CMT

Configuration Management Tool

Version v1r18p20041201

Christian Arnault

`arnault@lal.in2p3.fr`

Document revision date : 2004-12-01

General index

---

1 - 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 the *description* of 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.
- Packages sets may be structured in *areas* implementing a project oriented organization.
- The packages may or not be related to each other (defining a *direct acyclic graph* of packages - not just a single tree).
- The concept of package may be extended to implement structuring or organizing patterns such as those involved in project management.
- Project management policies and behavioural patterns can be easily expressed and automated by CMT.
- 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 one to three numbers the *major* id, the *minor* id, and the *patch* 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.
    - versions differing only by their patch id are said to be fully compatible with each other.
- 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 by `CMT` . 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 (which are often called *external* packages).

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, following projects or teams structuring paradigms,
- describing the configuration requirements for each package:
    - dependencies to other packages,
    - generic behavioural patterns meant to describe generic configuration items or project specific policies.
    - symbols to be exported to client packages (environment variables, make macros, etc...)
    - parameterized configuration activities (documentation generation, deploiement, installation, etc...)
    - components (also named *constituents*) of the packages (libraries, applications, generated documents)
    - parameterization of the build and test tools
    - parameterization of the deployment tools
    - strategies that `CMT` should follow at run time, overriding its default ones.
- 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).

2 - 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 (*obsolescent convention*)

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

  The <version mnemonic> directory level may also be omitted, in which case the version information will be stored inside the cmt directory in a conventional file named `version.cmt` leading to the following alternate organization:

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

  In both cases, the `cmt` directory holds the main source of information needed by `CMT` : the *requirements* file. All `CMT` -related operations are generally executed from this directory.

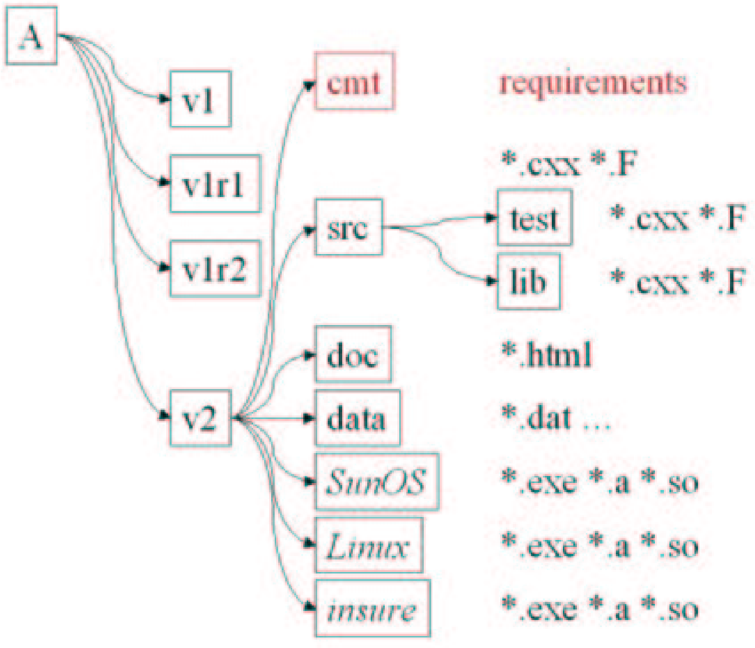
  This style of organization should be considered as the basic (and unique) criterion for a package to be recognized as a valid *CMT package* . Any other structuring convention will be supported by CMT and its operations can always be customized to follow them

  This structure is a central concept since all relationships between packages relies on the package identification which unambiguously and exclusively consists in the duet [*package-name* , *package-version*] (or *package-name* only when the version directory level is omitted).

- Constructing the internal structure of a package.

  Many other parallel directory branches (similar to `cmt`) such as `src` , `include` or `test` may be freely 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* outputs (those that compilers, linkers, archive managers or other kinds of code or pseudo-code generators can produce). Their name always corresponds to the particular *configuration tag* used to produce the output (such as the machine or operating system type). The `CMT` toolkit provides, through the `cmt system` utility, a default value for this token. An environment variable (`CMTCONFIG`) is also assigned to this value (See the complete description of configuration tags).

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



`1` - *Structuring a package* - *A typical example.*

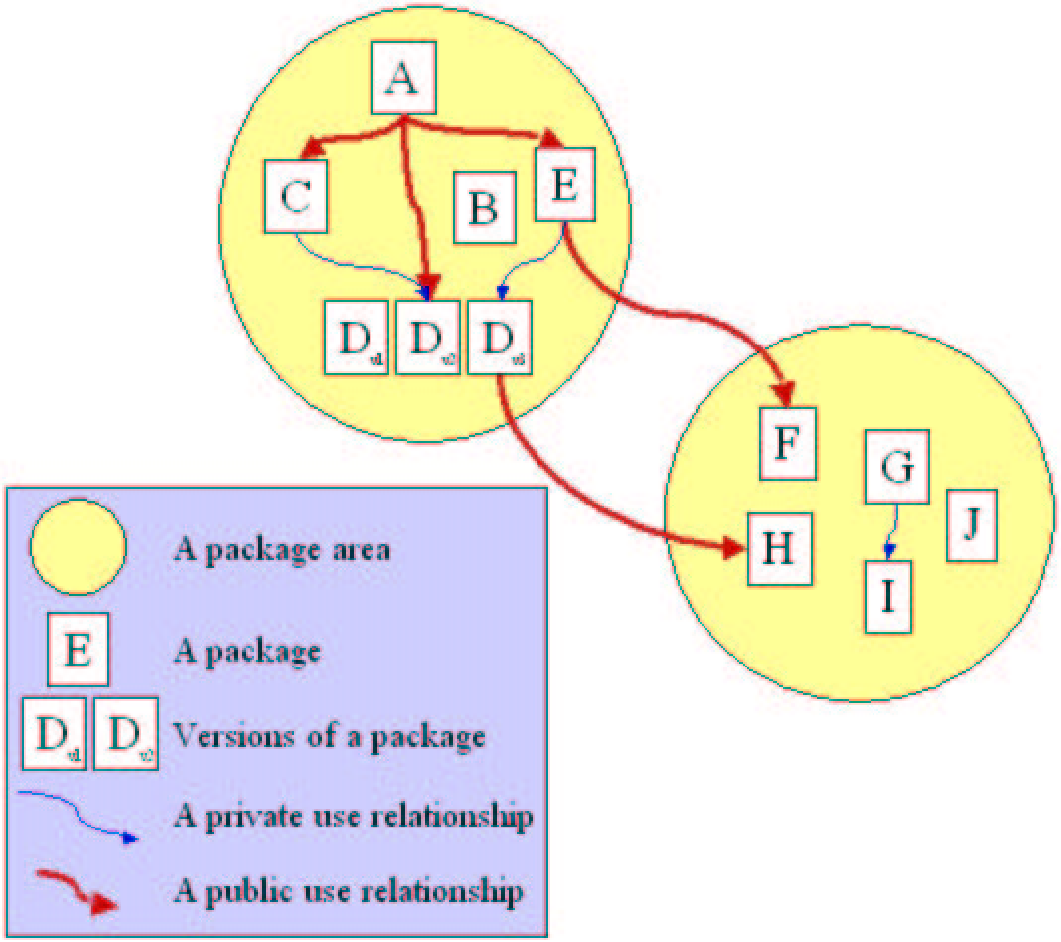
- Organizing a software base.

  A software base is generally composed of multiple coherent sets of packages, each installed in its specific root directory and forming different *package areas*

  There are no constraints on the number of such areas into which `CMT` packages are installed. We'll see later how the different areas can be declared and identified by `CMT` .

  Package areas implement the concept of *projects* or *sub-projects* which correspond to the practical organization of the software developments.

  examples of such organization can be :



2 - *Structuring a sofware base.*

---

3 - The architecture of the environment

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

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

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 area or in project specific 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 products, whereas packages installed elsewhere are rather meant to be managed in a specific 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 generate several minimal scripts (see the description of the create command),
- describing or monitoring :
  - the relationships between the package and other packages
  - the configuration features either specified in the current package, or imported from related (*used*) ones. (symbols, patterns, fragments)
  - the constituents of the package in terms of libraries, executables, or generated documents.
- 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 for packages (though locks) etc...).

---

3. 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
- All variants of Linux (RedHat, Debian, SuSe, ScientificLinux, ...)
- Windows 95/98/NT/Windows2000 in various environments:
  - `CYGWIN_NT-5.1` environment
  - `nmake` based environment
  - `MSDev/VisualC 6` environment
  - `MSDev/VisualC 7` environment
- Darwin (Mac OS X)

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. All platform specific tools will be dynamically reconfigured and parameterized transparently.

---

4 - Installing a new package

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

Therefore, we assume that *some* root directory has been selected by the system manager, and that `CMT` is already installed there. 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/v1r18p20041201/mgr/setup.csh

or

ksh> . /lal/CMT/v1r18p20041201/mgr/setup.sh

or

dos> call \lal\CMT\v1r18p20041201\mgr\setup.bat
```

Obviously, this operation `must` be performed (once) 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 configured package will have one such setup script automatically generated and installed by* `CMT` *. It is one important entry point to any package (and thus to* `CMT` *itself). It provides environment variable definitions and recursive invocations of setup scripts for related (used) packages (A corresponding cleanup script is also provided). 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).*

It is generally good to start by immediately defining a *project* . This project is our first disk area where CMT packages will be located. Remember that several such projects can be set up and defined. The simplest way to do this is:

```
> cmt create_project Dev
------------------------------------------
Configuring environment for project Dev
CMT version v1r18p20041201.
------------------------------------------
Installing the cmt directory
Creating a new project file
```

Once this project has been created we have a complete environment to start creating packages and working out our software base.

A package is primarily defined by a *name* and a *version* identifier (this duet actually forms the complete *package 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 v1r18p20041201.                             [1]
Root set to /home/arnault/mydev.
System is Linux-i686                                    [2]
------------------------------------------
Installing the package directory                        [3]
Installing the version directory
Installing the cmt directory
Installing the src directory
Creating setup scripts.
Creating cleanup scripts.
```

1. This shows which actual CMT version you are currently using
2. This shows the current configuration tag (also available by the `cmt system` command). In this example this is a `Linux` machine
3. This shows the detailed construction of the complete directory structure, starting from the top directory which has the name of the package. Since we are creating a completely new package, there will be by default only two branches below the version directory : `cmt` and `src` .

The package creation occured 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 /ProjectB
------------------------------------------
Configuring environment for package Foo version v1.
CMT version v1r18p20041201.
Root set to /ProjectB.
System is Linux-i686
------------------------------------------
Installing the path directory
Installing the package directory
Installing the version directory
Installing the cmt directory
Installing the src 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 when needed or generated at build time).
- an empty configuration specification file (named `requirements`) installed in the `cmt` branch.
- A minimal `Makefile` (on Unix environments only), containing

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

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

  This `Makefile` does not need any further 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 - if any - into account by `CMT` .

- A similar minimal `NMake` file (on Windows environments only), containing

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

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

- 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 the requirements file is empty) :

```
sh> cd ~/mydev/Foo/v1/cmt
sh> . setup.sh
```

or

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

or

```
dos> cd \mydev\Foo\v1\cmt
dos> call setup.bat
```

The `FOOROOT` and `FOOCONFIG` environment variables are defined automatically by this operation.

It should be noted that running the setup script of a package is not always 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 v1r18p20041201.
Root set to /home/arnault/mydev.
System is Linux-i686
------------------------------------------
Version v1 has been removed from /home/arnault/mydev/Foo
Package Foo has no more versions. Thus it has been removed.
```

or:

```
csh> cmt remove Foo v1 /ProjectB
------------------------------------------
Removing package Foo version v1.
CMT version v1r18p20041201.
Root set to /ProjectB.
System is Linux-i686
------------------------------------------
Version v1 has been removed from /ProjectB/Foo
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 ~/mydev/Foo/v1/cmt
csh> cat >../src/FooTest.c
#include <stdio.h>

int main ()
{
  printf ("Hello Foo\n");
  return (0);
}

csh> vi requirements
...
application FooTest FooTest.c
csh> gmake
csh> source setup.csh
csh> FooTest.exe
Hello Foo
```

Directly running the application is possible since the application has been installed after being built in an automatic *installation area* reachable through the standard `PATH` environment variable

This can also be integrated in the build process 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
```

---

5 - Localizing a package

In the next sections, we'll see that packages *reference* each other by means of *use* relationships. Generally packages are found in different locations, according to the project - or sub-project - they belong to. `CMT` provides a quite flexible mechanism for *localizing* the referenced packages.

The first ingredient we need at this level is to understand how projects themselves are localized, since packages will be found inside project areas.

Projects are localized according to the `CMTPATH` path list. Thus it is *always* required to define this environment variable as soon as several project areas are expected. However, there is one special case where this path list can be avoided, i.e. when only one project area is considered. In this case, the knowledge of this single project area can simply be deduced from the detection of the project file, created at the top of its disk space.

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 three *keys* :

- its name (such as `Bar`)
- its version (such as `v7r5`)
- optionally its expected absolute location or relative offset

```
use Bar v7r5                    [1]
```

or

```
use Bar v7r5 A                  [2]
```

or

```
use Bar v7r5 /ProjectB/A        [3]
```

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 the *use path* is absolute (case #3),
2. the access paths optionally registered in the configuration parameter - see below - `CMTPATH` (and in decreasing priority, the first element being searched for first).

