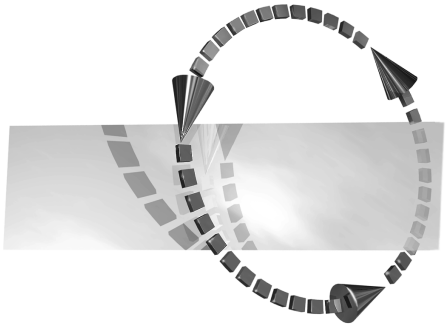

# VR-Link Developer's Guide Addendum B

This is the second addendum to *VR-Link Developer's Guide*, version VRL-980223a. It supports MÄK VR-Link release 3.3. This addendum covers the following topics:





Document ID: VRL-3.3-1B-981015

# *Overview of VR-Link 3.3*

VR-Link Release 3.3 offers the following changes and new features:

- VR-Link 3.3 supports RTI version 1.3 and RPR FOM version 0.5. (VR-Link 3.2 supported RTI version 1.0.3 and RPR FOM 0.3). Specifically, we have tested VR-Link 3.3 with version 1.3 of the MÄK RTI, and version 1.3v4 of the DMSO RTI.
- VR-Link 3.3 supports many more RPR FOM object and interaction classes (though it still does not support the entire RPR FOM), allows more configurability of HLA behavior, and contains numerous bug fixes.
- VR-Link now comes with a copy of the MÄK RTI.
- There were very few changes in the DIS portion of VR-Link.
- The on-line version of the *VR-Link Developer's Guide* contains class hierarchy diagrams and tables.

## *System Requirements*

Table 1 lists the platforms currently supported by MÄK VR-Link 3.3. Application code must be built with the indicated compilers in order to link to VR-Link libraries.

Table 1: Platforms supported

| Platform | Compiler |
|---|---|
| SGI IRIX6.2 or later (both n32 and o32 ABIs) | MipsPro7.1 C++ compiler (available from SGI) |
| Sun Sparcstation with SunOS4.1.3 or later | GNU C++ (g++) version 2.7.2.3 and G++ libraries version 2.7.2. (Available via anonymous FTP from prep.ai.mit.edu/pub/gnu.) |
| Sun Sparcstation with Solaris2.5 or later | SPARCompiler C++ version 4.1 (available from Sun) |
| Dec Alpha with OSF/1 V4.0 | DEC C++ for OSF/1 version 5.5 (available from DEC) |
| PC with Red Hat Linux 5.1 | GNU C++ (g++) distributed with Red Hat distribution. The command: `g++ -v` returns: `gcc version egcs-2.90.27 980315 (ecgs-1.02 release)` |
| PC with Windows NT or Windows 95 | Microsoft Visual C++ 5.0 or later.<br>Note: MÄK VR-Link 3.2 used MSVC++ 4.2. |

**Note:** When you use VR-Link and the MÄK RTI on the SunOS4 platform, the link order needs to be a little different from most platforms. Use:

```
-lvlRPR -lRTI -lmatrix -lmtl -lvlutil
```

RTI must precede mtl and vlutil. The VR-Link example Makefiles reflect this requirement.

# The MÄK RTI

All MÄK products, including VR-Link, now come with a copy of the MÄK RTI. On UNIX systems, if you accept the prompt to install the MÄK RTI as part of the installation procedure, there should be a link called *RTI* in the VR-Link root directory that points to the *makRti* directory, which contains the MÄK RTI software. On PCs, the MÄK RTI is installed in a default directory.

On UNIX, VR-Link example makefiles are now set up to build against either the DMSO RTI or the MÄK RTI. If you've sourced the DMSO RTI *rti.env* file, we'll use the DMSO RTI, otherwise the MÄK RTI, which we get to through the *RTI* link in the VR-Link root directory.

On PCs, the VR-Link example makefiles are set up to build against the MÄK RTI and assume that it is in the default directory. If you want to use other build settings, you must change them as described in Chapter 2, "Toolkit Overview" of the *VR-Link Developer's Guide* (Chapter 3, VR-Link Concepts in the on-line version of the manual.)

See MÄK RTI documentation for information about setting environment variables and other configuration information required to use the MÄK RTI.

You do not need an RTI server or central application such as *rtiexec* to use the MÄK RTI. Just start the applications, and they should be able to communicate. See MÄK RTI documentation for information about changing the networking parameters such as multicast addresses, UDP ports, and the rtiexec (optional).

