtEast, DtWest, DtNorth, DtSouth };

typedef struct
{
    double deg, min, sec;
    DtPolarDirection direction;
} DtDegMinSec;

// Sets reference point for all DtUtmCoords
// Vrlink presently supports only one UTM reference at a time,
// but the reference point may be reinitialized at any point.
//
// If standardOffsets is true, the standard false easting (500,000 m)
// and false northing (10,000,000 m only when south of equator) will
// be applied.
//
// DtUtmInit expects lonRef between +/- 180.
// The UTM zone will be set by the zone that the reference point falls
// in, where zone N extends from (N*6 - 186) to (N*6 - 180).

void DtUtmInit(const DtDegMinSec& latRef,
               const DtDegMinSec& lonRef,
               int standardOffsets=1,
               int verbose=1);
```



```
// Some constants:

// Origin of the IITSEC '92 terrain database
const DtDegMinSec DtUtmLatRef1_1 = { 35.0, 15.0, 0.0, DtNorth};
const DtDegMinSec DtUtmLonRef1_1 = {122.0, 3.0, 59.0, DtWest};

// Origin of the IITSEC '94 & '95 terrain database
const DtDegMinSec DtUtmLatRef_IITSEC_93 = { 35.0, 15.0, 0.0, DtNorth};
const DtDegMinSec DtUtmLonRef_IITSEC_93 = {122.0, 4.0, 0.0, DtWest};

// The standard UTM reference for a zone is the point along the
// zone's central meridian at the equator. Standard UTM positions
// also use the standard false easting and false easting (see DtUtmInit)
//
// This is the standard UTM reference position for UTM zone 10

const DtDegMinSec DtUtmLatRefZone10 = { 0.0, 0.0, 0.0, DtNorth};
const DtDegMinSec DtUtmLonRefZone10 = {123.0, 0.0, 0.0, DtWest};

// inline functions
```

utmView.h

```
#include "LibMatrix.h"
#include "utmCoord.h"
// class DtUtmView:
//
// Instances of DtUtmView are used to create a UTM coordinate view
// of the geocentric coordinates in an Entity State Repository.

class DtUtmView
{
public:

    // constructor
    DtUtmView(DtBaseEntityStateRepository* esr);

    // destructor
    virtual ~DtUtmView();

    // copy constructor
    DtUtmView(const DtUtmView& orig);

    // assignment operator
    DtUtmView& operator=(const DtUtmView& orig);

    // Set/Get the location in UTM coordinates.
    virtual void setLocation(const DtUtmCoord& aLocation);
    virtual const DtUtmCoord& location() const;

    // Set/Get the velocity.
    virtual void setVelocity(const DtUtmCoord& aVelocity);
    virtual const DtUtmCoord& velocity() const;

    // Set/Get the orientation.
    virtual void setOrientation(const DtTaitBryan& orient);
    virtual const DtTaitBryan& orientation() const;

    // Get the body to local coordinate matrix. This is just
    // a matrix representation of the euler angles returned by
    // orientation().
    virtual DtConstDcm bodyToLocal() const;

    // Set/Get the acceleration.
    virtual void setAcceleration(const DtUtmCoord& anAcceleration);
    virtual const DtUtmCoord& acceleration() const;

    // Set/Get the rotational velocity.
    virtual void setRotationalVelocity(DtConstVector bodyRates);
    virtual DtConstVector rotationalVelocity() const;

protected:

    // Cast off const-ness for trivial internal change.
    // Non-virtual since can't override on return type.
    DtUtmView* self() const;

    // Get the matrix which transforms from UTM to geocentric.
    virtual DtConstDcm utmToGeocMatrix(DtConstVector geocLoc) const;

    // Get the matrix which transforms form geocentric to UTM.
    virtual DtConstDcm geocToUtmMatrix(DtConstVector geocLoc) const;

protected:

    DtBaseEntityStateRepository* myESR;
```

```
// used as return values by inspectors. Do not necessarily
// hold current state.
DtUtmCoord myLocalLoc;
DtUtmCoord myLocalVel;
DtUtmCoord myLocalAccel;
DtTaitBryan myLocalOrient;
DtDcm myBodyToLocal;

DtDcm myUtmToGeocMatrix;
DtVector myGeocLocForUtmToGeoc;

DtDcm myGeocToUtmMatrix;
DtVector myGeocLocForGeocToUtm;
};
```

varDatumRec.h

```
#include "disEnums.h"
#include "datumIdType.h"
#include "packets/dataP.h"

// class DtVariableDatumRecord:
//
// Instances of DtVariableDatumRecord are used to hold a single variable
datum
// record and to provide access to its network representation.

class DtVariableDatumRecord
{
public:
    // default constructor
    DtVariableDatumRecord();

    // destructor
    virtual ~DtVariableDatumRecord();

    // copy constructor
    DtVariableDatumRecord(const DtVariableDatumRecord& orig);

    // assignment operator
    DtVariableDatumRecord& operator=(const DtVariableDatumRecord& orig);

    // comparison operators
    int operator==(const DtVariableDatumRecord& otherguy) const;
    int operator!=(const DtVariableDatumRecord& otherguy) const;

    // Set the working buffer to an externally supplied buffer
    virtual void setExternalBuffer(char* bufferPtr);

    // Set/Get the datum length in bits. This method will re-allocate memory
    // if the buffer is an internal buffer.
    virtual void setDatumBitLength(DtU32 newDatumBitLength);
    virtual DtU32 datumBitLength() const;

    // Get the datum length in bytes (padded to an 8 byte boundary)
    virtual int datumByteLength() const;

    // Set/Get the variable datum ID
    virtual void setDatumID(DtDatumParam value);
    virtual DtDatumParam datumID() const;

    // Set/Get datum value as a DtFloat64
    // Set will set the datum bit length to 64
    virtual void setDatumValFloat64(DtFloat64 value);
    virtual DtFloat64 datumValFloat64() const;

    // Set/Get datum value as a DtFloat32
    // Set will set the datum bit length to 32
    virtual void setDatumValFloat32(DtFloat32 value);
    virtual DtFloat32 datumValFloat32() const;

    // Set/Get datum value as a DtInt32
    // Set will set the datum bit length to 32
    virtual void setDatumValInt32(DtInt32 value);
    virtual DtInt32 datumValInt32() const;

    // Set/Get datum value as a DtU32
    // Set will set the datum bit length to 32
```

```

virtual void setDatumValU32(DtU32 value);
virtual DtU32 datumValU32() const;

// Set/Get datum value as a DtInt16
// Set will set the datum bit length to 16
virtual void setDatumValInt16(DtInt16 value);
virtual DtInt16 datumValInt16() const;

// Set/Get datum value as a DtU16
// Set will set the datum bit length to 16
virtual void setDatumValU16(DtU16 value);
virtual DtU16 datumValU16() const;

// Set/Get datum value as a DtU8
// Set will set the datum bit length to 8
virtual void setDatumValU8(DtU8 value);
virtual DtU8 datumValU8() const;

// Set/Get datum value as a byte array. Data is copied to and from
// the network data buffer according to the specified length of the
record.
virtual void setDatumValByteArray(const DtU8* setFrom);
virtual void getDatumValByteArray(DtU8* copyTo) const;

// Get a pointer to the datum value inside of the record
virtual char* datumValPtr();

// Get a const pointer to the datum value inside of the record
virtual const char* constDatumValPtr() const;

// Get the network byte length of the entire datum record
virtual int networkByteLength() const;

// Get a pointer to the buffer which stores the network representation of
// the datum record
virtual const char* networkDataBuffer() const;

// Print the contents of the record
virtual void print() const;

protected:

// Cast the buffer to a DtNetVarDatum struct
virtual DtNetVarDatum* netStruct();

// Cast the buffer to a const DtNetVarDatum struct
virtual const DtNetVarDatum* constNetStruct() const;

// Delete the internal buffer if there is no external buffer
virtual void deleteIfNoExternal();

// Compute the total datum size in bytes from the datum bit length
virtual int datumByteSize(int datumBitLength) const;

// Compute the total record size in bytes from the datum bit length
virtual int recordByteSize(int datumBitLength) const;

protected:

char* myRecordBuffer;
char* myExternalBuffer;

};

```

vcPdu.h

```
#include "chooseDefines.h"
#include "hostStructs.h"
#include "pdu.h"
#include "packets/viewControlP.h"
#include "mathTypes.h"
#include "globalObjDes.h"

#if !DtHLA

class DtExerciseConn;
class DtViewControlPdu;
typedef DtViewControlPdu DtViewControlInteraction;
typedef void (
    *DtViewControlInteractionCb)(DtViewControlInteraction *inter, void
    *usr);
#include "interaction.h"

#endif // !DtHLA

#if DtHLA
class DtViewControlPdu : public DtPdu
#else
class DtViewControlPdu : public DtInteraction
#endif
{
public:

    static DtPdu *create(const DtNetPacket *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

#if !DtHLA

    static void addCallback(DtExerciseConn *conn,
        DtViewControlInteractionCb cb, void *usr);
    static void removeCallback(DtExerciseConn *conn,
        DtViewControlInteractionCb cb, void *usr);

#endif

/***** INSPECTOR FUNCTIONS *****/

    const DtGlobalObjectDesignator& viewerId() const;
    const DtGlobalObjectDesignator& senderId() const;
    DtViewerCommand viewerCommand() const;
    unsigned long ignoreFlags() const;

/***** MUTATOR FUNCTIONS *****/

    void setViewerId(const DtGlobalObjectDesignator& id);
    void setSenderId(const DtGlobalObjectDesignator& id);
    void setIgnoreFlags(unsigned long flags);

    virtual void setFromString(const char *) {}
    virtual char *string() const {return "";}

    void printData() const;

protected:
    DtPduKind internalGetPduKind() const {return DtViewControlPduKind;}
    DtGlobalObjectDesignator retId;
    void setViewerCommand(DtViewerCommand com);
};
```

```

DtViewControlPdu *DtNewViewControlPdu(DtViewerCommand command);

class DtAbsoluteVCPdu : public DtViewControlPdu
{
public:
    /***** STANDARD CONSTRUCTORS *****/

    DtAbsoluteVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtAbsoluteVCPdu(const DtNetViewControlPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    /***** INSPECTOR FUNCTIONS *****/

    DtConstVector worldLocation() const;
    const DtTaitBryan &orientation() const;

    /***** MUTATOR FUNCTIONS *****/

    void setWorldLocation(DtConstVector loc);
    void setOrientation(const DtTaitBryan &ori);

    void setFromString(const char *str);
    char *string() const;
    void printData() const;

protected:
    DtVector theVec;
    DtTaitBryan theAngles;
};

class DtTetherVCPdu : public DtViewControlPdu
{
public:
    /***** STANDARD CONSTRUCTORS *****/

    DtTetherVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtTetherVCPdu(const DtNetViewControlPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

    /***** INSPECTOR FUNCTIONS *****/

    DtConstVector topoUpOffset() const;
    const DtTaitBryan &orientation() const;
    const DtGlobalObjectDesignator& attachId() const;

    /***** MUTATOR FUNCTIONS *****/

    void setTopoUpOffset(DtConstVector offset);
    void setOrientation(const DtTaitBryan &ori);
    void setAttachId(const DtGlobalObjectDesignator& id);

    void setFromString(const char *str);
    char *string() const;
    void printData() const;

protected:
    DtVector theVec;
    DtTaitBryan theAngles;
    DtGlobalObjectDesignator theId;
};

class DtCompassVCPdu : public DtViewControlPdu
{
public:

```

```

/***** STANDARD CONSTRUCTORS *****/

DtCompassVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
DtCompassVCPdu(const DtNetViewControlPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

float radius() const;
float bodyHeading() const;
float pitch() const;
float headingOffset() const;
float pitchOffset() const;
float roll() const;
const DtGlobalObjectDesignator& attachId() const;

/***** MUTATOR FUNCTIONS *****/

void setRadius(float r);
void setBodyHeading(float h);
void setPitch(float p);
void setHeadingOffset(float o);
void setPitchOffset(float p);
void setRoll(float r);
void setAttachId(const DtGlobalObjectDesignator& id);

void setFromString(const char *str);
char *string() const;
void printData() const;

protected:
DtGlobalObjectDesignator theId;
};

class DtMimicVCPdu : public DtViewControlPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

DtMimicVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
DtMimicVCPdu(const DtNetViewControlPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

DtConstVector offset() const;
const DtTaitBryan &bodyOrientation() const;
const DtGlobalObjectDesignator& attachId() const;

/***** MUTATOR FUNCTIONS *****/

void setOffset(DtConstVector offset);
void setBodyOrientation(const DtTaitBryan &ori);
void setAttachId(const DtGlobalObjectDesignator& id);

void setFromString(const char *str);
char *string() const;
void printData() const;

protected:
DtGlobalObjectDesignator theId;
DtVector theVec;
DtTaitBryan theAngles;
};

```



```

class DtOrbitVCPdu : public DtViewControlPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtOrbitVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtOrbitVCPdu(const DtNetViewControlPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

    float radius() const;
    float azimuthRate() const;
    float heading() const;
    float pitch() const;
    float headingOffset() const;
    float pitchOffset() const;
    float roll() const;
    const DtGlobalObjectDesignator& attachId() const;

/***** MUTATOR FUNCTIONS *****/

    void setRadius(float r);
    void setAzimuthRate(float a);
    void setHeading(float h);
    void setPitch(float p);
    void setHeadingOffset(float o);
    void setPitchOffset(float p);
    void setRoll(float r);
    void setAttachId(const DtGlobalObjectDesignator& id);

    void setFromString(const char *str);
    char *string() const;
    void printData() const;

protected:
    DtGlobalObjectDesignator theId;
};

class DtTrackVCPdu : public DtViewControlPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtTrackVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtTrackVCPdu(const DtNetViewControlPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

    DtConstVector worldLocation() const;
    const DtGlobalObjectDesignator& trackId() const;

/***** MUTATOR FUNCTIONS *****/

    void setWorldLocation(DtConstVector loc);
    void setTrackId(const DtGlobalObjectDesignator& id);

    void setFromString(const char *str);
    char *string() const;
    void printData() const;

protected:
    DtVector theVec;
    DtGlobalObjectDesignator theId;
};

```

```
};

class DtTetherTrackVCPdu : public DtViewControlPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtTetherTrackVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtTetherTrackVCPdu(const DtNetViewControlPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

    DtConstVector offset() const;
    const DtGlobalObjectDesignator& attachId() const;
    const DtGlobalObjectDesignator& trackId() const;

/***** MUTATOR FUNCTIONS *****/

    void setOffset(DtConstVector loc);
    void setAttachId(const DtGlobalObjectDesignator& id);
    void setTrackId(const DtGlobalObjectDesignator& id);

    void setFromString(const char *str);
    char *string() const;
    void printData() const;

protected:
    DtVector theVec;
    DtGlobalObjectDesignator theId;
};

class DtMimicTrackVCPdu : public DtViewControlPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtMimicTrackVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
    DtMimicTrackVCPdu(const DtNetViewControlPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

    DtConstVector offset() const;
    const DtGlobalObjectDesignator& attachId() const;
    const DtGlobalObjectDesignator& trackId() const;

/***** MUTATOR FUNCTIONS *****/

    void setOffset(DtConstVector loc);
    void setAttachId(const DtGlobalObjectDesignator& id);
    void setTrackId(const DtGlobalObjectDesignator& id);

    void setFromString(const char *str);
    char *string() const;
    void printData() const;

protected:
    DtVector theVec;
    DtGlobalObjectDesignator theId;
};

class DtGroupVCPdu : public DtViewControlPdu
{
public:
```

```

/***** STANDARD CONSTRUCTORS *****/

DtGroupVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
DtGroupVCPdu(const DtNetViewControlPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

float radialOffset() const;
float heading() const;
float pitch() const;
float headingOffset() const;
float pitchOffset() const;
int groupCmd() const;
int numEntitiesToRemove() const;
int numEntitiesToAdd() const;
const DtGlobalObjectDesignator& entityToRemove(int n) const;
const DtGlobalObjectDesignator& entityToAdd(int n) const;

/***** MUTATOR FUNCTIONS *****/

void setRadialOffset(float o);
void setHeading(float h);
void setPitch(float p);
void setHeadingOffset(float o);
void setPitchOffset(float o);
void setGroupCmd(int c);
void setNumEntitiesToRemove(int n);
void setNumEntitiesToAdd(int n);
void setEntityToRemove(int n, const DtGlobalObjectDesignator& id);
void setEntityToAdd(int n, const DtGlobalObjectDesignator& id);

void setFromString(const char *str);
char *string() const;
void printData() const;

protected:
DtGlobalObjectDesignator theId;
};

class DtModelessVCPdu : public DtViewControlPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

DtModelessVCPdu(DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
DtModelessVCPdu(const DtNetViewControlPdu *initial,
DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);

/***** INSPECTOR FUNCTIONS *****/

const DtGlobalObjectDesignator& primary() const;
const DtGlobalObjectDesignator& secondary() const;
int numEntitiesToRemove() const;
int numEntitiesToAdd() const;
const DtGlobalObjectDesignator& entityToRemove(int n) const;
const DtGlobalObjectDesignator& entityToAdd(int n) const;

/***** MUTATOR FUNCTIONS *****/

void setPrimary(const DtGlobalObjectDesignator& id);
void setSecondary(const DtGlobalObjectDesignator& id);
void setNumEntitiesToRemove(int n);
void setNumEntitiesToAdd(int n);
void setEntityToRemove(int n, const DtGlobalObjectDesignator& id);

```

```
void setEntityToAdd(int n, const DtGlobalObjectDesignator& id);

void printData() const;

protected:
    DtGlobalObjectDesignator theId;
};

class DtUnknownVCPdu : public DtViewControlPdu
{
public:

/***** STANDARD CONSTRUCTORS *****/

    DtUnknownVCPdu(const DtNetViewControlPdu *initial,
        DtBufferPtr buffer = DtUSE_INTERNAL_BUFFER);
};