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 developper or in `${CMTROOT}/mgr` . In the Windows environment, this configuration parameter may also be installed as a Registry under either the keys:

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

The project file (i.e. the file `cmt/project.cmt`), when it exists for the current package area, is able to provide an automatic value for the `CMTPATH` search list.

If the *path* argument is specified as a relative path (case #2 above) (ie. there is no leading *slash* character or it's not a *disk* on windows machines), 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 *colons* (*semi-colons* on Windows).

A very typical usage of thie `CMTPATH` search list is to assign one entry per project or sub-project considered in the workning context. This is consistent with an approach where every sub-project will have a dedicated management style, access rights, rsponsible persons, etc.

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

```
csh> setenv CMTPATH /home/arnault/mydev:/ProjectB
```

```
sh> export CMTPATH=/home/arnault/mydev:/ProjectB
```

```
bat> set CMTPATH=/home/arnault/mydev;/ProjectB
```

or (in a `requirements` file)

```
path_append CMTPATH "/home/arnault/mydev"
path_append CMTPATH "/ProjectB"
```

or (in a `.cmtrc` file)

```
CMTPATH=/home/arnault/mydev:/ProjectB
```

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

```
...
use Bar v7r5
use BarA v1 A
```

would look for the package `Bar` from :

1. `/home/arnault/mydev/Bar/v7r5/cmt`
2. `/ProjectB/Bar/v7r5/cmt`

Whereas the package `BarA` would be searched from :

1. `/home/arnault/mydev/A/BarA/v1/cmt`
2. `/ProjectB/A/BarA/v1/cmt`

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

1. there is a first directory level exactly named according to the package name (this is case sensitive),
2. then (optionally) the next directory level is named according to the version tag,
3. then there is a branch named `cmt` ,
4. lastly there is a *requirements* file within this `cmt` branch.

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

The actual complete search list can always be visualized by the command:

```
> cmt show path
# Add path /home/arnault/dev from CMTPATH
# Add path /ProjectB from CMTPATH
#
/home/arnault/dev:/ProjectB
```

---

6 - The concept of project

The purpose of the concept of *project* or *sub-project* is to structure the software bases developed using CMT. This concept basically assigns a semantics to the physical structuring mechanism of the software areas already described by the `CMTPATH` search list.

Some general properties can describe the knowledge of projects in CMT:

- One project is always explicitly or implicitly associated with every CMTPATH entry.
- Every project can be *named* inside a dedicated text file (the *project file*) named `cmt/project.cmt` , located immediately below the corresponding `CMTPATH` entry. The format of this file is

```
project <project-name>
strategy specifications...
```

- When not explicitly named, a sub-project is automatically assigned the default name `Project<N>` by CMT, where `<N>` is the sequence number of the project in the list
- The project file can be created using one of the commands:

  ```
  > cmt create_project <project-name>
  > cmt -project=<project-name> create ...
  > cmt -project=<project-name> checkout ...
  ```

- When being within one project area where a project file has been installed, its CMTPATH is automatically generated as the location where the first visible project file is detected.
- The same project may be associated with several CMTPATH entries, typically when one entry is a development area of another one being the production area of the same sub-project
- Since CMTPATH entries are ordered the projects are hierarchized accordingly. The last CMTPATH entry (i.e. the rightmost one) corresponds to the top project while the first CMTPATH entry (i.e. the leftmost one) corresponds to the bottom project. A project may have a predecessor and/or a successor.

  As an example, suppose we have set up the following `CMTPATH` :

  ```
  $HOME/work-for-A:/ProjectA:/ProjectB
  ```

  Then the following projects will appear displayed from bottom to top as

  ```
  > cmt show projects
  Project1 (in /.../work-for-A)
  Project2 (in /ProjectA)
  Project3 (in /ProjectB)
  CMT (in /afs/cern.ch/sw/contrib)
  ```

  And if projects have been explicitly named this display will become:

  ```
  > cmt show projects
  A (in /.../work-for-A)
  A (in /ProjectA)
  Project3 (in /ProjectB)
  CMT (in /afs/cern.ch/sw/contrib)
  ```

- CMT Strategies (for build or setup) are separately collected into each project. Therefore one can apply different strategies to different sub-projects. The strategy specifications may appear in requirements file of any package of a project or in the `cmt/project.cmt` project file.

  By default a project inherits the strategies of its predecessor. Or if it’s the top project, it follows the default strategies defined by CMT (Refer to this appendix to see the default strategies currently defined by CMT).

- A CMT command displays the projects and their associated CMTPATH:

  ```
  > cmt show projects
  > cmt show project <project-name>
  ```

- The installation area mechanism is strongly connected with the `CMTPATH` hierarchy. Therefore it will also be connected with the project concept. This will happen through several mechanisms:
    - The `<project>` parameter is expanded in the `cmtpath_pattern` construct in addition to the `<path>` parameter. This parameter is assigned the name of the project associated with the running `CMTPATH` entry.
    - The `<project>` parameter is also available for normal patterns. In this case it is assigned the project name associated with the cmtpath parameter for the current package.
    - Every project defines a tag of the same name, and the tag of the current project is active.
    - The `cmt_installarea_prefix` macro is specialized *per project* and every project may override the `<project>_installarea_prefix` macro. The default value of any `<project>_installarea_prefix` is `${cmt_installarea_prefix}`
- Every strategy setting defines two mutually exclusive tags and activates one of them.

```
<project>_<have_item>
<project>_<have_not_item>
```

Examples

```
<project>_prototypes
<project>_no_prototypes
<project>_with_install_area
<project>_without_install_area
<project>_config
<project>_no_config
<project>_root
<project>_no_root
<project>_cleanup
<project>_no_cleanup
```

---

7 - Assigning semantics to packages. Common practices

Generally speaking, CMT makes no assumption on how or why is used a package. However past experience has shown that packages can be categorized according to their purpose or their type of contents.

7. 1 - The primary package

This is the most general and basic pakage type, which provides actual pieces of software, such as libraries or applications. Generally the main activities performed by such a package include building the software (compiling, linking), testing, generating the documentation, installing, ...

A typical package of that kind will contain:

- a `../src` directory containing the sources of the package
- a directory for the include files, with a name that will depend on the structuring policies defined for the project. Tyical examples are

```
../include/
../<packagename>/
```

- a `../doc` directory for the documentation
- a `../test` directory for the test programs.

The requirements file will generally contain at least `library` and `application` statements.

---

7. 2 - The policy package

This kind of package only provides conventions, working methods, general purpose shell scripts but generally provides no software per se. It is designed to gather all policies and management conventions for a project or a sub project.

The basic contents of such a package is the requirements file including

- strategy definitions
- pattern definitions
- general purpose symbol definitions

In principle the idea when such a policy package is defined, is that all packages of the project or of the sub-project will use it

Global patterns may be specified so as to automate the applying of basic policies and conventions.

Typical examples of policies that are profitably specified in such a package are:

- include search path convention (using a global pattern with the `include_dir` statement)
- build or setup strategies
- compiler or linker generic options
- defining the project-wide production tools (compiler, documentation generator, etc...)
- tag associations need to describe the binary tag convention in the project

In large projects, it is even often desirable to split the policies into a set of specialized policies and to associate one dedicated policy package with each of those.

- policies for the test
- policies for each programming language
- policies for documentation management
- policies for installation and deployment
- policies for external software organisation

Then the global policy package will use them

7. 3 - The container or management package

In large projects, it’s often useful to decompose the software base into specialized domains (Core software, Graphics, Database, Online, etc...) or subsets of the software (eg per detector in a physics experiment). Then a container package consists in constructing a simple package with only one requirements file in it and only containing a set of use statements.

Management activities directly related with the associated sub-domain can then be undertaken through this special package:

- version management (such as CVS tagging) of packages belonging to the domain
- consistently building the domain
- Generally the `cmt broadcast` command is widely exploited to perform those management activities.

Generally the use statements installed in a container package make use of explicit version specification (and prohibit wild carding) since each version of this container package acts as a reference of the set of version tags validated for the packages of the domain.

7. 4 - The release package

This package is one particular example of the container concept, but dedicated to manage the project-wide activities. This release package is the primary target of the project manager. It will generally receive as its version tags the version tags assigned to the project releases themselves.

7. 5 - The glue or interface package

This kind of package defines an interface to an existing software product not managed in the context of the project itself. Typical examples concern:

- packages shared from external projects that don’t use CMT as their configuration tool
- third party software (free software, commercial products, ...) locally installed on the development platform.

The primary goal fo such a glue package is to convert the management conventions and policies expected by the referenced product to the ones appropriate for the current project.

- compiler and linker options
- run time settings such as environment variable definitions (PATH, LD_LIBRARY_PATH, etc..)
- data file access
- specification of local installation according to the project strategy

Generally this kind of package only provides a requirements file, or make fragments used to automate some actions (typically when document generation is expected from thie interfaced product)

8 - 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 those 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 *site* manager and can be set with a value representing the site (typical values may be `LAL` , `Virgo` , `Atlas` , `LHCb` , `CERN` , etc.).

This variable, when set, corresponds to a *tag* 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 location path of an external software package. Here we take the example of the Anaphe utility:

```
macro AnapheTOP "" \
      CERN  "/afs/cern.ch/sw/lhcxx" \
      BNL   "/afs/rhic/usatlas/offline/external/lhcxx" \
      LBNL  "/auto/atlas/sw/lhcxx"
```

9 - Configuring a package

The first ingredient of a proper package configuration is the set of configuration parameters which has to be specified in a text file uniquely named `requirements` and necessarily installed in the `cmt` branch of the package directory 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.

The primary goal of this configuration file is to specify *any* configuration information for this package. There is virtually no limit to what could be specified there. And we can expect to find exhaustive information about:

- the primary constituents of the package
- how to rebuild the software
- how to setup and use the software
- how to transport and deploy the package

Many configuration parameters are supposed to be described into this requirements file - 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 policy patterns to be applied by clients of this package
- 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 query 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 show ...` family of commands) are to be embedded in the various productivity tools, such as the setup shell scripts, or makefile fragment generators.

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

Conversely one may say that parameters defined in a requirements file are meant to be exported to the clients of the package.

Other configuration parameters are also optionally inserted from the `HOME` and `USER` *context* requirements files

Typical examples of these query functions are:

- `cmt setup` builds a shell command line for setting up environment variables
- `cmt show projects` gives the ordered sequence of sub-projects comprising the complete software base
- `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.
- `cmt show setup` combines in one display the result of `uses` , `tags` and `path`

---

10 - Selecting a specific configuration

A configuration describes the conditions in which the package has to be built (ie. compiled and linked) or applications can be run. This configuration can depend on :

- the operating system (such as *Linux* , *Windows* , ...)
- the platform (such as *Intel* , *Compaq* , *Sun* , etc...)
- the sub-project into which a package is inserted
- the choice of the compiler (such as *g++* , *egcs* , *CC* , etc...)
- options used for compiling (such as *optimizer* , *debugger* , etc...) or linking
- the context specifications (selecting a particular version of a firmware, selecting a database server, ...)
- the site itself
- the context of a constituent during its rebuild operation

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, or when a project has to be deployed on several sites.

`10. 1` - Describing a configuration

`CMT` relies on several complementary conventions or mechanisms for this description and the associated management. All these conventions rely on the concept of *configuration tags* .

- A tag is a symbol that describes one aspect of the configuration.
- A tag can be *active* when the corresponding aspect of the configuration is true or *inactive* otherwise
- The set of active tags represents the complete configuration known by CMT, and can be visualized with the `cmt show tags` command
- Tags can be combined using logical expressions to form tag associations

1. Some aspects of the configuration - and their associated tags - are automatically deduced from some standard environment variables that the user is expected to specify (typically using shell commands):

    - `CMTCONFIG` describes the current settings for producing binary objects. One default value is provided automatically by CMT, but generally project will override it to apply specific conventions.

The default value is computed by CMT in the ${CMTROOT}/mgr/cmt_system.sh shell script.

This script automatically builds a value characterizing both the machine type and the opebrating 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)

- `CMTSITE` characterizes the current site. Its syntax is completely free

- `CMTEXTRATAGS` may contain a space-separated list of additional tags to systematically activate

---

Note that the `CMTBIN` variable which represents the current binary installation of CMT itself does NOT correspond to any tag.

---

2. Some aspects of the configuration represents the implicit knowledge CMT gets of the current context:

    - The value given by the *uname* standard Unix facility is always a valid configuration tag. (eg. `Linux`)
    - The current major version id of CMT is a valid tag and takes the form `CMTv<n>` (eg. `CMTv1`)
    - The current minor version id of CMT is a valid tag and takes the form `CMTr<n>` (eg. `CMTr18`)
    - The current patch id of CMT is a valid tag and takes the form `CMTp<n>` (eg. `CMTp20030616`)

  - The current sub-project to which the current package belongs, and the various tags automatically generated by CMT to qualify the strategy options.
  - The current hardware understood as filled in the `cmt_hardware` macro
  - The current OS understood as filled in the `cmt_system_version` macro
  - The version of the C++ compiler understood as filled in the `cmt_compiler_version` macro

3. During a `make` session, each individual target being rebuilt may define its own context, when the `-target_tag` is set to the associated constituent, and this is materialized with a dedicated tag named `target_<constituent>` .

