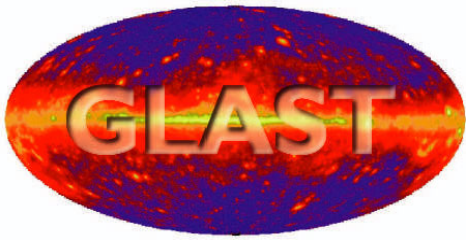

LAT Flight Software

CMX Manual

Type:	User Manual
Version:	V2-1-0
Author:	A.P.Waite
Created:	16 March 2004
Updated:	25 May 2004
Printed:	25 May 2004

Manual for the CMX code management system.

Contents

Figures

Tables

0 Preface

CMX (CMT eXtra) is my attempt to provide a cooperative code development environment for GLAST LAT flight software. It is by no means a polished, elegant solution, but I believe it incorporates the essential elements required.

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 (A Quick User Walkthrough) which shows how a developer works with CMX, and subsequent reference sections giving specifics of the directory layout, user commands and so on. Experienced developers should probably skip the "Introduction" and start at "A Quick User Walkthrough". To make it easy to go directly to "A Quick User Walkthrough", I give here the executive summary of the Introduction:

- Source file management is accomplished using the CVS application.
- Building is controlled using the CMT application.
- CMX defines a software life cycle whose structure is illustrated in section "A Quick User Walkthrough" and is described in more detail in section "The CMX Environment".
- The environment is supported by a number of commands and CMT extensions that provide facilities to manage individual user sessions, build releases, etc.

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. Those that do tend to be expensive commercial (and heavyweight) applications like SNiFF+ and ClearCase. Given that GLAST LAT ground software (with its larger user community and code base) has chosen to use the (free) CVS/CMT tools, it seemed appropriate to follow suit for flight software (though at the time of writing, there *will* be differences between our implementations).

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

1.0 Source File Management/Tracking

1.0.0 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 then checking them back in.

1.0.1 What's Implemented?

The tool of choice is CVS. CVS is used very widely and is free. I have some concerns about the application (how to protect the administrative files, how to remove empty directories in the repository, how to manage parallel editing and the resultant merge, etc.), but there's a lot of experience with this tool.

The basic CVS is a command line driven application which can talk to either a local file repository or (using client/server methods) a remote file repository. For Windows/NT users (not applicable to flight software) there are a number of add-on GUI applications like WinCVS, jCVS, etc.

Software repositories will be located at SLAC (in NFS filespace for a number of frustrating technical reasons). The flight CVS repository will be kept separate from the ground software repository. All members of the flight software team will need a SLAC computer account, and their home computer must have copies of both the CVS software and SSH (Secure SHell) software. This problem is common to both ground and flight software, and a first attempt to outline the procedures users need to follow can be found at:

http://www-sldnt.slac.stanford.edu/glast/Software/SLACcvs/

An online CVS manual can be found at:

http://www.cyclic.com/CVS/oa/Docs/manual/index.html

More traditional PS and PDF manuals (though these seem to updated less often than the online manual) can be found at:

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

Despite some reservations I have about CVS, it has solved one problem very well. Plain text files on Unix use a simple LF (line feed) 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 LF 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.

1.1 Module Management/Building

1.1.0 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 timestamps on the dependent files and the target file. If all dependent files have an older timestamp than the target file, then the make application assumes that the target is up-to-date and does not execute the transformation.

This all sounds very sane, but as anyone with practical experience can tell you, make file maintenance can be very difficult. This has resulted in a number of add-ons like the GNU automake/autoconf/libtools suite. These attempt to provide the user with a more generic module description language which the tools can then translate into the required compiler/platform/etc dependent make files.

1.1.1 What’s Implemented

Another such tool is CMT and this is the tool the ground software people have chosen to adopt. For the obvious reasons of compatibility and group expertise, flight software will also use CMT, but ground and flight software will *not* use exactly the same version. The bifurcation is a result of the different environments in which we work. Building flight software requires the ability to do cross-compilation to the embedded system architecture. Ground software has no such requirement. To avoid platform explosion, flight software is trying to restrict itself to Sun Unix and Linux machines, but *not* Windows/NT. Ground software must support Windows/NT. The flight software group will be very small and could thus be very nimble. Ground software will have many more developers and much more code, making for far higher inertia.

The authors of CMT maintain a web site at:

http://www.cmtsite.org

While avoiding the Windows/NT platform would certainly simplify flight software code management, it's not altogether clear to me that we can do this. An obvious point of contact would be the flight software event filtering code. This clearly falls in the flight software domain, but might well be requested by ground software for inclusion in detailed simulations. For this reason I have attempted to remain *compatible* with ground software usage of CMT, i.e. it might be awkward for ground software to use flight software, but it is doable.

1.2 Software Life Cycle

1.2.0 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!)

1.2.1 What's Implemented?

Unfortunately the kind of heavy-handed bureaucracy described in the last paragraph stifles the one great advantage LAT flight software enjoys … a small group of motivated, experienced developers. Such a group can be very nimble when minimally constrained by procedures. The CMX environment is geared to just such a small group and defines a minimal life cycle with just three stages:

0 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.

1 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.

2 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 CMX environment is in support of the software life cycle and of dovetailing it with the other code management elements.

1.3 Glue

1.3.0 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, CMT, 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, CMT 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. This is what CMX attempts to provide.

1.3.1 What's Implemented?

Originally written as `/bin/sh` scripts, the CMX tools are now implemented as a small set of (mostly) perl scripts invoked from the command line. The perl scripting language was adopted as the complexity of CMX increased and performance of `/bin/sh` scripts declined. Note however that these scripts will *not* run on Windows/NT, even if your NT box is outfitted with a Unix overlay like `cygwin`. These scripts take advantage of Unix specific techniques like symbolic file links and Unix specific (more accurately Sun and Linux specific) features like the use of the environment variable `LD_LIBRARY_PATH`.

All commands begin with `cmx` and take a number of parameters. A full description of the CMX commands can be found in section "The CMX Command".

2 A Quick User Walkthrough

The following tour is not designed to be reference material. It will blithely use undefined `cmx` commands and reference equally undefined and mysterious `/symlnk` directories. The only intent here is to give the reader a feel for how the CMX world works. All the unpleasant details of just how CMX does what it does can be found in later sections.

This section includes a number terminal session extracts shown in boxes. The format is as follows:

- Extracts are for a Sun Unix box called `tersk02`.
- The session prompt is `tersk02:apw>`
- User input in is **`boldface`**.

2.0 Working On A Pre-Existing Package

For this exercise I will assume that CMX has been installed on the developer's computer and that all the code for a host executable called `exeA` has been compiled in a public space. The executable `exeA` is just a shell that calls routines in a number of shareables … `shrA`, `shrB` and `shrC`. `exeA`, `shrA`, `shrB` and `shrC` are also the CMT package names corresponding to these images. Let's start by running the public version of `exeA`:

```
tersk02:apw> exeA
<stuff happens>
tersk02:apw>
```

Well that was pretty amazing. It worked! The user didn't do anything special to make `exeA` available but it activated anyway! That's because the implementer of `exeA` was careful enough to ensure that `exeA` was listed for export in his CMT requirements file. CMX reacted to that instruction and put a file link in a public `/symlnk` directory and the user's CMX login ensured that the public `/symlnk` directory was on the user's `PATH` variable.

That was too easy. Let's suppose that the user finds a bug in `shrB` while running `exeA`. How does the user set about fixing it?

The one thing the user is *not* going to do is hack on the source code in the public filespace! That version is fully tagged and bundled and *inviolate* (CMX even makes the source files in public spaces read only to avoid pratfalls).

The user does *not* hack on the source code in the public development area either!

The user needs a private playpen to work on `shrB`. CMX can provide that playpen. It's called a user private project and is created as follows:

```
tersk02:apw> cmx create project MyProject ${HOME}/glast
tersk02:apw>
```

For this to work, the directory `${HOME}/glast` must already exist and the directory `${HOME}/glast/MyProject` must *not* exist. Providing these conditions are met, CMX will lay down a user private directory structure at the top of which is the directory `${HOME}/glast/MyProject.`

So far so good. Now the user needs to get hold of the source code for `shrB`. This is safely tucked away in the flight software CVS repository. To get hold of it:

```
tersk02:apw> cmx fetch shrB –-test=${HOME}/glast/MyProject
==========================================================================
cvs checkout –d /<home>/glast/MyProject/source/shrB/test shrB
--------------------------------------------------------------------------
<lots of messages from CVS>
==========================================================================
tersk02:apw>
```

In addition to fetching the code for you the `cmx fetch` command also sets up your private copy of `shrB` as a CMT package by running the `cmt config` command on it.

At this point the user has a complete copy of the source for `shrB` in a standard (private) CMX directory structure, but there's no corresponding binary files. `shrB` must be built.

```
tersk02:apw> cmx build shrB
CMX: Requested branch differs from environment
tersk02:apw>
```

A classic blunder! The user wants to build a private test version, but the user's CMX environment still thinks the user wants to use the public production version. (Unless specified otherwise, the `cmx build` command defaults to doing `--test` builds, so it was the default test build that clashed with the user's production environment). This is how to ask CMX to use the test branch of package `shrB`:

```
tersk02:apw> cmx set branch shrB --test=${HOME}/glast/MyProject
tersk02:apw> cmx show branch
shrB         test     /<home>/glast/MyProject
tersk02:apw>
```

The first command sets `shrB` to branch test and (because the locations of user test branches are not permanently recorded by CMX) identifies what the user means by test (i.e. where the test directory is located). The second command simply confirms that the first command worked.

Now repeat the build command:

```
tersk02:apw> cmx build shrB
<lots of output from the build>
tersk02:apw>
```

Success! Now run exeA again.

```
tersk02:apw> exeA
<stuff happens>
tersk02:apw>
```

Wait a minute. Why should the user expect any difference? shrB might have been rebuilt, but the user never relinked exeA. Welcome to the world of shareables. There was no need to relink exeA because exeA will automatically pick up the first version of shrB it finds according to its resolution rules. CMX has carefully arranged that the first version of shrB that exeA finds is the newly built version in the user's private play pen. (It did this by placing a link in the user's process private …/symlnk/<platform>/shr directory once the test build was complete). To confirm what the image activator resolves for its shareables, try using the cmx ldd command:

```
tersk02:apw> cmx ldd exeA –-full
<cmx lists all the shareables activated and how they were found>
tersk02:apw>
```

At this point the user is home free. In his/her private area, the user goes round the edit/compile/build/test loop until the bug in shrB is fixed. This doesn't interfere with anyone else because this is a *private* area, and the link which finds the test shareable is hidden inside a user (and process) *private* directory.

OK the bug is fixed. How is the correction fed back into the public domain? First the user must return the source to the CVS directory:

```
tersk02:apw> cmx commit shrB
tersk02:apw>
```

Assuming the user has privileges to do public builds (I assume all developers will):

```
tersk02:apw> cmx update shrB –-development
tersk02:apw>
```

This updates the public development branch of shrB with the latest code from the repository. Notice that the user didn't have to tell CMX where to find the development branch of shrB, CMX knows where all the public directories are located. Now rebuild shrB:

```
tersk02:apw> cmx build shrB –-development
CMX: Requested branch differs from environment
tersk02:apw>
```

Damn! Did it again! (I'm emphasizing this pratfall because I've done it so often myself!).

```
tersk02:apw> cmx set branch shrB --development
tersk02:apw> cmx show branch
shrB         dev
tersk02:apw> cmx build shrB –-development
<lots of output from the build>
tersk02:apw>
```

This would probably be a good moment for our user to pause. At this point there is a publicly available version of shrB containing the user's bug fixes. Now would be a good time for the user to inform his/her cohorts that there's a new shrB out there in development and could they please beat on it to make sure it's correct. Other users can access this version by setting the shrB package to the development branch in their own environments. If shrB survives this abuse it becomes a candidate for promotion to full production status, which involves tagging this version in the CVS repository, fetching it from the repository to the public production space and rebuilding it. Let's suppose that this new version is tagged as version V2-0-1 then:

```
tersk02:apw> cmx fetch shrB –-production -–version=V2-0-1
tersk02:apw> cmx set branch shrB –-version=V2-0-1
tersk02:apw> cmx show branch
shrB         V2-0-1
tersk02:apw> cmx build shrB –-version=V2-0-1
<lots of output from the build>
tersk02:apw>
```

There is one last step to make `V2-0-1` the official production version and that is to tell CMX to replace (say) `V2-0-0` with `V2-0-1`:

```
tersk02:apw> cmx set production shrB –-version=V2-0-1 –-global
tersk02:apw>
```

That completes a complete walk through of the software life cycle for a user working on a pre-existing package. I've been very chatty and have oversimplified in many places. I'll try to move a little faster on the next walk through.

2.1 Working On A New Package

Not every package a user works on will already be in the CVS repository and available in the public filespace. This is how a new package (`shrX`) might come into existence.

First of all, the user needs a bare package. All packages must exist inside projects (public or private). This is a new package, so it starts life in a user private project. Let's assume we can reuse project `MyProject` from the previous example. To create a new package in `Myproject`:

```
tersk02:apw> cmx create package shrX ${HOME}/glast/MyProject
tersk02:apw>
```

This creates a new package prepared according to CMT's and CMX's conventions. It has a `/src` directory, a `/cmt` directory and a `/shrX` directory (for putting exported header files in). The `/cmt` directory contains a skeleton `requirements` file, a `.cvsignore` file (which stops a variety of auxiliary files generated by both CMT and CMX sloshing back and forth to the repository), and a few others (`Makefile` comes to mind). This command is even kind enough to do the equivalent of `cmx set branch shrX –-test=${HOME}/glast/MyProject`.

The user can now create new source files in the `/src` directory (or copy them from wherever), and edit the `requirements` file so that it builds the `shrX` shareable. The `shrX` package is not in CVS yet, but it can take full advantage of the `requirements` file, useing other packages which are part of the public filebase. The user will probably need to write a little test executable as well. This could be part of the `shrX` package (a good idea in general … the test travels with the shareable) or if the test is too convoluted, the test executable can be in another test package (just `cmx create package exeX` … to create it).

Eventually the `shrX` package passes muster in the user's private space and the time has come to expose to the outside world. The equivalent step when working on a pre-existing package is `cmx commit` …, but that's not going to work because this package isn't in CVS yet. CMX provides the solution:

```
tersk02:apw> cmx import shrX DAQ
tersk02:apw>
```

That's it! This command says import package `shrX` into CVS and associate it with the public project `DAQ` (the associated public project *must* pre-exist). It's certainly easy to call, but it's one of the most frightening `cmx` commands around, given the scope of what it attempts to do. Assuming the command passes all integrity checks and runs to completion, `cmx import`:

- Imports the complete `shrX` source tree into CVS. The vendor is set to the user ID of the person issuing the command and the primary tag is set to `V0-0-0`.
- Does a CVS checkout on what it just imported back into the user's private filespace replacing the previous contents (this is so that the user's copy is made into an official CVS directory tree).
- Does a `cmt config` on the user's new copy to make it CMT ready.
- Does a `cvs checkout` on the imported package into a newly created development area in public filespace.
- Does a `cmt config` in this new development area to make it CMT ready.
- Does a `cvs checkout` on the imported package into a newly created production area (a directory called `V0-0-0`) in the public filespace.
- Does a `cmt config` in the public production area to make it CMT ready.
- Updates CMX database files to register the new package and it's official production branch (`V0-0-0`).

Pretty scary stuff. Put your lead pants on before issuing this one. For all that, the process is not complete … notice that `cmx import` didn't attempt any builds. That is left to the user (and the process should mimic the description I gave for building public versions at the end of the previous section).

2.2 Summary

Clearly these are not very realistic examples (nor were they intended to be). It is left as an exercise for the interested reader (I've always wanted to say that) to consider how to handle:

- Maintaining packages with multiple CMT tags on the same platform
- Maintaining packages across platforms which share a filing system
- Maintaining packages across different sites

3 The CMX Environment

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

The CMX environment does the following:

- CMX provides a *process* based user environment. This means that every process (i.e. terminal session) maintains its own environment and the user is free to tailor each process environment as required (using the `cmx` command).
- CMX knows how to do cross-compilation for embedded systems. It also extends the syntax of the CMT module description file (the `requirements` file) to allow different content for host and embedded system constituents (that's a mouthful … simply put, a single CMT package could for instance support an embedded system transmit image and a host receive image simultaneously, with each image free to define an arbitrary list of source files from the package's pool of source code).
- CMX builds shareables *only*. There are no archive libraries in any package CMX maintains. Enforcing shareables provides the basis for the so-called "mix and match" style of image activation. This style allows the user to select the precise set of modules to execute *at image activation time*. There is no need to relink an executable to change its module version content.
- CMX maintains the run-time environment in parallel with the link-time environment. If a user has package `foo` in test and uses CMX to rebuild it, CMX guarantees that when the user runs `foo`, the test version of `foo` is activated.
- CMX commands are directory location insensitive. Any CMX command can be issued while the user is in any directory and the result will be the same (no more `cd`ing all over the place).

3.0 The Public Directory Structure

To achieve all this (particularly the last item), CMX enforces a strict directory layout. The public areas (i.e. where the shared code is maintained) look like (and note that some of these structures are nested):

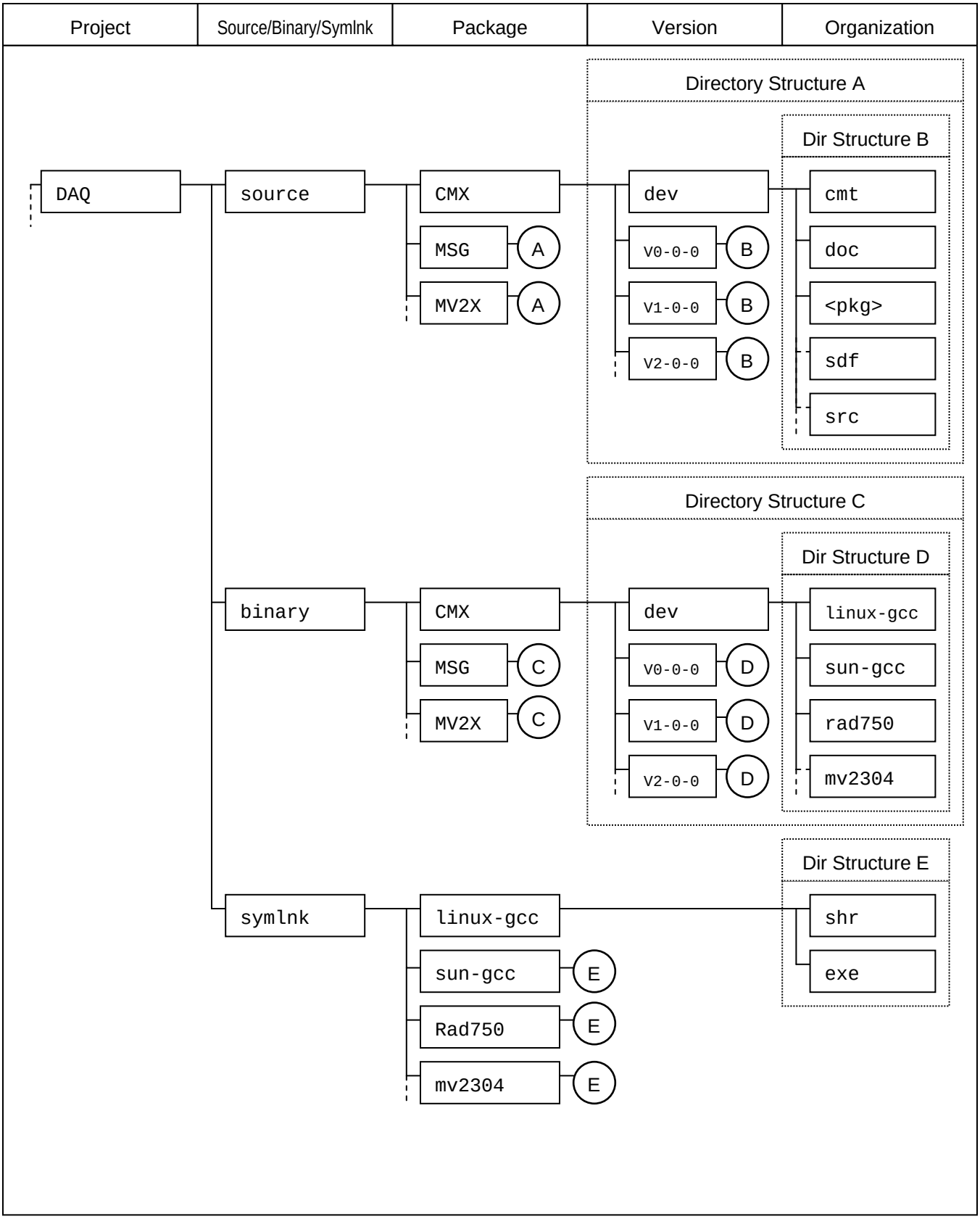

Figure 1 Public directory structure

This is clearly an extensive architecture, but there is method in my madness. Starting at the left hand side of the diagram and reading left to right:

3.0.0 Project

The project layer is intended to block the code out into large chunks. Unix restrictions limit the number of chunks to around half-a-dozen or so. My intent was to provide a chunk each for DAQ, ACD, CAL and TKR. This follows from my experience on a previous experiment where it proved advantageous to provide each subsystem a formal public area. This would be even more powerful if we can designate a manager for each chunk.

Each of these major chunks can appear anywhere in the file system, though for reasons of simplicity, it would probably be better to adopt the SLAC convention where the project directories are maintained in parallel with each other under a `/flight` directory.

3.0.1 Source/Binary/(Symlnk)

This will probably prove to be the most controversial piece of the structure. This design was driven by my previous effort to produce a code management system.

In my previous design I had the binary directories in parallel with the source directory (in much the same way that off-the-shelf CMT does it). This was always getting me into trouble with CVS. A careless import into CVS could result in all the binary directories ending up in CVS as well (and good luck trying to persuade CVS to get rid of them). Even if the import was done properly, later checkouts and builds resulted in the binary directories reappearing. Committing the module back to CVS does not copy these directories into CVS, but CVS does print a lot of nuisance messages complaining about the existence of directories it doesn't recognize.

Another source of frustration stemmed from SLAC Computer Services (SCS) preference for doling out disk space in dribs and drabs. Once a package spilled off its disk space (and that didn't take long when each package was compiled for four target platforms), it was very tricky to expand disk space by simply linking in another AFS mount. CVS thinks it's responsible for creating directories when a module is checked out and will refuse to do the checkout if any part of the directory tree already exists. This source of annoyance can only get worse with CMT, because CMT believes that it is responsible for creating the binary directories and so CVS and CMT end up fighting each other.

To avoid the whole problem, I introduced the source/binary split. Now CVS can interact with just the source code directories in the source branch. CMT has been instructed to throw the binaries across into the binary tree. If I need to introduce extra disk space for the binaries (source code is usually small by comparison), I can insert links from the package level of the binary tree to new AFS mount points. Problem solved (at the expense of some ugliness).

The `/symlnk` directory is a very special beast. Describing it now would not fit into the left to right flow, so I will defer discussion of this directory to a separate section at the end.

3.0.2 Package

This is where the directory structure enters the world of CMT proper. CMT itself mandates a specific directory structure starting with a directory giving the package name. Many packages can be placed in parallel.

The only restriction CMX adds is that all package names in all projects must be unique (the CMX commands enforce this so you're not likely to do this by accident). If project DAQ introduces a package UTIL, then projects ACD, CAL and TKR cannot introduce their own package called UTIL. There are no hard and fast rules for package naming (at least not yet). The only reason to mention this is to ask for cooperation on package names

3.0.3 Version

This is once again CMT's structure, though the `dev` directory is unconventional and was my invention. Other than the `dev` directory, directories at this level are named with a CMT version number (the CMT version of which allows a wide range of syntaxes containing one, two or three numbers used to characterize the version … CMX rules for the version string are rather more restrictive). By convention, this name is identical to a tag in the CVS repository. The `dev` directory is meant to reflect a build of the current head revisions in the repository (and thus might be unstable … caveat emptor).