**Note:** As with all MÄK products, make sure the license manager is running before you try to use the MÄK RTI.

# Backward Compatibility

Mostly because of the object-ID versus object-name changes described in "Object Id Versus Object Name" on page 5, HLA applications built against VR-Link 3.2 will not build against VR-Link 3.3 without being changed. However, it should not be a major effort to port. See the sections under "New Features" on page 5, to find out what has changed in the API. Many DIS applications will not require porting at all.

## Network Compatibility

***HLA only***

HLA applications based on VR-Link 3.3 cannot communicate with VR-Link-3.2-based HLA applications, because the HLA version of VR-Link 3.3 uses a new version of the RPR FOM (0.5), and a new version of the RTI (1.3).

***DIS only***

The DIS portion of VR-Link 3.3 is network compatible with previous VR-Link releases, as they use the same versions of the DIS protocol.

# New Features

The following sections describe the new and enhanced features in VR-Link 3.3.

## Object Id Versus Object Name

A major conceptual change took place in the HLA Interface Specification between version 1.1 and 1.3, and this change was reflected in RTI 1.3.

In RTI1.0.3, a particular object was known to all federates by the same unique number, its ObjectID. In RTI1.3, this is no longer the case. Now, an object is known by two different identifiers:

- ObjectHandle - a numerical value by which a particular object is known to a particular federate and that federate's Local RTI Component
- ObjectName - a character string by which a particular object is known to all federates

For example, federate A might decide to publish an object that it names "Object1". Upon registering this object with the RTI, the RTI tells federate A that it can refer to this object by handle number 25. When this object is discovered remotely by federate B, the object's name is still "Object1", but federate B's RTI might tell federate B that it can refer to this object by handle number 30.

This means that the RPR FOM's (and many other FOMs') scheme of identifying HLA objects within attributes and parameters by its ObjectID number is no longer workable. (If federate A sent a fire interaction indicating that object number 25 fired at someone, federate B would not know who he was talking about, since he knows that object as object number 30.) As a result, in RPR FOM 0.5, (which was meant to work with RTI1.3) we need to refer to objects by their ObjectName character string instead.

While we as VR-Link developers have tried to make this major RTI change as seamless as possible, it did require changes to the VR-Link API.

### Object IDs and Object Designators

The VR-Link class *DtObjectId*, which used to be a wrapper around *RTI::ObjectID* in HLA, is now a wrapper around *RTI::ObjectHandle*. The object publishers and reflected objects' id() and objectId() functions still return *DtObjectId*s. However, you can no longer use *DtObjectId*s in contexts where you are talking about data you wish to send to or receive from other federates.

A new type, *DtGlobalObjectDesignator* is used to communicate about objects with other federates. For example, DtFireInteraction::setAttackerId() now takes a *DtGlobalObjectDesignator* rather than a *DtObjectId*. *DtGlobalObjectDesignator* is currently typedefed to *DtString* (a VR-Link string type) in the HLA version of VR-Link, and to D*tEntityIdentifier* in the DIS version of VR-Link (since *DtEntityIdentifier* is the type that is used to identify objects within DIS PDUs).

### Publishers and Reflected Objects

The publishers and reflected objects have new name() and globalId() member functions that return the object name. This enables statements like the following:

```
fireInteraction.setAttackerId(ownshipPub-
lisher.globalId());
```

When constructing an HLA publisher, you can either choose a name for your object, or pass a NULL name (the default), and let the RTI choose for you. For example:

```
DtEntityPublisher(someEntityType, someExConn, "Object1");
```

or simply,

```
DtEntityPublisher(someEntityType, someExConn);
```

The reflected object lists have two lookup() functions - one that takes a *DtObjectId*, and one that takes a *DtGlobalObjectDesignator*.

The HLA version of *DtExerciseConn* has lost its nextId() member, since you can no longer choose the numerical ID for your object.

### Event IDs

The *DtEventId* class has changed. Its objectId() and setObjectId() functions now deal with *DtGlobalObjectDesignator*s rather than *DtObjectId*s. The class has moved to its own header file: *eventId.h*, which is included by *hostsStructs.h*.

Also, *DtExericseConn*'s setObjectIdForEventId() takes a *DtGlobalObjectDesignator* now.

