Development

Environment

Controller

0 Preface

Development Environment Controller (DEC) is my attempt to provide a first pass at a cooperative code development environment for LCD collaborators. It is by no means a polished, elegant solution, but I believe it incorporates the essential elements required.

Why DEC? My first choice was CDE (Cooperative Development Environment), but that was already taken by Sun’s Common Desktop Environment. At about the same time I was getting very frustrated at the Unix and NT environments (cries of ‘I want my VMS’). Digital had by that time already been bought by Compaq so I figured that if Digital Equipment Corporation no longer needed the acronym (and to keep the name in front of a computer savvy community)…

It’s always difficult to write a document like this because the readership will range from ‘newbies’ who have never encountered code management before, to jaded old hands who are faced with the prospect of learning yet another environment. My solution is to divide the document into a first section (Introduction) covering the principles and practice of code management, a second section (DEC environment) outlining the characteristics of this implementation, and subsequent, terser sections giving specifics of various commands. Experienced developers will probably want to skip the ‘Introduction’ but should not skip ‘DEC environment’. To make it easy to go directly to ‘DEC environment’, I give here the executive summary of the ‘Introduction’:

- Source file management is accomplished using the CVS application as described in section ‘CVS’.
- Building is controlled with make files and the GNU make application (always referred to as `gmake` in this document despite the fact that the executable is called `make` in Windows/NT).

- There is a well-defined software life cycle whose structure is described in section ‘DEC environment’.
- The environment is supported by a small number of commands that provide facilities to manage individual user sessions, build releases, etc. These are covered, either in individual sections for the more complicated commands, or in a ‘miscellany’ section covering a number of simple commands.

1 Introduction

Any sufficiently large and/or long-lived software project eventually runs into the problem of organization. If the code moves out of an individual developer's file space into a shared environment, how should incompatible edits of the shared code be handled? How can edits be tracked back to the editor? If code is used to generate long-lived datasets, how can the code that generated a particular dataset be traced? If the code is run on multiple target platforms, how can an exact source code match be ensured between the executables?

This is where code management comes in. Code management provides a structured working environment that facilitates cooperative code development. Of course, such environments always come at the cost of rules, anathema to the free spirited code developer! A good code management scheme should therefore be as light as possible to encourage compliance while being strong enough to achieve its aims. Around such a seemingly commonsensical statement revolve passionate debates about the relative merits of various code management implementations!

Despite these debates, the fundamental elements of code management are not in (much) dispute. My personal list would read:

- Source file management/tracking.
- Module management/building.
- Software life-cycle definition.
- The 'glue' to make a coherent whole of the above.

If the fundamentals are not in dispute, why not simply adopt an existing implementation? Surprisingly, not many general purpose implementations exist. Add the constraints peculiar to our environment…

- The code must run on a variety of platforms (Windows/NT and a variety of Unices including AIX, Sun, Linux, DEC,?)
- Must be cheap (where cheap ~ free).

… and I don't know of any!

The last constraint is the most demanding. There is one commercial product which looks very interesting. It's called SNiFF+ (their capitalization, not mine!) SLAC actually has a copy of this code and a bunch of licenses which are apparently going unused. If we ever achieve a higher status either within SLAC or as widespread collaboration, I would be interested to explore this product.

Looking at each member of my list of 'fundamental elements of code management' in turn…

1.1 Source File Management/Tracking.

1.1.1 What Is It?

Source file tracking is the process of keeping a complete record of the contents of source files as they evolve over time, along with information about who changed them and when. Developers check out and check in files and each checked in file generates a new version. All historical versions are kept available. This is achieved by an application which maintains a library or repository. Different implementations of this application offer different feature sets, but the most useful features include:

- File check out and check in (with transaction logging to identify who and when!) Some implementations additionally provide file locking to ensure serial file editing.
- Defining lists of files within the library. Some achieve this with arbitrarily named lists, others use a directory structure. Note that this does not specify a version of a file. If file foo.c belongs in list/directory bar, then foo.c version 1, foo.c version 2, foo.c version 3,… belong in list/directory bar.
- Defining lists of file/versions within the library. In this variation, the file version for each file *is* recorded with the list. This is invaluable for release control where, for instance, all the most recent versions of the library files might be recorded with the tag string `Version 1.01c`. From then on, this list of files can always be recovered using the tag string.

Note that only source files are tracked. Object files, libraries, shareables, executables, etc. are regarded as derived objects such that provided the correct set of source code can be identified, the derived object is only a compile (or link or whatever) away. Given the rate at which compiler/linker technology is changing, that may be a little optimistic!

The upshot is that there is one central location where the ‘official’ source code is stored. Development proceeds by users checking out source files into private areas, modifying them, testing the changes (please don’t forget this step!) then checking them back in.

1.1.2 What’s Implemented?

The only real contenders were SourceSafe (part of the Microsoft Visual Studio package) and CVS (a free product distributed under GNU license rules). Of the two I prefer SourceSafe, both philosophically (it has the capacity to enforce serial editing of files for instance) and aesthetically (it comes with a pretty GUI interface, though it can also be scripted). The downside is that there are few tools (that I know of) to make it viable as a code repository for Unix and of course, it costs money. CVS on the other hand is free, is in widespread use and already works in the Windows/NT world. Such is life.

Source file management/tracking has therefore been implemented in CVS. The repository resides in the SLAC NFS file space (not AFS for a variety of very frustrating technical reasons which I don’t want to get into here!). It is accessible from SLAC Unix boxes using the standard CVS command line. It is accessible from Windows/NT using a client/server version of CVS (which comes with a quite reasonable, and free, GUI interface).

Despite some reservations I have about CVS, it has solved one problem very well. Plain text files on Unix use a simple CR (carriage return) to end a line. Windows/NT uses CR/LF (carriage return followed by a linefeed). Files in the repository are stored Unix fashion, but if a file is checked out to Windows/NT using client/server CVS, all CR are replaced with CR/LF. The process is reversed when the file is checked back in.

Why bother to mention such a technical point? Because it has a side effect that might come back and bite you. Suppose you are working on a source file which is not a plain text file (a graphical icon for instance) and that file is used on both Unix and NT. You do not want CVS to start automatically editing what is essentially a binary file! CVS recognizes this and deals with it by marking all files as either ‘text’ or ‘binary’ as specified by the user, which makes you responsible for ensuring that files end up with the right tag!

1.2 Module Management/Building.

1.2.1 What Is It?

Module management/building is the process of creating and maintaining a picture of how sources are transformed into executable code and provides the tools to take the source code and coordinate the transformation. The standard methodology these days is a ‘make’ file. The make file lists the file dependencies (a source file depends on its included files, an object file depends on its source file, an object library depends on a list of object files, and so on). The make file also contains a set of rules which tells it how to convert dependent files into the target file. Thus there is a rule that says if an object file depends on a C language source file, the correct procedure is to use the C compiler to compile the source file.

The extra wrinkle that makes make files so popular is that they do the minimum amount of work. Before performing any transformation, the make application examines the datestamps on the dependent files and the target file. If all dependent files have an older datestamp than the target file, then the make application assumes that the target is up-to-date and does not execute the transformation. Sounds easy right? If you believe that then you have clearly never tangled with real world make files!

1.2.2 What’s Implemented.

Microsoft Visual Studio boasts ‘project management’ which is really just the Microsoft make application `nmake` under the hood, but with the advantage that the make file is auto-generated. Unfortunately, I have never found a satisfactory method for making this work in a multi-developer, multi-platform environment. One problem is the Visual Studio’s project file which two users can modify incompatibly with no real hope of merging their changes. Another is the lack of any support for Unix.

Almost all Unix platforms provide a vendor written make application. Each implementation is of course just slightly different and it’s easy to go mad trying to identify the lowest common denominator set of syntax.

Fortunately GNU provides a make application which is consistent across platforms (including Windows/NT). To distinguish it from the vendor supplied `make`, it is common practice on Unix to make this application available as the command `gmake`. On Windows /NT the vendor supplied make is *not* called `make`, so the GNU make application on Windows/NT *is* called `make` (confused yet?). From now on I shall refer to the GNU make application as `gmake` whatever the platform.

The solution implemented for module building is to use `gmake` across all platforms. The user is presented with a `gmake` template file where details of the target library/shareable/executable are filled in and the ‘system’ takes care of the rest, making the necessary adjustments for platform. Notice that there is only one `gmake` template file per module and it works on all target platforms.

In truth, some platform dependency has leaked back into the user template file, though in what I hope is a very controlled fashion. As a result I’m not completely satisfied with the current implementation, so please consider it a work in progress!

1.3 Software Life Cycle.

1.3.1 What Is It?

I’ve always felt that the importance of the software life cycle has been underestimated. It’s the combination of a well defined software life cycle and good code segmentation that’s the key to a flexible developer environment.

The software life cycle acknowledges and formalizes the fact that software goes through a number of stages from its initial creation in somebody’s private workspace to a fully debugged, reliable element in some publicly available application.

In commercial environments the life cycle can include five, six, seven or even more stages including requirements planning, functional definition, quality control, hardware crossbar tests, public beta releases, rework cycles and so on. There are usually formal procedures defining the conditions under which code can move from stage to stage. In other words, a whole heap of mechanism to track progress and keep managers happy (good for assessment-of-blame too!)

1.3.2 What’s Implemented?

If you think that last paragraph demonstrates the dead hand of bureaucracy then I’m right with you! It’s precisely that kind of approach that gets code management a bad name among developers. The DEC environment has just three stages:

1. Test. This all occurs in someone’s private workspace. The code could be either newly created or some existing code which the user has checked out for maintenance or upgrading. Activities in a private space have no impact on any other user, so the developer can run through the edit/compile/link/test cycle in perfect isolation.

2. Development. Once a developer is satisfied that the code has achieved sufficient stability it can be moved into development. Here it becomes

available for other developers to link against and thus gets exposed to a more demanding environment.

3. Production. If the code survives the testing in development, it becomes a candidate for transfer to production where it becomes available to the great unwashed public. This is most demanding environment of all!

There's an aphorism I've always been very fond of but never known the attribution. Can anyone help me out?

No system is foolproof. Fools are too ingenious.