3.0.4 Organization

Everything described so far is identical in the source and binary trees and is purely architectural (i.e. no real files, just other directories). At the organizational level the structure starts to diverge and the directories start to contain real files.

3.0.4.0 Organization In The Source Tree

The CMT mandated directories at this level are `/src` and `/cmt`. CMX further mandates the `/<pkg>`, `/doc`, `/ptd` and `/sdf` directories:

`/src`	Source code for the package.
`/cmt`	CMT control files (most importantly the `requirements` file).
`/<pkg>`	Dedicated to header files exported by the package.
`/doc`	Official package documentation files (manuals, final design documents, …)
`/ptd`	The "Package Test Directory". There can be more than one of these at the discretion of the package developer, but there is always at least this one.
`/sdf`	The "Software Development Folder". Used to store development cruft. Design fragments, unofficial testing results, a simple activity log and so on.

Any number of other directories can be introduced at this level on a package by package basis. As already pointed out there can be more package test directories. Some packages may sensibly introduce a `/dat` directory and so on.

It is this part of the directory tree that CVS operates on. In the repository, the top level directories are each of the packages. Below them are the `/src`, `/cmt`, `/<pkg>`, `/doc`, `/sdf`, … directories. (A CVS trick is used to introduce the `test`, `dev` or `<version>` directory when a package is checked out of CVS).

3.0.4.1 Organization In The Binary Tree

Each directory at this level corresponds to a CMT tag. Here we run into a nomenclature problem. Both CVS and CMT use the word "tag" but with completely different meanings.

A CVS tag is used to collect together a set of file/versions in the repository. This makes it possible to identify and extract a coherent set of source and is invaluable for release control. As far as possible I will avoid using the word tag for this case. Instead I will use the word version.

A CMT tag is a little harder to describe because it's used to cover a multitude of sins. For GLAST LAT flight software I have tried to restrict the use of CMT tags to representing a compilation/link environment. Four examples of CMT tags are shown in the figure. The complete list of available tags is as follows:

Tag	Target	Compiler	Comment
linux-gcc	Linux host	Native gcc suite	
sun-gcc	Sun host	Native gcc suite	gcc suite configured to use the Sun vendor linker (ld) command.
mv2304	Motorola MV2304 SBC (VME)	VxWorks suite	PPC 604 processor.
mcp750	Motorola MCP750 SBC (PCI)	VxWorks suite	COTS processor to emulate the flight SBC.
rad750	BAE RAD750 SBC (PCI)	VxWorks suite	Flight board (PPC 750).
i845e	Custom Pentium tower	VxWorks suite	PC to drive the Front End Simulator.
win-gcc	MS Windows host	Native gcc suite	gcc suite on cygwin overlay

Table 1 List Of Currently Supported CMT Tags

3.0.5 (Source/Binary)/Symlnk

This branch clearly does not follow the overall structure of the rest of the directory tree, but that is because it has a very distinct function.

Standard code management practice tries to break a software project into smaller and more manageable chunks with well-defined interfaces between the chunks. This is reflected in the project/package organization described in the previous sections. Unfortunately, this does not fit well with the image activation procedures of Unix, particularly if an all shareable environment is desired.

Unix image activation (more accurately Sun and Linux image activation … shareable support is *highly* variable) involves several strategies to locate the shareables an image requires. One of the early strategies looks for the shareables in the directory list defined in the environment variable LD_LIBRARY_PATH. It is this mechanism which CMX seeks to exploit. However, in the case of MS windows the image activation using LD_LIBRARY_PATH doesn't work. We actually have to put the shareable location on the PATH environment variable.

Unfortunately, this is not a very scalable mechanism. For practical purposes, environment variables are limited to about 1000 characters. By the time a software project contains twenty or thirty packages, listing the binary directory for each package in LD_LIBRARY_PATH will simply fail. On top of that there is the ease with which an environment variable like LD_LIBRARY_PATH can be pooched. It's a very popular mechanism and all sorts of software try to use it (VxWorks software for example). CMX tries to circumvent both problems by editing LD_LIBRAY_PATH only once (when you set up the CMX environment) and by placing a strictly limited number of elements on LD_LIBRARY_PATH.

CMX maintains symbolic links in each project's /symlnk directories pointing to the (host) products of all packages in the project. The /symlnk/<cmt-tag>/shr directory contains links for shareables and /symlnk/<cmt-tag>/exe contains links for executables (and sometimes script commands). The CMX set up procedure (not yet discussed) puts the /symlnk/<cmt-tag>/shr directory for each project in LD_LIBRARY_PATH and the /symlnk/<cmt-tag>/exe directory for each project in PATH. The value of <cmt-tag> is a default value set up by CMX and maintained in the environment variable CMX_L_CONFIG. (The CMX set up procedure actually places one more directory in each of LD_LIBRARY_PATH and PATH, but a discussion of the purpose of those extra directories is deferred until a user's private directory structure is discussed).

Thus if image foo wants to activate a shareable called libbar.so which is produced by package baz in project buzz, the image activator walks down LD_LIBRARY_PATH until it comes to the …/buzz/symlnk/<cmt-tag>/shr directory. In there it finds the link to libbar.so,

which points off to something like …/buzz/binary/baz/V1-0-0/sun-gcc/bar/libbar.so. The image activator is satisfied and executable foo starts up.

Just to illustrate how variable shareable lookup can be, the above description is *not* true for the MS windows image activator. Even when running a cygwin overlay (to make MS windows look like Unix), shareables must be found via the PATH environment variables (not LD_LIBRARY_PATH). Furthermore, the image activator will not traverse a Unix-style symbolic link. To make Windows operate like Unix, CMX handles the details of making the shareable search path part of PATH, and instead of making a link to the shareable, it performs a copy (yuch!).

The /symlnk directories can be thought of as providing an alternative and independent view of the binary tree. This view is optimized for consumption by the Unix mechanisms for image activation and command/executable name resolution.

3.1 The Private Directory Structure

The previous section described the publicly visible directory structure. Useful as it is to understand what's in there, a typical user will spend far more time working on projects in a private file area. If the project is part of the grand CMX scheme, then CMX will demand a particular directory structure in the user's private space as well:

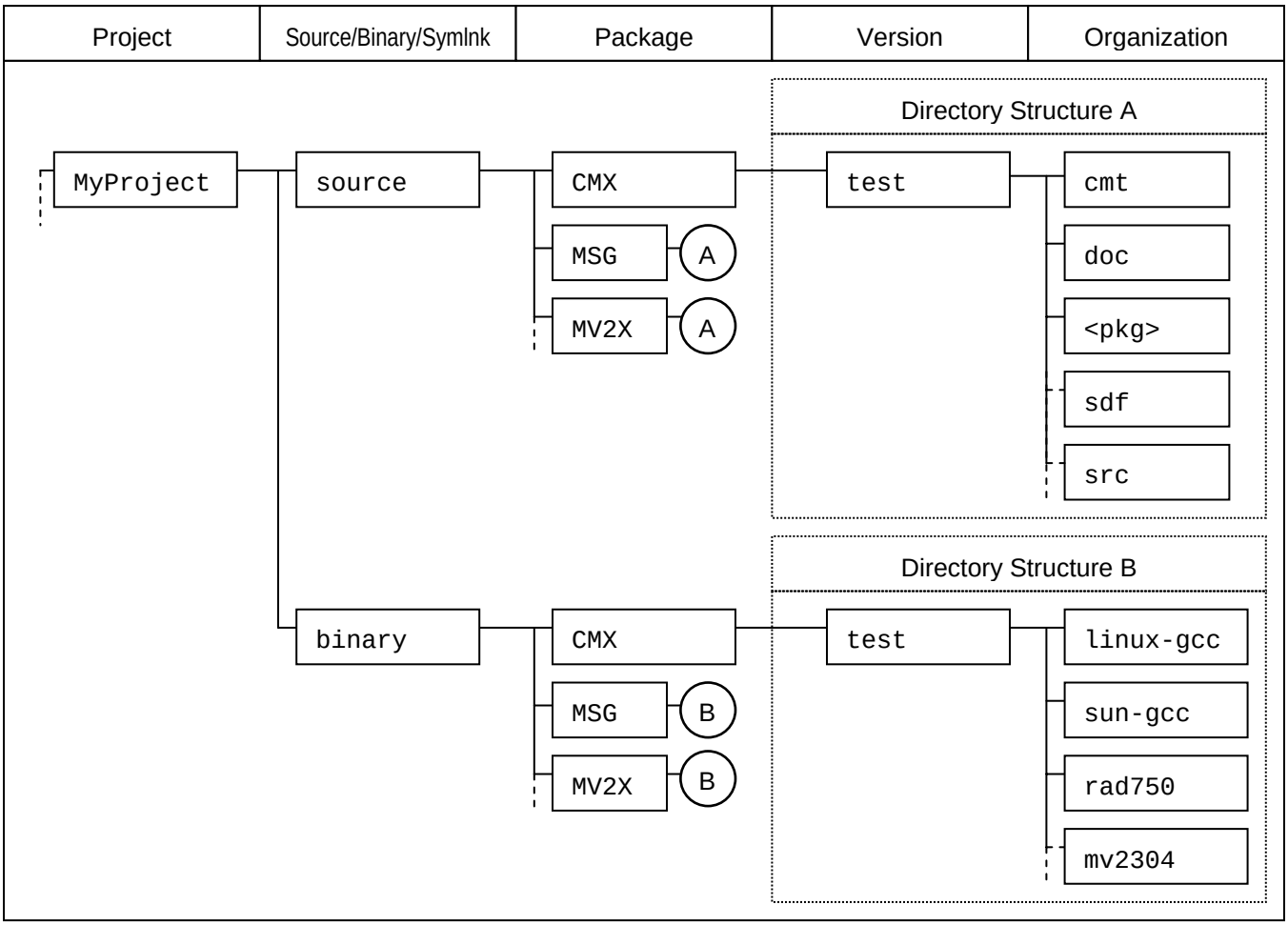

Figure 2 Private directory structure

The first thing to notice is that this is very similar to the public structure. In what follows, only differences will be noted.

3.1.0 Project

User projects can have any name and there is no restriction on the number of projects a user can have in existence at the same time. A user project does not contribute elements to either `LD_LIBRARY_PATH` or `PATH`. A user project can appear anywhere in the user's files area.

3.1.1 Source/Binary/(SymInk)

The major difference at this level is that there is no `/symlnk` directory. This will be further discussed in a later section.

3.1.2 Package

There is no restriction against mixing packages from different public projects in the same private project.

3.1.3 Version

Only `test` is allowed (CMX dixit).

3.1.4 Organization

Identical to the public case.

3.1.5 (Source/Binary)/SymInk

Why is there no `/symlnk` directory? Well there is! It's just that in the user's private environment, it doesn't appear in any project tree.

When a user starts up the CMX environment, CMX creates a process specific directory tree in the user's private file area (specifically, CMX creates `${HOME}/CMX/<host>-<procid>/<cmt-tag>/shr` and `${HOME}/CMX/<host>-<procid>/<cmt-tag>/exe`, where `<host>` is the host name and `<procid>` is the process ID). These two directories are the ones referred to earlier (in the discussion on the public `/symlnk` directories). The CMX startup procedure places directory `${HOME}/CMX/<host>-<procid>/<cmt-tag>/shr` in `LD_LIBRARY_PATH` *before* the shareable contributions from public projects. Similarly directory `${HOME}/CMX/<host>-<procid>/<cmt-tag>/exe` is placed in `PATH` *before* the executable contributions from public projects. This allows links placed in the user's process private `/symlnk` directories to override the public definitions, providing the basis for mix and match.

3.2 Other Things CMX Does To Your Environment

For the sake of completeness in this section, I need to point out that CMX has a somewhat greater impact on the user's environment than simply creating and maintaining directory structures. In addition CMX maintains a number of environment variables. A more detailed description of these variables can be found in "CMX Internals". For now, please note that CMX

uses approximately (10 + 3n) environment variables where n is the number of packages CMX is looking after. All CMX environment variables start with `CMX_`.

It should already be clear that CMX is quite an extensive environment and that a user needs to do something proactive to set it up. What is less clear is that to clean up the process specific directory tree, the user also needs to do something to shut the environment down. The commands `cmx login` or `cmx start` set up the CMX environment while `cmx logout` or `cmx stop` will shut it down. At the user's discretion these commands can be automated (in the Unix C shell) by putting `cmx login` or `cmx start` in the `.login` file and `cmx logout` or `cmx stop` in the `.logout` file (I'll leave it up to user's of other shells to figure out how to do this). It's not necessary to automate however, and the worst that can happen if CMX is not shut down is that the user accumulates a number of useless session specific directory trees in the `${HOME}/CMX` directory.

Before invoking `cmx login` or `cmx start`, a small number of environment variables must be set up. At SLAC this has also been automated by putting these initializations in a "group" script file which all users are requested to source from their login startup scripts. Please ask the manager of your site about site policies for setting up these environment variables. The environment variables in question are identified in section "CMX Internals".

4 The CMX Command

4.0 Syntax of CMX Commands

Frankly, I stole the command syntax for CMX from a previous effort I made at a code management system. The previous system had to work on both Unix and Windows/NT, so the syntax had to be compatible with the command line in both those environments. I never did find a way to make Unix look like NT or NT like Unix, so at that time I adopted the attitude “a plague on both your houses” and based my syntax on the old VMS command line. Because I’m so familiar with it, I have carried across that experience into CMX. Users familiar with the VMS command line 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 “ --“. (Beware the leading blank … it is mandatory.)
- Where VMS put parentheses around lists, omit the parentheses.

4.0.0 Syntactic Details

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

```
cmx build MSG --test --optimize --tags=mv2304
```

Verb	`build`
First parameter	`MSG`
First qualifier	`--test`
Second qualifier	`--optimize`
Third qualifier	`--tags`

To give you a sense of what’s going on, this means “build the package called MSG in the test branch, with optimization turned on and cross-compile using VxWorks tools targeting the Motorola MV2304 single board computer”.

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.

Qualifiers are introduced with “ --“ immediately followed by a keyword (no intervening blanks). Qualifiers can be valued (like `--tags` in the above example) or boolean (like `--test` and `--optimize`).

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 or the end of line).

Boolean qualifiers sometimes allow negation by prepending no to the keyword. To ask for no optimization:

```
cmx build MSG –-test –-nooptimize –-tags=mv2304
```

Qualifiers can appear anywhere on the line after the cmx keyword:

```
cmx –-test build MSG –-optimize –-tags=mv2304
```

Qualifier keywords can be shortened to their minimum unambiguous length:

```
cmx build MSG –-te –-o –-ta=mv2304
```

Parameters which are part of the cmx syntax can be shortened similarly:

```
cmx b MSG –-te –-o –-ta=mv2304
```

Obviously, if the parameter is something like a filename, that can't be shortened!

In some instances, CMX commands accept a comma delimited list as the value of a parameter or a qualifier. For example:

```
cmx show branch pkgA,pkgB,pkgC
```

In response, cmx will show the branch characteristics for the three packages pkgA, pkgB and pkgC. Similarly for qualifiers:

```
cmx build MSG –-test –-opt –-tags=rad750,mv2304,sun-gcc
```

If a parameter or qualifier value contains embedded blanks, the string must be quoted. This can be done the traditional unix way:

```
cmx commit MSG "–-comment=Comment message contains embedded blanks"
```

Though dedicated VMS fans may prefer the look of:

```
cmx commit MSG –-comment="Comment message contains embedded blanks"
```

4.0.1 Syntactic Shortfall

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

- This is Unix … everything is case sensitive.
- Unix loves to do command line processing. This should be familiar to Unix users as is the standard cure … lots of quotation marks!

4.1 Meta-Symbols Used In Command Descriptions

CMX attempts to present a self-consistent command syntax. This is useful to the user because experience with one command can help in guessing the syntax of another. It's also useful to manual writers because it's possible to set up a small set of meta-symbols which can be used across many command descriptions. This is what the following table seeks to do:

Note	Meta-Symbol	Meaning
	<file>	A file name (without path)
0	<files>	Comma delimited list of files (package relative)

Note	Meta-Symbol	Meaning
	<package>	Package name
1	<package_list>	Comma delimited list of package name patterns
	<project>	Project name
	<project_directory>	Location of the project directory in the site file system
	<tag>	A CMX/CMT tag
	<tags>	Comma delimited list of CMX/CMT tags
2	<test_directory>	Location of a package test directory
3	<version>	Package version name/number (a CVS tag string)

Table 2 Meta Symbols Used In Command Descriptions

Notes:

0 In command descriptions, <files> has a rather restrictive meaning. <files> only appears in commands which also define a package. The file names are then defined relative to that package. This is an example of adding a /src directory file called foo.c and a /cmt directory file called new_command for a package called PKGA:

```
cmx add PKGA src/foo.c,cmt/new_command
```

1 Rather than accept the name of a single package, some commands accept a comma delimited list of packages. Furthermore each member of the list is treated as a regular expression. A few examples might help:

```
cmx show branch MSG          Package MSG specifically.
cmx show branch MSG,CMX      Packages MSG and CMX specifically.
cmx show branch M.*          Packages that begin with M.
cmx show branch M.*,.*X      Packages that begin with M or end with X.
```

The only slight subtlety is that when using the names in the list as regular expressions, CMX implicitly adds a beginning of line and end of line to the pattern. Thus when the user types M.*, CMX actually uses the regular expression ^M.*$. This avoids the problem of one package name being an exact subset of another package name. Consider two packages already defined in CMX, BFD and BFDB:

```
cmx show branch BFD
```

Without the implicit beginning of line and end of line, CMX would find both BFD and BFDB. This is probably not what the user had in mind.

The list of commands which accept a package list/pattern might appear arbitrary at first sight, but there is a general rule of thumb being applied here. If a command is entirely passive (does not change the user's environment or the CMX database files) or if the result of executing the command could only harm a single user and a single session, the command will accept a package list/pattern. Potentially harmful commands will only accept a single package name.

2 A test directory is always specified by the project directory in which it lives and the project name should be the last directory specified in the test directory string. If the user has

successfully created a private project by issuing the command:

```
cmx create project PRJ ${HOME}/glast
```

Then to fetch a package PKGA into this location, the command would be:

```
cmx fetch PKGA –test=${HOME}/glast/PRJ
```

The string `${HOME}/glast/PRJ` is a well formed test directory specification.

3 A version string has a very specific format. It begins with the uppercase letter V, immediately followed by three groups of numbers separated by hyphens. In regular expression-speak this is `V[0-9]+-[0-9]+-[0-9]+` (i.e. pretty unintelligible, which is not uncommon for regular expressions). The numbers are the major, minor and patch versions of the *package*. Package version numbers are related but not identical to constituent interface numbers (more about these later).

Examples:

`V1-2-3`	Good	
`V42-7-999`	Good	
`V-1-2-3`	Bad	No hyphen allowed before first number.
`V0-0-`	Bad	Not enough numbers.
`v1r6`	Bad	CMX determines version syntax, not CMT!

This string is compatible with both CMT version naming conventions and CVS tag naming conventions. The most obvious use of this string is as a CVS tag name, but because the word tag is overloaded (a CMT tag should not be confused with a CVS tag), CMX tries to consistently refer to a CVS tag as a version.

4.2 CMX Commands

4.2.0 cmx add

Add files to the specified package (which must be set to branch test). This is a cover for the command `cvs add`.

```
cmx add <package> <files>
```

4.2.1 cmx build

The build command is one of the most complicated and most powerful commands provided by CMX. Its function is to build CMX packages, which it does by providing a shell around CMT's `gmake` command. It is unusual among CMX commands in that it allows extra command line tokens. If a token is not directly interpreted by the command, it is simply passed through to the `gmake` command. This is a powerful capability, but using it requires information about CMT's make file fragments and CMX's extensions to those fragments which haven't been discussed yet.

Note that CMX packages *must* be built using the `cmx build` command. Failure to do so (i.e. using the `gmake` command directly on the package's `Makefile`) will result in a not very informative nastygram.

```
cmx build <package>
    [--test | --development | --production | --version=<version>]
    [--[no]debug]
    [--[no]doxygen[=[fast|full]]]
    [--[no]optimize]
    [--alltags | --tags=<tags>]
    [<other tokens>]
```

The options which the `cmx build` command deals with directly:

- `[--test | --development | --production | --version=<version>]`

  Mutually exclusive options. Default is `--test`. This is here to avoid pratfalls. This option must agree with the branch set for this package in the user's environment (see the later commands `cmx set branch` and `cmx show branch`). Given that most packages in the user's environment will be set to the public production version, I wanted to avoid a carelessly typed command building said public production version.

- `[--[no]debug]`

  Force the production of debugging symbols on or off. The default is `--debug` which turns production on, so for practical purposes this option is only useful in its negated form.

- `[--[no]doxygen[=[fast|full]]`

  Qualifier to control documentation processing. The default is `--doxygen=fast` (equivalent to `--doxygen`) which turns normal documentation processing on. The form `--doxygen=full` turns on include and call graph generation (and the build command runs *much* slower). The negated form (which must not be valued) is only allowed when doing builds in the test branch.

The process that generates Doxygen documentation is now rather elaborate. In normal circumstances, a developer will take the defaults and not generate the graphs. When run in this way, the script that generates the documentation will also generate a file containing a one line command which will repeat the documentation step with graph generation turned on. The intention is for a cron job to go round in the wee hours and run all these files and then delete them. A user can generate the graphs on the fly by using the `--doxygen=full` option, but be warned, graph generation can be a slow and laborious process.

There's also a little intelligence built into the documentation generation script for sites that do not have the graph generation software installed. On such sites `--doxygen=full` will be treated the same as `--doxygen=fast`.

- `[--[no]optimize]`

  Force a particular optimization. If omitted the default is `--nooptimize` for `--test` builds and `--optimize` for all others.

- `[--alltags | tags=<tags>]`

  Mutually exclusive. If omitted, the default is `--tags=$CMX_C_CONFIG` where

`CMX_C_CONFIG` is an environment variable maintained by CMX. Tags in a tag list will be checked for validity (it's not possible to build CMT tag `linux-gcc` on a Sun platform!). To build all valid tags for the host machine use `--alltags`.

It is often useful to be able to trace the exact environment in which a package is built. This is particularly true of mix and match environments. For that reason, `cmx build` (in addition to all the other things it does) maintains build logs. These files, one per tag, can be found in the package's binary tree in the directory of the same name as the tag. Build logs record the timestamp of the build, the exact arguments passed to the `gmake` command, the outcome (success or failure) and probably most importantly of all, the exact CMT use tree extant when the build was run.

4.2.2 cmx check session

Do an integrity check on a CMX maintained session environment. This is a fairly obscure thing to want to do, but there can be cases, particularly when a session has been running for a long time, that the session CMX information can become desynchronized with the site or global CMX information. A simple example would be someone other than the session owner changing the site production version of a package. Such a change is *not* propagated to all sessions and can only be discovered by using this command.

All commands using environment variables run a check against the CMX site and global information. Any discrepancies will result in a recommendation for the user to issue this command. This is the most common way a user would end up here.

```
cmx check session
```

4.2.3 cmx check site

Compare a site's CMX installation against the global definition. The command reports missing projects, missing packages and packages whose site production version number differs from the global production version number. It does not attempt any fixes.

```
cmx check site [--[no]update]
```

- `[--[no]update]`

  Default is `--update`. Controls the updating of the CMX database files from the central CVS repository before the check is run.

