

# **The eCos Component Writer's Guide**

**Bart Veer**

**John Dallaway**

## **The eCos Component Writer's Guide**

by Bart Veer and John Dallaway

Published 2001

Copyright © 2000, 2001 Free Software Foundation, Inc.

# Table of Contents

<b>1. Overview .....</b>	<b>1</b>
Terminology .....	1
Component Framework .....	1
Configuration Option .....	1
Component .....	1
Package .....	2
Configuration .....	2
Target .....	2
Template .....	2
Properties .....	3
Consequences .....	3
Constraints .....	3
Conflicts .....	3
CDL .....	4
Component Repository .....	4
Why Configurability? .....	5
Approaches to Configurability .....	5
Degrees of Configurability .....	7
Warnings .....	9
<b>2. Package Organization .....</b>	<b>11</b>
Packages and the Component Repository .....	11
Package Versioning .....	12
Package Contents and Layout .....	13
Outline of the Build Process .....	15
Configurable Source Code .....	16
Compiler Flag Dependencies .....	16
Package Interfaces and Implementations .....	16
Source Code and Configuration Options .....	17
Exported Header Files .....	18
Configurable Functionality .....	18
Nested #include's .....	19
Including Configuration Headers .....	19
Package Documentation .....	20
Test Cases .....	20
Host-side Support .....	20
Making a Package Distribution .....	20
The eCos package distribution file format .....	21
Preparing eCos packages for distribution .....	21
<b>3. The CDL Language .....</b>	<b>25</b>
Language Overview .....	25
CDL Commands .....	26
CDL Properties .....	29
Information-providing Properties .....	30
The Configuration Hierarchy .....	31
Value-related Properties .....	31

Generating the Configuration Header Files .....	34
Controlling what gets Built .....	35
Miscellaneous Properties .....	36
Option Naming Convention .....	37
An Introduction to Tcl.....	40
Values and Expressions .....	44
Option Values .....	44
Is the Option Loaded? .....	44
Is the Option Active .....	45
Is the Option Enabled? What is the Data? .....	46
Some Examples.....	48
Ordinary Expressions .....	50
Functions .....	53
Goal Expressions .....	56
List Expressions.....	57
Interfaces .....	58
Updating the ecos.db database .....	60
<b>4. The Build Process .....</b>	<b>63</b>
Build Tree Generation.....	63
Configuration Header File Generation .....	64
The <code>system.h</code> Header.....	69
Building eCos.....	70
Updating the Build Tree .....	71
Exporting Public Header Files.....	71
Compiling .....	73
Generating the Libraries .....	75
The <code>extras.o</code> file .....	76
Compilers and Flags .....	76
Custom Build Steps .....	79
Startup Code .....	82
The Linker Script.....	82
Building Test Cases.....	83
<b>5. CDL Language Specification .....</b>	<b>85</b>
<code>cdl_option</code> .....	85
<code>cdl_component</code> .....	89
<code>cdl_package</code> .....	93
<code>cdl_interface</code> .....	97
<code>active_if</code> .....	101
<code>calculated</code> .....	105
<code>compile</code> .....	109
<code>default_value</code> .....	111
<code>define</code> .....	115
<code>define_format</code> .....	119
<code>define_header</code> .....	121
<code>define_proc</code> .....	123
<code>description</code> .....	125
<code>display</code> .....	127

doc .....	129
flavor .....	131
hardware .....	135
if_define .....	137
implements .....	141
include_dir .....	143
include_files .....	145
legal_values .....	147
library .....	149
make .....	151
make_object .....	153
no_define .....	155
parent .....	157
requires .....	159
script .....	161
<b>6. Templates, Targets and Other Topics .....</b>	<b>163</b>
Templates .....	163
Targets .....	163



# Chapter 1. Overview

eCos was designed from the very beginning as a configurable component architecture. The core eCos system consists of a number of different components such as the kernel, the C library, an infrastructure package. Each of these provides a large number of configuration options, allowing application developers to build a system that matches the requirements of their particular application. To manage the potential complexity of multiple components and lots of configuration options, eCos comes with a component framework: a collection of tools specifically designed to support configuring multiple components. Furthermore this component framework is extensible, allowing additional components to be added to the system at any time.

## Terminology

The eCos component architecture involves a number of key concepts.

### Component Framework

The phrase component framework is used to describe the collection of tools that allow users to configure a system and administer a component repository. This includes the `ecosconfig` command line tool, the graphical configuration tool, and the package administration tool. Both the command line and graphical tools are based on a single underlying library, the CDL library.

### Configuration Option

The option is the basic unit of configurability. Typically each option corresponds to a single choice that a user can make. For example there is an option to control whether or not assertions are enabled, and the kernel provides an option corresponding to the number of scheduling priority levels in the system. Options can control very small amounts of code such as whether or not the C library's `strtok` gets inlined. They can also control quite large amounts of code, for example whether or not the `printf` supports floating point conversions.

Many options are straightforward, and the user only gets to choose whether the option is enabled or disabled. Some options are more complicated, for example the number of scheduling priority levels is a number that should be within a certain range. Options should always start off with a sensible default setting, so that it is not necessary for users to make hundreds of decisions before any work can start on developing the application. Once the application is running the various configuration options can be used to tune the system for the specific needs of the application.

The component framework allows for options that are not directly user-modifiable. Consider the case of processor endianness: some processors are always big-endian or always little-endian, while with other processors there is a choice. Depending on the user's choice of target hardware, endianness may or may not be user-modifiable.

### Component

A component is a unit of functionality such as a particular kernel scheduler or a device driver for a specific device. A component is also a configuration option in that users may want to enable or disable all the functionality in a component. For example, if a particular device on the target hardware is not going to be used by the application,

directly or indirectly, then there is no point in having a device driver for it. Furthermore disabling the device driver should reduce the memory requirements for both code and data.

Components may contain further configuration options. In the case of a device driver, there may be options to control the exact behavior of that driver. These will of course be irrelevant if the driver as a whole is disabled. More generally options and components live in a hierarchy, where any component can contain options specific to that component and further sub-components. It is possible to view the entire eCos kernel as one big component, containing sub-components for scheduling, exception handling, synchronization primitives, and so on. The synchronization primitives component can contain further sub-components for mutexes, semaphores, condition variables, event flags, and so on. The mutex component can contain configuration options for issues like priority inversion support.

## Package

A package is a special type of component. Specifically, a package is the unit of distribution of components. It is possible to create a distribution file for a package containing all of the source code, header files, documentation, and other relevant files. This distribution file can then be installed using the appropriate tool. Afterwards it is possible to uninstall that package, or to install a later version. The core eCos distribution comes with a number of packages such as the kernel and the infrastructure. Other packages such as network stacks can come from various different sources and can be installed alongside the core distribution.

Packages can be enabled or disabled, but the user experience is a little bit different. Generally it makes no sense for the tools to load the details of every single package that has been installed. For example, if the target hardware uses an ARM processor then there is no point in loading the HAL packages for other architectures and displaying choices to the user which are not relevant. Therefore enabling a package means loading its configuration data into the appropriate tool, and disabling a package is an unload operation. In addition, packages are not just enabled or disabled: it is also possible to select the particular version of a package that should be used.

## Configuration

A configuration is a collection of user choices. The various tools that make up the component framework deal with entire configurations. Users can create a new configuration, output a savefile (by default `ecos.ecc`), manipulate a configuration, and use a configuration to generate a build tree prior to building eCos and any other packages that have been selected. A configuration includes details such as which packages have been selected, in addition to finer-grained information such as which options in those packages have been enabled or disabled by the user.

## Target

The target is the specific piece of hardware on which the application is expected to run. This may be an off-the-shelf evaluation board, a piece of custom hardware intended for a specific application, or it could be something like a simulator. One of the steps when creating a new configuration is need to specify the target. The component framework will map this on to a set of packages that are used to populate the configuration, typically HAL and device driver packages, and in addition it may cause certain options to be changed from their default settings to something more appropriate for the specified target.



## Template

A template is a partial configuration, aimed at providing users with an appropriate starting point. eCos is shipped with a small number of templates, which correspond closely to common ways of using the system. There is a minimal template which provides very little functionality, just enough to bootstrap the hardware and then jump directly to application code. The default template adds additional functionality, for example it causes the kernel and C library packages to be loaded as well. The uitron template adds further functionality in the form of a  $\mu$ ITRON compatibility layer. Creating a new configuration typically involves specifying a template as well as a target, resulting in a configuration that can be built and linked with the application code and that will run on the actual hardware. It is then possible to fine-tune configuration options to produce something that better matches the specific requirements of the application.

## Properties

The component framework needs a certain amount of information about each option. For example it needs to know what the legal values are, what the default should be, where to find the on-line documentation if the user needs to consult that in order to make a decision, and so on. These are all properties of the option. Every option (including components and packages) consists of a name and a set of properties.

## Consequences

Choices must have consequences. For an eCos configuration the main end product is a library that can be linked with application code, so the consequences of a user choice must affect the build process. This happens in two main ways. First, options can affect which files get built and end up in the library. Second, details of the current option settings get written into various configuration header files using C preprocessor `#define` directives, and package source code can `#include` these configuration headers and adapt accordingly. This allows options to affect a package at a very fine grain, at the level of individual lines in a source file if desired. There may be other consequences as well, for example there are options to control the compiler flags that get used during the build process.