Note that this is the 'fundamental element' of code management least well served by pre-existing solutions. Much of the coding I have done for the DEC environment is in support of the software life cycle and of dovetailing it with the other code management elements.

1.4 Glue.

1.4.1 What Is It?

Source file tracking, module management and a software life cycle definition are all important elements in a cooperative development environment. Each element can be tackled individually with available tools (`CVS`, `gmake`, etc.). What's missing is a coherent environment which glues together all the disparate parts and makes life convenient for the developer. Let's face it, if it's not convenient, developers will resist using it! What's required are a few tools that know about `CVS`, `gmake` and the software life cycle. A good set of tools can actually improve the developer's effectiveness by providing an individually tailored environment for each developer and by automating away some of the code management drudgery.

1.4.2 What's Implemented.

The tools are implemented as a number of procedures invoked from the command line. With rare, platform specific exceptions, they are all REXX scripts. In the age of Unix/perl, many people have questioned my choice of REXX, so let's get that out of the way right now. REXX is a very capable scripting language. Used with (commercial) extensions to make it object oriented and GUI capable, I have even seen it touted as a competitor for Java (OK so that's probably just wishful thinking on the part of REXX advocates!). It's also the scripting language I know best. In an unfamiliar Unix/NT world, I didn't want to take on another unknown in the form of perl. The only downside has been the instability of the available interpreters. However, the latest version of Regina REXX (a free product) seems perfectly stable and runs on all the platforms I've tried so far (AIX, Linux, NT, OSF1, SunOS).

The tools range in sophistication from the simple `pathlist` (which will parse an environment variable like `PATH` into its components and present the result as a readable list) up to `dec` (which is used to control a user environment) and `build` (which is used to build test, development and production releases). The last two are complicated enough to merit their own sections later on in this manual.

2 The DEC Environment

As if it wasn’t overloaded enough already, I seem to use the acronym DEC in two different ways. First there is DEC ‘the environment’ and then there is DEC ‘the command’. To make the distinction, I am devoting separate sections to each.

The DEC environment is an overlay on top of a standard terminal session (Unix) or MS/DOS session (NT). It consists of a small number of commands and setup files designed to achieve the following:

- To make the environment as similar as possible across platforms.
- To provide an independent and flexible user environment while maintaining a single body of source code.

To an old VMS hand, these design goals scream ‘logical names’ which of course are not available in either Unix or NT. Implementing a completely abstract VMS logical name system would be extremely time consuming and in the end be very frustrating because the rest of the tools (compilers and the like) don’t recognize them. The DEC environment is a kind of halfway house. Think of it as ‘logical names lite’!

The DEC environment makes a clean distinction between the ‘internal’ file space (the files and directories directly under our control) and the ‘external’ file space (the files and directories from which we import, e.g. CLHEP). The logical naming structure for our internal file space is far richer than for the external file space.

In practical terms, this is achieved by maintaining a number of environment variables. These are manipulated using the command `dec`.

2.1 Syntax of DEC Environment Commands

The first problem was to sort out a consistent command interface. Unix makes heavy use of the hyphen to introduce parameters or options, though ambiguities with that syntax have led to oddities like the double hyphen and so on. The Unix command line is also case sensitive. NT on the other hand introduces options with a `/` and is case insensitive. There didn't seem to be much chance of reconciling these two so originally I said 'a plague on both your houses' and developed a VMS lookalike syntax. This turned out to be such a pain to use at the Unix command line (Unix has usurped so many special characters for its own use) that I finally gave in and went with a more Unix style approach. People familiar with the VMS syntax should read the following 'executive summary' and then go to the section 'Syntactic Shortfall'. Other readers should go to 'Syntactic Details'.

- Where VMS used a `/` to introduce qualifiers, use `--`.
- Where VMS put parentheses around qualifier lists, omit the parentheses.

2.1.1 Syntactic Details

DEC environment commands expect a verb plus zero or more *parameters* plus zero or more *qualifiers*. The following is a valid (and working!) command:

```
rexxpand builder.rexxproc builder.rexx-–noquiet--make=builder.rdp
```

Verb	`rexxpand`
First parameter	`builder.rexxproc`
Second parameter	`builder.rexx`
First qualifier	`--noquiet`
Second qualifier	`--make=build.rdp`

To give you a sense of what's going on, this means 'expand the input file `builder.rexxproc` to the output file `builder.rexx`. Print informational messages and also generate the secondary file `builder.rdp`.'

Parameters are blank delimited and positional. Trailing parameters can be omitted if that's allowed by the specific command syntax, but all parameters preceding a required parameter must be present. If a parameter includes a blank, then the whole string must be enclosed in double quotes:

```
rexxpand builder.rexxproc "blanks in file name.rexx"
```

Qualifiers are introduced with `--` immediately followed by a keyword (no intervening blanks). Qualifiers can be 'valued' (like `-–make` in the above example) or 'boolean' (like `--noquiet`).

Valued qualifiers expect = immediately following the keyword (again, no intervening blanks). The value assigned to the qualifier is the string following the

= up to the next terminator (a blank, another qualifier or the end of line). To include blanks in the value, enclose the string in double quotes:

```
rexxpand builder.rexxproc builder.rexx--make="blank here.rdp"
```

Boolean qualifiers sometimes allow negation by prepending `NO` to the keyword. Technically speaking the ‘recognized’ keyword in `--noquiet` is `quiet`.

Qualifiers can appear anywhere on the line and can follow a parameter or preceding qualifier without an intervening blank (because the introductory `--` implicitly terminates the previous parameter/qualifier). The following is perfectly legal (if a little strange!):

```
rexxpand builder.rexxproc--make=builder.rdp builder.rexx--noquiet
```

Qualifier keywords can be shortened to their minimum unambiguous length:

```
rexxpand builder.rexxproc builder.rexx—-noq--m=builder.rdp
```

Parameters can be shortened in a similar way. This can’t be demonstrated with the `rexxpand` example because the parameters are file names. The `dec` command on the other hand accepts the following subcommands:

```
set external <other parameters>
show external
show environment
```

These can be shortened to:

```
se ex <other parameters>
sh ex
sh en
```

In rare instances, DEC commands accept a list as the value of a parameter or a qualifier. For example:

```
pathlist path,include
```

In response, `pathlist` will parse both of the list variables `PATH` and `INCLUDE`. Members of a parameter list are comma delimited with no blanks either side of the comma. Members containing a blank must be *individually* double quoted.

The syntax for a qualifier list is similar:

```
<verb> <param>--<key>=<mem_0>,<mem_1>,<mem_2>
```

2.1.2 Syntactic Shortfall.

This is close to the VMS command line interface, but there are differences:

1. Because the verb is used as a file name to look up the command, the verb cannot be shortened and must be followed by a blank. On Unix it must also be correctly cased.
2. Both Unix and NT do some command line processing before the script gets to see it. NT likes to ‘eat’ double quotes, so the command:

   ```
   rexxpand build.rexxproc "name with blanks.rexx"
   ```

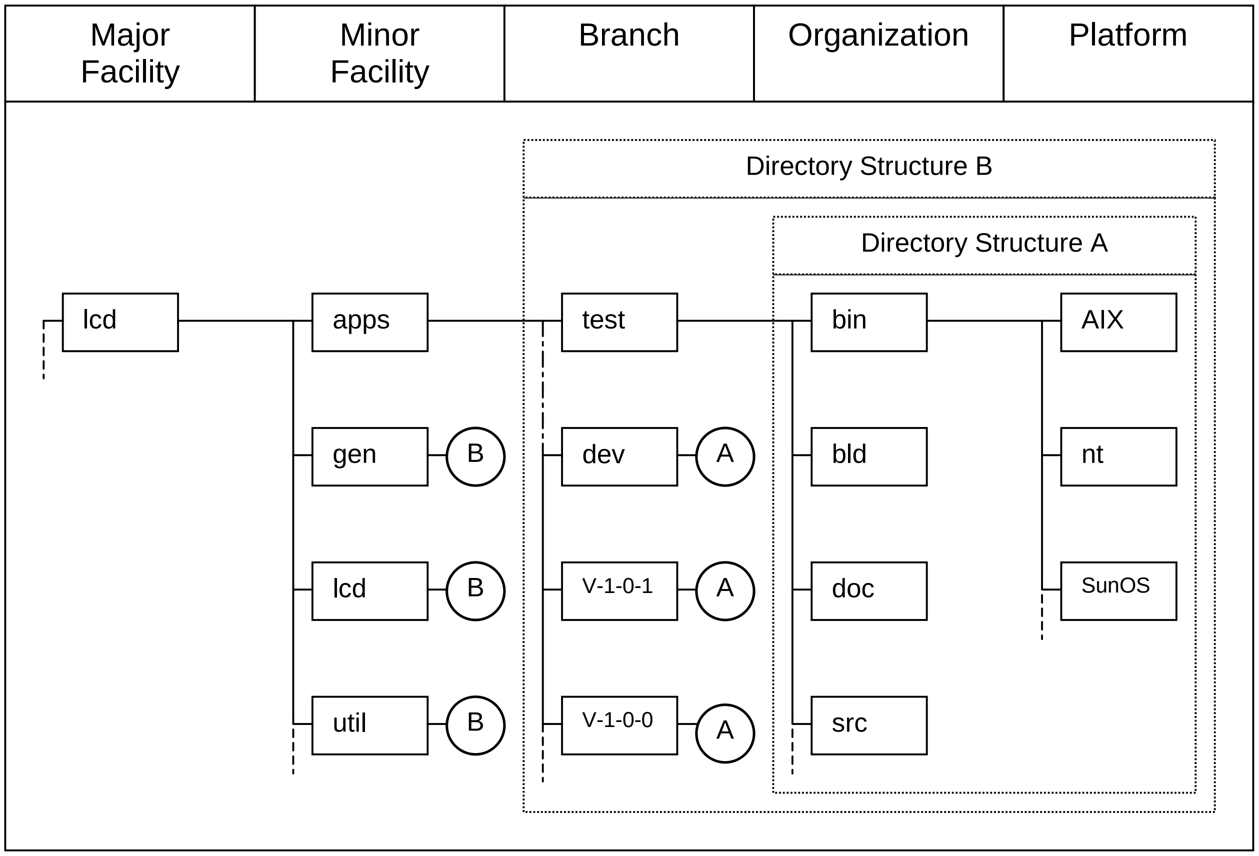
   has to be written:

   ```
   rexxpand build.rexxproc """name with blanks.rexx"""
   ```

   There is no end to Unix preprocessing, so I shan’t try to tabulate it all. I’m hopeful that the latest iteration of this command line interface will substantially reduce the likelihood of colliding with this problem.
3. Wherever possible, the DEC environment is case insensitive so parameters and qualifiers can be written in upper, lower or even mixed case. The exception is Unix filenames which are case sensitive. The DEC environment could be described as both ‘case insensitive’ (when it’s master of its own fate) and ‘case preserving’ (when it has to interact with the outside world).
4. The `dec` command has a mode where it operates as a subcommand environment. This works fine in NT but in Unix, all command line processing disappears. This is great for preventing Unix from messing up a command line containing special characters, but lousy when all command line editing goes away as well (on AIX I can’t even get the delete key to work!) This is one of the biggest frustrations in using this mode on Unix and I’m looking for ways to solve the problem.