4.2.4 cmx check version

Check the contents of CMX interface definition files. This is usually a preparatory step to making a CMX production release. All constituents in CMX packages must provide an interface definition file, which in addition to its role in the link and run-time environments, can be used as the basis of determining the version name to apply to a new production release. This topic is covered in more detail in section "Interface Versioning".

```
cmx check version <package_list>
```

4.2.5 cmx commit

Do a `cvs commit` for the specified files of the specified package.

```
cmx commit <package> [<files>]
    --comment=<comment>
   [--override-recursion-test]
```

- `--comment=<comment>`

  Provide a comment for the commit. Note that this qualifier is mandatory (and some very short comments might well be rejected). Quotes will be necessary to make this into a single token.

- `[--override-recursion-test]`

  When committing a CMT `requirements` file, the `cmx commit` command will run a recursion test to ensure that the use macros in the requirements file do not result in a circular use tree. CMX will not normally commit if such a recursion is discovered, but this behaviour can be overridden by using this qualifier. Note that this is such a powerful and useful check that in order to override it, the user will have to type this qualifier *in full* for it to take effect.

4.2.6 cmx compare

Compare a package in the filebase with either the head revision or a specified version in the CVS repository. It is in effect a cover for the `cvs –n update` command with all the correct options inserted.

```
cmx compare <package>
    --test=<test_directory> | --development | --production
   [--version=<version>]
```

- `--test=<test-directory> | --development | --production`

  The filebase part of the comparison.

- `[--version=<version>]`

  The CVS part of the comparison. If omitted, the comparison is made against head revisions.

There is no way with this command to compare an arbitrary filebase version against an arbitrary CVS version.

4.2.7 cmx create package

Create a new user package in a user private project. It creates the standard directory structure, installs templates for the `.cvsignore` and `requirements` files and executes a `cmt config` command to set the package up for use with CMX/CMT.

Note that packages can only be created in user private projects.

```
cmx create package <package> <test_directory>
```

4.2.8 cmx create project

Create a new project directory tree in either a user private filespace or in a site's public filespace.

```
cmx create project <project> <project_directory> [--global]
```

- `[--global]`

  Create the project globally (i.e. for all sites, but start with just this site). Default is to create a private project. In addition to creating the directory structure, global projects are recorded in CMX database files (`project.db` and `project.db.<site>` where `<site>` is the name the installer of CMX picked for your site).

Specifying the project directory can sometimes be a source of confusion. The simplest way to explain is to give an example. If a user wants to create a new private project tree called `PRJ` in directory `${HOME}/glast`, the correct command would be:

```
cmx create project PRJ ${HOME}/glast
```

The top level directory structure created by this command would look like:

```
${HOME}/glast/PRJ/source
${HOME}/glast/PRJ/binary
```

4.2.9 cmx create version

Create a new version of a package by tagging it in the CVS repository. To be successful, the package must meet some criteria:

- The package must already be present in CVS.
- The proposed version string must parse as a valid version string.
- The new version must be higher than any existing version for this package.

```
cmx create version <package> --version=<version>
```

4.2.10 cmx delete

Delete the publicly visible files associated with the specified version of a package (including the doxygen generated files). It does *not* remove the CVS tag in the repository. Furthermore, the command will reject requests to delete either the site defined or globally defined production version of a package.

This command is directed toward site managers who run into disk space problems and would like to delete old versions of packages. It should be used in conjunction with the `cmx show version` command.

```
cmx delete <package> --version=<version>
```

4.2.11 cmx fetch

Do a `cvs checkout` for the specified package. For the command to succeed, there must be no pre-existing directory structure at the destination (i.e. this command does not roll over to `cvs update` if the destination is already a CVS controlled tree).

```
cmx fetch <package>
    --test=<test_directory> | --development | --production
   [--version=<version>]
```

- `--test=<test-directory> | --development | --production`

  Mutually exclusive options. Defines the destination of the fetch. If the requested destination is `--production`, then the option `--version=<version>` must be provided (and the `<version>` string will be used to name the production area version directory). This command is almost never used with option `--development`.

- `[--version=<version>]`

  Fetch a particular version from the repository. Must be provided when requested destination is `--production`. Otherwise optional. If omitted, `cmx fetch` will check out the head revisions.

4.2.12 cmx help

Print a command description or a command syntax. Incomplete commands (e.g. `cmx help show`) are allowed in which case the help command will print the variants of the command. The help is intended to provide a quick reminder of a command's function or syntax. It is not intended to be a replacement for a manual.

```
cmx help [<command>] [--[no]description] [--[no]syntax]
```

- `[--[no]description]`

  Control the printing of the descriptive part of the help. Default is `--nodescription`.

- `[--[no]syntax]`

  Control the printing of the syntax part of the help. Default is `--syntax`.

4.2.13 cmx import

Import a new package into the CVS repository, tag it and re-export it to the public realm creating both a development version and an initial production version (with version number `V0-0-0`). An innocuous looking command, but very powerful.

```
cmx import <package> <project>
   [--override-recursion-test]
   [--dry-run]
```

- `[--override-recursion-test]`

  By default, the `cmx import` command runs a recursion test to ensure that the use macros in the package's requirements file does not result in a circular use tree. CMX will not normally import if such a recursion is discovered, but this behaviour can be overridden by using this qualifier. Note that this is such a powerful and useful check that in order to override it, the user will have to type this qualifier *in full* for it to take effect.

- `[--dry-run]`

  Go through the exercise of importing a package without actually doing it. With this option, an extensive summary of the actions CVS *would* take is printed on the screen.

`--dry-run` is a comparatively late addition to the `cmx import` command. On the other hand, better late than never. This is a very useful way to inspect the consequences of a `cmx import` command before actually committing to doing it.

4.2.14 cmx index

(Re)create the doxygen index pages from first principles. This command will overwrite any existing "Doxyidx.htm" or "Doxypkg.htm" files.

4.2.15 cmx ldd

This is the CMX analog of the `ldd` command. The native `ldd` command lists the shareables that will be activated when an executable/shareable provided as the first argument is invoked. Unfortunately, the shareable name listed is the first file name to satisfy the image activator's rules and may only be a link to another file. When the executable/shareable is a CMX shareable, it is *always* a link to another file. To compensate, `cmx ldd` will trace links and list the final resolution(s) of the file name(s) identified by `ldd`.

`cmx ldd` is also a little more forgiving than the host command. Host `ldd` expects a fully qualified executable/shareable name as its argument. `cmx ldd` tries this first, but if it can't find the file, it tries using the argument as an executable which can be looked up on `PATH` and then as a bare shareable which can be looked up on `LD_LIBRARY_PATH`.

```
cmx ldd <executable> [--full]
```

- `<executable>`

  Name of an executable/shareable.

- `[--full]`

  List all intermediate resolutions as well.

4.2.16 cmx login

Set up the CMX environment in a user terminal session. The command can be put in a user's login sequence if that's desirable. A synonym for the command `cmx start`.

```
cmx login
```

4.2.17 cmx logout

Shut down the CMX environment in a user terminal session. The command can be put in a user's logout sequence if that's desirable. A synonym for the command `cmx stop`.

```
cmx logout
```

4.2.18 cmx move

Move a package from one project to another.

```
cmx move <package> <project> [--global]
```

- `[--global]`

  CMX does not support the idea of forcing a change of parent project onto another site. It compensates by maintaining a global parent project (the currently accepted “best” parent project) and a site specific parent project for each package. If the caller omits this qualifier, the command updates only the local site’s parent project. If the caller provides the `--global` qualifier, the command updates both the local site’s parent project and the global parent project.

This is an extremely powerful and potentially destructive command. So much so that this is the only `cmx` command that will list the actions it’s about to take and demand user confirmation before proceeding.

4.2.19 cmx remove

Remove files from the specified package (which must be set to branch test). This is a cover for the command `cvs remove`.

```
cmx remove <package> <files>
```

4.2.20 cmx set branch

Set the branch for members of the package list. The choice affects the process/session in which the command is issued *only*.

```
cmx set branch <package_list>

    --test=<test_directory> |
    --development           |
    --production            |
    --version=<version>
```

- ```
  --test=<test_directory> |
  --development           |
  --production            |
  --version=<version>
  ```

  Branch to set. Mutually exclusive.

4.2.21 cmx set production

Identify a particular version of a package as the official production version.

```
cmx set production <package> --version=<version> [--global]
```

- `[--global]`

  CMX does not support the idea of forcing a new production version of a package across all sites. It compensates by maintaining a global production version (the currently accepted “best” production version) and a site specific production version for each package. If the caller omits the `--global` qualifier, the command updates only the local site’s version number of a package. If the caller provides the `--global` qualifier, the command updates both the local site’s version number and the global version number.
```

4.2.22 cmx set stem

Identify the directory where a project should be instantiated. *This should not be confused with the command* `cmx create project`. This command is only used when a project has already been set up at another site, but has not yet been instantiated on the caller's site.

```
cmx set stem <project> <project_directory>
```

4.2.23 cmx set vxworks

Set the VxWorks embedded system environment. At least qualifier must be specified.

```
cmx set vxworks
   [--architecture=<architecture>]
   [--license=<license>]
   [--registry=<registry>]
   [--vxworks=<vxworks>
```

- `--architecture=<architecture>`

  Specify the target architecture. Allowed architectures (and allowed architecture/version combinations) are determined from the CMX database file `arch.db.<site>`, but common values for architecture are `ppc` and `x86`.

- `--license=<license>`

  Redefine the value of the environment variable `WIND_LMHOST`. Not a particularly powerful or useful command, but provided as a convenience.

- `--registry=<registry>`

  Redefine the value of the environment variable `WIND_REGISTRY`. Not a particularly powerful or useful command, but provided as a convenience.

- `--vxworks=<vxworks>`

  Specify the VxWorks version. Allowed versions (and allowed architecture/version combinations) are determined from the CMX database file `arch.db.<site>`, but common values for version are `5.4` and `5.5`.

4.2.24 cmx show branch

List the branch settings in the local session for packages not set to the production branch for packages matching `<package_list>`. If `<package_list>` is omitted, the pattern `.*` (i.e. all packages) is used.

```
cmx show branch [<package_list>] [--full]
```

- `[--full]`

  List packages set to production as well.

4.2.25 cmx show hierarchy

Analyse the complete CMT use tree for all packages known to CMX then print any discontiguous hierarchies which contain a member of <package_list>. If <package_list> is omitted, a list of all packages is assumed.

Within a hierarchy, packages are ordered into “use order” from the top (packages which are not the target of another package’s use statement) to bottom (packages which do not have a use statement in them). Between these extremes, packages are displayed with boundaries showing the highest and lowest level they can occupy.

Sounds complicated (and it is!) and the output can be tricky to interpret, but this is invaluable to a site maintainer faced with updating a large number of packages and trying to figure out the order in which to do it.

```
cmx show hierarchy [<package_list>] [--left | --right]
```

- `[--left | --right]`

  Sort the package list for a hierarchy according to the highest level it can reach (`--left`) or the lowest level it can reach (`--right`). Default is `--left`.

This command has to work very hard to achieve its results, so don’t be surprised if it takes a while to execute. Don’t let that put you off though. This is a very interesting command.

4.2.26 cmx show project

List the public projects known to CMX (CMX does not keep records of user private projects).

```
cmx show project [--full]
```

- `[--full]`

  List the directories associated with the projects as well.

4.2.27 cmx show recursion

Analyse the complete CMT use tree for all packages known to CMX looking for cases where the use tree goes circular. It will report any such recursive loops that contain a package in <package_list>. If <package_list> is omitted, a list of all packages is assumed.

The underlying recursive analysis is used during `cmx commit` and `cmx import` commands to check that new or revised requirements files don’t introduce a recursion. This command accesses the same underlying analysis and prints the results. If no recursions are found, the command is silent.

```
cmx show recursion [<package_list>]
```

This command has to work very hard to achieve its results, so don’t be surprised if it takes a while to execute.

4.2.28 cmx show symbol

Scan through the complete set of binaries in the current environment using the nm command (or its VxWorks equivalent) looking for one or more symbols meeting the selection criteria specified in the qualifiers.

```
cmx show symbol <symbol_list>
    [--select=<select_list]
    [--alltags | --tags=<tag_list>]
```

- `<symbol_list>`

  Comma separated list of regular expressions to select the symbol name

- `[--select=<select_list>]`

  Comma separated list of nm symbol identifiers (please `man nm` for all the excruciating details of symbol identifiers). If omitted, defaults to `T,U` (i.e. find all the places symbol(s) is defined (`T`) or unresolved (`U`)).

- `[--alltags | tags=<tags>]`

  Select CMT tags to search. Mutually exclusive. `--alltags` creates a list of all viable CMT tags for the current platform and VxWorks architecture (if applicable). If omitted altogether the default tag in environment variable CMX_C_CONFIG is used.

Scanning the complete set of binaries in the current environment can be a very slow process, particularly when this command is used with the `--alltags` qualifier. Only unleash this command when you really need it and do not use this as an excuse to over-caffeinate yourself.

4.2.29 cmx show version

List all versions of the requested packages identifying those which are visible in the filebase (as opposed to those which are only defined by a tag in the CVS repository). The current site production versions and global production versions will be identified in the list.

This command is targeted at the site maintainer. During the development process, many back versions of packages accumulate in the filebase. This can lead to disk space problems. Use this command with the `--size` qualifier to identify back versions which can be deleted with the `cmx delete` command.

```
cmx show version <package_list> [--size]
```

- `[--size]`

  List the version sizes as well.

Looking up file sizes can be time consuming, so the `--size` qualifier may cause the command to execute slowly.

4.2.30 cmx show vxworks

List the characteristics of the current VxWorks environment.

```
cmx show vxworks
```

4.2.31 cmx start

Set up the CMX environment in a user terminal session. The command can be put in a user's login sequence if that's desirable. A synonym for the command `cmx login`.

```
cmx start
```

4.2.32 cmx stop

Shut down the CMX environment in a user terminal session. The command can be put in a user's logout sequence if that's desirable. A synonym for the command `cmx logout`.

```
cmx stop
```

4.2.33 cmx tornado

Based on technology provided by Dan Wood.

A scripting mechanism for target single board computers prepared for use with the WindRiver Tornado tool. This command is sufficiently complicated that it is described in more detail in section "CMX And VxWorks Tornado". This entry will only provide terse parameter/qualifier definitions.

```
cmx tornado <target_ip> <tag>
    [--common=<n>]
    [--exit=<exit_file>]
    [--initialize=<init_file>]
    [--interactive]
    [--log=<log_file>]
    [--quiet]
    [--retry=<n>]
    [--symbol=<n>]
    [--tcl]
    [--timeout=<n>]
```

- `<target_ip>`

  IP address for the single board computer to be addressed. Can be either a raw IP address or a node name.

- `<tag>`

  Yes this really is a CMT tag. Specifies the type of board at IP address `<target_ip>`.

- `[--common=<n>]`

Provide a default for the Tornado shell loader's "common symbol policy". Value must be a number between 0 and 2 (inclusive). Default is 1. See the Tornado API Guide for the meaning of these values.

- `[--exit=<exit_file>]`

  Specify a windSh script called `<exit_file>` to be run at exit.

- `[--initialize=<init_file>]`

  Specify a windSh script called `<init_file>` to be run at initialization.

- `[--interactive]`

  If specified, start an interactive windSh after running the initialization file (if an initialization file has been specified). Mutually exclusive with the `--log` option.

- `[--log=<log_file>]`

  Log output from the `<init_file>` and (optionally) the `<exit_file>` to `<log_file>`. If `--log` is specified, `--initialize` must also be specified. The qualifiers `--log` and `--interactive` are mutually exclusive.

- `[--quiet]`

  Use the `-quiet` option when starting windSh.

- `[--retry=<n>]`

  Specify the number of retries the WindRiver `tgtsvr` is allowed to attempt to reconnect to the target at startup or after a reboot. <n> is a number between 1 and 49 (inclusive). Default is 10. Used in conjunction with qualifier `--timeout`.

- `[--symbol=<n>]`

  Provide a default for the Tornado shell loader's "symbol visibility". Value must be a number between -1 and 1 (inclusive). Default is 0. See the Tornado API Guide for the meaning of these values.

- `[--tcl]`

  Use the `-Tclmode` option when starting windSh. In which case the scripts should obviously be written in TCL. This option has not been tested.

- `[--timeout=<n>]`

  Specify the timeout period (in seconds) for the WindRiver `tgtsvr` when it's trying to reconnect to the target at startup or after a reboot. <n> is a number between 1 and 49 (inclusive). Default is 2. Used in conjunction with qualifier `--retry`.

4.2.34 cmx update

Update the specified package from CVS. For the command to succeed, there must be pre-existing CVS controlled directory structure at the destination.

```
cmx update <package>
    --test=<test_directory> | --development | --production
    [--version=<version>]
```

- `--test=<test_directory> | --development | --production`

  Mutually exclusive options. Defines the target of the update. If the requested target is `--production`, then the option `--version` must be provided (though I find it very hard to imagine any circumstance where it would be valid to update a production branch).

- `[--version=<version>]`

  Update using a particular version from the repository. Must be provided when the requested target is `--production`. Otherwise optional. If omitted, `cmx update` will update using the head revisions.

4.2.35 cmx which

This is the CMX analog of the `which` (`tcsh`) or `type` (`sh/bash`) command. The host command lists the executable file that will be activated when a command name is typed at the terminal. Unfortunately, the executable file is the first file name to satisfy the command activation rules and may only be a link to another file. When the executable is a CMX executable, it is *always* a link to another file. To compensate, `cmx which` will trace links and list the final resolution of the file name identified by `which`.

`cmx which` is also a little more forgiving than the host command. The host command expects an executable name as its argument. `cmx which` tries this first, but if it can't find the file, it tries using the argument as a shareable file name which can be looked up on `LD_LIBRARY_PATH`.

```
cmx which <executable> [--full]
```

- `<exectuable>`

  Name of executable/shareable to trace.

- `[--full]`

  List all intermediate resolutions as well.

4.2.36 cmx who by

Construct a list of all packages directly used by the packages matching the entries in `<package_list>`. Synonym for the command `cmx who isused-by`.

```
cmx who by <package_list> [--full]
```

- `--full`

  Extend the list to include all packages directly or indirectly used by packages matching entries in `<package_list>`.

4.2.37 cmx who includes

Construct a list of all packages/constituents/files that include the specified (target) header file(s).

```
cmx who includes <package>/<file>[,<package>/<file>...] [--tag=<tag>]
```

- `<package>/<file>[,<package>/<file>...]`

  A comma delimited list of `<package>/<file>` combinations. It is assumed that `<file>` appears in the export directory of `<package>`.

- `--tag=<tag>`

  Perform the analysis for the specified tag. If omitted the default host tag is used.

This command searches *all* files known to CMX for those files that include the target(s). This can make the command quite slow. Nevertheless it allows CMX to perform a useful integrity test. All packages producing files that include the target(s) are tested to ensure that they have a 'use' relationship with the target package. If no such relationship exists, the offending package is flagged in the output with *> in the first two columns.
Please pay particular attention to this flag (it could indicate a very serious condition).

4.2.38 cmx who isused-by

Construct a list of all packages directly used by the packages matching the entries in `<package_list>`. Synonym for the command `cmx who by`.

```
cmx who isused-by <package_list> [--full]
```

- `--full`

  Extend the list to include all packages directly or indirectly used by packages matching entries in `<package_list>`.

4.2.39 cmx who uses

Construct a list of all packages which directly use the packages matching the entries in `<package_list>`.

```
cmx who uses <package_list> [--full]
```

- `--full`

  Extend the list to include all packages which directly or indirectly use the packages matching entries in `<package_list>`.

4.3 Command Classes

Having described the commands, it is now possible to divide them into a number of classes. The division is somewhat arbitrary, and I certainly don't want to discourage users from using commands simply because I've labeled them administrative, but a general classification might aid in general command navigation.

4.3.0 Maintenance Commands

```
cmx check site
cmx delete
cmx show version
```

Site maintainers will have most use for these commands, but all users to feel free to use at least `cmx check site` and `cmx show version` which are non-invasive. `cmx delete` should probably be left alone by the general user.

4.3.1 CVS Related Commands

```
cmx add
cmx commit
cmx compare
cmx fetch
cmx import
cmx remove
cmx update
```

These are bread and butter commands which will be used by almost all users. The standard development cycle will start with a `cmx fetch` command (to a test area) assorted `cmx add` and `cmx remove` commands as files are added and removed, and a `cmx commit` to put the code back into the CVS repository. `cmx compare` and `cmx update` comes in handy when two or more developers are working on the same piece of code simultaneously. `cmx import` is only slightly more exotic in that it is only used with new packages (though I'd like to emphasize that `cmx import` is powerful command … see also "Commands Affecting Global State").

The same commands can be used to maintain the development and production branches. This is a somewhat more administrative chore and the exact command syntax for these operations becomes a little more detailed.

Why provide these commands at all? Why not just use the CVS commands directly? For some operations, there will be no substitute for doing exactly that. The CMX commands certainly do not provide access to all the CVS options for instance. On the other hand, I think the CMX commands do provide some value added:

- CVS commands are the usual Unix mish-mash of curt, impenetrable options, most of which are never used. The CMX commands cover the vast majority of uses that developers really want without overloading those developers with huge amounts of detailed knowledge.
- The CMX commands can be issued from any directory anywhere. CVS commands require the developer to move to the appropriate directory.
- The CMX commands are CMT aware. When a CMX command results in the creation of a new branch of a package, the new branch is set up as a CMT package with the appropriate CMT commands.
- The CMX commands do a certain amount of validation which can keep the developer from harm. No system is foolproof, and I don't want to set myself up as den mother either, but a few well behaved commands can do wonders for a developer's confidence in the system.

4.3.2 Commands Affecting Global State

```
cmx create project --global
cmx import
cmx move –-global
cmx set production --global
```

These commands change entries in the CMX global database files, i.e. they have implications across all sites, not just the local site. Don't worry, CMX does not push state changes onto other sites, but after these commands are run at, say, siteA, a user at siteB will see various out-of-date messages in response to the command `cmx check site`. It's up to siteB to decide whether to accept these changes (which amounts to using other CMX commands to bring the CMX local database files in line with the global database files). For a fuller discussion of how CMX maintains state information, see section "CMX Internals".

4.3.3 Commands Relating To Process State

```
cmx ldd
cmx set branch
cmx show branch
cmx which
```

CMX ultimately controls the user's environment at the individual process level. These commands allow the user to control and inspect that environment. For a fuller discussion of how CMX maintains state information, see section "CMX Internals"

5 CMX And CMT

As stated in the introduction, CMT has been chosen as the build engine for GLAST LAT flight software. This section will detail how CMT has been modified and extended to support CMX. Note that this section will assume that users have familiarized themselves with the workings of CMT and in particular the syntax of a CMT requirements file. This section *augments* the information in the CMT manual. It is not a substitute.

This section will also make use of a number of meta-symbols which are defined in the following table:

Meta-Symbol	**Meaning**
`<pkg>`	A CMT package name
`<con>`	A CMT constituent name
`<version>`	A CMT version string
`<directory>`	A directory specification (used to localize a CMT package)

Out of the box, CMT has a number of problems:

- Does not support a cross-compilation environment like VxWorks
- Tries to generate `C` function prototypes. Gets them wrong.
- Tries to do include file analysis for `C` and `C++`. Gets it wrong.
- Does not do a very good job on shareables (treats them like a kind of super archive library).
- Still a little buggy.
- I could go on, but that's enough!

The above comments were accurate for CMT `v1r6 and v1r12`. CMT has since developed until it now stands at `v1r14`. Since CMT v1r12 is the stable version as we know, we are currently running v1r12.

That sounds pretty terrible, but CMT is a toolbox and it's possible to change its behaviour. The two major methods are:

- Hack on the source code of CMT itself

  I have avoided this option wherever possible. It's a maintenance nightmare trying to keep

a variant copy of CMT going while the original copy is still evolving. Unfortunately I have had to alter a small number of files in the CMT source.

- Overlay extra functionality on top of CMT using the hooks CMT provides.

  This (in part at least) is what the CMX package does.

Before getting into the details of what I did to CMT to make it suitable for GLAST LAT flight software, it would probably be worth outlining the behaviour I was pursuing:

- Fix or circumvent all the problems I listed at the start of this section!
- Establish a mix and match environment. This cannot be provided by CMT alone, but within CMT's scope this means enforcing the building of shareables only (CMX/CMT produces *no* archive libraries).
- Remain *compatible* with ground software usage of CMT. I emphasize the word compatible. I expect flight and ground software to maintain two separate infrastructures. Independent CVS repositories, independent copies of CMT, etc. The point of contact I'm trying to maintain is the CMT requirements file. Ground software should be able to access the *products* of flight software (include files, shareables) by `use`ing flight software requirements files. Ground software will not be able to *build* flight software packages (ground software CMT won't understand a complete flight software requirements file).

5.0 Direct Modifications Of CMT

We have edited the following files in CMT. They are:

5.0.0 src/cmt_use.cxx

This is the big one. I have changed how CMT interprets a use statement in a requirements file. The standard CMT syntax is:

```
use <package> [<version> [<directory>]]
```

CMT uses this line to localize the version of the package the user wants to use (i.e. find it in the filebase). CMT has a well documented search algorithm for this. I have simply augmented it so that before running its standard algorithm, it first looks up an environment variable based on the package name. If the environment variable doesn't exist, it continues with its standard algorithm. No harm, no foul (and completely backward compatible). If it does find the environment variable, it uses the value to do the localization. This solves a problem in CMT in that a project involving an extensive nest of requirements files, each containing a number of use statements can end up with ambiguities. `pkgA` in its requirements file says it wants to use `pkgB` and `pkgC`. `pkgB` in its requirements file says it wants to use `pkgZ` version 2, but `pkgC` in its requirements file says that it wants to use `pkgZ` version 3. CMT has resolution rules for these conflicts, but by using an environment variable to resolve the version, my technique never produces the ambiguity. Furthermore, by providing a tool (CMX) to manage that environment variable, I can switch in and out different versions of `pkgZ` without once editing a requirements file. This one change provides the basis for mix and match.

5.0.1 src/cmx_generator.cxx

Changes of the followings:

- commented out dependencies macros in the makefiles

- commented out constituents name in the output screen
- commented out the error output due to CMT dependencies

5.0.2 fragments/constituents_header

An extensively edited file. The one significant change is the removal of the library linking mechanism. This would interfere with my mix and match technology. I also have used this file to lump together all sorts of make file stuff to support CMX:

- Create the output binary directory if it's missing. CMT tries to do this, but the rule is implemented too early in the make chain (before CMX has an opportunity to touch the make variable which defines where the output binary directory is located).
- Enforce the use of the `cmx build` command to build CMX packages (i.e. abort builds started directly with `gmake`).
- Remove <pkg>setup.make file created by CMT, which have the same contain as build.log (created by CMX).
- Cosmetic changes to make output from this makefile look like all the others.

5.0.3 src/Makefile.header

Changes of the followings:

- Remove the dependency of "config" from "all" target, the dependency cause all target to create <tag>.make file twice, which is redundant.
- Remove the make command which use "first" as target. This target does nothing but prints build output.
- Cosmetic changes to make output from this makefile look like all the others.

5.0.4 fragments/make_header

Changes are the followings:

- Point the <constituent>.make files in binary/<pkg>/<ver>/<tag> to the <tag>.make in the same binary/<pkg>/<ver>/<tag> directory (<tag>.make is not under source/<pkg>/<ver> any more)

5.0.5 fragments/constituents_trailer, fragments/group, fragments/constituent, fragments/constituent_synchronized, fragments/application_header, fragments/check_application_header

Changes are extremely minimal and are mostly cosmetic, designed to make output from this makefile look like all the others

5.1 The CMX Overlay On A CMT Requirements File

This is where life gets interesting. The heart of the CMT build mechanism is the `requirements` file. Every package has a requirements file (in the `/cmt` directory) and it is the requirements file that describes to CMT what the user wants to build. A single requirements file can produce multiple products each of which appears in the requirements file as a *constituent*. Each constituent belongs to a constituent *type*. CMT defines two constituent types for its own purposes. These are the familiar `application` and `library` which produce an executable and an archive library respectively.

CMT also allows a user to extend the list of constituent types. This was originally intended to allow users to add documentation types (and it shows in the requirements file syntax). This is the mechanism CMX exploits. Probably the best feature of this mechanism is that once new constituent types are defined and the necessary make file fragments constructed to support them, handling of these types is completely independent of CMT's main line support (i.e. the standard `application` and `library`).

In addition to new constituent types, CMX expects a requirements file to provide a number of "magic" macros which it needs to do the things it does. Tables of document types and magic macros will be provided at the end of this section, but to get started I'd like to work through a series of requirements files of increasing complexity to show how things work.

5.1.0 A Host Only CMX Requirements File

Here is an annotated requirements file for a package producing a host shareable and a host executable, but no embedded system images. The example is based on the real CLI package, though the requirements file has been modernized and edited down to fit on one page:

```
    # ============================================================================
    # Public definitions
    # ============================================================================
    public
 0  package CLI
 1  author  apw@slac.stanford.edu
 2  manager apw@slac.stanford.edu
    # ----------------------------------------------------------------------------
    # Package usage
    # ----------------------------------------------------------------------------
 3  use GRL
    # ----------------------------------------------------------------------------
    # Export the link time access for this package's shareables.
    # ----------------------------------------------------------------------------
 4  macro CLI_bin "$(CLIROOT)/../../../binary/CLI/$(CLIVERSION)/$(CLI_tag)"
 5  macro CLI_cli__shr "" \
        linux-gcc "$(CLI_bin)/libcli.so" \
        win-gcc   "$(CLI_bin)/libcli.dll" \
        sun-gcc   "$(CLI_bin)/libcli.so"
    # ----------------------------------------------------------------------------
    # Export the run time access for this package's shareables.
    # ----------------------------------------------------------------------------
 6  macro CLI_shr "" \
        linux-gcc "cli" \
        win-gcc   "cli" \
        sun-gcc   "cli"
    # ----------------------------------------------------------------------------
    # Export the run time access for this package's executables.
    # ----------------------------------------------------------------------------
 7  macro CLI_exe "" \
        linux-gcc "TV" \
        sun-gcc   "TV" \
        sun-gcc   "TV"
    # ============================================================================
    # Private definitions
    # ============================================================================
 8  private
 9  use CMX
10  private
    # ----------------------------------------------------------------------------
    # Shareable cli
    # ----------------------------------------------------------------------------
11  macro cli__linkshr      "$(GRL_readline__shr)"
12  macro cli__buildtags    "linux-gcc win-gcc sun-gcc"
13  document shr cli \
        CLI_clearFiles.c \
        CLI_clearTokens.c \
        CLI_command.c \
        CLI_present.c \
        CLI_report.c \
        CLI_squeeze.c \
        CLI_syntax.c
    # ----------------------------------------------------------------------------
    # Executable TV
    # ----------------------------------------------------------------------------
    macro TV__linkshr   "$(CLI_cli__shr)"
    macro TV__buildtags  "linux-gcc win-gcc sun-gcc"
```

```
14  document exe TV \
        CLI_tv.c
```

Figure 3 A Host Only CMX Requirements File

0 This is standard CMT boilerplate. Every package must declare its name.

1 The author field has no real function in either CMT or CMX, but I would appreciate it if authors would sign their e-mail address here.

2 Likewise the manager field has no real function, but if the person currently maintaining the package differs from the original author, I'd appreciate it if the maintainer would sign their e-mail address here.

3 This is classic CMT syntax with a twist. In native CMT a use statement that does not specify at least part of the version string for the package being used is frowned upon as being too non-specific. In CMX this is perfectly normal usage because the used package is localized from an environment variable. There is nothing to stop a use command in a CMX requirements file from including the full CMT use syntax, but the environment variable resolution is evaluated first so any extra tokens on a line like this are ignored. Under those circumstances I personally prefer that tokens that can never have any meaning never appear in the file.

4 This macro is defined for convenience only, but it is so convenient that it has risen to the level of a convention. Nearly all packages now define a macro `<pkg>_bin` for use later in the file. This is not a magic macro because it's is never interrogated by either CMT or CMX.

5 This macro is not magic either, but it's the conventional method for a package to identify its shareable products to other packages so that the other package can find them *at link time*. Because it's not magic, there are no mandated rules for the formation of this macro name, but modern convention is that it be named `<pkg>_<con>__shr`. Older requirements files may still follow the previous convention of `<pkg>_lnk_<con>`.

6 The first genuinely magic macro. This macro is read by CMX and *must* follow an exact naming convention. The macro name must be `<pkg>_shr`, and its value is a blank delimited list of the package's shareable products (or at least, the shareables the package wishes to export). On the basis of this macro, CMX will construct symbolic links so that the shareable(s) are findable by a search of the environment variable `LD_LIBRARY_PATH`. This is a host specific mechanism so it makes no sense to add entries for embedded system tags (which this file does not).

7 The second genuinely magic macro. This macro is read by CMX and *must* follow an exact naming convention. The macro name must be `<pkg>_exe`, and its value is a blank delimited list of the package's executable products (or at least, the executables the package wishes to export). On the basis of this macro, CMX will construct symbolic links so that the executable(s) are findable by a search of the environment variable `PATH`. This is a host specific mechanism so it makes no sense to add entries for embedded system tags (which this file does not).

There's another magic macro analogous to the previous two for exposing script files. Script files to be exposed by this method must appear in the package's `/cmt` directory and the magic macro name is `<pkg>_cmt`. Such scripts are made available via symbolic link on the environment variable `PATH`.

8 The private directive is there in the hope that CMT's author will actually make this work properly. The idea is that when another package uses this package, it cannot see anything in this package's private section. Thus it would not see the next line which says `use CMX`. A package prepared this way could then be the target of a ground software package's `use` statement without ground software having to worry about finding package CMX.

9 This `use CMX` statement makes all of the CMX extensions to the CMT build environment available. Any package expecting to be built with CMX must include such a statement.

10 This simply reflects just how broken the public/private mechanism is in this version of CMT. Despite the private statement two lines previous, CMT can return from reading the CMX requirements file with context set to public.

11 A magic macro with name `<con>__linkshr`. This macro is a blank delimited list of *shareables* to be used when linking the product. Shareables should be the *only* thing to appear on this list (do not use it to pass arbitrary link time options … there's another mechanism available to do this). If the constituent doesn't link against any other shareables, this macro can be omitted.

This example is very characteristic. Package CLI needs access to the GNU readline (GRL) package, so at the top of the file this requirements file says `use GRL` and then at link time it specifies the shareable product provided by package GRL.

12 A magic macro to specify the CMT tags for which this package can be built in the form `<con>__buildtags`. This requirements file is for a host only package so it should be no surprise that the tags listed don't include any embedded system tags. This macro can be omitted without causing any kind of failure other than the fact that this constituent would never be built, so consider it mandatory.

13 Finally, a description of what the user wants built to make a particular constituent. This is the first example of a CMX defined constituent type. The first three tokens in the line are: `document`: a CMT mandated keyword (I told you this mechanism was originally designed to add documentation types), `shr`: a CMX defined constituent type and `cli`: a constituent name. The `shr` constituent type builds a shareable for a host target tag.

14 This introduces another constituent type. The `exe` constituent type builds an executable for a host target tag.

5.1.1 A Host And Embedded System CMX Requirements File

This example extends the requirements file to produce products for both unix hosts and embedded systems. Once again it is based on a real package (TTC in this case) but the requirements file has again been edited for size and illustrative purposes.

```
    # ============================================================================
    # Public definitions
    # ============================================================================
    public
    package TTC
    author  apw@slac.stanford.edu
    manager apw@slac.stanford.edu
    # ----------------------------------------------------------------------------
    # Package usage
    # ----------------------------------------------------------------------------
    use CLI
    use BFU
    use BFD
    use TCU
    use BCI
    use BFW
    use BGW
    use BIUD
    # ----------------------------------------------------------------------------
    # Export the link time access for this package's shareables.
    # ----------------------------------------------------------------------------
    macro TTC_bin "$(TTCROOT)/../../../binary/TTC/$(TTCVERSION)/$(TTC_tag)"
0   macro TTC_ttc__shr "" \
        linux-gcc   "$(TTC_bin)/libttc.so" \
        win-gcc     "$(TTC_bin)/libttc.dll" \
        sun-gcc     "$(TTC_bin)/libttc.so" \
        rad750      "$(TTC_bin)/libttc.so" \
        mv2304      "$(TTC_bin)/libttc.so"
    macro TTC_ttcdmp__shr "" \
        linux-gcc   "$(TTC_bin)/libttcdmp.so" \
        win-gcc     "$(TTC_bin)/libttcdmp.dll" \
        sun-gcc     "$(TTC_bin)/libttcdmp.so" \
        rad750      "$(TTC_bin)/libttcdmp.so" \
        mv2304      "$(TTC_bin)/libttcdmp.so"
    macro TTC_ttc_tx__shr "" \
        linux-gcc   "$(TTC_bin)/libttc_tx.so" \
        win-gcc     "$(TTC_bin)/libttc_tx.dll" \
        sun-gcc     "$(TTC_bin)/libttc_tx.so"
    # ----------------------------------------------------------------------------
    # Export run time access to this package's (host side only) shareables.
    # ----------------------------------------------------------------------------
    macro TTC_shr "" \
        linux-gcc   "ttc ttcdmp ttc_tx" \
        win-gcc     "ttc ttcdmp ttc_tx" \
        sun-gcc     "ttc ttcdmp ttc_tx"
    # ----------------------------------------------------------------------------
    # Export run time access to this package's (host side only) executables.
    # ----------------------------------------------------------------------------
    macro TTC_exe "" \
        linux-gcc   "TTC" \
        win-gcc     "TTC" \
        sun-gcc     "TTC"
    # ============================================================================
    # Private definitions
    # ============================================================================
    private
```

```
   use CMX
   private
   # ------------------------------------------------------------------------
   # Shareable: ttc
   # ------------------------------------------------------------------------
 1 macro          ttc__buildtags  "linux-gcc win-gcc sun-gcc rad750 mv2304"
 2 macro          ttc__linkshr    "" \
       linux-gcc "$(BCI_bci__shr) $(BGW_bgw__shr) $(TCU_tcu__shr)" \
       win-gcc   "$(BCI_bci__shr) $(BGW_bgw__shr) $(TCU_tcu__shr)" \
       sun-gcc   "$(BCI_bci__shr) $(BGW_bgw__shr) $(TCU_tcu__shr)" \
       rad750    "$(BCI_bci__shr) $(BGW_bgw__shr) $(TCU_tcu__shr)" \
       mv2304    "$(BCI_bci__shr) $(BGW_bgw__shr) $(TCU_tcu__shr)"
   document shr   ttc \
       TTC_dispatch.c \
       TTC_setup.c \
       TTC_transaction.c
   # ------------------------------------------------------------------------
   # Shareable: ttcdmp
   # ------------------------------------------------------------------------
   macro          ttcdmp__buildtags  "linux-gcc win-gcc sun-gcc rad750 mv2304"
 3 macro          ttcdmp__removedox  "src/TTC_prvdefs.h"
   macro          ttcdmp__linkshr    "" \
       linux-gcc "$(TTC_ttc__shr)" \
       win-gcc   "$(TTC_ttc__shr)" \
       sun-gcc   "$(TTC_ttc__shr)" \
       rad750    "$(TTC_ttc__shr)" \
       mv2304    "$(TTC_ttc__shr)"
   document shr   ttcdmp \
       TTC_dumpTrans.c
   # ------------------------------------------------------------------------
   # Shareable: ttc_tx
   # ------------------------------------------------------------------------
   macro          ttc_tx__buildtags  "linux-gcc win-gcc sun-gcc"
   macro          ttc_tx__removedox  "src/TTC_prvdefs.h"
   macro          ttc_tx__linkshr    "" \
       linux-gcc "$(TTC_ttc__shr) $(TCU_tcu__shr)    $(TCU_tcudmp__shr) \
                  $(CLI_cli__shr) $(CLI_clidmp__shr) $(BCI_bci_tx__shr) \
                  -lnsl" \
       win-gcc   "$(TTC_ttc__shr) $(TCU_tcu__shr)    $(TCU_tcudmp__shr) \
                  $(CLI_cli__shr) $(CLI_clidmp__shr) $(BCI_bci_tx__shr) \
                  -lnsl -lsocket"
       sun-gcc   "$(TTC_ttc__shr) $(TCU_tcu__shr)    $(TCU_tcudmp__shr) \
                  $(CLI_cli__shr) $(CLI_clidmp__shr) $(BCI_bci_tx__shr) \
                  -lnsl -lsocket"
   document shr   ttc_tx \
       TTC_callbacks.c \
       TTC_library.c \
       TTC_syntax.c
   # ------------------------------------------------------------------------
   # Executable: TTC
   # ------------------------------------------------------------------------
   macro          TTC__buildtags  "linux-gcc win-gcc sun-gcc"
   macro          TTC__removedox  "TTC/TTC_pubsntx.h"
   macro          TTC__linkshr    "" \
       linux-gcc "$(CLI_cli__shr) $(TTC_ttc_tx__shr) \
                  $(TTC_ttc__shr) $(TCU_tcu__shr)" \
```

```
        win-gcc   "$(CLI_cli__shr) $(TTC_ttc_tx__shr) \
                   $(TTC_ttc__shr) $(TCU_tcu__shr)"
        sun-gcc   "$(CLI_cli__shr) $(TTC_ttc_tx__shr) \
                   $(TTC_ttc__shr) $(TCU_tcu__shr)"
    document exe   TTC \
        TTC.c
```

Figure 4 A Host And Embedded System CMX Requirements File

0 At first glance, this macro definition looks absurd. It's well known that the object file coin of the realm for embedded systems is an incrementally linked object. The shareable object file format is meaningless to embedded systems. Not so fast! While it is true that a shareable file is unusable in an embedded system, it can be very valuable as an analysis tool. When document type `shr` is built for an embedded system tag, processing produces three files. The primary output is an entirely conventional, incrementally linked file named `lib<con>.o`. In addition, processing produces a relinkable file named `lib<con>.ro` (`.ro` files have a rather obscure role and a discussion of them will be deferred until later) and the shareable file named `lib<con>.so (lib<con>.dll for windows).` The value of the shareable file is its ability to participate in unresolved external reference analysis at *link* time. With the standard embedded system incrementally linked object, no unresolved external analysis is possible and unresolved references aren't detected until run time (which given the cruder embedded system run time environment can be inconveniently late).

   The rule of thumb for this macro is that there should be an entry for every tag for which the corresponding `shr` constituent is built.

1 Notice that shareables are defined even for embedded system tags. This is the other half of the mechanism described in the previous bullet. If these shareables were not defined, the build for the embedded system product would fail during unresolved external reference analysis.

2 This demonstrates the ability of CMX to build the same shareable for multiple targets including both host and embedded systems. The same is true for document type `exe`, though the product(s) when a document type `exe` is built for an embedded system tag are identical to document type `shr` product(s).

3 This is a new magic macro consumed by the document generation system. The macro name must be `<con>__removedox` and its value is a blank delimited list of files (specified package relative, i.e. the name begins with a directory name like `src/` or `<pkg>/`) which should be removed from the list of files submitted to the document generator doxygen.

   There is an equivalent magic macro `<con>__insertdox` to add extra files to the file list submitted to the documentation generator. This can be particularly useful for executables where the extra file can be a syntax definition along the lines of a unix man page.

5.1.2 A VxWorks Kernel/BSP CMX Requirements File

This is a very specialized CMX extension to CMT provided specifically to build VxWorks kernel/BSP products. Not many people will be interested in this art form, but because it introduces another document type and more magic macros, it is included for completeness.

5.1.2.0 A VxWorks Kernel/BSP CMX Requirements File, Not Standalone

Once again the example is based on a real package called MV2X, but the file has been edited and shortened for illustrative purposes (and my apologies to Dan Wood for playing fast and loose with his file):

```
   # ============================================================================
   # Public definitions
   # ============================================================================
   public
   package MV2X
   author  dwood@xip.nrl.navy.mil
   manager dwood@xip.nrl.navy.mil
   # ----------------------------------------------------------------------------
   # Export the linkage for this package's shareables.
   # ----------------------------------------------------------------------------
   macro MV2X_bin \
       "$(MV2XROOT)/../../../binary/MV2X/$(MV2XVERSION)/$(MV2X_tag)"
   macro MV2X_mv2x_usr__shr "" \
       rad750        "$(MV2X_bin)/mv2x_usr/libmv2x_usr.so" \
       mv2304        "$(MV2X_bin)/mv2x_usr/libmv2x_usr.so"
   macro MV2X_mv2x_extra__shr "" \
       rad750        "$(MV2X_bin)/mv2x_extra/libmv2x_extra.so" \
       mv2304        "$(MV2X_bin)/mv2x_extra/libmv2x_extra.so"
   macro MV2X_mv2x_rom__shr "" \
       rad750        "$(MV2X_bin)/mv2x_rom/libmv2x_rom.so" \
       mv2304        "$(MV2X_bin)/mv2x_rom/libmv2x_rom.so"
   macro MV2X_mv2x_rtos__shr "" \
       rad750        "$(MV2X_bin)/mv2x_rtos/libmv2x_rtos.so" \
       mv2304        "$(MV2X_bin)/mv2x_rtos/libmv2x_rtos.so"
   # ----------------------------------------------------------------------------
   # Export the list of commands/scripts this package supports (per tag)
   # ----------------------------------------------------------------------------
   macro MV2X_cmt "" \
       sun-gcc     "mv2x_term"
   # ============================================================================
   # Private definitions
   # ============================================================================
   private
   use CMX
   private
   # ----------------------------------------------------------------------------
   # BSP: mv2x_rtos
   # ----------------------------------------------------------------------------
   macro mv2x_rtos__buildtags    "rad750 mv2304"
0  macro mv2x_rtos__cmp_flags \
       "-I$(MV2XROOT)/MV2X \
        -I$(WIND_BASE)/target/config/all \
        -I$(WIND_BASE)/target/src/drv \
        -I$(WIND_BASE)/target/src/config \
        -fno-builtin -fno-for-scope"
1  macro mv2x_rtos__linkopt "" \
       rad750        "$(WIND_BASE)/target/lib/libPPC603gnuvx.a" \
       mv2304        "$(WIND_BASE)/target/lib/libPPC604gnuvx.a"
   macro mv2x_rtos_entry        "00100000"
2  macro mv2x_rtos__linkbsp     "-Ttext $(mv2x_rtos_entry) -e _sysInit"
3  document bsp mv2x_rtos \
       sysALib.S \
       sysLib.c \
       usrConfig.c \
       version.c
   # ----------------------------------------------------------------------------
```

```
# Shareable: mv2x_extra
# ------------------------------------------------------------------------------
macro mv2x_extra__buildtags        "rad750 mv2304"
macro mv2x_extra__linkshr "" \
    rad750        "$(MV2X_mv2x_rtos__shr)" \
    mv2304        "$(MV2X_mv2x_rtos__shr)"
document shr mv2x_extra \
    mk48T.c \
    mk48TDos.c \
    ravenDisconnect.c \
    universeLock.c
# ------------------------------------------------------------------------------
# Shareable: mv2x_rom
# ------------------------------------------------------------------------------
macro mv2x_rom__buildtags        "rad750 mv2304"
macro mv2x_rom__linkshr "" \
    rad750        "$(MV2X_mv2x_rtos__shr)" \
    mv2304        "$(MV2X_mv2x_rtos__shr)"
document shr mv2x_rom \
    i28F800.c \
    usrRom.c \
    romDrv.c
# ------------------------------------------------------------------------------
# Shareable: mv2x_usr
# ------------------------------------------------------------------------------
macro mv2x_usr__buildtags        "rad750 mv2304"
macro mv2x_usr__linkshr "" \
    rad750        "$(MV2X_mv2x_rtos__shr)" \
    mv2304        "$(MV2X_mv2x_rtos__shr)"
document shr mv2x_usr \
    usrLib.c
```

Figure 5 A VxWorks Kernel/BSP CMX Requirements File (not standalone)

0 This is a magic macro provided for the user to inject compilation options. It works with all document types. There are three variants of this magic macro. The form `<con>__cmp_flags` applies the compilation flags to all files defined in the constituent. The form `<file>___cmp_flags` applies to the compilation flags to a particular file wherever it appears (possibly in multiple constituents). The form `<con>__<file>___cmp_flags` applies the compilation flags specifically to the indicated file in the indicated constituent. `<file>` is the truncated filename without directory specification or filename extension. Thus to refer to the file `usrConfig.c` in constituent `mv2x_rtos`, `<file>` would be `usrConfig`. Please note that each source file goes through two compilation steps, the first to do include file analysis and the second to do the real compilation. The compilation options generated by the magic macros described here are applied in both steps.

Astute readers may notice the emergence of another convention in magic macro names. When `<pkg>` appears in a macro, it is followed by a single underscore. When `<con>` appears in a macro, it‘s followed by two underscores. When `<file>` appears in a macro it’s followed by three underscores. This isn’t a bad convention, and goes a long way to keeping the namespace well separated. Any future development of CMX will follow this convention. Unfortunately the convention wasn’t in place during the early CMX development, so it’s possible to find examples where it doesn’t hold true.

1 <con>__linkopt is a magic macro which allows the user to inject link time options and is applied for all document types. This is the right place to put link time options which are *not* shareables. In the example given, mv2x_rtos__linkopt is used to include the VxWorks provided archive library in the link step.

2 <con>__linkbsp is a magic macro specific to kernel/BSP building and is meaningless (and ignored) for document types shr and exe. It is used to define link options to the final link step. Yes the final link step. Behind the scenes, CMX is working very hard to do constituent linking. It currently takes one link pass to generate a Sun host shr or exe, three passes to generate a Linux host shr or exe and five or six passes to produce any document type for an embedded system. The CMX link step is complex enough that a description of it has been promoted to a section in its own right (see “The Link Step Of CMX Build”). In the context of this bullet, <con>_linkbsp is only applied to the fifth step of a kernel/BSP build.

3 Finally the new constituent type appears. The name is bsp and its only use is the construction of kernel/BSP products. It never makes sense to build a constituent of type bsp for a host system.

5.1.2.1 A VxWorks Kernel/BSP CMX Requirements File, Standalone

Once again the example is based on a real package called MV2X and once again, the file has been edited and shortened for illustrative purposes. Note that this example and the previous one could be merged so that the MV2X package produces both a non-standalone version and a standalone version of the kernel/BSP.

