

CMT

Configuration Management Tool

Version v1r8

Christian Arnault

arnault@lal.in2p3.fr

Contents

1. Copyright.
2. Presentation.
3. The conventions.
4. The architecture of the environment.
 1. Supported plateforms
5. Installing a new package.
6. Localizing a package - The **CMT**PATH environment variable.
7. Managing site dependent features - The **CMT**SITE environment variable.
8. Configuring a package.
9. Working on a package.
 1. Working on a library.
 2. Working on an application
 3. Selecting a specific configuration.
 4. Working on a test or external application
 5. Construction of a global environment
10. Defining a document generator.
 1. Writing a fragment pattern for a generator
11. The tools provided by CMT
 1. The **requirements** file
 1. The general requirements syntax
 2. The complete requirements syntax
 2. The concepts handled in the requirements file
 1. Meta-information : author, manager
 2. package, version
 3. Constituents : application, library, document
 4. Groups
 5. Languages
 6. Symbols : alias, set, set append, set prepend, set remove, macro, macro append, macro prepend, macro remove, macro remove all, path, path append, path prepend, path remove
 7. use
 8. pattern, apply pattern, ignore pattern

9. branches
10. build strategy, version strategy
11. setup script, cleanup script
12. include path
13. include dirs
14. make fragment
15. public, private
16. tag
3. The general **cmt** user interface
 1. **cmt broadcast** [-select=list] [-exclude=list] [-local] [-depth=n] [-all packages] <shell command>
 2. **cmt build** <option>
 3. **cmt check configuration**
 4. **cmt check files** <reference-file> <new-file>
 5. **cmt checkout** ...
 6. **cmt co** ...
 7. **cmt cleanup** [-csh|-sh]
 8. **cmt config**
 9. **cmt create** <package> <version> [<area>]
 10. **cmt filter** <in-file> <out-file>
 11. **cmt help**
 12. **cmt lock** [<package> <version> [<area>]]
 13. **cmt remove** <package> <version> [<area>]
 14. **cmt remove library links**
 15. **cmt run shell-command**
 16. **cmt setup** [-csh|-sh]
 17. **cmt show** <option>
 18. **cmt system**
 19. **cmt unlock** [<package> <version> [<area>]]
 20. **cmt version**
 21. **cmt cvstags** <module>
 22. **cmt cvsbranches** <module>
 23. **cmt cvssubpackages** <module>
4. The setup and cleanup scripts
5. **cmt build prototype**
12. Using **cvs** together with **CMT**
 1. Importing a package into a **cvs** repository
 2. Checking a package out from a **cvs** repository
 3. Querying **CVS** about some important infos
 4. Working on a package, creating a new release
 5. Getting a particular tagged version out of **cvs**
 1. Installing **CMT** on your Unix site
 2. Installing **CMT** on a Windows or Windows NT site
13. Interfacing an external package with **CMT**
14. Installing **CMT** for the first time
15. Differences with previous versions of **CMT**
 1. Converting a package that was managed with previous versions of **CMT** (or methods)

2. Operations in a Windows environment
 16. Appendices
 1. Standard make targets predefined in CMT
 2. Standard macros predefined in CMT
 1. Structural macros
 2. Language related macros
 3. Package customizing macros
 4. Constituent specific customizing macros
 5. Source specific customizing macros
 6. Generated macros
 7. Utility macros
 3. Standard templates for makefile fragments
 4. Makefile generation sequences
 5. The complete requirements syntax
 6. The internal mechanism of cmt cvs operations
-

1 - Copyright.

Copyright (c) 1996 LAL Orsay, UPS-IN2P3-CNRS (France).

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the
Computer Application Development Group at LAL Orsay
(Laboratoire de l'Accelérateur Linéaire - UPS-IN2P3-CNRS).

- Neither the name of the Institute nor of the Laboratory may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the LAL and contributors “as is” and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the LAL or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

2 - Presentation.

This environment, based on some management conventions and comprising several shell-based utilities, is an attempt to formalize software production and especially configuration management around a *package*-oriented principle.

The notion of *packages* represents hereafter a set of software components (that may be applications, libraries, documents, tools etc...) that are to be used for producing a *system* or a *framework*. In such an environment, several persons are assumed to participate in the development and the components themselves are either independent or related to each other.

The environment provides conventions (for *naming* packages, files, directories and for *addressing* them) and tools for *automating* as much as possible the implementation of these conventions. It permits to *describe* the configuration requirements and automatically deduce from the description the effective set of configuration parameters needed to operate the packages (typically for *building* them or *using* them).

CMT lays upon some organisational or managerial principles or mechanisms described below. However, it permits in many respects the users or the managers to *control*, specialize and customize these mechanisms, through parameterization, strategy control and generic specifications.

- Many such packages are produced and maintained.
- The packages may or not be related to each other (defining a *direct acyclic graph* of packages - not just a single tree).
- Each *executable application* (from now on simply named *applications*) either belongs to a particular package and/or defines its own environment and then makes use of some other packages.
- Each package can be uniquely identified within the system or the framework by a *name* which is usually a short *mnemonic* and which may be also used for isolating its name-space (eg. by *prefixing* components of the package by its mnemonic).
- A package installed in this environment may be *exported* to a site where the architecture is reproduced, and as long as the local organisation defined for the package is preserved through the transport, the reconstruction procedure will be preserved. Configuration specifications can be easily provided to cope with machine, site or system specific features.
- Packages are maintained consistently to their declared relationships to each other through a *version* identification model based on :
 - a version is defined with a mnemonic comprising at least two numbers for the *major* id and the *minor* id,
 - versions with different major ids are said to be incompatible,
 - versions with same major ids but different minor ids are said to be backward compatible with respect of the minor id ordering.
- Version control and management schemes (eg. by using **CVS**) are usually consistently operated, applying the conventions on organization and version identification.
- An application that uses one or several packages managed in this environment should not itself be constrained to be managed this way. The tools should only require a few exported features (such as a few environment variables) for referencing any given package.
- similarly, a package maintained in this environment must be able to use packages that are *not* managed in this environment.

Following these definitions, the basic configuration management operations involved here (and serviced by the **CMT** tools) consist of :

- installing the packages in conventional locations so that they can be referenced by each other,
 - describing the configuration requirements for each package:
 - dependencies to other packages,
 - symbols to be exported to client packages (environment variables, make macros, etc...)
 - components of the packages (libraries, applications, documents)
 - **Make** macros
 - Strategies that **CMT** should follow at run time, overriding its default ones.
 - Generic behavioural patterns meant to describe generic configuration items.
 - deducing the effective configuration parameters from the requirements so as to automatize the building phases and the run-time operations and connections between packages (typically for generating makefiles, generating compiler and linker options, shared libraries paths etc...). This construction mechanism follows customizable strategies (eg. for selecting among possible alternate versions of available packages).
-

3 - The conventions.

This environment relies on a set of conventions, mainly for organizing the directories where packages are maintained and developed :

- Each package is installed in a standard directory structure defined at least as follows:

```
<some root>/<Package mnemonic>/<version mnemonic>/cmt
```

or/and

```
<some root>/<Package mnemonic>/<version mnemonic>/mgr
```

This style of organization should be considered as the basic (and unique) criterion for a package to be recognized as a valid *CMT package*.

However, many other parallel branches (similar to **cmt** and **mgr**) such as **src**, **include** or **test** may be added to this list according to the specific needs of each package. In particular, a set of such parallel branches are expected to contain *binary* files (those that compilers, linkers, archive managers or other kinds of code or pseudo-code generators can produce). They are named - with a *token* - according to the way files have been produced and to the particular *configuration* used for that (such as the machine or operating system type). The **CMT** toolkit provides the **cmt system** utility (a shell script) that provides a default value for this token. An environment variable (**CMTCONFIG**) is also assigned to this value.

Each branch may in addition be freely structured, and there is no constraint to the complexity of this organization.

There are no constraints on the number of roots into which **CMT** packages are installed. We'll see later on how the different roots will be declared and identified by **CMT**.

examples of such structure can be :

```
.... /Cm/v7r7/src
.... /CMT/v1r8/src
```

- The typical *leaves* of a package directory structure are therefore:

cmt or mgr	for the management utilities (such as the Makefile)
src	the sources of the package
doc	for the package documentation
\${CMTCONFIG}	for the produced binaries (compiled objects, libraries, executables)

As many other useful branches as needed may also be defined according to the specific needs of the packages.

Any other deeper hierarchy structure may also be defined, such as for instance, organizing the sources into a hierarchy of branches.

4 - The architecture of the environment.

This environment is based on the fact that one of its packages (named **CMT**) provides the basic management tools. This package has very little specificities and as such must itself obey the general conventions. The major asymetry between **CMT** and all other packages is the fact that once **CMT** is installed it implicitly defines the *default* root for all other packages (through the environment variable **CMTROOT**).

Then packages may be installed either in the default root or in completely different areas. The only constraint in this case being that their root will have to be specified explicitly.

A typical configuration for this environment consists of selecting a public area (generally available from several machines through an **NFS** or **AFS**-like mechanism), installing the **CMT** basic package, and then installing user packages in this default root or in private ones. One frequent semantic given to this style of configuration is to consider the packages installed in the area hanging below default root as the publicly available version, whereas packages installed elsewhere are rather meant to be managed in a private context, or in the context of a non public project. However, dependencies between packages will always be possible (as long as the system based protections provide appropriate access rights).

CMT is operated through one main user interface : the **cmt** command, which handles the **CMT** conventions and which provides a set of services for :

- creating a new package, installing it below the default root or in a private area. This operation will create or check the local package directory tree and build up several minimal scripts that can be customized,

- describing or monitoring :
 - the relationships between the package and other packages and the (public) exporting features the package should provide,
 - the set of features needed for the package development (private features)
 - the components of the package in terms of libraries or executables,
- automatically generating the reconstruction scripts (**makefiles**) from this description.
- recursively acting upon the hierarchy of used packages.

Several other utilities are also provided for some specific activities (such as the automatic production of shared libraries, C prototypes, management of interactions between **CVS** and **CMT** itself, the management of a similar architecture for **Windows** or **OS9**, setting up protections onto packages (though locks) etc...).

4.1 - Supported platforms

CMT has been ported and tested on a wide range of machines/operating systems, including :

- DEC-Unix V4.0
- HP-UX-10 (several types of platforms)
- AIX-4
- Solaris
- IRIX
- Several variants of LynxOS
- Linux 2.0
- Windows 95/98/NT (**nmake** based environment and **MSDev/VisualC++** environment)

This in particular means that a package developped on one platform may be re-configured towards any of these platforms without any change to its configuration description (setup scripts, makefiles, ...).

5 - Installing a new package.

We consider here the installation of a user package. Installing **CMT** itself requires special attention and is described in dedicated section of this document.

Therefore, we assume that a root directory has been selected by the system manager, and that **CMT** is already installed in this area. One first has to *setup* **CMT** in order to gain access to the various management utilities, using for example the shell command:

```
csh> source /lal/CMT/v1r8/mgr/setup.csh
```

or

```
ksh> . /lal/CMT/v1r8/mgr/setup.sh
```

or

```
dos> \lal\CMT\v1r8\mgr\setup.bat
```

Obviously, this operation should be performed before any other **CMT** action. Therefore it is often recommended to install this setup action straight in the *login* script.

The *setup script* used in this example is a constant in the **CMT** environment : every package installed in it will have one such setup script defined. It is one important entry point to any package (and thus to **CMT** itself). It generally contains environment variable definitions and invocations of setup scripts for related packages (A corresponding cleanup script is also provided). However, most packages do not really require *using* at all the setup scripts especially if they do not define any environment variables. This script contains a uniform mechanism for *interpreting* the requirements file so as to dynamically define environment variables, aliases for the package itself and all its used packages. It is constructed once per package installation by the **cmt create** command, or restored by the **cmt config** command (if it has been lost).

A package is primarily defined by a *name* and a *version* identifier. These two attributes will be given as arguments to **cmt create** such as in the following example :

```
csh> cd mydev
csh> cmt create Foo v1
-----
Configuring environment for package Foo version v1.
CMT version v1r8.
Root set to /users/dsksi/arnault/mydev
System is alpha
-----
Installing the package directory
Installing the version directory
Installing the cmt directory
Creating setup scripts.
Creating cleanup scripts.
```

The package creation occurred from the current directory, creating from there the complete directory tree for this new package.

In the next example, we install the package in a completely different area, by explicitly specifying the path to it as a third argument to **cmt create** :

```
> cmt create Foo v1 ~/Packages
-----
Configuring environment for package Foo version v1.
CMT version v1r8.
Root set to /users/dsksi/arnault/Packages.
System is alpha
-----
Installing the package directory
Installing the version directory
Installing the cmt directory
Creating setup scripts.
Creating cleanup scripts.
```


Several file creations occurred at this level :

- a minimal directory tree for the package, including **src** and **cmt** (the other branches will be installed either at build time or if specified in the **requirements** file).
- an empty configuration specification file (named **requirements**) installed in the **cmt** branch.
- A minimal **Makefile**, containing

```
include $(CMTROOT)/src/Makefile.header

include $(CMTROOT)/src/constituents.make
```

This **Makefile** does not need any modification to build any of the constituents managed by **CMT**. The intermediate makefile fragments will always be re-generated transparently and automatically at build time. However (and thanks to this feature), this file will not be modified anymore by **CMT** itself. Thus you may insert any particular make statement you would feel appropriate, typically when you ask for operations that cannot be taken into account by **CMT**.

- A similar minimal **NMake** file, containing

```
!include $(CMTROOT)\src\NMakefile.header

!include $(CMTROOT)\src\constituents.nmake
```

- a first (almost empty) version of the setup and cleanup scripts (one flavour for each shell family).

One *may* then setup this new package by running the setup script (which will not have much effect yet since it's almost empty) :

```
csh> cd /lal/Foo/v1/cmt
csh> source setup.csh
```

or

```
csh> cd ~/Packages/Foo/v1/cmt
csh> source setup.csh
```

or

```
dos> cd \Packages\Foo\v1\cmt
dos> setup.bat
```

The **FOOROOT** environment variable is the only parameter defined automatically by this operation.

It should be noted that running the setup script of a package is no longer necessary for building operations. The only situation where running this script *may* become useful, is when an application is to be run, while requiring domain specific environment variables defined in one of the used packages. Besides this particular situation, running the setup scripts may not be needed at all.

Lastly, this newly created package may be removed by the quite similar remove command, using exactly the same arguments as those used for creating the package.

```

csh> cd mydev
csh> cmt remove Foo v1
-----
Removing package Foo version v1.
CMT version v1r8.
Root set to /users/dsksi/arnault/mydev/A.
System is alpha
-----
Version v1 has been removed from /users/dsksi/arnault/mydev/A
Package Foo has no more versions. Thus it has been removed.