#define DtIGNORE_EAST                (1<<0)
#define DtIGNORE_NORTH              (1<<1)
#define DtIGNORE_HEIGHT             (1<<2)
#define DtIGNORE_HEADING            (1<<3)
#define DtIGNORE_PITCH              (1<<4)
#define DtIGNORE_ROLL               (1<<5)
#define DtIGNORE_EAST_OFFSET        (1<<6)
#define DtIGNORE_NORTH_OFFSET       (1<<7)
#define DtIGNORE_HEIGHT_OFFSET      (1<<8)
#define DtIGNORE_RADIUS             (1<<9)
#define DtIGNORE_BODY_HEADING       (1<<10)
#define DtIGNORE_HEADING_OFFSET     (1<<11)
#define DtIGNORE_PITCH_OFFSET       (1<<12)
#define DtIGNORE_FORWARD            (1<<13)
#define DtIGNORE_RIGHT              (1<<14)
#define DtIGNORE_UP                 (1<<15)
#define DtIGNORE_BODY_PITCH         (1<<16)
#define DtIGNORE_BODY_ROLL          (1<<17)
#define DtIGNORE_AZIMUTH_RATE        (1<<18)
#define DtIGNORE_RADIAL_OFFSET       (1<<19)
#define DtIGNORE_PRIMARY_ID          (1<<20)
#define DtIGNORE_SECONDARY_ID        (1<<21)
#define DtIGNORE_GROUP_IDS           (1<<22)

#endif
```

vList.h

```

#include <stddef.h>

class DtListItem
{
    friend class DtList;
public:
    DtListItem *prev() const { return _prev; }
    DtListItem *next() const { return _next; }
    void *data() const { return _data; }
    virtual ~DtListItem() {}
protected:
    DtListItem(void *d) : _next(0), _prev(0), _data(d) { }
    DtListItem *_prev;
    DtListItem *_next;
    void *_data;
};

class DtList
{
public:
    DtList() : _first(0), _last(0), _count(0) { }
    /* Note: ~DtList() does not free memory of data associated
     * with the items. It only frees the items. */
    virtual ~DtList();

    DtListItem *first() const { return _first; }
    DtListItem *last() const { return _last; }
    unsigned count() const { return _count; }

    DtListItem *add(void *data)
    { /* synonym for addToEnd */
return addAfter(data, _last);
    }
    DtListItem *addToStart(void *data)
    {
return addAfter(data, NULL);
    }
    DtListItem *addToEnd(void *data)
    {
return addAfter(data, _last);
    }
    DtListItem *addBefore (void *data, DtListItem *before)
    {
return addAfter(data, before ? before->_prev : (DtListItem*)NULL);
    }
    DtListItem *addAfter(void *data, DtListItem *after);

    /* Note: remove does not free memory of data associated
     * with the item. It only frees the item. The data is
     * returned to the caller for deletion if required. */
    void *remove(DtListItem *item);

    // Does NOT delete the data in the list. Merely removes the data
    // from the list. Data must be deleted later.
    void removeAll();

    DtListItem *itemLookup(void *data) const;

protected:
    virtual void addToList(DtListItem *item, DtListItem *after);
    virtual void removeFromList(DtListItem *item);

```

```
DtListItem *_first;
DtListItem *_last;
unsigned _count;
};

// An intrusive list is a list that requires that the object type
// that it is listing is derived from DtListedObject.

class DtIntrusiveList : public DtList
{
public:
    void addToList(DtListItem *item, DtListItem *after);
    void removeFromList(DtListItem *item);
};

class DtListedObj
{
    friend class DtIntrusiveList;
public:
    DtListedObj() : myList(NULL), item(NULL) {}
    virtual ~DtListedObj() {}
    void *next() const { return (item && item->next()) ? item->next()-
>data() : NULL; }
    void *prev() const { return (item && item->prev()) ? item->prev()-
>data() : NULL; }
    void *wrapNext() const
    {
        return next() ? next() :
((myList && myList->first()) ? myList->first()->data() : NULL);
    }
    void *wrapPrev() const
    {
        return prev() ? prev() :
((myList && myList->last()) ? myList->last()->data() : NULL);
    }

protected:
    DtList *myList;
    DtListItem *item;
};

#define looplist(theData, list) \
    for (DtListItem *item=(list)->first(); \
    theData = (item ? item->data() : NULL); \
    item = item->next())

/* typed looplist for cfront
 * (ANSI C and GCC don't require casting from void *) */
#define looptlist(theData, type, list) \
    for (DtListItem *item=(list)->first(); \
    theData = (type)(item ? item->data() : NULL); \
    item = item->next())

/* looptlist for multiple loops in same scope with
 * C++ without proper scope of loop vars */
#define looptlist2(theData, type, list) \
    for (item=(list)->first(); \
    theData = (type)(item ? item->data() : NULL); \
    item = item->next())

#endif
```

vlprint.h

```

#include "vlconfig.h"

/* vlprint.h
 * These functions provide a simple error and print facility for vlink
 */

/*
 * DtNotifyLevel is used to control the level of diagnostics to one of
 * the following levels. Each level includes messages for all lower
 * levels as well. The default level is DtNlInfo, i.e. all messages
 * except debug statements.
 */

/* Notification Levels */
#define DtNlFatal 0
#define DtNlWarn 1
#define DtNlInfo 2
#define DtNlVerbose 3

extern int DtNotifyLevel; /* default is DtNlInfo */

/* These functions print messages using stdio depending on the notify level.
 * Fatal and Warn print to stderr.
 * Info prints to stdout.
 * Fatal funcs call DtAbort at end, even if (notifyLevel < DtNlFatal)
 * Fatal prints are preceded by "***FATAL ERROR**:"
 * the DtFatalError print is followed by \n
 * Perror prints are followed by a call to perror()
 */
void DtFatalError(const char *s, ...);
void DtFatalPerror(const char *s, ...);
void DtWarnPerror(const char *s, ...);
void DtFatalNetPerror(const char *s, ...);
void DtWarnNetPerror(const char *s, ...);
void DtWarn(const char *s, ...);
void DtInfo(const char *s, ...);
void DtVerbose(const char *s, ...);

/* DtAbort() exits the program.
 * The user can provide for different behaviour [e.g. calling abort()]
 * by linking with a another definition DtAbort [i.e. It is in its own
 * object file.], but the user's DtAbort() should not return.
 */
void DtAbort(void);

#ifdef WIN32
void setFileOutput(constchar *filename, int con=0, int son=0);
#endif

```

v/Time.h

```
#include "MachineTypes.h"

typedef DtReal DtTime;
const double DtHour2Sec = 3600;
const double DtSec2Hour = 1./3600;

// Initialize all VR-Link time functionality
// This must be called before any other time-related functions
void DtTimeInit(void);

// Simulation Time Calls

// Returns the last value passed to DtSetSimTime - the current value
// of VR-Link simulation time.
DtTime DtGetSimTime(void);

// Set the current value of VR-Link simulation time. Should be
// called once per simulation frame to let VR-Link know the value
// of simulation time for the frame.
void DtSetSimTime(DtTime NewSimTime);

// Abort if DtSetSimTime was never called
void DtIsSimTimeSet(void);

// Return real seconds elapsed since DtTimeInit was called
DtTime DtGetElapsedRealTime(void);

// Obsolete. This function exists for backwards compatibility with
// older versions of VR-Link. Use DtGetElapsedRealTime instead.
DtTime DtTimeNow(void);

// Returns the value of DtGetElapsedRealTime at the time that
// DtSetSimTime was last invoked.
DtTime DtGetApproximateElapsedRealTime(void);

// Return current wallclock time - seconds since Jan 1, 1970
DtTime DtAbsRealTime(void);

extern int DtUseSimTimeForApproxRealTime;
```


vlutil.h

```

#include "vlconfig.h"

#if WIN32
#include <winsock.h>
#endif

#if DT_HAVE_UNISTD_H
#include <unistd.h>
#endif

#ifdef PCFTP
#include <4bsddefs.h>
#endif

extern "C" {
#include <stdlib.h>
#include <sys/types.h>
}
#include "vlTime.h"
#include "math.h"

#define DtDELETE(x) \
    if (x) \
    { \
        delete x; \
        x = NULL; \
    }

#ifndef DtISON
#define DtISON(a, x) (a & x)
#define DtISOFF(a, x) (!DtISON(a, x))
#define DtTURNON(a, x) (a |= x)
#define DtTURNOFF(a, x) (a &= ~x)
#endif

#define DtMIN(x, y) ((x)<(y) ? (x) : (y))
#define DtMAX(x, y) ((x)>(y) ? (x) : (y))
#define DtLIMIT(low, x, hi) DtMAX(DtMIN(x, hi), low)
#define DtABS(x) ((x) < 0 ? -(x) : (x))
#define DtSQR(x) ((x)*(x))

#define DtNUMBER(foo) (sizeof(foo)/sizeof(foo[0]))
#define DtOFFSET(type, field) ((size_t)&((type *)NULL)->field)
#define DtSIZEOF(type, field) sizeof(((type *)NULL)->field)

// Rounds num up to a multiple of mult.
/* some versions of strtol returns INT_MAX for input larger than
   0x7fffffff. As long as strtoul accepts and properly processes
   negative input, this macro will work for both unsigned and signed
   ints (as other strtol do). */

#if DT_HAVE_STRTOUL
#define Dtatoi(s) ((int)(strtoul(s, 0, 0)))
#else
#define Dtatoi(s) (int)strtol(s, 0, 0)
#endif

char *DtDeg2DmsString(double val, char *buff, int prec);
char *DtDeg2DmsString(double val);

// Returns -1 if num is not in lst. Otherwise, returns the index
// in the list at which num is first found. lst must be a zero-terminated
// array of integers.

```

```
int DtIsInList(const int *lst, int num);

#if DT_FD_ZERO_USES_MEMSET
#include <string.h>
#else /* need bzero */

#if DT_HAVE_BZERO_PROTO_IN_MEMORY_H
#include <memory.h>
#elif DT_HAVE_BZERO_PROTO_IN_CC_LIBC_H
#include <CC/libc.h>
#elif DT_HAVE_BZERO_PROTO_IN_BSTRING_H
#include <bstring.h>
#else
extern "C" { void bzero(char *b, int length); }
#endif

#endif /* DT_FD_ZERO_USES_MEMSET */

#if __SUNPRO_CC
#include <sys/file.h>
#endif

int DtSelect(int n, fd_set *i, fd_set *o, fd_set *e, DtTime t);

#if WIN32
#else
#endif

class DtSelectParam
{
public:
    DtSelectParam() {
        FD_ZERO(&read0);
        read = read0;
        maxfd = -1;
    }
    int select(DtTime timeout) {
        read = read0;
        return DtSelect(maxfd+1, &read, NULL, NULL, timeout);
    }
    void addRead(int fd) {
        FD_SET(fd, &read0);
        maxfd = DtMAX(maxfd, fd);
    }
    void delRead(int fd) {
        FD_CLR(fd, &read0);
        /* should recalc maxfd */
    }
    int canRead(int fd) {
        return ((int)(FD_ISSET(fd, &read)));
    }
private:
    int maxfd;
    fd_set read0, read;
};

// Strdup wrapper. Returns NULL if str is NULL.
// Otherwise returns a newed char* containing a copy of str.
// Caller deletes using delete[].
char* DtStrdup(const char* str);

// Convert str to all lowercase.
void DtToLower(char* str);

// Convert str to all uppercase.
void DtToUpper(char* str);
```

```
/* ----- inet utils ----- */

typedef DtU32 DtInetAddr;

const DtInetAddr DtInetAddrAny = 0;
const DtInetAddr loopbackAddr = 0x7f000001; /* 127.0.0.1 */
/* converts ulong to internet dot notation (returns internal static buffer)
*/
const char *DtInetAddrString(DtInetAddr addr);

/* converts to internet dot notation to ulong in host order */
#if _PowerUX
DtInetAddr DtStringToInetAddr(char *str);
#else
DtInetAddr DtStringToInetAddr(const char *str);
#endif

DtInetAddr DtInetBroadcastOfDevice(const char *device);
DtInetAddr DtInetAddrOfDevice(const char *device);
DtInetAddr DtNetMaskOfDevice(const char *device);
DtInetAddr DtFirstHostInetAddr();
int DtIsMulticastAddr(DtInetAddr addr);
```

v/Vector.h

```
enum DtVectorOffset
{
    DtX,
    DtY,
    DtZ
};

typedef const DtVector& DtConstVector;
typedef DtVector& DtVectorRef;

// class DtVector:
//
// No virtual functions, because we want the size of a DtVector
// to 3 * 8 bytes - so that you can make the same assumptions about it
// that we used to make about its size and layout. This is especially
// important in DtDcm, where we use an array of 3 DtVectors as the myRep.
// If DtVector had any virtual functions, there would be 4 bytes between
// vector myReps.
class DtVector
{
public:
    // default constructor
    DtVector();

    // construct from three doubles.
    DtVector(double x, double y, double z);

    // destructor
    ~DtVector();

    // copy constructor
    DtVector(const DtVector& vec);

    // assignment operator
    DtVector& operator=(const DtVector& vec);

    // Convert the vector to a pointer to double.
    operator double* ();
    operator const double* () const;

    // Get the value for the specified subscript. (int version)
    double& operator[](int ind);
    double operator[](int ind) const;

    // Get the value for the specified subscript. (enum version)
    double& operator[](DtVectorOffset ind);
    double operator[](DtVectorOffset ind) const;

    // Get whether another vector equals this vector.
    int operator==(const DtVector& vec) const;

    // Get a string representation of the vector.
    const char* string() const;

public:
    // Get a zero'ed vector.
    static DtConstVector zero();

    // Get a (1,0,0) vector.
    static DtConstVector i();
```

```
// Get a (0,1,0) vector.
static DtConstVector j();

// Get a (0,0,1) vector.
static DtConstVector k();

// Get a vector set to all ones.
static DtConstVector ones();

protected:
    double myRep[3];
};
// Inline DtVector functions
```

vrIFedAmb.h

```
#if DtHLA

#if DtRTI103
#include "vFedAmb103.h"
#else

#include "RTI.hh"
#include "NullFederateAmbassador.hh"
#include "callbacks.h"

#include "chooseDefines.h"

// class DtVrIFederateAmbassador:
//
// Instances of DtVrIFederateAmbassador are used to manage the
// interface between the RTI and the VR-Link Federate.

class DtVrIFederateAmbassador : public NullFederateAmbassador
{
public:

    // default constructor
    DtVrIFederateAmbassador();

    // destructor
    virtual ~DtVrIFederateAmbassador();

    // copy constructor
    DtVrIFederateAmbassador(const DtVrIFederateAmbassador& orig);

    // assignment operator
    DtVrIFederateAmbassador& operator=(const DtVrIFederateAmbassador& orig);

    // Initialize the ambassador.
    // Needs to be called after construction.
    virtual void init(DtReflectedHlaObjectManager* reflectedObjMgr,
        DtLocalHlaObjectManager* localObjMgr,
        DtInteractionManager* interMgr);

    // Add/Remove (register/deregister) function to be called when
    // the RTI's reflectAttributeValues service is invoked.
    virtual void addReflectAttributeValuesCb(RTI::ObjectHandle id,
        DtReflectAttributeValuesCb cb,
        void* usr);
    virtual void removeReflectAttributeValuesCb(RTI::ObjectHandle id,
        DtReflectAttributeValuesCb cb,
        void* usr);

    // Add/Remove (register/deregister) function to be called when
    // the RTI's discoverObject service is invoked.
    virtual void addDiscoverObjectCb(RTI::ObjectClassHandle hand,
        DtDiscoverObjectCb cb,
        void* usr);
    virtual void removeDiscoverObjectCb(RTI::ObjectClassHandle hand,
        DtDiscoverObjectCb cb,
        void* usr);

    // Add/Remove (register/deregister) function to be called when
    // the RTI's removeObject service is invoked.
    virtual void addRemoveObjectCb(RTI::ObjectHandle id,
        DtRemoveObjectCb, void* usr);
    virtual void removeRemoveObjectCb(RTI::ObjectHandle id,
        DtRemoveObjectCb, void* usr);
```

```

// Add/Remove (register/deregister) function to be called when
// the RTI's startRegistration service is invoked.
virtual void addStartRegCb(RTI::ObjectClassHandle hand,
    DtStartRegCb, void* usr);
virtual void removeStartRegCb(RTI::ObjectClassHandle hand,
    DtStartRegCb, void* usr);

// Add/Remove (register/deregister) function to be called when
// the RTI's stopReg service is invoked.
virtual void addStopRegCb(RTI::ObjectClassHandle hand,
    DtStopRegCb, void* usr);
virtual void removeStopRegCb(RTI::ObjectClassHandle hand,
    DtStopRegCb, void* usr);

// Add/Remove (register/deregister) function to be called when
// the RTI's turnUpdatesOn service is invoked.
virtual void addUpdatesOnCb(RTI::ObjectHandle hand,
    DtUpdatesOnCb, void* usr);
virtual void removeUpdatesOnCb(RTI::ObjectHandle hand,
    DtUpdatesOnCb, void* usr);

// Add/Remove (register/deregister) function to be called when
// the RTI's turnUpdatesOff service is invoked.
virtual void addUpdatesOffCb(RTI::ObjectHandle hand,
    DtUpdatesOffCb, void* usr);
virtual void removeUpdatesOffCb(RTI::ObjectHandle hand,
    DtUpdatesOffCb, void* usr);

// Add/Remove (register/deregister) function to be called when
// the RTI's startInteractionGeneration service is invoked.
virtual void addStartInteractionsCb(RTI::InteractionClassHandle hand,
    DtStartStopInterCb, void* usr);
virtual void removeStartInteractionsCb(RTI::InteractionClassHandle hand,
    DtStartStopInterCb, void* usr);

// Add/Remove (register/deregister) function to be called when
// the RTI's stopInteractionGeneration service is invoked.
virtual void addStopInteractionsCb(RTI::InteractionClassHandle hand,
    DtStartStopInterCb, void* usr);
virtual void removeStopInteractionsCb(RTI::InteractionClassHandle hand,
    DtStartStopInterCb, void* usr);

// Add/Remove (register/deregister) function to be called when
// the RTI's receiveInteraction service is invoked.
virtual void addReceiveInteractionCb(RTI::InteractionClassHandle hand,
    DtReceiveInteractionCb cb, void* usr);
virtual void removeReceiveInteractionCb(RTI::InteractionClassHandle
hand,
    DtReceiveInteractionCb cb, void* usr);

protected:

    virtual void startRegistrationForObjectClass (
        RTI::ObjectClassHandle theClass);

    virtual void stopRegistrationForObjectClass (
        RTI::ObjectClassHandle theClass);

    virtual void turnInteractionsOn (
        RTI::InteractionClassHandle theHandle);

    virtual void turnInteractionsOff (
        RTI::InteractionClassHandle theHandle);

    virtual void discoverObjectInstance (
        RTI::ObjectHandle theObject,

```

```
    RTI::ObjectClassHandle theObjectClass,
    const char* theObjectName);

virtual void reflectAttributeValues (
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleValuePairSet& theAttributes,
    const RTI::FedTime& theTime,
    const char* theTag,
    RTI::EventRetractionHandle theHandle);

virtual void reflectAttributeValues (
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleValuePairSet& theAttributes,
    const char* theTag);

virtual void receiveInteraction (
    RTI::InteractionClassHandle theInteraction,
    const RTI::ParameterHandleValuePairSet& theParameters,
    const RTI::FedTime& theTime,
    const char* theTag,
    RTI::EventRetractionHandle theHandle);

virtual void receiveInteraction (
    RTI::InteractionClassHandle theInteraction,
    const RTI::ParameterHandleValuePairSet& theParameters,
    const char* theTag);

virtual void removeObjectInstance (
    RTI::ObjectHandle theObject,
    const RTI::FedTime& theTime,
    const char* theTag,
    RTI::EventRetractionHandle theHandle);

virtual void removeObjectInstance (
    RTI::ObjectHandle theObject,
    const char* theTag);

virtual void provideAttributeValueUpdate (
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes);

virtual void turnUpdatesOnForObjectInstance (
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes);

virtual void turnUpdatesOffForObjectInstance (
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes);

public:

    // returns a new DtVrlFedAmbassador
    static DtVrlFederateAmbassador* create();

protected:

    DtInteractionManager* myInteractionManager;
    DtReflectedHlaObjectManager* myReflectedObjectManager;
    DtLocalHlaObjectManager* myLocalObjectManager;

    DtReflectAttributeValuesCbList myRavCbList;
    DtUpdatesOnCbList myUpdatesOnCbList;
    DtUpdatesOffCbList myUpdatesOffCbList;
    DtStartRegCbList myStartRegCbList;
    DtStopRegCbList myStopRegCbList;
    DtStartStopInterCbList myStartIntCbList;
    DtStartStopInterCbList myStopIntCbList;
```



```
};  
#endif  
#endif
```

vrlFedTime.h

```
// This file is based on MyFedTime.h, which appears in the DMSO RTI
// MyFedTime example in RTI_HOME/lang/C++/demo/MyFedTime.