```
   # ============================================================================
   # Public definitions
   # ============================================================================
   public
   package MV2X
   author  dwood@xip.nrl.navy.mil
   manager dwood@xip.nrl.navy.mil
   # ----------------------------------------------------------------------------
   # Export the linkage for this package's shareables.
   # ----------------------------------------------------------------------------
   macro MV2X_bin \
       "$(MV2XROOT)/../../../binary/MV2X/$(MV2XVERSION)/$(MV2X_tag)"
   macro MV2X_mv2x_standalone__shr "" \
       rad750        "$(MV2X_bin)/mv2x_standalone/libmv2x_standalone.so" \
       mv2304        "$(MV2X_bin)/mv2x_standalone/libmv2x_standalone.so"
   # ============================================================================
   # Private definitions
   # ============================================================================
   private
   use CMX
   private
   # ----------------------------------------------------------------------------
   # BSP: mv2x_standalone
   # ----------------------------------------------------------------------------
0  macro target_architecture "" \
       rad750   "mv2304" \
       mv2304   "mv2304"
   macro mv2x_standalone__buildtags    "rad750 mv2304"
   macro mv2x_standalone__cmp_flags \
       '-I$(MV2XROOT)/MV2X \
        -I$(WIND_BASE)/target/config/all \
        -I$(WIND_BASE)/target/src/drv \
        -I$(WIND_BASE)/target/src/config \
        -I$(WIND_BASE)/target/config/$(target_architecture) \
        -DFLASH '
1  macro mv2x_standalone__usrConfig___cmp_flags "-DSTANDALONE"
2  macro mv2x_standalone__linkopt "" \
       rad750   "-defsym usrVxwVer=0x50000001 \
                 $(WIND_BASE)/target/lib/libPPC603gnuvx.a" \
       mv2304   "-defsym usrVxwVer=0x50000001 \
                 $(WIND_BASE)/target/lib/libPPC604gnuvx.a"
3  macro mv2x_standalone__linkbsp "" \
       rad750   "-Ttext 00100000 -e _sysInit" \
       mv2304   "-Ttext 00100000 -e _sysInit"
4  macro mv2x_standalone__linkrom "" \
       rad750   "-Ttext 00280000 -e _romInit \
                 $(WIND_BASE)/target/lib/libPPC603gnuvx.a" \
       mv2304   "-Ttext 00280000 -e _romInit \
                 $(WIND_BASE)/target/lib/libPPC604gnuvx.a"
5  macro mv2x_standalone__linkvar "--romstub=bootInit.o,romInit.o \
                                   --romsize=0x00100000"
   document bsp mv2x_standalone \
6      bootInit.c \
7      romInit.S \
       sysALib.S \
       sysLib.c \
       usrConfig.c \
```

```
        version.c
```

Figure 6 A VxWorks Kernel/BSP CMX Requirements File (standalone)

0 This is just for convenience (it's not magic). In this build, it was necessary to find a header file in an architecture specific directory. This was an easy way to feed the (partial) name of the directory into the `mv2x_standalone__cmp_flags` macro.

1 This is a demonstration of how to feed a C macro specifically into one file in the build.

2 `<con>__linkopt` has been encountered before. The use here is the same as previous, but in light of what's coming in later annotations, it should be pointed out that it is these options that are used in link steps 1, 2, 3 and 4.

3 `<con>__linkbsp` has also been noted before and once again, its use is the same. Link options provided by this magic macro are directed specifically to link step 5.

4 `<con>__linkrom` is new and only appears in the preparation of a standalone kernel/BSP. Link options defined here are directed specifically at link step 6 when the kernel/BSP is wrappered in a ROM stub.

5 `<con>__linkvar` is also new. This is a magic macro and is used to annotate "variants". In this case the `--romstub` and `--romsize` qualifiers indicate that the variant target is a standalone version (so there will be a sixth link step where the contents of `<con>__linkrom` will be used). The values defined by the `--romstub` and `--romsize` qualifiers are also significant. Object files defined in `--romstub` will be *omitted* from link steps one to five and will only go into link step 6. `--romsize` is used after the final link step to check that the resulting standalone system will fit into the available EEPROM on the target board.

6 (and 7). These are the wrapper routines that pull the compressed standalone operating system image out of EEPROM and inflate it into RAM. Once they've done their job, execution branches to the conventional `_sysInit` entry point and the operating system does its normal startup.

5.1.3 A Cover Requirements File

This type of requirements file does not really follow this section's model of annotating requirements files of increasing complexity so please consider this whole heading as an aside.

Cover requirements files offer an interesting technique which users should be aware of. The basic principles are described in the CMT manual and I have put some work into making CMX support them, but the implementation should at this stage be regarded as a little experimental.

Many CMX host programs, both shareable and executable, would like to link against standard libraries and header files. This is a no-brainer for obvious candidates like the C run time library and its menagerie of supporting header files. It's not quite so obvious for more obscure facilities. A case in point is the `readline` facility which is used by the CMX package CLI. Both the Sun and Linux systems provide this facility, but not in any standard form. On Sun it's not even available as a shareable but only as an archive library. Fortunately `readline` is a small facility and in the name of expediency (CLI was written for the balloon flight) the easiest solution was to grab the `readline` source code, mash it around into the form of a CMX package and build it. Dan Wood went one better and faced with a similar situation with `Tcl/Tk`, took the source code for that application and turned it into a CMX package as well. This sledgehammer approach certainly has some advantages. In the case of `Tcl/Tk`, we are guaranteed that all sites are running the same version and migrating between versions is completely under our control. On the other hand, I suspect that putting the TCL package together cost Dan a lot of time and effort.

An alternative approach is to write cover requirements files. I've experimented with this, producing a CMX package called MySQL which is a cover for a conventionally built (read automake/autoconf/libtools) MySQL application.

MySQL is a free random access database implementation. It's a large and complicated piece of code, so I would *not* consider it a candidate for munging into a CMX/CMT package.

This is what a cover requirement s file looks like:

```
   # ============================================================================
   # Public definitions
   # ============================================================================
   public
   package MySQL
   author  apw@slac.stanford.edu
   manager apw@slac.stanford.edu
   # ----------------------------------------------------------------------------
   # Override the CMX patterns which create the standard header export directory
   # ----------------------------------------------------------------------------
0  ignore_pattern CMX_branches
1  ignore_pattern CMX_exp
   # ----------------------------------------------------------------------------
   # Define useful intermediate macros
   # MySQL_r    Root directory of MySQL build area
   # MySQL_h    Take out host directory variation (and include a fixed string)
   # MySQL_l    Route down to the MySQL shareable libraries
   # ----------------------------------------------------------------------------
2  macro MySQL_r "" \
       linux-gcc-slac   "/afs/slac/g/glast/applications/mysql/3.23.16-alpha" \
       sun-gcc-slac     "/afs/slac/g/glast/applications/mysql/3.23.16-alpha"
3  macro MySQL_h "" \
       linux-gcc-slac   "i386_linux22/mysql-3.23.16-alpha" \
       sun-gcc-slac     "sun4x_56/mysql-3.23.16-alpha"
4  macro MySQL_i "" \
       linux-gcc-slac   "$(MySQL_r)/$(MySQL_h)/include" \
       sun-gcc-slac     "$(MySQL_r)/$(MySQL_h)/include"
   # ----------------------------------------------------------------------------
   # Override the standard CMX macro which defines where to find the exported
   # header files and put in the real MySQL header file include path.
   # ----------------------------------------------------------------------------
5  ignore_pattern CMX_include_path
6  include_path  "$(MySQL_i)"
   # ----------------------------------------------------------------------------
   # Export the link time access for this package's shareables.
   # ----------------------------------------------------------------------------
   macro MySQL_mysqlclient__shr \
       "$(MySQL_r)/$(MySQL_h)/libmysql/.libs/libmysqlclient.so" \
   # ----------------------------------------------------------------------------
   # Export run time access to this package's (host side only) shareables.
   # ----------------------------------------------------------------------------
7  macro MySQL_shr             "mysqlclient"
8  macro MySQL_shr_mysqlclient "$(MySQL_r)/$(MySQL_h)/libmysql/.libs"
   # ============================================================================
   # Private definitions
   # ============================================================================
   private
   use CMX
```

Figure 7 A Cover Requirements File

0 By default, CMX is set up to enforce a standard package directory structure. In particular, a CMX created package will always have a directory in parallel with the `/src` directory named after the package, which is used as the standard location for exported header files. When working with covers for external applications this is obviously useless, so the creation of these directories must be suppressed.

1 This is just another part of the suppression process outlined in 0.

2 This group of macros has been set up to make the file a little more readable. They also introduce the heart of the cover requirements file mechanism. This is the first time CMT tags like `linux-gcc-slac` and `sun-gcc-slac` have appeared. The tag is constructed as one of the conventional CMX tags with `-<site>` appended. These tags are sensitive, not to the usual tag definition given in (for instance) a `cmx build` command, but to the value of an environment variable called `CMTSITE`. Behind the scenes, CMX is arranging for `CMTSITE` to be set appropriately, so you will not see this environment variable listed in your environment.

3 See 2.

4 See 2.

5 When CMT builds a package, it constructs a grand include path by accumulating the include path contributions from each package in the use tree descending from the package being built. The grand include path is then available during compilation so that all the necessary header files can be found. This is one of the great convenience features provided by CMT. By default, a CMX package always contributes the standard package header export directory. Unfortunately when the requirements file is a cover requirements file the standard header export directory provided by vanilla CMX is just plain wrong (and doesn't even exist because of the actions annotated in 0 and 1 above). This line suppresses the standard include path contribution.

6 This completes the work started in 5 by defining a different include path pointing to the directory that in fact contains the MySQL header files.

7 This line has been annotated because there's more going on here than meets the eye. Now that CMX is making a cover for a real world object, it has to deal with real world conventions. The `mysqlclient` shareable which CMX is seeking to make available follows the increasingly common Unix shareable presentation conventions. Look in the real MySQL `/.libs` directory and you will find a symbolic link called `libmysqlclient.so` which points to a symbolic link called `libmysqlclient.so.9` which points to the real shareable called `libmysqlclient.so.9.0.0`. When another package *links* against MySQL it will ask for simply `libmysqlclient.so`. If the link is successful however, it will record that it linked against `libmysqlclient.so.9` specifically, and *at image activation time*, it will demand that it find a file (or symbolic link) called `libmysqlclient.so.9`. It's up to CMX to make sure that that name is available. CMX is aware of this responsibility and correctly constructs a symbolic link for `libmysqlclient.so.9` instead of `libmysqlclient.so`, which is what it would normally do for a standard CMX package.

8 This is the final trick to make the MySQL client shareable accessible through this package. Just like the header files, the client shareable is not in a conventional location, but by using this magic macro (yes it is a magic macro interrogated by CMX), it's possible to introduce a final indirection. The magic macro name must be `<pkg>_shr_<name>` where `<name>` is the short form name of the shareable (i.e. without the `lib` prefix and the `.so` suffix … exactly the name defined in the `<pkg>_shr` magic macro in fact).

The upshot is that by constructing a MySQL cover package containing a requirements file like this, other CMX packages wishing to use MySQL can put a `use MySQL` statement in their requirements file and not know the difference. The cover will make both the MySQL header files and the MySQL shareables available.

So far this requirements file only supports the SLAC installation of MySQL. For another site to use the same requirements file, it would be necessary to extend the `MySQL_r`, `MySQL_h` and `MySQL_i` macro definitions for the local `CMTSITE` pseudo-tags (e.g. `sun-gcc-nrl`). This can be

a bit tricky given that the installation locations on different sites may vary widely, but it should always be possible to come up with a recipe that ends up pointing to the right header include directory and shareable object on all sites.

5.1.4 Requirements File Support For Segregating Test Code

One thing this section is not about is how to embed test code into a package. That is far too large a topic and will be covered in “Flight Software Development Guidelines”. This section is here to annotate a little feature I put in to satisfy a JJ request to have a way to segregate a package’s real source code from its test code. The method is illustrated in the following entirely notional requirements file shard:

```
   # ------------------------------------------------------------------------
   # Shareable: ttc_test
   # ------------------------------------------------------------------------
   macro          ttc_test__buildtags  "linux-gcc sun-gcc rad750 mv2304"
   macro          ttc_test__linkshr    "" \
       linux-gcc "$(TTC_ttc__shr)" \
       sun-gcc   "$(TTC_ttc__shr)" \
       rad750    "$(TTC_ttc__shr)" \
       mv2304    "$(TTC_ttc__shr)"
0  document shr   ttc_test ptd=../ptd/ \
       TTC_dispatch_test.c \
       TTC_setup_test.c \
       TTC_transaction_test.c
```

Figure 8 Segregating Test Code

Everything you need to know is on the line labeled 0. The token `ptd=../ptd/` defines a location relative to the package’s `/src` directory where the test code can be found. Thus in this example, CMX expects to find the files `TTC_dispatch_test.c`, `TTC_setup_test.c`, etc. in a directory called `/ptd` which lies in parallel with the package’s `/src` directory. The `ptd=` piece of the token is mandatory syntax, the value of `ptd` is up to you (though I’d recommend following the convention just described). `ptd` just stands for “package test directory” (OK so I was lacking in imagination when I came up with that one). The mechanism supports a maximum of one directory per constituent, though the same directory can be reused for many constituents.

There are some subtle effects when building a constituent this way. When `gmake` does the build, it first `cd`s to the `/src` directory. That is still true when it builds one of these test constituents. Keep that in mind when coding your `#include` statements.

5.2 CMX Requirements Files Reference

5.2.0 CMX Defined Constituent Types

CMX defines three constituent types and maintains nine (three each) CMT makefile fragments to support them. The following table summarizes the constituent types, their products and their product naming conventions, when used to build a constituent called `<con>`.

Type	Fragments	Host System		Embedded System	
		Name	Type	Name	Type
bsp	bsp	(no meaning)		<con>.o	Absolute image
	bsp_header			lib<con>.ro	Relinkable object
	bsp_trailer			lib<con>.so	Shareable
exe	exe	<con>	Executable	lib<con>.o	Relocateable object
	exe_header			lib<con>.ro	Relinkable object
	exe_trailer			lib<con>.so	Shareable
shr	shr	lib<con>.so	Shareable	lib<con>.o	Relocateable object
	shr_header			lib<con>.ro	Relinkable object
	shr_trailer			lib<con>.so	Shareable

Table 3 CMX Defined Constituent Types And Naming Conventions For Their Products

Actually, I lied. CMX supports a fourth constituent type called msg. This constituent type is dedicated to the support of the MSG package and the message system. Details of how to use the msg constituent type are detailed in the manual for the MSG package. For the purposes of this document I would just point out that the msg type is very simple and does not impact anything discussed here.

Actually, I lied twice. CMX supports a fifth constituent type called cat. This constituent type is dedicated to the support of the LCAT package and the message system. Details of how to use the cat constituent type are detailed in the manual for the LCAT package. For the purposes of this document I would just point out that the cat type is very simple and does not impact anything discussed here.

5.2.1 CMX Defined Magic Macros

Magic macros for exporting CMX products to the Unix environment. These are Unix host specific macros. It makes no sense to define them for embedded system tags.

Name	Comment
<pkg>_cmt	Blank delimited list of scripts to export. Scripts must be in the /cmt directory.
<pkg>_exe	Blank delimited list of executables to export.
<pkg>_shr	Blank delimited list of shareables to export.
<pkg>_shr_<con>	Indirection to the real location of a shareable (used in cover requirements files only).

Table 4 Magic Macros For Exporting CMX Products To The Unix Environment

Magic macros for adding compilation options:

Name	Comment
<con>__cmp_flags	Additional compilation flags to apply to all files in a constituent.

<file>___cmp_flags	Additional compilation flags to apply to a file in all constituents in which it appears.
<con>__<file>___cmp_flags	Additional compilation flags to apply to a specific file in a specific constituent.

Table 5 Magic Macros For Adding Compilation Flags

Magic macro for directing the build target(s):

Name	Comment
<con>__buildtags	Blank delimited list of CMT tags identifying the tags for which a constituent can be built.

Table 6 Magic Macros For Directing The Build Target(s)

Magic macros for directing the link:

Name	Comment
<con>__linkshr	List of shareables *only* to be used in the link (links 1-4)
<con>__linkopt	Additional link flags to be used in the link (links 1-4)
<con>__linkxro	Define a list of relinkable objects to include in the link (links 1-5). Has no meaning (and is ignored) for constituent types shr and exe.
<con>__linkbsp	Link flags to be used during link step 5 of a kernel/BSP build. Has no meaning (and is ignored) for constituent types shr and exe.
<con>__linkrom	Link flags to be used during link step 6 of a kernel/BSP build. Has no meaning (and is ignored) for constituent types shr and exe.
<con>__linkvar	Used to introduce extra options. The contents of this macro follow the CMX qualifier syntax. --nocab suppresses the generation of CMX as built information in the target image. Allowed for all document types (exe, shr and bsp). --symbols causes a symbol table to be generated and linked into an absolute image. If this flag appears *without* the --romstub and --romsize qualifiers, the produced image is not compressed and wrappered, but is suitable for downloading and (presuming the VxWorks configuration was set up correctly) driving from the embedded shell. Only allowed for document type bsp. --romstub=<file>,<file> --romsize=<size>. These qualifiers always appear together and produce a compressed, ROMable absolute image (thus forcing a sixth link step). Files specified by --romstub are omitted from links 1-5. The size specified by --romsize is the size of the available EEPROM on the target and is used in a final sanity check. Only allowed for document type bsp. Must always be accompanied by the --symbols qualifier.

Table 7 Magic Macros For Directing The Link

Magic macros for editing the list of files submitted to doxygen:

Name	Comment
<con>__insertdox	Add a file to the list of files submitted to doxygen.
<con>__removedox	Remove a file from the list of files submitted to doxygen.

Table 8 Magic Macros For Editing The List Of Files Submitted To Doxygen

Magic macro to embed a user defined text string into the CMX-as-built information:

Name	Comment
<con>__usertxt	Requested by Curt.

Table 9 Magic macro to embed a user defined text string into CMX-as-built information

Not magic macros but certainly conventional:

Name	Comment
<pkg>_bin	Helps make the requirements file a little more readable.
<pkg>_<con>__shr	Export link time access to a package's shareables.

Table 10 Not Magic Macros But Certainly Conventional

5.3 Restrictions On CMT Macro Expansions

To maintain some performance, CMX sometimes parses the requirements file directly. It does this to avoid calling the `cmt` command to do macro expansions (to do its macro expansion, the `cmt` command always parses all requirements files in the use tree down from the current package which can be very expensive). This direct parsing results in some types of requirements file statements that are restricted to literal strings. The same restrictions may in fact be true for CMT itself, but this is the list of restricted syntaxes deriving from CMX requirements file parsing:

```
# ------------------------------------------------------------------------
# The document type cannot be a macro
# ------------------------------------------------------------------------
macro      document_type  "shr"
document $(document_type) <document_name> \
.
.
```

Figure 9 Document type cannot be a macro

```
# ------------------------------------------------------------------------
# The document name cannot be a macro
# ------------------------------------------------------------------------
macro          document_name "some_constituent"
document shr $(document_name) \
.
.
```

Figure 10 Document name cannot be a macro

```
# ------------------------------------------------------------------------
# The target of a "use" directive cannot be a macro
# ------------------------------------------------------------------------
macro   package_name "some_package"
use   $(package_name)
.
.
```

Figure 11 The target of a “use” statement cannot be a macro

```
# ------------------------------------------------------------------------
# The list of build tags cannot be a macro
# ------------------------------------------------------------------------
macro                              bld_list "sun-gcc linux-gcc"
macro <constituent>__buildtags "$(bld_list)"
.
.
```

Figure 12 The list of build tags cannot be a macro

5.4 Extra Tokens To The Build Command

The description of the `cmx build` command noted that the command accepts extra tokens that are not consumed by the command itself but are simply passed on to the `gmake` step. True, but not very informative. Extra tokens to the `gmake` step fall into three categories:

5.4.0 Options/Switches Directed At **`gmake`** Itself

`gmake` itself can take a number of options/switches like `-t/--touch` which simply touches files to alter their timestamps and artificially bring the package up to date. If a user is operating at this level, good luck. These are dangerous operations and I see no reason to provide the information in this manual by which users can hang themselves.

5.4.1 Specifying The Target Of A Build

The make files constructed by CMT are by and large very conventional, so they do provide standard targets. If no target is specified then `gmake` will build target `all`, which translates to all constituents. If `gmake` is asked to build target `clean`, `gmake` will delete all the products of all constituents. A less well-known target is the individual constituent. If a package `FOO` defines a constituent `bar`, then the following command will build constituent `bar` only:

```
cmx build FOO bar
```

Even less well known is the ability to clean individual constituents. Simply concatenate `clean` onto the end of the constituent name:

```
cmx build FOO barclean
```

Finally, there is a CMT special target called `configclean`. This target will delete all the makefiles and makefile fragments constructed by CMT. This target is sometimes useful when a build aborts in some untidy state and can’t be restarted.

5.4.2 Controlling the Noise Level of a Build

The final category of extra tokens is macro definitions. Obviously the make files and fragments have to be set up to understand and interpret the macros, so this list is limited to an enumerable set of known macros. All of them are provided to control the noise level of a build, so let’s start by looking at the output from a build:

```
tersk03:apw> cmx build MSG


[CM.0] Start package build with tag....... sun-gcc
[MH.0]     Build make file................ constituents.make
[CH.0]     Build make file................ sun-gcc.make
[CON0]     Build make file................ msg_st.make
[CON1]     Start build of constituent..... msg_st
[ST04]         Build shareable............ libmsg_st.so
[LINK]             First (and only) link.. libmsg_st.so
[ST05]         Build documentation ....... msg_st
[CON2]     End build of constituent....... msg_st
[CON0]     Build make file................ msg_mt.make
[CON1]     Start build of constituent..... msg_mt
[ST02]         Update dependencies........ msg_mt.cidc
[S000]         Update dependencies........ MSG_signal_mt.c
[S000]         Update dependencies........ MSG_signal.c
[S000]         Update dependencies........ MSG_output.c
[S000]         Update dependencies........ MSG_msgs.c
[S000]         Update dependencies........ MSG_database.c
[S000]         Update dependencies........ MSG_control.c
[S001]         Compile.................... MSG_control.c
[S001]         Compile.................... MSG_database.c
[S001]         Compile.................... MSG_msgs.c
[S001]         Compile.................... MSG_output.c
[S001]         Compile.................... MSG_signal.c
[S001]         Compile.................... MSG_signal_mt.c
[ST03]         Preprocess ................ msg_mt.cidc
[ST04]         Build shareable............ libmsg_mt.so
[LINK]    [CON2]     End build of constituent....... msg_mt
[CON0]     Build make file................ msg_print.make
[CON1]     Start build of constituent..... msg_print
[ST02]         Update dependencies........ msg_print.cidc
[S000]         Update dependencies........ MSG_printProc.c
[S001]         Compile.................... MSG_printProc.c
[ST03]         Preprocess ................ msg_print.cidc
[ST04]         Build shareable............ libmsg_print.so
[LINK]             First (and only) link.. libmsg_print.so
[ST05]         Build documentation ....... msg_print
[CON2]     End build of constituent....... msg_print
[CON0]     Build make file................ msg2src.make
[CON1]     Start build of constituent..... msg2src
[ET02]         Update dependencies........ msg2src.cidc
[E000]         Update dependencies........ MSG_readMsg.c
[E000]         Update dependencies........ MSG_createFacility.c
[E000]         Update dependencies........ M2S_msgs.c
[E000]         Update dependencies........ msg2src.c
[E001]         Compile.................... msg2src.c
[E001]         Compile.................... M2S_msgs.c
[E001]         Compile.................... MSG_createFacility.c
[E001]         Compile.................... MSG_readMsg.c
[ET03]         Preprocess ................ msg2src.cidc
[ET04]         Build executable........... msg2src
[LINK]             First (and only) link.. msg2src
[ET05]         Build documentation ....... msg2src
                   First (and only) link.. libmsg_mt.so
```

```
    [ST05]          Build documentation ....... msg_mt
    [CON2]      End build of constituent....... msg2src
    [CON0]      Build make file................ test_msg_st.make
    [CON1]      Start build of constituent..... test_msg_st
    [ET02]          Update dependencies........ test_msg_st.cidc
    [E000]          Update dependencies........ MTS_msgs.c
    [E000]          Update dependencies........ MSG_testUtility.c
    [E000]          Update dependencies........ MSG_testThread.c
    [E000]          Update dependencies........ MSG_testState.c
    [E000]          Update dependencies........ MSG_testOptions.c
    [E000]          Update dependencies........ MSG_testFormat.c
    [E000]          Update dependencies........ MSG_testBuffer.c
    [E000]          Update dependencies........ MSG_test.c
    [E001]          Compile.................... MSG_test.c
    [E001]          Compile.................... MSG_testBuffer.c
    [E001]          Compile.................... MSG_testFormat.c
    [E001]          Compile.................... MSG_testOptions.c
    [E001]          Compile.................... MSG_testState.c
    [E001]          Compile.................... MSG_testThread.c
    [E001]          Compile.................... MSG_testUtility.c
    [E001]          Compile.................... MTS_msgs.c
    [ET03]          Preprocess ................ test_msg_st.cidc
    [ET04]          Build executable........... test_msg_st
    [LINK]              First (and only) link.. test_msg_st
    [ET05]          Build documentation ....... test_msg_st
    [CON2]      End build of constituent....... test_msg_st
    [CON0]      Build make file................ test_msg_mt.make
    [CON1]      Start build of constituent..... test_msg_mt
    [ET02]          Update dependencies........ test_msg_mt.cidc
    [E000]          Update dependencies........ MTS_msgs.c
    [E000]          Update dependencies........ MSG_testUtility.c
    [E000]          Update dependencies........ MSG_testThread.c
    [E000]          Update dependencies........ MSG_testState.c
    [E000]          Update dependencies........ MSG_testShotgun.c
    [E000]          Update dependencies........ MSG_testOptions.c
    [E000]          Update dependencies........ MSG_testFormat.c
    [E000]          Update dependencies........ MSG_testBuffer.c
    [E000]          Update dependencies........ MSG_test.c
    [E001]          Compile.................... MSG_test.c
    [E001]          Compile.................... MSG_testBuffer.c
    [E001]          Compile.................... MSG_testFormat.c
    [E001]          Compile.................... MSG_testOptions.c
    [E001]          Compile.................... MSG_testShotgun.c
    [E001]          Compile.................... MSG_testState.c
    [E001]          Compile.................... MSG_testThread.c
    [E001]          Compile.................... MSG_testUtility.c
    [E001]          Compile.................... MTS_msgs.c
    [ET03]          Preprocess ................ test_msg_mt.cidc
    [ET04]          Build executable........... test_msg_mt
    [LINK]              First (and only) link.. test_msg_mt
    [ET05]          Build documentation ....... test_msg_mt
    [CON2]      End build of constituent....... test_msg_mt
    [CM.1] End package build with tag......... sun-gcc
```

Figure 13 Output From A Successful Build Command

As you can see, this is considerably tidier than the average gmake output! A lot of the cosmetic changes referred to in the list of CMT files I have edited were in support of tidying up the build output. I hope that most of this figure is self-explanatory. The most obscure part is probably those chicken scratchings like [xxxx] at the start of each line. Each variety is a pointer back to a CMT or CMX file. If the name contains a "." it points to a genuine makefile. If it does not, it points to a makefile fragment. Trailing digits are used to distinguish different messages in the same file. In the example above:

```
[CM.0]      CMX file .../src/cmx.make
[MH.0]      CMT file .../src/Makefile.header
[CH.0]      CMT file .../src/constituents_header
[CH.1]      CMT file .../src/constituents_header
[CH.2]      CMT file .../src/constituents_header
[CON0]      CMT file .../fragments/constituent
[S000]      CMX file .../fragments/shr
[S001]      CMX file .../fragments/shr
[SH00]      CMX file .../fragments/shr_header
[ST00]      CMX file .../fragments/shr_trailer
[E000]      CMX file .../fragments/exe
[EH00]      CMX file .../fragments/exe_header
[ET00]      CMX file .../fragments/shr_trailer
```

This is all very handy for someone like me who's trying to understand how CMT does its thing, but a subset of these is very handy for the user too. To demonstrate this, let's introduce a deliberate error into the previous build:

```
tersk03:apw> cmx build MSG

[CM.0] Start package build with tag....... sun-gcc
[MH.0]     Build make file................ constituents.make
[CH.0]     Build make file................ sun-gcc.make
[CON0]     Build make file................ msg_host.make
[CON1]     Start build of constituent..... msg_host
[S000]         Update dependencies........ MSG_connect.c
[S001]         Compile.................... MSG_connect.c
..//src//MSG_connect.c: In function `MSG_connect':
..//src//MSG_connect.c:40: parse error before `;'
gmake[2]: *** [../../../../binary/MSG/test/sun-gcc/MSG_connect.o] Error 1
gmake[1]: *** [msg_host] Error 2
gmake: *** [all] Error 2

tersk03:apw>
```

Figure 14 Output From An Unsuccessful Build Without Dump

Well the build blew up. It even printed the compile error. Most of the time however, that simply isn't enough information. At the very least it would be nice to see the actual compile command, and for CMT experts it would be useful to see the make macro substitutions that contributed to the compile command. To do this, define a macro in the build command. The macro name should be dmp_<fragment_prefix> where <fragment_prefix> is the letters that start the [xxxx] notation. Thus to dump the [S001] step in the above example, use macro name dmp_s. The value of the macro determines what is dumped. It is interpreted as a two bit field where bit 0 requests a dump of the actual command executed and bit 1 requests a dump of all the CMT macro substitutions. Dumping with bit 1 set can produce a lot of information, so the following example restricts itself to setting just bit 0:

```
tersk03:apw> cmx build MSG dmp_s=1

[CM.0] Start package build with tag....... sun-gcc
[MH.0]     Build make file............... constituents.make
[CH.0]     Build make file............... sun-gcc.make
[CON0]     Build make file............... msg_host.make
[CON1]     Start build of constituent..... msg_host
[S001]         Compile.................... MSG_connect.c
gcc -c -Wall -Wno-format -fPIC -g  \
                  \
                 -I..//src// \
-I"/afs/slac.stanford.edu/u/ey/apw/NEW/source//MSG//test" \
-I"/afs/slac.stanford.edu/g/glast/applications/mysql/3.23.16-alpha/sun4x_5\
6/mysql-3.23.16-alpha/include"   \
                 -o ../../../../binary/MSG/test/sun-gcc/MSG_connect.o \
..//src//MSG_connect.c
..//src//MSG_connect.c: In function `MSG_connect':
..//src//MSG_connect.c:40: parse error before `;'
gmake[2]: *** [../../../../binary/MSG/test/sun-gcc/MSG_connect.o] Error 1
gmake[1]: *** [msg_host] Error 2
gmake: *** [all] Error 2

tersk02:apw>
```

Figure 15 Output From An Unsuccessful Build With Dump

The dumping facility is only available for the CMX constituent types (`bsp`, `exe` and `shr`).

6 The Link Step Of CMX Build

The complexity of the CMX link step has already been alluded to in section 5.1.2 “A VxWorks Kernel/BSP CMX Requirements File”. This section will describe in more detail what happens during a link step and provide a rationale for the complexity.

6.0 Goals For The Link Step

The most obvious goal is of course to link the code, but the CMX link attempts to do even more:

- Perform unresolved external reference analysis and abort should any unresolved external references be found. Perform this analysis symmetrically across all target groups (Sun Unix, Linux and embedded systems).
- Use the linker’s facilities for versioning shareables (“map” files in Sun-speak or “version-script” files in Linux-speak) to (a) closely define the interface presented by a shareable (and enforce it through “strength reduction”) and (b) enforce shareable version compatibility. Make this facility available symmetrically across all target groups.
- Stamp all code modules with information about the build (who, where, when, etc.). Do this symmetrically across all target groups.

A not unreasonable list. The complexity creeps in by virtue of non uniformity of both the linker executable across different target groups (`gcc` runs the GNU linker on Linux but the Sun vendor linker on Sun boxes) and the variety of products being built (incrementally linked relocateable objects for embedded systems being roughly equivalent to shareables on host systems). This turns the problem into a three by three matrix, which is reflected in the following sections.

I’ve said that `gcc` invokes the Sun vendor linker on Sun hosts. That’s true for all Sun boxes at SLAC, but it’s not mandatory. CMX currently assumes this to be true, but if a site has been set up to use the GNU linker as part of `gcc` processing, it should be reasonably easy to convert the CMX link step to accommodate this configuration.

6.1 Unresolved External Reference Analysis

Unresolved external reference analysis mandates that all libraries necessary to resolve subroutine calls (among other things) must be available at link time. If any unresolved external references remain at the end of the link, the unresolved references are listed and the link aborts.

This gives early notice to a developer that something bad is going on and is particularly useful when building embedded system objects. Without unresolved external reference analysis, unresolved references in embedded system objects do not surface until modules are actually loaded into a target system, which can be inconveniently late.

There remains a philosophical question however. Unresolved external reference analysis can be either “shallow” or “deep”. Consider the case of shareable `shrA` which calls entry points in `shrB` but not in `shrC`. Shareable `shrB` does call entry points in `shrC`. When linking `shrA`, should the unresolved external analysis extend to the calls `shrB` makes into `shrC`, or should the link be satisfied if all the unresolved external references in `shrA` are resolved? This is a fundamental difference between the Sun vendor linker and the GNU linker. The Sun linker does shallow analysis … it’s satisfied provided all unresolved external references from `shrA` are satisfied. The GNU linker does deep analysis and chases down the complete shareable tree looking for unresolved references in all modules. Curiously enough, and contrary to its unresolved external reference analysis, the GNU linker only captures the names of shareables satisfying direct calls from `shrA`.

I’m sure opinions will vary on the matter, but I personally prefer shallow analysis. Apart from anything else it obviates the need to build trees of modules to satisfy a link step (which CMT can do, but it’s the start of the sticky slope to reduced modularity). CMX performs shallow analysis.

6.1.0 UER Analysis For Sun Host Modules

This will be the shortest subsection. The Sun linker “does the right thing” right out of the box. CMX links for Sun shareables or executables are accomplished with just one invocation of the linker.

6.1.1 UER Analysis For Linux Host Modules

Linking Linux hosts shareables or executables is done with three invocations of the linker. The first link performs the link *without* the list of shareables provided by the user through the `<con>__linkshr` magic macro in the requirements file. This pass constructs a list of unresolved external references directly attributable to the module being built. The second link is made *with* the shareable list. If the unresolved external references generated by the first link do not appear in the unresolved external reference list generated by the second link, then the shareables added in the second link have satisfied all the unresolved references for the module being built. Unfortunately both these link steps will (almost always) fail to produce an output file because there are (almost always) unresolved external references. The third link pass simply repeats the second link pass with unresolved external reference analysis turned off so that the linker doesn’t care and will produce the output file.

6.1.2 UER Analysis For Embedded System Targets

As usual, embedded systems provide the biggest challenge. The standard product for embedded systems (excluding kernel/BSP products) is the incrementally linked object which can then be fed directly or via Tornado into an embedded system. Incrementally linked objects do not even admit to doing unresolved external reference analysis.

The solution I have adopted is to construct pseudo-products. The first pseudo-product is a genuine shareable linked from the cross-compiled object files. This is produced using the same first three link steps described in the previous section (with the proviso that when asked to link an executable, the link actually creates a shareable). The resulting shareable cannot be executed

anywhere (well maybe on a PPC Macintosh but that's not all that useful). These shareables exist only so that they can be used as analysis instruments.

The actual executable used for the first three links is not the standard VxWorks `ldppc`. The VxWorks provided linker is considerably out of date and does not understand things like version script files. This is not a problem in the specific case of unresolved external reference analysis, but is a show stopper for interface versioning. The first three link steps must be performed with a more up to date linker, so all sites will need to provide an executable `ldppc-2.11.2` (sorry about embedding the version in the name) which should be a cross-linker to the PPC architecture constructed from `binutils 2.11.2` or better. To do the strength reduction (described in the next section), each site must also provide `objcopyppc-2.11.2`.

These pseudo-shareables manifest themselves in a package's requirements file. When describing an embedded system constituent, the developer may have to specify a list of these pseudo-shareables to satisfy the unresolved external reference analysis for the constituent being built. This is entirely analogous to the real shareable specification for host executables/shareables. It just looks odd because there are no shareables in the embedded world.

This three pass link process is sufficient to do unresolved external reference analysis for embedded objects, but so far the link hasn't produced anything that could actually be used in embedded systems. The link process goes on to make two more link passes (this time with the regular VxWorks `ldppc` executable) to produce a second pseudo-product called a relinkable object (with file extension `.ro`) and finally the familiar relocatable object file (with file extension `.o`). These are discussed in more detail in later sections.

6.2 Interface Versioning

In the mix and match world of shareables, the shareables required to satisfy the image are found at image activation time and are not guaranteed to be the same shareables that were used to satisfy the link step. This is of course that great advantage of shareables but consider the downside. What if the image activator picks up an out of date version of a shareable? The old callable interface may well be deficient compared to its up to date version and the image activator can be left with hanging unresolved external references. Most image activators ignore this problem and just keep going which can result in spectacular aborts when processing tries to jump to a non-existent routine.

These types of problems have not escaped the notice of the Sun and Linux system engineers. Sun has put together a control file definition and link time processing which provides tight control over the content of a shareable interface and the Linux crowd has essentially adopted Sun's solution in the GNU linker. This is quite an extensive topic and developers are encouraged to read Sun's "Linker and Libraries Guide" chapter 5 for a grounding in the theory and use of interface versioning.

At a more practical level, CMX has adopted the Sun interface versioning technique with a few CMX specific twists. First of all, Sun refers to the interface control file passed to the linker as a "map" file. Linux refers to the same file as a "version-script" file. Following my usual "a plague on both your houses" philosophy, I have rechristened the CMX instantiation of these files as CMX Interface Definition ("`cid`") files. The renaming is somewhat justified by the fact that while `cid` files are a proper subset of the full Sun syntax, they *are* a subset and do not implement the complete syntax. Furthermore they take advantage of the fact that interface names are

completely arbitrary in the strict Sun/Linux world and `cid` files overload the interface names with a specific syntax to make interface evolution analysis possible.

6.2.0 Processing **`cid`** Files

Mechanically, `cid` files look somewhat like and are treated somewhat like C source files. They come in four varieties with the following extensions:

Extension	Use
`.cidc`	`cid` “source” file. Preprocessed to produce both the `.cidd` and `.cido` files.
`.cidh`	`cid` “header” file. Can be included into `cid` “source” files.
`.cidd`	`cid` “dependency” file. Tracks the header files included by a `cid` “source” file and makes the results available to the build system. Analogous to dependency tracking for normal C source files. Created in a package’s binary tree and rarely seen by the developer.
`.cido`	`cid` “object” file. The result of running the C preprocessor on `cid` “source” files. Created in a package’s binary tree and rarely seen by the developer. This is really an ASCII file in the correct format for submission to the linker.

Table 11 Varieties Of Files Handled By The `cid` (CMX interface definition) System.

The CMX link step now demands that every constituent comes with a `.cidc`. file named after the constituent and located in the same directory as the constituent source code. Thus a constituent called `foo` built from code in the `/src` directory must supply the file `/src/foo.cidc`.

If the developer fails to provide a constituent `.cidc` file, CMX will provide a default which the developer is guaranteed to hate. It defines all symbols as local and the shareable will be completely uncallable.

6.2.1 Example **`.cidc`** Files

6.2.1.0 A Well Formed **`.cidc`** File

The following figure demonstrates the anatomy of a `.cidc` file. It’s part of the test suite I used to develop the interface versioning strategy:

```
   // ------------------------------------------------------------------------
   // CVS $Id$
   // ------------------------------------------------------------------------
   //
   // Interface definition file
   //
   //     Package: PKGB
   // Constituent: pkgb
   //
   // ------------------------------------------------------------------------

   // ------------------------------------------------------------------------
   // Original version
   // ------------------------------------------------------------------------
0  PKGB_pkgb_0.0
   {
1      global:
2          PKGB_external;

3       local:
4           *;
   };

   // ------------------------------------------------------------------------
   // Add new routine
   // ------------------------------------------------------------------------
   PKGB_pkgb_0.1
   {
       global:
5          PKGB_addition;

6  } PKGB_pkgb_0.0;

   // ------------------------------------------------------------------------
   // Performance improvement
   // ------------------------------------------------------------------------
7  PKGB_pkgb_0.1.0 {} PKGB_pkgb_0.1;
```

Figure 16 Example `.cidc` File

0 The interface string, which must follow a strict convention. Everything to the left of the right-most underscore (which must be present) is the interface *name*. The interface name must follow C variable naming rules but is otherwise arbitrary. In this case (and this is the default for CMX generated `.cidc` files), the interface name is simply the package and constituent names concatenated with an underscore. This is guaranteed to keep the namespace clean. Everything to the right of the right-most underscore is the interface instance. The instance must be either two or three groups of digits separated by full stops. The digits represent the major, minor and (optionally) the patch definition of the interface.

A single constituent can define one or more interfaces (i.e. unique interface *names*).

1 `global` is a keyword which introduces a list of entry points which make up the callable interface for this constituent.

2 This is a very limited interface. The only callable routine is `PKGB_external`.

3 `local` is a keyword which can only appear in a “root” interface definition. A root interface definition is never derived from another interface definition in the file. See point 6 for an example of a derived interface.

4 The `local` keyword *must* be followed by the wildcard `*`. This forces all symbols not defined to be global (by being mentioned in the global section) to be local (and therefore uncallable from outside the shareable).

5 This extends the `PKGB_pkgb` interface to include a new entry point called `PKGB_addition`.

6 This syntax defines `PKGB_pkgb_0.1` to be a new interface instance based on interface instance `PKGB_pkgb_0.0`. Note that this interface instance does not contain a local keyword (because it’s a derived interface instance) and that the minor interface instance number has been bumped (this change is entirely backward compatible but has changed the interface).

7 This is an example of a patch increment. The new interface instance `PKGB_pkgb_0.1.0`, derived from instance `PKGB_PKGB_0.1`, does not define any new entry points so the interface is entirely unaltered. This type of interface evolution indicates that the changes to the constituent are entirely internal but worth noting.

6.2.1.1 A `.cidc` File With Errors

The example just given was a well formed `.cidc` file containing no syntax errors. The next figure gives example of `.cidc` file errors:

```
    // ------------------------------------------------------------------------
    // CVS $Id$
    // ------------------------------------------------------------------------
    //
    // Interface definition file
    //
    //     Package: PKGB
    // Constituent: pkgb
    //
    // ------------------------------------------------------------------------

    // ------------------------------------------------------------------------
    // Original version
    // ------------------------------------------------------------------------
    PKGB_pkgb_1.2
    {
        global:
            PKGB_external;

         local:
             *;
    };

    // ------------------------------------------------------------------------
    // Add new routine
    // ------------------------------------------------------------------------
    PKGB_pkgb_1.1
    {
        global:
            PKGB_addition;

0   } PKGB_pkgb_1.2;

    // ------------------------------------------------------------------------
    // A major release
    // ------------------------------------------------------------------------
1   PKGB_pkgb_2.0
    {
        global:
            PKGB_addition;

         local:
             *;
    };
```

Figure 17 Example `.cidc` File Containing Errors

0 `PKGB_pkgb_1.1` derived from `PKGB_pkgb_1.2`? The minor release number decreased! Believe it or not this is perfectly legal in terms of Sun and Linux syntax, but is illegal by CMX rules. Derived interfaces must always increment either the patch or the minor interface instance number. The major interface instance number is dealt with in the next bullet…

1 An example of bumping the major interface instance number. The motivation for this change is certainly correct. With the definitions in this file, the symbol `PKGB_external` has been *removed* from the `PKGB_pkgb` interface, making the `PKGB_pkgb_2.0` interface instance backward incompatible with the `PKGB_pkgb_1.1` (or

PKGB_pkgb_1.2). The problem is that there is no way to present incompatible interfaces in a single .cidc file. The rule is that if the development of a constituent interface demands a major release for whatever reason, the previous content of the constituent interface must be stripped back and a brand new major release started.

6.2.1.2 A .cidc File Using The Preprocessor

The preprocessing step from .cidc to .cido is literally the C preprocessor with all that that implies. The following is a made up example of how this might be used (I still haven't made up my mind if using it this way would be a good idea!) I'm sure Dan Wood will recognize a number of the included file names.