```

or:

```

csh> cmt remove Foo v1 ~/Packages
-----
Removing package Foo version v1.
CMT version v1r8.
Root set to /users/dsksi/arnault/Packages/A.
System is alpha
-----
Version v1 has been removed from /users/dsksi/arnault/Packages/A
Package Foo has no more versions. Thus it has been removed.

```

So far our package is not very useful since no constituent (application or library) is installed yet. You can jump to the section showing how to work on an [application](#) or on a [library](#) for details on these operations or we can roughly draw the sequence used to specify and build the simplest application we can think of as follows:

```

csh> cd ../src
csh> cat >FooTest.c
#include
int main ()
{
    printf ("Hello Foo\n");
    return (0);
}

csh> cd ../cmt
csh> vi requirements
...
application FooTest FooTest.c
csh> gmake
csh> ../${CMTCONFIG}/FooTest.exe
Hello Foo

```

This can still be simplified by providing the -check option to the application definition:

```

csh> cd ../cmt
csh> vi requirements
...
application FooTest -check FooTest.c
csh> gmake check
Hello Foo

```

6 - Localizing a package - The CMTPATH configuration parameter.

In the next sections, we'll see that packages *reference* each other by means of *use* relationships. **CMT** provides a quite flexible mechanism for *localizing* the referenced packages.

A given version of a given package is always referred to by using a *use* statement within its **requirements** file. This statement should specify the package through some *keys* :

1. its name (such as **Cm**)
2. its version (such as **v7r5**)
3. optionally its supposed location (such as **/lal**)

Given these keys, the referenced package is looked for according to a prioritized search list which is (in decreasing priority order) :

1. the absolute access path, if specified in the *use* statement,
2. the path where the current package (the one *using* the referenced package) is installed,
3. the access paths optionally registered in the configuration parameter **CMTPATH** (and in decreasing priority, the first element being searched for first),
4. and lastly the default root.

The configuration parameter **CMTPATH** can be specified either in the environment variable named **CMTPATH** or in **.cmtrc** files, which can themselves be located either in the current directory, in the *home* directory of the developer or in **\${CMTROOT}/mgr**. In the Windows environment, this configuration parameter must be installed as a *Registry* under either the keys:

- **HKEY_LOCAL_MACHINE/Software/CMT/path**
- **HKEY_CURRENT_USER/Software/CMT/path**

(A graphical tool available in **%CMTROOT%\VisualC\install.exe** permits to interactively modify this list)

If the *path* argument is specified as a relative path (ie. there is no leading **slash** character or it's not a disk), it will be used as an *offset* to each search case. The search is done starting from the list specified in the **CMTPATH** configuration parameter, then using the default root; and the offset is appended at each searched location.

The **CMTPATH** parameter is thus used as a search list for the packages, and the individual paths are separated in this list by *spaces* or by *colons*.

As an example, if we specify the **CMTPATH** parameter as follows :

```
csh> setenv CMTPATH /users/dsksi/arnault/dev:/exp/virgo/projects
```

or (in a **.cmtrc** file)

```
CMPATH=/users/dsksi/arnault/dev:/exp/virgo/projects
```

Then a *use* statement (defined within a given package) containing :

```
...  
use Cm v7r9  
use Cmo v1 Cm  
...
```

(and assuming that the default root is */lal*) would look for the package **Cm** from :

1. the same path as the current package
2. **/users/dsksi/arnault/dev/Cm/v7r9/cmt**
3. **/exp/virgo/projects/Cm/v7r9/cmt**
4. **/lal/Cm/v7r9/cmt**

Whereas the package **Cmo** would be searched from :

1. the directory **Cm** within the same path as the current package,
2. **/users/dsksi/arnault/dev/Cm/Cmo/v1/mgr**
3. **/exp/virgo/projects/Cm/Cmo/v1/mgr**
4. **/lal/Cm/Cmo/v1/mgr**

The packages are searched assuming that the directory hierarchy below the access paths always follow the convention :

1. there is a first directory level named according to the package name,
2. then the next directory level is named according to the version tag,
3. then there is a branch named either **cmt** or **mgr**,
4. lastly there is a *setup* script within this **cmt** or **mgr** branch.

Thus the list of access paths is searched for until these conditions are properly met.

7 - Managing site dependent features - The CMTSITE environment variable.

Software bases managed by **CMT** are often replicated to multiple geographically distant sites (as opposed to machines connected through AFS-like WAN). In this kind of situation, some of the configuration parameters (generally environment variables, used for instance to reference local installations of *external* software) take different values.

The **CMTSITE** environment variable or *registry* in Windows environments, is entirely under the control of the *sitemanager* and can be set up with a value representing the site (typical values may be **LAL**, **Virgo**, **Atlas**, **LHCb**, **CERN**, etc.).

This variable, when set, corresponds to a *tag* (with the same priority as **CMTCONFIG**) which can be used to select different values for make macros or environment variables.

A typical use for this tag is to build up actual values for the **everywhere** macro (which is used to launch remote make actions over a complete site). The following example (extracted from the **requirements** file of **CMT** itself) presents how this special standard macro is filled in with different sets of host names, according to some possible sites:

```
macro everywhere "hosts" \  
  LAL "as7 lx1 virgo-controll1 papoul-fe hp2 aleph hp1" \  
  Cascina "ax7 vm38a rio0a" \  
  LHCB "lxtest hpplus dxplus rsplus" \  
  Atlas "lxtest rsplus atlas dxplus"
```

8 - Configuring a package.

The first ingredient of a proper package configuration is the set of configuration parameters which has to be specified in a textual file named **requirements** and installed in the **cmt** (or **mgr**) branch of the package local tree.

An empty version of this file is automatically created the first time the package is installed, and the package manager is expected to augment it with configuration specifications.

Many configuration parameters are supposed to be described into this **requirements** file (one per package) - see the detailed syntax specifications here - namely :

- the package information about its author(s) and manager(s)
- the relationships with other packages
- the package constituents (libraries, applications, documents, etc.)
- the parameterization of the tools used in the build process (eg. make macros)
- the parameterization of the run-time activity (eg. environment variables, search paths, etc.)

Generally, every such appropriate parameter will be deduced on demand from the **requirements** file(s) through the various queries functions available from the **cmt** main driver. Therefore there is no systematic package re-configuration per se, besides the very first time a package is newly installed in its location (using the **cmt create** action).

Query actions (generally provided using the **cmt setup** or the **cmt show** family of commands) are embedded in the various productivity tools, such as the setup shell scripts, or makefile fragments.

These query actions always imply interpreting the set of **requirements** files obtained from the packages of the *use* chain. Symbols, tags and other definitions are then computed and built up according to inheritance-like mechanisms set up between used packages.

Most typical examples of these query functions are:

- **cmt setup** builds a shell command line for setting up environment variables
- **cmt show macros** construct the effective set of inherited make macros
- **cmt show uses** gives the ordered and flattened set of used packages
- **cmt show constituents** lists the package's constituents
- **cmt show path** lists the effective search path for packages.
- **cmt show strategies** shows the current setup of various functional **CMT** strategies.

9 - Working on a package.

In this section, we'll see, through a quite simple scenario, the typical operations generally needed for installing, defining and building a package. We are continuing the example of the **Foo** package already used in this document.

9.1 - Working on a library.

Let's assume, as a first example, that the **Foo** package is originally composed of one library **libFoo.a** itself made from two sources : **FooA.c** and **FooB.c**. A shared flavour of the library **libFoo.so** or **libFoo.sl** or **libFoo.dll**) is also foreseen.

The minimal set of branches provided by **CMT** (once the **cmt create** operation has been performed) for a package includes **src** for the sources and **cmt** for the *Makefiles* and other scripts.

The various tools **CMT** provide will be fully exploited if one respects the roles these branches have to play. However it is quite often possible to extend the default understanding **CMT** gets on the package by appropriate modifiers.

Assuming the conventional usage is selected, the steps described in this section can be undertaken in order to actually develop a software package.

We first have to create the two source files into the **src** branch (typically using our favourite text editor). Then a description of the expected library (ie. built from these two source files) will be entered into the **requirements** file. The minimal syntax required in our example will be :

```
csh> cd ../cmt
csh> vi requirements          (1)
library Foo FooA.c FooB.c
```

1. the **requirements** file located in the **cmt** branch of the package receives the description of this *library* component. This is done using one **library** statement.

The **cmt create** command did generate a simple *Makefile* (and **NMake** file) containing the following lines which you may extend freely (e.g. to add very special commands that cannot be automatically deduced from the requirements file):

```
include ${CMTROOT}/src/Makefile.header
include ${CMTROOT}/src/constituents.make
```

and

```
include $(CMTROOT)\src\NMakefile.header
include $(CMTROOT)\src\constituents.nmake
```

Generally, this simple makefile is sufficient for all standard operations, since **CMT** continuously and transparently manages the automatic reconstruction of any intermediate makefile fragment. We therefore immediately execute it as follows :

```
...v1/cmt> [g]make
Checking configuration
Rebuilding cmt_path.make
Rebuilding constituents.make
Rebuilding setup.make
Rebuilding alpha.make
alpha.make ok
-----> starting Foo
Library Foo
gmake[1]: Entering directory `/lal/Foo/v1/cmt
Rebuilding Foo_dependencies.make
gmake[1]: Leaving directory `/lal/Foo/v1/cmt
gmake[1]: Entering directory `/lal/Foo/v1/cmt
Now rebuilding ../src/FooA.pp
Now rebuilding ../src/FooB.pp
Foo : Protos ok
cd ../alpha/; cc -c -I../src/ -I../src/ -stdl ../src/FooA.c
cd ../alpha/; cc -c -I../src/ -I../src/ -stdl ../src/FooB.c
cd ../alpha/; ar -clr ../alpha/libFoo.a ../alpha/FooA.o ../alpha/FooB.o
ranlib ../alpha/libFoo.a
cat /dev/null >../alpha/Foo.stamp
cd ../alpha/; /lal/CMT/vlr8/mgr/cmt_make_shlib_common.sh noextract alpha Foo ; cat /dev/null >../alpha/Foo.shstamp
-----> Foo : library ok
-----> Foo ok
gmake[1]: Leaving directory `/lal/Foo/v1/cmt
-----> all ok.
```

or, for nmake:

```
...v1/cmt> nmake /f nmake
```

One sees from this example that :

- The very first time this rebuilding operation occurs, some makefile fragments have automatically been built so as to contain the extended set of Makefile macros deduced from the effective configuration (read from the **requirements** file). These fragments are automatically rebuilt (if needed) each time one of the **requirements** file change.
- The directory which is used for the binaries (i.e. the results of compilation or the libraries) has been automatically created by a generic target which is defined within **[N]Makefile.header**. A new binary directory will be created each time a new value of the **CMTCONFIG** environment variable is defined or a *tag* is provided on the command line to **make**.
- Each component of the package (be it a particular *library* or a particular *executable*) will have its own *makefile* fragment (named **<name>.[n]mak[e]**). This dedicated *makefile* takes care of filling up the library and creating the shared library (on the systems where this is possible - currently **OSF1**, **Linux**, **HP-UX** and **Windows/NT**).
- These dedicated *makefiles* are automatically *executed* from the main one, and the make macro **constituents** can be redefined (e.g. in the **requirements** file) so as to customize the building sequence.

This mechanism relies on some conventional *macros* and incremental *targets* used within the specific makefiles :

9.2 - Working on an application

Assume we now want to add a test program to our development. Then we create a **FooTest.c** source, and generate the associated makefile (specifying that it will be an executable instead of a library) :

```

csh> cd ../src
csh> emacs FooTest.c
...
csh> cd ../cmt
csh> vi requirements
...
application FooTest FooTest.c

```

So that we may simply build the complete stuff by running :

```

> [g]make
Checking configuration
Rebuilding cmt_path.make
Rebuilding constituents.make
Rebuilding setup.make
Rebuilding alpha.make
alpha.make ok
-----> starting FooTest
Application FooTest
gmake[1]: Entering directory `/lal/Foo/v1/cmt'
Rebuilding FooTest_dependencies.make
gmake[1]: Leaving directory `/lal/Foo/v1/cmt'
gmake[1]: Entering directory `/lal/Foo/v1/cmt'
Now rebuilding ../src/FooTest.pp
FooTest : Protos ok
cd ../alpha/; cc -c -I../src/ -I../src/ -stdl      ../src/FooTest.c
cd ../alpha/; cc -o FooTest.exe.new ../alpha/FooTest.o  ; mv -f FooTest.exe.new FooTest.exe
-----> FooTest ok
gmake[1]: Leaving directory `/lal/Foo/v1/cmt'
-----> all ok.

```

Which shows that a program **FooTest.exe** has been built from our sources. Assuming now that this program needs to access the **Foo** library, we'll just add the following definition in the requirements file :

```

...
macro Foo_linkopts " -L$(FOOROOT)/$(Foo_tag) -lFoo " \
    WIN32          " $(FOOROOT)/$(Foo_tag)/Foo.lib "
...

```

The **Foo_linkopts** conventional macro will be automatically inserted within the **use_linkopts** macro.

Like all other make macros used to build a component, the **Foo_linkopts** may be specified within the **makefile** itself or in the requirements file (as in the example above). However, systematically using the requirements file for defining macros gives several benefits:

- variants of the macro definition can be provided
- monitoring features of **CMT** such as the **cmt show macros** command can be used later on
- macros defined this way may be later on inherited by client packages which will *use* our package.

9.3 - Selecting a specific configuration.

A configuration describes the conditions in which the package had been built (ie. compiled and linked). This configuration can depend on :

- the operating system

- the platform
- the choice of the compiler
- options used for compiling (such as optimizer, debugger, etc...) or linking

Carefully describing this configuration is essential both for maintenance operations (so as to remember the precise conditions in which the package was built) and when the development area is *shared* between machines running different operating systems.

CMT relies on several complementary conventions for this description and management.

First of all, the **CMTCONFIG** environment variable is globally defined for any package installed within the **CMT** environment. Its value is by default computed by the **cmt system** command.

Cmt system automatically builds a value characterizing both the machine type and the operating system type (using a mixing of the **uname** standard **UNIX** command with various operating system specific definitions such as the **AFS** based **fs sysname** command). This results in a *tag* used both to *name* the so-called *binary* branch (i.e. the one where binary files are put) and to *select* one of the values for the parameters within the requirements file.