## Other Changes for RTI 1.3

### FED File Format

A new *.fed* file format was introduced in RTI 1.3. The standard *VR-Link.fed* file was updated to be usable with RTI 1.3.

### Specifying a FED file

A new HLA *DtExerciseConn* constructor allows you to specify a FED file name in addition to the federation execution name. If you use the older constructor, we still assume a FED file name of <*exeuctionName*>*.fed*.

### Fully Qualified Names

If you are passing HLA Object or Interaction class names to VR-Link or RTI functions, you must now use names that are "fully qualified" by base class names. For example, use "BaseEntity.PhysicalEntity.MilitaryEntity.MilitaryPlatformEntity" rather than just "MilitaryPlatformEntity". This change was made so that RTIs could support multiple classes with the same name, as long as they had a different ancestry.

### DtFom

*DtFom* has the following new member functions for iterating through all of its object and interaction classes:

- firstObjClass()
- lastObjClass()
- nextObjClass()
- firstInterClass()
- lastInterClass()
- nextInterClass()

These allow the following kind of loop:

```
for (DtObjClassDesc* desc = fom.firstInterClass;
desc; desc = fom.nextInterClass(desc))
```

The old mechanism:

```
DtObjClassDesc desc = NULL;
for (RTI::ObjectHandle hand = 1;
   desc = fom.objClassByHandle(hand); hand++)
```

made too many assumptions about the RTI's handle assignment algorithm.

### Timestamps and Tags

You can now set and inspect an HLA *DtStateMsg* or *DtInteraction*'s time stamp with their setTimeStamp() and timeStamp() functions. These work the same way as the DIS *DtPdu*'s function - taking a time, and a time stamp type (either *DtTimeStampRelative* or *DtTimeStampAbsolute*).

DtExerciseConn::sendStamped() now sets the timestamp of an outgoing HLA message to the current time, with the type returned by DtExerciseConn::timeStampType().

According to the RPR FOM convention, our sending functions encode the time in the "tag" string argument to the RTI's sendInteraction() and updateAttributeValues() calls. The time is encoded as the 8-byte ASCII representation of the hex number that would have been sent in a DIS timestamp.

If you do not want us to use this convention, use DtInteraction::setTagIsAscii-Time(DtFalse); and DtStateMsg::setTagIsAsciiTime(DtFalse);

If you do this, your timestamps will not be sent at all, since the RTI does not provide a mechanism for sending timestamps along with your messages unless you are using RTI time management, which real-time simulations typically do not do.

### Federation Time

In RTI 1.3, RTI users have the flexibility to choose their own representation of federation time for use by time management services. Unfortunately, even if you are not using RTI time-management services, (and most VR-Link-based applications do not), RTI users must tell the RTI how they would like federation time to be represented in case they use a time management service.

The mechanism for communicating this information to the RTI is by subclassing the abstract base class *RTI::FedTime*, and telling the RTI to use the subclass by providing implementations for the functions RTI::FedTimeFactory::decode() and RTI::FedTimeFactory::makeZero(). (These functions are declared, but not defined by the RTI's API.)

Both the DMSO RTI and MÄK RTI provide an extra library called *libfedtime.so* that satisfies this RTI requirement, however to avoid forcing all applications to link with this extra library, we have provided an *RTI::FedTime* subclass, and definitions for the required function, within VR-Link. The subclass is called *DtVrlFedTime*, and is defined in *vrlFedTime.h*.

## More RPR FOM Classes

VR-Link 3.3 supports many more RPR FOM classes than VR-Link 3.2, though we still do not support the entire RPR FOM.

### Interaction Classes

We now support the following interaction classes, in addition to WeaponFire, MunitionDetonation and Collision, which were supported in VR-Link 3.2:

- Acknowledge
- ActionRequest
- RadioSignal and its four subclasses
- StartResume
- StopFreeze

The new classes derived from *DtInteraction* are:

- *DtAcknowledgeInteraction* (in *hAckInter.h*)
- *DtActionRequestInteraction* (in *hActReqInter.h*)
- *DtSignalInteraction* (in *hSignalInter.h*)
- *DtStartResumeInteraction* (in *hStartInter.h*)
- *DtStopFreezeInteraction* (in *hStopInter.h*)

These interaction names can be used in DIS too, for example, *DtAcknowledgeInteraction* is now typedefed to *DtAcknowledgePdu* in DIS.