   For instance, a package defines a library `A` and an application `P` , both in the default group. Both constituents have their `-target_tag` option set. Thus, when the standard `make` command is run, then those two targets will be rebuilt successively (eg `A` then `B`). Then, during the build of `A` (and only then) the tag named `target_A` will be active and during the build of `B` , the tag named `target_B` will in turn be active.

   ```
   library A -target_tag A.cxx
   application P -target_tag P.cxx

   macro_append cppflags "" target_A "-DA" target_P "-DP"
   ```

4. During the execution of an `action` , a specific context is created, which is materialized with a dedicated tag named `target_<action>` , very similarly to the target tags for constituents.

5. User defined tags can be explicitly or implicitly activated:

   - explicitly from the `cmt` command line, using the `-tag=<tag-list>` option
   - explictly from requirements files using the `apply_tag <tag>` syntax
   - implicitly from requirements files using the tag association syntax, when a tag is associated with an otherwise activated tag. One example is the `Unix` tag associated by CMT itself with most Unix variants

The minimal tag set available from CMT can be visualized as follows (note that the exact output will obviously not necessarily be the one presented in this document according to the context effectively used):

```
> cd ${CMTROOT}
> cmt show tags
CMTv1 (from CMTVERSION)                                                 [1]
CMTr18 (from CMTVERSION) package CMT implies [CMTr14]                   [1]
CMTp20040701 (from CMTVERSION)                                          [1]
Linux (from uname) package CMT implies [Unix]                           [2]
i686-rh73-gcc32-opt (from CMTCONFIG)                                    [3]
CERN (from CMTSITE)                                                     [4]
CMT_prototypes (from PROJECT) excludes [CMT_no_prototypes]              [5]
CMT_with_installarea (from PROJECT) excludes [CMT_without_installarea]
CMT_setup_config (from PROJECT) excludes [CMT_setup_no_config]
CMT_setup_root (from PROJECT) excludes [CMT_setup_no_root]
CMT_setup_cleanup (from PROJECT) excludes [CMT_setup_no_cleanup]
CMTr14 (from package CMT)
i686 (from package CMT)                                                 [6]
rh73 (from package CMT)                                                 [7]
gcc32 (from package CMT)                                                [8]
Unix (from package CMT) excludes [WIN32 Win32]                          [9]
```

1. Implicit tags deduced from the current version of CMT
2. Implicit tag obtained from the uname command (note that there is an associated tag defined here)
3. The current value of CMTCONFIG
4. The current value of CMTSITE
5. The strategy tags
6. Automatic detection of the hardware
7. Automatic detection of the current OS
8. Automatic detection of the C++ compiler version
9. A indirectly activated tag (associated with another active tag)

`10. 2` - Defining the user tags

The user configuration tags can generally be specified though various complementary mechanisms:

- CMTSITE and CMTCONFIG can be specified using standard shell commands (setenv, export, set)

  ```
  sh> export CMTSITE=CERN
  ```

- CMTSITE and CMTCONFIG can alternatively be specified using the `set` statement in a requirements file

  ```
  set CMTSITE "CERN"
  set CMTCONFIG "${CMTBIN}" sun "Solaris-CC-dbg"
  ```

- Additional tags may also be associated with other tags, using the `tag` statement (in a requirements file):

  ```
  tag newtag tag1 tag2 tag3
  ```

  which means that:
  - `newtag` defines a tag (inactive by default)
  - when `newtag` is active, then both tag1, tag2 and tag3 are simultaneously active

- Tags may be declared as *exclusive* using the `tag_exclude` syntax.

  ```
  tag_exclude debug optimized
  ```

  This example implies that the two tags `debug` and `optimized` should never become active simultaneously.

- Tags are assigned priorities according to the way they have been defined. The priority is particularly useful for specifying exclusion. The tag association promotes the priority of the associated tags to the priority of the defining tag. The following decreasing priorities are currently defined by CMT:

  1. tag specified in the command line using the -tag=<tag-list> option
  2. tag deduced from CMTCONFIG
  3. tag defined in a requirements file using the `tag` syntax
  4. tag deduced from CMTSITE

5. tag deduced from *uname*
6. tags deduced from the version of CMT

---

10. 3 - Activating tags

By default, `CMTCONFIG` , *`uname`* and `CMTSITE` (also named system tags) are always active.

The tag associated with the current project name as well as those describing the strategy properties of all projects are also always active.

The target tags associated with constituents (when the `-target_tag` option was set on them) or with actions are automatically activated during the build of the constituent or during the execution of the action.

It is possible to *activate* other tags through the following arguments to *any* cmt command:

- `-tag=<tag-list>`

  will cleanup the complete current tag set, and activate the new tags (the system tags are restored).

- `-tag_add=<tag-list>`

  will add to the current tag set the tags specified in the comma separated list

- `-tag_remove=<tag-list>`

  will remove from the current tag set the tags specified in the comma separated list

---

Beware that giving these arguments generally make the selected tag set active only during the selected command. Therefore two different CMT commands run with different tag sets will generally yield different results.

However it's often useful to state that a given tag or tag set should be active. This can be obtained by the following mechanisms:

1. *Forcing a tag in a requirements file using the `apply_tag` syntax*

   Eg the following syntax installed in a requirements file will force the tag `foo` :

```
tag_apply foo

> cmt show tags
CMTv1 (from CMTVERSION)
CMTr18 (from CMTVERSION)
CMTp0 (from CMTVERSION)
Linux (from uname)
Linux-i686 (from CMTCONFIG) package CMT implies [Linux]
A (From PROJECT)
Default (from Default)
foo (from package Foo)
```

2. *Implying a tag from another one using the tag association syntax*

   ```
   tag Linux foo

   > cmt show tags
   CMTv1 (from CMTVERSION)
   CMTr18 (from CMTVERSION)
   CMTp0 (from CMTVERSION)
   Linux (from uname) package Foo implies [foo]
   Linux-i686 (from CMTCONFIG) package CMT implies [Linux]
   A (From PROJECT)
   Default (from Default)
   foo (from package Foo)
   ```

3. *Through conventionally encoded values of* `CMTCONFIG`

   ```
   tag Linux-foo Linux foo

   > export CMTCONFIG=Linux-foo
   > cmt show tags
   CMTv1 (from CMTVERSION)
   CMTr18 (from CMTVERSION)
   CMTp0 (from CMTVERSION)
   Linux (from uname)
   Linux-foo (from CMTCONFIG) package Foo implies [Linux foo]
   A (From PROJECT)
   Default (from Default)
   Linux-i686 (from package CMT) package CMT implies [Linux]
   foo (from package Foo)
   ```

The current active tag set can always be visualized using the `cmt show tags` command.

```
> cmt show tags
CMTv1 (from CMTVERSION)
CMTr18 (from CMTVERSION)
CMTp0 (from CMTVERSION)
Linux (from uname)
Linux-i686 (from CMTCONFIG) package CMT implies [Linux]
A (From PROJECT)
Default (from Default)
> cmt -tag_add=tag1,tag2,tag3 show tags
CMTv1 (from CMTVERSION)
CMTr18 (from CMTVERSION)
CMTp0 (from CMTVERSION)
Linux (from uname)
Linux-i686 (from CMTCONFIG) package CMT implies [Linux]
A (From PROJECT)
tag1 (from arguments)
tag2 (from arguments)
tag3 (from arguments)
Default (from Default)
```

Typical usages of those extra tags are:

- 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)

All symbol definitions providing specific values triggered by the active selectors will be selected, such as in:

```
macro_append cppflags "" \
             debug    " -g "
```

---

11 - 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.

---

11. 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 always possible to extend the default understanding `CMT` gets on the package by appropriate modifiers (typically by overriding standard macros).

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.cxx FooB.cxx
```

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 had generated a simple *Makefile* (or `NMake` file) which is generaly sufficient for all standard operations, since `CMT` continuously and transparently manages the automatic reconstruction of all intermediate makefile fragments. We therefore simply and immediately execute gmake as follows:

```
...v1/cmt> gmake QUIET=1
------> (Makefile.header) Rebuilding constituents.make
------> (constituents.make) Rebuilding setup.make Linux-i686.make         [1]
setup.make ok
------> (constituents.make) Rebuilding library links
------> (constituents.make) all done
```

```
------> (constituents.make) Building Foo.make                          [2]
Library Foo
------> (constituents.make) Starting Foo
------> (Foo.make) Rebuilding ../Linux-i686/Foo_dependencies.make         [3]
rebuilding ../Linux-i686/FooA.o
rebuilding ../Linux-i686/FooB.o
rebuilding library
------> Foo : library ok
------> Foo ok
Installing library libFoo.so into /home/arnault/mydev/InstallArea/Linux-i686/lib
installation done                                                      [4]
------> (constituents.make) Foo done
 all ok.
Linux-i686.make ok
gmake[2]: ‘config’ is up to date.
gmake[2]: ‘all’ is up to date.
```

1. Some intermediate makefile fragments are automatically built to reflect the current effective set of Makefile macros deduced from the configuration (read from the `requirements` file). These fragments are automatically rebuilt (if needed) each time one of the `requirements` file changes.
2. Each component of the package (be it a particular *library* or a particular *executable*) will have its own *makefile* fragment (named `../${CMTCONFIG}/<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).
3. 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 (`dirs`) 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` .
4. An automatic installation mechanism is applied for all successfully built binaries.

or, for nmake:

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

This mechanism relies on some conventional *macros* and incremental *targets* used within the specific makefiles. Some are automatically generated, some have to be specified in user packages. It’s quite important to understand the list of possible customization macros, since this is the main communication medium between `CMT` and the package manager. See the complete table of those conventional macro when you want to interact with the standard CMT behaviour.

However, it is also possibble to use a simplified and platform independent form to build a constituent

```
...v1/cmt> cmt make
```

This syntax is identical on all platforms, and also does not require any `cmt config` nor `source setup` operation

---

11. 2 - Working on an application

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

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

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

```
> cmt make QUIET=1
------> (Makefile.header) Rebuilding constituents.make
------> (constituents.make) Rebuilding setup.make Linux-i686.make
setup.make ok
------> (constituents.make) Rebuilding library links
------> (constituents.make) all done
------> (constituents.make) Building Foo.make
Library Foo
------> (constituents.make) Starting Foo
------> Foo : library ok
------> Foo ok
installation done
------> (constituents.make) Foo done
------> (constituents.make) Building FooTest.make
Application FooTest
------> (constituents.make) Starting FooTest
------> (FooTest.make) Rebuilding ../Linux-i686/FooTest_dependencies.make
rebuilding ../Linux-i686/FooTest.o
rebuilding ../Linux-i686/FooTest.exe
------> FooTest ok
Installing application FooTest.exe into /home/arnault/mydev/InstallArea/Linux-i686/bin
installation done
------> (constituents.make) FooTest done
 all ok.
Linux-i686.make ok
gmake[2]: ‘config’ is up to date.
gmake[2]: ‘all’ is up to date.
```

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 " -lFoo " \
      WIN32        " $(FOOROOT)/$(Foo_tag)/Foo.lib "
...
```

The `Foo_linkopts` conventional macro will be automatically inserted within the `use_linkopts` macro. And the shared library location will be automatically set to the installation areas.

It is also possible to select extra tag sets when running gmake as follows (in this example we first cleanup the previous build and rebuild with debug options added to the compiler and linker commands) :

```
> cmt make clean
> cmt make CMTEXTRATAGS=debug
```

Like all other make macros used to build a component, the `Foo_linkopts` will be specified within the `requirements` which gives several benefits:

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

11. 3 - 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.

11. 4 - 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 `CMTPATH` environment variable).

In such contexts, it is often desirable that a given project 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 /ProjectB
```

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 VSUUI v3  Su/VSU
use VMM   v1
use VPC   v3
```

Then any user wanting 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 /ProjectB/MyProject/v1/cmt/setup.csh
```

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

---

12 - 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 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 all its default generators within its `requirements` file), allowing all its client packages to transparently acquire the capacity to generate documents of that sort.

`CMT` manages two categories of constituents:

1. *Applications* and *Libraries* are handled using pre-defined make fragments (mainly related with languages) and behaviour.
2. *Documents* offer a quite general framework for introducing completely new behaviours through user-defined make fragments. This includes actually generating documents, but also simply performing an operation (in which case sometimes no real *document* is produced).

In this section we only discuss the latter category and the following paragraphs explain the framework used for defining new document types.

The main concept of this framework is that each document to be generated or manipulated must be associated with a "document-type" (also sometimes named "document-style"), which corresponds to a dedicated make fragment of that name. Then, when specified in a `document` statement, this make fragment will be *instanciated* once or several times (typically once per source file) to construct a complete and functional make fragment, containing one main target. Both the resulting make fragment and the make target will have the name of the constituent.

`12. 1` - An example : the tex document-style

This section discusses one simple example (the production of postscript from latex files) available in the standard `CMT` distribution kit.

Converting a latex source file into a postcript output implies to chain two text processors, with an intermediate dvi format.

The fragment described here exactly performs this sequence, taking care of intermediate file deletion. The document style is named "tex" (the associated fragment shown here and named "tex" is actually provided by `CMT` itself, and can be looked at in `${CMTROOT}/fragments/tex` .) :

```
============ tex =====================================
${CONSTITUENT} :: ${FILEPATH}/${NAME}.ps

${FILEPATH}/${NAME}.dvi : ${FULLNAME}
        cd ${doc}; latex ${FULLNAME}

${FILEPATH}/${NAME}.ps : ${FILEPATH}/${NAME}.dvi
        cd ${doc}; dvips ${FILEPATH}/${NAME}.dvi

${CONSTITUENT}clean ::
        cd $(doc); /bin/rm -f ${FILEPATH}/${NAME}.ps ${FILEPATH}/${NAME}.dvi

======================================================
```

- They are declared in the `CMT` 's requirements file as follows :

  ```
  make_fragment tex -header=tex_header
  ```

  where:

  1. "tex" represents both the fragment name and the document style.
  2. the `-header=tex_header` option indicates that the generated makefile fragment will first include this header (which is actually an empty file in this case)

- A user package willing to apply this behaviour will have to include in its requirements file a statement similar to the following:

```
document tex MyDoc -s=../doc doc1.tex 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 `$CMTROOT/fragments/make_header` generic fragment
    2. Install a copy of the `$CMTROOT/fragments/tex_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’

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

MyDoc :: ../doc/doc1.ps

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

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

MyDocclean ::
        cd $(doc); /bin/rm -f ../doc/doc1.ps ../doc/doc1.dvi

MyDoc :: ../doc/doc2.ps

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

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

MyDocclean ::
        cd $(doc); /bin/rm -f ../doc/doc2.ps ../doc/doc2.dvi

clean :: MyDocclean
```

```
        cd .