This tag value is therefore filled in automatically by **CMT**. However, it is always possible to override it using the **-tag=<value>** modifier of the **cmt** commands. This is particularly useful in the following situations :

- when using special compiler options (e.g. optimization, debugging, ...)
- for switching to different compilers (e.g. **gcc** versus the native compiler)
- when one uses a special debugging environment such as **Insure** or **Purify**
- when using special system specific features (such as whether one uses thread-safe algorithms or not)

This selection should be possible package by package, using the dedicated copy of the **tag** macro, under the name **Foo_tag** which is assigned by default to the generic value of **tag**.

If one needs to specify a special configuration environment (say we want a set of options tagged by the name **alphadb** to be used in a debug environment), the following actions could be undertaken :

```
## install some definitions in the requirements file :

csh> cd ../cmt
csh> vi requirements

tag    alphadb          alpha debug          #1

private

macro  debug_cflags    " " \                      #2
      debug            "-g"

macro_append cflags    " $(debug_cflags) "          #3

macro  debug_linkflags " " \                      #4
      debug            "-g"

macro  Foo_linkopts    "-L$(FOOROOT)/$(Foo_tag) -lFoo $(debug_linkflags)" #5
```

1. The **alphadb** tag is defined to combine the **alpha** tag and the **debug** tag.
2. The **debug_cflags** macro provides the compiler option to be used in a *debugging* environment.
3. The standard **cflags** macro is incremented by the value of the **debug_cflags** macro.
4. The **debug_linkflags** macro provides the linker options to be used on **alpha** machines when debugging is needed (thus when the tag is **alphadb**).
5. The standard **Foo_linkopts** macro, which provides linker options to build a client application is defined so that the **Foo** library is used and possible support for debugging as well. This is possible since the **Foo_tag** macro is automatically set to be equivalent to the standard **tag** macro.

It is then possible to activate this new **alphadb** tag by running **make** as follows :

```
> [g]make Foo_tag=alphadb
Checking configuration
Rebuilding cmt_path.make
Rebuilding constituents.make
Rebuilding setup.make
Rebuilding alphadb.make
alphadb.make ok
-----> starting Foo
Library Foo
gmake[1]: Entering directory '/lal/Foo/v1/cmt'
Rebuilding Foo_dependencies.make
gmake[1]: Leaving directory '/lal/Foo/v1/cmt'
gmake[1]: Entering directory '/lal/Foo/v1/cmt'
Foo : Protos ok
cd ../alphadb/; cc -c -I../src/ -I../src/ -stdl -g          ../src/FooA.c
cd ../alphadb/; cc -c -I../src/ -I../src/ -stdl -g          ../src/FooB.c
cd ../alphadb/; ar -clr ../alphadb/libFoo.a ../alphadb/FooA.o ../alphadb/FooB.o
ranlib ../alphadb/libFoo.a
cat /dev/null > ../alphadb/Foo.stamp
cd ../alphadb/; /users/dskai/arnault/Packages/CMT/CMT/v1r8/mgr/cmt_make_shlib_common.sh noextract alphadb Foo ; cat /dev/null > ../alphadb/Foo.shstamp
-----> Foo : library ok
-----> Foo ok
gmake[1]: Leaving directory '/lal/Foo/v1/cmt'
-----> starting FooTest
Application FooTest
gmake[1]: Entering directory '/lal/Foo/v1/cmt'
Rebuilding FooTest_dependencies.make
gmake[1]: Leaving directory '/lal/Foo/v1/cmt'
gmake[1]: Entering directory '/lal/Foo/v1/cmt'
FooTest : Protos ok
cd ../alphadb/; cc -c -I../src/ -I../src/ -stdl -g          ../src/FooTest.c
cd ../alphadb/; cc -o FooTest.exe.new ../alphadb/FooTest.o -L/users/dskai/arnault/test/CMT/Foo/v1/alphadb -lFoo -g ; mv -f FooTest.exe.new FooTest.exe
-----> FooTest ok
gmake[1]: Leaving directory '/lal/Foo/v1/cmt'
-----> all ok.
```

- The **alphadb** tag is specified on the **make** command line.
- The tag is used to build a binary branch named **alphadb**
- Reconstruction of both the library **Foo** and the application **FooTest** occur in this binary branch, using the appropriate compiler and linker options.

9.4 - Working on a test or external application

It is also possible to work on a *test* or *external* application, ie. when one does not wish to configure the development for this application using **CMT**. Even in this case, it is possible to benefit from the packages configured using **CMT** by partially using **CMT**, just for *used* relationships.

Here, no special convention is assumed on the location of the sources, the binaries, the management scripts, etc... However, it is possible to describe in a **requirements** file the *use* relationships, as well as the **make** macro definitions, quite similarly to the package entirely configured using **CMT**.

Most of the options provided by the **cmt** user interface are still available in these conditions.

9.5 - Construction of a global environment

A software base generally consists in many *packages*, some of them providing *libraries* or *documents*, others providing *applications*, some providing both, some providing just *glues* towards external software products.

On another view, this software base may a mix of packages shared between several projects and sets of packages specific to various projects. One may have several software bases as well (combined using the **CMPATH** environment variable).

In such contexts, it is often desirable that a given projects defines its own selection of all existing packages. This can easily be done with **CMT** by defining a *project* package, containing only **use** statements towards the appropriate selection of packages for this particular project.

Let's consider as an example the project named **MyProject**. We may create the package named **MyProject** similarly to any other package :

```
csh> cd .....  
csh> cmt create MyProject v1
```

Then the **requirements** file of this new package will simply contain a set of **use** statements, defining the *official* set of validated versions of the packages required for the project. This mechanism also represents the notion of *global release* traditionally addressed in configuration management environments

```
package MyProject  
  
use Cm v7r6  
use Db v4r3  
use El v4r2  
use Su v5  
use DbUI v1r2 Db  
use ElUI v1r1 El  
use VSUI v3 Su/VSU  
use VMM v1  
use VPC v3  
  
setup_script set_path  
cleanup_script reset_path
```

In this example we have also specified that this global environment will provide an additional setup script (found by default in **\${MYPROJECTROOT}/cmt/set_path.csh**) and containing specific shell commands.

Then any user willing to access the so-called *official* release of the package set appropriate to the project **MyProject** will simply do (typically within its login shell script) :

```
# a login script  
  
...  
  
source /MyProjectDevArea/MyProject/v1/cmt/setup.csh.csh
```

Later on, future evolutions of the **MyProject** package will reflect progressive integration steps, which *validate* the evolutions of each referenced package.

10 - Defining a document generator

In a Unix environment, documents are built using **make** (well generally its *gnu* flavour) or **nmake** in Windows environments. The basic mechanism provided in **CMT** relies on *make fragment patterns* containing instructions on how to rebuild document pieces. Many such generators are provided by **CMT** itself so as to take care of the most usual cases (e.g. compilations, link operations, archive manipulations, etc...). In addition to those, any package has to possibility to provide a new generator for its own purpose, i.e. either for providing rules for a special kind of document, or even to override the default ones provided by **CMT**. This mechanism is very similar to the definition or re-definition of *macros* or environment variables in that every new generator has to be first declared in a **requirements** file belonging to a package (**CMT** actually declares its default generators within its **requirements** file), allowing all its client packages to transparently acquire the capacity to generate documents of that sort.

The following sections explain the framework for defining new document types.

10.1 - Writing a fragment pattern for a generator

CMT proposes three types of constituents: on the one hand Applications and Libraries are handled using pre-defined make fragments. On the other hand "Documents" form the most general constituent, and make use of user-defined make fragments.

Each document must be associated with a "document-type" (also named "document-style"), which corresponds to a dedicated make fragment of that name.

The following sections first discuss one simple example (the production of postscript from latex files) available in the standard **CMT** distribution kit, and then present and detail the complete framework.

10.1.1 - An example : the "tex" document-style

This fragment permits to convert latex files into dvi files then into postscript files. The document style is named "tex" and the following fragment named "tex" is installed in **CMT** itself :

```
===== tex =====
${PREFIX}/${NAME}.dvi : ${FULLNAME}
    cd ${doc}; latex ${FULLNAME}

${PREFIX}/${NAME}.ps : ${PREFIX}/${NAME}.dvi
    cd ${doc}; dvips ${PREFIX}/${NAME}.dvi
=====
```

- The fragment is located into **\$CMTROOT/fragments**
- It is (and must be) declared in the **CMT's requirements** file as follows :

```
make_fragment tex -suffix=ps
```

where:

1. "tex" represents both the fragment name and the document style.
 2. the **-suffix=ps** option indicates that output files produced by this document generator will have this suffix.
- A user package's requirements file may contain the following statement:

```
document tex MyDoc ../doc/doc1.tex ../doc/doc2.tex
```

where:

1. The first parameter "tex" is the document-style
2. The second parameter "MyDoc" is used for building the constituent's makefile (under the name MyDoc.make) and for providing the make target "MyDoc".
3. The other parameters (doc1.tex and doc2.tex) are the sources of the document. Explicit location is required (since default is currently defined to be ../src)
4. The constituent's makefile MyDoc.make is built as follows :
 1. Install a copy of the **\$CMTROOT/fragments/make_header** fragment
 2. Install a copy of the **\$CMTROOT/fragments/document_header** fragment
 3. For each of the sources, install a copy of the fragment "tex"
 4. Install a copy of the **\$CMTROOT/fragments/cleanup_header** fragment

The result for our example is:

```
===== MyDoc.make =====

#####
# Document MyDoc
#
# Generated by
#
#####

help ::
    @echo 'MyDoc'

MyDoc_output = $(doc)

MyDoc :: $(MyDoc_output)doc1.ps $(MyDoc_output)doc2.ps
    @/bin/echo MyDoc ok

doc1_dependencies = ../doc/doc1.tex
doc2_dependencies = ../doc/doc2.tex

../doc/doc1.dvi : $(doc)doc1.tex
    cd ${doc}; latex $(doc)doc1.tex

../doc/doc1.ps : ../doc/doc1.dvi
    cd ${doc}; dvips ../doc/doc1.dvi
```

```

../doc/doc2.dvi : $(doc)doc2.tex
                  cd ${doc}; latex $(doc)doc2.tex

../doc/doc2.ps : ../doc/doc2.dvi
                  cd ${doc}; dvips ../doc/doc2.dvi
clean :: MyDocclean
          @cd .

MyDocclean ::
=====

```

10.1.2 - How to create and install a new document style

- Select a name for the document style. It should not clash with existing ones (use the `cmt show fragments` for a complete list of document types currently defined).
- A fragment exactly named after the document style name must be installed into a subdirectory named "fragments" below the `cmt/mgr` branch of a given package (which becomes the "provider" package). It should be noticed that previous versions of CMT used to look after this fragments directory at the same level as the `mgr/cmt` branch (instead of inside it). However, for backward compatibility reasons, the former mechanism is still supported and will eventually become obsolete in a future release.
- It must be declared in the requirements file of the provider package as follows:

```
make_fragment [ options... ]
```

where options may be :

<code>-suffix=<suffix></code>	provide the suffix of the output files (without the dot)
<code>-header=<header></code>	provide another make fragment, located close to the base fragment, and meant to be installed at the beginning of the constituent's make fragment. A similar declaration is also required for this fragment
<code>-trailer=<trailer></code>	provide another make fragment, located close to the base fragment, and meant to be installed at the end of the constituent's make fragment. A similar declaration is also required for this fragment
<code>-dependencies</code>	install the automatic generation of dependencies into the constituent's make fragment

Once a fragment is installed and declared, it may be used by any client package, re-defined by any client package (using the same declaration mechanism), and it may be queried upon using the command

```
> cmt show fragment
```

which will show where this fragment is defined (ie. in which of the used packages).

The "cmt show fragments" commands lists all declared fragments.

For building a fragment, one may use pre-defined generic "templates" (which will be substituted when a fragment is copied into the final constituent's makefile).

CONSTITUENT	the constituent name
CONSTITUENTSUFFIX	an optional constituent's output suffix
FULLNAME	the full source path name (including directory and suffix)
FILENAME	the complete source file name (only including the suffix)
NAME	the short source file name (without directory and suffix)
PREFIX	the source directory
SUFFIX	the suffix provided in the -suffix option
OBJS	(only available in headers) the list of outputs, formed by a set of expressions : $$(\${CONSTITUENT}_output)\${NAME}\${SUFFIX}$

Templates must be enclosed between $\${$ and $}$ and will be substituted at the generation time. Thus, if a fragment contains the following text :

$$$(\${CONSTITUENT}_output)\${NAME}\${SUFFIX}$$

then, the expanded constituent's makefile will contain (referring to the "tex" example)

$$$(MyDoc_output)doc1.ps$$

Which shows that make macros may be dynamically generated.

10.1.3 - Some remarks

This scheme is rather poorly experimented yet. Only few users have created such document styles (see section below). Thus elements of the mechanism, of the conventions (eg. the choice of the templates) is likely to benefit from improvements. Users are kindly invited to contribute to the design of this scheme.

On another hand, it might be useful to provide as contributions, fragments for general purpose document styles. Any idea for a scheme for supporting such contributions is welcome.

10.1.4 - Examples

1. rootcint

It generates C++ hubs for the Cint interpreter in Root.

```
===== rootcint =====
$(src)${NAME}.cc :: ${FULLNAME}
    ${rootcint} -f $(src)${NAME}.cc -c ${FULLNAME}
=====
```

2. agetocxx and agetocxx_header.

It generates C++ source files (xxx.g files) from Atlas' AGE description files.

```
===== agetocxx =====
output=$(CONSTITUENT)_output)

$(output)${NAME}.cxx : ${NAME}_cxx_dependencies
    (echo '#line 1 "${FULLNAME}"'; cat ${FULLNAME}) > /tmp/${NAME}.gh.c
    gcc -E -I$(output) $(use_includes) -D_GNU_SOURCE \
        -c /tmp/${NAME}.gh.c > /tmp/${NAME}.gh
    cd $(output); $(agetocxx) -o ${NAME} -ohd ${PREFIX} -ohp ${PREFIX} /tmp/${NAME}.gh
    rm -f /tmp/${NAME}.gh /tmp/${NAME}.gh.c
    cd $(bin); $(cppcomp) $(use_cppflags) ${CONSTITUENT}_cppflags ${NAME}_cppflags ${ADDINCLUDE} $(output)${NAME}.cxx
    cd $(bin); $(ar) ${CONSTITUENT}lib ${NAME}.o; /bin/rm -f ${NAME}.o
=====

===== agetocxx_header =====
${CONSTITUENT}lib      = $(bin)lib${CONSTITUENT}.a
${CONSTITUENT}stamp    = $(bin)${CONSTITUENT}.stamp
${CONSTITUENT}shstamp  = $(bin)${CONSTITUENT}.shstamp

${CONSTITUENT} :: dirs ${CONSTITUENT}LIB
    @/bin/echo ${CONSTITUENT} ok

${CONSTITUENT}LIB :: ${CONSTITUENT}lib ${CONSTITUENT}shstamp
    @/bin/echo ${CONSTITUENT} : library ok

${CONSTITUENT}lib ${CONSTITUENT}stamp :: ${OBSJ}
    $(ranlib) ${CONSTITUENT}lib
    cat /dev/null >${CONSTITUENT}stamp

${CONSTITUENT}shstamp :: ${CONSTITUENT}stamp
    cd $(bin); $(make_shlib) $(tag) ${CONSTITUENT} ${CONSTITUENT}shlibflags; cat /dev/null >${CONSTITUENT}shstamp
=====
```

It must be declared as follows :

```
make_fragment agetocxx_header
make_fragment agetocxx -suffix=cxx -dependencies -header=agetocxx_header
```

11 - The tools provided by CMT

The set of conventions and tools provided by CMT is mainly composed of :

- the syntax of the **requirements** file,
- and the general **cmt** user interface, available in the **mgr** branches of the **CMT** package.