## Constraints

Configuration choices are not independent. The C library can provide thread-safe implementations of functions like `rand`, but only if the kernel provides support for per-thread data. This is a constraint: the C library option has a requirement on the kernel. A typical configuration involves a considerable number of constraints, of varying complexity: many constraints are straightforward, option A requires option B, or option C precludes option D. Other constraints can be more complicated, for example option E may require the presence of a kernel scheduler but does not care whether it is the bitmap scheduler, the mlqueue scheduler, or something else.

Another type of constraint involves the values that can be used for certain options. For example there is a kernel option related to the number of scheduling levels, and there is a legal values constraint on this option: specifying zero or a negative number for the number of scheduling levels makes no sense.

## Conflicts

As the user manipulates options it is possible to end up with an invalid configuration, where one or more constraints are not satisfied. For example if kernel per-thread data is disabled but the C library's thread-safety options are left enabled then there are unsatisfied constraints, also known as conflicts. Such conflicts will be reported by the configuration tools. The presence of conflicts does not prevent users from attempting to build eCos, but the consequences are undefined: there may be compile-time failures, there may be link-time failures, the application may completely fail to run, or the application may run most of the time but once in a while there will be a strange failure... Typically users will want to resolve all conflicts before continuing.

To make things easier for the user, the configuration tools contain an inference engine. This can examine a conflict in a particular configuration and try to figure out some way of resolving the conflict. Depending on the particular tool being used, the inference engine may get invoked automatically at certain times or the user may need to invoke it explicitly. Also depending on the tool, the inference engine may apply any solutions it finds automatically or it may request user confirmation.

## CDL

The configuration tools require information about the various options provided by each package, their consequences and constraints, and other properties such as the location of on-line documentation. This information has to be provided in the form of CDL scripts. CDL is short for Component Definition Language, and is specifically designed as a way of describing configuration options.

A typical package contains the following:

1. Some number of source files which will end up in a library. The application code will be linked with this library to produce an executable. Some source files may serve other purposes, for example to provide a linker script.
2. Exported header files which define the interface provided by the package.
3. On-line documentation, for example reference pages for each exported function.
4. Some number of test cases, shipped in source format, allowing users to check that the package is working as expected on their particular hardware and in their specific configuration.
5. One or more CDL scripts describing the package to the configuration system.

Not all packages need to contain all of these. For example some packages such as device drivers may not provide a new interface, instead they just provide another implementation of an existing interface. However all packages must contain a CDL script that describes the package to the configuration tools.

## Component Repository

All eCos installations include a component repository. This is a directory structure where all the packages get installed. The component framework comes with an administration tool that allows new packages or new versions of a package to be installed, old packages to be removed, and so on. The component repository includes a simple database, maintained by the administration tool, which contains details of the various packages.

Generally application developers do not need to modify anything inside the component repository, except by means of the administration tool. Instead their work involves separate build and install trees. This allows the component

repository to be treated as a read-only resource that can be shared by multiple projects and multiple users. Component writers modifying one of the packages do need to manipulate files in the component repository.

## Why Configurability?

The eCos component framework places a great deal of emphasis on configurability. The fundamental goal is to allow large parts of embedded applications to be constructed from re-usable software components, which does not a priori require that those components be highly configurable. However embedded application development often involves some serious constraints.

Many embedded applications have to work with very little memory, to keep down manufacturing costs. The final application image that will get blown into EPROM's or used to manufacture ROMs should contain only the code that is absolutely necessary for the application to work, and nothing else. If a few tens of kilobytes are added unnecessarily to a typical desktop application then this is regrettable, but is quite likely to go unnoticed. If an embedded application does not fit on the target hardware then the problem is much more serious. The component framework must allow users to configure the components so that any unnecessary functionality gets removed.

Many embedded applications need deterministic behavior so that they can meet real-time requirements. Such deterministic behavior can often be provided, but at a cost in terms of code size, slower algorithms, and so on. Other applications have no such real-time requirements, or only for a small part of the overall system, and the bulk of the system should not suffer any penalties. Again the component framework must allow the users control over the timing behavior of components.

Embedded systems tend to be difficult to debug. Even when it is possible to get information out of the target hardware by means other than flashing an LED, the more interesting debugging problems are likely to be timing-related and hence very hard to reproduce and track down. The re-usable components can provide debugging assistance in various ways. They can provide functionality that can be exploited by source level debuggers such as gdb, for example per-thread debugging information. They can also contain various assertions so that problems can be detected early on, tracing mechanisms to figure out what happened before the assertion failure, and so on. Of course all of these involve overheads, especially code size, and affect the timing. Allowing users to control which debugging features are enabled for any given application build is very desirable.

However, although it is desirable for re-usable components to provide appropriate configuration options this is not required. It is possible to produce a package which does not provide a single configuration option — although the user still gets to choose whether or not to use the package. In such cases it is still necessary to provide a minimal CDL script, but its main purpose would be to integrate the package with the component framework's build system.

## Approaches to Configurability

The purpose of configurability is to control the behavior of components. A scheduler component may or may not support time slicing; it may or may not support multiple priorities; it may or may not perform error checking on arguments passed to the scheduler routines. In the context of a desktop application a button widget may contain some text or it may contain a picture; the text may be displayed in a variety of fonts; the foreground and background color may vary. When an application uses a component there must be some way of specifying the desired behavior. The component writer has no way of knowing in advance exactly how a particular component will end up being used.

One way to control the behavior is at run time. The application creates an instance of a button object, and then instructs this object to display either text or a picture. No special effort by the application developer is required, since a button can always support all desired behavior. There is of course a major disadvantage in terms of the size of the final application image: the code that gets linked with the application has to provide support for all possible behavior, even if the application does not require it.

Another approach is to control the behavior at link-time, typically by using inheritance in an object-oriented language. The button library provides an abstract base class `Button` and derived classes `TextButton` and `PictureButton`. If an application only uses text buttons then it will only create objects of type `TextButton`, and the code for the `PictureButton` class does not get used. In many cases this approach works rather well and reduces the final image size, but there are limitations. The main one is that you can only have so many derived classes before the system gets unmanageable: a derived class `TextButtonUsingABorderWidthOfOnePlusAWhiteBackgroundAndBlackForegroundAndATwelvePointTimesFontAndN` is not particularly sensible as far as most application developers are concerned.

The eCos component framework allows the behavior of components to be controlled at an even earlier time: when the component source code gets compiled and turned into a library. The button component could provide options, for example an option that only text buttons need to be supported. The component gets built and becomes part of a library intended specifically for the application, and the library will contain only the code that is required by this application and nothing else. A different application with different requirements would need its own version of the library, configured separately.

In theory compile-time configurability should give the best possible results in terms of code size, because it allows code to be controlled at the individual statement level rather than at the function or object level. Consider an example more closely related to embedded systems, a package to support multi-threading. A standard routine within such a package allows applications to kill threads asynchronously: the POSIX routine for this is `pthread_cancel`; the equivalent routine in  $\mu$ ITRON is `ter_tsk`. These routines themselves tend to involve a significant amount of code, but that is not the real problem: other parts of the system require extra code and data for the kill routine to be able to function correctly. For example if a thread is blocked while waiting on a mutex and is killed off by another thread then the kill operation may have to do two things: remove the thread from the mutex's queue of waiting threads; and undo the effects, if any, of priority inheritance. The implementation requires extra fields in the thread data structure so that the kill routine knows about the thread's current state, and extra code in the mutex routines to fill in and clear these extra fields correctly.

Most embedded applications do not require the ability to kill off a thread asynchronously, and hence the kill routine will not get linked into the final application image. Without compile-time configurability this would still mean that the mutex code and similar parts of the system contain code and data that serve no useful purpose in this application. The eCos approach allows the user to select that the thread kill functionality is not required, and all the components can adapt to this at compile-time. For example the code in the mutex lock routine contains statements to support the killing of threads, but these statements will only get compiled in if that functionality is required. The overall result is that the final application image contains only the code and data that is really needed for the application to work, and nothing else.

Of course there are complications. To return to the button example, the application code might only use text buttons directly, but it might also use some higher-level widget such as a file selector and this file selector might require buttons with pictures. Therefore the button code must still be compiled to support pictures as well as text. The configuration tools must be aware of the dependencies between components and ensure that the internal constraints are met, as well as the external requirements of the application code. An area of particular concern is conflicting requirements: a button component might be written in such a way that it can only support either text buttons or picture buttons, but not both in one application; this would represent a weakness in the component itself rather than in the component framework as a whole.

Compile-time configurability is not intended to replace the other approaches but rather to complement them. There will be times when run-time selection of behavior is desirable: for example an application may need to be able to change the baud rate of a serial line, and the system must then provide a way of doing this at run-time. There will also be times when link-time selection is desirable: for example a C library might provide two different random number routines `rand` and `lrand48`; these do not affect other code so there is no good reason for the C library component not to provide both of these, and allow the application code to use none, one, or both of them as appropriate; any unused functions will just get eliminated at link-time. Compile-time selection of behavior is another option, and it can be the most powerful one of the three and the best suited to embedded systems development.