MyDocclean ::
======================================================
```

---

12. 2 - How to create and install a new document style

This section presents the general framework for designing a document generator.

---

1. 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).
2. A fragment exactly named after the document style name must be installed into a subdirectory named `fragments` below the `cmt` branch of a given package (which becomes the *provider* package).
3. Optionally, two other fragments may be installed into the same subdirectory, one of them will be the *header* of the generated complete fragment, the other will be its *trailer*
4. Those fragments *must* be declared in the requirements file of the provider package as follows:

   ```
   make_fragment <fragment-name> [ options... ]
   ```

   where options may be :

`-suffix=<suffix>`	provide the suffix of the output files (without the dot)
`-header=<header>`	provide another make fragment meant to be prepended to the constituent's make fragment.
`-trailer=<trailer>`	provide another make fragment meant to be appended to the constituent's make fragment.
`-dependencies`	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 (ie a package *using* the provider), and queried upon using the command

```
> cmt show fragment <fragment name>
```

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

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

If a package re-defines an already declared make fragment, ie it provides a new copy of the fragment (possibly with new copies of the header and the trailer), and declares it inside its requirements file, then this package becomes the new provider for the document style.

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`	the 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)
`FILEPATH`	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}`

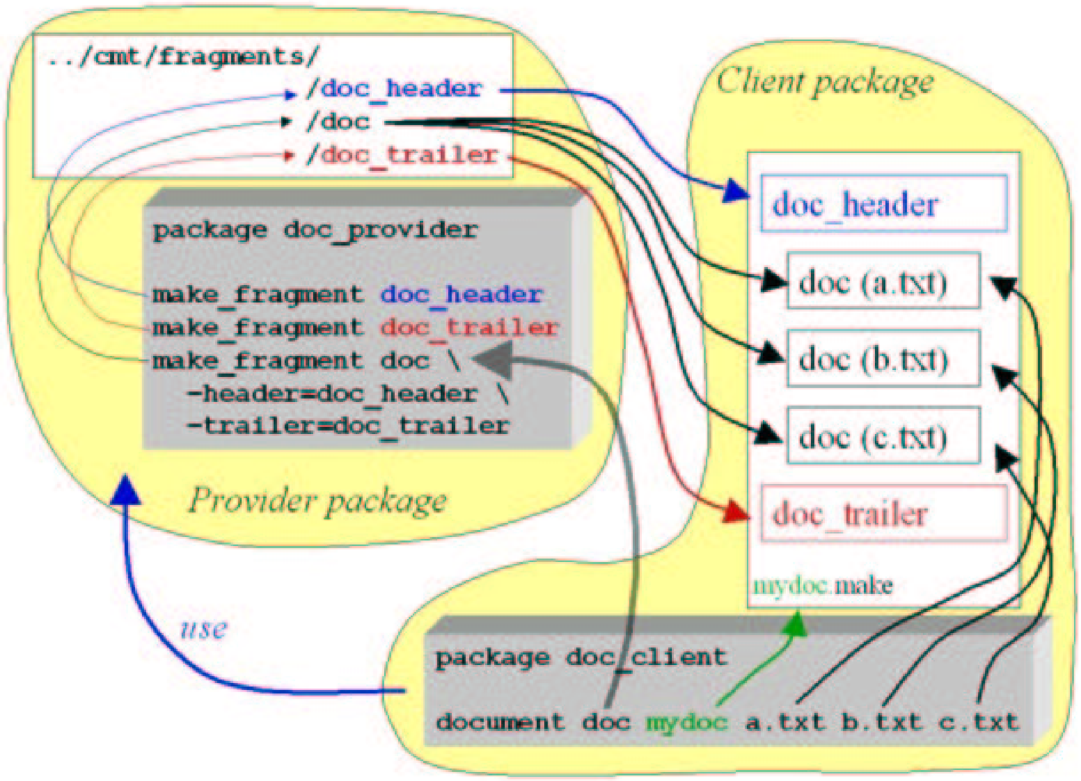
Templates must be enclosed between `${` and `}` or between `$(` and `)` and will be substituted at the generation time. Thus, if a fragment contains the following text :

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

then, the expanded constituent’s makefile will contain (refering to the "tex" example)

```
$(MyDoc_output)doc1.ps
```

Which shows that make macros may be dynamically generated.



3 - *The architecture of document generation.*

---

12. 3 - 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 \
          cd ${output}; $(agetocxx) -o ${NAME} -ohd ${FILEPATH} \
          -ohp ${FILEPATH} /tmp/${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) :: ${OBJS}
        $(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 -suffix=cxx -dependencies -header=agetocxx_header
```

---

13 - 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` branch 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 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.

---

13. 1 - The requirements file

---

13. 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 must be only followed by space characters).

Each statement is composed of words separated with spaces or tabulations.

The first word of a statement is the name 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 end of the current line.

The complete syntax specification is available in Appendix .

13. 2 - The concepts handled in the requirements file

13. 2. 1 - The package structuring style

13. 2. 2 - Meta-information : author, manager

The author and manager names

13. 2. 3 - package, version

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

13. 2. 4 - 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:

option	*validity*	*usage*
`-s=`*`directory`*	any	switch to a new default directory (1)
`-x=`*`regexp`*	any	specify an exclusion regular expression to be applied to the sources (1)
`-k=`*`regexp`*	any	specify a finer selection regular expression to be applied to the sources (1)
`-no_share`	libraries	do not generate the shared library
`-no_static`	libraries	do not generate the static library (*not yet implemented*)
`-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 (2)
`-import=<package>`	applications, libraries	explicitly import for this constituent the standard macros from a package that has the `-no_auto_imports` option set
`-target_tag`	any	construct a specific tag named `target_<constituent>` . This tag will only be active during the make session for this constituent. (4)
`-windows`	applications	When used in a Windows environment, generates a GUI-based application (rather than a console application)
`<var-name>=<var-value>`	any	define a variable and its value to be given to the make fragment (3)

1.

The sources of the constituents are generally specified as a set of file names with their suffixes, and are by default expected from the `../src` directory

```
library A A.cxx B.cxx
```

Then it is possible to change the default search location as well as to use a simplified wildcarding syntax:

```
library A -s=A *.cxx -s=B *.cxx
```

- `-s=A` means that next source files should be taken searched from `../src/A`
- `-s=B` means that next source files should be taken searched from `../src/B` . Note that this new specification is *not* relative to the previous `-s=A` but relative to the default search path `../src`
- `*.cxx` indicates that all files with a `.cxx` suffix in the current search path should be considered

It’s also possible to select or exclude files using regular expressions from general wildcarding techniques:

```
library A -s=A -x=[0-9] *.cxx -s=B -k=^B *.cxx
```

- The exclusion specification `-x=[0-9]` added to the statement will exclude all files from `../src/A` containing a *number* in their name.
- The selection specification `-k=^B` added to the statement will select files from `../src/B` strictly starting with the `B` letter.

2.

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
```

3.

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.

4.

For any constituent that has the `-target_tag` option set, a dedicated *tag* named `target_<constituent>` is automatically constructed by CMT. This tag becomes active during the construction of this constituent when using `make` , and therefore can be used as any other tag to select symbol values, or other configuration parameters.

---

13. 2. 5 - Groups

Groups permit the organization of the constituents that must be consistently built at the same development 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 examplify 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 controled 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 :

```
# Constituents belonging to the default all group

... constituents without group specification ...
library Foo *.cxx

# Constituents belonging to specific groups

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

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

in the context of the `Foo` package would rebuild *objy* related or *graphics* related constituents.

---

13. 2. 6 - 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 langage.

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 support is simply named `fortran90` and is declared by the following statement:

```
language fortran90 \
  -suffix=f90 -suffix=F90 \              [1]
  -linker=$(f90link) \                   [2]
  -preprocessor_command=$(ppcmd)
```

1. The recognized suffixes for source files will be `f90` and `F90`
2. The linker command used to build a Fortran90 application is described inside the macro named `f90link` (which must defined in this requirements file but which can also be overridden by clients)

The language support being named `fortran90` , two associated make fragments are expected, one under the name `fortran90` (for building application modules), the other with the name `fortran90_library` (for modules meant to be archived), both without extension.

These two fragments should be installed in the `fragments` sub-directory of the cmt 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` .

---

13. 2. 7 - Symbols

This is a generic concept supporting the notion of valued symbols. Several alternate semantics are implemented by these symbols, all specified using the same syntactic schema, but leading to different behaviours or interpretations by CMT:

- 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 prioritized *path* -like environment variable, which is supposed to be composed of search paths separated with colon characters `':'` (on Unix) or semi-colon characters `';'` (on Windows). It is generally recommended to construct such a variable by iteratively concatenating individual items one by one using `path_append` or `path_prepend`
- The `action` keyword is translated into a shell command definition, that can be activated using the `cmt do <action>` command or the associated `make` target.
- The `alias` keyword is translated into a shell alias definition,

Variants of these keywords are also provided for modifying already defined symbols. This generally happens when a package needs to modify (append, prepend or subtract) an inherited symbol (ie. which has been already defined by a used package).

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

All these definitions follow the same pattern:

symbol : *symbol-type symbol-name default-value* [*tag-expr value* ...]
symbol-type : *definition*
| *modification*
definition : macro
| set
| path
| action
| alias
modification : macro_prepend
| macro_append
| macro_remove
| macro_remove_regexp
| macro_remove_all
| macro_remove_all_regexp
| set_prepend
| set_append
| set_remove
| set_remove_regexp
| path_prepend
| path_append
| path_remove
| path_remove_regexp
tag-expr : *tag* [& *tag* ...]

- The symbol-name identifies the symbol.
- Values are generally quoted strings (using either simple or double quotes). They may be unquoted only if they are composed of one single non-empty word, since the general syntax parsing relies on space separated words.
- The default-value is mandatory (although it can be an empty string) optionally followed by a set of tag/value pairs, each representing an alternate value for this symbol.

- Each tag-value pair describes an alternate value to be used when the corresponding tag or tag-expression is active.

- When several alternate values are specified through several tag-value pairs the *first* matching condition is selected. Therefore one should always specify the most contraining condition first.

- The removal operations can be specified using either plain sub-strings or regular rexpressions. One should notice that even for the `path_remove_regexp` operation, full regular expression are expected rather than file-system wild carding syntaxes.

- The `path_remove` keyword is slightly specialized since it removes all individual search paths that *contain* the specified sub-string.

Be aware that there is only one name space for all kinds of symbols. Therefore, if a symbol was originally defined using a `macro` statement, using `set_append` to modify it will produce an undefined result (and a warning message).

The tag expression is used to select one alternate value to replace the default value, using the following matching rule:

- The first matching condition in the ordered list of alternate values is selected, ignoring the following ones
- A tag expression matches when all tags in the expression are active.

Examples of such definition are :

```
package CMT

macro cflags          "" \
      LynxOS-VGPW2    "-X" \
      insure          "-std1" \
      HP-UX           "+Z" \
      hp700_ux101     "-fpic -ansi" \
      alpha           "-std1" \
      alphat          "-std1" \
      SunOS           "-KPIC" \
      WIN32           '/nologo /DWIN32 /MD /W3 $(includes) /c'

macro pp_cflags       "" \
      LynxOS-VGPW2    "-DVGPW2" \
      HP-UX           "-D_HPUX_SOURCE" \
      alphat          "-DCTHREADS" \
      AIX             "-D_ALL_SOURCE -D_BSD" \
      Linux           "-Di586"

macro ccomp           "$(cc) -c $(includes) $(cdebugflags) $(cflags) $(pp_cflags)" \
      VisualC         "cl.exe $(cdebugflags) $(cflags) $(pp_cflags)"

macro clinkflags      ""

macro clink           "$(cc) $(clinkflags)" \
      VisualC         "link.exe /nologo /machine:IX86 "
```

13. 2. 7. 1 - actions

Actions are one of the possible symbols. Their definition as said previously follow the generic conventions for any symbol type, and they implement the concept of a generic shell command.

An example of a symple action:

```
action directory "ls $(dir_options)" WIN32 "dir $(dir_options)"
```

Like other symbols, actions can be visualized using the `cmt show actions` or the `cmt show action <name>` command.

Some specialized mechanims are available on actions, in order to execute in various ways the corresponding shell commands.