The *setup* script found in the **CMT** installation directory actually adds its location to the definition of the standard **UNIX PATH** environment variable in order to give direct access to the various **CMT** tools, such as the main **cmt** user interface.

The sections below will detail the complete syntax of the **requirements** file since it is the basis of most information required to run the tools as well as the main commands available through the **cmt** user interface.

11.1 - The requirements file

11.1.1 - The general requirements syntax

- A requirements file is made of statements, each describing one named configuration parameter.

Statements generally occupy one single line, but may be split into several lines using the reverse-slash character (in this case the reverse-slash character *must* be the last character on the line or only followed by space characters).

Each statement is composed of words.

The first word of a statement is the type of the configuration parameter.

The rest of the statement provides the value assigned to the configuration parameter.

- Words composing a statement are separated with space or tab characters. They may also be enclosed in quotes when they have to include space or tab characters. Single or double quotes may be freely used, as long as the same type of quote is used on both sides of the word.

Special characters (tabs, carriage-return and line-feed) may be inserted into the statements using an XML-based convention:

tabulation	<cmt:tab/>
carriage-return	<cmt:cr/>
line-feed	<cmt:lf/>

- Comments : they start with the # character and extend up to the of the current line.
-

11.1.2 - The complete requirements syntax

11.2 - The concepts handled in the requirements file

11.2.1 - Meta-information : author, manager

The author and manager names

11.2.2 - package, version

The package name and version. These statements are purely informational.

11.2.3 - Constituents : application, library, document

Describe the composition of a constituent. Application and library correspond to the standard meaning of an application (an executable) and a library, while document provides for a quite generic and open mechanism for describing any type of document that can be generated from sources.

Applications and libraries are assigned a name (which will correspond to a generated make fragment, and a dedicated make target).

A document is first associated with a document type (which must correspond to a previously declared make fragment). The document name is then used to name a dedicated make fragment and a make target.

Various options can be used when declaring a constituent:

<i>option</i>	<i>validity</i>	<i>usage</i>
-windows	applications	When used in a Windows environment, generates a GUI-based application (rather than a console application)
-no_share	libraries	do not generate the shared library
-no_static	libraries	do not generate the static library
-prototypes	applications, libraries	do generate the prototype header files
-no_prototypes	applications, libraries	do not generate the prototype header files
-check	applications	generate a check target meant to execute the rebuilt application
-group=group-name	any	install the constituent within this group target
-suffix=suffix	applications, libraries	provide a suffix to names of all object files generated for this constituent (see 1 below)
-import=package	applications, libraries	explicitly import for this constituent the standard macros from a package that has the -no_auto_imports option set
variable-name=variable-value	any	define a variable and its value to be given to the make fragment (see 2 below)

1. When several constituents need to share source files, (a typical example is for building different libraries from the same sources but with different compiler options), it is possible to specify an optional output suffix with the **-suffix=<suffix>** option. With this option, every object file name will be automatically suffixed by the character string

"<suffix>", avoiding name conflicts between the different targets, as in the following example:

```
library AXt -suffix=Xt *.cxx
library AXaw -suffix=Xaw *.cxx
```

2. It's possible to specify in the list of parameters one or more pairs of **variable-name=variable-value** (without any space characters around the "=" character), such as in the next example:

```
make_fragment doc_to_html (1)
document doc_to_html Foo output=FooA.html FooA.doc (2) (3)
```

1. This makefile fragment is meant to contain some text conversion actions and defines a document type named **doc_to_html**.
2. This constituent exploits the document type **doc_to_html** to convert the source **FooA.doc** into an html file.
3. The user defined template variable named **output** is specified and assigned the value **FooA.html**. If the fragment **doc_to_html** contains the string **\${output}**, then it will be substituted to this value.

11.2.4 - Groups

Groups permit to organize constituents that must be built at same phases or with similar constraints.

Each group is associated with a make target (of the same name) which, when used in the make command, selectively rebuilds all constituents of this group.

The default group (into which all constituents are installed by default) is named **all**, therefore, running make without argument, activates the default target (ie. **all**).

As a typical usage of this mechanism, one may detail the case in which one or several constituents are making use of one special facility (such as a database service, real-time features, graphical libraries) and therefore might require a controlled re-build. This is especially useful for having these constituents only rebuilt on demand rather than rebuilt automatically when the default make command is run.

One could, for instance specify within the requirements file :

```
other constituents without group specification...

library Foo-objy -group=objy <sources making use of Objectivity>

application FooGUI -group=graphics <sources making use of Qt>
```

(Beware of the position of the -group option which must be located after the constituent name. Any other position will be misunderstood by CMT)

Then, running **gmake all** would only rebuild the un-grouped constituents, whereas running

```
> gmake objy
> gmake graphics
```

would rebuild objy related or graphics related constituents.

11.2.5 - Languages

Some computer languages are known by default by **CMT** (**C**, **C++**, **Fortran77**, **Java**, **lex**, **yacc**). However it is possible to extend this knowledge to any other language.

We consider here languages that are able to produce object files from sources.

Let's take an example. We would like to install support for Fortran90. We first have to *declare* this new language support to **CMT** within the **requirements** file of one of our packages (Notice that it's not at all required to modify **CMT** itself since all clients of the selected package will inherit the knowledge of this language).

The language is declared by the following statement:

```
language fortran90 -suffix=f90 -suffix=F90 -preprocessor_command=$(ppcmd) -linker=$(f90link)
```

- The recognized suffixes for source files will be **f90** and **F90**
- The linker command used to build a Fortran90 application is described inside the macro named **f90link**
- The language being named **fortran90**, two make fragments are expected, one under the name **fortran90** (for building modules not meant to be archived), the other with the name **fortran90_library** (for modules meant to be archived).

These two fragment should be installed in the **fragments** directory inside the cmt/mgr branch of our package.

Due to the similarity of the example to fortran77, we may easily provide the expected fragments simply by copying the f77 fragments found in **CMT** (thus the fragments **\${CMTROOT}/fragments/fortran** and **\${CMTROOT}/fragments/fortran_library**

These fragments make use of the **fcomp** macro, which holds the fortran77 compiler command (through the **for** macro).

```
macro for          "f77" \  
...  
macro fcomp        "$(for) -c $(fincludes) $(fflags) $(pp_fflags)"
```

We therefore simply replace these macros by new macros named **f90comp** and **f90**, defined as follows:

```
macro f90          "f90" \  
...  
macro f90comp      "$(f90) -c $(fincludes) $(fflags) $(pp_fflags)"
```

Some languages (this has been seen for example in the IDL generators in Corba environments) do provide several object files from one unique source file. It is possible to specify this feature through the (repetitive) **-extra_output_suffix** option like in:

```
language idl -suffix=idl -fragment=idl -extra_output_suffix=_skel
```

where, in this case, two object files are produced for each IDL source file, one named **<name>.o** the other named **<name>_skel.o**.

11.2.6 - Symbols : alias, set, set_append, set_prepend, set_remove, macro, macro_append, macro_prepend, macro_remove, macro_remove_all, path, path_append, path_prepend, path_remove

The **alias** keyword is translated into a shell alias definition,

The **set** keyword is translated into an environment variable definition.

The **macro** keyword is translated into a **make**'s macro definition.

The **path** keyword is translated into a *path*-like environment variable, which is supposed to be composed of search paths separated with colon characters (':').

Variants of these keywords are also provided for modifying already defined symbols. This generally happens when a package needs to modify an inherited symbol (ie. which has been already defined by a used package). Through these keywords (**set_append**, **set_prepend**, **set_remove**, **macro_append**, **macro_prepend**, **macro_remove**, **macro_remove_all**, **path_append**, **path_prepend**, **path_remove**) one can append or prepend a text to the existing symbol value or remove a part from it. The **path_remove** keyword removes one of the search paths as soon as it contains the specified value.

The translations occur while running either the setup scripts (for alias, set or path) or the make command (for macro).

All these definitions follow the same pattern:

```
symbol-type symbol-name default-value [ tag value ... ]
```

The symbol-name identifies the symbol for modification operations. The default-value is optionally followed by a set of tag/value pairs, each representing an alternate value for this symbol.

The tag is used to select one alternate value to replace the default value, when one of the following condition is met:

- It matches the value of the CMTSITE environment variable (or registry)
- It matches the value provided by the uname Unix command (when available)
- It matches the value of the CMTCONFIG environment variable (or registry)
- It matches the value specified in the -tag=*tag* argument to the cmt command.
- It matches one user defined tag (see the tag keyword) which itself is associated with a matching tag (Note that this is a recursive definition).

Examples of such definition are :

```
package CMT

set CMTCC "cc" \
    HP-UX      "cc -Aa +z -D_HPUX_SOURCE"

public

macro cflags          "-g" \
    HP-UX             "-g -Aa +z -D_HPUX_SOURCE" \
    hp700_ux101       "-g -fpic -ansi" \
    alpha             "-g -std1" \
    alphas             "-g -std1 -DCTHREADS" \
    insure            "-g -Zuse -std1" \
    AIX               "-g -D_ALL_SOURCE -D_BSD"

macro cppflags        "-g" \
    HP-UX             "-g -Aa +z" \
    hp700_ux101       "-g -fpic"

macro fflags          "-g"

macro src              "../src/"
macro inc              "../src/"
macro mgr              "../cmt/"

macro SHELL            "/bin/sh"
```

11.2.7 - use

Describe the relationships with other packages; the generic syntax is :

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

Omitting the version specification means that the most recent version (ie. the one with highest ids) that can be found from the search path list will be automatically selected.

The root specification can be relative (ie. on Unix it does contain a leading '/' character). In this case, this is prefix systematically considered when the package is looked for in the search path list. But it can also be absolute (ie. with a leading '/' character on Unix), in which case this path takes precedence over the standard search path list (see CMTPATH).

Examples of such relationships are :

```
# Packages installed in the default root :
use OnX v5r2
use CSet v2r3
use Gb v2r1

# A package installed in a root one step below the root :
use CS v3r1 virgo

# Back to the default root :
use Cm v7r3

# Get the most recent version of CERNLIB
use CERNLIB
```

By default, a set of standard macros, which are expected to be specified by used packages, is automatically imported from them (see the [detailed list](#) of these macros). This automatic feature can be discarded using the

-no_auto_imports option to the use statement, or re-activated using the **-auto_imports**. When it is discarded, the macros will not be transparently inherited, but rather, each individual constituent willing to make use of them will have to explicitly import them using the **-import=<package>** [option](#).

11.2.8 - pattern, apply_pattern, ignore_pattern

Often, similar configuration items are needed over a set of packages (sometimes over all packages of a project). This reflects either similarities between packages or generic conventions established by a project or a team.

Typical examples are the definition of the search path for shared libraries (through the **LD_LIBRARY_PATH** environment variable), the systematic production of test applications, etc.

The concept of pattern proposed here implements this genericity. Patterns may be either *global*, in which case they will be systematically applied onto every package, or *local* (the default) in which case they will be applied on demand only by each package.

The general principle of a pattern is to associate a templated (set of) **cmt** statement(s) with the pattern name. Then every time the pattern is applied, its associated statements are applied as if they were directly specified in the requirements file, replacing the template with its current value. If several statement are to be associated with a given pattern, they will be separated with the " ; " separator pattern (beware of really enclosing the ";" between two space characters).

Pattern templates are names enclosed between the '<' and '>' characters. A set of predefined templates are automatically provided by **CMT**:

package	the name of the current package
PACKAGE	the name of the current package in upper case
version	the version tag of the current package
path	the access path of the current package

Then, in addition, user defined templates can be installed within the pattern definitions. Their actual values will be provided as arguments to the `apply_pattern` statement.

User defined patterns that are not valued when applied are simply ignored.

Some examples:

1. Changing the standard include search path.

The standard include path is set by default to `${<package>_root}/src`. However, often projects need to override this default convention, and typical example is to set it to a branch named with the package name. This convention is easily applied by defining a pattern which will apply the `include_path` statement as follows:

```
pattern -global include_path include_path ${<package>_root}/<package>/
```

For instance, a package named **PackA** will expand this pattern as follows:

```
include_path ${PackA_root}/PackA/
```

2. Providing a value to the **LD_LIBRARY_PATH** environment variable

On some operating systems (eg. Linux), shared library paths must be explicited, through an environment variable. The following pattern can automate this operation:

```
pattern ld_library_path \  
  path_remove LD_LIBRARY_PATH "/"<package>/" ; \  
  path_append LD_LIBRARY_PATH ${<PACKAGE>ROOT}/${CMTCONFIG}
```


In this example, the pattern was not set global, so that only packages actually providing shared libraries would be concerned. These packages will simply have to apply the pattern as follows:

```
apply_pattern ld_library_path
```

The schema installed by this pattern provides first a cleanup of the **LD_LIBRARY_PATH** environment variable and then the new assignment. This choice is useful in this case to avoid conflicting definitions from two different versions of the same package.

3. Installing a systematic test application in all packages

Quality assurance requirements might specify that every package should provide a test program. One way to enforce this is to build a global pattern declaring this application. Then every make command would naturally ensure its actual presence.

```
pattern quality_test application <package>test <package>test.cxx <other_sources>
```

In this example, an additional pattern (<other_sources>) permits the package to specify extra source files to the test application (the pattern assumes at least one source file <package>test.cxx).

11.2.9 - branches

Describe the specific directory branches to be added while configuring the package.

```
branches <branch-name> ...
```

These branches will be created (if needed) at the same level as the **cmt** branch. Typical examples of such required branches may be **include**, **test** or **data**.

11.2.10 - build_strategy, version_strategy

CMT opens its behaviours to user control through a set of strategy specifications. The current implementation only provides such control over two mechanisms :

- the way version tags are interpreted and compared to each other.