2.2 The Directory Structure.

For internal code, the DEC environment mandates a very specific directory structure. This does not apply to the whole of your file space, only that part you use for internal code development. The directory structure looks like this (with explanatory notes to follow):



2.2.1 *General Observations.*

A letter inside a circle indicates that the complete directory structure with the same letter (within the dotted box) should be reproduced at this point. Beware the recursion of multiple directory structures A appearing in directory structure B!

The dotted vertical lines extending the directory lists at each level indicate that the directory list is not exclusive. Other directories can be added as appropriate to the situation.

Taken at face value, this directory structure implies some horrendously long file names. Who would want to type things like

```
s:\nld\lcd\util\dev\bin\nt\foo.obj
```

all the time? Have patience and read on, the DEC environment will help you out.

2.2.2 *Major and Minor Facilities.*

Major and minor facilities are used to break the code up into logical chunks. The diagram only shows one major facility but there can be many (major facilities `vxd`, `trk`, `cal` already exist). Minor facilities are just subprojects of major facilities. This choice of layout is somewhat arbitrary. Other experiments have gone with a single layer solution (no minor facilities) and have ended up with huge numbers of facilities with no clear organization. At the other extreme, some developers advocate an arbitrary depth directory tree where the tree reflects the link or run-time structure of the code. I'm pretty convinced that a single depth tree is inadequate, but arbitrary depth trees are not strictly mandatory and would be far more difficult to implement. I punted and chose a depth two tree.

The minor facility represents the fundamental code chunk handled by the DEC environment. Many commands described later will operate on minor facilities, so now might be a good time to introduce the standard syntax used to reference a minor facility. The syntax is:

```
<major>/<minor>
```

e.g.

```
lcd/util
```

Note that the major facility must be given too. This is to avoid potential name clashes if two major facilities contain a minor facility of the same name. The 'punctuation' is *always* a `/` (on both Unix and NT).

The dotted vertical line at the major facility level indicates that other major facilities could appear at this point. This is correct and very likely to be true, but it's not mandatory. A major facility can located anywhere in the file space.

2.2.3 Branch.

The branch layer is used to implement the software life cycle. The dot/dash line connecting the `test` directory to `dev` denotes alternative, exclusive structures. The test directory will only appear in private user file spaces (and files in the directory tree will be read/write). The `dev/V-1-0-0` directories will only appear in the shared public file spaces (and files in the directory tree will be read only).

Directories `test` and `dev` you were probably expecting, but what are `V-1-0-0` and `V-1-0-1`? These are production releases. Any number of production releases can be kept around governed by need and available file space. You may also have noticed the absence of a `prod` directory. This was forced on me by the lack of a symbolic link mechanism in NT. The upshot is that you can specify that you want to use the `prod` directory, which the DEC environment resolves for you to the highest release number production release.

Occasionally, you may see other directories appear at the branch level in the public workspaces. You may even see a `test` directory. This will only occur when a minor facility is being rebuilt. The build facility does a lot of directory manipulation at this level to ensure that an existing build doesn't get destroyed until a new build completes successfully.

2.2.4 Organization.

The organization layer exists to separate out the various parts and stages of a minor facility. Putting all the source, include, documentation, build, etc. files together into a single directory at this level makes for very muddy file space and can render some tools nearly inoperable (e.g. performing a string search across all files).

The `bin` directory in the diagram is special. If the DEC environment finds a `bin` directory at this level, it will trace down one further directory layer, the name of which is determined by the platform on which you're working. It also does this for the directories `shr` and `exe`, but for no others.

2.2.5 Platform.

The function of the platform layer should I hope be self evident. This layer only exists below the special `bin`, `shr` and `exe` directories at the organization level.

2.3 The Power of Indirection.

One of the great values of logical names is their power to provide *indirect* file references. In 'native' Unix and NT there are only two ways to refer to the location of a file. Absolutely:

```
/afs/slac/g/nld/lcd-working/source/nld/foo.cxx
s:\nld\gismo\nld\foo.cxx
```

or relatively:

```
../../nld/foo.cxx
..\..\nld\foo.cxx
```

References like these end up in all sorts of places. Make files, project settings, embedded in code, etc. Because they can end up in so many untraceable nooks and crannies, we become very reluctant to do anything in the file base which might invalidate them. The absolute style results in making files impossible to move at all. The relative style still allows files to be moved but only on the condition that all the directory relationships remain the same.

Indirect file references remove these constraints. Instead of referring to a file as:

```
/afs/slac/g/nld/lcd-working/source/nld/foo.cxx
s:\nld\gismo\nld\foo.cxx
```

An environment variable can be used to represent the directory and can be concatenated with the file name:

```
$NLD/foo.cxx
%NLD%\foo.cxx
```

Assuming this is done consistently throughout the facility, then the only action required from the developer if `foo.cxx` changes location is to change the environment variable to point to the new location. *All* references to `foo.cxx` change automatically.

This is nothing particularly new even in the Unix and NT worlds. Many applications demand that the installer/user define a few environment variables to make the application work (at bare minimum, putting the directory containing the application in the `PATH` variable). All I've done is used more variables and automated their creation and maintenance.

2.4 User/Group Layering.

The simplest implementation of file indirection would be to set up a single command file somewhere containing all the indirections needed by the group, then require that all group members run it. For instance, the group currently needs access to the CLHEP libraries, so one line in the shared group file might read (in NT syntax):

```
set CLHEP_DIR=V:\LCD\CLHEP\clhep-1.4.0.0\bin\nt
```

Then the great day arrives and a new, better, faster CLHEP distribution becomes available. Assuming that all references to CLHEP libraries (in make files, project settings, …) have been properly prefixed

```
$(CLHEP_DIR)\<library_name>      # This is in gmake syntax
```

then the only change needed to access it is to go to this central file and replace the above line with:

```
set CLHEP_DIR=V:\LCD\CLHEP\clhep-1.5.0.0\bin\nt
```

It is *not* necessary to hunt around for and edit each and every reference to CLHEP libraries in the public files and everyone's private files.

That's already a big boon, but file indirection can be exploited even further to tailor individual user environments. In the above example, I don't think we'd make new CLHEP libraries available without testing them first! To do this, the person tasked to do the testing would simply change the environment variable `CLHEP_DIR` in a *private* session and relink. Nothing else! All the make files and project documents remain the same so the tester doesn't have to make private copies of them.

This is the basis of user/group layering. The group decides on a shared environment and sets up shared files to be run by all group members. Each member can then override the shared environment to suit the task at hand, without interfering with other members of the group and without having to make a complete private copy of the code.

2.5 Indirection of Internal Objects.

The example of the CLHEP libraries demonstrates how to make indirect references to *external* objects. For *internal* objects (the code we control directly), environment variable indirection can be made richer still.

The directory structure used to implement the software life cycle is very formal. The layers always go from major facility to minor facility to branch to organization to platform. The structure is also very deep! When the directory structure was described, it was noted that nobody would (or should) be happy typing file references like:

```
s:\nld\lcd\util\dev\bin\nt\foo.obj
```

As far as possible the DEC environment relieves you of that burden. The pathname for this file can be split into six pieces:

```
s:\nld           <stem>
lcd              <major>
util             <minor>
dev              <branch>
bin              <organization>
nt               <platform>
```

Of these six, the DEC environment knows how to deal with three. The stem is defined in a group shared file, the branch is user selected but is part of the environment (more on this later) and the platform is implicit in where your session is hosted. The user has to provide the other three. How the other three are provided depends on the situation. Many DEC environment commands accept what they refer to as ‘DEC logical names’ which is constructed as follows (on all systems):

```
<major>/<minor>/<organization>
```

The actual environment variable names maintained by DEC are constructed as follows:

```
<major>_<minor>_<organization>
```

Using indirection, the above file could be referenced by a DEC environment savvy command as:

```
lcd/util/bin/foo.obj
```

Alternatively, it could be referenced in a ‘I know nothing about the DEC environment’ command as (Unix then NT):

```
$LCD_LUTL_BIN/foo.obj
%lcd_lutl_bin%\foo.obj
```

The only differences between the operating systems is the syntax to dereference the environment variable (`$var` on Unix and `%var%` on NT), and the fact that Unix is case sensitive, so the environment variable must be presented in the right case. I could have put the NT environment variable in upper case to match Unix, but I wanted to emphasize the difference in case sensitivity between the platforms. In fact, all environment variables for internal structures are generated in upper case.

2.6 Setting Branches.

It's easy to see how a shared file can list the stems for each major facility. Neither is it a stretch to believe that a platform name can be constructed at session start up. But how about that branch directory? How does the DEC environment know whether to substitute `test` or `dev` or `V-1-0-0`? The basic answer is of course that you tell it! A major function of the `dec` command is to allow you to specify the branches to use.

Each minor facility can be set to any available branch by each user per session, and these choices have no effect on other users or other sessions for the same user. User A might be testing out a new development release of some tracking code and have set `trk/recon` to `dev` (and have kept all other facilities in `prod`). User B on the other hand might be testing calorimetry code by setting `cal/cluster` to `dev` (likewise keeping all other facilities in `prod`). Each user can construct an executable application tailored to their particular task without interfering with other users and taking maximum advantage of common code.