Actually two operating modes are supported:

1. Immediate mode

   This can be done via the `cmt do` command:

   ```
   > cmt <action-name>
   ```

   or, when the action name conflicts with a native CMT keyword,

   ```
   > cmt do <action-name>
   ```

   This mode immediately executes the specified command, after locally setting all environment variables known from the current package.

2. Through `make`

   ```
   > cmt make <action-name>
   ```

   - Actions are always associated with a make target of the same name
   - Action are always defined under a constituent group named `cmt_actions` . This means that action targets are never activated by default. Instead they must be explicitly called.
   - Action targets can be made dependent to other make targets (or vice versa), similarly to other constituents (libraries, applications, documents), using the `<name>_dependencies` macro.

     Example 1

     ```
     library A ...
     action B ...
     macro B_dependencies " A "
     ```

     In this example when doing `gmake B` , the library `A` will be rebuilt first.

     Example 2

```
library A ...
action B ...
macro A_dependencies " B "
```

In this example when doing `gmake A` (or simply `gmake`), the action `B` will be executed first.

---

13. 2. 8 - use

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

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

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 *offset* specification can be relative (i.e. on Unix it does not contain a leading '/' character). In this case, this offset is 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 :

```
use OnX v5r2
use CSet v2r3
use Gb v2r1

# A package installed in a sub-directory 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 .

When a `use` statement is in a `private` section, the corresponding used package will only be reached if when `CMT` operations occur in the context of the holder package. Otherwise (ie if the operation occurs in some upper level client package), then this *privately* used package will be entirely hidden. (*This behaviour follows a very similar pattern to the private or public inheritance of C++*). Suppose we have the following organization:

```
----------------
package A

use B v1
use D v1
----------------

----------------
package B

private
use C v1
use D v1
----------------
```

- all operations done in the context of package B will *see* both packages C and D
- all operations done in the context of package A will *see* both packages B and D, but not package C

---

13. 2. 9 - patterns

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 statements are to be associated with a given pattern, they will be separated with the `" ; "` separator pattern (beware of really enclosing the `;` character between two space characters).

The general syntax for defining a pattern in a requirements file is:

pattern : pattern [-global] *pattern-name* *cmt-statement*
[; *cmt-statement* ...]

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
`project`	the project name of the current package

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

User defined templates that have not been assigned a value when the pattern is applied are simply ignored (ie. replaced with an empty string).

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).

---

13. 2. 9. 1 - Applying a pattern

According to whether the `-global` qualifier was used in the pattern definition, the aplication mode will be completely different.

1. *Normal patterns*

   Such patterns must be applied explicitly using the `apply_pattern` construct

   Doing so, it is possible to specify custmization values for user defined template parameters

   ```
   pattern TA macro <base>AAA "AAA"

   apply_pattern TA base=abc
   apply_pattern TA base=def
   ```

   In the `apply_pattern` syntax, it is even possible to simply *ommit* the keyword itself, and thus using the pattern name as a plain CMT keyword. The previous example becomes:

   ```
   TA base=abc
   TA base=def
   ```

   This can be seen as a way to *extend* the CMT language. Notice that there is a risk of a conflict between the primary CMT keywords and pattern names then. Suppose that a pattern name is defined to be exactly a primary CMT keyword. In this case, the syntax parser will always understand this name as the CMT primary keyword, and thus won’t override the original syntax. When this (not recommended) situation occurs, it is therefore required to use the full notation with an explicit `apply_pattern` keyword, so as to avoid any possible ambiguity.

2. *Global patterns* (ie when the `-global` qualifier is used

   In this case, the pattern is automatically applied to *all* packages that effectively see the pattern definition, which includes all clients of the package defining the pattern.

Another consequence of the automatic application of the pattern, is that it is not possible to give values to parameters. Therefore it is not recommended to design global patterns with user defined parameters.

Conversely it is possible to inhibit the automatic application of a global pattern in a particular package by using the following statement:

```
ignore_pattern <name>
```

`13. 2.10` - cmtpath_patterns

These patterns act quite similarly to the *global* patterns previously described, ie they defines a set of CMT statements to be applied in a generic way. The difference is that instead of being applied to *packages* , they are automatically applied to all entries in the `CMTPATH` list.

Only few system parameters can be used here:

- `<path>` which stands for any entry in the CMTPATH list.
- `<project>` which stands for the project name associated with an entry in the CMTPATH list.

As an example suppose we define

```
path        CMTPATH "/ProjectA"
path_append CMTPATH "/ProjectB"

cmtpath_pattern \
  macro_prepend pp_cppflags " -I<path>/InstallArea/include "
```

this will assemble one `-I` option (towards the preprocessor) per entry in CMTPATH, implementing a mechanism for a multiple installation area for header files. In the example above the resulting macro will be

```
 -I/ProjectA/InstallArea/include -I/ProjectB/InstallArea/include
```

This can be combined with the standard and automatic macros (automatically setup for all used packages)

```
<package>_cmtpath
<package>_offset
```

which provide the CMTPATH entry and the directory offset in this CMTPATH for all used packages.

`13. 2.11` - 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` .

---

13. 2.12 - Strategy specifications

Users can control the behaviour of `CMT` through a set of strategy specifications. The current implementation provides such control over several aspects :

1. The build strategy

   This controls some aspects of the building process.

   The following keywords are available:

`prototypes`	C source files will automatically produce a header file containing a prototype of all global entry points
`no_prototypes`	No production of automatic prototype header files for C sources
`with_installarea`	The installation area mechanisms are activated. This implies applying the cmtpath_patterns that may be defined (eg in CMT itself)
`without_installarea`	The installation area mechanisms are inhibited

2. The setup strategy

   This controls various actions that may be performed during the sourcing of the setup scripts.

   The following keywords are available:

`config`	An environment variable <PACKAGE>CONFIG will be generated for all packages in the dependency chain
`no_config`	The <PACKAGE>CONFIG environment variable is not generated
`root`	An environment variable <PACKAGE>ROOT will be generated for all packages in the dependency chain
`no_root`	The <PACKAGE>ROOT environment variable is not generated
`cleanup`	The automatic cleanup operation to the current installation area is launched
`no_cleanup`	The automatic cleanup operation to the current installation area is not launched

The strategy specifications are setup on a per-project basis. This means that they are generally applicable to all packages of a given sub-project, and can be oerridden in other sub-projecs of the same software base.

Every strategy setting defines two mutually exclusive tags and activates one of them.

```
<project>_<have_item>
<project>_<have_not_item>
```

Examples

```
<project>_prototypes
<project>_no_prototypes
<project>_with_install_area
<project>_without_install_area
<project>_config
<project>_no_config
<project>_root
<project>_no_root
<project>_cleanup
<project>_no_cleanup
```

`13. 2.13` - 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`).

`13. 2.14` - include_path

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

Specifying the value *none* (a reserved CMT keyword) means that no default include search path is expected from CMT, and thus no `-I` compiler option will be generated by default (generally this means that user include search paths should be specified via explicit `include_dirs` instead).

`13. 2.15` - include_dirs

Add explicit specifications for include access paths.

`13. 2.16` - 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 :

1. some are provided by `CMT` itself (they correspond to its internal behaviour)
2. others handle the language support
3. and the last serve as specialized document generators.

The fragments defined in `CMT` can be:

- those used to construct the application or library constituents. Their semantic is standardized (they are all associated with a `language` statement in the CMT requirements file).

---

c c_library cpp cpp_library lex lex_library fortran fortran_library yacc yacc_library jar jar_header java java_copy java_header check_java cleanup_java

---

- those used internally by CMT as primary building blocks for assembling the makefile. (Generally developers should not see them).

---

cleanup_objects application constituent application_header constituents_header buildproto protos_header os9_header dependencies check_application dependencies_and_triggers check_application_header document_header library cleanup library_header cleanup_application library_no_share cleanup_header make_header cleanup_library

---

- some document generators which *may* be used if needed, but are not mandatory:

---

installer installer_header readme readme_header readme_trailer readme_use dvi tex generator generator_header

---

- those used to generate configuration files for MSVisualC++:

---

dsp_windows_header dsw_all_project dsw_all_project_dependency dsw_all_project_header dsw_all_project_trailer dsw_header dsw_project dsw_trailer dsp_all dsp_application_header dsp_contents dsp_library_header dsp_shared_library_header dsp_trailer

---

Language fragments should provide two forms, one for the applications (in which case they are named exactly after the language name eg c, cpp, fortran) and the other for the libraries (in which case they have the `_library` suffix (eg. c_library, cpp_library, fortran_library). 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. Event if the make fragment meant to be the implementation of a language support is declared, the language support itself must be declared too, using the `language statement`

All make fragments are provided as (suffixless) files which must be located in the `fragments` sub-directory inside the cmt/mgr branch of one package. They must also be declared in the requirements file (through the `make_fragment` statement) 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` freely override any make fragment provided by `CMT` itself (although in this case a deep understanding of what the original fragment does is recommended).

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

`13. 2.17` - public, private

The `public` or `private` keywords introduce sections containing *public* or *private* statements. This concerns:

- the definition of symbols
- the specification of use relationships
- the declaration of make fragments
- the declaration of patterns

Public definitions are meant to be exported to any client of the package whereas private ones are only available for the package *developper* ie. when the current directory is within the package itself.

Public use relationships expose the complete sub-tree to the package clients, whereas private ones entirely hide the sub-tree, expanding it only when the operator really acts from within the context of the package. It should be noticed that private use relationships are completely unvisible from clients, which implies that none of the definitions (not only symbols) will be set.

However, the `cmt broadcast` and `cmt show uses` commands are configured to always ignore the private specification and therefore will always traverse the sub-trees whether they are public or private (in order to ensure the hierarchy dependencies)

---

13. 2.17. 1 - Scoping sections

By using the `public` or `private` keywords, one defines *scoping* sections. This sections continues until:

- another scoping statement is found, which simply switch to this new mode
- an `end_private` or `end_public` keyword is found, in which case the scoping mode is reset to the state prior to the previous matching `private` or `public` statement. This latter mechanisms permits in particular to define autonomous scoping sections within `pattern`s .

By default `cmt` commands operate according to the scoping specifications found in the requirements files of the reachable packages. Ie. in the current package all statements are considered whether being prublic or private, while in used packages, only public statements are considered.

This standard behaviour though is not applied when running `cmt broadcast` or `cmt show uses` , and in this case all statements public or private, are always considered, even in used packages.

However it is always possible to override the default behaviours by using the `-private` or `-public` modifier to the `cmt` command:

- `-private`

  Force the command to consider all definitions even those installed in private sections

- `-public`

  Force the command to really mask the private sections

---

13. 2.18 - tag, apply_tag

The `tag` keyword provides tag definitions, while the `apply_tag` keyword *activates* a tag.

A tag is a token which can be used to select particular values of symbols.

Some tags are automatically constructed by CMT according to its knowledge of the context (see this section for more details), but they may be also defined within a requirements file as follows :

```
tag Foo                [1]
tag Bar Foo FooA FooB  [2]
apply_tag Bar          [3]
```

1. This simply declares a tag. This does not activate it by default

2. This construct declares that the tags `Foo` , `FooA` and `FooB` will become active if `Bar` becomes active. Note that this statement implicitly *declares* `FooA` and `FooB`
3. This activates the `Bar` tag. Tags that have been associated with it (in [2]), will all become active as well.

Running the setup script (through the *source setup.[c]sh* or *call setup.bat* command) can also receive tag specifications using the *-tag=tag-list* options.

---

13. 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 :
   none
   awk
   broadcast [-select=list] [-exclude=list] [-local] [-global] [-begin=pattern] [-depth=n] <comm
        apply a command to [some of] the used packages
   build <option>          : build actions. (Try cmt help build)
   build constituent_makefile <constituent>  : generate constituent Makefile fragment
   build constituents_makefile : generate constituents.make
   build dependencies      : generate dependencies
   build library_links     : build symbolic links towards all imported libraries
   build make_setup        : build a compiled version of setup scripts
   build msdev             : generate MSDEV files
   build CMT_pacman        : generate PACMAN manifest file for CMT
   build vsnet             : generate VS.NET files
   build os9_makefile      : generate Makefile for OS9
   build prototype         : generate prototype file
   build readme            : generate README.html
   build tag_makefile      : generate tag specific Makefile
   build temporary_name    : generate a name for a temprary file
   build triggers <constituent> : generate library trigger file
   build windefs <library_name> : generate def file for Windows shared libraries
   check <option>          : check actions. (Try cmt help check)
   check configuration     : check configuration
   check files <old> <new> : compare two files and overrides <old> by <new> if different
   check version <name>    : check if a name follows a version tag syntax
   co | checkout           : 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
   create_project <project>  : create and configure a new project
   cvsbranches <module>      : display the subdirectories for a module
   cvssubpackagess <module>  : display the subpackages for a module
   cvstags <module>          : display the CVS tags for a module
   do <action> [<param>=<value>] ... : Execute an action
   expand model <model>    :
   filter <in> <out>       : filter a file against CMT macros and env. variables
   help | -help | --help   : display this help
   load
   lock [<p> <v> [<path>]] : lock a package
   remove <package> <version> [<path>] : remove a package version
   remove library_links    : remove symbolic links towards all imported libraries
   run ’<command>’         : apply a command
```

```
  run_sequence <sequence file> : execute a cmt equence file
  set version <version>   : generate a version file in the current package
  set versions            : generate version files into packages
  setup [-csh|-sh|-bat]   : generate a setup script
  show <option>           : query actions. (Try cmt help show)
  show  action <name>     :  a formatted action definition
  show  action_value <name> :  a raw action definition
  show  action_names      :  all action names
  show  actions           :  all action definitions
  show  all_tags          :  all defined tags
  show  applied_patterns  :  all patterns actually applied
  show  author            :  package author
  show  branches          :  added branches
  show  clients           :  package clients
  show  cmtpath_patterns  :  cmtpath_patterns
  show  constituent <name>:  constituent definition
  show  constituent_names :  constituent names
  show  constituents      :  constituent definitions
  show  cycles            :  cycles in the use graph
  show  fragment <name>   :  one fragment definition
  show  fragments         :  fragment definitions
  show  groups            :  group definitions
  show  include_dirs      :
  show  language <name>   :  language definition
  show  languages         :  language definitions
  show  macro <name>      :  a formatted macro definition
  show  macro_value <name>  :  a raw macro definition
  show  macro_names       :  all macro names
  show  macros            :  all macro definitions
  show  manager           :  package manager
  show  packages          :  packages reachable from the current context
  show  path              :  the package search list
  show  pattern <name>    :  the pattern definition and usages
  show  pattern_names     :  pattern names
  show  patterns          :  the pattern definitions
  show  projects          :  project definitions
  show  setup             :  setup definitions
  show  pwd               :  filtered current directory
  show  set <name>        :  a formatted set definition
  show  set_names         :  set names
  show  set_value <name>  :  a raw set definition
  show  sets              :  set definitions
  show  strategies        :  all strategies (build & version)
  show  tags              :  all currently active tags
  show  use_paths <pack>  :  all paths to the used package
  show  uses              :  used packages
  show  version           :  version of the current package
  show  versions <name>   :  visible versions of the selected package
  system                  : display the system tag
  unlock [<p> <v> [<path>]] : unlock a package
  version                 : version of CMT
global options :
  -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-list>         : select a new tag-set
  -tag_add=<tag-list>     : add specific comma-separated tag(s)
  -tag_remove=<tag-list>  : remove specific comma-separated tag(s)
```

```
-with_version_directory : reset to default structuring style
-without_version_directory : switch structuring style
-cleanup                : activate install area cleanup
-no_cleanup             : inhibit install area cleanup
```