## Degrees of Configurability

Components can support configurability in varying degrees. It is not necessary to have any configuration options at all, and the only user choice is whether or not to load a particular package. Alternatively it is possible to implement highly-configurable code. As an example consider a typical facility that is provided by many real-time kernels, mutex locks. The possible configuration options include:

1. If no part of the application and no other component requires mutexes then there is no point in having the mutex code compiled into a library at all. This saves having to compile the code. In addition there will never be any need for the user to configure the detailed behavior of mutexes. Therefore the presence of mutexes is a configuration option in itself.
2. Even if the application does make use of mutexes directly or indirectly, this does not mean that all mutex functions have to be included. The minimum functionality consists of lock and unlock functions. However there are variants of the locking primitive such as try-lock and try-with-timeout which may or may not be needed.

Generally it will be harmless to compile the try-lock function even if it is not actually required, because the function will get eliminated at link-time. Some users might take the view that the try-lock function should never get compiled in unless it is actually needed, to reduce compile-time and disk usage. Other users might argue that there are very few valid uses for a try-lock function and it should not be compiled by default to discourage incorrect uses. The presence of a try-lock function is a possible configuration option, although it may be sensible to default it to true.

The try-with-timeout variant is more complicated because it adds a dependency: the mutex code will now rely on some other component to provide a timer facility. To make things worse the presence of this timer might impact other components, for example it may now be necessary to guard against timer interrupts, and thus have an insidious effect on code size. The presence of a lock-with-timeout function is clearly a sensible configuration option, but the default value is less obvious. If the option is enabled by default then the final application image may end up with code that is not actually essential. If the option is disabled by default then users will have to enable the option somehow in order to use the function, implying more effort on the part of the user. One possible approach is to calculate the default value based on whether or not a timer component is present anyway.

3. The application may or may not require the ability to create and destroy mutexes dynamically. For most embedded systems it is both less error-prone and more efficient to create objects like mutexes statically. Dynamic creation of mutexes can be implemented using a pre-allocated pool of mutex objects, involving some extra

code to manipulate the pool and an additional configuration option to define the size of the pool. Alternatively it can be implemented using a general-purpose memory allocator, involving quite a lot of extra code and configuration options. However this general-purpose memory allocator may be present anyway to support the application itself or some other component. The ability to create and destroy mutexes dynamically is a configuration option, and there may not be a sensible default that is appropriate for all applications.

4. An important issue for mutex locks is the handling of priority inversion, where a high priority thread is prevented from running because it needs a lock owned by a lower priority thread. This is only an issue if there is a scheduler with multiple priorities: some systems may need multi-threading and hence synchronization primitives, but a single priority level may suffice. If priority inversion is a theoretical possibility then the application developer may still want to ignore it because the application has been designed such that the problem cannot arise in practice. Alternatively the developer may want some sort of exception raised if priority inversion does occur, because it should not happen but there may still be bugs in the code. If priority inversion can occur legally then there are three main ways of handling it: priority ceilings, priority inheritance, and ignoring the problem. Priority ceilings require little code but extra effort on the part of the application developer. Priority inheritance requires more code but is automatic. Ignoring priority inversion may or may not be acceptable, depending on the application and exactly when priority inversion can occur. Some of these choices involve additional configuration options, for example there are different ways of raising an exception, and priority inheritance may or may not be applied recursively.
5. As a further complication some mutexes may be hidden inside a component rather than being an explicit part of the application. For example, if the C library is configured to provide a `malloc` call then there may be an associated mutex to make the function automatically thread-safe, with no need for external locking. In such cases the memory allocation component of the C library can impose a constraint on the kernel, requiring that mutexes be provided. If the user attempts to disable mutexes anyway then the configuration tools will report a conflict.
6. The mutex code should contain some general debugging code such as assertions and tracing. Usually such debug support will be enabled or disabled at a coarse level such as the entire system or everything inside the kernel, but sometimes it will be desirable to enable the support more selectively. One reason would be memory requirements: the target may not have enough memory to hold the system if all debugging is enabled. Another reason is if most of the system is working but there are a few problems still to resolved; enabling debugging in the entire system might change the system's timing behavior too much, but enabling some debug options selectively can still be useful. There should be configuration options to allow specific types of debugging to be enabled at a fine-grain, but with default settings inherited from an enclosing component or from global settings.
7. The mutex code may contain specialized code to interact with a debugging tool running on the host. It should be possible to enable or disable this debugging code, and there may be additional configuration options controlling the detailed behavior.

Altogether there may be something like ten to twenty configuration options that are specific to the mutex code. There may be a similar number of additional options related to assertions and other debug facilities. All of the options should have sensible default values, possibly fixed, possibly calculated depending on what is happening elsewhere in the configuration. For example the default setting for an assertion option should generally inherit from a kernel-wide assertion control option, which in turn inherits from a global option. This allows users to enable or disable assertions globally or at a more fine-grained level, as desired.

Different components may be configurable to different degrees, ranging from no options at all to the fine-grained configurability of the above mutex example (or possibly even further). It is up to component writers to decide what options should be provided and how best to serve the needs of application developers who want to use that

component.

## Warnings

Large parts of eCos were developed concurrently with the development of the configuration technology, or in some cases before design work on that technology was complete. As a consequence the various eCos packages often make only limited use of the available functionality. This situation is expected to change over time. It does mean that many of the descriptions in this guide will not correspond exactly to how the eCos packages work right now, but rather to how they could work. Some of the more extreme discrepancies such as the location of on-line documentation in the component repository will be mentioned in the appropriate places in the guide.

A consequence of this is that developers of new components can look at existing CDL scripts for examples, and discover discrepancies between what is recommended in this guide and what actually happens at present. In such cases this guide should be treated as authoritative.

It is also worth noting that the current component framework is not finished. Various parts of this guide will refer to possible changes and enhancements in future versions. Examining the source code of the configuration tools may reveal hints about other likely developments, and there are many more possible enhancements which only exist at a conceptual level right now.





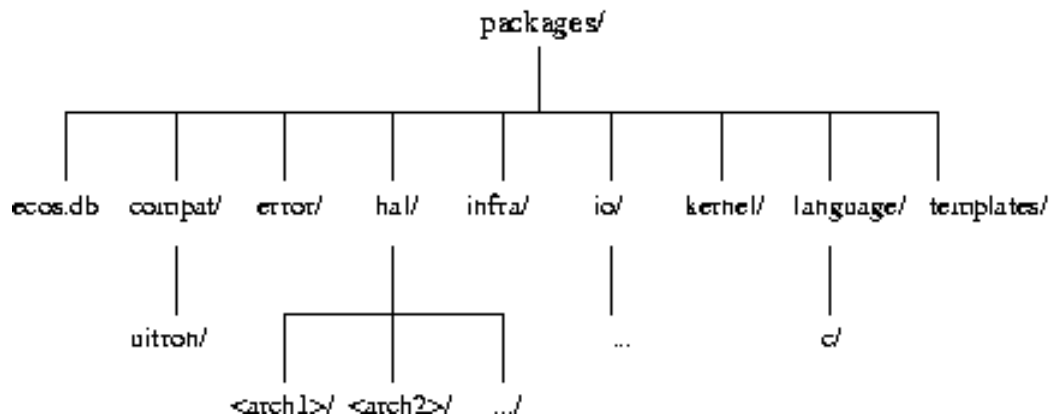
## Chapter 2. Package Organization

For a package to be usable in the eCos component framework it must conform to certain rules imposed by that framework. Packages must be distributed in a form that is understood by the component repository administration tool. There must be a top-level CDL script which describes the package to the component framework. There are certain limitations related to how a package gets built, so that the package can still be used in a variety of host environments. In addition to these rules, the component framework provides a number of guidelines. Packages do not have to conform to the guidelines, but sticking to them can simplify certain operations.

This chapter deals with the general organization of a package, for example how to distinguish between private and exported header files. [Chapter 3](#) describes the CDL language. [Chapter 4](#) details the build process.

### Packages and the Component Repository

All eCos installations include a component repository. This is a directory structure for all installed packages. The component framework comes with an administration tool that allows new packages or new versions of a package to be installed, old packages to be removed, and so on. The component repository includes a simple database, maintained by the administration tool, which contains details of the various packages.



Each package has its own little directory hierarchy within the component repository. Keeping several packages in a single directory is illegal. The error, infra and kernel packages all live at the top-level of the repository. For other types of packages there are some pre-defined directories: `compat` is used for compatibility packages, which implement other interfaces such as  $\mu$ ITRON or POSIX using native eCos calls; `hal` is used for packages that port eCos to different architectures or platforms, and this directory is further organized on a per-architecture basis; `io` is intended for device drivers; `language` is used for language support libraries, for example the C library. There are no strict rules defining where new packages should get installed. Obviously if an existing top-level directory such as `compat` is applicable then the new package should go in there. If a new category is desirable then it is possible to create a new sub-directory in the component repository. For example, an organization planning to release a number of eCos packages may want them all to appear below a sub-directory corresponding to the organization's name — in the hope that the name will not change too often. It is possible to add new packages directly to the top-level of the component repository, but this should be avoided.