#ifndef DtVrlFedTime_H_
#define DtVrlFedTime_H_

#if DtHLA

#include <RTI.hh>
#include <sys/types.h>
#include <string.h>

class RTI_EXPORT DtVrlFedTime : public RTI::FedTime {
//-----
// Constructors and Destructors
//-----
public:
    DtVrlFedTime();
    DtVrlFedTime(const double&);
    DtVrlFedTime(const DtVrlFedTime&);
    virtual ~DtVrlFedTime();

//-----
// Overloaded functions from RTI::FedTime
//-----
public:
    virtual void                setZero();
    virtual RTI::Boolean        isZero();
    virtual void                setEpsilon();
    virtual void                setPositiveInfinity();
    virtual RTI::Boolean        isPositiveInfinity();
    virtual int                 encodedLength() const;
    virtual void                encode(char *buff) const;
    virtual int                 getPrintableLength() const;
    virtual void                getPrintableString(char*);

//-----
// Overloaded operators from RTI::FedTime
//-----
public:
    virtual RTI::FedTime& operator+= (const RTI::FedTime&)
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator-= (const RTI::FedTime&)
        throw (RTI::InvalidFederationTime);

    virtual RTI::Boolean operator<= (const RTI::FedTime&) const
        throw (RTI::InvalidFederationTime);

    virtual RTI::Boolean operator< (const RTI::FedTime&) const
        throw (RTI::InvalidFederationTime);

    virtual RTI::Boolean operator>= (const RTI::FedTime&) const
        throw (RTI::InvalidFederationTime);

    virtual RTI::Boolean operator> (const RTI::FedTime&) const
        throw (RTI::InvalidFederationTime);

    virtual RTI::Boolean operator== (const RTI::FedTime&) const
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator= (const RTI::FedTime&)
        throw (RTI::InvalidFederationTime);
};
```

```

//-----
// Implementation functions
//-----
public:
    virtual double          getTime() const;

//-----
// Implementation operators
//-----
    virtual RTI::Boolean operator== (const double&) const
        throw (RTI::InvalidFederationTime);

    virtual RTI::Boolean operator!= (const RTI::FedTime&) const
        throw (RTI::InvalidFederationTime);

    virtual RTI::Boolean operator!= (const double&) const
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator= (const DtVrIFedTime&)
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator= (const double&)
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator*= (const RTI::FedTime&)
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator/= (const RTI::FedTime&)
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator+= (const double&)
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator-= (const double&)
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator*= (const double&)
        throw (RTI::InvalidFederationTime);

    virtual RTI::FedTime& operator/= (const double&)
        throw (RTI::InvalidFederationTime);

    virtual DtVrIFedTime operator+ (const RTI::FedTime&)
        throw (RTI::InvalidFederationTime);

    virtual DtVrIFedTime operator+ (const double&)
        throw (RTI::InvalidFederationTime);

    virtual DtVrIFedTime operator- (const RTI::FedTime&)
        throw (RTI::InvalidFederationTime);

    virtual DtVrIFedTime operator- (const double&)
        throw (RTI::InvalidFederationTime);

    virtual DtVrIFedTime operator* (const RTI::FedTime&)
        throw (RTI::InvalidFederationTime);

    virtual DtVrIFedTime operator* (const double&)
        throw (RTI::InvalidFederationTime);

    virtual DtVrIFedTime operator/ (const RTI::FedTime&)
        throw (RTI::InvalidFederationTime);

    virtual DtVrIFedTime operator/ (const double&)
        throw (RTI::InvalidFederationTime);

```

```
//-----  
// Implementation friends  
//-----  
public:  
    friend ostream& operator<< (ostream&, RTI::FedTime&);  
  
//-----  
// Implementation member variables  
//-----  
private:  
    double          _fedTime;  
    double          _zero;  
    double          _epsilon;  
    double          _positiveInfinity;  
};  
//-----  
// Global operators  
//-----  
  
DtVrlFedTime operator+ (const double&, const RTI::FedTime&);  
DtVrlFedTime operator- (const double&, const RTI::FedTime&);  
DtVrlFedTime operator* (const double&, const RTI::FedTime&);  
DtVrlFedTime operator/ (const double&, const RTI::FedTime&);  
  
#endif
```

vrlRtiAmb.h

```

#if DtHLA

#if DtRTI103
#include "vRtiAmb103.h"
#else

#ifndef DtVrlRtiAmbassador_H_
#define DtVrlRtiAmbassador_H_

#include "RTI.hh"

// class DtVrlRtiAmbassador:
//
// DtVrlRtiAmbassador is a purely virtual class, designed to provide
// a foundation for different ways to send RTI messages.

class DtVrlRtiAmbassador
{
public:

    // default constructor
    DtVrlRtiAmbassador();

    // destructor
    virtual ~DtVrlRtiAmbassador();

    // copy constructor
    DtVrlRtiAmbassador(const DtVrlRtiAmbassador& orig);

    // assignment operator
    DtVrlRtiAmbassador& operator=(const DtVrlRtiAmbassador& orig);

    // Get the real RTI ambassador, if the implementation of the derived
    // object permits. Otherwise returns NULL.
    virtual RTI::RTIAmbassador* rtiAmb();

    virtual void createFederationExecution (
        const char *executionName,
        const char *FED) = 0;

    virtual void destroyFederationExecution (
        const char *executionName) = 0;

    virtual RTI::FederateHandle joinFederationExecution (
        const char* yourName,
        const char* executionName,
        RTI::FederateAmbassadorPtr federateAmbassadorReference) = 0;

    virtual void resignFederationExecution(
        RTI::ResignAction theAction) = 0;

    virtual void registerFederationSynchronizationPoint(
        const char* label,
        const char* theTag) = 0;

    virtual void registerFederationSynchronizationPoint(
        const char* label,
        const char* theTag,
        const RTI::FederateHandleSet& syncSet) = 0;

    virtual void synchronizationPointAchieved(
        const char* label) = 0;

```

```
virtual void requestFederationSave (
    const char* label,
    const RTI::FedTime& theTime) = 0;

virtual void requestFederationSave (
    const char *label) = 0;

virtual void federateSaveBegun() = 0;

virtual void federateSaveComplete() = 0;

virtual void federateSaveNotComplete() = 0;

virtual void requestFederationRestore(const char* label) = 0;

virtual void federateRestoreComplete() = 0;

virtual void federateRestoreNotComplete() = 0;

virtual void publishObjectClass(
    RTI::ObjectClassHandle theClass,
    const RTI::AttributeHandleSet& attributeList) = 0;

virtual void unpublishObjectClass(
    RTI::ObjectClassHandle theClass) = 0;

virtual void publishInteractionClass(
    RTI::InteractionClassHandle theInteraction) = 0;

virtual void unpublishInteractionClass(
    RTI::InteractionClassHandle theInteraction) = 0;

virtual void subscribeObjectClassAttributes(
    RTI::ObjectClassHandle theClass,
    const RTI::AttributeHandleSet& attributeList,
    RTI::Boolean active = RTI::RTI_TRUE) = 0;

virtual void unsubscribeObjectClass(
    RTI::ObjectClassHandle theClass) = 0;

virtual void subscribeInteractionClass(
    RTI::InteractionClassHandle theClass,
    RTI::Boolean active = RTI::RTI_TRUE) = 0;

virtual void unsubscribeInteractionClass(
    RTI::InteractionClassHandle theClass) = 0;

virtual RTI::ObjectHandle registerObjectInstance(
    RTI::ObjectClassHandle theClass, const char* theObject) = 0;

virtual RTI::ObjectHandle registerObjectInstance(
    RTI::ObjectClassHandle theClass) = 0;

virtual RTI::EventRetractionHandle updateAttributeValues(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleValuePairSet& theAttributes,
    const RTI::FedTime& theTime,
    const char* theTag) = 0;

virtual void updateAttributeValues(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleValuePairSet& theAttributes,
    const char* theTag) = 0;

virtual RTI::EventRetractionHandle sendInteraction(
    RTI::InteractionClassHandle theInteraction,
    const RTI::ParameterHandleValuePairSet& theParameters,
```

```

    const RTI::FedTime& theTime,
    const char* theTag) = 0;

virtual void sendInteraction(
    RTI::InteractionClassHandle theInteraction,
    const RTI::ParameterHandleValuePairSet& theParameters,
    const char* theTag) = 0;

virtual RTI::EventRetractionHandle deleteObjectInstance(
    RTI::ObjectHandle theObject,
    const RTI::FedTime& theTime,
    const char* theTag) = 0;

virtual void deleteObjectInstance(
    RTI::ObjectHandle theObject,
    const char* theTag) = 0;

virtual void localDeleteObjectInstance(
    RTI::ObjectHandle theObject) = 0;

virtual void changeAttributeTransportationType (
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes,
    RTI::TransportationHandle theType) = 0;

virtual void changeInteractionTransportationType (
    RTI::InteractionClassHandle theClass,
    RTI::TransportationHandle theType) = 0;

virtual void requestObjectAttributeValueUpdate (
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes) = 0;

virtual void requestClassAttributeValueUpdate (
    RTI::ObjectClassHandle theClass,
    const RTI::AttributeHandleSet& theAttributes) = 0;

virtual void unconditionalAttributeOwnershipDivestiture(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes) = 0;

virtual void negotiatedAttributeOwnershipDivestiture(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes,
    const char* theTag) = 0;

virtual void attributeOwnershipAcquisition(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& desiredAttributes,
    const char* theTag) = 0;

virtual void attributeOwnershipAcquisitionIfAvailable(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& desiredAttributes) = 0;

virtual RTI::AttributeHandleSet* attributeOwnershipReleaseResponse(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes) = 0;

virtual void cancelNegotiatedAttributeOwnershipDivestiture(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes) = 0;

virtual void cancelAttributeOwnershipAcquisition(
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes) = 0;

```

```
virtual void queryAttributeOwnership(  
    RTI::ObjectHandle theObject,  
    RTI::AttributeHandle theAttribute) = 0;  
  
virtual RTI::Boolean isAttributeOwnedByFederate(  
    RTI::ObjectHandle theObject,  
    RTI::AttributeHandle theAttribute) = 0;  
  
virtual void enableTimeRegulation (  
    const RTI::FedTime& theFederateTime,  
    const RTI::FedTime& theLookahead) = 0;  
  
virtual void disableTimeRegulation() = 0;  
  
virtual void enableTimeConstrained() = 0;  
  
virtual void disableTimeConstrained() = 0;  
  
virtual void timeAdvanceRequest(  
    const RTI::FedTime& theTime) = 0;  
  
virtual void timeAdvanceRequestAvailable (  
    const RTI::FedTime& theTime) = 0;  
  
virtual void nextEventRequest(  
    const RTI::FedTime& theTime) = 0;  
  
virtual void nextEventRequestAvailable (  
    const RTI::FedTime& theTime) = 0;  
  
virtual void flushQueueRequest (  
    const RTI::FedTime& theTime) = 0;  
  
virtual void enableAsynchronousDelivery() = 0;  
  
virtual void disableAsynchronousDelivery() = 0;  
  
virtual void queryLBTS(RTI::FedTime& theTime) = 0;  
  
virtual void queryFederateTime(RTI::FedTime& theTime) = 0;  
  
virtual void queryMinNextEventTime (  
    RTI::FedTime& theTime) = 0;  
  
virtual void modifyLookahead (  
    const RTI::FedTime& theLookahead) = 0;  
  
virtual void queryLookahead (  
    RTI::FedTime& theTime) = 0;  
  
virtual void retract(  
    RTI::EventRetractionHandle theHandle) = 0;  
  
virtual void changeAttributeOrderType(  
    RTI::ObjectHandle theObject,  
    const RTI::AttributeHandleSet& theAttributes,  
    RTI::OrderingHandle theType) = 0;  
  
virtual void changeInteractionOrderType(  
    RTI::InteractionClassHandle theClass,  
    RTI::OrderingHandle theType) = 0;  
  
virtual void createRegion (  
    RTI::SpaceHandle theSpace,  
    RTI::ULong numberOfExtents) = 0;  
  
virtual void notifyAboutRegionModification (  

```



```

    RTI::Region &theRegion) = 0;

virtual void deleteRegion(RTI::Region *theRegion) = 0;

virtual RTI::ObjectHandle registerObjectInstanceWithRegion(
    RTI::ObjectClassHandle theClass,
    const char* theObject,
    RTI::AttributeHandle theAttributes[],
    RTI::Region* theRegions[],
    RTI::ULong theNumberOfHandles) = 0;

virtual RTI::ObjectHandle registerObjectInstanceWithRegion(
    RTI::ObjectClassHandle theClass,
    RTI::AttributeHandle theAttributes[],
    RTI::Region* theRegions[],
    RTI::ULong theNumberOfHandles) = 0;

virtual void associateRegionForUpdates (
    RTI::Region& theRegion,
    RTI::ObjectHandle theObject,
    const RTI::AttributeHandleSet& theAttributes) = 0;

virtual void unassociateRegionForUpdates(
    RTI::Region& theRegion,
    RTI::ObjectHandle theObject) = 0;

virtual void subscribeObjectClassAttributesWithRegion(
    RTI::ObjectClassHandle theClass,
    RTI::Region& theRegion,
    const RTI::AttributeHandleSet &attributeList,
    RTI::Boolean active = RTI::RTI_TRUE) = 0;

virtual void unsubscribeObjectClassWithRegion(
    RTI::ObjectClassHandle theClass,
    RTI::Region& theRegion) = 0;

virtual void subscribeInteractionClassWithRegion (
    RTI::InteractionClassHandle theClass,
    RTI::Region& theRegion,
    RTI::Boolean active = RTI::RTI_TRUE) = 0;

virtual void unsubscribeInteractionClassWithRegion (
    RTI::InteractionClassHandle theClass,
    RTI::Region& theRegion) = 0;

virtual RTI::EventRetractionHandle sendInteractionWithRegion(
    RTI::InteractionClassHandle theInteraction,
    const RTI::ParameterHandleValuePairSet &theParameters,
    const RTI::FedTime& theTime,
    const char* theTag,
    const RTI::Region& theRegion) = 0;

virtual void sendInteractionWithRegion(
    RTI::InteractionClassHandle theInteraction,
    const RTI::ParameterHandleValuePairSet &theParameters,
    const char* theTag,
    const RTI::Region& theRegion) = 0;

virtual void requestClassAttributeValueUpdateWithRegion(
    RTI::ObjectClassHandle theClass,
    const RTI::AttributeHandleSet& theAttributes,
    const RTI::Region& theRegion) = 0;

virtual RTI::ObjectClassHandle getObjectClassHandle (
    const char* theName) = 0;

virtual char* getObjectClassName (

```

```
RTI::ObjectClassHandle theHandle) = 0;

virtual RTI::AttributeHandle getAttributeHandle (
    const char* theName,
    RTI::ObjectClassHandle whichClass) = 0;

virtual char* getAttributeName (
    RTI::AttributeHandle theHandle,
    RTI::ObjectClassHandle whichClass) = 0;

virtual RTI::InteractionClassHandle getInteractionClassHandle (
    const char* theName) = 0;

virtual char* getInteractionClassName (
    RTI::InteractionClassHandle theHandle) = 0;

virtual RTI::ParameterHandle getParameterHandle (
    const char* theName,
    RTI::InteractionClassHandle whichClass) = 0;

virtual char* getParameterName (
    RTI::ParameterHandle theHandle,
    RTI::InteractionClassHandle whichClass) = 0;

virtual RTI::ObjectHandle getObjectInstanceHandle(
    const char* theName) = 0;

virtual char* getObjectInstanceName(
    RTI::ObjectHandle theHandle) = 0;

virtual RTI::SpaceHandle getRoutingSpaceHandle (
    const char* theName) = 0;

virtual const char* getRoutingSpaceName (
    const RTI::SpaceHandle theHandle) = 0;

virtual RTI::DimensionHandle getDimensionHandle(
    const char* theName,
    RTI::SpaceHandle whichSpace) = 0;

virtual char* getDimensionName(
    RTI::DimensionHandle theHandle,
    RTI::SpaceHandle whichSpace) = 0;

virtual RTI::SpaceHandle
    getAttributeRoutingSpaceHandle(
        RTI::AttributeHandle theHandle,
        RTI::ObjectClassHandle whichClass) = 0;

virtual RTI::ObjectClassHandle getObjectClass (
    RTI::ObjectHandle theObject) = 0;

virtual RTI::SpaceHandle getInteractionRoutingSpaceHandle(
    RTI::InteractionClassHandle theHandle) = 0;

virtual RTI::TransportationHandle getTransportationHandle(
    const char *theName) = 0;

virtual char* getTransportationName(
    RTI::TransportationHandle theHandle) = 0;