```
   // ------------------------------------------------------------------------
   // CVS $Id$
   // ------------------------------------------------------------------------
   //
   // Interface definition file
   //
   //     Package: RTOS
   // Constituent: rtos_tornado
   //
   // ------------------------------------------------------------------------

   // ------------------------------------------------------------------------
   // Common to all targets
   // ------------------------------------------------------------------------
0  #include "VXW_ansi_stdlib_5.4.cidh"
   #include "VXW_ansi_stdio_5.4.cidh"
   #include "VXW_ansi_string_5.4.cidh"
   #include "VXW_ansi_time_5.4.cidh"
   #include "VXW_ansi_math_5.4.cidh"
   #include "VXW_ansi_ctype_5.4.cidh"
   .
   .

   // ------------------------------------------------------------------------
   // Specific to MV2x targets
   // ------------------------------------------------------------------------
1  #ifdef MV2300
   #include "VXW_mk48t.cidh"
   .
   #endif

   // ------------------------------------------------------------------------
   // Specific to RAD750 target
   // ------------------------------------------------------------------------
   #ifdef RAD750
   #include "RTOS_memscrub.cidh"
   .
   #endif
```

Figure 18 Example .cidc File Using The C Preprocessor

0 Looks a lot like C code to me…

1 …even down to conditional handling!

6.2.2 Interface Versioning For Sun Host Modules

Once the interface definition file is defined and included (automatically by CMX) in the link step, Sun linking proceeds quite naturally. Shareables are built knowing what interface(s)/version(s) they present to the outside world and what interface(s)/version(s) they require of other shareables to activate correctly at run time. Any symbols not identified as global in the interface file are demoted to local symbols and essentially disappear from view. The latter is the process usually called “strength reduction”.

6.2.3 Interface Versioning For Linux Host Modules

The Linux implementation of interface definition files is a (near) clone of the Sun implementation, so linking shareables on Linux falls out quite as naturally as the Sun.

6.2.4 Interface Versioning For Embedded System Targets

Once again, it’s the embedded system modules that present the challenge. Fortunately the same pseudo-shareable files created for unresolved external reference analysis also provide the solution for interface versioning. The pseudo-shareables are created using the same interface definition files as used for host systems. The Sun `pvs` command is used to ask a pseudo-shareable what interface instances it provides and what interface instances it requires and on the basis of these lists, the link script creates a small assembly file where provided interfaces are defined as global symbols and required interfaces are defined as unresolved symbols. This small file is compiled and included in the fourth and later link steps. The upshot of this is that when a relocateable file is loaded into an embedded system, it will try to resolve all it’s unresolved external references (i.e. the interface versions it requires) and will complain if it can’t find them. If loaded successfully, the same module provides symbol resolutions for the interfaces it defines so that later module loads can resolve the interfaces *they* require.

The final step to symmetrize host and embedded system handling is provided by a combination of the object file analysis tool `nm` and the object file copy tool `objcopy` (strictly, the executables are `nmppc` and `objcopyppc-2.11.2`). The list of global symbols in the pseudo-shareable can be determined by using `nmppc` with the appropriate options. Likewise the list of globals in the relinkable file (the output from the fourth link step) can be created. Compare the lists and it’s possible to construct an `objcopyppc` command requesting that extra global symbols in the relinkable file be reduced to local strength. Because the relinkable file is used as input to the fifth step which creates the relocateable file, the strength reduction persists to the relocatable file.

6.3 CMX-As-Built Information

The final goal for the CMX link step was to embed information in all modules recording who, when and where a module was built along with CVS tag information whenever that was available. This information should be available at run time, thus tracing the executing environment directly back to source code.

More ambitiously, the original goal was to provide this functionality “user blind”, requiring no input from the developer either at build time or at run time. This turned out to be a little more than I could persuade the linker to do. The original implementation of CMX-as-built (which was provided “user blind” but was very fragile) has now been superceded with a more robust version. The price of the more robust version is that users will need to be aware of the CMX-as-built function in all requirements files.

Despite the re-implementation, the basic principles are unaltered:

- The link script constructs a small C file where the CMX-as-built information is defined in what amount to data statements.
- The C file also includes two routines. One for inserting the CMX-as-built information onto a singly linked list, and one to remove the information from the singly linked list. These routines are given the special attributes of (respectively) a *constructor* and a *destructor*.
    - On host systems, the linker arranges for such constructor routines to be called before entering the `main` routine. Similarly, it arranges for destructor routines to be called after the `main` routine exits (or after a call to `exit`).
    - On embedded systems, the same arrangements can be made, but the process is far more manual. It involves running a special “munch” step that inspects an interim link output for the names of constructor and destructor routines. The names are listed in another small C file, the file is compiled and the output used in a subsequent link step. The VxWorks loader/unloader commands know to run these routines automatically.
- The C routine is compiled and linked into the target module.

Given that description the only outstanding issue in the implementation is how to allocate a single location to provide the head of the singly linked list. This varies by host/embedded system.

6.3.0 CMX-As-Built Information For Host Modules

Allocation of the list root on host systems is entirely conventional. The `cmx_asBuilt` shareable allocates it. The only down side of this is that all CMX built host modules (shareable or executable) must now specify the `cmx_asBuilt` shareable in their link step.

6.3.1 CMX-As-Built Information For Embedded Systems

Allocation of the list root on embedded systems is a little less conventional. The `cmx_asBuilt` module still provides it, but the whole of the `cmx_asBuilt` module is sucked into the embedded system kernel/BSP using the “external relinkable object” mechanism (see section 6.4). This is necessary because the kernel/BSP itself wants to put information on the CMX-as-built list so the list root has to be available at the time the kernel/BSP is built. The up side of this method is that users need not specify the `cmx_asBuilt` module in the linkage of embedded system objects because the necessary entry points are built into the operating system.

6.3.2 CMX-As-Built User Interface

To keep the amount of code sucked into embedded system kernel/BSPs to a minimum, the `cmx_asBuilt` module is very bare bones, providing just the functionality needed to maintain/interrogate the CMX-as-built information. The CMX package provides a second constituent called `cmx_asBuiltSpy` which demonstrates how to use the `cmx_asBuilt` entry points to construct print dumps. These routines can be used as the basis for other higher level functionality at a later date. For more details, see the doxygen documentation for these modules.

One obvious example of “higher level functionality” not yet implemented would be to construct a telemetry packet immediately after secondary-boot-and-load completes which lists all modules loaded.

6.4 Special Considerations For Kernel/BSP Linking

Still unexplained is the reason for preserving the relinkable (`.ro`) output from the fourth step of an embedded system link. This is a little esoteric and has to do with building kernel/BSPs.

A goal of the CMX environment is to keep the code base extremely modular. This is sometimes contrary to the needs of kernel/BSP building. To give an example: Dan has to write a boot procedure that ends with the SIU communicating with the spacecraft. The communication packets have to be CCSDS. That would imply that the CCSDS code has to go into the kernel package and the CCSDS package would become defunct. Do this a few times and almost everything goes into the kernel and modularity is lost. What Dan would like to do is keep CCSDS as a separate CMX package, but have the option to link it into the kernel if that's the appropriate thing to do. At first sight that seems simple … simply expose the natural product (the `.o` file) in the CCSDS package and then include it in the link of the kernel/BSP package. (This technique is sometimes referred to as "static linking").

Sorry, that doesn't work. It turns out that despite what your mother told you about chewing your food, you can only munch once. A module prepared to run static constructors cannot be used as input to another incremental link (lots of error messages about doubly defined global symbols). For this reason, the unmunched output from the fourth link step is preserved and there is a special magic macro (`<con>__linkxro`) for kernel/BSP building to allow a kernel/BSP package to include `.ro` files from other packages.

I hope that this explanation goes some way to explaining the nomenclature "relinkable object" (the output of the fourth link step) and "relocateable object" (output of the fifth link step for non `bsp` builds).

In some instances, the build process will go on to a sixth link step, but this only occurs when building standalone images for burning into EEPROM. The sixth link step has no bearing on the subject matter of this section.

6.5 Tying Link Interface Instances To CVS Versions

The link step completes the organizational hierarchy of the code. Starting at the top:

- Project: CMX concept, not version controlled.
- Package: CMX concept, version controlled through CVS repository tags.
- Constituent: CMT concept, not individually version controlled.
- Interface: Linker concept, version controlled through `cid` files.

This demonstrates what could turn into a conflict. Both the CMX package level and the shareable interface level think they are doing version control. These need to be dovetailed together.

Part of that dovetailing was already in evidence in section 6.2 "Interface Versioning". The CMX mandated `cid` file interface string syntax enforces a straightforward incrementing interface instance number. Unfortunately, a single constituent/shareable can present multiple interfaces and a package can contain multiple constituents/shareables, so how should package level versioning reflect interface level versioning? The CMX solution is to demand that package versions (the ones managed by CVS tags) reflect the worst case increment defined by any interface instance contained in the package. If any interface of any constituent has undergone a major revision then the CVS version should likewise bump the major version number, and so on. This defines how package versions change in terms of interface instances. The following defines how interface instance numbers change:

- If the interface is *exactly* unchanged, increment the patch number.
- If the interface has been augmented *only*, increment the minor number.
- In all other cases, increment the major number.

Please note that the `cid` file alone cannot make a definitive statement about which element to increment. Here are just a few examples that would leave the `cid` file unchanged but would demand a major increment:

- Changing the number or order of the arguments to a routine.
- Changing a structure exposed in a header file.
- Changing definitions exposed in a header file.
- Changing routines exposed as inline functions in a header file.

The number of times the phrase “in a header file” appears in this list should emphasize the importance of keeping header files stable. This is reflected in the great lengths to which the doxygen phase of package building goes to make it possible to keep inline documentation *out* of header files. Changing comments in a header file does not of course require a version bump, but even trivially changed header files get a new timestamp which results in a very jumpy make system.

This all sounds very draconian and difficult to manage, but CMX has some commands to guide the user through the process. Perhaps the most powerful command is `cmx check version`. This command works by comparing the `cid` files in a user test package against the corresponding `cid` files in the most recently tagged CVS version. On a `cid` file by `cid` file basis, the command prints the results of its analysis and then makes a final recommendation about what element of the version number should be bumped. The following figure shows the output from the command when run on the CMX package itself:

```
   tersk03:apw> cmx check version CMX
   ================================================================================
   Analyzing constituents/interfaces for package CMX
   --------------------------------------------------------------------------------
 0       Highest CVS version: V1-3-0    (used in this comparison)
     Site production version: V1-3-0
   Global production version: V1-3-0
   --------------------------------------------------------------------------------
   Constituent        Tag       Interface                V1-3-0   test     Result
   ------------------ --------- ------------------------ -------- -------- ------
 1 cid                linux-gcc CMX_cid                  0.0      0.0      same
                      ---------
                      sun-gcc   CMX_cid                  0.0      0.0      same
   ---------------------------
 2 cmx_asBuilt        i845e     CMX_cmx_asBuilt          0.0      <undef>  major
                                                         <undef>  1.0      major
                      ---------
                      linux-gcc CMX_cmx_asBuilt          0.0      <undef>  major
                                                         <undef>  1.0      major
                      ---------
                      mcp750    CMX_cmx_asBuilt          0.0      <undef>  major
                                                         <undef>  1.0      major
                      ---------
                      mv2304    CMX_cmx_asBuilt          0.0      <undef>  major
                                                         <undef>  1.0      major
                      ---------
                      rad750    CMX_cmx_asBuilt          0.0      <undef>  major
                                                         <undef>  1.0      major
                      ---------
                      sun-gcc   CMX_cmx_asBuilt          0.0      <undef>  major
                                                         <undef>  1.0      major
   ---------------------------
 3 cmx_asBuiltSpy     i845e     <undefined in V1-3-0>                        minor
                      ---------
                      linux-gcc <undefined in V1-3-0>                        minor
                      ---------
                      mcp750    <undefined in V1-3-0>                        minor
                      ---------
                      mv2304    <undefined in V1-3-0>                        minor
                      ---------
                      rad750    <undefined in V1-3-0>                        minor
                      ---------
                      sun-gcc   <undefined in V1-3-0>                        minor
   ---------------------------
   cmx_count          linux-gcc CMX_cmx_count            0.0      0.0      same
                      ---------
                      sun-gcc   CMX_cmx_count            0.0      0.0      same
   ---------------------------
   cmx_interface      linux-gcc CMX_cmx_interface        0.0      0.0      same
                      ---------
                      sun-gcc   CMX_cmx_interface        0.0      0.0      same
   --------------------------------------------------------------------------------

 4 Recommend: V2-0-0 (major)

   --------------------------------------------------------------------------------
 5 WARNING: Test area differs from CVS repository head for package CMX
```

```
? Visual
A CMX/CMX_asBuiltPub.h
A CMX/CMX_asBuiltSpy.h
R CMX/CMX_asBuilt_pub.h
? CMX/CMX_asBuilt_pub.h.defunct
M cmt/cmx_command.pl
M cmt/cmx_link.pl
M cmt/requirements
? cmt/Doxyindx.htm
? cmt/cmx_convert.pl
? cmt/cmx_doxyidx.pl
? cmt/cmx_doxypkg.pl
? cmt/cmx_test.pl
? cmt/cmx_test.txt
? cmt/cmx_test2.txt
M doc/CMX-manual.doc
? doc/CMX-manual-bak.doc
? doc/CMX-next.doc
? doc/DocSoup.doc
? doc/Template-internal.pdf
? doc/~$X-manual.doc
? doc/~WRL3082.tmp
M fragments/bsp_trailer
M fragments/exe_trailer
M fragments/shr_trailer
M sdf/CMX.log
A src/CMX_asBuilt.c
R src/CMX_asBuiltPrint.c
A src/CMX_asBuiltPrv.h
A src/CMX_asBuiltSpy.c
R src/CMX_asBuilt_prv.h
M src/cmx_asBuilt.cidc
A src/cmx_asBuiltSpy.cidc
? src/CMX_asBuiltPrint.c.defunct
? src/CMX_asBuilt_prv.h.defunct
================================================================================
tersk03:apw>
```

Figure 19 Example Output From The `cmx check version` Command

0 The command will always pick the most recent (highest) CVS tag to work with. The corresponding version must be available in the filebase so that the command can find the package's `cid` files.

1 Constituent `cid` has not changed

2 Constituent `cmx_asBuilt` has altered its interface completely and has correspondingly bumped the major interface number.

3 Constituent `cmx_asBuiltSpy` is new in this release. This counts as an interface augmentation and would only warrant a minor version number bump.

4 The most significant bump in the constituent interfaces is the major bump to `cmx_asBuilt`. This results in the recommendation to bump the major element in the CVS version (tag) string.

5 (From this line to the end of the file). Working with this command I found it very tempting to immediately issue the CMX command which would create the new tag in the CVS

repository (`cmx create version CMX --version=V2-0-0`). This usually doesn't work very well unless all the file changes in the test branch have been funneled back into CVS as well. To avoid pratfalls like this, the `cmx check version` runs a final `cvs -n update` step. If the test branch files differ from the repository head revisions, you will see an output like this.

Something the developer should be aware of which I did not put into the figure is that the `cid` file comparison is rather strict. It will for example check that a given interface instance appearing in both old and new `cid` files contain *identical* lists of symbols. Errors (syntactical or logical) will result in a constituent result of `error` or `symbol` and there will be no recommendation (other than a suggestion that the developer go back and fix up the errors). The `cmx check version` command does not itself detail the nature of the error. See the description of the `cid` executable in section 11.0 for how a developer might trace `cid` file errors.

7 CMX Internals

This section will be of most use to site maintainers, but a working knowledge of how CMX organizes its information can be useful to all. Understanding the underlying structure can go a long way to de-confuse the array of CMX commands.

7.0 Information Layering

CMX maintains information in three layers:

- The global layer. True across all sessions on all sites.
- The site (or local) layer. True across all sessions at one site.
- The session layer. True for an individual session.

Information for the first two layers is maintained in database files kept in a directory called `CDB` which lies in parallel with the public releases of CMX.

The database files are simple ASCII files and users are welcome to inspect them. Users are *not* welcome to edit them by hand. That's CMX's job.

Information for the third layer is maintained in environment variables.

7.0.0 The Global Layer

Global information is maintained in two global database files:

```
project.db
package.db
```

Ignoring blank or comment lines (comments start with a # in column one) `project.db` lines consist of three blank delimited fields:

- `A` or `D` to indicate the recommendation whether this project should be attached or detached by default.
- The name of the project.
- The string `<reserved>` (this field is currently unused but I have a plan for it should it become necessary to go to multiple CVS repositories).

Ignoring blank or comment lines (comments start with a # in column one) `package.db` lines of three blank delimited fields:

- The name of the package.
- The name of the recommended project with which to associate this package.
- The recommended production version of this package.

7.0.1 The Site Layer

Site information is maintained in three site database files:

```
project.db.<site>
package.db.<site>
tag.db.<site>
```

`project.db.<site>` associates a project name with a directory location at `<site>`. Each line consists of three fields:

- `A` or `D` to indicate whether this project should be attached or detached by default at this site.
- The name of the project.
- The directory where the project is located.

`package.db.<site>` defines the production version actually implemented at `<site>`. Each line consists of three fields:

- The name of the package
- The name of the project with which this package is associated at this site.
- The instantiated production release of this package at this site.

`tag.db.<site>` is used to define the CMT tags applicable at this site. This file is the exception that proves the rule in that it is maintained by hand. Details of its content can be found in section 8.4 “Identifying Site Appropriate Tags”.

Both `project.db` and `project.db.<site>` refer to “attaching” and “detaching” a project. This feature is not fully implemented yet.

7.0.2 The Session Layer

Session information is kept in environment variables, all of which begin with `CMX_`.

Environment variables `CMXROOT` and `CMXCONFIG` also exist. These are not part of the CMX system per se, but CMX is a CMT package and these variables are generated by CMT.

The next letter denotes the type of information:

`CMX_B`	Variables used to maintain per package binary path information.
`CMX_C`	Assorted CMX constants.
`CMX_I`	Variables used to maintain per package include path information.

CMX_L	Variables used to maintain the symbolic links.
CMX_P	Variables used to maintain per package branch information.

7.0.2.0 CMX Binary Path Information

CMX_B_<package>	Location of binary directory for package <package>.

7.0.2.1 CMX Assorted Constants

CMX_C_CDB	Location of CDB directory.
CMX_C_DOXYGEN	Filebase location where doxygen documentation is stored (optional).
CMX_C_DOXYURL	URL location where doxygen documentation is stored (optional).
CMX_C_LOGIN	Flag to indicate CMX environment initialization.
CMX_C_SITE	Site name.
CMX_C_STATUS	Status returned by most recent CMX command.
CMX_C_UNIQUE	A unique session name.

CMX_C_DOXYGEN and CMX_C_DOXYURL form a matched pair. Either both are absent or both are present. If present, CMX_C_DOXYGEN is the location of the CMX doxygen web site as viewed from the filebase on which the web site is maintained. CMX_C_DOXYURL is the URL for the same location, i.e. it is the “web address”.

7.0.2.2 CMX Include Path Information

CMX_I_<package>	Include file export directory for package <package>.

7.0.2.3 CMX Symbolic Link Maintenance Variables

CMX_L_CONFIG	Currently selected configuration (a CMX tag).
CMX_L_EXEPATH	CMX’s contribution to the PATH variable.
CMX_L_PROCESS	Location of session private link directory.
CMX_L_SHRPATH	CMX’s contribution to the LD_LIBRARY_PATH variable.

7.0.2.4 CMX Package Information

CMX_P_<package>	Three pieces of information about the package: Branch (test, dev, prod, or version number). Branch version (when branch is prod, otherwise the same as branch). Directory corresponding to the selected branch.

7.0.2.5 CMX Initialization Variables

The following CMX environment variables must be initialized before the CMX environment can be set up by issuing a cmx login or cmx start command:

CMX_C_CDB	Location of CDB directory
CMX_C_SITE	Site name
CMX_I_CMX	CMX export directory (used to locate the CMX scripts)

8 Installing CMX/CMT

Installing CMX is a multi-step command line driven process. It looks daunting at first, but can be done in about ten minutes by an expert (though it took me five or six efforts before I could describe myself as an expert!)

8.0 Prerequisites

Before attempting the installation, the installer will need to:

- Ensure that he/she has access to the flight software CVS code repository. This is covered elsewhere (see section 1.1.2 “What’s Implemented”). The installation procedures will check for this and abort if access is unavailable.
- Choose a disk location for the public development/production flight software. At SLAC this is:

  `/afs/slac.stanford.edu/g/glast/flight`

  I will refer to this location as `<flight>`
- Pick a site name. This must be unique across all participating sites, but is otherwise a completely trivial name. The site name for SLAC is `slac`, and for my test installations on the Stanford campus computers I used `hepl` (Hansen Experimental Physics Laboratory). Lower case is not mandatory, I just make fewer typing errors that way. I will refer to this name as `<site>`.
- Find out the current production version of CMT to install. At time of writing, this is `V1-6-3`, but that’s likely to change as CMT evolves. To find out the latest version number, contact me (apw@slac.stanford.edu). I will refer to this version as `<CMT-version>`
- Build or find an up to date copy of the GNU `binutils` package. The package should be built for cross platform work with the PPC CPU as the target architecture. In particular, CMX will want to use the linker and the object copy utilities and expects to find them under the names `ldppc-2.11.2` and `objcopyppc-2.11.2`. My apologies for embedding the version number in the executable name. It perfectly acceptable to use any version of `binutils` version 2.11 or better. It just gets confusing if someone builds `binutils` 2.12 but to satisfy CMX has to embed 2.11.2 in the name. At the time I was just trying to avoid a name clash with the VxWorks provided tools on the same name but of ancient vintage.

I will also use one other meta-string: <CMX-version>. This is the version number of CMX being installed. The installer need not worry too much about the value of this string because the installation process will always pick up the latest production version of CMX. To discover the actual value of <CMX-version>, look for a line in the output from INSTALL_CMX which begins "Identify version of CMX to fetch..".

8.1 Grab The Installation Scripts

The installation scripts are themselves maintained in CVS so the first step is to obtain a throw away copy of CMX and to copy the scripts to the proper location:

```
tersk02:apw> cd $HOME
tersk02:apw> mkdir deleteme
tersk02:apw> cd deleteme
tersk02:apw> cvs checkout CMX
.
<lots of output from the cvs checkout command>
.
tersk02:apw> cp CMX/cmt/INSTALL_CMT <flight>/
tersk02:apw> cp CMX/cmt/INSTALL_CMX <flight>/
tersk02:apw> cd ..
tersk02:apw> rm –rf deleteme
tersk02:apw>
```

8.2 Do The CMT Installation

Because I have done some modifications to CMT for flight software, CMT should *not* be installed from Christian Arnault's site at IN2P3. The following script correctly fetches the modified CMT from the flight software CVS repository. Output from this script is sent to both the screen and a log file called install_cmt.log (which the installer should send to me if something goes wrong):

```
tersk02:apw> cd <flight>
tersk02:apw> ./INSTALL_CMT <CMT-version>
.
<lots of output from the INSTALL_CMT command ending with>
.
Run the CMT INSTALL script.....................(result deferred)
==========================================
       CMT installation terminated.
           --------------------
 cmt.exe is not built on this site
 you might need to build it as follows:
 > source setup.csh
 > [g]make
==========================================
Run the CMT INSTALL script.....................OK
*************************************************************************
INSTALL_CMT has completed successfully, but CMT is not yet installed!
Please follow instructions from CMT's own INSTALL (about sourcing the
file setup.csh and running gmake) to complete the CMT installation.