If you are writing code for both DIS and HLA, you can include these new header files, instead of having to include both DIS and HLA versions of class's header files (for example, both *startPdu.h* and *hStartInter.h*):

- *ackInter.h*
- *actReqInter.h*
- *signalInter.h*
- *startInter.h*
- *stopInter.h*

## Object Classes

The HLA version of VR-Link 3.3 supports many more types of objects than VR-Link 3.2. We now support all RPR FOM object classes except *SimulationManager*.

To assist in the implementation of this new functionality, we created several new base classes:

- *DtObjectPublisher* (in *hObjPub.h*)
- *DtReflectedObject* (in *hReflectedObj.h*)
- *DtReflectedObjectList* (in *hRefObjList.h*).

These classes encapsulate the general concept of a publisher, reflected object, and reflected object list, without being tied to a specific type of object.

In addition to the new derived publishers, reflected objects, and reflected object lists we'll describe below, the ones that existed in 3.2 in the absence of these base classes (*DtEntityPublisher*, *DtAggregatePublisher, DtReflectedEntity, DtReflectedAggregate, DtReflectedEntityList, DtReflectedAggregateList*) are now derived from the appropriate base classes. This change did not affect the API of the existing classes, however, many member functions that used to be declared explicitly as members of, for example, *DtEntityPublisher*, are now inherited from the base *DtObjectPublisher*.

The new classes that support the additional RPR FOM functionality are as follows:

- For designator objects
  - *DtDesignatorRepository* (in *designatorSR.h*)
  - *DtDesignatorPublisher* (in *hDesignatrPub.h*)
  - *DtReflectedDesignator* (in *hReflDesigntr.h*)
  - *DtReflectedDesignatorList* (in *hRefDesigList.h*)

- For radio receiver objects
  - *DtRadioReceiverRepository* (in *radioRecvrSR.h*)
  - *DtRadioReceiverPublisher* (in *hRadioRcvrPub.h)*
  - *DtReflectedRadioReceiver* (in *hReflRadRecvr.h*)
  - *DtReflectedRadioReceiverList* (in *hRefRadRvList.h*)
- For radio transmitter objects
  - *DtRadioTransmitterRepository* (in *radioXmitSR.h*)
  - *DtRadioTransmitterPublisher* (in *hRadioXmitPub.h*)
  - *DtReflectedRadioTransmitter* (in *hReflRadXmitr.h*)
  - *DtReflectedRadioTransmitterList* (in *hRefRadXmList.h*)

All of these objects work just like the corresponding entity classes in VR-Link 3.2 as shown in the following procedures and examples.

#### Publishing Objects

1. When simulating an object locally, create the appropriate publisher.
2. Obtain a pointer to the publisher's state repository using the appropriate member function.
3. Use the state repository's mutators to set the values of the various attributes.
4. Use the publisher's tick() function to have VR-Link send necessary data to the exercise. For example:

```
DtDesignatorPublisher desPub(exConn);
DtDesignatorRepository* dsr = desPub.dsr();
// main loop
while(...)
{
   ...
   dsr->setPower(...);
   desPub.tick();
   ...
}
```

### Managing Reflected Objects

1. Create the appropriate reflected object list to manage the set of reflected objects of a particular type. It will automatically be notified by the RTI of relevant objects when they come and go, and when attribute updates are received for those objects, from within DtExerciseConn::drainInput().
2. Use the reflected object list's first() and the reflected object's next() to iterate through the list of objects that currently exist in the exercise.
3. Obtain a pointer to a particular reflected object's state repository using the appropriate member function
4. Use the pointer to examine the current values of the attributes. For example:

```
DtReflectedDesignatorList desList(exConn);
...
// main loop
while(...)
{
   exConn.drainInput();
   ...
   for (DtReflectedDesignator* refDes = desList.first();
      refDes; refDes = refDes->next())
   {
      DtDesignatorRepository* dsr = refDes->dsr();
      float power = dsr->power();
      ...
   }
   ...
}
```

### Managing Emissions

Dealing with emissions is a little more complicated. The classes that you need to use are:

- *DtEmitterSystemPublisher* (in *hEmittrSysPub.h*)
- *DtReflectedEmitterSystemList* (in *hRefEmSysList.h*)
- *DtReflectedEmitterSystem* (in *hRefEmitrSys.h*)
- *DtEmitterSystemRepository* (in *emitterSysSR.h*)
- *DtEmitterBeamRepository* (in *emittrBeamSR.h*).