The `ecos.db` file holds the component repository database and is managed by the administration tool. The various configuration tools read in this file when they start-up to obtain information about the various packages that

have been installed. When developing a new package it is necessary to add some information to the file, as described in [the Section called \*Updating the ecos.db database\* in Chapter 3](#). The `templates` directory holds various configuration templates.

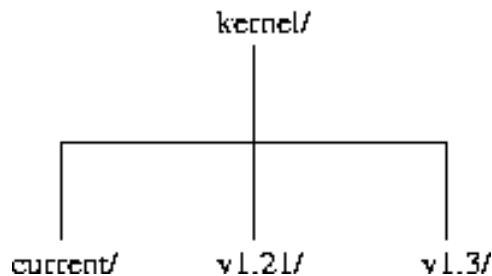
**Note:** Earlier releases of eCos came with two separate files, `targets` and `packages`. The `ecos.db` database replaces both of these.

### Caution

The current `ecos.db` database does not yet provide all of the information needed by the component framework. Its format is subject to change in future releases, and the file may be replaced completely if necessary. There are a number of other likely future developments related to the component repository and the database. The way targets are described is subject to change. Sometimes it is desirable for component writers to do their initial development in a directory outside the component repository, but there is no specific support in the framework for that yet.

## Package Versioning

Below each package directory there can be one or more version sub-directories, named after the versions. This is a requirement of the component framework: it must be possible for users to install multiple versions of a package and select which one to use for any given application. This has a number of advantages to users: most importantly it allows a single component repository to be shared between multiple users and multiple projects, as required; also it facilitates experiments, for example it is relatively easy to try out the latest version of some package and see if it makes any difference. There is a potential disadvantage in terms of disk space. However since eCos packages generally consist of source code intended for small embedded systems, and given typical modern disk sizes, keeping a number of different versions of a package installed will usually be acceptable. The administration tool can be used to remove versions that are no longer required.



The version `current` is special. Typically it corresponds to the very latest version of the sources, obtained by anonymous CVS. These sources may change frequently, unlike full releases which do not change (or only when patches are produced). Component writers may also want to work on the `current` version.

All other subdirectories of a package correspond to specific releases of that package. The component framework allows users to select the particular version of a package they want to use, but by default the most recent one will

be used. This requires some rules for ordering version numbers, a difficult task because of the wide variety of ways in which versions can be identified.

1. The version `current` is always considered to be the most recent version.
2. If the first character of both strings are either `v` or `V`, these are skipped because it makes little sense to enforce case sensitivity here. Potentially this could result in ambiguity if there are two version directories `v1.0` and `V1.0`, but this will match the confusion experienced by any users of such a package. However if two subsequent releases are called `v1.0` and `v1.1`, e.g. because of a minor mix-up when making the distribution file, then the case difference is ignored.
3. Next the two version strings are compared one character at a time. If both strings are currently at a digit then a string to number conversion takes place, and the resulting numbers are compared. For example `v10` is a more recent release than `v2`. If the two numbers are the same then processing continues, so for `v2b` and `v2c` the version comparison code would move on to `b` and `c`.
4. The characters dot `.`, hyphen `-` and underscore `_` are treated as equivalent separators, so if one release goes out as `v1_1` and the next goes out as `v1.2` the separator has no effect.
5. If neither string has yet terminated but the characters are different, ASCII comparison is used. For example `v1.1b` is more recent than `v1.1alpha`.
6. If one version string terminates before the other, the current character determines which is the more recent. If the other string is currently at a separator character, for example `v1.3.1` and `v1.3`, then the former is assumed to be a minor release and hence more recent than the latter. If the other string is not at a separator character, for example `v1.3beta`, then it is treated as an experimental version of the `v1.3` release and hence older.
7. There is no special processing of dates, so with two versions `ss-20000316` and `ss-20001111` the numerical values `20001111` and `20000316` determine the result: larger values are more recent. It is suggested that the full year be used in such cases rather than a shorthand like `00`, to avoid Y2100 problems.
8. There is no limit on how many levels of versioning are used, so there could in theory be a `v3.1.4.1.5.9.2.7` release of a package. However this is unlikely to be of benefit to typical users of a package.

The version comparison rules of the component framework may not be suitable for every version numbering scheme in existence, but they should cope with many common cases.

### Caution

There are some issues still to be resolved before it is possible to combine the `current` sources available via anonymous CVS and full releases of eCos and additional packages in a single component repository. The first problem relates to the `ecos.db` database: if a new package is added via the CVS repository then this requires a database update, but the administration tool is bypassed. The second problem arises if an organization chooses to place its component repository under source code control using CVS, in which case different directories will belong to different CVS servers. These issues will be addressed in a future release.

## Package Contents and Layout

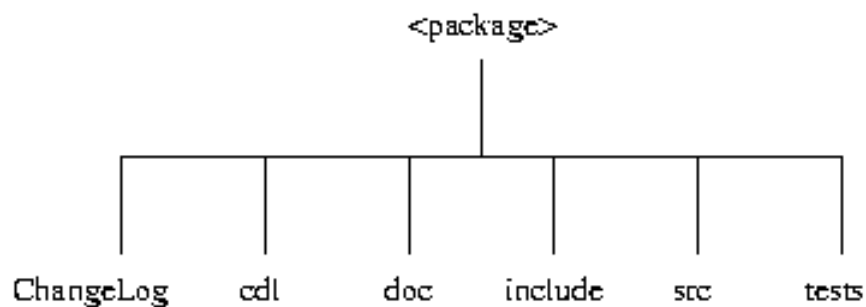
A typical package contains the following:

1. Some number of source files which will end up in a library. The application code will be linked with this library to produce an executable. Some source files may serve other purposes, for example to provide a linker script.
2. Exported header files which define the interface provided by the package.
3. On-line documentation, for example reference pages for each exported function.
4. Some number of test cases, shipped in source format, allowing users to check that the package is working as expected on their particular hardware and in their specific configuration.
5. One or more CDL scripts describing the package to the configuration system.

It is also conventional to have a per-package `ChangeLog` file used to keep track of changes to that package. This is especially valuable to end users of the package who may not have convenient access to the source code control system used to manage the master copy of the package, and hence cannot find out easily what has changed. Often it can be very useful to the main developers as well.

Any given packages need not contain all of these. It is compulsory to have at least one CDL script describing the package, otherwise the component framework would be unable to process it. Some packages may not have any source code: it is possible to have a package that merely defines a common interface which can then be implemented by several other packages, especially in the context of device drivers; however it is still common to have some code in such packages to avoid replicating shareable code in all of the implementation packages. Similarly it is possible to have a package with no exported header files, just source code that implements an existing interface: for example an ethernet device driver might just implement a standard interface and not provide any additional functionality. Packages do not need to come with any on-line documentation, although this may affect how many people will want to use the package. Much the same applies to per-package test cases.

The component framework has a recommended per-package directory layout which splits the package contents on a functional basis:



For example, if a package has an `include` sub-directory then the component framework will assume that all header files in and below that directory are exported header files and will do the right thing at build time. Similarly if there is `doc` property indicating the location of on-line documentation then the component framework will first look in the `doc` sub-directory.

This directory layout is just a guideline, it is not enforced by the component framework. For simple packages it often makes more sense to have all of the files in just one directory. For example a package could just contain the files `hello.cxx`, `hello.h`, `hello.html` and `hello.cdl`. By default `hello.h` will be treated as an exported header file, although this can be overridden with the `include_files` property. Assuming there is a `doc` property referring to `hello.html` and there is no `doc` sub-directory then the tools will search for this file relative to the package's top-level and everything will just work. Much the same applies to `hello.cxx` and `hello.cdl`.

**Tip:** Older versions of the eCos build system only supported packages that followed the directory structure exactly. Hence certain core packages such as `error` implement the full directory structure, even though that is a particularly simple package and the full directory structure is inappropriate. Component writers can decide for themselves whether or not the directory structure guidelines are appropriate for their package.

## Outline of the Build Process

The full build process is described in [Chapter 4](#), but a summary is appropriate here. A build involves three directory structures:

1. The component repository. This is where all the package source code is held, along with CDL scripts, documentation, and so on. For build purposes a component repository is read-only. Application developers will only modify the component repository when installing or removing packages, via the administration tool. Component writers will typically work on just one package in the component repository.
2. The build tree. Each configuration has its own build tree, which can be regenerated at any time using the configuration's `ecos.ecc` savefile. The build tree contains only intermediate files, primarily object files. Once a build is complete the build tree contains no information that is useful for application development and can be wiped, although this would slow down any rebuilds following changes to the configuration.
3. The install tree. This is populated during a build, and contains all the files relevant to application development. There will be a `lib` sub-directory which typically contains `libtarget.a`, a linker script, start-up code, and so on. There will also be an `include` sub-directory containing all the header files exported by the various packages. There will also be a `include/pkgconf` sub-directory containing various configuration header files with `#define`'s for the options. Typically the install tree is created within the build tree, but this is not a requirement.