In the example just given I had users A and B accessing `dev` branches. What if one of them finds an error? `dev` branches should not be edited and recompiled in situ! The files are all read only for a start and these are publicly shared directories! How does this fit in with grabbing 'official' source code out of CVS?

This is where the `test` branch comes in. Suppose user A found an error in `trk/recon` and wants to fix it. The correct course of action is to:

1. Check `trk/recon` out of CVS into a private area.
2. Tell the DEC environment that you want to use this test area by setting the branch to `test`. This is a little trickier than setting the branch to `dev` or `prod` because DEC doesn't know the stem name for your private workspace, so you have to tell it (the full syntax for doing this will be described in the DEC command section).
3. Do the edit/compile/link/test thing.
4. Return edited files to CVS.
5. Rebuild the `trk/recon dev` branch (for all platforms?!)
6. Set the `trk/recon` branch back to `dev`

2.7 Directory List Environment Variables

Both NT and Unix make heavy use of environment variables which contain lists of directories. The most (in)famous is probably `PATH` (it has the same name on both systems). `PATH` and its kin are used to resolve all manner of file references. They're used as a search path to locate commands, libraries, shareables, etc. Maintaining these variables has to be one of the most laborious and error prone tasks on both systems.

The DEC environment helps out by guaranteeing that any minor facility with an `exe` directory will have that directory included in the `PATH` environment variable. Any minor facility with a `shr` directory will have that directory included in whatever variable is used to indicate a shareable path (`LD_LIBRARY_PATH` on SunOS, `LIBPATH` on AIX, `PATH` (again!) on NT). Furthermore, it will ensure that the correct, branch specific directory is maintained in the environment variable. Suppose you have `bat/bat` in `prod` and change it to `test`. Before the switch `PATH` will contain:

```
PATH=…;…;…;v:\LCD\GismoApps\bat\bat\prod\exe\nt;…;…;…
```

After the switch it will contain:

```
PATH=…;…;…;c:\<your_dir>\bat\bat\test\exe\nt;…;…;…
```

This mechanism is so useful that the `dec` command has extended the principle to arbitrary directory lists and external objects. The syntax for manipulating directory search lists is covered in the `dec` command section.

2.8 How a DEC Environment Session Is Started

The exact method by which a DEC environment session is started varies by operating system. Unix has a very clear concept of multiple independent terminal sessions so overlaying the DEC environment is easily accomplished by adding a few lines to a user's 'login' file. NT is a little more complicated because the basic terminal session, an MS/DOS session, tends to get confused about multiple users, or single users running multiple sessions. Doing the standard NT install should result in an icon on the NT desktop which goes through a rather convoluted process to make it possible to start up multiple independent MS/DOS sessions with the DEC environment overlaid.

However the session is started, both systems initialize the DEC environment the same way. Somewhere during start up the following command is executed:

```
dec run profile profile.dec
```

This operates like a 'DEC logon'. Each user tailors a private copy of `profile.dec` to set up a default session environment. A typical `profile.dec` will execute standard scripts from a common area (NT then Unix):

```
S:\nld\dec_tables
/afs/slac.stanford.edu/g/nld/dec_tables
```

It is these files that make it possible for users to share a group-wide environment.

3 CVS

This section is *not* intended to be a CVS manual. The standard CVS manual is available in PDF and PS formats from:

`http://www.loria.fr/cgi-bin/molli/wilma.cgi/doc.847210383.html`

(I also have a copy in my office).

The intention of this section is to describe the general philosophy of CVS, and to outline the four or five most useful commands.

3.1 Philosophy

CVS is organized internally as a conventional directory tree structure and encourages check in and check out of files to occur in large chunks, usually the ‘straight line descent’ directory structure from the CVS root directory to the requested directory, and then the complete directory tree and files at and below the requested directory. When checking out, CVS reproduces this directory tree in your current directory and populates it with the latest (or ‘head’) revisions of the files in those directories. If you go and look at the checked out directory tree in your own file space you will notice that CVS also creates a whole bunch of `CVS` directories (one for each directory in the main tree). CVS uses files in these directories to keep control information, so these files should never be edited by hand.

Any number of developers can have the same files/directory trees checked out at the same time. CVS does not ‘lock’ files to enforce serial editing. Two users can therefore edit their checked out, private copies of the same file at the same time. The first user to check that file back in will succeed. When the second user attempts to check the file back in, CVS detects the file merge problem and the second user is stuck with sorting out the conflicts. If you get the impression that I don't like this way of doing business, you're right!

With this philosophy, the accepted working style is to check out complete projects (minor facilities in DEC nomenclature), do the standard edit/compile/link/test loop in the developer's private space, and replace individual files or complete directory structures as the code is completed. It doesn't take developers long to discover that it's wise to get the files back in as soon as possible so that the merge problem lands in the lap of the other guy. Wise developers also get into the habit of doing regular updates (re-extracting the same directory tree from CVS) to absorb changes from other users and to get early warning of merge problems in the future.

Putting files back into CVS does not implicitly delete the private copies. There is a command to CVS which means 'I relinquish my interest in this facility' though it is rarely used. Frequently, developers simply delete the directory tree in their private workspace when they are done (this does no harm to CVS). More often, developers forget about a given project until the next time they have to work on it, at which point they discover that they already have it checked out. This can be dangerous. If there is any chance that the private file copies have gone out of date compared to the official CVS copies, be sure to update the private copies before doing anything else.

Putting files back into CVS does not change the contents of the `dev` release. CVS is only responsible for tracking the source files. It knows nothing about the software life cycle. Rebuilding `dev` (or a production version for that matter) can only be accomplished by running the `build` command.

3.2 Using CVS on Unix

The following assumes that CVS will be driven from the command line of a Unix session. Pretty GUI interfaces probably exist in Unix though I have not come across one.

To use CVS, you must first ensure access to the CVS command and identify the location of the CVS repository.

We use the standard SLAC CVS command, so almost any standard SLAC login file should make it available. If you have doubts, issue the command:

```
which cvs
```

You should receive the reply: `/usr/local/bin/cvs`

The location of the repository is kept in the environment variable `CVSROOT.` `CVSROOT` is set for you in the 'group login' which you should source in your own login file.

Unfortunately the Unix side is not as well developed as the NT side and the 'group login' doesn't yet exist. If this is still true when you read this, `CVSROOT` should be set to:

```
/afs/slac.stanford.edu/g/nld/cvsroot/lcd
```

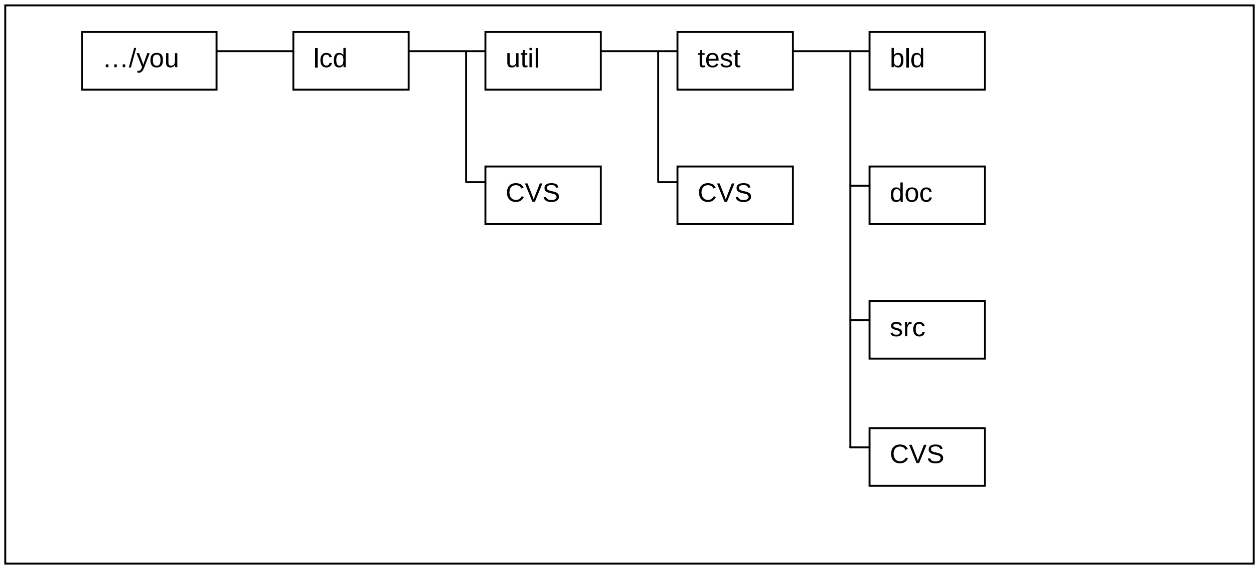
I realize that looks like an AFS directory, but it is in fact just a link to a directory in NFS space.

3.2.1 Checking Out

Suppose you are in your home directory (`/afs/slac/u/ey/you`) and you want to check out minor facility `lcd/util`. You should issue the command:

```
cvs checkout lcd/util
```

you will end up with a directory structure like…



…populated with all the head revisions of the files in these directories. Note that CVS knows nothing about `bin`, `shr` or `exe` directories and does not create them.

Suppose you now want to check out `lcd/gen` as well. Nothing too surprising here. From your home directory, issue the command:

```
cvs checkout lcd/gen
```

CVS will correctly overlay the additional directory structure on top of what already exists, i.e. in directory `…you/lcd` you will find three directories: `gen`, `util` and `CVS`.

3.2.2 Checking In

To check the whole structure back into CVS:

```
cvs commit –m "Your comment goes here" lcd/util
```

To check in an individual file:

```
cvs commit –m "Some comment" lcd/util/test/src/foo.cxx
```

You must be in the correct directory (the stem directory into which you checked out) when you issue these commands.

3.2.3 Adding Files

```
cvs add lcd/util/test/src/my_new_file.cxx
```

Note that this command only marks the file for addition. The file doesn't actually get added into the CVS repository until you `commit` it (either specifically or as part of a complete directory you are committing).

3.2.4 Deleting Files

```
cvs remove lcd/util/test/src/file_to_delete.cxx
```

Once again, the file is only marked for deletion. Nothing really happens until you `commit` the file or the directory containing the file.