Although emitter systems and emitter beams are implemented as separate objects in the RPR FOM, we represent emitter beams as components of emitter systems in VR-Link. So although VR-Link has a publisher, reflected list, and reflected object for emitter beams, which we use internally, you should not have to use them. Beam data is set and inspected through the emitter system objects.

#### Publishing Emissions

On the outgoing side, suppose you are simulating an emitter system with two beams.

1. Create a *DtEmitterSystemPublisher*.
2. Obtain a pointer to the *DtEmitterSystemRepository*.
3. Use *DtEmitterSystemRepository*'s addBeam() member to add a beam to the system. This will return a pointer to a *DtEmitterBeamRepository* that you can use to set beam data. (To remove a beam from a system, use removeBeam()).
4. Call tick() on the *DtEmitterSystemPublisher*. We will send any necessary updates for the system object AND its constituent beam objects.

#### Managing Reflected Emissions

1. Create a *DtReflectedEmitterSystemList*.
2. Use the normal functions to iterate through the *DtReflectedEmitters*, and to obtain a pointer to each *DtReflectedEmitter*'s *DtEmitterSystemRepository*.
3. Use *DtEmitterSystemRepository*'s beamSRById() to obtain a pointer to a particular constituent *DtEmitterBeamRepository*, or use beamList() member to obtain a *DtList* of all of the constituent DtEmitterBeamRepositories. These can be used to inspect the beam data.

## Other API Changes

- We have added a general purpose class called *DtObjectIdHashlist* (defined in *objIdHashLst.h*) for storing and looking up arbitrary pieces of data using a *DtObjectId* as a key. *DtGlobalIdHashlist* (*globIdHLst.h*) works similarly, but uses a *DtGlobalObjectDesignator* as a key. Both of these classes are protocol-independent.
- You can now set flags that control whether or not we request attribute updates:
  - For all objects of each class that is relevant to a particular reflected object list when it is created
  - For a particular new object when it is discovered.

  Defaults are DtTrue for both flags, but they can be set and inspected with the following static members of *DtReflectedObjectList*:
  - setRequestClassUpdateFlag()
  - requestClassUpdateFlag()
  - setRequestObjectUpdateFlag()
  - requestObjectUpdateFlag()
- Timeout processing can now be controlled on a per reflected object list basis. setTimeoutProcessing() and setTimeoutInterval() are now non-static members of the base *DtReflectedObjectList*. They used to be static members of *DtReflectedEntityList* and *DtReflectedAggregateList*.

- *DtInterClassDesc* paramHandleSet() and *DtObjClassDesc's* attrHandleSet() are now const member functions. Also, ownParamHandles() and ownAttrHandles() were added. These functions return the set of parameters or attributes of the class that have not been inherited from a base class.
- *DtVrlRtiAmbassador* and *DtDirectRtiAmbassador*, VR-Link's wrappers around the RTI's *RTI::RTIambassador* class now have public rtiAmb accessors that return a pointer to the *RTI::RTIambassador*.
- Some of *DtPdu*'s functionality was split out into a more general *DtBaseMsg* base class. This change should be transparent.
- *DtTaitBryan* now has an equality operator.
- *DtExerciseConn* now has a flag that determines whether or not it will try to destroy the federation execution in its destructor. The default is true, and can be set with setDestroyFedExecFlag() or inspected with destroyFedExecFlag().
- *DtIntClassDesc* now has a deregisterInterest() which can be used to unsubscribe to an interaction class.
- We moved the default RPR-FOM knowledge from *DtFomMapper* into a *DtRpr-FomMapper* class. The base *DtFomMapper* can now be more easily subclassed to create a mapper that knows about other FOMs. *DtExerciseConn* now has a static setFomMapperCreator() function that can be used to specify a function that creates the kind of *DtFomMapper* that you would like to use. Call this function before creating a *DtExerciseConn*, and when you create the *DtExerciseConn*, it will use the type of FOM mapper that you specified.