The build process involves the following steps:

1. Given a configuration, the component framework is responsible for creating all the directories in the build and install trees. If these trees already exist then the component framework is responsible for any clean-ups that may be necessary, for example if a package has been removed then all related files should be expunged from the build and install trees. The configuration header files will be generated at this time. Depending on the host environment, the component framework will also generate makefiles or some other way of building the various packages. Every time the configuration is modified this step needs to be repeated, to ensure that all option consequences take effect. Care is taken that this will not result in unnecessary rebuilds.

**Note:** At present this step needs to be invoked manually. In a future version the generated makefile may if desired perform this step automatically, using a dependency on the `ecos.ecc` savefile.

2. The first step in an actual build is to make sure that the install tree contains all exported header files. All compilations will use the install tree's `include` directory as one of the places to search for header files.
3. All source files relevant to the current configuration get compiled. This involves a set of compiler flags initialized on a per-target basis, with each package being able to modify these flags, and with the ability for the user to override the flags as well. Care has to be taken here to avoid inappropriate target-dependencies in packages

that are intended to be portable. The component framework has built-in knowledge of how to handle C, C++ and assembler source files — other languages may be added in future, as and when necessary. The `compile` property is used to list the files that should get compiled. All object files end up in the build tree.

4. Once all the object files have been built they are collected into a library, typically `libtarget.a`, which can then be linked with application code. The library is generated in the install tree.
5. The component framework provides support for custom build steps, using the `make_object` and `make` properties. The results of these custom build steps can either be object files that should end up in a library, or other files such as a linker script. It is possible to control the order in which these custom build steps take place, for example it is possible to run a particular build step before any of the compilations happen.

## Configurable Source Code

All packages should be totally portable to all target hardware (with the obvious exceptions of HAL and device driver packages). They should also be totally bug-free, require the absolute minimum amount of code and data space, be so efficient that cpu time usage is negligible, and provide lots of configuration options so that application developers have full control over the behavior. The configuration options are optional only if a package can meet the requirements of every potential application without any overheads. It is not the purpose of this guide to explain how to achieve all of these requirements.

The eCos component framework does have some important implications for the source code: compiler flag dependencies; package interfaces vs. implementations; and how configuration options affect source code.

## Compiler Flag Dependencies

Wherever possible component writers should avoid dependencies on particular compiler flags. Any such dependencies are likely to impact portability. For example, if one package needs to be built in big-endian mode and another package needs to be built in little-endian mode then usually it will not be possible for application developers to use both packages at the same time; in addition the application developer is no longer given a choice in the matter. It is far better for the package source code to adapt the endianness at compile-time, or possibly at run-time although that will involve code-size overheads.

**Note:** A related issue is that the current support for handling compiler flags in the component framework is still limited and incapable of handling flags at a very fine-grain. The support is likely to be enhanced in future versions of the framework, but there are non-trivial problems to be resolved.

## Package Interfaces and Implementations

The component framework provides encapsulation at the package level. A package A has no way of accessing the implementation details of another package B at compile-time. In particular, if there is a private header file somewhere in a package's `src` sub-directory then this header file is completely invisible to other packages. Any attempts to cheat by using relative pathnames beginning with `././.` are generally doomed to failure because of the presence of package version directories. There are two ways in which one package can affect another: by means of the exported header files, which define a public interface; or via the CDL scripts.

This encapsulation is a deliberate aspect of the overall eCos component framework design. In most cases it does not cause any problems for component writers. In some cases enforcing a clean separation between interface and implementation details can improve the code. Also it reduces problems when a package gets upgraded: component writers are free to do pretty much anything on the implementation side, including renaming every single source file; care has to be taken only with the exported header files and with the CDL data, because those have the potential of impacting other packages. Application code is similarly unable to access package implementation details, only the exported interface.

Very occasionally the inability of one package to see implementation details of another does cause problems. One example occurs in HAL packages, where it may be desirable for the architectural, variant and platform HAL's to share some information that should not be visible to other packages or to application code. This may be addressed in the future by introducing the concept of `friend` packages, just as a C++ class can have `friend` functions and classes which are allowed special access to a class internals. It is not yet clear whether such cases are sufficiently frequent to warrant introducing such a facility.

## Source Code and Configuration Options

Configurability usually involves source code that needs to implement different behavior depending on the settings of configuration options. It is possible to write packages where the only consequence associated with various configuration options is to control what gets built, but this approach is limited and does not allow for fine-grained configurability. There are three main ways in which options could affect source code at build time:

1. The component code can be passed through a suitable preprocessor, either an existing one such as `m4` or a new one specially designed with configurability in mind. The original sources would reside in the component repository and the processed sources would reside in the build tree. These processed sources can then be compiled in the usual way.

This approach has two main advantages. First, it is independent from the programming language used to code the components, provided reasonable precautions are taken to avoid syntax clashes between preprocessor statements and actual code. This would make it easier in future to support languages other than C and C++. Second, configurable code can make use of advanced preprocessing facilities such as loops and recursion. The disadvantage is that component writers would have to learn about a new preprocessor and embed appropriate directives in the code. This makes it much more difficult to turn existing code into components, and it involves extra training costs for the component writers.

2. Compiler optimizations can be used to elide code that should not be present, for example:

```
...
if (CYGHWR_NUMBER_UARTS > 0) {
    ...
}
```

If the compiler knows that `CYGHWR_NUMBER_UARTS` is the constant number 0 then it is a trivial operation to get rid of the unnecessary code. The component framework still has to define this symbol in a way that is acceptable to the compiler, typically by using a `const` variable or a preprocessor symbol. In some respects this is a clean approach to configurability, but it has limitations. It cannot be used in the declarations of data structures or classes, nor does it provide control over entire functions. In addition it may not be immediately obvious that this code is affected by configuration options, which may make it more difficult to understand.

3. Existing language preprocessors can be used. In the case of C or C++ this would be the standard C preprocessor, and configurable code would contain a number of `#ifdef` and `#if` statements.

```
#if (CYGHWL_NUMBER_UARTS > 0)
    ...
#endif
```

This approach has the big advantage that the C preprocessor is a technology that is both well-understood and widely used. There are also disadvantages: it is not directly applicable to components written in other languages such as Java (although it is possible to use the C preprocessor as a stand-alone program); the pre-processing facilities are rather limited, for example there is no looping facility; and some people consider the technology to be ugly. Of course it may be possible to get around the second objection by extending the preprocessor that is used by gcc and g++.

The current component framework generates configuration header files with C preprocessor `#define`'s for each option (typically, there various properties which can be used to control this). It is up to component writers to decide whether to use preprocessor `#ifdef` statements or language constructs such as `if`. At present there is no support for languages which do not involve the C preprocessor, although such support can be added in future when the need arises.

## Exported Header Files

A package's exported header files should specify the interface provided by that package, and avoid any implementation details. However there may be performance or other reasons why implementation details occasionally need to be present in the exported headers.

**Note:** Not all programming languages have the concept of a header file. In some cases the component framework would need extensions to support packages written in such languages.

Configurability has a number of effects on the way exported header files should be written. There may be configuration options which affect the interface of a package, not just the implementation. It is necessary to worry about nested `#include`'s and how this affects package and application builds. A special case of this relates to whether or not exported header files should `#include` configuration headers. These configuration headers are exported, but should only be `#include`'d when necessary.

## Configurable Functionality

Many configuration options affect only the implementation of a package, not the interface. However some options will affect the interface as well, which means that the options have to be tested in the exported header files. Some implementation choices, for example whether or not a particular function should be inlined, also need to be tested in the header file because of language limitations.

Consider a configuration option `CYGFUN_KERNEL_MUTEX_TIMEDLOCK` which controls whether or not a function `cyg_mutex_timedlock` is provided. The exported kernel header file `cyg/kernel/kapi.h` could contain the following:



```
#include <pkgconf/kernel.h>
...
#ifdef CYGFUN_KERNEL_MUTEX_TIMEDLOCK
extern bool cyg_mutex_timedlock(cyg_mutex_t*);
#endif
```

This is a correct header file, in that it defines the exact interface provided by the package at all times. However it has a number of implications. First, the header file is now dependent on `pkgconf/kernel.h`, so any changes to kernel configuration options will cause `cyg/kernel/kapi.h` to be out of date, and any source files that use the kernel interface will need rebuilding. This may affect sources in the kernel package, in other packages, and in application source code. Second, if the application makes use of this function somewhere but the application developer has misconfigured the system and disabled this functionality anyway then there will now be a compile-time error when building the application. Note that other packages should not be affected, since they should impose appropriate constraints on `CYGFUN_KERNEL_MUTEX_TIMEDLOCK` if they use that functionality (although of course some dependencies like this may get missed by component developers).

An alternative approach would be:

```
extern bool cyg_mutex_timedlock(cyg_mutex_t*);
```

Effectively the header file is now lying about the functionality provided by the package. The first result is that there is no longer a dependency on the kernel configuration header. The second result is that an application file using the timed-lock function will now compile, but the application will fail to link. At this stage the application developer still has to intervene, change the configuration, and rebuild the system. However no application recompilations are necessary, just a relink.