Source file tracking systems never physically delete a file. The file in question may not be useful in the latest version of the code, but it may be essential in a prior version. When you tell CVS to delete a file, CVS obliges by hiding the file in a special directory called the 'Attic'. It can be recovered if needed, but in everyday use it simply disappears.

3.2.5 Updating Your Private Copy

Finally, to update your checked out copy with the latest and greatest versions in CVS:

```
cvs update lcd/util
```

or:

```
cvs update lcd/util/test/src/update_this.cxx
```

3.3 Using CVS on Windows/NT

Interactions with CVS from NT fall under two headings. User directed interactions like the ones discussed in the previous section and scripted interactions which occur in some DEC commands. The distinction was not made in the Unix discussion because the scripted interactions are completely hidden from the user and the session set up for both types is the same. This is not true for NT which has the additional complication that with SLAC's security mandates, the client/server interaction between NT and the (Unix based) CVS repository must be secure (which amounts to saying; must be based on SSH).

User directed interactions with CVS are most easily accomplished using a free GUI application called WinCVS. If you've done a standard install of DEC on NT, this application should be on your local disk and you may even have set up a desktop shortcut to it. The first time you start up this application, you will need to set up your preferences. The preferences dialog box usually appears the first time you start the application, but just in case it doesn't, you can find it in the 'admin' menu.

The three items you need to specify are:

Enter the CVSROOT:	<id>@flora:/afs/slac.stanford.edu/g/nld/cvsroot/lcd
Authentication:	SSH server
Use version:	Use cvs 1.10

where <id> is your unix ID.

In the great tradition of GUI applications (particularly free ones) the author of WinCVS believes that documentation is superfluous. To some extent he is right. Once you understand the general philosophy of CVS and have used CVS from a command line, the meanings of the various icons and menus becomes relatively clear. For complete CVS newbies I would therefore advise that you read through the section 'Using CVS on Unix' and try to soak up the general command structure. You can then start playing with the GUI (carefully please, though it's hard to do any real damage), trying to predict the command that the GUI will issue (and conveniently print in the bottom 'log' frame) in response to your input.

To enable scripted calls to CVS on NT, you will need to do a little more work. The standard DEC install should have done this for you already, but just to be sure you should check that you have the following environment variables defined:

```
CVSROOT=:ext:<id>@flora:/afs/slac.stanford.edu/g/nld/cvsroot/lcd
CVS_RSH=SSH
```

Where <id> is again your unix ID. Note that CVSROOT is defined *differently* inside WinCVS and in the DEC session. This little detail probably cost me a week's work to get right!

4 The DEC Command

This section will include many terminal session extracts shown in boxes. The format is as follows:

- Extracts are for the NT system.
- The session prompt is `Z:\`
- User input in is boldface.

4.1 Function

The `dec` command provides the user with the tools to interrogate and manipulate the DEC environment. For external objects, `dec` can:

- Set and clear external symbols using a layered user/group approach (a user can override a group symbol with a user symbol and then revert to the group symbol simply by removing the user symbol).
- Manipulate search list variables like `PATH` to insert and remove extra directories. This again follows a layered approach so that if a user addition overrides a group addition, the user addition wins. If the user removes this addition, the group addition is reestablished.

For internal objects, `dec` can also:

- Make arbitrary subsets of all known major facilities available.
- Manipulate the branches of minor facilities.
- Save and restore named DEC environments.

4.2 Syntax

All dec commands follow the general syntax described in ‘The DEC Environment’. The specific syntax looks like:

```
dec <verb> [<parameters and qualifiers>]
```

Where the square brackets mean that further parameters and qualifiers may or may not be needed. Simple verbs (like exit) require nothing further. More complex verbs (like set) can require anything up to four more parameters/qualifiers. In session extracts the dec prefix will always be shown. In command descriptions, the dec prefix will be dropped for clarity.

4.3 Help

Inline help is available for all commands. The command structure is sufficiently rich to require a kind of directory structure of help thus:

```
Z:\dec help

Commands in DEC:

@                   *  Execute a file of DEC commands
ATTACH              *  Attach major facilities
DETACH              *  Detach major facilities
EXIT                   Leave DEC
HELP                   This command
QUIT                   Leave DEC (synonym for EXIT)
RUN     PROFILE     *  Run  a ’profile’ file
SAVE    PROFILE     *  Save a ’profile’ file
SET     BRANCH      *  Set a branch
        EXTERNAL    *  Set an external name
        STEM        *  Set a major facility stem
SHOW    ATTACHED       Show attached major facilities
        BRANCH         Show branches
        ENVIRONMENT    Show all environment variables
        EXTERNAL       Show externally defined names
        LOGICAL     *  Translate a logical name
        STEM           Show all major facility stems

A * indicates more detailed information available; be more specific e.g.

HELP ATTACH

( There are one or two more undocumented commands which I use for   )
( debugging.  Do not be alarmed if a typo suddenly does something!  )
( Consider it an Easter Egg from you friendly local developer. APW. )

Z:\
Z:\dec help attach

Syntax of the ATTACH command:

ATTACH <majorlist>

Where  <majorlist>  A comma delimited list of major facilities to attach.
                    The major facilities must be known to the system.

Z:\
```

4.4 Commands for External Objects

To manipulate/examine external objects, use the commands:

```
set external
show external
```

To manipulate simple external objects:

```
set external <name> <value>

where <name>   Name of external symbol.
      <value>  Value of external symbol. If omitted, <name> is
               -removed- from the external name table.
```

To manipulate path-modifying external objects:

```
set external <name>:<label> <value>

where <name>   Name of path to be modified.
      <label>  Arbitrary but unique label so that this entry can
               be distinguished from other modifications to the
               same path.
      <value>  Value to insert into the path. If omitted, the
               labeled entry is -removed- from the path.

The colon is an integral part of the syntax. This implies that
‘simple’ external names can never include a colon.
```

To examine external object settings:

```
show external
```

Example (with notes):

```
Z:\dec set external my_variable my_value
Z:\dec set external another anything
Z:\dec set external CLHEP_INC c:\CLHEP\clhep-1.5.0.0
Z:\dec set external CLHEP_DIR c:\CLHEP\clhep-1.5.0.0\bin\nt
Z:\dec set external path:my_lib c:\mylib
Z:\dec set external path:your_lib c:\yourlib
Z:\dec set external class_path:javalib c:\javalib
Z:\dec show external
==========================================================================
Source  Name                Value
------  ------------------  ------------------------------------------
user    my_variable         my_value
        another             anything
        CLHEP_INC           c:\CLHEP\clhep-1.5.0.0
        CLHEP_DIR           c:\CLHEP\clhep-1.5.0.0\bin\nt
        path:my_lib         c:\mylib
        path:your_lib       c:\yourlib
        class_path:javalib  c:\javalib
group   CLHEP_INC           v:\LCD\CLHEP\clhep-1.4.0.0
        CLHEP_DIR           v:\LCD\CLHEP\clhep-1.4.0.0\bin\nt
        STDHEP_INC          v:\LCD\STDHEP\v0.00
        STDHEP_DIR          v:\LCD\STDHEP\v0.00\stdHEP\stdhep\Release
==========================================================================

Z:\
```

- External names in the list are grouped according to their source. User defined names are put in 'user' and group defined names are put in 'group'.
- Where both the user and the group define the same external variable name (e.g. `CLHEP_INC` and `CLHEP_DIR`), the user value takes precedence. If the user definition is removed, the group value is restored.
- The same pathlist can be modified independently by separate entries (e.g. `path:my_lib` and `path:your_lib` both modify `PATH`). This allows `c:\my_lib` and `c:\your_lib` to be inserted into and removed from `PATH` separately. If these libraries are always used in conjunction with each other then a better idea might be:

  ```
  set external path:libs c:\my_lib;c:\your_lib
  ```

4.5 Commands for Internal Objects

4.5.1 Defining the List of Attached Major Facilities

As the complexity of LCD code increases, it may not be necessary to access all major facilities all the time. To manipulate/examine the list of attached major facilities, use:

```
attach
detach
show attached
```

To add more major facilities to the environment:

```
attach <majorlist>

where  <majorlist>  A comma delimited list of major facilities to
                    add to the environment.
```

To remove some major facilities from the environment:

```
detach <majorlist>

where  <majorlist>  A comma delimited list of major facilities to
                    remove from the environment.
```

To examine the list of major facilities currently attached:

```
show attached
```

Example:

```
Z:\dec show attached
dec,cal,det,lcd,muon,trk

Z:\dec attach vxd,rootapps

Z:\dec show attached
dec,cal,det,lcd,muon,trk,vxd,rootapps

Z:\dec detach trk,rootapps

Z:\dec show attached
dec,cal,det,lcd,muon,vxd

Z:\
```

4.5.2 Changing the Branch Setting of a Minor Facility

A minor facility must be a member of an attached major facility for it to be accessible for manipulation. The commands to change a minor facility branch, and to examine the branch settings of all attached minor facilities are:

```
set branch
show branch
```

To set a minor facility branch:

```
set branch <major>/<minor> <branch> [<stem>]

where  <major>  Major facility name.
       <minor>  Minor facility name.
       <branch> Requested branch. Could be ‘test’ or ‘dev’ or
                ‘prod’ or a specific production release (e.g.
                V-1-0-0).  ‘prod’ is interpreted as meaning
                ‘the most recent production release’.
       <stem>   Directory stem where the code can be found. Only
                needed if <branch> is ‘test’ (ignored otherwise).

Defining <stem> can be confusing, so let's try an example:

You are working on the minor facility <major>/<minor>. You
fetched a working copy from CVS which you placed in directory
'c:\user\you'.

CVS generated a complete directory structure for you so that (for
instance) the source code for <major>\<minor> is in:

c:\user\you\<major>\<minor>\test\src

The correct value of <stem> is:

c:\user\you
```