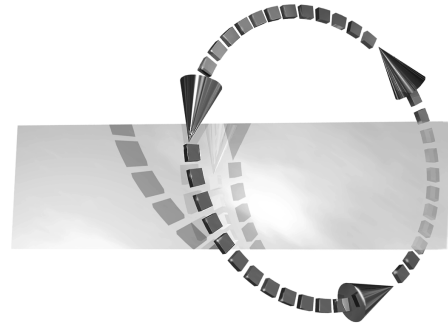
virtual RTI::OrderingHandle getOrderingHandle(
    const char *theName) = 0;

virtual char* getOrderingName(
    RTI::OrderingHandle theHandle) = 0;
```

```
virtual void enableClassRelevanceAdvisorySwitch() = 0;
virtual void disableClassRelevanceAdvisorySwitch() = 0;
virtual void enableAttributeRelevanceAdvisorySwitch() = 0;
virtual void disableAttributeRelevanceAdvisorySwitch() = 0;
virtual void enableAttributeScopeAdvisorySwitch() = 0;
virtual void disableAttributeScopeAdvisorySwitch() = 0;
virtual void enableInteractionRelevanceAdvisorySwitch() = 0;
virtual void disableInteractionRelevanceAdvisorySwitch() = 0;
virtual RTI::Boolean tick() = 0;

virtual RTI::Boolean tick(
    RTI::TickTime minimum,
    RTI::TickTime maximum) = 0;