The following keywords are available:

best_fit	This is the default behaviour. Version tags truly consider major ids, minor ids and patch ids with their complete backward compatibility semantics
best_fit_no_check	Same as best_fit except that different major ids are not seen as incompatible. The greatest id (for major, minor and patch ids) is always selected
first_choice	The first version tag specified in the use chain is selected
last_choice	The last version tag specified in the use chain is selected
keep_all	Internal use only : all referenced versions are kept

- the way makefile fragments for applications and libraries are generated.

Currently this only concerns the automatic generation of prototype header files for C source files. Thus only one keyword is possible : prototypes (and his opposite no_prototypes), the default **CMT** behaviour being to generate prototype headers.

11.2.11 - setup_script, cleanup_script

Specify user defined configuration scripts, which will be activated together with the execution of the main **setup** and **cleanup** scripts.

The script names may be specified without any access path specification, in this case, they are looked for in the **cmt** or **mgr** branch of the package itself. They may also be specified without any **.csh** or **.sh** suffix, the appropriate suffix will be appended accordingly when needed. Therefore, when such a user configuration script is specified, **CMT** expects that the corresponding shell scripts actually exist in the appropriate directory (the **cmt** branch by default) and is provided in whatever format is appropriate (thus suffixed by **.csh** and/or **.sh**).

11.2.12 - include_path

Override the specification for the default include search path, which is internally set to `${<package>_root}/src`.

11.2.13 - include_dirs

Add specifications for non-standard include access paths.

11.2.14 - make_fragment

This statement specifies a specialized makefile fragment, used as a building brick to construct the final makefile fragment dedicated to build the constituents.

There are basically three categories of such fragments : some are provided by **CMT** itself (they correspond to its internal behaviour), others handle the languages and the last serve as specialized document generators.

The fragments defined in **CMT** are used to construct the application or library constituents, since the corresponding semantic is assumed to be standardized.

Language fragments should provide two forms, one for the applications (in which case they are named exactly after the language name) and the other for the libraries (in which case they have the **_library** suffix). A set of language definitions (C, C++, Fortran, Java, Lex, Yacc) is provided by CMT itself but it is expected that projects add new languages according to their needs.

The user defined fragments define by themselves a document-type.

All make fragments are provided as a file which must be located in the **fragments** directory inside the **cmt/mgr** branch of one package. **CMT** itself has such a branch, where all default implementations of make fragments are originally found. All make fragment are implemented as files with exactly the same name as declared in the requirements file. Conversely, a make fragment file provided in the **cmt/fragments** directory of a package must be declared in the requirements file so as to be visible.

A package declaring, and implementing a make fragment may override a fragment of the same name when it is already declared by a used package. This implies in particular that any package may override any make fragment provided by **CMT** itself.

Makefile fragments may take any form convenient to the document style, and some special pre-built templates (see the [appendix](#)) can be used in their body to represent running values, meant to be properly expanded at actual generation time :

CONSTITUENT	the constituent name
FULLNAME	the full source path
FILENAME	the source file name without its path
NAME	the source file name without its path and suffix
FILESUFFIX	the dotted file suffix
FILEPATH	the output path
SUFFIX	the default suffix for output files

11.2.15 - public, private

Introduce a section for *public* or *private* symbols (meant to be implemented as environment variables or aliases in a **Unix** environment or as *logical names* or *symbols* in a **VMS** one). *Macros* to be used within makefiles can also be defined at this level. Public symbols are meant to be exported to any external user of the package whereas private ones are only defined for the package *developer*. Currently the selection between these two categories is done when the setup script is executed : if it is executed while actually being in the **cmt** branch of the package, the developer category is assumed. If the script is executed from another directory the user category is assumed.

11.2.16 - tag

Provide tag definitions.

A tag is a token which can be used to select particular values of symbols. Generally a tag need not being explicitly declared, since the reference to it will declare the tag automatically. However, tags may be used to *name* a particular association of several other tags. In this case, this association must be declared within a requirements file as follows :

```
tag <association-tag-name> <tag1> <tag2> ...
```

11.3 - The general cmt user interface

This utility (a shell script combined with a C application) provides a centralised access to various commands to the CMT system. The first way to use **cmt** is to run it without argument, this will print a minimal help text showing the basic commands and their syntax :

```
> cmt command [option...]
command :
  broadcast [-select=list] [-exclude=list] [-local] [-depth=n] [-all_packages] <command> : apply a command to [some of] the used packages
  build <key> : build various components :
    constituent_makefile : generate Makefile
    constituents_makefile : generate constituents.make
    dependencies : generate dependencies
    library_links : build symbolic links towards all imported libraries
    make_setup : build a compiled version of setup scripts
    msdev : generate MSDEV files
    os9_makefile : generate Makefile for OS9
    prototype : generate prototype file
    readme : generate README.html
    tag_makefile : generate tag specific Makefile

  check <key> : perform various checks
    configuration : check configuration
    files <old> <new> : compare two files and overrides <old> by <new> if different
    version <name> : check if a name follows a version tag syntax
  check_files <old> <new> : compare two files and overrides <old> by <new> if different
  checkout : perform a cvs checkout over a CMT package
  co : perform a cvs checkout over a CMT package
  cleanup [-csh|-sh|-bat] : generate a cleanup script
  config : generate setup and cleanup scripts
  create <package> <version> [<path>] : create and configure a new package
  filter <in> <out> : filter a file against CMT macros and env. variables
  help : display this help
  lock : lock the current package
  lock <package> <version> [<path>] : lock a package
  remove <package> <version> [<path>] : remove a version of a package
  remove library_links : remove symbolic links towards all imported libraries
  run <command> : apply a command
  setup [-csh|-sh|-bat] : generate a setup script
  show <key> : display various infos on :
    author : package author
    branches : added branches
    clients : package clients
    constituent_names : constituent names
    constituents : constituent definitions
    uses : the use tree
    fragment <name> : one fragment definition
    fragments : fragment definitions
    groups : group definitions
    languages : language definitions
    macro <name> : a formatted macro definition
    macro_value <name> : a raw macro definition
    macros : all macro definitions
    manager : package manager
    packages : packages reachable from the current context
    path : the package search list
    pattern <name> : the pattern definition and usages
    patterns : the pattern definitions
    pwd : filtered current directory
    set_value <name> : a raw set definition
    set <name> : a formatted set definition
    sets : set definitions
    strategies : all strategies (build & version)
    tags : all defined tags
    uses : used packages
    version : version of the current package
    versions <name> : visible versions of the selected package

  system : display the system tag
  unlock : unlock the current package
  unlock <package> <version> [<path>] : unlock a package
  version : version of CMT

  cvstags <module> : display the CVS tags for a module
  cvsbranches <module> : display the subdirectories for a module
```

```

cvssubpackage <module> : display the subpackages for a module
global option :
-quiet                : don't print errors
-use=<p>:<v>:<path>      : set package version path
-pack=<package>        : set package
-version=<version>      : set version
-path=<path>           : set root path
-f=<requirement-file>   : set input file
-e=<statement>         : add a one line statement
-tag=<tag-name>        : select specific tag

```

The following sections present the detail of each available command.

11.3.1 - cmt broadcast [-select=list] [-exclude=list] [-local] [-depth=<n>] [-all_packages] <shell command>

This command tries to repeatedly execute a shell command in the context of each of the used package of the current package. The used packages are listed using the **cmt show uses** command, which also indicates in which order the broadcast is performed. When the **all_packages** option, the set of packages reached by the broadcast is rather the same as the one shown by the **cmt show packages** command, ie all **CMT** packages and versions available through the current **CMPATH** list.

Typical uses of this *broadcast* operation could be :

```

csh> cmt broadcast cmt config
csh> cmt broadcast - gmake
csh> cmt broadcast '(cd ../; cvs -n update)'

```

The loop over used packages will stop at the first error occurrence in the application of the command, except if the command was preceded by a '-' (minus) sign (similarly to the make convention).

It is possible to specify a list of selection or exclusion criteria set onto the package path, using the following options, right after the **broadcast** keyword:

```

> cmt broadcast -select=Cm gmake                (1)
> cmt broadcast -select='/Cm/ /CSet/' gmake      (2)
> cmt broadcast -select=Cm -exclude=Cmo gmake    (3)
> cmt broadcast -local gmake                     (4)
> cmt broadcast -depth=<n> gmake                  (5)
> cmt broadcast -all_packages gmake              (6)

```

According to the option, the loop will only operate onto:

1. the packages which path contains the string "**Cm**"
2. the packages which path contains either the string **"/Cm/"** or the string **"/CSet/"**
3. the packages which path contains the string "**Cm**", but which does not contain the string **"Cmo"**
4. the packages at the same level as the current package
5. the packages at the same level as the current package or among the <n> first entries in the **CMPATH** list
6. all the packages and versions currently available through the **CMPATH** list

11.3.2 - cmt build <option>

All build commands are generally meant to change the current package (some file or set of files is generated). Therefore a check against conflicting locks (ie. a lock owned by another user) is performed by all these commands prior to execute it.

- **[-nmake] constituent_makefile <constituent-name>**

This command is internally used by **CMT** in the standard Makefile.header fragment. It generates a specific makefile fragment (named <constituent-name>.make) which is used to re-build this fragment.

All such constituent fragments are automatically included from the main Makefile.

Although this command is meant to be used internally (and transparently) by **CMT** when the make command is run, a developer may find useful in very rare cases to manually re-generate the constituent fragment, using this command.

The **-nmake** option (which must precede the command) provides exactly the same features but in a Windows/nmake context. In this case, all generated makefiles are suffixed by **.nmake** instead of **.make** for Unix environments. The main makefile is expected to be named **NMake** and the standard header is named **NMakefile.header**

- **[-nmake] constituents_makefile**

This command is internally (and transparently) used by **CMT** in the standard Makefile.header fragment, and when the make command is run, to generate a specialized make fragment containing all "cmt build constituent_makefile" commands for a given package.

The **-nmake** option (which must precede the command) provides exactly the same feature but in a Windows/nmake context. In this case, all generated makefiles are suffixed by **.nmake** instead of **.make** for Unix environments. The main makefile is expected to be named **NMake** and the standard header is named **NMakefile.header**

- **dependencies**

This command is internally (and transparently) used by **CMT** from the constituent specific fragment, and when the make command is run, to generate a fragment containing the dependencies required by a source file.

This fragment contains a set of macro definitions (one per constituent source file), each containing the set of found dependencies.

- **library_links**

This command builds a local symbolic link towards all exported libraries from the used packages. A package exports its libraries through the <package>_libraries macro which should contain the list of constituent names corresponding to libraries that must be exported.

```

library Foo ...
library Foo-utils ...
...
macro Foo_libraries "Foo Foo-utils"
...

```

The corresponding **cmt remove library_links** command will remove all these links.

- **make_setup**

This command is internally (and transparently) used by **CMT** from the standard **Makefile**.header fragment, and when the make command is run, to generate another fragment containing all platform (or tag) specific macro definitions.

One copy of this fragment (named <tag>.make) is created per flavour of tag used at build time. The tag considered in this operation is either the default tag value (obtained from the CMTCONFIG environment variable) or specified to the make command using the -tag=<tag> option)

This tag specific fragment represents the actual context that was considered at the most recent make activation. It is automatically rebuilt when one of the used requirements is modified.

- **msdev**

This command generates workspace (.dsw) and project (.dsp) files required for the MSDev tool.

- **os9_makefile**

This command generates external dedicated *makefile* fragments for each individual component of the package (ie. libraries or executable applications) to be used in OS9 context. It generates specific syntaxes for the **OS9** operating systems.

The output of this tool is a set of files (named with the components' name and suffixed by **.os9make**) that are meant to be *included* within the main **Makefile** that the developer has to write anyhow.

The syntax of the **cmt build os9_makefile** utility is as follows :

```
> cmt build os9_makefile <package>
```

- **prototype <source-file-name>**

This command is internally (and transparently) used by **CMT** from the constituent specific fragment, and when the make command is run, to generate prototype header files from C source files.

The prototype header files (named <file-name>.ph) will contain prototype definitions for every global entry point defined in the corresponding C source file.

The effective activation of this feature is controlled by the build strategy of **CMT**. The build strategy may be freely and globally overridden from any requirements file, using the **build_strategy** cmt statement, providing either the "prototypes" or the "no_prototypes" values.

In addition, any constituent may locally override this strategy using the "-prototypes" or "-no_prototypes" modifiers.

- **readme**

This command generates a README.html file into the cmt branch of the referenced package. This html file will include

- a table containing URLs to equivalent pages for all used packages,
- a copy of the local README file (if it exists).

- **tag_makefile**

This command produces onto the standard output, the exhaustive list of all macros controlled by **CMT**, ie. those defined in the requirements files as well as the standard macros internally built by **CMT**, taking into account all used packages.

11.3.3 - cmt check configuration

This command reads the hierarchy of requirements files referenced by a package, check them, and signals syntax errors, version conflicts or other configuration problems.

An empty output means that everything is fine.

11.3.4 - cmt check files <reference-file> <new-file>

This command compares the reference file to the new file, and substitutes the reference file by the new one if they are different.

If substitution is performed, a copy (with additional extension **sav**) is produced.

11.3.5 - cmt checkout ...

See the paragraph on how to use cvs together with **CMT**, and more specifically the details on checkout oprations.

11.3.6 - cmt co ...

This is simply a short cut to the **cmt checkout** command.

11.3.7 - cmt cleanup [-csh|-sh]

This command generates (to the standard output) a set of shell commands (either for csh or sh shell families) meant to unset all environment variables specified in the requirements files of the used packages.

This command is internally used in the cleanup.[c]sh shell script, itself generated by the **cmt config** command.

11.3.8 - cmt config

This command (re-)generates the setup scripts and the minimal Makefile (when it does not exist yet or have been lost).

```
csh> cd ~/Packages/Foo/v1/cmt
csh> cmt config
```

To be properly operated, one must *already* be in the **cmt** or **mgr** branch of the package (where the requirements file can be found).

The situations in which it is useful to run this command are:

- When the setup or cleanup scripts have been lost
 - When the minimal Makefile have been lost
 - When the version of **CMT** is changed
 - After restoring a package from CVS
 - After having manually moved a package (using a manual copy operation)
-

11.3.9 - cmt create <package> <version> [<area>]

This command creates a new package or a new version of a package

```
csh> cmt config Foo v1
```

or:

```
csh> cmt config Foo v1 ~/dev
```

In the first mode (ie. without the *area* argument) the package will be created in the default path.

The second mode explicitly provides an alternate path.

A minimal configuration is installed for this new package:

- An **src** and an **cmt** branch
- A very minimal requirements file

- The setup and cleanup shell scripts
- The minimal Makefile

11.3.10 - cmt filter <in-file> <out-file>

This command reads <in-file>, substitutes all occurrences of macro references (taking either the form `$(macro-name)` or `${macro-name}`) by values deduced from corresponding macro specifications found in the requirements files, and writes the result into <out-file>.