To examine branches:

```
show branch
```

Example:

```
Z:\dec show branch
Major  Minor  Branch          Test Stem Directory
-----  -----  --------------  -------------------
dec    dec    prod (V-1-1-4)
bat    bat    prod (V-2-0-3)
cal    cal    prod (V-2-0-1)
det    det    prod (V-2-0-1)
lcd    gen    prod (V-2-0-1)
       lcd    prod (V-2-0-1)
       util   prod (V-2-0-1)
lum    lum    prod (V-2-0-1)
muon   muon   prod (V-2-0-1)
trk    trk    prod (V-2-0-1)
       util   prod (V-2-0-1)
vxd    vxd    prod (V-2-0-1)
```

```
win    win    prod (V-2-0-3)

Z:\dec set branch trk/trk dev

Z:\dec show branch
Major  Minor  Branch          Test Stem Directory
-----  -----  --------------  -------------------
dec    dec    prod (V-1-1-4)
bat    bat    prod (V-2-0-3)
cal    cal    prod (V-2-0-1)
det    det    prod (V-2-0-1)
lcd    gen    prod (V-2-0-1)
       lcd    prod (V-2-0-1)
       util   prod (V-2-0-1)
lum    lum    prod (V-2-0-1)
muon   muon   prod (V-2-0-1)
trk    trk    dev
       util   prod (V-2-0-1)
vxd    vxd    prod (V-2-0-1)
win    win    prod (V-2-0-3)

Z:\dec set branch lcd/util test c:\mytest

Z:\dec show branch
Major  Minor  Branch          Test Stem Directory
-----  -----  --------------  -------------------
dec    dec    prod (V-1-1-4)
bat    bat    prod (V-2-0-3)
cal    cal    prod (V-2-0-1)
det    det    prod (V-2-0-1)
lcd    gen    prod (V-2-0-1)
       lcd    prod (V-2-0-1)
       util   test            c:\mytest
lum    lum    prod (V-2-0-1)
muon   muon   prod (V-2-0-1)
trk    trk    dev
       util   prod (V-2-0-1)
vxd    vxd    prod (V-2-0-1)
win    win    prod (V-2-0-3)

Z:\dec set branch cal/cal V-2-0-0

Z:\dec show branch
Major  Minor  Branch          Test Stem Directory
-----  -----  --------------  -------------------
dec    dec    prod (V-1-1-4)
bat    bat    prod (V-2-0-3)
cal    cal    V-2-0-0
det    det    prod (V-2-0-1)
lcd    gen    prod (V-2-0-1)
       lcd    prod (V-2-0-1)
       util   test            c:\mytest
lum    lum    prod (V-2-0-1)
muon   muon   prod (V-2-0-1)
```

```
trk    trk    dev
       util   prod (V-2-0-1)
vxd    vxd    prod (V-2-0-1)
win    win    prod (V-2-0-3)

Z:\
```

4.6 Saving and Restoring Environments

In the course of working on LCD code, a developer might switch between different environments. Today is the day for testing `trk` code, tomorrow it might be `cal` code, the next day it might be `trk` again. Setting up the environment each day might be laborious so DEC provides a mechanism to save and restore named environments.

The commands to save and restore an environment are:

```
save profile
run profile
```

To save an environment:

```
save profile <filename>

where <filename>  The name of the file in which to save the
                  profile.
```

```
This command generates a file called <filename> containing DEC
commands. <filename> can be expressed using DEC logical names in
addition to the more conventional file notations. If the file
extension is missing, the file extension ‘.dec’ is added.
```

To restore an environment:

```
run profile <filename>

where <filename>  The name of the file which contains the
                  commands to restore the environment. Usually
                  the product of a prior ‘save profile’ command.
```

```
<filename> can be expressed using DEC logical names in addition
to the more conventional file notations. If the file extension is
missing, the file extension ‘.dec’ is assumed.
```

The run profile command is very careful to remove all prior environmental settings before recreating the new environment. It even removes anything set up by a user's `profile.dec`. This can be very powerful when you want to start from scratch. If you just want to remember small incremental changes to your DEC environment, you may prefer to use the `@` command, described next.

4.7 Other Commands

4.7.1 '@<filename>'

The `@<filename>` command directs the DEC command processor to read its input from the file `<filename>`. Files can be nested to any depth (or until system resources run out). Note that this command is very different to the run profile command. The run profile command guarantees that the user environment will be cleared before executing the profile requested.

`<filename>` can be expressed in DEC logical name format as well as the more conventional file notations.

4.7.2 'Show Logical'.

Many DEC commands accept 'logical names' to reference files. To find out how a file reference resolves:

```
show logical <major>/<minor>/<organization>

where <major>         Major facility name.
      <minor>         Minor facility name.
      <organization>  Organizational directory.
```

Example:

```
Z:\dec show logical bat/bat/exe
v:\LCD\GismoApps\bat\bat\V-2-0-3\exe\nt

Z:\
```

4.7.3 'Show Environment'.

Underneath the commands to manage external and internal object references, DEC has to maintain standard environment variables. To see what DEC is *really* doing, I wrote this command for debugging purposes. It turned out to be so useful that I made it available in the interface. To show the environmental variables being maintained by DEC:

```
show environment [--all | --simple | --path]

where  --all     List simple and path modifying entries (default)
       --simple  List simple entries.
       --path    List path modifying entries

--all, --simple and --path are mutually exclusive.
```

I've had to edit the next terminal session to get it to fit, but the principles of the display are intact:

```
Z:\dec show environment
===========================================================================
Simple environment variables
---------------------------------------------------------------------------

Name          User        Logical                     Group
------------  ----------  --------------------------  --------------------
lcd_source    oneevt
lcd_events    1
lcd_run       1
lcd_seed      19780513
arve_inc      c:\my_arve                              s:\nld\nld-oct\arve
DEC_DEC_EXE               s:\nld\dec\dec\prod\exe\nt
DEC_DEC_BLD               s:\nld\dec\dec\prod\bld
DEC_DEC_DOC               s:\nld\dec\dec\prod\doc
DEC_DEC_SRC               s:\nld\dec\dec\prod\src
CLHEP_INC                                             s:\nld\nld-oct
===========================================================================
Path modifying environment variables
---------------------------------------------------------------------------

Name        Source   Label      Value
----------  -------  ---------  -----
PATH        user     myexe      c:\myexe
                     yourexe    c:\yourexe
            logical  dec_dec    s:\nld\dec\dec\prod\exe\nt
PEGS4PATH   user     original   v:\LCD\GismoApps\PEGS4
CLASS_PATH  user     javalib    c:\javalib
===========================================================================

Z:\
```

- The display demonstrates the layering of user names against group names, and let's the secret out of the bag that the layering is really three deep … user/logical/group.
- Simple variables are always laid out with the environment variables defined by the user level first, by the logical layer second and by the group layer third.

Note that the DEC environment remembers group layer definitions even when they are over-ridden by the user layer (see `arve_inc` above).

- In the path modifying section, the environment variable `PATH` has had three directories added to its base definition. Two of these were direct user requests, the third was inserted by the logical name handling of DEC when it processed minor facility `dec/dec`. Note that the user entries precede the logical entry in the display, just as they do in the ‘real world’ value of `PATH`.
- Path modifying is not restricted to environment variable `PATH`. This display shows that environment variables `PEGS4PATH` and `CLASS_PATH` are also being modified.

4.7.4 ‘Exit’ and ‘Quit’

`exit` and `quit` are synonymous and are only used to leave the subcommand environment described in the next section. Be careful with the `exit` command! If it’s given at the NT command prompt instead of in the subcommand environment, it will end the session! Get into the habit of using `e` or `q` to exit the subcommand environment. That way you can never accidentally blow out of your DEC session with the command `exit`.

4.8 Modes of Operation

The DEC command is unusual in that it has two modes of operation. The following two terminal extracts demonstrate the difference (while exercising identical functions):

```
Z:\dec show attached
dec,bat,cal,det,lcd,lum,muon,trk,vxd,win

Z:\dec show branch

Major  Minor  Branch          Test Stem Directory
-----  -----  --------------  -------------------
dec    dec    prod (V-1-1-4)
bat    bat    test            c:\GismoApps
cal    cal    test            c:\GismoApps
det    det    test            c:\GismoApps
lcd    gen    test            c:\GismoApps
       lcd    test            c:\GismoApps
       util   test            c:\GismoApps
lum    lum    test            c:\GismoApps
muon   muon   test            c:\GismoApps
trk    trk    test            c:\GismoApps
       util   test            c:\GismoApps
vxd    vxd    test            c:\GismoApps
win    win    test            c:\GismoApps

Z:\
```

```
Z:\dec
DEC> show attached
dec,bat,cal,det,lcd,lum,muon,trk,vxd,win

DEC> show branch

Major  Minor  Branch          Test Stem Directory
-----  -----  --------------  -------------------
dec    dec    prod (V-1-1-4)
bat    bat    test            c:\GismoApps
cal    cal    test            c:\GismoApps
det    det    test            c:\GismoApps
lcd    gen    test            c:\GismoApps
       lcd    test            c:\GismoApps
       util   test            c:\GismoApps
lum    lum    test            c:\GismoApps
muon   muon   test            c:\GismoApps
trk    trk    test            c:\GismoApps
       util   test            c:\GismoApps
vxd    vxd    test            c:\GismoApps
win    win    test            c:\GismoApps

DEC> exit

Z:\
```

In the first extract the `dec` commands are issued directly to the system and the response follows the command.

In the second extract the `dec` subcommand environment is entered by issuing the bare `dec` command. Once in the subcommand environment any `dec` command can be issued (without prefixing it with `dec`) and execution will continue in the subcommand environment until the command `exit` is found, at which point control returns to the usual command prompt.

Which should you use? Depends on what you’re doing! The `dec` command has some start up overhead. If you issue several `dec` commands directly to the system, you pay that overhead for each command. Enter the subcommand environment and you pay that overhead only once. On the other hand, once you are in the subcommand environment, you can only access `dec` commands. You pays your money and you takes your choice.

5 The BUILD Command

5.1 Function

The function of the `build` command is (unsurprisingly) to build test, development and production releases of the LCD code. The target of a `build` is a single minor facility and at heart, it’s an easy problem:

1. If necessary, fetch the source code from CVS to a standard location.
2. Run the `gmake` application on the minor facility make file (which it expects to find in the `bld` directory specified by the DEC environment).

To make `build` a little more useful, it should also:

1. For development and production builds, keep a record of when builds are run, by whom and against which platform.
2. For production builds, generate tagged groups of files in CVS so that the build can be recreated at a later date.
3. It should be easy for the developer to do the build. The developer has the best appreciation of when a build is required and should not be held up by the ‘we build every Tuesday fortnight’ code management approach.