virtual RTI::RegionToken getRegionToken(RTI::Region*) = 0;
virtual RTI::Region* getRegion(RTI::RegionToken) = 0;
};
#endif // DtVrIRtiAmbassador_H_
#endif // DtRTI103
```

Glossary

For additional terms and definitions, see the HLA glossary at:
<http://hla.dmsomil/hla/general/hlagloss.html>

absolute dead reckoning

A method of [dead reckoning](#) in which the VR-Link uses the time stamps contained within state updates if the sender is using absolute time stamping. Contrast with [relative dead reckoning](#).

aggregate

The combination of individual entities, aggregates, or both into a singular entity. For example, an organizational group such as a platoon.

articulated part

A part of an entity that is capable of movement relative to the entity, such as a turret or gun.

body coordinate frame

A coordinate framework centered on an entity. When the eyepoint moves with respect to an entity, we describe it as movement in a body coordinate frame.

channel

A channel is a particular [view control messages](#) when you are displaying multiple views of a simulation.

dead reckoning

A process by which the VR-Link calculates the expected location of an entity during periods in which it does not receive state updates.

Defense Modeling and Simulation Office

The executive secretariat for the Executive Council on Modeling and Simulation (EXCIMS), which provides a full-time focal point for information concerning Department of Defense modeling and simulation (M&S) activities. Currently the DMSO promulgates M&S policy, initiatives, and guidance to promote cooperation among DoD components to maximize efficiency and effectiveness.

DIS

[Distributed Interactive Simulation.](#)

Distributed Interactive Simulation

The DIS protocol is a set of standards that govern how participating applications share information about the virtual world. The protocol specifies a set of packets, called protocol data units (PDUs), that communicate this information. Each PDU identifies the sender and contains other information depending on the PDU type. The DIS protocol also specifies when and how frequently PDUs are sent.

The DIS protocol has been superseded by the [High-Level Architecture](#) protocol for use by the Department of Defense.

DMSO

[Defense Modeling and Simulation Office.](#)

entity

An element in a simulation, such as a vehicle or a person, that is represented in the simulation through the issuance of state messages.

entity-type

A list of seven components as specified by the DIS protocol and RPR FOM. If none of the seven components contains a wildcard (-1) value, then the entity type refers to a specific, narrowly-defined entity. If some components contain a wildcard, then the entity type refers to a class of entities. A larger number of wildcards indicates a broader, more general class.

The components of an entity-type are:

- ♦ Entity kind
- ♦ Domain
- ♦ Country
- ♦ Category
- ♦ Subcategory
- ♦ Specific
- ♦ Extra

Euler angles

A set of three angles used to describe the orientation of an entity as a set of three successive rotations about three different orthogonal axes (x, y, and z). These angles specify successive rotations needed to transform from the world coordinate system to the entity coordinate system.

exercise

The DIS term for a one or more interacting simulation applications. Compare to [federation execution](#) in HLA.

exercise ID

A numeric identification for a DIS simulation exercise.

FED file

Federation Execution Data file. The Federation Execution Data is the subset of [FOM](#) data needed and read by the [RTI](#). VR-Link also reads the FED file.

federate

A connection to the [RTI](#). Typically a single simulation application can be thought of as a federate.

federation

A group of HLA federates capable of playing in the same [federation execution](#).

Federation Object Model

Defines the data content of a federation execution.

federation execution

The federation execution represents the actual operation, over time, of a subset of the federates and the RTI initialization data taken from a particular federation. It is the step where the executable code is run to conduct the exercise and produce the data for the measures of effectiveness for the federation execution. A federation execution is the logical equivalent of a DIS simulation exercise.

FOM

[Federation Object Model](#).

guise

An alternative entity type used to display an object depending on the force ID. For example, a tank could look like an M1A1 to friendly forces and a T72 to hostile forces.

heartbeat

In DIS, the frequency with which current PDUs are sent to the network regardless of whether or not the entity's state has changed.

High-Level Architecture

The High Level Architecture (HLA) for simulations is a DOD-wide initiative to provide an architecture to support interoperability and reuse of simulations. The HLA supersedes DIS for the Department of Defense.

HLA

[High-Level Architecture](#).

interaction

A [message](#) describing a simulation event. An HLA interaction describes an event, it does not update an object's state.

MÄK Technologies Lisp

An adaptation of the Lisp language used in configuration files for MÄK Technologies products.

message

A general term used to refer to [DIS PDUs](#) and [HLA interactions](#) and state updates.

MTL

[MÄK Technologies Lisp](#).

PDU

[Protocol Data Unit](#).

Protocol Data Unit

A unit of data that is passed on a network between DIS simulation applications.

Realtime Platform Reference FOM

An [HLA](#) reference [FOM](#) based on the [DIS](#) protocol.

reflected entity list

A list of entities simulated by remote applications (federates in HLA) and about which the local simulation has received information over the network.

reference FOM

A [FOM](#) designed to be used as a whole, or with modification, by a wide variety of [federation executions](#).

relative dead reckoning

A method of dead reckoning in which the VR-Link approximates the location of an object based on the local time of receipt of the last state update message.

RTI Initialization Data (RID)

The data required by the RTI for operation. The required data come from two distinct sources, the [Federation Object Model](#) product, and the Federation Execution Details.

RPR FOM

[Realtime Platform Reference FOM.](#)

RTI

[Run-Time Infrastructure.](#)

Run-Time Infrastructure

A library and other supporting software that implements the HLA interface specification. All federates communicate with one another in an HLA environment through RTI functions.

state

The current status of an object, including location, direction of movement, extent of damage, and so on.

topographic coordinate frame

A coordinate frame in the context of the terrain. When the eyepoint moves with respect to the terrain, we describe it as movement in a topographic coordinate frame.

UDP port

A network channel through which the VR-Link sends and receives data for DIS exercises.

Universal Transverse Mercator

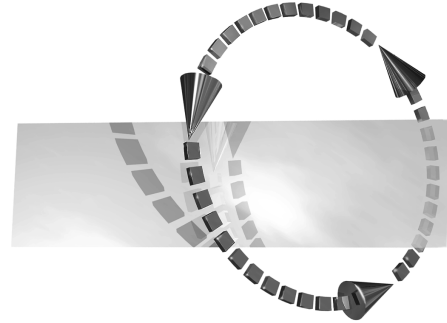
In general, a non-cartesian coordinate system in which the X, Y, and Z correspond to easting (nearly east), northing (nearly north), and height above an ellipsoid that approximates the surface of the earth. When in UTM mode (the default), the MÄK VR-Link uses an offset UTM coordinate system, one in which coordinates are expressed with respect to an origin located at the latitude and longitude specified in a configuration file. Typically, this will be the location of the origin of your terrain database.

UTM coordinates

[Universal Transverse Mercator](#) coordinates.

view control messages

A set of programmatic messages that let you control the VR-Link remotely.



Index

A

- abort behavior, modifying 7-23
- absNotifyObs.h* header file B-2
- acceleration() function 4-18
- ackInter.h* header file 9-2
- ackPdu.h* header file B-3
- ackRPdu.h* header file B-5
- actionPdu.h* header file B-7
- actReqInter.h* header file 9-2
- actReqRPdu.h* header file B-9
- actRespRPdu.h* header file B-11
- add() function 7-19
- addAfter() function 7-19
- addBefore() function 7-19
- addCallback() function 3-10, 4-9, 5-3
 - in interaction class 6-17
 - receiving interactions 4-8
 - receiving PDUs 5-3
 - writing DtPdu 5-15
- addCreator() function 5-20, 6-22
 - in DtPduFactory 5-11
 - in HLA 6-13
- addDiscoverObjectCallback() function 6-8
- adding
 - callback functions 3-10, 4-8
 - elements to *DtList* 7-19
- addInteractionCallback() function 6-11
- addInterestInMcastAddr() function 5-24, 5-28
- addInterestInPduKind() function 5-5, 5-25
- addPduCallback() function 5-4
- addPostDrainCallback() function 4-4, 4-6
- addPostReflectCb() function 6-10
- addRemoveObjectCallback() function 6-8
- addRemoveObjectCb() function 6-8
- address
 - broadcast, returning 7-22
 - destination in DIS 5-21
 - IP 7-22
- MÄK Technologies xx
- addStartUpdatesCb() function 6-7
- addStopUpdatesCb() function 6-7
- addToEnd() function 7-19
- addToStart() function 7-19
- Adobe Acrobat xviii
- AF-UNIX socket file 8-11
- aggAggList.h* header file B-13
- aggCbInfo.h* header file B-15
- aggEntDec.h* header file B-17
- aggEntEnc.h* header file B-19
- aggEntList.h* header file B-21
- aggPub.h* header file 9-2, B-23
- aggregateSR.h* header file B-30
- aggStatePdu.h* header file B-24
- ahvps() function 6-10
- allHlaObjects() function 6-5, 6-9
- allocating
 - memory for PDU 5-16, 5-18
- alt() function 7-8
- angles() function 4-29
- antPatRec.h* header file B-34
- appearance, setting for entity 4-13
- application
 - buffgrep 8-21
 - hlaNetdump 8-8
 - included with VR-Link 8-1
 - nethex 8-9
 - sample, source code 8-2
- application number
 - DIS 4-3
- application number in DIS 4-17
- approximator.h* header file B-35
- apSpec.h* header file B-37
- arealObjStPdu.h* header file B-38
- argument
 - destAddr** 5-24
 - DtObjectId** 4-12
- articulated parts 4-28

- setting values 4-30
- `artParamVal()` function 4-31
- `artPartFromPartNum()` function 4-29
- `artPartFromPartType()` function 4-29
- artPart.h* header file B-41
- `artPartList()` function 4-28, 4-30
- artPartList.h* header file B-44
- artPartThresh.h* header file B-48
- attPartList.h* header file B-51
- attribute updates 5-29, 6-4
 - receiving 6-10
- `AttributeHandleValuePairSet()` function 6-10
- `attributesNeededByFederation()` function 6-6

B

- background
 - packet server 8-19
- baseEmbedDec.h* header file B-54
- baseEmbedEnc.h* header file B-56
- baseEmitttrDec.h* header file B-58
- baseEmitttrEnc.h* header file B-60
- baseEntDec.h* header file B-62
- baseEntEnc.h* header file B-64
- baseEntitySR.h* header file B-67
- baseExConn.h* header file B-71
- baseHashKey.h* header file B-73
- baseMsg.h* header file B-74
- baseReflAgg.h* header file B-76
- baseReflEnt.h* header file B-77
- beamAntRec.h* header file B-78
- beamCbInfo.h* header file B-80
- bin* directories 8-2
- blank PDU 5-8
- body coordinate system 7-5
- `bodyToGeoc()` function 4-18, 4-20, 7-15
- `bodyToLocal()` function 4-27
- broadcast address
 - as default PDU destination 2-13, 4-3, 5-21
 - returning 7-22
- broadcast packets 5-21
- buffer
 - argument to PDU constructor 5-9 to 5-11
 - draining 8-19
 - external, in PDU 5-11
- buffrep example application 8-6, 8-21
- bundling 5-25 to 5-26
- byte swapping 5-14 to 5-15, 5-26, 6-15

C

- C language 3-18
- callback 5-3
 - and PDU kind list 5-25
 - calling sequence of 4-9, 5-4
 - example 3-6
 - execution of 4-4
 - for *DtPdu* subclass 5-15
 - in HLA 6-6
 - incoming PDU for 5-3
 - post drain 4-5
 - receiving interactions 4-8
 - registering and unregistering
 - HLA 6-7 to 6-8
 - protocol independent 3-10
 - registering on entity state PDUs 5-29
 - registering with PDU kind 5-14
 - RTI 4-22
- Cartesian coordinate system 4-25, 4-27, 7-7
- cartView.h* header file B-82
- `changeSize()` function 5-18
- class
 - creating interaction 6-14
 - DtAcknowledgePdu* A-2
 - DtActionRequestPdu* A-4
 - DtActionResponsePdu* A-6
 - DtApproximator* 4-20
 - DtArtPart* 4-28, 4-30
 - DtArtPartList* 4-28
 - DtArtPartSpec* 4-30
 - DtBaseExerciseConn* 4-2
 - DtBaseReflectedEntity* 4-16 to 4-17
 - DtBoolean* 4-14
 - DtCartesianView* 4-25, 4-27
 - DtClientSocket* 5-5, 5-21, 5-24
 - configuring 5-23
 - DtCollisionInteraction* 4-7
 - DtCollisionPdu* 4-7, A-8
 - DtCommentPdu* A-10
 - DtConstVector* 7-3
 - DtCoordTransform* 7-10
 - DtCreateEntityPdu* A-12
 - DtDataPdu* A-13
 - DtDataQueryPdu* A-15
 - DtDcm* 4-27, 7-2 to 7-3, 7-6 to 7-7
 - DtDeadReckoner* 4-20
 - DtDesignatorPdu* A-17
 - DtDetonationInteraction* 4-7
 - DtDetonationPdu* 4-7, A-19

-
- DtDisPduFactory* 5-11
 - DtEmissionPdu* A-22
 - DtEntityIdentifier* 4-5, 4-17
 - DtEntityPublisher*
 - articulated part list 4-30
 - entity state 4-13
 - example 4-12
 - in HLA 6-4
 - thresholds 4-15
 - tick function 4-14
 - time management 7-17
 - use of ESR 4-20
 - DtEntityStatePdu* 4-7, 5-14, A-27
 - DtEntityStateRepository*
 - articulated parts 4-30
 - DIS 5-29
 - entity state 4-13
 - inspecting state 4-17
 - setting state information 4-14
 - DtEntityType* 4-11, 6-4
 - DtEventReportPdu* A-32
 - DtExerciseConn* 4-3
 - basic description 4-2
 - creating 4-2
 - DIS constructors 5-21
 - drainInput() function 4-16
 - filtering PDUs 5-24
 - functions 4-4
 - interactions class, and 6-13
 - multicast subscription 5-24
 - new class, and 5-20
 - packet server, with 5-5
 - registering class with 6-22
 - RTI and 6-2
 - sending interactions 4-7
 - telling about derived class 6-3
 - time management 7-17
 - DtFilterTests* 5-28
 - DtFireInteraction* 4-7, 6-17
 - DtFirePdu* 4-7, A-34
 - DtGeodeticCoord* 7-8
 - DtHlaObject* 6-4, 6-9
 - DtInteraction*
 - callbacks 4-8
 - draining input 4-4
 - HLA 6-12
 - interaction classes 4-7
 - print functions 6-12
 - publishing interactions 6-11
 - subclassing 6-17
 - user-defined 6-14
 - DtInteractionFactory* 6-13, 6-22
 - DtLaserPdu* A-36
 - DtList* 7-19
 - DtListItem* 7-19
 - DtLocalHlaObjectManager* 6-5
 - DtMapDatum* 7-9
 - DtMessagePdu* A-38
 - DtNetInt32* 5-14 to 5-15
 - DtNetSocket* 5-21 to 5-22
 - DtObjectId* 4-5, 4-17
 - DtPdu* 4-4, 4-7, 5-2 to 5-4, 5-7
 - constructor 5-12
 - deriving classes from 5-14
 - external buffers 5-11
 - notifying VR-Link about 5-20
 - DtPduFactory* 5-9 to 5-10, 5-20
 - DtPduWithData* A-44
 - DtReceiverPdu* A-51
 - DtReflectedEntity* 6-9
 - entity list 4-15
 - HLA 6-4
 - incoming PDUs 5-29
 - looking up by ID 4-17
 - subclassing 4-23
 - DtReflectedEntityList* 4-15
 - and time 7-17
 - HLA 6-9
 - notification of entities coming or going 4-21
 - subscriptions 6-7
 - timeouts 4-23
 - with PDUs 5-29
 - DtReflectedHlaObjectManager* 6-9
 - DtRemoveEntityPdu* A-52
 - DtRepairCompletePdu* A-53
 - DtRepairResponsePdu* A-54
 - DtResupplyCancelPdu* A-55
 - DtResupplyOfferPdu* A-56
 - DtResupplyReceivedPdu* A-58
 - DtSetDataPdu* A-60
 - DtSignalPdu* A-62
 - DtSimanPdu* A-64
 - DtSocket* 5-5, 5-21 to 5-22, 5-24 to 5-26
 - DtStartResumePdu* A-66
 - DtStateMsg* 6-10
 - DtStopFreezePdu* A-68
 - DtTaitBryan* 4-13, 4-18, 7-2, 7-5, 7-7
 - DtTopoView* 4-25 to 4-26
 - DtTransmitterPdu* A-70
 - DtUnknownInteraction* 6-13 to 6-14

- receiving interactions 6-15
- sending interactions 6-14
- DtUnknownPdu* 5-13
- DtUtmCoord* 4-27, 7-12
- DtUtmView* 4-25
- DtVector* 4-27, 7-2 to 7-3, 7-7
 - topographic coordinate 7-10
- DtViewControlPdu* A-75
- DtVrlFederateAmbassador* 6-2 to 6-3
- FederateAmbassadorRTI*
 - FederateAmbassador 6-2
- listed alphabetically 9-4
- listed by header file 9-16
- ObjectClassHandle* 6-4
- overview 3-2
- registering with *DtExerciseConn* 6-22
- RTIambassador* 6-2
- subscribing to 6-7
- WeaponFire* 6-17
- classHandle()* function 6-12, 6-17
- cleanupFromUpdating()* function 6-5
- ClientSocket.h* header file B-84
- cliServer.h* header file B-87
- collideInter.h* file 9-2
- collidePdu.h* header file B-87
- collisElasPdu.h* header file B-89
- commentPdu.h* header file B-92
- commentRPdu.h* header file B-93
- compiling 3-13, 3-15
 - conditional 4-3
- conceptual overview of VR-Link 3-2
- conditional compiling 4-3
- configRTI* script 2-11
- configRTI* script 4-2
- configuring
 - DtNetSocket* 5-22
 - fl8* application 8-4
 - network connection for DIS 5-21
 - packet server 8-12
 - system for DMSO RTI 2-11
 - system for MÄK RT 2-10
- connecting to network
 - DIS 5-21
- controlling
 - packet server 8-17
- conventions
 - manual xxi
- converting
 - coordinates 7-7, 7-15
 - Euler angles between geocentric and topographic 7-11
 - geocentric to topographic 7-10
 - orientation 7-7
 - vectors 7-7
- coordinate
 - conversion 4-25, 7-7
 - conversion functions 7-15
 - geocentric 4-13, 4-18
 - getting and setting geocentric 7-8
 - right-hand geocentric 7-7
 - systems 3-12
 - views 4-25
- coordinate system
 - body 7-5
 - Cartesian 4-25, 4-27, 7-7
 - converting between geocentric and topographic 7-10
 - geocentric 7-5, 7-8
 - topographic 4-25 to 4-26, 7-5, 7-10
 - UTM 4-25, 7-12
- coordTrans()* function 7-11
- copying PDUs 5-12
- count()* function 4-16
- create()* function
 - interaction 6-13, 6-17
 - PDU 5-15, 5-17
 - PDU creator 5-11
- createEntRPdu.h* header file B-95
- createInteraction()* function 6-13
- createPdu()* function 5-10 to 5-11
- createPdu.h* header file B-97
- creating
 - exercise connection 4-2
 - exercise connection for DIS 4-3
 - interaction classes 6-14
 - interactions 6-17
 - linked lists 7-19
 - PDUs 5-13
 - reflected entity lists 4-16
 - socket for DIS 5-22

D

daemon controller. *See* vdc

- dAggDec.h* header file B-98
- dAggEnc.h* header file B-99
- dAggPub.h* header file B-101

- damageState()* function 4-18
- data()* function 7-19

- dataPdu.h* header file B-102
- dataQueryRPdu.h* header file B-103
- dataRPdu.h* header file B-105
- datumIdType.h* header file B-107
- datumMap.h* header file B-108
- datumRecSet.h* header file B-110
- dcm.h* header file B-116
- dead reckoning 4-20
 - algorithm, specifying 4-12
 - disabling 3-11, 4-29
 - and DtEntityStateRepository 4-20
 - and tick function 4-15
 - and VR-Link simulation time 7-17
 - when processed 3-11
- deadReckoner.h* header file B-117
- decFactory.h* header file B-120
- default broadcast address 4-3
- defining
 - interaction 6-17
- deleteBytes() function 5-18