Theoretically it would be possible for a tool to analyze linker errors and suggest possible configuration changes that would resolve the problem, reducing the burden on the application developer. No such tool is planned in the short term.

It is up to component writers to decide which of these two approaches should be preferred. Note that it is not always possible to avoid `#include`'ing a configuration header file in an exported one, for example an option may affect a data structure rather than just the presence or absence of a function. Issues like this will vary from package to package.

## Nested `#include`'s

As a general rule, unnecessary `#include`'s should be avoided. A header file should `#include` only those header files which are absolutely needed for it to define its interface. Any additional `#include`'s make it more likely that package or application source files become dependent on configuration header files and will get rebuilt unnecessarily when there are minor configuration changes.

## Including Configuration Headers

Exported header files should avoid `#include`'ing configuration header files unless absolutely necessary, to avoid unnecessary rebuilding of both application code and other packages when there are minor configuration changes. A `#include` is needed only when a configuration option affects the exported interface, or when it affects some implementation details which is controlled by the header file such as whether or not a particular function gets inlined.

There are a couple of ways in which the problem of unnecessary rebuilding could be addressed. The first would require more intelligent handling of header file dependency handling by the tools (especially the compiler) and the build system. This would require changes to various non-eCos tools. An alternative approach would be to support finer-grained configuration header files, for example there could be a file `pkgconf/libc/inline.h` controlling which functions should be inlined. This could be achieved by some fairly simple extensions to the component framework, but it makes it more difficult to get the package header files and source code correct: a C preprocessor `#ifdef` directive does not distinguish between a symbol not being defined because the option is disabled, or the symbol not being defined because the appropriate configuration header file has not been `#include'd`. It is likely that a cross-referencing tool would have to be developed first to catch problems like this, before the component framework could support finer-grained configuration headers.

## Package Documentation

On-line package documentation should be in HTML format. The component framework imposes no special limitations: component writers can decide which version of the HTML specification should be followed; they can also decide on how best to cope with the limitations of different browsers. In general it is a good idea to keep things simple.

## Test Cases

Packages should normally come with one or more test cases. This allows application developers to verify that a given package works correctly on their particular hardware and in their particular configuration, making it slightly more likely that they will attempt to find bugs in their own code rather than automatically blaming the component writers.

At the time of writing the application developer support for building and running test cases via the component framework is under review and likely to change. Currently each test case should consist of a single C or C++ source file that can be compiled with the package's set of compiler flags and linked like any application program. Each test case should use the testing API defined by the infrastructure. A magically-named calculated configuration option of the form `CYGPKG_<PACKAGE-NAME>_TESTS` lists the test cases.

## Host-side Support

On occasion it would be useful for an eCos package to be shipped with host-side support. This could take the form of an additional tool needed to build that package. It could be an application intended to communicate with the target-side package code and display monitoring information. It could be a utility needed for running the package test cases, especially in the case of device drivers. The component framework does not yet provide any such support for host-side software, and there are obvious issues related to portability to the different machines that can be used for hosts. This issue may get addressed in some future release. In some cases custom build steps can be subverted to do things on the host side rather than the target side, but this is not recommended.

## Making a Package Distribution

Developers of new eCos packages are advised to distribute their packages in the form of eCos package distribution files. Packages distributed in this format may be added to existing eCos component repositories in a robust manner using the Package Administration Tool. This chapter describes the format of package distribution files and details how to prepare an eCos package for distribution in this format.

### The eCos package distribution file format

eCos package distribution files are gzipped GNU tar archives which contain both the source code for one or more eCos packages and a data file containing package information to be added to the component repository database. The distribution files are subject to the following rules:

- a. The data file must be named `pkgadd.db` and must be located in the root of the tar archive. It must contain data in a format suitable for appending to the eCos repository database (`ecos.db`). [the Section called \*Updating the ecos.db database\* in Chapter 3](#) describes this data format. Note that a database consistency check is performed by the eCos Administration Tool when `pkgadd.db` has been appended to the database. Any new target entries which refer to unknown packages will be removed at this stage.
- b. The package source code must be placed in one or more `<package-path>/<version>` directories in the tar archive, where each `<package-path>` directory path is specified as the directory attribute of one of the packages entries in `pkgadd.db`.
- c. An optional license agreement file named `pkgadd.txt` may be placed in the root of the tar archive. It should contain text with a maximum line length of 79 characters. If this file exists, the contents will be presented to the user during installation of the package. The eCos Package Administration Tool will then prompt the user with the question "Do you accept all the terms of the preceding license agreement?". The user must respond **"yes"** to this prompt in order to proceed with the installation.
- d. Optional template files may be placed in one or more `templates/<template_name>` directories in the tar archive. Note that such template files would be appropriate only where the packages to be distributed have a complex dependency relationship with other packages. Typically, a third party package can be simply added to an eCos configuration based on an existing core template and the provision of new templates would not be appropriate. [the Section called \*Templates\* in Chapter 6](#) contains more information on templates.
- e. The distribution file must be given a `.epk` (not `.tar.gz`) file extension. The `.epk` file extension serves to distinguish eCos package distributions files from generic gzipped GNU tar archives. It also discourages users from attempting to extract the package from the archive manually. The file browsing dialog of the eCos Package Administration Tool lists only those files which have a `.epk` extension.
- f. No other files should be present in the archive.
- g. Files in the tar archive may use LF or CRLF line endings interchangeably. The eCos Administration Tool ensures that the installed files are given the appropriate host-specific line endings.
- h. Binary files may be placed in the archive, but the distribution of object code is not recommended. All binary files must be given a `.bin` suffix in addition to any file extension they may already have. For example, the GIF image file `myfile.gif` must be named `myfile.gif.bin` in the archive. The `.bin` suffix is removed during file extraction and is used to inhibit the manipulation of line endings by the eCos Administration Tool.

## Preparing eCos packages for distribution

Development of new eCos packages or new versions of existing eCos packages will take place in the context of an existing eCos component repository. This section details the steps involved in extracting new packages from a repository and generating a corresponding eCos package distribution file for distribution of the packages to other eCos users. The steps required are as follows:

- a. Create a temporary directory `$PKGTMP` for manipulation of the package distribution file contents and copy the source files of the new packages into this directory, preserving the relative path to the package. In the case of a new package at `mypkg/current` in the repository:

```
$ mkdir -p $PKGTMP/mypkg
$ cp -p -R $ECOS_REPOSITORY/mypkg/current $PKGTMP/mypkg
```

Where more than one package is to be distributed in a single package distribution file, copy each package in the above manner. Note that multiple packages distributed in a single package distribution file cannot be installed separately. Where such flexibility is required, distribution of each new package in separate package distribution files is recommended.

- b. Copy any template files associated with the distributed packages into the temporary directory, preserving the relative path to the template. For example:

```
$ mkdir -p $PKGTMP/templates
$ cp -p -R $ECOS_REPOSITORY/templates/mytemplate $PKGTMP/templates
```

- c. Remove any files from the temporary directory hierarchy which you do not want to distribute with the packages (eg object files, CVS directories).
- d. Add a `.bin` suffix to the name of any binary files. For example, if the packages contains GIF image files (`*.gif`) for documentation purposes, such files must be renamed to `*.gif.bin` as follows:

```
$ find $PKGTMP -type f -name '*.gif' -exec mv {} {}.bin ';' 
```

The `.bin` suffix is removed during file extraction and is used to inhibit the manipulation of line endings by the eCos Package Administration Tool.

- e. Extract the package records for the new packages from the package database file at `$ECOS_REPOSITORY/ecos.db` and create a new file containing these records at `$PKGTMP/pkgadd.db` (in the root of the temporary directory hierarchy). Any target records which reference the distributed packages must also be provided in `pkgadd.db`.
- f. Rename the version directories under `$PKGTMP` (typically `current` during development) to reflect the versions of the packages you are distributing. For example, version 1.0 of a package may use the version directory name `v1_0`:

```
$ cd $PKGTMP/mypkg
$ mv current v1_0
```

the [Section called \*Package Versioning\*](#) describes the version naming conventions.

- g. Rename any template files under `$PKGTMP` (typically `current.ect` during development) to reflect the version of the template you are distributing. For example, version 1.0 of a template may use the filename `v1_0.ect`:

```
$ cd $PKGTMP/templates/mytemplate
$ mv current.ect v1_0.ect
```

It is also important to edit the contents of the template file, changing the version of each referenced package to match that of the packages you are distributing. This step will eliminate version warnings during the subsequent loading of the template.

- h. Optionally create a licence agreement file at `$PKGTMP/pkgadd.txt` containing the licensing terms under which you are distributing the new packages. Limit each line in this file to a maximum of 79 characters.
- i. Create a GNU tar archive of the temporary directory hierarchy. By convention, this archive would have a name of the form `<package_name>-<version>`:

```
$ cd $PKGTMP
$ tar cf mypkg-1.0.tar *
```

Note that non-GNU version of tar may create archive files which exhibit subtle incompatibilities with GNU tar. For this reason, always use GNU tar to create the archive file.