This mechanism is widely internally used by **CMT**, especially for instantiating make fragments. Then, users may use it for any kind of document, including manual generation of MSDev configuration files, etc...

11.3.11 - cmt help

This command shows the list of options of the **cmt** driver.

11.3.12 - cmt lock

cmt lock [<package> <version> [<area>]]

This command tries to set a lock onto the current package (or onto the specified package). This consists in the following operations:

1. Check if a conflicting lock is already set onto this package (ie. a lock owned by another user).
2. If not, then install a small text file named **lock.cmt** into the **cmt/mgr** branch of the package, containing the following text:

```
locked by <user-name> date <now>
```

3. Run a shell command described in the macro named **lock_command** meant to install physical locks onto all files for this version of this package. A typical definition for this macro could be:

```
macro lock_command    "chmod -R a-w ../*" \
WIN32                "attrib /S /D +R ../*"
```

11.3.13 - cmt remove <package> <version> [<area>]

This command removes one version of the specified package. If the package does not contain a conflicting lock, and if the user is granted enough access rights to remove files, *all* files below the version directory will be definitively removed. Therefore this command should be used with as much care as possible.

The arguments needed to reach the package version to be removed are the same as the ones which had been used to create it.

If the removed version is the last version of this package, (and only if its directory is really empty) the package directory itself will be deleted.

11.3.14 - cmt remove library_links

This command removes symbolic links towards all imported libraries which had been installed using the **cmt build library_links** command. This command is generally transparently executed when one runs **gmake clean**

11.3.15 - cmt run shell-command

This command runs any shell command, in the context of the current package.

This may not appear to be very useful for the current package one is working at, but when combined with global options `-pack=package`, `-version=version`, `-path=access-path`, this gives a direct access to any package context.

11.3.16 - cmt setup [-csh|-sh|-bat]

This command generates (to the standard output) a set of shell commands (either for csh, sh or bat shell families) meant to set all environment variables specified in the requirements files of the used packages.

This command is internally used in the `setup.[c]sh` or `setup.bat` shell script, itself generated by the **cmt config** command.

11.3.17 - cmt show <option>

- **author**
- **branches**
- **clients** <package> [<version>]

This command displays all packages that express an explicit **use** statement onto the specified package. If no version is specified on the argument list, then all uses of that package are displayed.

- **constituent_names**
- **constituents**
- **uses**
- **fragment** <name>

This command displays the actual location where the specified make fragment is currently found by **CMT**, taking into account possible overridden definitions.

- **fragments**
- **groups**

This command displays all groups possibly defined in constituents of the current package (using the **-group=<group-name>** option).

- **languages**
- **macro <name>**

This command displays a quite detailed explanation on the value assigned to the macro specified as the additional argument. It presents in particular each intermediate assignments made to this macro by the hierarchy of used statements, as well as the final result of these assignment operations.

By adding a **-tag=<tag>** option to this command, it is possible to simulate the behaviour of this command in another context, without actually going to a machine or an operating system where this configuration is defined.

- **macro_value <name>**

This command displays the raw value assigned to the macro specified as the additional argument. It only presents the final result of the assignment operations performed by used packages.

By adding a **-tag=<tag>** option to this command, it is possible to simulate the behaviour of this command in another context, without actually going to a machine or an operating system where this configuration is defined.

The typical usage of the **show macro_value** command is to get at the shell level (rather than within a **Makefile**) the value of a macro definition, providing means of accessing them (quite similarly to an environment variable) :

```
csh> set compiler='cmt show macro_value cppcomp'
csh> ${compiler} ....
```

- **macros**

This command extracts from the requirements file(s) the complete set of macro definitions, selects the appropriate *tag* definition (or uses the one provided in the **-tag=<tag>** option) and displays the effective macro values corresponding to this tag.

This command is typically used to show the effective list of macros used when running make and can be also used to build, as an argument list, the make command as follows :

```
csh> eval make 'cmt show macros'
```

This use of **cmt show macros** is directly installed within the default target provided in the standard **Makefile.header** file. Therefore if this file is included into the package's **Makefile**, macro definitions provided in the requirements files (the one of the currently built package as well as the ones of the used packages) will be expanded and provided as arguments to make.

By adding a **-tag=<tag>** option to this command, it is possible to simulate the behaviour of this command in another context, without actually going to a machine or an operating system where this configuration is defined.

- **manager**
- **packages**

This command displays all packages (and all versions of them) currently reachable through the current access path definition (which can be displayed using the **cmt show path** command).

- **path**

This command displays the complete and effective access path currently defined using any possible alternate way.

- **pattern <name>**

This command displays how and where the specified pattern is defined, and which packages do apply it.

- **patterns**

This command displays all pattern definitions.

- **pwd**

This command displays a filtered version of the standard **pwd** unix command. The applied filter takes into account the set of aliases installed in the standard configuration file located in **\${CMTROOT}/mgr/cmt_mount_filter**.

This configuration file contains a set of path aliases (one per line) each proposing a translation for non-portable file paths (imposed by mount constraints on some contexts).

- **set_value <name>**
- **set <name>**
- **sets**
- **strategies**
- **tags**
- **uses**

This command displays a quite comprehensive and detailed explanation of the hierarchy of use statements, with the effective selection made between possibly compatible versions.

```
# use CMT v1r8 /lal
# use Cm v7r5
#   use CSet v2r5
#
# Selection :
use CSet v2r5 /lal
use Cm v7r5 /lal
use CMT v1r8 /lal
```

The **-quiet** option may be used to remove from the output, the comments (beginning with the **#** character), so as to display a simple list of used packages, starting from the deepest uses.

- **version**

This command displays the version tag of the current package.

- **versions <name>**

This command displays the reachable versions of the specified package, looking at the current access paths.

11.3.18 - cmt system

This command displays the current value assigned by default to the **CMTCONFIG** environment variable.

11.3.19 - cmt unlock

cmt unlock [<package> <version> [<area>]]

This command tries to remove a lock from the current package (or from the specified package). This consists in the following operations:

1. Check if a conflicting lock is already set onto this package (ie. a lock owned by another user).
2. If not, then remove the text file named **lock.cmt** from the **cmt/mgr** branch of the package.
3. Run a shell command described in the macro named **unlock_command** meant to remove physical locks from all files for this version of this package. A typical definition for this macro could be:

```
macro unlock_command "chmod -R g+w ../*" \  
WIN32               "attrib /S /D -R ../*"
```

11.3.20 - cmt version

This command shows the current version of **CMT**, including (if applicable) the actual patch level. This always corresponds to the corresponding CVS tag assigned to **CMT** sources.

11.3.21 - cmt cvstags <module>

(see the section on *how to use CVS together with CMT* for more details on this command)

11.3.22 - **cmt cvsbranches** <module>

11.3.23 - **cmt cvssubpackages** <module>

11.4 - The setup and cleanup scripts

They are produced by the **cmt config** command and their contents is built according to the specifications stored in the requirements file.

One flavour of these scripts is generated per shell family (**csh**, **sh** and **bat**), yielding the following scripts :

```
setup.csh
setup.sh
setup.bat
cleanup.csh
cleanup.sh
```

The main sections installed within a setup script are :

1. Connection to the current version of the **CMT** package.
2. Setting the set of user defined public variables specified in the requirements file (including those defined by all used packages). This is achieved by running the **cmt setup** utility into a temporary file and running this temporary file.
3. Activation of the user defined setup and cleanup scripts (those specified using the **setup_script** and **cleanup_script** statements).

It should be noted that these setup scripts do *not* contain machine specific information (due to the online use of the **cmt setup** command). Therefore, it is perfectly possible to use the same setup script on various platforms (as soon as they share the directories) and this also shows that the configuration operation (the **cmt config** command) is required only once for a set of multiple platforms sharing a development area.

11.5 - **cmt build prototype**

This command is only provided for development of **C** modules. It generates a **C** header file containing the set of prototype statements for all public functions of a given module. Its output is a file with the same name as the input source (given as the argument) and suffixed with **.ph**.

The generated prototype header file is meant to be included wherever it is needed (in the module file itself for instance).

A typical example of the use of **cmt build prototype** could be :

```
csh> cd ../src
csh> cmt build prototype FooA.c
Building FooA.ph
```

Running **cmt build prototype** will only produce a new prototype header file if the output is actually different from the existing one (if it exists) in order to avoid confusing *make* checks.

The effective use of this facility (which may not be appropriate in all projects) is controlled by one option of the build strategy, which can take one of the two values:

```
build_strategy prototypes
build_strategy no_prototypes
```

In addition to this global strategy specification, each application or library may individually override it using the **-prototypes** or **-no_prototypes** options.

Lastly, the actual behaviour of the prototype generator is defined in the standard make macro **build_prototype** (which default to call the **cmt build prototype** command, allowing a user defined behaviour to this feature)

12 - Using cvs together with CMT

Nothing special is apriori required by **CMT** with respect to the use of **CVS**. Nevertheless, one may advertize some well tested conventions and practices which turned out to ensure a good level of consistency between the two utilities.

Although none of these are required, the **cmt** general command provides a few utilities so as to simplify the use of these practices. It should be noted that the added features provided by **cmt** rely on the possibility to *query* **CVS** about the existing **CMT** packages and the possible tags setup for these packages. **CVS** does not by default permit such query operations (since they require to scan the physical **CVS** repository). Therefore **CMT** provides a hook to **CVS** (based upon standard **CVS** features - not patches) for this. This hook can be installed by the following procedure (see sections below for more details):

```
> (cd ${CMTROOT}/mgr; gmake installcvs)
```

12.1 - Importing a package into a cvs repository

Generally, everything composing a package (below the *version* directory and besides the *binary* directories) is relevant to be imported. Then choosing a **cvs module** name is generally done on the basis of the package name. Taking the previous examples, one could import the **Foo** package as follows :

```
csh> cd ...../Foo/v1
csh> cvs import -m "First import" -I alpha -I hp9000s700 Foo LAL v1
```

In this example,

- we have ignored the currently existing binary directories (here **alpha** and **hp9000s700**)
- the **cvs** module name is identical to the package name (**Foo**)
- the original symbolic insertion tag is identical to the version identifier (**v1**)

The choice of the module name can generally be identical to the package name. However, some

site specific management issues may lead to different choices (typically, a top directory where groups of packages are gathered may be inserted).

Conversely, using symbolic tags identical to version identifiers appears to be a very good practice. The only constraint induced by **cvs** is that the symbolic tags may not contain *dot* characters ('.'), whereas no restriction exist from **CMT** itself. Thus version identifiers like **v3r2** will be preferred to the **v3.2** form.

12.2 - Checking a package out from a cvs repository

Assuming the previous conventions on module name and version identifier have been selected when importing a package, the following operations will naturally intervene when one need to check a package out (typically to work on it or to install it on some platform) :

```
csh> cd <some root>           (1)
csh> mkdir Foo                (2)
csh> cd Foo
csh> cvs checkout -d v1 Foo   (3)
csh> cd v1/cmt
csh> cmt config               (4)
csh> source setup.csh         (5)
csh> [g]make                  (6)
```

1. one always have to select a root directory where to settle down this copy of the extracted package. This may either be the so-called *default root* or any other appropriate directory. In both cases, the next **cmt config** operation will automatically take care of this effective location.
2. creating a base directory with the package name is mandatory here, and is *not* taken into account by **cvs** during the *chaekout* operation since one wants to insert the *version* branch in between.
3. the package is checked out into a directory named with the expected version identifier exactly corresponding to the version currently stored in the **cvs** repository.
4. then using the **cmt config** command (from the **cmt** branch) will update the setup scripts against the **requirements** file and the effective current package location.
5. using this updated version of the setup script provides the appropriate set of environment variables
6. lastly, rebuilding the entire package is possible simply using the **[g]make** command.

The actions decribed just above (from number 2 to number 4 included) can also be performed using the **cmt checkout** command.

```
> cd <some work area>
> cmt checkout [modifier ...] <package> ...

modifier :
-l      Do not process used packages (default).
-R      Process used packages recursively.
-r rev  Check out version tag. (is sticky)
-d dir  Check out into dir instead of module name.
-n      simulation mode on
-v      verbose mode on
```

Thus the previous example would become:

```
csh> cd <some root>
csh> cmt checkout Foo
csh> cd Foo/v1/cmt
csh> source setup.csh
csh> [g]make
```

12.3 - Querying CVS about some important infos

It is possible, using the commands :

- **cmt cvstags <module>**
- **cmt cvsbranches <module>**
- **cmt cvssubpackages <module>**

to query the **CVS** repository about the existing tags installed onto a given **CVS** module, the subdirectories and the subpackages (in the **CMT** meaning, i.e. when a **requirements** file exists).

```
> cmt cvstags Cm
v7r6 v7r5 v7r4 v7r3 v7r1 v7
> cmt cvstags Co
v3r7 v3r6 v3
```

One should notice here that the **cvstags** command can give informations about any type of module, even if it is not managed in the **CMT** environment.

However, in order to let this mechanism operate, it is required to install some elements into the physical **CVS** repository (*which may require some access rights into it*). This installation procedure (to be done only once in the life of the repository) can be achieved through the following command:

```
> (cd ${CMTROOT}/mgr; gmake installcvs)
```

However, the details of the procedure is listed below (this section is preferably reserved for system managers and can easily be skipped by standard users):

1. copy the **cmt_buildcvsinfos2.sh** shell script into the management directory **\${CVSROOT}/CVSROOT** as follows :

```
> cp ${CMTROOT}/mgr/cmt_buildcvsinfos2.sh ${CVSROOT}/CVSROOT
```

2. install one special statement in the **loginfo** administrative file as follows :

```
> cd ...
> cvs checkout CVSROOT
> cd CVSROOT
> vi loginfo
...
.cmtcvsinfos ${CVSROOT}/CVSROOT/cmt_buildcvsinfos2.sh
> cvs commit -m "set up commitinfo for CMT"
```

12.4 - Working on a package, creating a new release

This section presents the way to instantiate a new release of a given package, which happens when the foreseen modifications will yield additions or changes to the application programming interface of the package.

Then the version tag is supposed to be moved forward, either increasing its minor identifier (in case of simple additions) or its major identifier (in case of changes).