You will need to 'cd CMT/<CMT-version>/mgr' first.

Quick tip for Sun installations.  CMT needs help to compile on Sun
using the gcc compiler.  Provide an extra flag as follows:

gmake cppflags=-D_BOOL

*************************************************************************
tersk02:apw>
```

At this point, CMT source code has been downloaded and laid out correctly in the `<flight>` directory. Unfortunately the next step is a `source` command, which cannot be done inside a script, so to complete the installation (i.e. follow the instructions given in the last few lines printed by `INSTALL_CMT`):

```
tersk02:apw> cd CMT/<CMT-version>/mgr
tersk02:apw> source setup.csh
.
<output from the source command>
.
tersk02:apw> gmake              (or gmake cppflags=-D_BOOL on a Sun machine)
.
<lots of output from the make process>
.
tersk02:apw>
```

The CMT installation is now complete.

8.3 Do The CMX Installation

CMX installation is also done with a script and the output is again logged (to `install_cmx.log`) as well as being sent to the screen:

```
tersk02:apw> cd <flight>
tersk02:apw> ./INSTALL_CMX <site>
.
<lots of output from the INSTALL_CMX command ending with>
.
Fetch (development) CMX from repository..........OK
Configure (development) CMX for CMT..............(result deferred)
Configure (development) CMX for CMT..............OK
*************************************************************************
INSTALL_CMX has completed successfully.
*************************************************************************
tersk02:apw>
```

The CMX installation is now complete.

8.4 Identifying Site Appropriate Tags

At this point, the installer will need to identify appropriate CMT tags for the site. The CMX installation will have made an attempt to come up with the right tags, but there is no way to fully automate this task.

In parallel with the CMX version directories there is a directory called `CDB` containing the CMX database files. The tag file for site `<site>` is called:

`<flight>/DAQ/source/CMX/CDB/tag.db.<site>`

The comments at the top of the file should help out, but I'll try to give a fuller description here.

CMT constructs a default tag which it keeps in the environment variable `CMTCONFIG`. CMX uses this value to index a list of CMX tags which are valid for the given CMT default tag. Thus in the SLAC file `tag.db.slac`:

```
sun4x_58          sun-gcc,-mv2304,-mcp750,-rad750,+i845e
i386_linux22      linux-gcc
```

This says that for platforms at SLAC that come up with the CMT default tag `sun4x_58`, the valid CMX tags are `sun-gcc`,`mv2304`, `mcp750`, `rad750` and `i845e`. For platforms at SLAC that come up with the default CMT flag of `i386_linux22`, the only valid CMX tag is `linux-gcc`.

In addition, the CMX tag listed first is considered to be the default CMX tag. Thus when someone logs into a `sun4x_58` box at SLAC, all the CMX links are set up for CMX packages compiled with the `gcc` compiler. The only other piece of significant notation is the minus sign before `mv2304`, etc., and the plus sign before `i845e`. Both the minus sign and the plus sign indicate a cross-compilation tag, the minus sign indicating a PowerPC target and the plus sign a Pentium target.

The installer should thus edit the file `tag.db.<site>` to ensure that it corresponds to local conditions. For single platform sites the file will contain just one (active) line. The first token on the line should be the value of the environment variable `CMTCONFIG`, and the subsequent list of tokens should reflect the capabilities of the site.

8.5 Identifying Site Appropriate Architectures

For sites supporting either multiple target architectures (like PowerPC and Pentium) or multiple versions of the VxWorks embedded system tools, it will be necessary to set up an architecture database as well. Once again, the CMX installation cannot outguess the capabilities of the

installation site and so only provides a template for this file which the installer should copy (analogously to the `tag.db.<site>`) file to:

`<flight>/DAQ/source/CMX/CDB/arch.db.<site>`

Ignoring comments, the lines of this file look like (this is the content of the SLAC file):

```
D 5.4:ppc /afs/slac.stanford.edu/package/vxworks/devel/tor-2.0.2/glast
A 5.5:ppc /afs/slac.stanford.edu/g/glast/flight/vendor/vxworks/tor-2.2/ppc
A 5.5:x86 /afs/slac.stanford.edu/g/glast/flight/vendor/vxworks/tor-2.2/x86
```

Figure 20 Contents for the file `$CMX_C_CDB/arch.db.<site>`

The key to the database (the second token on the line) is simply a VxWorks version number concatenated to an architecture name with a colon. The value of this key (the third token on the line) is the location of the VxWorks installation (effectively the value that would normally be found in the environment variable `WIND_BASE` for that version/architecture combination).

For the moment, CMX only understands VxWorks version `5.4` and `5.5` and architectures `ppc` and `x86`.

8.6 Adding Users

At this point CMX/CMT is available to users, but the user must do a little extra work to take advantage of it. Each user must of course have access to the flight software CVS repository at SLAC before attempting this.

In the user's login file `.cshrc` (the strings `<site>` and `<flight>` should be substituted as appropriate):

```
.
.
setenv CMX_C_SITE <site>
setenv CMX_C_CDB  <flight>/DAQ/source/CMX/CDB
source ${CMX_C_CDB}/cmx_start
.
.
```

In the user's login file `.login`:

```
.
.
cmx login
.
.
```

In the user's logout file `.logout`:

```
.
.
cmx logout
.
.
```

The `.login` and `.logout` files are more for convenience than necessity. If the `cmx login` command is not performed in the `.login` file, it will be performed by the first CMX command issued during the session. The command `cmx logout` cleans up the directory where CMX stores session associated information. If `cmx logout` is not performed, that directory is not destroyed (and the user ends up with a lot of junk directories in the directory `${HOME}/CMX`).

8.7 Bringing The Site Up To Date

For this section I will assume:

- CMT and CMX installed correctly
- The installer has made the necessary modifications to `.cshrc`, `.login` and `.logout` for the installer's account
- The installer has started a new session to access CMX

8.7.0 Bringing Projects Up To Date

Look at the output from the command `cmx check site`. If there are any global projects not yet instantiated on the local site, there will be a section in the output listing them. Any uninstantiated projects will need to be given an identified home in the local site's filing system. There is no real restriction on where a project is installed, but the habit at SLAC is to keep all projects in parallel under the `<flight>` directory.

Having decided where to place a new project (I will call this location `<project>` where `<project>=<flight>` at SLAC), the installer should identify that location to CMX:

```
tersk02:apw> cmx set stem <project_name> <project>
tersk02:apw>
```

Repeat this process until there are no uninstantiated projects listed in the output from `cmx check site`.

8.7.1 Bringing Packages Up To Date

Once all projects have been instantiated, the output from `cmx check site` should only be listing uninstantiated packages. For each such package:

```
tersk02:apw> cmx fetch <package_name> --dev
<a few lines from CMX>
<lots of lines from CVS>
<a few lines from CMT>
tersk02:apw> cmx fetch <package_name> --prod –-version=<version>
<a few lines from CMX>
<lots of lines from CVS>
<a few lines from CMT>
tersk02:apw>
```

Where `<version>` is the version number listed in the `cmx check site` output.

At this point the source for the development and production versions of the package have been downloaded, but the package has not been built. To build the development version:

```
tersk02:apw> cmx set branch <package_name> --dev
tersk02:apw> cmx build <package_name> --dev –alltags
<output from the build, one group of lines for each CMT tag>
tersk02:apw>
```

and for the production version:

```
tersk02:apw> cmx set branch <package_name> --version=<version>
tersk02:apw> cmx build <package_name> --version=<version> --alltags
<output from the build, one group of lines for each CMT tag>
tersk02:apw>
```

If this all succeeds, identify the production version of this package at this site:

```
tersk02:apw> cmx set production <package_name> --version=<version>
tersk02:apw>
```

Finally, make sure that everything has taken and that the local session correctly reflects the installation of the package:

```
tersk02:apw> cmx set branch <package_name> --prod
tersk02:apw> cmx show branch <package_name> –-full
<package_name>  prod     <version>
tersk02:apw>
```

Now repeat for other packages, ad nauseam!

8.8 Continuing Site Maintenance

For packages maintained by CMX (a list which includes CMX itself), site maintenance (i.e. keeping up with code development at other sites) is achieved with exactly the same methods as have just been outlined. Upgrades to CMT (which is *not* maintained by CMX) are somewhat more problematic and will probably be achieved by my sending out very specific instructions.

9 CMX And Doxygen

The preparation of documentation using doxygen is a big topic in its own right and will not be covered in detail in this manual (consider this a promissory note for another document “LAT Flight Software Development Guidelines”). This section will only deal with the interaction between CMX and doxygen.

9.0 When Does CMX Run Doxygen

CMX runs doxygen as the last step of building a constituent, thus documentation is prepared on a constituent by constituent basis. For each constituent, CMX constructs a list of files by taking all the source files for a constituent and all files included by those source files. It then rejects all files which do not come from the package being built. The user can then adjust this list using the magic macros `<con>__insertdox` and `<con>__removedox`. This list of files is then embedded in a standardized doxygen command file and doxygen run on it.

Well it’s a little more complicated than that … see the CMX script file `cmx_doxygen.pl` for all the gory details … but that is certainly the gist of CMX doxygen processing.

Note that the doxygen step can be side-stepped for test builds using the `--nodoxygen` qualifier to the build command, but cannot be side-stepped for development or production builds.

9.1 Where CMX Puts The Output

The output from a doxygen run can go to one of two places, depending on whether a particular site has chosen to publish the doxygen output on a web site or to keep it privately. To become a web site, the site must obviously have all the infrastructure to support a web site but in addition each user on the site must define two extra environment variables:

`CMX_C_DOXYGEN`	Filebase location where documentation is stored
`CMX_C_DOXYURL`	URL location where documentation is stored

In fact I regard this as a CMX misfeature (to be polite). It should not be up to individual users to decide if a site is a web site or not. This decision should be made once by the site maintainer and users should know nothing about it. SLAC is a web site and the problem has been partially solved by asking all users to source a group login file during login. The correct solution would be to add

another site maintained file in the CMX database directory. Ah well, there's always something.

9.1.0 Where CMX Puts The Output For A Non Web Site

From a CMX perspective, doxygen is just another transformation engine, like a compiler. Source goes in one end, documentation comes out the other. For this reason, the documentation produced by CMX/doxygen is put in the binary tree of a package (and does not go into CVS). Doxygen adds the following elements to the (public) binary tree:

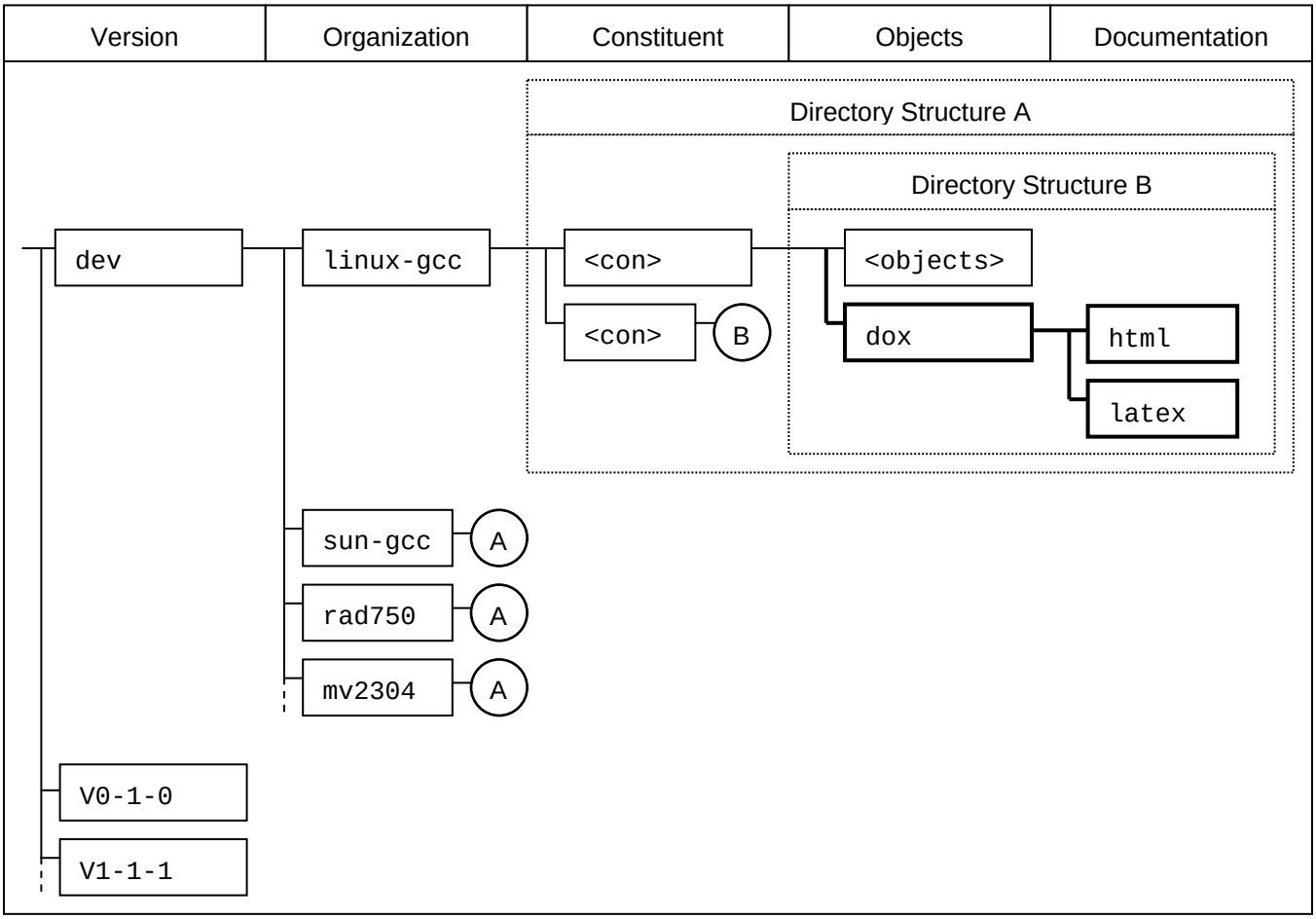

Figure 21 Public Directory Structure Extensions For Doxygen

For documentation built in a private package, the layout is similar (though obviously the version level would read `test`)

The `dox` directory at the objects level unfortunately pollutes the name space a little. Please do not create a host executable with name `dox`.

9.1.1 Where CMX Puts The Output For A Web Site

First of all, it's only possible to direct the output from development and production builds to the web site. Output from test builds always follows the rules for a non-web site. This is convenient for developers because it's possible to build and inspect documentation in a private space before launching it into wider community.

The web site directory structure from the project down is identical to that which can be found in the normal binary tree descent in the public filespace. The only constraint imposed by CMX is that all projects appear in the directory specified by the environment variable `CMX_C_DOXYGEN`. At SLAC, `CMX_C_DOXYGEN` is defined to be:

/afs/slac.stanford.edu/www/exp/glast/flight/doxygen

Of course, this isn't very useful if you want to reach the site via a URL. This is what is defined in the `CMX_C_DOXYURL` environment variable, which for SLAC is (you might want to bookmark this):

http://www.slac.stanford.edu/exp/glast/flight/doxygen/

This is where the web ugliness comes in. That address is simply the top of a directory structure similar to the CMX binary tree. Navigating to the documentation for a particular package/constituent requires a fair knowledge of CMX directory structures (and about eight mouse clicks!)

10 CMX And VxWorks Tornado

The CMX command reference section defined the syntax to the `cmx tornado` command but gave few details for its use. This section will seek to rectify that.

10.0 The Sequence Of Operations

The simplest way to demonstrate what the `cmx tornado` command is doing is to write it as a pseudo-script:

```
If an initialization script has been provided, parse it to a temporary file.
If an exit script has been provided, parse it to a temporary file.
Start a VxWorks target server.
If initialization script provided, run VxWorks windSh with the parsed version.
If interactive session specified, run VxWorks windSh interactively.
If exit script provided, run VxWorks windSh with the parsed version.
Delete any parsed initialization or exit scripts.
Kill the VxWorks target server.
```

Figure 22 A `cmx tornado` Command Pseudo-Script

Doesn't look hard … but you should see the size of the real script.

At face value, this doesn't seem to provide much value added over simply typing the commands themselves. The value added derives from two factors:

- There's magic in the parsing of the initialization and exit scripts to temporary files. This will be the topic of the next section.
- This is the start of an embedded system scripting environment. It can be used as the basis for an automated test system.

10.1 Parsing Initialization And Exit Scripts

It is the parsing of the initialization and exit scripts that ties the embedded system environment back to the CMX environment. By burying a small set of commands in comment blocks (this works for both C style scripts and TCL scripts (I hope)), script writers can make logical references to CMX objects. This is most easily demonstrated with an example. This is a real script used during the development of the `cmx tornado` command:

```
 0  /* cmx> boot                    */
 1  /* cmx> load CMX cmx_asBuilt    */
    /* cmx> load PKGC pkgc          */
    /* cmx> load PKGB pkgb          */
    /* cmx> load PKGA pkga          */
 2  CMX_asBuiltPrint
```

Figure 23 A `cmx tornado` command Initialization Script

0 The general syntax of a buried command is always:

Comment introduction (`/*` in this C style script but # in a TCL script)
White space
The string `cmx>` (exactly)
White space
Command (command verbs are so far limited to `boot` and `load`)
In C style scripts, white space up to the closing `*/`

This line demonstrates the `boot` verb. It takes no parameters or qualifiers. In the parsed version of the script, this is replaced with the (uncommented) word `reboot`.

1 The `load` command interprets the first and second parameters as a package name and a constituent name respectively. These are translated through the current CMX environment description to produce an absolute file name. In the parsed file, the line is then replaced with:

```
ld <n>,<m>,"<filename>"
```

Where `<filename>` is the absolute file just derived and `<n>` and `<m>` are the traditional arguments to the windSh `ld` command controlling “Symbol Visibility” and “Common Symbol Policy” respectively. I have written them as meta-symbols because they can be controlled by qualifiers added to the command. The following line would force symbol visibility to 1 and common symbol policy to 2 for the package/constituent `CMX/cmx_asBuilt` *only:*

```
/* cmx> load CMX cmx_asBuilt –-symbol=1 –-common=2  */
```

If the qualifiers are omitted, then `<n>` and `<m>` are taken from the global options entered on the `cmx tornado` command line. If those in turn are omitted, the program defaults are `--symbol=0` and `--common=1`.

2 Finally, a “natural” script line. This is copied across to the parsed file as is.

11 Assorted CMX Goodies

I'm using this section to accumulate documentation for scripts/executables that have become part of CMX but which don't fall simply into other sections of this manual.

11.0 Analyzing **`cid`** files

Analyzing `cid` files is not completely trivial and was not something I wished to undertake in a scripting language. I therefore wrote an executable called `cid` to do the job. This was originally intended for purely internal use by CMX, but it turned out to be so useful, I've spruced it up a little and it's now available as a command. Note that `cid` operates on the files used as input to the linker, so the `<cid_file>` in what follows will normally have the extension `.cido` (`cid` will fail if run on `.cidc` files).

The basic syntax follows the same rules as for the `cmx` command, so I will not repeat them. The commands are as follows:

11.0.0 cid check

Do a basic syntax check. If the `.cido` file contains no errors, the command is silent.

```
cid check <cid_file>
```

11.0.1 cid compare

Compare an "old" and a "new" `.cido` file to ensure that the new file is a reasonable evolution of the old file. Used internally by the `cmx check version` command, but `cmx check version` does not use the `--full` qualifier and the printing of error messages is suppressed. Use this `cid` command with the `--full` qualifier to list the error messages.

```
cid compare <old_cid_file> <new_cid_file> [--full] [--[no]title]
```

- `--full`

  Print error messages. Unless this qualifier is specified, `cid compare` only prints a one line summary for each interface found.

- `--[no]title`

Default is `--title`. `--notitle` suppresses the printing of the title block above the interface summary. Useful inside `cmx` commands to make the output easier to parse.

11.0.2 cid find

Identify the interface in which a global symbol appears. Multiple symbols can be searched for with a single command.

```
cid find <cid_file> <global> [<global> ...]
```

11.0.3 cid show global

List all global symbols in all interfaces in the `.cido` file.

```
cid show global <cid_file>
```

11.0.4 cid show interface

List the interface hierarchy in a `.cido` file with optional display of interface instance strength and interface instance global symbol content.

```
cid show interface <cid_file> [--global] [--[no]indent] [--strength]
```

- `--global`

  List global symbols defined in interface instances as well.

- `--[no]indent`

  Default is `--indent`. `--noindent` suppresses the pretty printing (indenting) of the instance hierarchy. Useful inside `cmx` commands to make the output easier to parse.

- `--strength`

  Indicate the strength of an interface instance by placing an `S` (for strong) or `W` (for weak) after the interface instance name. Patch revisions are always weak, all others are strong. Used inside `cmx` commands (usually with the `--noindent` qualifier) to make the output easier to parse.

11.1 Counting Lines Of Code

Flight software has been under a lot of pressure to produce lines-of-code counts, both estimated and measured. In self defense, I put together a trivial lines-of-code counter. It's not at all sophisticated and only works on (well formed) C/C++ source code. About the only thing in its favour is that it's very quick.

The syntax is likewise very simple. Just invoke the command `cmx_count` on a list of files. This is the result of running the counter on CMX source code:

```
tersk08:apw> cmx_count src/*.c src/*.h CMX/*.h
================================================================================
Pproc  Semi-  <--Comment--->  <----Code---->  Filename
Direc  Colon    Chars  Lines    Chars  Lines
-----  -----  -------------  -------------  --------------------------------
    8     24     1477     42     1570     58  src/CMX_asBuiltPrint.c
    7     47     2896     76     1954     89  src/CMX_asBuiltUpdate.c
    6     21     1847     50     1008     40  src/CMX_interfaceCollate.c
    7    137     2854    109    10168    313  src/CMX_interfaceCompare.c
    6      6     1001     31      441     23  src/CMX_interfaceFind.c
   10    355    11129    366    20447    728  src/CMX_interfaceRead.c
    6     15     1768     44     1105     58  src/CMX_interfaceRecurse.c
    6     22     2078     62     1527     71  src/CMX_interfaceShow.c
    7    178    10915    335    11255    435  src/cid.c
   10    119     3433    106     6735    195  src/cmx_count.c
    7     34     4158     80     1462     48  src/CMX_asBuilt_prv.h
    8     29     3610     79     1525     55  src/CMX_interface_prv.h
    7      1      413      9      196     11  CMX/CMX_asBuilt_pub.h
   12      6     1444     29     1009     35  CMX/CMX_interface_pub.h
-----  -----  -------------  -------------  --------------------------------
  107    994    49023   1418    60402   2159  Total (14 files)
================================================================================
tersk08:apw>
```

Figure 24 Example Output From The `cmx_count` Executable

12 Future Development

It's my hope that CMX can now go to a maintenance mode. There will doubtless be minor dinks to fix bugs but I do not expect to implement any more major functionality. There are always exceptions of course:

12.0 Substantial Developments

12.0.0 More Kernel/BSP Support

CMX can now produce:

- Kernel/BSPs without symbols and without compression. Typically used in a Tornado style development environment.
- Kernel/BSPs with symbols but without compression. Typically used in conjunction with the embedded shell to download and drive a target system without Tornado.
- Kernel/BSPs with symbols, with compression. Typically used to burn into target EEPROM for test stands et al.

I hope that this is sufficient, but I've said that before…

12.0.1 Support For The Front End Simulators

The current plan calls for code management for the front end simulators to be done with CMX. This will involve adding a new tag for x86 (Pentium) embedded systems. More news when we have the WindRiver x86 architecture/support package on site!

12.1 Minor Developments

12.1.0 Improving The Web Site

I have recently made some improvements to web site navigation when hunting down doxygen generated documentation. A web whiz could improve it still further, but I intend to let it lie at this point.