- deleteObject RTI service 6-5
- dEntIdHashLst.h* header file B-122
- dEntityDec.h* header file B-123
- dEntityEnc.h* header file B-124
- dEntityPub.h* header file B-126
- deriving
 - interactions 6-17
- deriving classes from 5-14
- designatePdu.h* header file B-127
- designatorDec.h* header file B-129
- designatorEnc.h* header file B-131
- designatorSR.h* header file B-133
- designCbInfo.h* header file B-135
- destAddr** argument 5-24
- destination address
 - default 5-26
 - specifying in DIS 5-3, 5-21
- detecting new entities 4-16
- detonateInter.h* header file 9-2
- detonatePdu.h* header file B-137
- detonation
 - f18* 8-5
- device name, returning IP address of 7-22
- dExerciseConn.h* header file B-140
- diagnostic information
 - printing 7-21
 - setting notify level 7-12
- diagnostics
 - vdc 8-11
- Dial-a-Sim xix
- dInteraction.h* header file B-143
- direction cosine matrix 7-2, 7-5
- directory
 - bin* 8-2
 - flexlm* 2-7
- directory tree, VR-Link 2-4
- directRotRate.h* header file B-144
- dirRtiAmb.h* header file B-145
- DIS
 - configuring network connection 5-21
 - creating exercise connection 4-3
 - DIS-specific functionality 5-1
 - executables 2-5
 - network configuration 2-13
 - PDUs 5-2, A-1
 - UDP port 4-3
 - versions 1-3
- disableFiltering() function 5-5, 5-25
- disabling
 - dead reckoning 4-29
- discovering
 - HLA objects 6-8
- discoverObject RTI service 4-22, 6-2, 6-7 to 6-8
- disEnums.h* header file B-155
- disPduFactory.h* header file B-194
- displaying HLA messages and PDUs 8-6, 8-8 to 8-9
- DMSO 1-3
- DMSO RTI 2-11
 - configuring 2-11
- documentation, location of 2-5
- draining
 - buffers 8-19
- drainInput() function 5-4
 - and readUntil() 5-6
 - example 3-6
 - in HLA 4-10
 - protocol independent 4-4
 - receiving PDUs 5-3
 - timeouts 4-23
 - with reflected entities 4-16
- dReflAggList.h* header file B-195
- dReflectedAgg.h* header file B-198
- dReflectedEnt.h* header file B-200
- dReflEntList.h* header file B-202
- dropMulticastGroup() function 5-28
- dscvObjCbInfo.h* header file B-205
- dStateMsg.h* header file B-207
- DtAbort() function 7-21, 7-23
- DtAbsRealTime() function 7-17

- DtAcknowledgePdu* class A-2
- DtActionRequestPdu* class A-4
- DtActionResponsePdu* class A-6
- DtApproximator* class 4-20
- DtArtParamType* 4-28, 4-30
- DtArtPart* class 4-28, 4-30
- DtArtPartList* class 4-28
- DtArtPartSpec* class 4-30
- DtArtPartType* 4-28, 4-30
- DtAttachedToBase* 4-29 to 4-30
- DtBaseExerciseConn* class 4-2
- DtBaseReflectedEntity* class 4-16 to 4-17
- DtBecomeNonBlocking()* function 5-27
- DtBindToDestAddr* flag 5-23
- DtBodyToRef_to_Euler()* function 7-6, 7-15
- DtBoolean* class 4-14
- DtBufferPtr* 5-11
- DtCartesianView* class 4-25, 4-27
- DtClientSocket* class
 - configuring 5-23
- DtClientSocket* class 5-5, 5-21, 5-24
- DtLoadLispFile()* function 8-4
- DtCollisionInteraction* class 4-7
- DtCollisionPdu* class 4-7, A-8
- DtCommentPdu* class A-10
- DtConstDcm* 7-3
- DtConstVector* 7-3
- DtConstVector* class 7-3
- DtCoordTransform* class 7-10
- DtCreateEntityPdu* class A-12
- DtDataPdu* class A-13
- DtDataQueryPdu* class A-15
- DtDcm* class 4-27, 7-2 to 7-3, 7-6 to 7-7
- DtDcm&* 7-3
- DtDcmRef* 7-3
- DtDcmVelMul()* function 7-15
- DtDeadReckoner* class 4-20
- DtDegMinSec* 7-12
- DtDesignatorPdu* class A-17
- DtDetonationInteraction* class 4-7
- DtDetonationPdu* class 4-7, A-19
- DtDisPduFactory* class 5-11
- DtDISVERS* 5-9, 5-11
- DtEmissionPdu* class A-22
- DtEntityIdentifier* class 4-5, 4-17
- DtEntityPublisher* class
 - articulated part list 4-30
 - entity state 4-13
 - example 4-12
 - in HLA 6-4
 - thresholds 4-15
 - tick function 4-14
 - time management 7-17
 - use of ESR 4-20
- DtEntityPublisher::tick()* function 4-13
- DtEntityStatePdu* class 4-7, 5-14, A-27
- DtEntityStateRepository* class
 - articulated parts 4-30
 - DIS 5-29
 - entity state 4-13
 - inspecting state 4-17
 - setting state information 4-14
- DtEntityType* class 4-11, 6-4
- DtEuler_to_BodyToRef()* function 7-6
- DtEuler_to_RefToBody()* function 7-6
- DtEulerToEuler()* function 7-15
- DtEventReportPdu* class A-32
- DtExerciseConn* class
 - basic description 4-2
 - creating 4-2
 - DIS constructors 5-21
 - drainInput()* function 4-16
 - filtering PDUs 5-24
 - functions 4-4
 - interaction class, and 6-13
 - multicast subscription 5-24
 - new class, and 5-20
 - packet server, with 5-5
 - registering and unregistering functions with 4-4
 - registering class with 6-22
 - RTI and 6-2
 - See also* exercise connection
 - sending interactions 4-7
 - telling about derived class 6-3
 - time management 7-17
- DtExerciseConn* class 4-3
- DtFatalError()* function 7-21
- DtFatalPerror()* function 7-21
- DtFedAmbCreator()* function 6-3
- DtFilterTests* class 5-28
- DtFireInteraction* class 4-7, 6-17
- DtFirePdu* class 4-7, A-34
- DtFirstHostInetAddr()* function 7-22
- DtGeocToGeod()* function 7-9
- DtGeocToTopoTransform()* function 7-10
- DtGeocToUtm()* function 7-13
- DtGeodeticCoord* class 7-8
- DtGeodToUtm()* function 7-13
- DtGetElapsedRealTime()* function 7-17
- DtHLA* compile line option 4-2, 4-11, 4-16

-
- DtHlaObject* class 6-4, 6-9
 - DtInetAddr* 7-22
 - DtInetAddrOfDevice()* function 7-22
 - DtInetAddrString()* function 7-22
 - DtInetBroadcastOfDevice()* function 7-22
 - DtInfo()* function 7-21
 - DtInteraction* class
 - draining input 4-4
 - HLA 6-12
 - interaction classes 4-7
 - print functions 6-12
 - publishing interactions 6-11
 - registering callbacks 4-8
 - subclassing 6-17
 - user-defined 6-14
 - DtInteractionFactory* class 6-13, 6-22
 - DtIsMulticastAddr()* function 7-22
 - DtLaserPdu* class A-36
 - DtLatLon_to_GeocToTopo()* function 7-15
 - DtLatLon_to_TopoToGeoc()* function 7-15
 - DtList* class 7-19
 - DtListItem* class 7-19
 - DtLocalHlaObjectManager* class 6-5
 - DtMapDatum* class 7-9
 - DtMessagePdu* class A-38
 - DtNetInt32* class 5-14 to 5-15
 - DtNetmask* environment variable 2-14
 - DtNetMaskOfDevice()* function 7-22
 - DtNetSocket* class 5-21 to 5-22
 - DtNotifyLevel* variable 7-21
 - DtNullArtPartSpec* macro 4-31
 - DtObjectId* argument 4-12
 - DtObjectId* class 4-5, 4-17
 - DtPacketServerIsAlive()* function 5-23
 - DtPdu* class
 - constructor 5-12
 - deriving 5-15
 - external buffers 5-11
 - notifying VR-Link about 5-20
 - DtPdu* class 4-4, 4-7, 5-2 to 5-4, 5-7, 5-14
 - DtPduFactory* class 5-9 to 5-10, 5-20
 - DtPduWithData* class A-44
 - DtProtocolVersionToRecvMax* variable 5-11
 - DtProtocolVersionToRecvMin* variable 5-11
 - DtProtocolVersionToSend* variable 5-9
 - DtQuiet* flag 5-22
 - DtRead* flag 5-22
 - DtReceiverPdu* class A-51
 - DtRecv()* function 5-21, 5-25 to 5-27
 - DtReflectedEntity* class
 - entity list 4-15
 - HLA 6-4
 - incoming PDUs 5-29
 - looking up by ID 4-17
 - subclassing 4-23
 - DtReflectedEntityList* class 6-9
 - DtReflectedEntityList* class
 - and time 7-17
 - HLA 6-9
 - notification of entities coming or going 4-21
 - subscriptions 6-7
 - timeouts 4-23
 - with PDUs 5-29
 - DtReflectedEntityList* class 4-15
 - DtReflectedHlaObjectManager* class 6-9
 - DtRefToBody_to_Euler()* function 7-6
 - DtRemoveEntityPdu* class A-52
 - DtRepairCompletePdu* class A-53
 - DtRepairResponsePdu* class A-54
 - DtReportPktsRcvd()* function 5-27
 - DtResupplyCancelPdu* class A-55
 - DtResupplyOfferPdu* class A-56
 - DtResupplyReceivedPdu* class A-58
 - DtSelect()* function 5-27, 7-23
 - DtSend()* function 5-21, 5-25 to 5-26
 - DtSetDataPdu* class A-60
 - DtSetSimTime()* function 4-20, 7-17 to 7-18
 - DtSignalPdu* class A-62
 - DtSimanPdu* class A-64
 - DtSleep()* function 7-23
 - DtSocket* class 5-5, 5-21 to 5-22, 5-24 to 5-26
 - DtStartResumePdu* class A-66
 - DtStateMsg* class 6-10
 - DtStopFreezePdu* class A-68
 - DtStringToInetAddr()* function 7-22
 - DtTaitBryan* class 4-13, 4-18, 7-2, 7-5, 7-7
 - DtTimeInit()* function 7-17
 - DtTopoView* class 4-25 to 4-26
 - DtTransmitterPdu* class A-70
 - DtUnknownInteraction* class
 - receiving interactions 6-15
 - sending interactions 6-14
 - DtUnknownInteraction* class 6-13 to 6-14
 - DtUnknownPdu* class 5-13
 - DtUseMapDatum()* function 7-9
 - DtUtmCoord* class 4-27, 7-12
 - DtUtmInit()* function 4-27, 7-12
 - using 7-12
 - DtUtmToGeoc()* function 7-13
 - DtUtmToGeod()* function 7-13

DtUtmView class 4-25
DtUtmView view 4-27
DtVector class
 topographic coordinate 7-10
DtVector class 4-27, 7-2 to 7-3, 7-7
DtVector& 7-3
DtVectorRef 7-3
DtViewControlPdu class A-75
DtVrlFederateAmbassador class 6-2 to 6-3
DtWarn() function 7-21
DtWarnPerror() function 7-21
DtWrite flag 5-22

E

elecEmitPdu.h header file B-208
elements
 adding to *DtList* 7-19
embedSystemSR.h header file B-212
emissionPdu.h header file B-213
emittSysCbInf.h header file B-216
emitterSysDec.h header file B-218
emitterSysEnc.h header file B-220
emitterSysSR.h header file B-222
emittBeamSR.h header file B-224
enableFiltering() function 5-25
encFactory.h header file B-227
engineSmokeOn() function 4-18
entCbInfo.h header file B-230
entity
 appearance, getting 4-18
 articulated parts 4-28
 counting in reflected entity list 4-16
 detecting new 4-16
 joining or leaving an exercise 4-21
 locally-simulated 4-11
 locating in reflected entity list 4-16
 low-fidelity state 4-15
 management 3-2
 managing 4-11
 number, in DIS 4-17
 and protocol independence 4-11
 publisher 1-2, 4-11
 removing from list 4-21
 setting appearance 4-13
 setting state 4-13
 timing out 4-23
 updates 4-11
entity ID 4-11, 5-29
 getting 4-4, 4-17
 uniqueness 4-5
entity list. *See* reflected entity list
entity state PDU 5-14
 receiving 5-29
entity state repository 3-6, 3-8 to 3-9, 4-13
 in example 3-6, 3-8
 printing contents of 4-14
 setting entity state 4-13
entityAdded() function 4-21 to 4-22, 6-8
entityPub.h header file 9-2
entitySR.h header file B-232
entityStateRep() function 4-13, 4-17
entityType() function 5-2
EntityTypes.h header file B-237
enum2string.h header file B-253
enumerations 4-11, 4-14, 5-7
envDec.h header file B-254
envEnc.h header file B-255
environment variable
 DtNetmask 2-14
 license manager 2-6
 LM_LICENSE_FILE 2-6
 RTI_CONFIG 2-10
enviroTypeRec.h header file B-257
envProcPdu.h header file B-258
error, fatal 7-23
esPdu.h header file B-260
ESR. *See* entity state repository
esUpdatePdu.h header file B-263
Euler angles
 converting between geocentric and
 topographic 7-11
 in dead reckoning 4-20
 representing orientation 7-5
 topographic 7-15
 UTM coordinates 4-27
Euler.h header file B-266
eulerTrans() function 7-11
event ID 4-4
event PDU. *See* interactions
eventId.h header file B-267
eventReptPdu.h header file B-269
eventReptRPdu.h header file B-270
example applications
 and licenses 8-2
 fl8 8-3
 hlaNetdump 8-8
 listen 8-9
 netdump 8-6
 nethex 8-9

- source code for 1-2, 8-2
- talk 8-9
- executables
 - DIS 2-5
 - HLA 2-5
- execution name 4-2
- exercise
 - joining or leaving 4-21
- exercise connection 1-2, 3-2, 4-2
 - creating 4-2
 - differences between HLA and DIS 4-2
 - parameters 4-2 to 4-3
 - and protocol 4-2
 - and RTI ambassador 4-3
 - See also* DtExerciseConn
- exercise ID
 - examining 5-21
 - global 5-21
 - in constructor 5-21
 - requirement in DIS 4-3
 - setting in PDU 5-2, 5-21
- exerciseConn.h* header file 9-2
- exerciseId() function 5-7, 5-21, 5-28
- external buffers
 - using in *DtPdu* 5-11
- external mode
 - in packet server 8-12
 - vrlink3di* 8-19

F

- f18
 - configuring 8-4
- f18* 8-3
- fatal error 7-23
- fax
 - MÄK Technologies xx
- features, VR-Link 1-2
- FED file 2-10
- fedAmb() function 6-3
- federate name 4-2
- FederateAmbassador* class 6-2
- federation execution name 4-2
- file
 - AF-UNIX socket 8-11
 - FED 2-10
 - license.dat* 2-6 to 2-7
 - RID 2-10
 - VR-Link.fed* 1-3, 4-2
- file descriptors, reading and writing 5-27

- filtering PDUs 5-5, 5-24, 5-28
- fireInter.h* header file 9-3
- firePdu.h* header file B-273
- first() function 4-16, 7-19
- firstValidParamType() function 4-29
- flag
 - DtBindToDestAddr 5-23
 - DtQuiet 5-22
 - DtRead 5-22
 - DtWrite 5-22
- flamesPresent() function 4-18
- flexlm* directory 2-7
- FLEXlm license manager 2-6
- flush() function 5-25
- FOM support 1-3
- fom.h* header file B-275
- fomMapper.h* header file B-277
- force ID
 - specifying 4-12
- function
 - acceleration() 4-18
 - add() 7-19
 - addAfter() 7-19
 - addBefore() 7-19
 - addCallback() 3-10, 4-9, 5-3
 - in interaction class 6-17
 - receiving interactions 4-8
 - receiving PDUs 5-3
 - writing DtPdu 5-15
 - addCreator() 5-20, 6-22
 - in DtPduFactory 5-11
 - in HLA 6-13
 - addDiscoverObjectCallback() 6-8
 - adding callback 4-8
 - addInteractionCallback() 6-11
 - addInterestInMcastAddr() 5-24, 5-28
 - addInterestInPduKind() 5-5, 5-25
 - addPduCallback() 5-4
 - addPostDrainCallback() 4-4, 4-6
 - addPostReflectCb() 6-10
 - addRemoveObjectCallback() 6-8
 - addRemoveObjectCb() 6-8
 - addStartUpdatesCb() 6-7
 - addStopUpdatesCb() 6-7
 - addToEnd() 7-19
 - addToStart() 7-19
 - ahvps() 6-10
 - allHlaObjects() 6-5, 6-9
 - alt() 7-8
 - angles() 4-29

artParamVal() 4-31
artPartFromPartNum() 4-29
artPartFromPartType() 4-29
artPartList() 4-28, 4-30
AttributeHandleValuePairSet() 6-10
attributesNeededByFederation() 6-6
bodyToGeoc() 4-18, 4-20, 7-15
bodyToLocal() 4-27
changeSize() 5-18
classHandle() 6-12, 6-17
cleanupFromUpdating() 6-5
converting coordinates 7-15
coordTrans() 7-11
count() 4-16
create()
 interaction 6-13, 6-17
 PDU 5-15, 5-17
 PDU creator 5-11
createInteraction() 6-13
createPdu() 5-10 to 5-11
damageState() 4-18
data() 7-19
deleteBytes() 5-18
disableFiltering() 5-5, 5-25
drainInput() 5-4
 and readUntil() 5-6
 example 3-6
 in HLA 4-10
 protocol independent 4-4
 receiving PDUs 5-3
 timeouts 4-23
 with reflected entities 4-16
dropMulticastGroup() 5-28
DtAbort() 7-21, 7-23
DtAbsRealTime() 7-17
DtBecomeNonBlocking() 5-27
DtBodyToRef_to_Euler() 7-6, 7-15
DtLoadLispFile() 8-4
DtDcmVelMul() 7-15
DtEuler_to_BodyToRef() 7-6
DtEuler_to_RefToBody() 7-6
DtEulerToEuler() 7-15
DtFatalError() 7-21
DtFatalPerror() 7-21
DtFedAmbCreator() 6-3
DtFirstHostInetAddr() 7-22
DtGeocToGeod() 7-9
DtGeocToTopoTransform() 7-10
DtGeocToUtm() 7-13
DtGeodToUtm() 7-13
DtGetElapsedRealTime() 7-17
DtInetAddrOfDevice() 7-22
DtInetAddrString() 7-22
DtInetBroadcastOfDevice() 7-22
DtInfo() 7-21
DtIsMulticastAddr() 7-22
DtLatLon_to_GeocToTopo() 7-15
DtLatLon_to_TopoToGeoc() 7-15
DtNetMaskOfDevice() 7-22
DtPacketServerIsAlive() 5-23
DtRecv() 5-21, 5-25 to 5-27
DtRefToBody_to_Euler() 7-6
DtReportPktsRcvd() 5-27
DtSelect() 5-27, 7-23
DtSend() 5-21, 5-25 to 5-26
DtSetSimTime() 4-20, 7-17 to 7-18
DtSleep() 7-23
DtStringToInetAddr() 7-22
DtTimeInit() 7-17
DtUseMapDatum() 7-9
DtUtmInit() 4-27, 7-12
 using 7-12
DtUtmToGeoc() 7-13
DtUtmToGeod() 7-13
DtWarn() 7-21
DtWarnPerror() 7-21
enableFiltering() 5-25
engineSmokeOn() 4-18
entityAdded() 4-21 to 4-22, 6-8
entityStateRep() 4-13, 4-17
entityType() 5-2
eulerTrans() 7-11
exerciseId() 5-7, 5-21, 5-28
fedAmb() 6-3
first() 4-16, 7-19
firstValidParamType() 4-29
flamesPresent() 4-18
flush() 5-25
geoc2Topo() 7-15
geocentric() 7-8, 7-13
geodetic() 7-13
getArtParamVal() 4-28
getGeocentric() 7-8, 7-13
getGeodetic() 7-13
getObjectClassHandle() 6-4
getVal() 4-29
gridDeclaration() 7-14
guessTimeValid() 5-7
hatchState() 4-18
hlaObject() 6-4, 6-6, 6-9

- id() 4-17
- identity() 7-3
- init() 4-30 to 4-31
- initArtParts() 4-31
- initiatePause() 6-3
- initPdu() 5-16, 5-19
- insertBytes() 5-18
- interactionClassHandle() 6-12, 6-17
- interactionFactory() 6-13
- internalGetPduKind() 5-15 to 5-16
- ipAddr() 5-28
- isBundling() 5-25
- isOk() 5-28
- kind() 5-7
- last() 4-16, 7-19
- lastSourceInetAddr() 5-27
- lat() 7-8
- length() 5-7
- listenToMulticastGroup() 5-28
- localObjectManager() 6-5
- location() 4-18, 4-20
- lon() 7-8
- lookup() 4-17
- name() 6-12, 6-17
- netRead() 5-4, 5-25 to 5-26
 - using 5-5
- newReflectedEntity() 4-23, 6-9
- next() 4-16, 7-19
- nextEventId() 4-4
- nextId() 4-4, 4-12
- nextValidParamType() 4-29
- notifyMe() 5-27
- numArtParams() 4-28
- numArtParts() 4-28
- numParameters() 6-12, 6-17
- numParams() 4-29
- objectClassHandle() 6-5
- objectId() 6-10
- orientation() 4-18, 4-20
- packet() 5-11
- packet() 5-8
- ParameterHandleValuePairSet() 6-13 to 6-14, 6-17
- partAttachedTo() 4-29
- partNum() 4-29
- partType() 4-29
- pduFactory() 5-10
- pduKind() 5-28
- phi() 7-5
- phvps() 6-12, 6-17, 6-21
- postDrainCallback() 4-10
- prev() 4-16, 7-19
- print for *DtInteraction* 6-12
- print() 6-17
 - articulated parts 4-29
 - interactions for 6-12
 - PDU 5-7
- printData() 5-7
 - entity state repository 4-19
 - PDU 5-7
 - printing entity state repository 4-14
 - protocol independent 6-12
- printHeader() 5-7, 6-12
- printParams() 6-12
- processEntityState() 5-29
- processPdu() 5-4
- protocolFamily() 5-7
- protocolVersion() 5-7, 5-28
- psi() 7-5
- readAndProcess() 5-4
- readFd() 5-27
- readUntil() 5-6
- receiving 5-26
- reflectedObjectManager() 6-9
- registering and unregistering 4-8
- registering with PDU class in DIS 5-3
- removeAndDelete() 4-21, 6-8 to 6-9
- removeCallback() 4-8, 5-4
 - interactions for 6-17
 - PDU 5-3, 5-15
- removeDiscoverObjectCallback() 6-8
- removePostDrainCallback() 4-4
- removePostReflectCb() 6-10
- removeRemoveObjectCallback() 6-8
- removeRemoveObjectCb() 6-8
- removeStartUpdatesCb() 6-7
- removeStopUpdatesCb() 6-7
- removing callback 4-8
- requestedAttributes() 6-6
- rotationalVelocity() 4-18, 4-25
- RTI
 - ParameterHandleValuePairSet() 6-12
- rtiAmb() 6-2
- send()
 - definition 4-4
 - in HLA 4-8
 - multicast address 5-24
 - obtaining object for RTI 6-12
 - optional address in DIS 5-3
 - publishing interactions 6-11

- sending DIS PDUs 5-26
- sending interactions 6-15
- sending PDUs 5-2
- unknown PDU 5-13
- with user-defined classes 5-14
- sending 5-26
- sendStamped()
 - definition 4-4
 - in example 3-8
 - in HLA 4-8
 - multicast address 5-24
 - obtaining object for RTI 6-12
 - optional address in DIS 5-3
 - publishing interactions 6-11
 - sending DIS PDUs 5-26
 - sending interactions 6-15
 - sending PDUs 5-2
 - unknown PDU 5-13
 - with user-defined classes 5-14
- sendTo() 5-25 to 5-26
- setAlt() 7-8
- setApproximating() 4-29
- setApproximator() 3-11, 4-20
- setArtParamVal() 4-31
- setBundling() 5-25
- setByInverse() 7-11
- setDamageState() 4-13
- setDfltRotationThreshold() 4-15