- j. Compress the archive using gzip and give the resulting file a `.epk` file extension:

```
$ gzip mypkg-1.0.tar
$ mv mypkg-1.0.tar.gz mypkg-1.0.epk
```

The resulting eCos package distribution file (`*.epk`) is in a compressed format and may be distributed without further compression.



# Chapter 3. The CDL Language

The CDL language is a key part of the eCos component framework. All packages must come with at least one CDL script, to describe that package to the framework. The information in that script includes details of all the configuration options and how to build the package. Implementing a new component or turning some existing code into an eCos component always involves writing corresponding CDL. This chapter provides a description of the CDL language. Detailed information on specific parts of the language can be found in [Chapter 5](#).

## Language Overview

A very simple CDL script would look like this:

```
cdl_package CYGPKG_ERROR {
    display      "Common error code support"
    compile      strerror.cxx
    include_dir   cyg/error
    description   "
        This package contains the common list of error and
        status codes. It is held centrally to allow
        packages to interchange error codes and status
        codes in a common way, rather than each package
        having its own conventions for error/status
        reporting. The error codes are modelled on the
        POSIX style naming e.g. EINVAL etc. This package
        also provides the standard strerror() function to
        convert error codes to textual representation."
}
```

This describes a single package, the error code package, which does not have any sub-components or configuration options. The package has an internal name, `CYGPKG_ERROR`, which can be referenced in other CDL scripts using e.g. `requires CYGPKG_ERROR`. There will also be a `#define` for this symbol in a configuration header file. In addition to the package name, this script provides a number of properties for the package as a whole. The `display` property provides a short description. The `description` property involves a rather longer one, for when users need a bit more information. The `compile` and `include_dir` properties list the consequences of this package at build-time. The package appears to lack any on-line documentation.

Packages could be even simpler than this. If the package only provides an interface and there are no files to be compiled then there is no need for a `compile` property. Alternatively if there are no exported header files, or if the exported header files should go to the top-level of the `install/include` directory, then there is no need for an `include_dir` property. Strictly speaking the `description` and `display` properties are optional as well, although application developers would not appreciate the resulting lack of information about what the package is supposed to do.

However many packages tend to be a bit more complicated than the error package, containing various sub-components and configuration options. These are also defined in the CDL scripts and in much the same way as the package. For example, the following excerpt comes from the infrastructure package:

```
cdl_component CYGDBG_INFRA_DEBUG_TRACE_ASSERT_BUFFER {
    display      "Buffered tracing"
    default_value 1
```

```

    active_if      CYGDBG_USE_TRACING
    description    "
        An output module which buffers output from tracing and
        assertion events. The stored messages are output when an
        assert fires, or CYG_TRACE_PRINT() (defined in
        <cyg/infra/cyg_trac.h>) is called. Of course, there will
        only be stored messages if tracing per se (CYGDBG_USE_TRACING)
        is enabled above."

    cdl_option CYGDBG_INFRA_DEBUG_TRACE_BUFFER_SIZE {
        display      "Trace buffer size"
        flavor       data
        default_value 32
        legal_values 5 to 65535
        description  "
            The size of the trace buffer. This counts the number of
            trace records stored. When the buffer fills it either
            wraps, stops recording, or generates output."
    }

    ...
}

```

Like a `cdl_package`, a `cdl_component` has a name and a body. The body contains various properties for that component, and may also contain sub-components or options. Similarly a `cdl_option` has a name and a body of properties. This example lists a number of new properties: `default_value`, `active_if`, `flavor` and `legal_values`. The meaning of most of these should be fairly obvious. The next sections describe the various CDL commands and properties.

There is one additional and very important point: CDL is not a completely new language; instead it is implemented as an extension of the existing Tcl scripting language. The syntax of a CDL script is Tcl syntax, which is described below. In addition some of the more advanced facilities of CDL involve embedded fragments of Tcl code, for example there is a `define_proc` property which specifies some code that needs to be executed when the component framework generates the configuration header files.

## CDL Commands

There are four CDL-related commands which can occur at the top-level of a CDL script: `cdl_package`, `cdl_component`, `cdl_option` and `cdl_interface`. These correspond to the basic building blocks of the language (CDL interfaces are described in [the Section called \*Interfaces\*](#)). All of these take the same basic form:

```

cdl_package <name> {
    ...
}

cdl_component <name> {
    ...
}

cdl_option <name> {

```



```

    ...
}

cdl_interface <name> {
    ...
}

```

The command is followed by a name and by a body of properties, the latter enclosed in braces. Packages and components can contain other entities, so the `cdl_package` and `cdl_component` can also have nested commands in their bodies. All names must be unique within a given configuration. If say the C library package and a TCP/IP stack both defined an option with the same name then it would not be possible to load both of them into a single configuration. There is a [naming convention](#) which should make accidental name clashes very unlikely.

It is possible for two packages to use the same name if there are no reasonable circumstances under which both packages could be loaded at the same time. One example would be architectural HAL packages: a given eCos configuration can be used on only one processor, so the architectural HAL packages `CYGPKG_HAL_ARM` and `CYGPKG_HAL_I386` can re-use option names; in fact in some cases they are expected to.

Each package has one top-level CDL script, which is specified in the packages [ecos.db database entry](#). Typically the name of this top-level script is related to the package, so the kernel package uses `kernel.cdl`, but this is just a convention. The first command in the top-level script should be `cdl_package`, and the name used should be the same as in the `ecos.db` database. There should be only one `cdl_package` command per package.

The various CDL entities live in a hierarchy. For example the kernel package contains a scheduling component, a synchronization primitives component, and a number of others. The synchronization component contains various options such as whether or not mutex priority inheritance is enabled. There is no upper bound on how far components can be nested, but it is rarely necessary to go more than three or four levels deeper than the package level. Since the naming convention incorporates bits of the hierarchy, this has the added advantage of keeping the names down to a more manageable size.

The hierarchy serves two purposes. It allows options to be controlled en masse, so disabling a component automatically disables all the options below it in the hierarchy. It also permits a much simpler representation of the configuration in the graphical configuration tool, facilitating navigation and modification.

By default a package is placed at the top-level of the hierarchy, but it is possible to override this using a parent property. For example an architectural HAL package such as `CYGPKG_HAL_SH` typically re-parents itself below `CYGPKG_HAL`, and a platform HAL package would then re-parent itself below the architectural HAL. This makes it a little bit easier for users to navigate around the hierarchy. Components, options and interfaces can also be re-parented, but this is less common.

All components, options and interfaces that are defined directly in the top-level script will be placed below the package in the hierarchy. Alternatively they can be nested in the body of the `cdl_package` command. The following two script fragments are equivalent:

```

cdl_package CYGPKG_LIBC {
    ...
}

cdl_component CYGPKG_LIBC_STRING {
    ...
}

cdl_option CYGPKG_LIBC_CTYPE_INLINE {

```

```

    ...
}

and:

cdl_package CYGPKG_LIBC {
    ...

    cdl_component CYGPKG_LIBC_STRING {
        ...
    }

    cdl_option CYGPKG_LIBC_CTYPE_INLINE {
        ...
    }
}

```

If a script defines options both inside and outside the body of the `cdl_package` then the ones inside will be processed first. Language purists may argue that it would have been better if all contained options and components had to go into the body, but in practice it is often convenient to be able to skip this level of nesting and the resulting behavior is still well-defined.

Components can also contain options and other CDL entities, in fact that is what distinguishes them from options. These can be defined in the body of the `cdl_component` command:

```

cdl_component CYGPKG_LIBC_STDIO {

    cdl_component CYGPKG_LIBC_STDIO_FLOATING_POINT {
        ...
    }

    cdl_option CYGSEM_LIBC_STDIO_THREAD_SAFE_STREAMS {
        ...
    }
}

```

Nesting options inside the bodies of components like this is fine for simple packages with only a limited number of configuration options, but it becomes unsatisfactory as the number of options increases. Instead it is possible to split the CDL data into multiple CDL scripts, on a per-component basis. The `script` property should be used for this. For example, in the case of the C library all `stdio`-related configuration options could be put into `stdio.cdl`, and the top-level CDL script `libc.cdl` would contain the following:

```

cdl_package CYGPKG_LIBC {
    ...

    cdl_component CYGPKG_LIBC_STDIO {
        ...
        script stdio.cdl
    }
}

```

The `CYGPKG_LIBC_STDIO_FLOATING_POINT` component and the `CYGSEM_LIBC_STDIO_THREAD_SAFE_STREAMS` option can then be placed at the top-level of `stdio.cdl`. It is possible to have some options nested in the body of

a `cdl_component` command and other options in a separate file accessed by the script property. In such a case the nested options would be processed first, and then the other script would be read in. A script specified by a script property should only define new options, components or interfaces: it should not contain any additional properties for the current component.

It is possible for a component's CDL script to have a sub-component which also has a script property, and so on. In practice excessive nesting like this is rarely useful. It is also possible to ignore the CDL language support for constructing hierarchies automatically and use the parent property explicitly for every single option and component. Again this is not generally useful.

**Note:** At the time of writing interfaces cannot act as containers. This may change in a future version of the component framework. If the change is made then interfaces would support the script property, just like components.