The following sections present the detail of each available command.

---

13. 3. 1 - cmt broadcast

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 throught the current `CMTPATH` 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 occurence 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. These selection criteria may be combined (eg one may combine the *begin* and *select* modifiers)

```
sh> cmt broadcast -begin=Cm gmake                 (1)
sh> cmt broadcast -select=Cm gmake                (2)
sh> cmt broadcast -select='/Cm/ /CSet/' gmake     (3)
sh> cmt broadcast -select=Cm -exclude=Cmo gmake   (4)
sh> cmt broadcast -local gmake                    (5)
sh> cmt broadcast -depth=<n> gmake                (6)
sh> cmt broadcast -global gmake                   (7)
sh> cmt broadcast -all_packages gmake             (8)
```

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

1. the first package which path contains the string `"Cm"` , and then all other reachable packages (in case other specifiers are used)
2. the packages which path contains the string `"Cm"`
3. the packages which path contains either the string `"/Cm/"` or the string `"/CSet/"`
4. the packages which path contains the string `"Cm"` , but which does not contain the string `"Cmo"`
5. the packages at the same level as the current package
6. the packages at the same level as the current package or among the <n> first entries in the `CMTPATH` list
7. the packages at any level of the CMTPATH search list

8. all the packages and versions currently available through the `CMTPATH` list

---

13. 3. 1. 1 - Specifying the shell command

A priori any Unix or DOS shell command can be specified in a boadcast command. However, it’s important to understand the order of the various parsing actions:

1. The current shell first parses the complete command line
2. CMT catches all possible options given to the broadcast command itself
3. CMT then gets the rest of the command line and makes it the shell command to be executed during the broadcast scan.
4. This command line may be subject to template substitution (see below) by CMT
5. Eventually the command line is passed to the local shell (which may then perform additional parsing actions)

Considering this complex sequence of parsing, it may be appropriate to selectively enclose the shell command passed to the broadcast action into quotes. It may even be sometimes useful to have two levels of quotes

---

13. 3. 1. 2 - Templates in the shell command

Similarly to what exists in the pattern mechanism, some standard *templated* values can be embedded inside the command to be executed by the broadcast action. They take a standard form of `<template-name>` . These templates acquire their value on each package effectively reached during the broadcast scan, and the effective value is substituted before launching the command. The possible templates are:

`<package_cmtpath>`	The element in the CMTPATH search list where the package has been found
`<package_offset>`	The directory offset to cmtpath
`<package>`	The package name
`<version>`	The version of the package

The next example shows a typical broadcast command listing the header files as expected in the conventional location `../<package>` :

```
> cmt broadcast ’ls ../<package>’
[...]
#--------------------------------------------------------------
# Now trying [ ls ../GenzModuleEvent] in /.../GenzModuleEvent/.../cmt (149/609)
#--------------------------------------------------------------
CVS  KineHepMcmap.h
#--------------------------------------------------------------
# Now trying [ ls ../Tauola_i] in /.../Tauola_i/.../cmt (150/609)
#--------------------------------------------------------------
CVS     Jaki.icc        Tauola_i.h    Taurad.h    config.h    rn_tau.h
```

```
Jaki.h  ReadPDGtable.h  Tauola_i.icc  Taurad.icc  polhep.inc  tauola_cblk.inc
#--------------------------------------------------------------
# Now trying [ ls ../NavigationEvent] in /.../NavigationEvent/.../cmt (151/609)
#--------------------------------------------------------------
CVS  INavigable.h  INavigationCondition.h  INavigationSelector.h
INavigationToken.h  NavigationToken.h
[...]
```

One should note that when templates are used in a broadcast command, it's important to enclose the command in quotes so as to inhibit any possible parsing of the <> syntax by the shell.

13. 3. 2 - cmt build <option>

The actions associated with the build options are generally meant for internal use only, and users will rarely (if ever!) apply them manually

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.

- `msdev`

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

- `vsnet`

  This command generates workspace and project files required for the Visual.net 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 :

  ```
  sh> 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 controled 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 controled 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.

---

13. 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.

---

13. 3. 4 - cmt check files <reference-file> <new-file>

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

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

---

13. 3. 5 - cmt checkout ...

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

---

13. 3. 6 - cmt co ...

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

---

13. 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.

---

13. 3. 8 - cmt config

This command (re-)generates the setup scripts and the manimal 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 a package (where the requirements file can be found).

This command also performs some cleanup operations (eg. removing all makefile fragments previously generated). Generally speaking, one may say that this command restores the CMT-related files to their original state (ie before any document generation)

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 changed the directory structure of a package (using a manual copy operation, or renaming one of its parent directory, such as the version directory)

It should be noted however that when using *actions* , such as the one defined by default to launch `make` tools neither `cmt config` nor `source setup` are required any longer.

---

13. 3. 9 - cmt create <package> <version> [<area>]

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

```
csh> cmt create Foo v1
```

or:

```
csh> cmt create 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:

- A `src` and a `cmt` branch
- A very minimal requirements file
- The setup and cleanup shell scripts
- The minimal Makefile

`13. 3.10` - cmt expand model [-strict] <model-string>

This command produces on the standard output an *expansion* of the model string given in the argument.

The expansion consists in:

- Expanding macros referenced in the model string using the standard notations `$()` or `${}` or `%%`

  ```
  > cmt expand model "abcd $(CMTVERSION) efgh"
  abcd v1r18p20041201 efgh
  ```

- *Recursively* expanding text files with parameters. The model string must then take a conventional XML-based syntax:

  ```
  > cmt expand model "text <file-name parameter=value ... /> text ..."
  ```

  or

  ```
  > cmt expand model -strict "text <cmts:file-name parameter='value' ... /> text ..."
  ```

  or

  ```
  > cmt expand model -strict "text <cmtv:file-name parameter='v1 v2 v3 ...' ... /> text ..."
  ```

  Where:

  - *file-name* is the name of a file declared using the `make_fragment` statement
  - `parameter =value` specifies the value of a parameter that will be substituted in the file when referenced using the `$(parameter )` notation. Several such values may be specified in one model string
  - the `-strict` form is useful to handle XML files, and file names must be prefixed (*namespaced*) by a special keyword `cmts:` or `cmtv:` .

`cmts:`	Perform a unique substitution over one copy of the file
`cmtv:`	Perform a multiple substitution over N copies of the file, taking the N space-separated values specified for the parameters

The following examples will explain some of the mechanisms.

We consider `A` containing:

```
A$(P1)B$(P2)C
```

And `B` containing:

```
i<cmts:A P1=’j’ P2=’${P2}’/>k
```

```
> cmt expand model "abcd <A P1=XXX/> efgh"                                     [1]
abcd AXXXBC efgh
> cmt expand model -strict "abcd <cmts:A P1=’XXX’ P2=’YYY’/> efgh"             [2]
abcd AXXXBYYYC efgh
> cmt expand model -strict "abcd <cmts:A P1=’XXX’/> <cmts:B P2=’YYY’/> efgh"   [3]
abcd AXXXBC iAjBYYYCk efgh
> cmt expand model -strict "abcd <cmtv:A P1=’X Y Z’ P2=’YYY’/> efgh"           [4]
abcd AXBYYYCAYBCAZBC efgh
> cmt expand model -strict "abcd <cmtv:A P1=’X Y Z’ P2=’\"\" <cmt:null/> ZZZ’/> efgh"  [5]
abcd AXBCAYBCAZBZZZC efgh
...
```

1. A simple expansion using the non-strict model syntax. `P1` is substituted, but `$(P2)` becomes empty
2. The same using the strict model. Here P2 is valued.
3. A more complex model, using a recursive expansion described in `B` showing how parameter values are transmitted
4. A model showing the multiple expansion. Here `P1` receives 3 values. `P2` has only one value. The largest vector (3 values) dictates the number of copies. smaller vectors are completed by empty values.
5. A model showing the multiple expansion with empty values in a vector. Here `P1` receives 3 values. `P2` has also 3 values, but only one non-empty. This examples show the two possible means of specifying empty vector values: using either `\"\"` or the reserved keyword `<cmt:null/>` .

- The file expansion is recursive. This means that files specified in model elements will themselves be considered as model texts (ie following the same syntax), and be expanded in turn. This processus is entirely recursive, with no limit to the depth. The -*strict* option, when selected is propagated during the recursion.

`13. 3.11` - cmt filter <in-file> <out-file>

This command reads <in-file>, substitutes all occurences 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 instanciating make fragments. Then, users may use it for any kind of document, including maual generation of MSDev configuration files, etc...

---

13. 3.12 - cmt help | --help

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

---

13. 3.13 - 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 ../*"
   ```

---

13. 3.14 - 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 whic 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.

---

13. 3.15 - 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`

---

13. 3.16 - cmt run [shell-command]

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

In particular all environment variables defined in requirements file are first set before running the command. This may be seen as a generic application launcher.

This may be combined with the global options `-pack=`*`package`* , `-version=`*`version`* , `-path=`*`access-path`* , to give a direct access to any package context.

---

13. 3.17 - cmt set version <version>

This command creates and/or fills in the `version.cmt` file for a package structured without the version directory.

This command has no effect when run in the context of a package structured *with* the version directory

This command must be run while being in the context of one CMT package.

---

13. 3.18 - cmt set versions

This command applies recursively the `cmt set version ...` command onto all used packages using a broadcast operation.

Packages reached during the broadcast scan acquire their version from the original use statement. This is this specified version which will be stored inside the `version.cmt` files

---

13. 3.19 - 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.

---

13. 3.20 - cmt show <option>

- `all_tags`

  This command displays all currently defined tags, even when not currenty active

- `applied_patterns`

  This command displays all patterns actually applied in the current package

- `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.

Note that the search on clients is *not* performed recusively. Thus only clients explicitly using the specified package will be displayed.

- `constituent_names`
- `constituents`
- `cycles`

  This command displays all cycles in the use graph of the current package. Although CMT smoothly accepts such cycles, it is generally a bad practice to have cycles in a use graph, because CMT can never decide on the prefered entry point in the cycle, leading to somewhat unpredictable results, eg in constructing the `use_linkopts` macro.

- `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`

  Display the effective location of all declared make fragments

- `groups`

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

- `languages`

  Display al languages declared using the `language` keyword

- `macro <name>`
  `set <name>`
  `action <name>`

  This set of commands displays a quite detailed explanation on the value assigned to the symbol (macro, set or action) specified as the additional argument. It presents in particular each intermediate assignments made to this symbol 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>`
  `set_value <name>`
  `action_value <name>`

  This set of commands displays the raw value assigned to the symbol (macro, set or action) 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`
  `sets`
  `actions`

  This set of commands extracts from the `requirements` file(s) the complete set of symbol (macro, set or action) definitions, selects the appropriate *tag* definition (or uses the one provided in the `-tag=<tag>` option) and displays the effective symbol 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 atcually 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.

- `projects`

  This command displays the current knowledge of sub-project definitions and settings. It shows the project names and their location (ie the corresponding item in `CMTPATH`

  ```
  > cmt show projects
  PA (in C:\Arnault\test\tprojects\PC) (current)
  Project1 (in C:\Arnault\test\tprojects\PB)
  PA (in C:\Arnault\test\tprojects\PA\1.1)
  Project2 (in C:\Arnault\test\tprojects\P0)
  CMT (in C:\Arnault)
  ```

- `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).

- `setup`

  This command combines in one go the output of:

  ```
  > cmt show uses
  > cmt show tags
  > cmt show path
  ```

- `strategies`

- `tags`

  This command displays all currently *active* tags, and what part of the configuration actually activates them

- `uses`

This command displays the use graph fo the current package. Private sections of used packages are reached and considered. This behavior can be changed to effectively hide the private sections in used packages by using the `-public` modifier

```
> cmt -public show uses
```

A typical output produced by this command is:

```
> cmt show uses
# use GaudiPolicy v*                             [1]
# use GaudiKernel v*
#   use GaudiPolicy v5r*                         [2]
# use CLHEP v*  (native_version=1.8.2.0)         [3]
#   use ExternalLibs v4r*
#
# Selection :                                    [4]
use CMT v1r18p20041201 (/afs/cern.ch/sw/contrib)
use ExternalLibs v4r2p0  (/afs/cern.ch/atlas/offline/external/Gaudi/0.12.1.5)    [5]
use CLHEP v2r1820p0  (/afs/cern.ch/atlas/offline/external/Gaudi/0.12.1.5)
use GaudiPolicy v5r11p2  (/afs/cern.ch/atlas/offline/external/Gaudi/0.12.1.5)
use GaudiKernel v13r5p1  (/afs/cern.ch/atlas/offline/external/Gaudi/0.12.1.5)
```

1. The first section of the display (up to the `Selection` keyword) displays the hierarchical use graph.

   Use statements as specified in the requirements files are displayed, rather than the result of the effective selection performed by CMT

2. Sub-uses are expanded only once and indented according to the depth in the graph

3. Various precisions on the use statements are shown in the first section, such as the scoping section, the `-no_auto_imports` modifier, and the *native version* of this package, (when a `<package>_native_version` macro has been defined)

4. The second section shows the effective ordered set of use statements resolved by CMT according to the combined use specifications.

5. On every line the effective location of the found package is displayed.

The `-quiet` option may be used to remove the first section from the output so as to only display a simple list of used packages, starting from the deepest uses.

- `use_paths <target-package>`

  This command displays all possible paths between the current package and the specified used target package.

  In particular this will detect if a package has no access to another one, due to private use statements

- `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.

---

13. 3.21 - cmt system

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

---

13. 3.22 - 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 ../*"
```

---

13. 3.23 - cmt version | --version

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

---

13. 3.24 - cmt cvstags <module>

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

---

13. 3.25 - cmt cvsbranches <module>

---

13. 3.26 - cmt cvssubpackages <module>

---

13. 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.

---

13. 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 whereever 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 behavious to this feature)

---

14 - 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 following a recipe explained in the dedicated appendix .

---

14. 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.

14. 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 *chaeckout* 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.
-o offset Offset in the CVS repository
-n        Simulation mode on
-v        Verbose mode on
-help     Print this help
```

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
```

14. 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 repositiory) can be achieved through the following command:

```
sh> (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 :

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

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

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

14. 4 - Working on a package, creating a new release

This section presents the way to instanciate 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 :

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

4. configure this new release, and rebuild it :

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

---

14. 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
```

---

15 - Interfacing an external package with CMT

Very often, external packages (typically commercial products, or third party software) are to be used by packages developped in the context of the `CMT` environment. Although this can obviously 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,
- adapt the version tag convention consistently to the project, hiding the version tag specificities of eg. commercial packages.
- provide compiler options using the 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 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) $(Htmlo_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.

---

16 - The installation area mechanism

CMT proposes and implements a flexible architecture for installation areas, meant to group the results of the build process or any other information belonging to packages into shared disk spaces. The typical usage of such installation areas is classical and expect to make only visible to the clients of a given (sub-)project the results of the build process while hiding the details of the package sources.

the basics of the mechanisms supported by CMT are the following:

1.

   All mechanisms are customizable on a per-project basis, so as to easily follow the project specific conventions

2.

However CMT proposes a minimal default behaviour based on the concrete experience in large projects, as well as frequently met practices

3. A typical well supported convention is to map the set of installation areas onto the set of CMTPATH entries, associating the concept of CMTPATH splitting with the sub-project organization

4. A typical consequence of this approach is that many configuration parameters need to be set according to the list of CMTPATH items. Eg on a Unix system, if one expects to find shared libraries in every installation area, each of them being created in a corresponding CMTPATH entry, one also expects to have LD_LIBRARY_PATH entries accordingly. The mechanism of `cmtpath_pattern` is exactly designed for that.

5. The mechanism easily supports the extension for installing binary files (libraries, applications, java classes), runtime files, documentation and header files.

---

16. 1 - The default implementation

It is provided in terms of

1. A set of `cmtpath_pattern` s defined in the CMT requirements file. This can be displayed using the command

   ```
   > cmt show cmtpath_patterns
   ```

2. A consistent set of actions added to the following make_fragments

application	applications
library	shared libraries
library_no_share	static libraries
java_header	Java applications
jar	Java libraries

3. One shell script for installing or uninstalling files or directories

```
${CMTROOT}/mgr/cmt_install_action.sh
${CMTROOT}/mgr/cmt_uninstall_action.sh
${CMTROOT}/mgr/cmt_install_action.bat
${CMTROOT}/mgr/cmt_uninstall_action.bat
```

The default architecture of this installation scheme is by default set for each CMTPATH entry to:

```
<path>/InstallationArea/$(tag)/bin/...                [1]
                       /$(tag)/lib/...                [2]
                       /include/<package>/...         [3]
                       /share/bin/...                 [4]
                       /share/lib/...                 [5]
                             /...                     [6]
                       /doc/<package>/...             [7]
                           /...                       [8]
```

1. Platform dependent executables
2. Platform dependent libraries
3. Public header files
4. Platform independent applications (eg Java applications)
5. Platform independent libraries (eg Java libraries)
6. other platform independent files
7. package specific documentations
8. project-wide documentation

The cmtpath_patterns are designed in this implementation for constructing a proper and consistent sequence of system specific environment variables (eg PATH, LD_LIBRARY_PATH, CLASSPATH) as well as compiler or linker options so as to transparently refer to the installation area only when it is appropriate to ovverride the local patterns.

---

16. 2 - Tuning the installation area mechanisms

First of all every individual sub-project may activate or inhibit the installation area mechanisms using the `build_strategy` statement, with either `with_installarea` or `without_installarea` option.

Then a dedicated tag materializes the selected strategy:
`<project>_with_installarea` or `<project>_without_installarea`

This tag set will be used in various macro or set definitions to produce or not the appropriate values

CMT manipulate some standard macros or environment variables according to the effective strategy:

name	*purpose*	*default*
`cmt_installarea_prefix`	The default prefix for all projects	`InstallArea`
`<project>_installarea_prefix`	The prefix for a given sub-project	`$(cmt_installarea_prefix)`
`<project>_installarea_prefix_remove`	The regexp pattern to cleanup symbols	`$(<project>_installarea_prefix)`
`cmt_installarea_linkopts`	Implicit linker options due to the installation area	...
`PATH`	Accessing the executables in the installation area	...
`LD_LIBRARY_PATH`	Accessing the shared libraries in the installation area	...
`CLASSPATH`	Accessing the jar files in the installation area	...

---

17 - Installing CMT for the first time

These sections are of interest only if `CMT` is not yet installed on your site, of 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.cmtsite.org . 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` .

17. 1 - Installing CMT on your Unix site

The very first operation after dowloading `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:

```
sh> gmake cpp=CC
```

The `cppflags` macro can also be used to override the 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.cmtsite.org
```

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> tar xzf CMTv1r18p20041201.tar.gz
csh> cd CMT/v1r18p20041201/mgr
csh> ./INSTALL
csh> source setup.csh
csh> gmake
```

17. 2 - Installing CMT on a Windows or Windows NT site

You first have to fetch the distribution kit from the Web at http://www.cmtsite.org . 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 interactively configures 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` ("v1r18p20041201" 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\v1r18p20041201\mgr
dos> call INSTALL
dos> call setup.bat
dos> nmake /f nmake
```

18 - Appendices

18. 1 - Copyright

Copyright LAL and Christian Arnault LAL-Orsay CNRS

`arnault@lal.in2p3.fr`

This software is a computer program whose purpose is to describe and manage software configuration activities.

This software is governed by the CeCILL license under French law and abiding by the rules of distribution of free software. You can use, modify and/ or redistribute the software under the terms of the CeCILL license as circulated by CEA, CNRS and INRIA at the following URL

`http://www.cecill.info`

As a counterpart to the access to the source code and rights to copy, modify and redistribute granted by the license, users are provided only with a limited warranty and the software's author, the holder of the economic rights, and the successive licensors have only limited liability.

In this respect, the user's attention is drawn to the risks associated with loading, using, modifying and/or developing or reproducing the software by the user in light of its specific status of free software, that may mean that it is complicated to manipulate, and that also therefore means that it is reserved for developers and experienced professionals having in-depth computer knowledge. Users are therefore encouraged to load and test the software's suitability as regards their requirements in conditions enabling the security of their systems and/or data to be ensured and, more generally, to use and operate it in the same conditions as regards security.

The fact that you are presently reading this means that you have had knowledge of the CeCILL license and that you accept its terms.

---

18. 2 - Standard make targets predefined in CMT

These targets can always be listed through the following command :