### Hashlist

In VR-Link 3.3, we have changed the syntax and semantics of the *DtHashlist* class, which hopefully makes it easier to use. If you would like to continue to use the old *DtHashlist*, it still exists, though it has been renamed *DtOldHashlist*, and has been moved to *oldHashlist.h*. In order to use the old hashlist, simply replace *DtHashlist* with *DtOldHashlist* in your code, and things should still work. However, we strongly encourage you to move to the new hashlist class, because the old one will be removed from VR-Link in the future.

To use the new *DtHashlist*, rather than subclassing *DtHashlist* to define a hashing function:

1. Subclass *DtBaseHashKey*.
2. In your subclass, define a hashing function, and a function that can convert your key to a string representation, so that two keys can be compared without *DtHashlist* knowing anything about them.

We have provided some *DtBaseHashKey* (*baseHashKey.h*) subclasses that might be useful if you want to use integers or strings as a hash keys:

- *DtStringHashKey* (*strHashKey.h*)
- *DtIntHashKey* (*intHashKey.h*)

Here is an example of adding two objects to a hashlist, hashed by integers:

```
SomeClass* a;
SomeClass* b;

DtHashlist list;
list.add(DtIntHashKey(7), a);
list.add(DtIntHashKey(10), b);
```

and pulling them out again:

```
SomeClass* a = list.lookup(DtIntHashKey(7));
SomeClass* b = list.lookup(DtIntHashKey(10));
```

If you want to use *DtObjectIds* or *DtGlobalObjectDesignators* as hash keys, we have provided higher level classes (*DtObjectIdHashlist* and *DtGlobalIdHashlist*) that use *DtHashlists*, but eliminate the need to explicitly deal with *DtHashKeys*.

## Miscellaneous Enhancements

This section describes several smaller enhancements and changes to VR-Link.

- *HlaNetdump* now prints attribute and parameter names along with their handles.
- The *configRti* script that is used to configure the DMSO RTI directory no longer modifies the *rti.env* environment file. It now merely copies the *VR-Link.fed* file into the RTI's config directory.

## Updated VR-Link Developer's Guide

The electronic version (*VRL3.3-1-981012.pdf*, in Adobe Acrobat 3.0 format) of the *VR-Link Developer's Guide* provided with VR-Link 3.3 has been updated to include new and revised header files. Chapter 9, Guide to Classes and Header Files (Chapter 8 in the printed manual) has been updated with class hierarchy diagrams and tables. There are also miscellaneous corrections and clarifications in the rest of the manual. However, the body has not been updated with the information in this addendum to the manual or Addendum A.

The complete printed documentation for VR-Link 3.3 is:

- *VR-Link Developer's Guide*, Revision VRL-980223a
- *VR-Link Developer's Guide, Addendum A*
- This addendum

**Note:** You must have Adobe Acrobat Reader 3.0 to view the online documentation.

### Documentation Updates

The following errors in the *VR-Link Developer's Guide* have been corrected in the on-line version of the manual:

- On page 3-8, the last line of the example should be:

```
DtFireInteraction::addCallback(&ExerciseConn, \
MyObj::theCallback, &obj):
```

- On page 3-16, the last line should be:

```
printf("ID: %s\n", firstEnt->entityId().string());
```

## Bug Fixes

- On little endian machines, nethex and netdump (in cases like the data portion of a signal PDU) no longer byte-swap raw hex words.
- Fixed a bug whereby in HLA, we would fail to send an update if attached part data was out of data. (This caused anomalies in the number of updates sent by our launcher example.)
- DtEntityStateRepository::setSmokePlumePresent() works properly now.
- DtEntityStatePdu::printData() no longer leaves out **FinalPdu** and **PaintScheme** flags.
- Fixed problem of redefining #defines when you include *vlInc.h*.
- Fixed a bug that could cause crashes when using timeouts in HLA.
- The HLA version of *DtReflectedEntityList* now correctly uses smoothing when constructed with the **smoothCapable** parameter set to DtTrue.

## Known Problems

- When using the O32 version of VR-Link on IRIX6 with the MÄK RTI, applications may hang while trying to clean up before exiting. You may need to press CTRL-C, or kill such applications (with the `kill` command).
- There is a bug in the DtString::operator+ that causes it to act like DtString::operator+=. That is, it modifies the argument to the left of the plus sign.
- When you configure the Flexlm license manager on a PC, make sure that when you type pathnames, the drive letter is uppercase.