- setDfltTimeThreshold() 4-15
- setDfltTranslationThreshold() 4-15
- setEngineSmokeOn() 4-13
- setEntityType() 5-2
- setExerciseId() 5-7, 5-21
- setFedAmbCreator() 6-3
- setFlamesPresent() 4-13
- setFromPhyps() 6-12, 6-17, 6-19, 6-21
- setGeocentric() 7-8
- setHatchState() 4-13
- setLat() 7-8
- setLon() 7-8
- setMcastTtl() 5-24, 5-28
- setOffset() 4-27
- setOrientation() 4-13
- setPhi() 7-5
- setPsi() 7-5
- setRotation() 4-27
- setRotationalVelocity() 4-13, 4-25
- setTheta() 7-5
- setTimeoutInterval() 4-23
- setTimeoutProcessing() 4-23

- setTimeStamp() 5-7
- setTimeThreshold() 4-15
- setUnbundling() 5-25
- setupUpdating() 6-5
- setVal() 4-31
- sleep() 7-23
- socket() 5-22
- status() 5-10
- string() 7-2, 7-5
- subtractInterestInMcastAddr() 5-24, 5-28
- subtractInterestInPduKind() 5-25
- theta() 7-5
- tick() 3-11, 4-4, 4-14
- tick(), DtEntityPublisher 4-13
- timeStamp() 5-7
- timeStampType() 5-7
- translations() 4-29
- update() 6-6
- val() 4-29
- vecTrans() 7-11
- velocity() 4-18, 4-20
- wrapNext() 4-16
- wrapPrev() 4-16
- writeFd() 5-27
- zero() 7-3
- functions
 - registering and unregistering with
DtExerciseConn 4-4
 - user-defined 4-8

G

- gedBasFixWing.h* header file B-280
- gedBasRotWing.h* header file B-282
- gedBasSoldier.h* header file B-285
- gedBasVehicle.h* header file B-287
- gedEnhFixWing.h* header file B-289
- gedEnhRotWing.h* header file B-291
- gedEnhSoldier.h* header file B-293
- gedEnhVehicle.h* header file B-295
- gedLogVehicle.h* header file B-297
- geoc2Topo() function 7-15
- geocentric coordinate 4-13, 4-18, 7-5
 - converting to topographic 7-10
 - right-hand 7-7
- geocentric() function 7-8, 7-13
- geodCoord.h* header file B-299
- geodetic coordinates 7-8
- geodetic() function 7-13
- getArtParamVal() function 4-28

getGeocentric() function 7-8, 7-13
 getGeodetic() function 7-13
 getObjectClassHandle() function 6-4
 getting
 appearance information 4-18
 entity ID 4-4, 4-17
 exercise ID 5-7
 geocentric and geodetic coordinates 7-13
 geocentric coordinates 7-8
 interaction information 4-8
 kind 5-7
 length 5-7
 network representation 5-8
 parameter values 4-29
 protocol family 5-7
 protocol version 5-7
 time, space, and position information 4-18
 UTM coordinates 7-13
 getVal() function 4-29
globalObjDes.h header file B-301
globIdHLst.h header file B-302
gridAxiRecFix.h header file B-303
gridAxiRecVar.h header file B-305
gridDataPdu.h header file B-307
gridDataRec.h header file B-311
gridDataRecT0.h header file B-312
gridDataRecT1.h header file B-314
gridDataRecT2.h header file B-316
 gridDeclaration() function 7-14
 guessTimeValid() function 5-7
 guise, specifying 4-12

H

hAckDec.h header file B-318
hAckEnc.h header file B-319
hAckInter.h header file B-320
hActReqDec.h header file B-322
hActReqEnc.h header file B-323
hActReqInter.h header file B-324
hAggPub.h header file B-326
hAppRadSigDec.h header file B-328
hAppRadSigEnc.h header file B-329
hashlist.h header file B-330
 hatchState() function 4-18
hCollideDec.h header file B-332
hCollideEnc.h header file B-334
hCollideInter.h header file B-335
hDbiRadSigDec.h header file B-337
hDbiRadSigEnc.h header file B-338

hDesignatrPub.h header file B-339
hDetDec.h header file B-340
hDetEnc.h header file B-342
hDetInter.h header file B-343
 header
 PDU 5-9
 header file B-1
 fireInter.h 9-3
 reflectedEnt.h 9-3
 stateMsg.h 9-3
 vcInter.h 9-3
 absNotifyObs.h B-2
 ackInter.h 9-2
 ackPdu.h B-3
 ackRPdu.h B-5
 actionPdu.h B-7
 actReqInter.h 9-2
 actReqRPdu.h B-9
 actRespRPdu.h B-11
 aggAggList.h B-13
 aggCbInfo.h B-15
 aggEntDec.h B-17
 aggEntEnc.h B-19
 aggEntList.h B-21
 aggPub.h 9-2, B-23
 aggregateSR.h B-30
 aggStatePdu.h B-24
 antPatRec.h B-34
 approximator.h B-35
 apSpec.h B-37
 arealObjStPdu.h B-38
 artPart.h B-41
 artPartList.h B-44
 artPartThresh.h B-48
 attPartList.h B-51
 baseEmbedDec.h B-54
 baseEmbedEnc.h B-56
 baseEmittrDec.h B-58
 baseEmittrEnc.h B-60
 baseEntDec.h B-62
 baseEntEnc.h B-64
 baseEntitySR.h B-67
 baseExConn.h B-71
 baseHashKey.h B-73
 baseMsg.h B-74
 baseReflAgg.h B-76
 baseReflEnt.h B-77
 beamAntRec.h B-78
 beamCbInfo.h B-80
 cartView.h B-82

ClientSocket.h B-84
cliServer.h B-87
collideInter.h 9-2
collidePdu.h B-87
collisElasPdu.h B-89
commentPdu.h B-92
commentRPdu.h B-93
createEntRPdu.h B-95
createPdu.h B-97
cross referenced to classes 9-4
dAggDec.h B-98
dAggEnc.h B-99
dAggPub.h B-101
dataPdu.h B-102
dataQueryRPdu.h B-103
dataRPdu.h B-105
datumIdType.h B-107
datumMap.h B-108
datumRecSet.h B-110
dcm.h B-116
deadReckoner.h B-117
decFactory.h B-120
dEntIdHashLst.h B-122
dEntityDec.h B-123
dEntityEnc.h B-124
dEntityPub.h B-126
designatePdu.h B-127
designatorDec.h B-129
designatorEnc.h B-131
designatorSR.h B-133
designCbInfo.h B-135
detonateInter.h 9-2
detonatePdu.h B-137
dExerciseConn.h B-140
dInteraction.h B-143
directRotRate.h B-144
dirRtiAmb.h B-145
disEnums.h B-155
disPduFactory.h B-194
dReflAggList.h B-195
dReflectedAgg.h B-198
dReflectedEnt.h B-200
dReflEntList.h B-202
dscvObjCbInfo.h B-205
dStateMsg.h B-207
elecEmitPdu.h B-208
embedSystemSR.h B-212
emissionPdu.h B-213
emitrSysCbInf.h B-216
emitterSysDec.h B-218
emitterSysEnc.h B-220
emitterSysSR.h B-222
emitttrBeamSR.h B-224
encFactory.h B-227
entCbInfo.h B-230
entityPub.h 9-2
entitySR.h B-232
EntityTypes.h B-237
enum2string.h B-253
envDec.h B-254
envEnc.h B-255
enviroTypeRec.h B-257
envProcPdu.h B-258
esPdu.h B-260
esUpdatePdu.h B-263
Euler.h B-266
eventId.h B-267
eventReptPdu.h B-269
eventReptRPdu.h B-270
exerciseConn.h 9-2
firePdu.h B-273
fom.h B-275
fomMapper.h B-277
gedBasFixWing.h B-280
gedBasRotWing.h B-282
gedBasSoldier.h B-285
gedBasVehicle.h B-287
gedEnhFixWing.h B-289
gedEnhRotWing.h B-291
gedEnhSoldier.h B-293
gedEnhVehicle.h B-295
gedLogVehicle.h B-297
geodCoord.h B-299
globalObjDes.h B-301
globIdHLst.h B-302
gridAxiRecFix.h B-303
gridAxiRecVar.h B-305
gridDataPdu.h B-307
gridDataRec.h B-311
gridDataRecT0.h B-312
gridDataRecT1.h B-314
gridDataRecT2.h B-316
for grouping other header files 9-2
hAckDec.h B-318
hAckEnc.h B-319
hAckInter.h B-320
hActReqDec.h B-322
hActReqEnc.h B-323
hActReqInter.h B-324
hAggPub.h B-326

hAppRadSigDec.h B-328
hAppRadSigEnc.h B-329
hashtable.h B-330
hCollideDec.h B-332
hCollideEnc.h B-334
hCollideInter.h B-335
hDbiRadSigDec.h B-337
hDbiRadSigEnc.h B-338
hDesignatrPub.h B-339
hDetDec.h B-340
hDetEnc.h B-342
hDetInter.h B-343
hEmitrBeamPub.h B-346
hEmittrSysPub.h B-348
hEncARSigDec.h B-350
hEncARSigEnc.h B-351
hEntityPub.h B-352
hExerciseConn.h B-353
hFireDec.h B-357
hFireEnc.h B-358
hFireInter.h B-359
hierAdpt.h B-361
hInteraction.h B-361
hInterFactory.h B-364
hInterWData.h B-365
hIntWEncDec.h B-369
hlaObjFact.h B-375
hlaObj.h B-371
hlaObjWithSR.h B-377
hLcDec.h B-379
hLcEnc.h B-380
hLcInter.h B-381
hObjPub.h B-382
hostStructs.h B-384
hRadioRcvrPub.h B-389
hRadioXmitPub.h B-390
hRawBinRSDec.h B-391
hRawBinRSEnc.h B-392
hRefBeamList.h B-393
hRefDesigList.h B-395
hRefEmSysList.h B-397
hRefEntHList.h B-400
hReflAggList.h B-401
hReflBeam.h B-403
hReflDesigntr.h B-405
hReflectedAgg.h B-407
hReflectedEnt.h B-409
hReflEmitrSys.h B-418
hReflEntList.h B-420
hReflRadRecvr.h B-422
hReflRadXmitr.h B-424
hRefRadRvList.h B-426
hRefRadXmList.h B-428
hSignalInter.h B-430
hStartDec.h B-432
hStartEnc.h B-433
hStartInter.h B-434
hStateDecoder.h B-436
hStateEncoder.h B-439
hStateMsg.h B-442
hStopDec.h B-444
hStopEnc.h B-445
hStopInter.h B-446
hUnknownInter.h B-448
hVcDec.h B-450
hVcEnc.h B-451
hVcInter.h B-452
iffAtcNavBeam.h B-453
iffAtcNavLayr.h B-455
iffAtcNavOper.h B-456
iffAtcNavOpr2.h B-458
iffAtcNavParm.h B-459
iffAtcNavPdu.h B-461
iffAtcNavSyID.h B-464
intClassDesc.h B-466
intClassMap.h B-468
intDecFact.h B-469
interaction.h 9-3
intercmCtlPdu.h B-473
intercmEntDst.h B-476
intercmGrpAsg.h B-478
intercmGrpDst.h B-480
intercmParRec.h B-481
intercmSigPdu.h B-482
interDecoder.h B-484
interEncoder.h B-487
interMgr.h B-489
intHashKey.h B-491
isGroupOfPdu.h B-492
isPartOfPdu.h B-495
laserPdu.h B-497
lcEnd.h B-498
lcEofAction.h B-499
lcHeartbeat.h B-500
lcInter.h 9-3, B-501
lcPause.h B-502
lcPdu.h B-503
lcPlayAction.h B-507
lcPlayFile.h B-508
lcPlay.h B-506

lcQuit.h B-509
lcRecAction.h B-510
lcRecFile.h B-511
lcRecord.h B-512
lcSetTime.h B-513
lcSkipTime.h B-514
lcStart.h B-515
lcStop.h B-516
lcTimeout.h B-518
lcTimeScale.h B-517
leAppearPdu.h B-519
leArtPartsPdu.h B-522
leDeadReck.h B-525
leDetonPdu.h B-526
leEntityID.h B-530
leEntityOri.h B-531
leEntityVel.h B-533
leFirePdu.h B-534
leLocEntity.h B-537
leLocWorld.h B-538
leOriError.h B-540
lePosError.h B-541
leTspiPdu.h B-542
LibMatrix.h B-545
linObjStPdu.h B-550
locObjMgr.h B-554
mathTypes.h B-557
matricInc.h 9-2
milEntDec.h B-560
milEntEnc.h B-562
milPlatDec.h B-564
milPlatEnc.h B-566
mineDataPdu.h B-568
minePerimeter.h B-574
mineQueryPdu.h B-575
mineRsNackPdu.h B-578
mineStatePdu.h B-580
mtlInc.h 9-2
munEntDec.h B-583
munEntEnc.h B-584
netBoolean.h B-586
NetSocket.h B-587
NetStructs.h B-591
netTimeStamp.h B-598
NetTypes.h B-599
objClassDesc.h B-602
objDesList.h B-605
objectType.h B-606
objIdHashKey.h B-607
objIdHashLst.h B-608
objRefCbInfo.h B-609
pairSock.h B-610
pduCbInfo.h B-613
pduCbMgr.h B-614
pduDatRecSet.h B-615
pduFactory.h B-616
pdu.h B-611
pduWithData.h B-617
physEntDec.h B-621
physEntEnc.h B-623
pointObjStPdu.h B-625
printUtil.h B-628
ProcControl.h B-629
queryPdu.h B-630
radioRecvrDec.h B-632
radioRecvrEnc.h B-634
radioRecvrSR.h B-636
radioXmitDec.h B-637
radioXmitEnc.h B-639
radioXmitSR.h B-641
radRcvrCbInfo.h B-644
radXmitCbInfo.h B-631
receivePdu.h B-645
recIntrCbInfo.h B-646
recordPdu.h B-647
recordSet.h B-649
recordSetPdu.h B-651
recQueryPdu.h B-653
refAttrCbInfo.h B-655
refAttrCbList.h B-656
reflAggList.h 9-3, B-658
reflectedAgg.h 9-3
reflEntList.h 9-3
refObjMgr.h B-661
regCtlCbInfo.h B-663
regCtlCbList.h B-664
remObjCbInfo.h B-666
removeEntRPdu.h B-667
removePdu.h B-669
repairPdu.h B-670
resupplyPdu.h B-672
rprrDecFact.h B-675
rprrEncFact.h B-676
rprrFomMap.h B-677
rprrIDecFact.h B-679
rprrIEncFact.h B-680
rprrInterFact.h B-681
rprrObjFact.h B-682
script.h B-683
servReqPdu.h B-684

- setDataPdu.h* B-686
- setDataRPdu.h* B-687
- setRecordPdu.h* B-689
- signalInter.h* 9-3
- signalPdu.h* B-691
- simanPdu.h* B-693
- smoothUtil.h* B-699
- smUtil.h* B-702
- Socket.h* B-703
- soldierDec.h* B-706
- soldierEnc.h* B-707
- startInter.h* 9-3
- startPdu.h* B-709
- startResRPdu.h* B-710
- stateRep.h* B-712
- stIntCbInfo.h* B-713
- stIntCbList.h* B-714
- stopFrzRPdu.h* B-716
- stopInter.h* 9-3
- stopPdu.h* B-718
- stopUpdCbInfo.h* B-719
- stopUpdCbList.h* B-720
- str.h* B-722
- strtUpdCbInfo.h* B-726
- strtUpdCbList.h* B-727
- supEmisEsNozz.h* B-729
- supEmisEsPdu.h* B-730
- supEmisEsProp.h* B-732
- taitBryan.h* B-733
- threshold.h* B-734
- topoCoord.h* B-736
- topoView.h* B-737
- transCtlPdu.h* B-738
- transmitPdu.h* B-740
- trkJamBeamDec.h* B-743
- trkJamBeamEnc.h* B-745
- undwaAcBeam.h* B-748
- undwaAcEmitSys.h* B-750
- undwaAcPasAct.h* B-752
- undwaAcPdu.h* B-753
- undwaAcShaft.h* B-756
- unknownHObj.h* B-757
- unknownPdu.h* B-759
- updCtlCbInfo.h* B-760
- updCtlCbList.h* B-761
- utmCoord.h* B-763
- utmView.h* B-766
- varDatumRec.h* B-768
- vcPdu.h* B-770
- vlInc.h* 9-2
- vlList.h* B-777
- vlprint.h* B-779
- vlTime.h* B-780
- vlutil.h* B-781
- vlutilInc.h* 9-2
- vlVector.h* B-784
- urlFedAmb.h* B-786
- urlFedTime.h* B-790
- urlRtiAmb.h* B-793
- heading 7-15
- heading, pitch and roll, obtaining 7-11
- heartbeat 4-15
- hEmitrBeamPub.h* header file B-346
- hEmittrSysPub.h* header file B-348
- hEncARSigDec.h* header file B-350
- hEncARSigEnc.h* header file B-351
- hEntityPub.h* header file B-352
- hExerciseConn.h* header file B-353
- hFireDec.h* header file B-357
- hFireEnc.h* header file B-358
- hFireInter.h* header file B-359
- hierAdpt.h* header file B-361
- hInteraction.h* header file B-361
- hInterFactory.h* header file B-364
- hInterWData.h* header file B-365
- hIntWEncDec.h* header file B-369
- HLA
 - creating exercise connection 4-2
 - discovering and removing objects 6-8
 - executables 2-5
 - objects
 - listing 6-5
 - objects list 6-9
 - objects, managing 6-4
 - subscribing to object classes 6-7
 - subscriptions 6-11
 - Web site 1-3
- hlaNetdump example application 8-8
- hlaObject() function 6-4, 6-6, 6-9
- hlaObjFact.h* header file B-375
- hlaObj.h* header file B-371
- hlaObjWithSR.h* header file B-377
- hLcDec.h* header file B-379
- hLcEnc.h* header file B-380
- hLcInter.h* header file B-381
- hObjPub.h* header file B-382
- hostStructs.h* header file B-384
- hRadioRcvrPub.h* header file B-389
- hRadioXmitPub.h* header file B-390
- hRawBinRSDec.h* header file B-391

hRawBinRSEnc.h header file B-392
hRefBeamList.h header file B-393
hRefDesigList.h header file B-395
hRefEmSysList.h header file B-397
hRefEntHList.h header file B-400
hReflAggList.h header file B-401
hReflBeam.h header file B-403
hReflDesigntr.h header file B-405
hReflectedAgg.h header file B-407
hReflectedEnt.h header file B-409
hReflEmitrSys.h header file B-418
hReflEntList.h header file B-420
hReflRadRecv.h header file B-422
hReflRadXmitr.h header file B-424
hReflRadRvList.h header file B-426
hReflRadXmList.h header file B-428
hSignalInter.h header file B-430
hStartDec.h header file B-432
hStartEnc.h header file B-433
hStartInter.h header file B-434
hStateDecoder.h header file B-436
hStateEncoder.h header file B-439
hStateMsg.h header file B-442
hStopDec.h header file B-444
hStopEnc.h header file B-445
hStopInter.h header file B-446
hUnknownInter.h header file B-448
hVcDec.h header file B-450
hVcEnc.h header file B-451
hVcInter.h header file B-452

I

ID

- obtaining for entity 4-4
- specifying for entity 4-11

id() function 4-17

identity() function 7-3

IEEE

1278.1-1995 1-3

Web site 1-3

iffAtcNavBeam.h header file B-453

iffAtcNavLayr.h header file B-455

iffAtcNavOper.h header file B-456

iffAtcNavOpr2.h header file B-458

iffAtcNavParm.h header file B-459

iffAtcNavPdu.h header file B-461

iffAtcNavSyID.h header file B-464

information message 7-21

init() function 4-30 to 4-31

initArtParts() function 4-31

initiatePause() function 6-3

initPdu() function 5-16, 5-19

insertBytes() function 5-18

inspecting

 PDU data 5-7

installation 2-4

installing

 DMSO RTI 2-11

 MÄK RTI 2-10

intClassDesc.h header file B-466

intClassMap.h header file B-468

intDecFact.h header file B-469

interaction 4-4, 4-7, 4-10

 and protocol independence 4-7

 between classes 3-3

 callbacks 4-8

 classes 3-2

DtInteraction 6-12

 factory 6-22

 publishing 6-11

 receiving 4-4, 4-8

 receiving with *DtUnknownInteraction* 6-15

 sending 4-4, 4-7

 sending with *DtUnknownInteraction* 6-14

 subscribing 6-11

 unknown 6-14

 user-defined 6-14, 6-17

interactionClassHandle() function 6-12, 6-17

interactionFactory() function 6-13

interaction.h header file 9-3

intercmCtlPdu.h header file B-473

intercmEntDst.h header file B-476

intercmGrpAsg.h header file B-478

intercmGrpDst.h header file B-480

intercmParRec.h header file B-481

intercmSigPdu.h header file B-482

interDecoder.h header file B-484

interEncoder.h header file B-487

interest, registering 5-5, 5-24 to 5-25

interMgr.h header file B-489

internal mode

 in packet server 8-12

internalGetPduKind() function 5-15 to 5-16

intHashKey.h header file B-491

IP address

 manipulating 7-22

ipAddr() function 5-28

IP/UDP network, communicating with 5-21

IRIX patches 2-2

isBundling() function 5-25
isGroupOfPdu.h header file B-492
 isOk() function 5-28
isPartOfPdu.h header file B-495

J

joining exercises 4-21

K

kind() function 5-7

L

laserPdu.h header file B-497
 last() function 4-16, 7-19
 lastSourceInetAddr() function 5-27
 lat() function 7-8
lcEnd.h header file B-498
lcEofAction.h header file B-499
lcHeartbeat.h header file B-500
lcInter.h header file 9-3, B-501
lcPause.h header file B-502
lcPdu.h header file B-503
lcPlayAction.h header file B-507
lcPlayFile.h header file B-508
lcPlay.h header file B-506
lcQuit.h header file B-509
lcRecAction.h header file B-510
lcRecFile.h header file B-511
lcRecord.h header file B-512
lcSetTime.h header file B-513
lcSkipTime.h header file B-514
lcStart.h header file B-515
lcStop.h header file B-516
lcTimeout.h header file B-518
lcTimeScale.h header file B-517
leAppearPdu.h header file B-519
leArtPartsPdu.h header file B-522
 leaving exercises 4-21
leDeadReck.h header file B-525
leDetonPdu.h header file B-526
leEntityID.h header file B-530
leEntityOri.h header file B-531
leEntityVel.h header file B-533
leFirePdu.h header file B-534
leLocEntity.h header file B-537
leLocWorld.h header file B-538
 length() function 5-7

leOriError.h header file B-540
lePosError.h header file B-541
leTspiPdu.h header file B-542
LibMatrix.h header file B-545
 libraries 3-13
 license manager 2-6
 location of 2-5
 setting up 2-6
 license server 2-6
 running 2-8
 sample applications, for 8-2
 status 2-8
 stopping 2-8
 license, environment variable 2-6
license.dat file 2-6 to 2-7
 linked-list, creating 7-19
 linking 3-13
linObjStPdu.h header file B-550
 listen example application 3-4, 8-9
 listenToMulticastGroup() function 5-28
 listing HLA objects 6-5
 lists 7-19
 little endian machines 5-14 to 5-15, 6-15
 LM_LICENSE_FILE environment variable 2-6
lmdown program 2-8
lmhostid program 2-6 to 2-7
lmstat program 2-8
 local object manager 6-5
 locally-simulated
 entities 4-11
 object 6-4
 localObjectManager() function 6-5
 location() function 4-18, 4-20
locObjMgr.h header file B-554
 Logger xix
 lon() function 7-8
 lookup() function 4-17
 lower-level classes 3-2
 low-fidelity entity state representation 4-15

M

machine
 little endian 5-14, 6-15
 maintenance agreement xix
 MÄK Logger xix
 MÄK Plan View Display xix
 MÄK RTI xix
 configuring 2-10
 installing 2-10

- MÄK Stealth xix
- MÄK Technical Support xx
- makefile 3-14
- makeMods files 3-14
- managing
 - entities 4-11
 - HLA objects 6-4
 - time 7-17
- manipulating
 - IP addresses 7-22
- manual
 - conventions xxi
 - location of 2-5
- map datum 7-9
- mappings, for PDU kind 5-11
- mathTypes.h* header file 9-2, B-557
- matrix 7-2, 7-4
 - direction cosine 7-2
 - multiplying by vector 7-15
- matrixInc.h* file 9-2
- memory, allocating for PDU 5-16, 5-18
- message
 - diagnostic 7-21
 - sending and receiving 4-2