```
sh> gmake help
```

target	*usage*
`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
`binclean`	completely remove the `../$(tag)` binary directory
`configclean`	remove all intermediate makefile fragments
`install`	install binaries of this package to the current installation area
`uninstall`	uninstall binaries of this package from the current installation area
`check`	run all applications defined with the -check option
component-name	only build this particular component (as opposed to the `all` target that tries to build all components of this package)
group-name	build all constituents belonging to this group (ie. those defined using the same -group=<group-name> option)

These targets have to be specified as follows :

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

---

18. 3 - Standard macros predefined in CMT

18. 3. 1 - CMT static macros

These macros provide static data about CMT itself. They cannot be modified by the user.

macro	*usage*	*default value*
`CMTrelease`	gives the current release number of CMT	`18`
`CMTVERSION`	gives the current complete version tag of CMT	`v1r18p20041201`

18. 3. 2 - 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.

macro	*usage*	*default value*
`tag`	gives the binary tag	`${CMTCONFIG}`
`src`	the src branch	`../src/`
`inc`	the include branch	`../src/`
`mgr`	the cmt or mgr branch	`../cmt/` or `../mgr/`
`bin`	the branch for binaries	`../$(<package>_tag)/`
`javabin`	the branch for java classes	`../classes/`
`doc`	the doc branch	`../doc/`
`cmt_hardware`	the description of the current hardware	`<none>`
`cmt_system_version`	the version of the current OS	`<none>`
`cmt_compiler_version`	the version of the currently visible C++ compiler	`<none>`

18. 3. 3 - 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.

macro	*usage*	*default value*
`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	*none*
`pp_cflags`	The preprocessor flags for C	*none*
`clinkflags`	The C link flags	*none*

`cpp`	The C++ compiler	`g++`
`preproc`	The C++ preprocessor	`g++ -MD -c`
`cppcomp`	The C++ compiling command	`$(cpp) -c -I$(inc) $(includes) $(cppflags)`
`cpplink`	The C++ linking command	`$(cpp) $(cpplinkflags)`
`cppflags`	The C++ compilation flags	*none*
`pp_cppflags`	The preprocessor flags for C++	*none*
`cpplinkflags`	The C++ link flags	*none*

`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	*none*
`pp_fflags`	The preprocessor flags for fortran	*none*
`flinkflags`	The Fortran link flags	*none*
`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	*none*
`yacc`	The Yacc command	`yacc $(yaccflags)`
`yaccflags`	The Yacc flags	*none*
`ar`	The archive command	`ar -clr`
`ranlib`	The ranlib command	`ranlib`

18. 3. 4 - 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.

macro	*usage*
`<`*`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`	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. A typical example of how to define such a macro could be : `macro Cm_linkopts "-L$(CMROOT)/$(Cm_tag) -lCm -lm"`
`<`*`package`*`>_stamps`	may contain a list of *stamp* 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 : `macro Cm_stamps "$(Cm_root)/$(Cm_tag)/Cm.stamp"` 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.

The following macros are not subject to automatic concatenation (and therefore are not hidden by the `-no_auto_imports` modifier).

macro	*usage*
`<`*`package`*` >_native_version`	specifies the native version of the external package referenced by this *interface* package. When this macro is provided, its value is displayed by the `cmt show uses` command
`<`*`package`*` >_export_paths`	specifies the list of files or directories that should be exported during the deployment process for this package. Generally this is only useful for glue packages refering to external software
`<`*`package`*` >_home`	specifies the base location for external software described in glue packages. This macro is generally used to specify the previous one

---

18. 3. 5 - 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 type (either `lib_` , `app_` or `doc_`).

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

`<type >_<constituent >_cflags`	specific C flags
`<type >_<constituent >_pp_cflags`	specific C preprocessor flags
`<type >_<constituent >_cppflags`	specific C++ flags
`<type >_<constituent >_pp_cppflags`	specific C++ preprocessor flags
`<type >_<constituent >_fflags`	specific Fortran flags
`<type >_<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 *have* 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).
`<group >_dependencies`	provides user defined dependency specifications for each group. The typical use of this macro is fill it with the name of the list of some other constituents which *have* 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).

18. 3. 6 - 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

18. 3. 7 - Generated macros

These macros are automatically *generated* when any cmt connand is run (and thus 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
`package_offset`	The directory offset of the current package
`package_cmtpath`	The package area where the current package has been found
`<package >_project`	The project name to which the corresponding package belongs
`<package >_cmtpath`	The package area where the corresponding package has been found
`<package >_offset`	The directory offset of the corresponding 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

---

18. 3. 8 - Macros related with the installation area mechanisms

These macros contain the parameterisation of the installation area mechanisms.

macro	*usage*	*default value*
cmt_installarea_command		
cmt_uninstallarea_command		
cmt_install_action		$(CMTROOT)\mgr\cmt_install_action.bat
cmt_installdir_action		$(CMTROOT)\mgr\cmt_installdir_action.bat
cmt_uninstall_action		$(CMTROOT)\mgr\cmt_uninstall_action.bat
cmt_uninstalldir_action		$(CMTROOT)\mgr\cmt_uninstalldir_action.bat
cmt_installdir_excludes		$(CMTROOT)\mgr\cmt_installdir_excludes.txt
cmt_installarea_prefix		InstallArea
<project>_installarea_prefix		$(cmt_installarea_prefix)
CMTINSTALLAREA		C:\Arnault\test\tprojects\PC\InstallArea
cmt_installarea_paths		
cmt_installarea_linkopts		

`18. 3. 9` - Utility macros

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

macro	*usage*	*default on Unix*
`X11_cflags`	compilation flags for X11	`-I/usr/include`
`Xm_cflags`	compilation flags for Motif	`-I/usr/include`
`X_linkopts`	Link options for XWindows (and Motif)	
`make_shlib`	The command used to generate the shared library from the static one	`${CMTROOT}/mgr/cmt_make_shlib_common.sh extract`
`shlibsuffix`	The system dependent suffix for shared libraries	`so`
`shlibbuilder`	The loader used to build the shared library	`g++`
`shlibflags`	The additional options given to the shared library builder	`-shared`
`application_suffix`	The default extension for applications	`.exe`
`library_prefix`	The default name prefix of libraries	`lib`
`library_suffix`	The default name suffix of libraries	

`symlink`	The command used to install a symbolic link	`/bin/ln -fs`
	The command used to remove a symbolic link	`/bin/rm -f`
`build_prototype`	The command used to generate the C prototype header file (default to the internal cmt dedicated command)	`$(cmtexe) build prototype`
`build_dependencies`	The command used to generate dependencies (default to the internal cmt dedicated command)	`$(cmtexe) -quiet -tag=$(tags) build dependencies`
`lock_command`	The command used to physically lock a package	`chmod -R a-w ../*`
`unlock_command`	The command used to physically unlock a package	`chmod -R g+w ../*`

`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	

---

18. 4 - Standard tags generated by CMT

tag name	*usage*
`CMTv<n>`	Primary version id of CMT
`CMTr<n>`	Secondary version id of CMT
`CMTp<n>`	Patch id of CMT
`‘uname‘`	The basic platform id
`<project-name>`	The current project name

`<project>_prototypes`
`<project>_no_prototypes`
	The prototypes strategy for each project

`<project>_with_installarea`
`<project>_without_installarea`
	The installation area strategy for each project

`<project>_setup_config`
`<project>_setup_no_config`
	The strategy for generating `<P>CONFIG` for each project

`<project>_setup_root`
`<project>_setup_no_root`
	The strategy for generating `<P>ROOT` for each project

`<project>_setup_cleanup`
`<project>_setup_no_cleanup`
	The installation area cleanup strategy for each project

18. 5 - Standard templates for makefile fragments

template name	*usage*	*used in fragment*
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 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

`18. 6` - 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.

Generated makefile	*description*	*used make fragments*
`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

`18. 7` - The complete requirements syntax

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

cmt-statement : *application*
| *apply_pattern*
| *apply_tag*
| *author*
| *branches*
| *build_strategy*
| *cleanup_script*
| *cmtpath_pattern*
| *document*
| *ignore_pattern*
| *include_dirs*
| *include_path*
| *language*
| *library*
| *make_fragment*
| *manager*
| *package*
| *pattern*
| *private*
| *public*
| *setup_script*
| *setup_strategy*
| *symbol*
| *tag*
| *tag_exclude*
| *use*
| *version*

alias : alias *alias-name default-value* [*tag-expr value* ...]

application : application *application-name* [*constituent-option* ...]
[*source* ...]

constituent-option : -no_share
| -no_static
| -prototypes
| -no_prototypes
| -check
| -target_tag
| -group=*group-name*
| -suffix=*output-suffix*
| -import=*package-name*
| *variable-name* = *variable-value*
| -OS9
| -windows

source : *name.suffix*
| **.suffix*
| *.*
| -s=*new-search-path*
| -k=*selection-regexp*
| -x=*exclusion-regexp*

apply_pattern : apply_pattern *pattern-name* [*template-name* = *value* ...]

apply_tag : apply_tag *tag-name* [*tag-name* ...]

author : author *author-name*

branches : branches *branch-name* ...

build_strategy : build_strategy *build-strategy-name*

build-strategy-name : prototypes
| no_prototypes
| with_installarea
| without_installarea

cleanup_script : cleanup_script *script-name*

cmtpath_pattern : cmtpath_pattern *cmt-statement*
[; *cmt-statement* ...]
document : document *document-name* [*constituent-option* ...]
[*source* ...]
ignore_pattern : ignore_pattern *pattern-name*
include_dirs : include_dirs *search-path*
include_path : include_path *search-path*
language : language *language-name* [*language-option* ...]
language-option : -suffix=*suffix*
| -linker=*linker-command*
| -prototypes
| -preprocessor_command=*preprocessor_command*
| -fragment=*fragment*
| -output_suffix=*output-suffix*
| -extra_output_suffix=*extra-output-suffix*
library : library *library-name* [*constituent-option* ...]
[*source* ...]
action : action *action-name* [*tag-expr* *value* ...]
macro : macro *macro-name* [*tag-expr* *value* ...]
macro_append : macro_append *macro-name* [*tag-expr* *value* ...]
macro_prepend : macro_prepend *macro-name* [*tag-expr* *value* ...]
macro_remove : macro_remove *macro-name* [*tag-expr* *value* ...]
macro_remove_regexp : macro_remove_regexp *macro-name* [*tag-expr* *value* ...]
macro_remove_all : macro_remove_all *macro-name* [*tag-expr* *value* ...]
macro_remove_all_regexp : macro_remove_all_regexp *macro-name* [*tag-expr* *value* ...]
make_fragment : make_fragment *fragment-name* [*fragment-option* ...]
fragment-option : -suffix=*suffix*
| -dependencies
| -header=*fragment*

| -trailer=*fragment*
manager : manager *manager-name*
package : package *package-name*
path : path *path-name* [*tag-expr* *value* ...]
path_append : path_append *path-name* [*tag-expr* *value* ...]
path_prepend : path_prepend *path-name* [*tag-expr* *value* ...]
path_remove : path_remove *path-name* [*tag-expr* *value* ...]
path_remove_regexp : path_remove_regexp *path-name* [*tag-expr* *value* ...]
pattern : pattern [-global] *pattern-name* *cmt-statement*
[; *cmt-statement* ...]
private : private
public : public
set : set *set-name* [*tag-expr* *value* ...]
set_append : set_append *set-name* [*tag-expr* *value* ...]
set_prepend : set_prepend *set-name* [*tag-expr* *value* ...]
set_remove : set_remove *set-name* [*tag-expr* *value* ...]
set_remove_regexp : set_remove_regexp *set-name* [*tag-expr* *value* ...]
setup_script : setup_script *script-name*
setup_strategy : setup_strategy *setup-strategy-name*
setup-strategy-name : config
| no_config
| root
| no_root
| cleanup
| no_cleanup
symbol : *alias*
| *action*
| *macro*
| *macro_append*
| *macro_prepend*

| *macro_remove*

| *macro_remove_regexp*

| *macro_remove_all*

| *macro_remove_all_regexp*

| *path*

| *path_append*

| *path_prepend*

| *path_remove*

| *path_remove_regexp*

| *set*

| *set_append*

| *set_prepend*

| *set_remove*

| *set_remove_regexp*

tag : tag *tag-name* [*tag-name* ...]

tag_exclude : tag_exclude *tag-name* [*tag-name* ...]

tag-expr : *tag-name* [& *tag-name* ...]

use : use *package-name* [*version-tag* [*access-path*]]
[*use-option*]

version : version *version-tag*

version-tag : *key* *version-number*
[*key* *release-number* [*key* *patch-number*]]

use-option : -no_auto_imports

| -auto_imports

key : *letter* ...

---

18. 8 - The default strategies defined in CMT

```
DefaultPrototypesStrategy = Prototypes,
DefaultInstallAreaStrategy = WithoutInstallArea
```

18. 9 - 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:

```
sh> cd ${CMTROOT}/mgr
sh> 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 cvs`*`xxx`* `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)

Contents

1 Presentation

2 The conventions

3 The architecture of the environment

3. 1 Supported platforms

4 Installing a new package

5 Localizing a package

6 The concept of project

7 Assigning semantics to packages. Common practices

7. 1 The primary package

7. 2 The policy package

7. 3 The container or management package

7. 4 The release package

7. 5 The glue or interface package

8 Managing site dependent features - The CMTSITE environment variable

9 Configuring a package

10 Selecting a specific configuration

10. 1 Describing a configuration

10. 2 Defining the user tags

10. 3 Activating tags

11 Working on a package

11. 1 Working on a library

11. 2 Working on an application

11. 3 Working on a test or external application

11. 4 Construction of a global environment

12 Defining a document generator

12. 1 An example : the tex document-style

12. 2 How to create and install a new document style

12. 3 Examples
13 The tools provided by CMT
13. 1 The requirements file
13. 1. 1 The general requirements syntax
13. 2 The concepts handled in the requirements file
13. 2. 1 The package structuring style
13. 2. 2 Meta-information : author, manager
13. 2. 3 package, version
13. 2. 4 Constituents : application, library, document
13. 2. 5 Groups
13. 2. 6 Languages
13. 2. 7 Symbols
13. 2. 7. 1 actions
13. 2. 8 use
13. 2. 9 patterns
13. 2. 9. 1 Applying a pattern
13. 2.10 cmtpath_patterns
13. 2.11 branches
13. 2.12 Strategy specifications
13. 2.13 setup_script, cleanup_script
13. 2.14 include_path
13. 2.15 include_dirs
13. 2.16 make_fragment
13. 2.17 public, private
13. 2.17. 1 Scoping sections
13. 2.18 tag, apply_tag
13. 3 The general cmt user interface
13. 3. 1 cmt broadcast
13. 3. 1. 1 Specifying the shell command
13. 3. 1. 2 Templates in the shell command

13. 3. 2 cmt build <option>
13. 3. 3 cmt check configuration
13. 3. 4 cmt check files <reference-file> <new-file>
13. 3. 5 cmt checkout ...
13. 3. 6 cmt co ...
13. 3. 7 cmt cleanup [-csh|-sh]
13. 3. 8 cmt config
13. 3. 9 cmt create <package> <version> [<area>]
13. 3.10 cmt expand model [-strict] <model-string>
13. 3.11 cmt filter <in-file> <out-file>
13. 3.12 cmt help | --help
13. 3.13 cmt lock [<package> <version> [<area>]]
13. 3.14 cmt remove <package> <version> [<area>]
13. 3.15 cmt remove library_links
13. 3.16 cmt run [shell-command]
13. 3.17 cmt set version <version>
13. 3.18 cmt set versions
13. 3.19 cmt setup [-csh|-sh|-bat]
13. 3.20 cmt show <option>
13. 3.21 cmt system
13. 3.22 cmt unlock [<package> <version> [<area>]]
13. 3.23 cmt version | --version
13. 3.24 cmt cvstags <module>
13. 3.25 cmt cvsbranches <module>
13. 3.26 cmt cvssubpackages <module>
13. 4 The setup and cleanup scripts
13. 5 cmt build prototype
14 Using cvs together with CMT
14. 1 Importing a package into a cvs repository
14. 2 Checking a package out from a cvs repository

14. 3 Querying CVS about some important infos
14. 4 Working on a package, creating a new release
14. 5 Getting a particular tagged version out of CVS
15 Interfacing an external package with CMT
16 The installation area mechanism
16. 1 The default implementation
16. 2 Tuning the installation area mechanisms
17 Installing CMT for the first time
17. 1 Installing CMT on your Unix site
17. 2 Installing CMT on a Windows or Windows NT site
18 Appendices
18. 1 Copyright
18. 2 Standard make targets predefined in CMT
18. 3 Standard macros predefined in CMT
18. 3. 1 CMT static macros
18. 3. 2 Structural macros
18. 3. 3 Language related macros
18. 3. 4 Package customizing macros
18. 3. 5 Constituent specific customizing macros
18. 3. 6 Source specific customizing macros
18. 3. 7 Generated macros
18. 3. 8 Macros related with the installation area mechanisms
18. 3. 9 Utility macros
18. 4 Standard tags generated by CMT
18. 5 Standard templates for makefile fragments
18. 6 Makefile generation sequences
18. 7 The complete requirements syntax
18. 8 The default strategies defined in CMT
18. 9 The internal mechanism of cmt cvs operations

Images

1 Structuring a package - A typical example.
2 Structuring a sofware base.
3 The architecture of document generation.

Christian Arnault