Real life adds its own complications:

1. At SLAC we have four platforms sharing the same filing system. The DEC environment specifies that the `bin`, `shr` and `exe` are kept separate, but that

source files are shared. If a build has already been made for SunOS, then the source files should not be refetched from CVS to do the AIX build.

2. It’s ‘unhealthy’ to build a new development release (say) directly in the extant public development directory structure. If we convert to shareables, it becomes downright impossible to overwrite a shareable if it’s already in use.
3. Builds don’t always work! If a build fails it should do nothing to harm what’s already in the public file space.

To satisfy all these conditions, the build command becomes considerably more complicated than a simple ‘fetch-then-make’ script!

5.2 Syntax

`build` follows the standard VMS lookalike syntax described in ‘The DEC environment’. The specific syntax is:

```
build    <major>/<minor>
         [--ARGUMENTS=<args>] [--OPTIONS=<opts>]   <-  General
         [--REPORT=<rep>]                          <-  General
         [--TEST | --DEVELOPMENT | --PRODUCTION]   <-  Tier 0
         [--OLD]                                   <-  Tier 1
         [--NEW | --RELEASE=<rel>]                 <-  Tier 2
         [--MAJOR | --MINOR | --VERSION]           <-  Tier 3

where    <major>   Major facility.
         <minor>   Minor facility.
         <args>    String to pass to the make process (e.g.clean)
         <opts>    Options to pass to the make process (e.g.–n)
         <rep>     Reporting level (1 or 2) (default is 1)
         <rel>     Release number.


General qualifiers (allowed for all types of builds):

--ARGUMENTS=<args> String to pass into the make process. Make
                   files take standard strings like ‘clean’.
--OPTIONS=<opts>   Options to pass to the make process.  These
                   are the 'real' make options like '-n', '-p',
                   etc.
--REPORT=<rep>     Reporting level.  Must be 1 or 2.
                   1 => Report errors (default).
                   2 => Report activities and errors.


Tier 0 qualifiers determine the type of build (mutually
exclusive):

--TEST             Build a test release (default).
--DEVELOPMENT      Build a development release.
--PRODUCTION       Build a production release.


Tier 1 qualifiers are only allowed for --DEVELOPMENT builds:

--OLD              Do not update from CVS before build (this
                   makes it possible to produce binaries from
                   identical sources on machines that share a
                   filebase, i.e. the morgans and floras at
                   SLAC).  Default is to update from CVS.


Tier 2 qualifiers are only allowed for --PRODUCTION builds (no
default):

--NEW              Build a brand new production release.
--RELEASE=<rel>    Reproduce a production build on another
                   platform.

<release> is a string of the form V-<x>-<y>-<z> where <x>, <y>
and <z> are all integers and are, respectively, the major release
number, the minor release number and the version number.


Tier 4 qualifiers are only allowed for –PRODUCTION --NEW builds:

--MAJOR            Increment the major release number.
--MINOR            Increment the minor release number.
--VERSION          Increment the version number (default).
```

5.3 Help

Help is available online with the special command:

```
build help
```

The output looks a lot like the above section!

5.4 Version Numbers and CVS Tags

A close inspection of the build command syntax will reveal that version numbers are only generated for production releases. This was a deliberate choice. Development builds are likely to be rapidly moving targets, and no ‘real’ work (i.e. generating large, long lived datasets) should be done with them.

Production release version numbers follow a pattern. The version number has the format:

```
V-<x>-<y>-<z>
```

Where `<x>`, `<y>` and <z> are integers and represent the major release number, the minor release number and the version number respectively. Build understands version numbers and makes sure that nothing bad happens to them (e.g. that they start decreasing!) Incrementing any number other than the rightmost (the version number) results in numbers to the right resetting to zero.

Deciding which number to increment when building a production release is always difficult. The following are just general guidelines:

- If the new build is not backward compatible, bump the major release number (this is about the only hard and fast rule).
- If the new build is backward compatible, but the internal coding has undergone considerable change, bump the minor release number.
- If the new build is just a minor bug fix away from the previous release, bump the version number.

In addition to indicating the character of the new release, the version number plays a vital role in CVS. When a production release is made, all head revisions of files in the specified minor facility are ‘tagged’ in CVS with the version number. Once the tag is established, that precise set of files can always be retrieved.

5.5 Build History Files

Each time a development or production build is run, it puts a line in the respective development or production history file (unimaginatively named `development.his` and `production.his` and stored in the minor facility’s `bld` directory). The line specifies:

1. The build version. The exact version number for production builds, `<new>` or `<old>` for development builds.
2. The date and time the build was run.
3. The result of the build (`Good` or `Fail`).

4. The platform it was run on.
5. Who ran it.
6. If the qualifier `--ARGUMENTS=<args>` was specified to the build, the string `Args: <args>`. Because this string tends to fall off the side of an 80 column screen, using the `--ARGUMENTS=<args>` qualifier will also result in a `*` appearing after the `Good` or `Fail` string.

5.6 When Builds Go Right

`build` never builds production or development releases directly into the target directory (`dev` or `V-1-1-1`). If something goes wrong the existing release could end up destroyed. `build` always builds in a branch level directory called `test` in the public file space. This should be the only time you find a `test` directory in the public file space. If all goes well, build finishes up by renaming directories. Note the even then, the existing version is not destroyed. When building `dev` for instance, the previously existing `dev` directory is renamed `dev_0` and build's `test` directory is renamed to `dev`. If `dev_0` already exists, the preexisting `dev` is renamed to `dev_1` and so on. Removing historical `dev` releases (`dev_0`, `dev_1`, etc.) is a hand operation.

For production builds, the `test` directory is renamed to the release version name. If the name is already in use (this can happen when different platforms share the same file space), the preexisting directory is renamed with a `_0, _1, _2,`… suffix much like the process described for `dev` releases.

When deciding whether to prune away historical releases remember that a historical `dev` release cannot be recreated whereas historical `prod` releases can. (Actually that's not strictly true, but it's certainly much harder!).

5.7 Mixed Up Branches

When you run `build`, you do not need to worry about the branch setting of the target minor facility in your environment. `build` will save your current setting, do whatever it needs to do, then restore your setting. *This is the only minor facility that build protects this way*, which brings us to the topic of what to do about the branch settings of other minor facilities when you run a build.

When you're testing in your own area, the combination of branches you link together to form an executable is nobody's business but your own. When you're building something in a public area you must be *much* more careful:

- Never link the build target against other minor facilities you have in `test`. The rebuilt library/shareable/executable may work for you but will very likely break for other developers who can't see your `test` minor facilities.

- Never link a production release against other minor facilities in `dev`. This further implies that if a combination of minor facilities depend on each other and are all currently in `dev`, production releases should be made in reverse hierarchical order (release callee libraries before caller libraries).
- Only under demanding circumstances should you make a `dev` build link against other minor facilities in `dev`. If you must do this, make sure everyone else knows this so that they can set those minor facilities to `dev` too. I realize that minor facility A (for which you are building a new `dev` release) might work with both the `dev` and `prod` versions of minor facility B which you are linking in `dev`. If this is true then other users can run minor facility B in `dev` or `prod` without a problem. My experience however is that the most frequent reason for having two minor facilities in `dev` is because something in the interface has changed, forcing minor facilities A and B to march in lock-step, `prod` with `prod` and `dev` with `dev`.

5.8 The Make File

There is *just one* gmake file per minor facility. It is stored in the `bld` directory and is named after the minor facility, e.g.:

```
v:\LCD\GismoApps\lcd\util\dev\bld\lcd_util.gmak
```

It is designed as a template which the user fills out according to the instructions in the file. I am not listing those instructions here because they are far from complete (shareables are not yet supported) and are likely to evolve. As far as possible the user is relieved of all responsibility for platform specific dependencies and should only need to fill in half a dozen fields or so.

Note that on NT the building facilities provided by the Microsoft Developer Studio are completely ignored. Please get into the habit of doing *all* builds (including NT builds) using the `build` command.

6 Other Commands

There is an assortment of other commands I've written for the DEC environment. They fulfill a variety of needs and are not easy to classify so I've just lumped them together in this section.

6.1 'pathlist' (Unix and NT)

I found I was forever inspecting environmental variable search lists like `PATH`, which gets old very quickly. `pathlist` just parses such variables and formats readable output.

```
pathlist <list>

where <list>  Comma delimited list of environment variables which
              pathlist assumes are search lists. If omitted, PATH
              is assumed.
```

```
Z:\pathlist path

path = c:\mylib
       c:\yourlib
       s:\nld\dec\dec\prod\exe\nt
       C:\Program Files\DevStudio\SharedIDE\BIN
       C:\Program Files\DevStudio\VC\BIN
       C:\Program Files\DevStudio\VC\BIN\WINNT
       C:\WINNT\system32
       C:\WINNT
       c:\lcd_tools
       s:\nld\dec\dec\prod\exe\nt

Z:\
Z:\pathlist path,include

   path = c:\mylib
          c:\yourlib
          s:\nld\dec\dec\prod\exe\nt
          C:\Program Files\DevStudio\SharedIDE\BIN
          C:\Program Files\DevStudio\VC\BIN
          C:\Program Files\DevStudio\VC\BIN\WINNT
          C:\WINNT\system32
          C:\WINNT
          c:\lcd_tools
          s:\nld\dec\dec\prod\exe\nt

include = C:\Program Files\DevStudio\VC\INCLUDE
          C:\Program Files\DevStudio\VC\MFC\INCLUDE
          C:\Program Files\DevStudio\VC\ATL\INCLUDE

Z:\
```

6.2 *'pathlook' (Unix and NT)*

Another annoying manual search is trying to discover which version of a file will be resolved by a path search. `pathlook` will scan a path and list the first (or all) file(s) which match a file name..

```
pathlist <filename>
          --PATH=<path>
          --ALL

where <filename>  File name to search for. Can contain the *
                  wildcard.
      <path>      Name of pathlist to search.  Defaults to PATH.
      --ALL       List all resolutions.  Default is to stop at
                  the first resolution.
```

```
Z:\>pathlook --all --path=include *stream
C:\PROGRA~1\MICROS~2\VC98\INCLUDE\FSTREAM
C:\PROGRA~1\MICROS~2\VC98\INCLUDE\IOSTREAM
C:\PROGRA~1\MICROS~2\VC98\INCLUDE\ISTREAM
C:\PROGRA~1\MICROS~2\VC98\INCLUDE\OSTREAM
C:\PROGRA~1\MICROS~2\VC98\INCLUDE\SSTREAM
C:\PROGRA~1\MICROS~2\VC98\INCLUDE\strstream

Z:\
```

6.3 'which' (NT Emulation of Unix Command)

`which` identifies which (if any) file will be run in response a command line command. The NT emulation follows NT resolution rules. It does not cache information and properly follows changes to the environment variables `PATH` and `PATHEXT`.

```
which <command>

where <command>  Command to be traced.
```

```
Z:\which dec_command
s:\nld\dec\dec\V-1-1-4\exe\nt\dec_command.rexx

Z:\which nonesuch
nonesuch: Command not found

Z:\
```

6.4 'Rexxpand' (Unix and NT)

Different REXX scripts frequently need access to the same set of utilities. Things like 'does file xxx exist', 'rename directory yyy to zzz'. Rather than copy and paste utility code between scripts I invented `rexxpand` to preprocess and expand REXX scripts written with a special format.