The following actions should be undertaken then :

1. understand what is the latest version tag (typically by using the **cmt cvstags** command).
Let's call it **old-tag**.
2. select (according to the foreseen amount of changes) what will be the next version tag. Let's call it **new-tag**.
3. check the most recent version of the package in your development area :

```
> cd <development area>
> cvs checkout -d <new-tag> <package>
```

4. configure this new release, and rebuild it :

```
> cd <new-tag>/cmt
> cmt config
> source setup.csh
> [g]make
```

12.5 - Getting a particular tagged version out of cvs

The previous example presented the standard case where one gets the *most recent* version of a given package. The procedure is only slightly modified when one wants to extract a previously tagged version. Let's imagine that the **Foo** package has evolved by several iterations, leading to several tagged releases in the **cvs** repository (say **v2** and **v3**). If the **v2** release is to be used (e.g. for understanding and fixing a problem discovered in the running version) one will operate as follows :

```
csh> cd <some root>
csh> mkdir Foo
csh> cd Foo
csh> cvs checkout -d v2 -r v2 Foo
csh> cd v2/cmt
csh> cmt config
csh> source setup.csh
csh> make
```

13 - Interfacing an external package with CMT

Very often, external packages (typically commercial products, or third party software) are to be used by packages developed in the context of the **CMT** environment. Although this can obviously be done simply by specifying compiler or linker options internally to the client packages, it can be quite powerful to interface these so-called *external* packages to **CMT** by defining a *glue* package, where configuration specifications for this external package are detailed.

Using this approach, one may :

- provide a *nickname* for this external package,
- control a uniformly defined version tag,
- provide compiler options using the standard make macros `<package>_cflags`, `<package>_cppflags` or `<package>_fflags`,
- specify a set of search paths for the include files, using the `include_dirs` statement,
- provide linker options using the standard make macros `<package>_linkopts`

Let's consider the example of the **OPACS** package. This package is provided outside of the **CMT** environment. Providing a directory tree following the **CMT** conventions (ie. a branch named after the version identifier, then an **cmt** branch) then a **requirements** file, containing (among other statements not shown for the sake of clarity) :

```
package OPACS

include_dirs ${Wo_root}/include ${Co_root}/include ${Xx_root}/include \
${Ho_root}/include ${Go_root}/include ${Xo_root}/include

public
macro OPACS_cflags      "-DHAS_XO -DHAS_XM"
macro OPACS_cppflags    " $(OPACS_cflags) "

macro OPACS_linkopts    "$$(Wo_linkopts) $(Xo_linkopts) $(Go_linkopts) \
$(Glo_linkopts) $(Xx_linkopts) $(Ho_linkopts) $(Html_o_linkopts) \
$(W3o_linkopts) $(Co_linkopts) $(X_linkopts)"
```

Then every package or application, client of this **OPACS** package would have just to provide a use statement like :

```
use OPACS v3
```

This procedure gives the complete benefit of the use relationships between packages (a client application transparently inherits all configuration specifications) while keeping unchanged the original referenced package, allowing to apply this approach even to commercial products so that they may be integrated in resource usage surveys similarly to local packages.

14 - Installing CMT for the first time

These sections are of interest only if **CMT** is not yet installed on your site, or if you need a private installation.

The first question you need to answer is the location where to install **CMT**. This location is typically a disk area where most of packages managed in your project will be located.

Then, you have to fetch the distribution kit from the Web at <http://www.lal.in2p3.fr/SI/CMT/CMT.htm>. You must get at least the primary distribution kit containing the basic configuration information and the **CMT** sources. This operation results in a set of directories hanging below the **CMT** root and the version directory. The src branch contains the sources of **CMT**, the fragments branch contains the makefile fragments and the mgr branch contains the scripts needed to build or operate **CMT**.

14.1 - Installing CMT on your Unix site

The very first operation after downloading **CMT** consists in running the INSTALL shell script. This will build the setup scripts required by any **CMT** user.

Then you may either decide to build **CMT** by yourself or fetch a pre-built binary from the same Web location. The prebuilt binary versions may not exist for the actual platform you are working on. You will see on the distribution page the precise configurations used for building those binaries.

In case you have to build **CMT** yourself, you need a C++ compiler capable of handling templates (although the support for STL is not required). There is a Makefile provided in the distribution kit which takes g++ by default as the compiler. If you need a specific C++ compiler you will override the cpp macro as follows:

```
> gmake cpp=CC
```

The cppflags macro permits too to override behaviour of the compilation.

Another important concern is the way **CMT** will identify the platform. **CMT** builds a configuration tag per each type of platform, and uses this tag for naming the directory where all binary files will be stored. As such this tag has to be defined prior to even build **CMT** itself.

CMT builds the default configuration by running the cmt_system.sh script found in the mgr branch of **CMT**. Run it manually to see what is the default value provided by this script. You might consider changing its algorithm for your own convenience.

A distribution kit may be obtained at the following URL :

```
http://www.lal.in2p3.fr/SI/CMT/CMT.htm
```

Once the **tar** file has been downloaded, the following operations must be achieved :

1. Select a root directory where to install **CMT**. Typically this will correspond to a development area or a public distribution area.
2. Import the distribution kit mentioned above.
3. Uncompress and untar it.
4. Configure **CMT**.
5. **CMT** is ready to be used for developing packages.

A typical corresponding session could look like :

```
csh> cd /Packages
csh> <get the tar file from the Web>
csh> uncompress CMTv1r8.tar.Z
csh> tar xvf CMTv1r8.tar
csh> cd CMT/v1r8/mgr
csh> ./INSTALL
csh> source setup.csh
csh> gmake
```

14.2 - Installing CMT on a Windows or Windows NT site

You first have to fetch the distribution kit from the Web at <http://www.lal.in2p3.fr/SI/CMT/CMT.htm>. You must get at least the primary distribution kit containing the basic configuration information and the **CMT** sources. This operation results in a set of directories hanging below the **CMT** root and the version directory. The binary kit provided for Windows environments will generally fit your needs.

You should consider getting the pre-compiled (for a Windows environment) applications, and especially the **..\VisualC\install.exe** application, which permits to interactively configure the registry entries as described in the next paragraph.

The next operation consists in defining a few registries (typically using the standard RegEdit facility or the **install.exe** special application):

- HKEY_LOCAL_MACHINE/Software/CMT/root will contain the root directory where **CMT** is installed (eg. "e:").
- HKEY_LOCAL_MACHINE/Software/CMT/version will contain the current version tag of **CMT** ("v1r8" for this version).
- HKEY_LOCAL_MACHINE/Software/CMT/path/ may optionally contain a set of text values corresponding to the different package global access paths.
- HKEY_LOCAL_MACHINE/Software/CMT/site will contain the conventional site name.
- HKEY_CURRENT_USER/Software/CMT/path/ may contain a set of text of text values corresponding to the different package private access paths.

CMT can also be configured to run on DOS-based environments using the **nmake** facility. In this case, the installation procedure is very similar to the Unix one:

A typical corresponding session could look like :

```
dos> cd Packages
dos> <get the tar file from the Web>
dos> cd CMT\v1r8\mgr
dos> INSTALL
dos> setup.bat
dos> nmake /f nmake
```

15 - Differences with previous versions of CMT

15.1 - Converting a package that was managed with previous versions of CMT (or methods)

The primary source of information handled by **CMT**, i.e. the syntax - and semantics - of the **requirements** file is supposed to be maintained as backward compatible with previous versions. Therefore we expect that the effects of using a new version of **CMT** to a package already managed by previous versions of **CMT**, will remain limited.

Generally, it is enough to just *re-configure* the package, using the well known command

```
> cmt config
```

This will result in re-generating the **setup** scripts, and verifying **Makefile**. A proper **CMT Makefile** contains at least the two following lines:

```
include ${CMTROOT}/src/Makefile.header  
include ${CMTROOT}/src/constituents.make
```

These two lines are the only required lines to be present in an operational **Makefile**. However, the user is entirely free to install additional make statements at any location for his/her own purpose.

No further operation is then needed. All other **makefile** fragments will be automatically generated at make time. It is even recommended to remove any existing **makefile** fragment generated by previous versions of **CMT**. This can be easily done by using the dedicated **configclean** target as follows

```
> gmake configclean
```

it might also be useful (if not recommended !) to clean the binary directories and rebuild it as follows:

```
> gmake config  
> gmake
```

Lastly, it's often usefull to broadcast these actions (and primarily the **cmt config** action) towards all used packages at once. This of course can easily be done through the **cmt broadcast** command as follows:

```
> cmt broadcast cmt config  
> cmt broadcast cmt gmake configclean
```

15.2 - Operations in a Windows environment

A graphical and interactive application (**cmtw**) is now provided on Windows (95/98/NT) environments. This application permits to browse package directories, to select any version of any package. Its configuration is shown, and interactive edition is possible on its requirements file. A few operations are also possible, such as the generation of MSDev configuration files, so as to permit to directly work of packages managed by **CMT** with the MSDev development environment. Currently this support is still evolving and user might see limitations in the dialog between **CMT** and MSDev (only the constituent definitions - applications and libraries - and the use mechanism - package relationships - are understood in the context of MSDev). Users of these new facilities are kindly invited to send their comments, bug observations, suggestions or even contributions to the author.

16 - Appendices

16.1 - Standard make targets predefined in CMT

These targets can always be listed through the following command :

```
> gmake help
```

<i>target</i>	<i>usage</i>
help	Get the list of possible make target for this package.
all	build all components of this package.
clean	remove everything that can be rebuilt by make
configclean	remove all intermediate makefile fragments

check	run all applications defined with the -check option
<i>component-name</i>	only build this particular component (as opposed to the all target that tries to build all components of this package)
<i>group-name</i>	build all constituents belonging to this group (ie. those defined using the same -group= option)

These targets have to be specified as follows :

```
> gmake clean
> gmake Foo
```

16.2 - Standard macros predefined in CMT

16.2.1 - Structural macros

These macros describe the structural conventions followed by **CMT**. They receive a conventional default value from the **CMT** requirements file. However, they can be overridden in any package for its own needs.

<i>macro</i>	<i>usage</i>	<i>default value</i>
tag	gives the binary tag	<code>\${CMTCONFIG}</code>
src	the src branch	<code>../src/</code>
inc	the include branch	<code>../src/</code>
mgr	the cmt or mgr branch	<code>../cmt/</code> or <code>../mgr/</code>
bin	the branch for binaries	<code>../\${CMTCONFIG}/</code>
javabin	the branch for java classes	<code>../classes/</code>
doc	the doc branch	<code>../doc/</code>

16.2.2 - Language related macros

These macros are purely conventional. They are expected in the various make fragments available from **CMT** itself for providing the various building actions.

During the mechanism of new language declaration and definition available in the **CMT** syntax, developers are expected to define similar conventions for corresponding actions.

Their default values are originally defined inside the requirements file of the **CMT** package itself but can be *redefined* by providing a new definition in the package's requirements file using the **macro** statement. The original definition can be *completed* using the **macro_append** or **macro_prepend** statements.

cc	The C compiler	cc
ccomp	The C compiling command	\$(cc) -c -I\$(inc) \$(includes) \$(cflags)
clink	The C linking command	\$(cc) \$(clinkflags)
cflags	The C compilation flags	<i>none</i>
pp_cflags	The preprocessor flags for C	<i>none</i>
clinkflags	The C link flags	<i>none</i>

cpp	The C++ compiler	g++
cppomp	The C++ compiling command	\$(cpp) -c -I\$(inc) \$(includes) \$(cppflags)
cpplink	The C++ linking command	\$(cpp) \$(cpplinkflags)
cppflags	The C++ compilation flags	<i>none</i>
pp_cppflags	The preprocessor flags for C++	<i>none</i>
cpplinkflags	The C++ link flags	<i>none</i>

for	The Fortran compiler	f77
fcomp	The Fortran compiling command	\$(for) -c -I\$(inc) \$(includes) \$(fflags)
flink	The Fortran linking command	\$(for) \$(clinkflags)
fflags	The Fortran compilation flags	<i>none</i>
pp_fflags	The preprocessor flags for fortran	<i>none</i>
flinkflags	The Fortran link flags	<i>none</i>
ppcmd	The include file command for Fortran	-I

javacomp	The java compiling command	javac
jar	The java archiver command	jar

lex	The Lex command	lex \$(lexflags)
lexflags	The Lex flags	<i>none</i>
yacc	The Yacc command	yacc \$(yaccflags)
yaccflags	The Yacc flags	<i>none</i>
ar	The archive command	ar -clr
ranlib	The ranlib command	ranlib

16.2.3 - Package customizing macros

These macros do not receive default values. They are all prefixed by the package name. They are meant to provide specific variant to the corresponding generic language related macros.

They are automatically and by default concatenated by **CMT** to fill in the corresponding global *use* macros (see appendix on [generated macros](#)). However, this concatenation mechanism is discarded when the **-no_auto_imports** option is specified in the corresponding use statement.

<package>_cflags	specific C flags
<package>_pp_cflags	specific C preprocessor flags
<package>_cppflags	specific C++ flags
<package>_pp_cppflags	specific C++ preprocessor flags
<package>_fflags	specific Fortran flags
<package>_pp_fflags	specific Fortran preprocessor flags
<package>_libraries	gives the (space separated) list of library names exported by this package. This list is typically used in the cmt build library_links command.
<package>_linkopts	<p>provide the linker options required by any application willing to access the different libraries offered by the package. This may include support for several libraries per package.</p> <p>A typical example of how to define such a macro could be :</p> <pre>macro Cm_linkopts "-L\$(CMROOT)/\$(Cm_tag) -lCm -lm"</pre>
<package>_stamps	<p>may contain a list of <i>stamp</i> file names (or make targets). Whenever a library is modified, one dedicated stamp file is re-created, simply to mark the reconstruction date. The name of this stamp file is conventionally <library>.stamp. Thus, a typical definition for this macro could be :</p> <pre>macro Cm_stamps "\$(Cm_root)/\$(Cm_tag)/Cm.stamp"</pre> <p>Then, these stamp file references are accumulated into the standard macro named use_stamps which is always installed within the dependency list for applications, so that whenever one of the libraries used from the hierarchy of used packages changes, the application will be automatically rebuilt.</p>

16.2.4 - Constituent specific customizing macros

These macros do not receive any default values (ie they are empty by default). They are meant to provide for each constituent, specific variants to the corresponding generic language related macros.

By convention, they are all prefixed by the constituent name. But macros used for defining compiler options are in addition prefixed by the constituent category (either **lib_**, **app_** or **doc_**

They are used in the various make fragments for fine customization of the build command parameters.