- Microsoft C++ IDE 3-17
- milEntDec.h* header file B-560
- milEntEnc.h* header file B-562
- milPlatDec.h* header file B-564
- milPlatEnc.h* header file B-566
- mineDataPdu.h* header file B-568
- minePerimeter.h* header file B-574
- mineQueryPdu.h* header file B-575
- mineRsNackPdu.h* header file B-578
- mineStatePdu.h* header file B-580
- mode
 - packet server 8-12
- moving parts 4-28
- mtlInc.h* header file 9-2
- multicast 5-23 to 5-24, 5-28
- multiple applications, running 8-12
- multithreaded libraries 3-13
- munEntDec.h* header file B-583
- munEntEnc.h* header file B-584
- munitonFlightTime parameter 8-5

N

- N32 ABI 2-2
 - compile and link flag for 2-2
- name() function 6-12, 6-17

- net types 5-14 to 5-15
- netBoolean.h* header file B-586
- netdump example application 8-6
- nethex example application 8-9
- netmask
 - configuring 2-13 to 2-14
 - obtaining for a device 7-22
- netRead() function 5-4, 5-25 to 5-26
 - using 5-5
- NetSocket.h* header file B-587
- NetStructs.h* header file B-591
- netTimeStamp.h* header file B-598
- NetTypes.h* header file B-599
- network
 - connecting to for DIS 5-21
- network representation of PDU 5-9
- newReflectedEntity() function 4-23, 6-9
- next() function 4-16, 7-19
- nextEventId() function 4-4
- nextId() function 4-4, 4-12
- nextValidParamType() function 4-29
- no vlink server* error message 8-11, 8-20
- non-blocking reads and writes 5-27
- notify level 7-21
- notifyMe() function 5-27
- numArtParams() function 4-28
- numArtParts() function 4-28
- numParameters() function 6-12, 6-17
- numParams() function 4-29

O

- objClassDesc.h* header file B-602
- objDesList.h* header file B-605
- object
 - discovering and removing 6-8
 - locally-simulated 6-4
- object ID 4-4
- ObjectClassHandle* class 6-4
- objectClassHandle() function 6-5
- objectId() function 6-10
- objectType.h* header file B-606
- objIdHashKey.h* header file B-607
- objIdHashLst.h* header file B-608
- objRefCbInfo.h* header file B-609
- offset, translational 7-16
- orientation 4-18, 7-5 to 7-6
 - conversion 7-7
 - setting 4-13
- orientation() function 4-18, 4-20

P

packet

- broadcast 5-21
- bundling 5-25
- throughput, improving 8-13
- unbundling 8-18
- unicast 5-21

packet () function 5-11

packet server 5-5, 5-21, 5-24, 8-12, 8-14

- and reuse port 8-13
- as daemon 8-19
- background operation 8-19
- communicating through 5-21
- configurations 8-12
- controlling with vdc 8-10
- controls 8-17
- detecting 5-23
- DrClientSocket* interface to 8-19
- and exercise connection 4-3
- external mode 8-19
- modes 8-12
- pinging clients 8-18
- queue size 8-19
- resource constraints 8-20
- and server socket file 8-20
- starting 8-15
- startup options 8-15
- status commands 8-18
- stopping 8-17
- unbundling packets 5-26, 8-18
- use of 8-13

packet() function 5-8

pairSock.h header file B-610

parameter

- getting value 4-29
- munitionFlightTime* 8-5

ParameterHandleValuePairSet()

- function 6-12 to 6-14, 6-17

part

- moving 4-28

partAttachedTo() function 4-29

partNum() function 4-29

partType() function 4-29

patches for IRIX6 2-2

PDU

- class reference A-1
- constructing blank 5-8
- constructing from network representation 5-9
- constructing unknown type 5-10

copying 5-12

creating 5-13

destination address 5-3

examining 5-15, 5-17

filtering 5-5, 5-24, 5-28

getting network representation 5-8

header information 5-9

inspecting 5-7

protocol version 5-11

receiving 5-3

registering callbacks 5-14

representing 5-2

sending 5-2

setting data in 5-7, 5-15, 5-17

user-defined 5-13 to 5-14

variable length 5-18

version 5-9

pduCbInfo.h header file B-613

pduCbMgr.h header file B-614

pduDatRecSet.h header file B-615

pduFactory() function 5-10

pduFactory.h header file B-616

pdu.h header file B-611

pduKind() function 5-28

pdu.h grouping file 9-2

pduWithData.h header file B-617

phi() function 7-5

phone number

- MÄK Technologies xx

phvps() function 6-12, 6-17, 6-21

physEntDec.h header file B-621

physEntEnc.h header file B-623

pinging clients with packet server 8-18

pitch 7-15

- obtaining 7-11

Plan View Display xix

platform

- supported 2-2

pointObjStPdu.h header file B-625

port number 4-3

Portable Document Format (PDF) xviii

position information

- obtaining 4-18

- setting 4-13

postDrainCallback() function 4-10

prev() function 4-16, 7-19

print() function 6-17

- articulated parts 4-29

- interactions for 6-12

- PDU 5-7

- printData() function 5-7
 - entity state repository 4-19
 - PDU 5-7
 - printing entity state repository 4-14
 - protocol independent 6-12
- printHeader() function 5-7, 6-12
- printing
 - entity state data 4-14
- printParams() function 6-12
- print-related functions 6-12
- printUtil.h* header file B-628
- ProcControl.h* header file B-629
- processEntityState() function 5-29
- processPdu() function 5-4
- products
 - Dial-a-Sim xix
 - Logger xix
 - Plan View Display xix
 - RTI xix
 - Stealth xix
 - technical support xx
 - upgrades xx
- protocol 1-3
 - selecting at compilation 3-13
- protocol version
 - receiving 5-11
 - setting in PDU 5-9
- protocolFamily() function 5-7
- protocol-independent API 1-1
- protocol-specific classes 3-2
- protocolVersion() function 5-7, 5-28
- provideAttributeValues RTI service 6-6
- psi() function 7-5
- publishing 6-5
 - interactions 6-11
- publishObjectClass RTI service 6-2

Q

- queryPdu.h* header file B-630
- queue size 8-20
 - packet server 8-19

R

- radioRecvrDec.h* header file B-632
- radioRecvrEnc.h* header file B-634
- radioRecvrSR.h* header file B-636
- radioXmitDec.h* header file B-637
- radioXmitEnc.h* header file B-639

- radioXmitSR.h* header file B-641
- radRcvrCbInfo.h* header file B-644
- radXmitCbInfo.h* header file B-631
- readAndProcess() function 5-4
- readFd() function 5-27
- reads
 - non-blocking 5-27
- readUntil() function 5-6
- Realtime-Platform Reference FOM 1-3
- receiveInteraction RTI service 6-2, 6-12
- receivePdu.h* header file B-645
- receiving
 - functions 5-26
 - interactions 4-4, 4-8
 - interactions with
 - DtUnknownInteraction* 6-15
 - PDU 5-3
 - state updates 6-10
- recIntrCbInfo.h* header file B-646
- recordPdu.h* header file B-647
- recordSet.h* header file B-649
- recordSetPdu.h* header file B-651
- recQueryPdu.h* header file B-653
- refAttrCbInfo.h* header file B-655
- refAttrCbList.h* header file B-656
- reference ellipsoid 7-9
- reflAggList.h* header file 9-3, B-658
- reflectAttributeValues RTI service 6-2, 6-7
- reflected entity list 1-2, 3-6, 4-15
 - creating 4-16
 - differences in HLA and DIS 4-15
- reflected entity list. *See also* *DtReflectedEntityList* and *DtReflectedEntity* 7-17
- reflectedAgg.h* header file 9-3
- reflectedEnt.h* header file 9-3
- reflectedObjectManager() function 6-9
- reflEntList.h* header file 9-3
- refObjMgr.h* header file B-661
- regCtlCbInfo.h* header file B-663
- regCtlCbList.h* header file B-664
- registering
 - callbacks
 - HLA 6-7
 - protocol independent 3-10
 - class with *DtExerciseConn* 6-22
 - functions 4-8
 - functions with *DtExerciseConn* 4-4
 - incoming PDU 5-3
 - interest 5-5, 5-24 to 5-25
- registerObject service 6-5

-
- REL. *See* reflected entity list
 - remObjCbInfo.h* header file B-666
 - removeAndDelete() function 4-21, 6-8 to 6-9
 - removeCallback() function 4-8, 5-4
 - interactions for 6-17
 - PDU 5-3, 5-15
 - removeDiscoverObjectCallback() function 6-8
 - removeEntRPdu.h* header file B-667
 - removeObject RTI service 6-8
 - removePdu.h* header file B-669
 - removePostDrainCallback() function 4-4
 - removePostReflectCb() function 6-10
 - removeRemoveObjectCallback() function 6-8
 - removeRemoveObjectCb() function 6-8
 - removeStartUpdatesCb() function 6-7
 - removeStopUpdatesCb() function 6-7
 - removing
 - callback functions 4-8
 - entity 4-21
 - HLA objects 6-8
 - repairPdu.h* header file B-670
 - requestClassAttributeValueUpdate RTI
 - service 6-7
 - requestedAttributes() function 6-6
 - requestPause RTI service 6-2
 - resource, system
 - packet server 8-20
 - resupplyPdu.h* header file B-672
 - returning
 - broadcast address 7-22
 - reuse port 8-13
 - RID file 2-10
 - right-hand geocentric coordinates 7-7
 - roll 7-15
 - obtaining 7-11
 - rotation
 - matrix 7-15
 - threshold 4-15
 - rotationalVelocity() function 4-18, 4-25
 - RPR FOM 1-3
 - rprDecFact.h* header file B-675
 - rprEncFact.h* header file B-676
 - rprFomMap.h* header file B-677
 - rprIDecFact.h* header file B-679
 - rprIEncFact.h* header file B-680
 - rprInterFact.h* header file B-681
 - rprObjFact.h* header file B-682
 - RTI xix, 6-2
 - ambassador 4-3
 - calling directly 6-2
 - classes 6-2
 - definition of 2-10
 - DMSO, installing 2-11
 - installing MÄK 2-10
 - ParameterHandleValuePairSet()
 - function 6-12 to 6-14, 6-17
 - RTIambassador 6-2
 - tick() function 4-4
 - RTI_CONFIG environment variable 2-10
 - rtiAmb() function 6-2
 - RTIambassador* class 6-2
 - RTIambassador::tick() function 6-2
 - rtiexec 2-12
 - rtiexec* 4-3
 - RTI-initiated services 6-2
 - runLm* program 2-8
 - running
 - license server 2-8
 - multiple VR-Link applications 8-12
- ## S
- script
 - configRTI 2-11
 - script.h* header file B-683
 - select system call, interface to 7-23
 - send() function
 - definition 4-4
 - in example 3-8
 - in HLA 4-8
 - multicast address 5-24
 - obtaining object for RTI 6-12
 - optional address in DIS 5-3
 - publishing interactions 6-11
 - sending DIS PDUs 5-26
 - sending interactions 6-15
 - sending PDUs 5-2
 - unknown PDU 5-13
 - with user-defined classes 5-14
 - sending
 - functions 5-26
 - interactions 4-4, 4-7, 6-14
 - PDUs 5-2
 - PDUs of different versions 5-9
 - sendInteraction RTI service 6-2
 - send-only application 3-7
 - sendStamped() function
 - definition 4-4
 - in example 3-8
 - in HLA 4-8

- multicast address 5-24
- obtaining object for RTI 6-12
- optional address in DIS 5-3
- publishing interactions 6-11
- sending DIS PDUs 5-26
- sending interactions 6-15
- sending PDUs 5-2
- unknown PDU 5-13
- with user-defined classes 5-14
- sendTo() function 5-25 to 5-26
- server
 - license 2-6
 - socket file 8-20
- service
 - deleteObject 6-5
 - discoverObject 4-22, 6-2, 6-7 to 6-8
 - provideAttributeValues 6-6
 - publishObjectClass 6-2
 - receiveInteraction 6-2, 6-12
 - reflectAttributeValues 6-2, 6-7
 - registerObject 6-5
 - removeObject 6-8
 - requestClassAttributeValueUpdate 6-7
 - requestPause 6-2
 - RTI 6-2
 - sendInteraction 6-2
 - startUpdates 6-6
 - stopUpdates 6-6
 - subscribeInteractionClass 6-11
 - updateAttributeValues 6-2, 6-6
- servReqPdu.h* header file B-684
- setAcceleration() function 4-13
- setAlt() function 7-8
- setApproximating() function 4-29
- setApproximator() function 3-11, 4-20
- setArtParamVal() function 4-31
- setBundling() function 5-25
- setByInverse() function 7-11
- setDamageState() function 4-13
- setDataPdu.h* header file B-686
- setDataRPdu.h* header file B-687
- setDefaultRotationThreshold() function 4-15
- setDefaultTimeThreshold() function 4-15
- setDefaultTranslationThreshold() function 4-15
- setEngineSmokeOn() function 4-13
- setEntityType() function 5-2
- setExerciseId() function 5-7, 5-21
- setFedAmbCreator() function 6-3
- setFlamesPresent() function 4-13
- setFromPhyps() function 6-12, 6-17, 6-19, 6-21
- setGeocentric() function 7-8
- setHatchState() function 4-13
- setLat() function 7-8
- setLocation function 3-11
- setLocation() function 4-13
- setLon() function 7-8
- setMcastTtl() function 5-24, 5-28
- setOffset() function 4-27
- setOrientation() function 4-13
- setPhi() function 7-5
- setPsi() function 7-5
- setRecordPdu.h* header file B-689
- setRotation() function 4-27
- setRotationalVelocity() function 4-13, 4-25
- setTheta() function 7-5
- setTimeoutInterval() function 4-23
- setTimeoutProcessing() function 4-23
- setTimeStamp() function 5-7
- setTimeThreshold() function 4-15
- setting
 - articulated part data 4-30
 - data in new PDU 5-15
 - entity state 4-13
 - geocentric coordinates 7-8
 - orientation 4-13
 - PDU data 5-7
 - space, time, and position 4-13
 - thresholds 4-15
- setting up
 - license manager 2-6
- setUnbundling() function 5-25
- setupUpdating() function 6-5
- setVal() function 4-31
- setVelocity function 3-11
- setVelocity() function 4-13
- shutting down
 - license server 2-8
- signalInter.h* header file 9-3
- signalPdu.h* header file B-691
- simanPdu.h* header file B-693
- simple demo applications 3-4
- simulation time 7-17
- simulation time. *See* VR-Link
 - simulation time
- site ID
 - DIS 4-3, 4-17
- sleep 3-9
- sleep() function 7-23
- smoothUtil.h* header file B-699
- smUtil.h* header file B-702

- socket
 - creating for DIS 5-22
- socket file
 - AF-UNIX 8-11
- socket() function 5-22
- Socket.h* header file B-703
- soldierDec.h* header file B-706
- soldierEnc.h* header file B-707
- source code
 - sample applications 8-2
- space information
 - obtaining 4-18
 - setting 4-13
- specifying
 - dead reckoning algorithm 4-12
 - force ID 4-12
 - guise 4-12
- starting
 - packet server 8-15
- startInter.h* header file 9-3
- startPdu.h* header file B-709
- startResRPdu.h* header file B-710
- startUpdates RTI service 6-6
- state
 - setting for entity 4-13
 - update 4-16
 - communicating in HLA 6-4
 - receiving 4-22, 5-29, 6-10
- stateMsg.h* header file 9-3
- stateRep.h* header file B-712
- status
 - license server 2-8
- status() function 5-10
- Stealth xix
- stIntCbInfo.h* header file B-713
- stIntCbList.h* header file B-714
- stopFrzRPdu.h* header file B-716
- stopInter.h* header file 9-3
- stopPdu.h* header file B-718
- stopping
 - license server 2-8
 - packet server 8-17
- stopUpdates RTI service 6-6
- stopUpdCbInfo.h* header file B-719
- stopUpdCbList.h* header file B-720
- str.h* header file B-722
- string() function 7-2, 7-5
- strrtUpdCbInfo.h* header file B-726
- strrtUpdCbList.h* header file B-727
- subclassing

- DtReflectedEntity* 4-23
- subscribeInteractionClass RTI service 6-11
- subscribing
 - HLA classes to 6-11
 - interactions to 6-11
 - object classes, to 6-7
 - to multicast in DIS 5-24
- subtractInterestInMcastAddr() function 5-24, 5-28
- subtractInterestInPduKind() function 5-25
- supEmisEsNozz.h* header file B-729
- supEmisEsPdu.h* header file B-730
- supEmisEsProp.h* header file B-732
- support, technical xx
- supported platforms 2-2
- system requirements 2-2

T

- Tait-Bryan sequence 4-13, 7-5
- taitBryan.h* header file B-733
- talk example application 8-9
- technical support xx
- terminating
 - packet server 8-17
- theta() function 7-5
- threshold 7-17
 - rotation 4-15
 - setting 4-15
 - time 4-15
 - translation 4-15
- thresholder.h* header file B-734
- throughput, improving for DIS packets 8-13
- tick() function 3-11, 4-14
 - DtEntityPublisher 3-8
 - RTI 4-4, 4-10
 - RTI ambassador 6-2
- time
 - managing 7-17
 - simulation 4-28, 7-17
 - threshold 4-15
- time information
 - obtaining 4-18
 - setting 4-13
- time to live, for multicast packets 5-24, 5-28
- timeout 4-23, 5-6
 - argument to drainInput() 4-5
- timestamp 5-2
- timeStamp() function 5-7
- timeStampType() function 5-7

- timing out
 - entities 4-23
- topoCoord.h* header file B-736
- topographic coordinate
 - converting to geocentric 7-10
- topographic coordinate system 4-25 to 4-26, 7-5, 7-10
 - and UTM coordinate system, compared 7-13
- topographic view 3-6, 3-8
- topoView.h* header file B-737
- transCtlPdu.h* header file B-738
- translation threshold 4-15
- translational offset 7-16
- translations() function 4-29
- transmitPdu.h* header file B-740
- trkJamBeamDec.h* header file B-743
- trkJamBeamEnc.h* header file B-745
- troubleshooting
 - vdc 8-11
- TTL. *See* time to live

U

- UDP port number 2-13, 4-3, 5-21
- unbundling 5-25
 - packets 8-18
- undwaAcBeam.h* header file B-748
- undwaAcEmitSys.h* header file B-750
- undwaAcPasAct.h* header file B-752
- undwaAcPdu.h* header file B-753
- undwaAcShaft.h* header file B-756
- unicast 5-21, 5-23
- Universal Transverse Mercator. *See* UTM
- unknown
 - interactions 6-14
 - PDU, constructing 5-10
- unknownHObj.h* header file B-757
- unknownPdu.h* header file B-759
- unregistering
 - callbacks
 - HLA 6-7
 - protocol independent 3-10
 - functions 4-8
 - functions with *DtExerciseConn* 4-4
 - incoming PDU 5-3
- update
 - determining need for 6-6
 - processing 4-16
 - state 4-22
 - state, in HLA 6-4

- update() function 6-6
- updateAttributeValues RTI service 6-2, 6-6
- updCtlCbInfo.h* header file B-760
- updCtlCbList.h* header file B-761
- upgrades xx
- user-defined
 - interaction 6-14, 6-17
 - PDU 5-13 to 5-14
- using
 - DtInteraction* 6-12
 - external buffers in PDU 5-11
 - packet server 8-13
- utility functions 3-2
- UTM coordinate system 4-25, 4-27, 7-12
 - and topographic system, compared 7-13
- utmCoord.h* header file B-763
- utmView.h* header file B-766

V

- val() function 4-29
- varDatumRec.h* header file B-768
- variable
 - DtNotifyLevel 7-21
 - DtProtocolVersionToSend 5-9
 - license environment 2-6
- variable length PDU 5-18
- vcInter.h* header file 9-3
- vcPdu.h* header file B-770
- vdc 8-10
 - application 8-10
 - examples 8-11
 - diagnostics 8-11
- vector
 - conversion 7-7
 - multiplying matrix 7-15
 - obtaining 7-3
 - representing 7-2
- vecTrans() function 7-11
- velocity() function 4-18, 4-20
- VERS argument 3-14
- version, protocol
 - receiving 5-11
 - setting in PDU 5-9
- view 4-27
- vllnc.h* header file 9-2
- vlList.h* header file B-777
- vlprint.h* header file B-779
- vlTime.h* header file B-780
- vlutil.h* header file B-781

vlutilInc.h header file 9-2
vlVector.h header file B-784
vrIFedAmb.h header file B-786
vrIFedTime.h header file B-790
VR-Link
 Daemon Controller. *See* vdc
VR-Link
 daemon. *See* packet server
 defined 1-1
 features summary 1-2
 files 2-4
 previous versions 3-10
 simulation time 3-6, 3-9, 4-20, 4-28, 7-17
VR-Link simulation time 3-6, 4-20
vrlinkd. *See* packet server
vrlinkd3i
 external mode 8-19
 packet server 8-12

vrlinkdu
 packet server 8-12
VR-Link.fed file 1-3, 4-2
vrIRtiAmb.h header file B-793

W

WeaponFire class 6-17
wrapNext() function 4-16
wrapPrev() function 4-16
writeFd() function 5-27
writes
 non-blocking 5-27

X-Y-Z

zero() function 7-3



MÄK Technologies

*185 Alewife Brook Parkway
Cambridge, MA 02138
617.876.8085*