`rexxpand` reads REXX scripts looking for special format comment lines which direct `rexxpand` to insert another file at this point. These special comment lines can also be made conditional (which was very useful when trying to produce a body of utility routines which would provide the same facilities on NT and Unix). Think of `rexxpand` doing for REXX what the `#include` directive does for the C/C++ preprocessor.

This is pretty obscure stuff, so I shall just outline the command syntax and leave it at that. If anyone's interested in more details, try `rexxpand help` or come and see me.

```
rexxpand <i_file> <o_file>
         [--[NO]QUIET]
         [--[NO]EXPAND]
         [--MAKE[=<m_file>]]
         [--FLAGS=(<flag0>,<flag1>,...)]

where <i_file>     Input file.
      <o_file>     Output file.
      <m_file>     Make file extract.
```

```
--[NO]QUIET  Control information messages (default:/QUIET)
--[NO]EXPAND Control creation of <o_file> (default:/EXPAND)
--MAKE       Control creation of <m_file> (default:/NOMAKE)
--FLAGS      Comma delimited list of control flags.
```

6.5 ‘MSD’ (NT only)

Getting Microsoft’s Visual Studio to recognize the DEC environment is quite a trick. If you start (I don’t want to type all that again, just call it MSD), from it’s icon or by double-clicking an MSD project file, MSD will start up *and inherit the completely vanilla system environment*.

This of course is completely unacceptable! It destroys all the advantages gained by using DEC. The `msd` command solves this problem.

Whenever a process is started in NT it inherits the environment of the place from which it’s started. That’s why a ‘bare’ MSD startup inherits the vanilla system environment. To inherit the DEC environment, MSD must be started from the terminal session where the DEC environment is operating. This is exactly what `msd` does:

```
msd [<major>/<minor>]

where <major>  Major facility name.
      <minor>  Minor facility name.
```

In response, `msd` will search for the workspace file for this minor facility in a standard place (the `bld` directory) and start MSD with this workspace open. If the workspace file is missing or omitted from the command line MSD still starts, but does not load a workspace.

Once the MSD application is started, its environment is frozen. To change the DEC environment you must exit MSD, change the environment at the command line and then restart MSD.

This could conceivably be fixed by rewriting the `dec` command in Visual Basic and making the commands available as macros in MSD. Anyone want to take this on? No? Thought not!

Once inside MSD you can open and close other workspaces as you see fit, but be *very careful* to open the workspace file corresponding to your environment. If you’re tracking down an error in `dev lcd/util`, it will do you no good to open the workspace file for `prod lcd/util`.

6.6 'Run' (NT only)

NT does not have a batch system. Without a batch system it's awkward to run background jobs to soak up all those wasted CPU cycles. The `run` command provides a dorky sort of solution. `run` can also spawn off another interactive session (this was how I originally started MSD sessions, but it was so awkward I eventually made `msd` a command in it's own right).

Syntax of the 'batch' RUN command:

```
RUN--LOG=<log>[--PRIORITY=LOW|NORMAL] <filename> <args>
```

Syntax of the interactive RUN command:

```
RUN[--[NO]SPAWN][--[NO]WAIT] <filename> <args>
```

Where:

```
<filename>    File to run.  <filename> is logically translated
              first.

<args>        Arguments to be forwarded to <filename>.
```

Qualifiers used only in 'batch':

```
--LOG=<log>   <log> is the name of a log file where batch output
              is to be written. Presence of the --LOG qualifier
              is what makes this a 'batch' RUN command.
--PRIORITY=   Batch execution priority. Accepts LOW and NORMAL
```

Qualifiers used only interactively:

```
--[NO]SPAWN   Spawn into another window       (default:/SPAWN)
--[NO]WAIT    Wait for spawned window to end  (default:/NOWAIT)
```

7 Installing DEC on NT

7.1 Prerequisites

You must have Microsoft's 'Visual Design Studio' installed. If you don't, stop now! This is quite a big and complicated installation, especially as you need to apply a service pack afterwards. Details from Ramon Berger, Tony Johnson, Richard Dubois and Tony Waite.

You must have Cygwin installed. This is a Unix-like overlay on top of NT which gives you access to a Unix-like terminal session (Bourne shell only). DEC does not take advantage of these facilities (though it might be a good idea!)

DEC *does* take advantage of the fact that Cygwin provides both a `gcc` compiler (to do include analysis of C/C++ source code) and the `gmake` application.

Cygwin installation is trivial:

- Copy the directory `s:\nld\cygnus_download` to your `c:` disk.
- Log in as administrator.
- Double click on the one file in the `cygnus_download` directory.
- Answer the usual questions.
- Delete the `cygnus_download directory`.
- Log back in as yourself.

All other tools that you need (e.g. REXX and WinCVS) are stored in directory `s:\nld\lcd_tools` which you should copy to your `c:` disk.

7.2 Very Nasty Fiddly Bit

Don't ask me why, just do it! (I don't want to have to even think about this more than I have to!)

In your own account (you should *not* be an administrator for this), pull up the control panel directory. Most easily accomplished by mousing:

```
Start->Settings->Control Panel
```

In the control panel window, double click on the `System` icon. In the new window this produces, select the `Environment` tab. In the lower list, look for a variable called `path`. *Ignore the path variable in the upper list.*

What happens next depends on whether you found a `path` variable in the lower list.

If the `path` variable does *not* exist:

1. Use the `Variable` and `Value` fields at the bottom of the window to create a new variable called `pathusr` with value '' (i.e. the empty string). Remember to click the `Set` and `Apply` buttons when you've got the fields right.
2. Use the `Variable` and `Value` fields at the bottom of the window to create a new variable called `path` with value `%pathusr%`. Remember to click the `Set` and `Apply` buttons when you've got the fields right.

If the `path` variable *does* exist:

1. Click on the `path` variable to get it's name and value into the editing boxes.
2. Change the *name* from `path` to `pathusr` and apply the change. Remember to click the `Set` and `Apply` buttons when you've got the fields right.
3. Use the `Variable` and `Value` fields at the bottom of the window to create/edit a variable called `path` with value `%pathusr%`. Remember to click the `Set` and `Apply` buttons when you've got the fields right.

Exit the system properties window and the control panel. End of very nasty fiddly bit.

7.3 Run the DEC Install Script

Double click on the file:

```
s:\nld\dec\dec\<best_version>\exe\nt\NT_install.cmd
```

where `<best_version>` is the highest production version available (currently `V-1-1-4`, but that's likely to change!)

If your NT box is set up to ignore ‘known’ file extensions, the `.cmd` may not be visible.

This will start a pretty dumb line-by-line install script. The script will issue it's own instructions which you should follow.

7.4 Convenience

Neither of the following items is mandatory and you may have a different way you prefer to work. I just found these tricks convenient.

Create a desktop shortcut to the DEC ‘start’ command file. You may have installed it somewhere else, but the traditional location is:

```
c:\lcd_tools\DEC_start.cmd
```

Rename the shortcut `DEC`. Double clicking on this icon will start a DEC session. You can then tailor the DEC window to your heart's content (colours, fonts, size, etc.) by pulling down the menu from the MS/DOS icon (top left corner of the window) and selecting `Properties…`

Create a desktop shortcut to the WinCVS executable file. You may have installed it somewhere else but the traditional location is:

```
c:\lcd_tools\wincvs.exe
```

Rename the shortcut WinCVS.

7.5 WinCVS Preferences

The first time you run WinCVS you may be presented with a screen asking you to select preferences. If this happens to you, then the important answers are:

- `‘Enter the CVSROOT:’`
  `<your_unix_ID>@flora:/afs/slac.stanford.edu/nld/cvsroot/lcd`
- `‘Authentication:’` SSH server
- ‘Use version:’ `Use cvs 1.10`

8 Installing DEC on Unix

No formal installation procedures have been developed yet. Badger APW at SLAC to get your Unix environment set up. If enough people do the badgering, APW will have the incentive to produce formal procedures!

9 Future Development

When I started DEC I knew little or nothing about either Unix or Windows/NT. Scripting at the DEC level has certainly taught me a lot about both (most of which I would have preferred not to know). My current feeling is that DEC is becoming overly elaborate for the return it provides. In the same breath I have to say that I know of no product, commercial or otherwise, that can do what DEC can do. Rather than turn DEC into a black hole of effort, I've decided to let DEC in its current form stand or fall on its merits while I get back to doing something 'real'.

Which is not to say that DEC is complete. Among the many deficiencies I invite people to consider and even provide solutions for:

1. There is no support for shareables.
2. Creating a new facility is difficult (there's a nasty little bootstrap problem).
3. I would love to have code version information (i.e. the CVS tag) flow all the way into created datasets, thus tying the dataset to the version of code that created it. This is quite doable, I just haven't done it!
4. Building for five platforms is still manual. This can be very time consuming even with the help of `build`. The process could be automated, but it would require servers on the major hosts (the methodology adopted by DUCS) and error detection and recovery would be difficult.