<category>_<constituent>_cflags	specific C flags
<category>_<constituent>_pp_cflags	specific C preprocessor flags
<category>_<constituent>_cppflags	specific C++ flags
<category>_<constituent>_pp_cppflags	specific C++ preprocessor flags
<category>_<constituent>_fflags	specific Fortran flags
<category>_<constituent>_pp_fflags	specific Fortran preprocessor flags
<constituent>linkopts	provides additional linker options to the application. It is complementary to - and should not be confused with - the <package>_linkopts macro, which provides exported linker options required by clients packages to use the package libraries.
<constituent>_shlibflags	provides additional linker options used when building a shared library. Generally, a simple shared library does not need any external reference to be resolved at build time (it is in this case supposed to get its unresolved references from other shared libraries). However, (typically when one builds a dynamic loading capable component) it might be desired to statically link it with other libraries (making them somewhat private).
<constituent>_dependencies	provides user defined dependency specifications for each constituent. The typical use of this macro is fill it with the name of the list of some other constituents which <i>have</i> to be rebuilt first (since each constituent is associated with a target with the same name). This is especially needed when one want to use the parallel gmake (ie. the -j option of gmake).

16.2.5 - Source specific customizing macros

These macros do not receive any default values (ie they are empty by default). They are meant to provide for each source file, specific variants to the corresponding generic language related macros.

By convention, they are all prefixed by the source file name followed by the source file suffix (either **_c**, **_cxx**, **_f**, etc.)

They are used in the various make fragments for fine customization of the build command parameters.

<constituent>_<suffix>_cflags	specific C flags
<constituent>_<suffix>_cppflags	specific C++ flags
<constituent>_<suffix>_fflags	specific Fortran flags

16.2.6 - Generated macros

These macros are automatically *generated* when **make** is run.

The first set of them provide constant values corresponding to **CMT** based information. They are not meant to be overridden by the user, since they serve as a communication mean between **CMT** and the user.

<PACKAGE>ROOT	The access path of the package (including the version branch)
<package>_root	The access path of the package (including the version branch). This macro is very similar to the <PACKAGE>ROOT macro except that it tries to use a relative path instead of an absolute one.
<PACKAGE>VERSION	The used version of the package
PACKAGE_ROOT	The access path of the current package (including the version branch)
package	The name of the current package
version	The version tag of the current package

The second set is deduced from the context and from the requirements file of the package. They can be overridden by the user so as to customize the **CMT** behaviour.

<package>_tag	The specific configuration tag for the package. By default it is set to \$(tag) but can be freely overridden
constituents	The ordered set of constituents declared without any group option
<group-name>_constituents	The ordered set of all constituents declared using a group=<group-name> option

The third set of generated macros are the *global use macros*. They correspond to the concatenation of the corresponding package specific customizing options that can be deduced from the ordered set of *use* statements found in the requirements file (taking into account the complete hierarchy of used packages with the exception of those specified with the **-no_auto_imports** option in their use statement) :

use_cflags	C compiler flags
use_pp_cflags	Preprocessor flags for the C language
use_cppflags	C++ compiler flags
use_pp_cppflags	Preprocessor flags for the C++ language
use_fflags	Fortran compiler flags
use_pp_fflags	Preprocessor flags for the Fortran language
use_libraries	List of library names
use_linkopts	Linker options
use_stamps	Dependency stamps
use_requirements	The set of used requirements
use_includes	The set of include search paths options for the preprocessor from the used packages
use_fincludes	The set of include search paths options for the fortran preprocessor from the used packages
includes	The overall set of include search paths for the preprocessor
fincludes	The overall set of include search paths options for the fortran preprocessor

16.2.7 - Utility macros

These macros are used to specify the behaviour of various actions in CMT.

X11_cflags	compilation flags for X11
Xm_cflags	compilation flags for Motif
X_linkopts	Link options for XWindows (and Motif)
make_shlib	The command used to generate the shared library from the static one
shlibsuffix	The system dependent suffix for shared libraries
shlibbuilder	The loader used to build the shared library
shlibflags	The additional options given to the shared library builder
symlink	The command used to install a symbolic link
	The command used to remove a symbolic link
build_prototype	The command used to generate the C prototype header file (default to the internal cmt dedicated command)
build_dependencies	The command used to generate dependencies (default to the internal cmt dedicated command)
lock_command	The command used to physically lock a package
unlock_command	The command used to physically unlock a package
make_hosts	The list of remote host names which exactly require the make command
gmake_hosts	The list of remote host names which exactly require the gmake command

16.3 - Standard templates for makefile fragments

<i>template name</i>	<i>usage</i>	<i>used in fragment</i>
ADDINCLUDE	additional include path	<language> java
CONSTITUENT	name of the constituent	<language> java jar make_header jar_header java_header library_header application_header protos_header library_no_share library application dependencies cleanup_header cleanup_library cleanup_application check_application document_header <document> trailer dsw_all_project_dependency dsw_project dsp_library_header dsp_shared_library_header dsp_windows_header dsp_application_header dsp_trailer constituent check_application_header
DATE	now	make_header
FILENAME	file name without path	buildproto <language> <document>

FILEPATH	file path	buildproto <language> <document>
FILESUFFIX	file suffix (without dot)	<language>
FILESUFFIX	file suffix (with dot)	<document>
FULLNAME	complete file path and name	<language> cleanup <document> dsp_contents
GROUP	group name	constituents_header
LINE	source files	<language> dependencies constituent
LINKMACRO	link macro	application
NAME	file name without path and suffix	buildproto <language> java <document>
OBJS	object files	jar_header java_header jar library_no_share library application cleanup_java document_header trailer
OUTPUTNAME	output file name	java
PACKAGE	current package name	<language> dsw_header dsw_all_project dsw_all_project_trailer dsw_trailer dsp_all make_setup_header make_setup readme_header readme readme_use readme_trailer
PACKAGEPATH	current package location	readme_use
PROTOSTAMPS	prototype stamp files	protos_header
PROTOTARGET	prototype target name	library_header application_header
SUFFIX	document suffix	<document>
TITLE	title for make header	make_header
USER	user name	make_header
VERSION	current package version tag	readme_header readme readme_use

16.4 - Makefile generation sequences

This section describes the various makefile generation sequences provided by **CMT**. Each sequence description shows the precise set of make fragments used during the operation.

<i>Generated makefile</i>	<i>description</i>	<i>used make fragments</i>
setup.make	Configuration files for make	1. make_setup_header 2. make_setup
constituents.make	the main entry point point for all constituent targets	1. constituents_header 2. constituent 3. check_application_header
<constituent>.make	application or library specific make fragment	1. make_header 2. java_header jar_header library_header application_header 3. protos_header 4. buildproto 5. jar library library_no_share application 6. dependencies 7. <language> <language>_library java 8. cleanup_header 9. cleanup 10. cleanup_application 11. cleanup_objects 12. cleanup_java 13. cleanup_library 14. check_application
<constituent>.make	document specific make fragment	1. make_header 2. document_header 3. dependencies 4. <document> 5. <document-trailer> 6. cleanup_header
<package>.dsw	Visual workspace configuration files	1. dsw_header 2. dsw_all_project_header 3. dsw_all_project_dependency 4. dsw_all_project_trailer 5. dsw_project 6. dsw_trailer 7. dsp_all

<constituent>.dsp	Visual project configuration files	1. dsp_library_header dsp_shared_library_header dsp_windows_header dsp_application_header 2. dsp_contents 3. dsp_trailer
README	.	1. readme_header 2. readme 3. readme_use 4. readme_trailer

16.5 - The complete requirements syntax

The syntax of specification statements that can be installed in a **requirements** file are :

```

cmt-statement      :  alias
                    |  application
                    |  apply_pattern
                    |  author
                    |  branches
                    |  build_strategy
                    |  cleanup_script
                    |  document
                    |  ignore_pattern
                    |  include_dirs
                    |  include_path
                    |  language
                    |  library
                    |  macro
                    |  macro_append
                    |  macro_prepend
                    |  macro_remove
                    |  macro_remove_all
                    |  make_fragment
                    |  manager

```

		<u>package</u>
		<u>path</u>
		<u>path append</u>
		<u>path prepend</u>
		<u>path remove</u>
		<u>pattern</u>
		<u>private</u>
		<u>public</u>
		<u>set</u>
		<u>set append</u>
		<u>set prepend</u>
		<u>set remove</u>
		<u>setup script</u>
		<u>tag</u>
		<u>use</u>
		<u>version</u>
		<u>version strategy</u>
<i>alias</i>	:	alias <i>alias-name</i> <i>value</i> [<i>tag value ...</i>]
<i>application</i>	:	application <i>application-name</i> [<u><i>constituent-option</i></u> ...] <u><i>source</i></u> ...
<i>constituent-option</i>	:	-OS9
		-windows
		-no_share
		-no_static
		-prototypes
		-no_prototypes
		-check
		-group= <i>group-name</i>
		-suffix= <i>output-suffix</i>
		-import= <i>package-name</i>
		<i>variable-name</i> = <i>variable-value</i>

<i>source</i>	:	[-s=new-search-path] <i>file-name</i>
<i>apply_pattern</i>	:	apply_pattern <i>pattern-name</i> [<i>template-name=value ...</i>]
<i>author</i>	:	author <i>author-name</i>
<i>branches</i>	:	branches <i>branch-name ...</i>
<i>build_strategy</i>	:	build_strategy <i>build-strategy-name</i>
<i>build-strategy-name</i>	:	prototypes
		no_prototypes
		keep_makefiles
		rebuild_makefiles
<i>cleanup_script</i>	:	cleanup_script <i>script-name</i>
<i>document</i>	:	document <i>document-name</i> [<u><i>constituent-option</i></u> ...] <u><i>source</i></u> ...
<i>ignore_pattern</i>	:	ignore_pattern <i>pattern-name</i>
<i>include_dirs</i>	:	include_dirs <i>search-path ...</i>
<i>include_path</i>	:	include_path <i>search-path</i>
<i>language</i>	:	language <i>language-name</i> [<u><i>language-option</i></u> ...]
<i>language-option</i>	:	-suffix= <i>suffix</i>
		-linker= <i>linker-command</i>
		-prototypes
		-preprocessor_command= <i>preprocessor-command</i>
		-fragment= <i>fragment</i>
		-output_suffix= <i>output_suffix</i>
		-extra_output_suffix= <i>extra_output_suffix</i>
<i>library</i>	:	library <i>library-name</i> [<u><i>constituent-option</i></u>] <i>source ...</i>
<i>macro</i>	:	macro <i>macro-name value</i> [<i>tag value ...</i>]
<i>macro_append</i>	:	macro_append <i>macro-name value</i> [<i>tag value ...</i>]
<i>macro_prepend</i>	:	macro_prepend <i>macro-name value</i> [<i>tag value ...</i>]
<i>macro_remove</i>	:	macro_remove <i>macro-name value</i> [<i>tag value ...</i>]
<i>macro_remove_all</i>	:	macro_remove_all <i>macro-name value</i> [<i>tag value ...</i>]
<i>make_fragment</i>	:	make_fragment <i>fragment-name</i> <u><i>fragment-option</i></u>
<i>fragment-option</i>	:	-suffix= <i>suffix</i>

		-dependencies
		-header= <i>fragment</i>
		-trailer= <i>fragment</i>
<i>manager</i>	:	manager <i>manager-name</i>
<i>package</i>	:	package <i>package-name</i>
<i>path</i>	:	path <i>path-name</i> value [tag value ...]
<i>path_append</i>	:	path_append <i>path-name</i> value [tag value ...]
<i>path_prepend</i>	:	path_prepend <i>path-name</i> value [tag value ...]
<i>path_remove</i>	:	path_remove <i>path-name</i> value [tag value ...]
<i>pattern</i>	:	pattern [-global] <i>pattern-name</i> <u>cmt-statement</u> [; <u>cmt-statement</u> ...]
<i>private</i>	:	private
<i>public</i>	:	public
<i>set</i>	:	set <i>set-name</i> value [tag value ...]
<i>set_append</i>	:	set_append <i>set-name</i> value [tag value ...]
<i>set_prepend</i>	:	set_prepend <i>set-name</i> value [tag value ...]
<i>set_remove</i>	:	set_remove <i>set-name</i> value [tag value ...]
<i>setup_script</i>	:	setup_script <i>script-name</i>
<i>tag</i>	:	tag <i>tag-name</i> [tag ...]
<i>use</i>	:	use <i>package-name</i> [<u>version-tag</u> [<i>access-path</i>]] [<u>use-option</u>]
<i>version</i>	:	version <u>version-tag</u>
<i>version-tag</i>	:	key <i>version-number</i> [key <i>release-number</i> [key <i>patch-number</i>]]
<i>use_option</i>	:	-no_auto_imports -auto_imports
<i>key</i>	:	<i>letter</i> ...
<i>version_strategy</i>	:	version_strategy <u>version-strategy-name</u>
<i>version-strategy-name</i>	:	best_fit best_fit_no_check first_choice last_choice

16.6 - The internal mechanism of cmt cvs operations

Generally, CVS does not handle queries upon the repository (such as knowing all installed modules, all tags of the modules etc..). Various tools such as CVSWeb, LXR etc. provide very powerful answers to this question, but all through a Web browser.

CMT provides a hook that can be installed within a CVS repository, based on a helper script that will be activated upon a particular CVS command, and that is able to perform some level of scan within this repository and return filtered information.

More precisely, this helper script (found in `${CMTROOT}/mgr/cmt_buildcvsinfos2.sh`) is meant to be declared within the **loginfo** management file (see the [CVS manual](#) for more details) as one entry named **.cmtcvsinfos**, able to launch the helper script. This installation can be operated at once using the following sequence:

```
> cd ${CMTROOT}/mgr
> gmake installcvs
```

This mechanism is thus fully compatible with standard remote access to the repository.

Once the helper script is installed, the mechanism operates as follows (this actually describes the algorithms installed in the **CvsImplementation::show_cvs_infos** method available in **cmt_cvs.cxx** and is transparently run when one uses the **cmt cvsxxx commands**):

1. Find a location for working with temporary files. This is generally deduced from the `${TMPDIR}` environment variable or in `/tmp` (or in the current directory if none of these methods apply).
2. There, install a directory named **cmtcvs/<unique-name>/cmtcvsinfos**
3. Then, from this directory, try to run a fake import command built as follows:

```
cvs -Q import -m cmt .cmtcvsinfos/<package-name> CMT v1
```

Obviously this command is fake, since no file exist in the temporary directory we have just created. However,

4. This action actually triggers the **cmt_buildcvsinfos2.sh** script, which simply receives in its argument the module name onto which we need information. This information is obtained by scanning the files into the repository, and an answer is built with the following syntax:

```
[error=error-text]          (1)
tags=tag ...                (2)
branches=branch ...         (3)
subpackages=sub-package ... (4)
```

1. In case of error (typically when the requested module is not found in the repository) a text explaining the error condition is returned
2. The list of tags found on the requirements file
3. The list of branches defined in this packages (ie subdirectories not containing a

- requirements file)
4. The list of subpackages (ie subdirectories containing a requirements files)
-

Christian Arnault

Wed Mar 3 16:20:27 MET 1999