## CDL Properties

Each package, component, option, and interface has a body of properties, which provide the component framework with information about how to handle each option. For example there is a property for a descriptive text message which can be displayed to a user who is trying to figure out just what effect manipulating the option would have on the target application. There is another property for the default value, for example whether a particular option should be enabled or disabled by default.

All of the properties are optional, it is legal to define a configuration option which has an empty body. However some properties are more optional than others: users will not appreciate having to manipulate an option if they are not given any sort of description or documentation. Other properties are intended only for very specific purposes, for example `make_object` and `include_files`, and are used only rarely.

Because different properties serve very different purposes, their syntax is not as uniform as the top-level commands. Some properties take no arguments at all. Other properties take a single argument such as a description string, or a list of arguments such as a `compile` property which specifies the file or files that should be compiled if a given option is active and enabled. The `define_proc` property takes as argument a snippet of Tcl code. The `active_if`, `calculated`, `default_value`, `legal_values` and `requires` properties take various expressions. Additional properties may be defined in future which take new kinds of arguments.

All property parsing code supports options for every property, although at present the majority of properties do not yet take any options. Any initial arguments that begin with a hyphen character - will be interpreted as an option, for example:

```
cdl_package CYGPKG_HAL_ARM {
    ...
    make -priority 1 {
        ...
    }
}
```

If the option involves additional data, as for the `-priority` example above, then this can be written as either `-priority=1` or as `-priority 1`. On occasion the option parsing code can get in the way, for example:

```
cdl_option CYGNUM_LIBC_TIME_DST_DEFAULT_STATE {
    ...
    legal_values -1 to 1
    default_value -1
}
```

Neither the `legal_values` nor the `default_value` property will accept `-1` as a valid option, so this will result in syntax errors when the CDL script is read in by the component framework. To avoid problems, the option parsing code will recognize the string `--` and will not attempt to interpret any subsequent arguments. Hence this option should be written as:

```
cdl_option CYGNUM_LIBC_TIME_DST_DEFAULT_STATE {
    ...
    legal_values  -- -1 to 1
    default_value  -- -1
}
```

The property parsing code involves a recursive invocation of the Tcl interpreter that is used to parse the top-level commands. This means that some characters in the body of an option will be treated specially. The `#` character can be used for comments. The backslash character `\`, the dollar character `$`, square brackets `[` and `]`, braces `{` and `}`, and the quote character `"` may all receive special treatment. Most of the time this is not a problem because these characters are not useful for most properties. On occasion having a Tcl interpreter around performing the parser can be very powerful. For more details of how the presence of a Tcl interpreter can affect CDL scripts, see [the Section called \*An Introduction to Tcl\*](#).

Many of the properties can be used in any of `cdl_package`, `cdl_component`, `cdl_option` or `cdl_interface`. Other properties are more specific. The `script` property is only relevant to components. The `define_header`, `hardware`, `include_dir`, `include_files`, and `library` properties apply to a package as a whole, so can only occur in the body of a `cdl_package` command. The `calculated`, `default_value`, `legal_values` and `flavor` properties are not relevant to packages, as will be explained later. The `calculated` and `default_value` properties are also not relevant to interfaces.

This section lists the various properties, grouped by purpose. Each property also has a full reference page in [Chapter 5](#). Properties related to values and expressions are described in more detail in [the Section called \*Values and Expressions\*](#). Properties related to header file generation and to the build process are described in [Chapter 4](#).

## Information-providing Properties

Users can only be expected to manipulate configuration options sensibly if they are given sufficient information about these options. There are three properties which serve to explain an option in plain text: the [display](#) property gives a textual alias for an option, which is usually more comprehensible than something like `CYGPKG_LIBC_TIME_ZONES`; the [description](#) property gives a longer description, typically a paragraph or so; the [doc](#) property specifies the location of additional on-line documentation related to a configuration option. In the context of a graphical tool the display string will be the primary way for users to identify configuration options; the description paragraph will be visible whenever the option is selected; the on-line documentation will only be accessed when the user explicitly requests it.

```
cdl_package CYGPKG_UITRON {
    display      "uITRON compatibility layer"
    doc          ref/ecos-ref.a.html
    description  "
        eCos supports a uITRON Compatibility Layer, providing
```

```

    full Level S (Standard) compliance with Version 3.02 of
    the uITRON Standard, plus many Level E (Extended) features.
    uITRON is the premier Japanese embedded RTOS standard."
    ...
}

```

All three properties take a single argument. For display and description this argument is just a string. For doc it should be a pointer to a suitable HTML file, optionally including an anchor within that page. If the [directory layout conventions](#) are observed then the component framework will look for the HTML file in the package's doc sub-directory, otherwise the doc filename will be treated as relative to the package's top-level directory.

## The Configuration Hierarchy

There are two properties related to the hierarchical organization of components and options: [parent](#) and [script](#).

The parent property can be used to move a CDL entity somewhere else in the hierarchy. The most common use is for packages, to avoid having all the packages appear at the top-level of the configuration hierarchy. For example an architectural HAL package such as CYGPKG\_HAL\_SH is placed below the common HAL package CYGPKG\_HAL using a parent property.

```

cdl_package CYGPKG_HAL_SH {
    display      "SH architecture"
    parent      CYGPKG_HAL
    ...
}

```

The parent property can also be used in the body of a `cdl_component`, `cdl_option` or `cdl_interface`, but this is less common. However care has to be taken since excessive re-parenting can be confusing. Care also has to be taken when reparenting below some other package that may not actually be loaded in a given configuration, since the resulting behavior is undefined.

As a special case, if the parent is the empty string then the CDL entity is placed at the root of the hierarchy. This is useful for global preferences, default compiler flags, and other settings that may affect every package.

The script property can only be used in the body of a `cdl_component` command. The property takes a single filename as argument, and this should be another CDL script containing additional options, sub-components and interfaces that should go below the current component in the hierarchy. If the [directory layout conventions](#) are observed then the component framework will look for the specified file relative to the `cdl` subdirectory of the package, otherwise the filename will be treated as relative to the package's top-level directory.

```

cdl_component CYGPKG_LIBC_STDIO {
    display      "Standard input/output functions"
    flavor      bool
    requires     CYGPKG_IO
    requires     CYGPKG_IO_SERIAL_HALDIAG
    default_value 1
    description  "
        This enables support for standard I/O functions from <stdio.h>."

    script      stdio.cdl
}

```

## Value-related Properties

There are seven properties which are related to option values and state: [flavor](#), [calculated](#), [default\\_value](#), [legal\\_values](#), [active\\_if](#), [implements](#), and [requires](#). More detailed information can be found in [the Section called Values and Expressions](#).

In the context of configurability, the concept of an option's value is somewhat non-trivial. First an option may or may not be loaded: it is possible to build a configuration which has the math library but not the kernel; however the math library's CDL scripts still reference kernel options, for example `CYGSEM_LIBM_THREAD_SAFE_COMPAT_MODE` has a `requires` constraint on `CYGVAR_KERNEL_THREADS_DATA`. Even if an option is loaded it may or may not be active, depending on what is happening higher up in the hierarchy: if the C library's `CYGPKG_LIBC_STDIO` component is disabled then some other options such as `CYGNUM_LIBC_STDIO_BUFSIZE` become irrelevant. In addition each option has both a boolean enabled/disabled flag and a data part. For many options only the boolean flag is of interest, while for others only the data part is of interest. The `flavor` property can be used to control this:

`flavor none`

This flavor indicates that neither the boolean nor the data parts are user-modifiable: the option is always enabled and the data is always set to 1. The most common use for this is to have a component that just acts as a placeholder in the hierarchy, allowing various options to be grouped below it.

`flavor bool`

Only the boolean part of the option is user-modifiable. The data part is fixed at 1.

`flavor data`

Only the data part of the option is user-modifiable. The boolean part is fixed at enabled.

`flavor booldata`

Both the boolean and the data part of the option are user-modifiable.

For more details of CDL flavors and how a flavor affects expression evaluation, and other consequences, see [the Section called Values and Expressions](#). The `flavor` property cannot be used for a package because packages always have the `booldata` flavor. Options and components have the `bool` flavor by default, since most configuration choices are simple yes-or-no choices. Interfaces have the `data` flavor by default.

The `calculated` property can be used for options which should not be user-modifiable, but which instead are fixed by the target hardware or determined from the current values of other options. In general calculated options should be avoided, since they can be confusing to users who need to figure out whether or not a particular option can actually be changed. There are a number of valid uses for calculated options, and quite a few invalid ones as well. The [reference packages](#) should be consulted for further details. The property takes an [ordinary CDL expression](#) as argument, for example:

```
# A constant on some target hardware, perhaps user-modifiable on other
# targets.
cdl_option CYGNUM_HAL_RTC_PERIOD {
    display      "Real-time clock period"
    flavor       data
    calculated    12500
}
```

The `calculated` property cannot be used for packages or interfaces. The value of a package always corresponds to the version of that package which is loaded, and this is under user control. Interfaces are implicitly calculated, based on the number of active and enabled implementors.

The `default_value` property is similar to `calculated`, but only specifies a default value which users can modify. Again this property is not relevant to packages or interfaces. A typical example would be:

```
cdl_option CYGDBG_HAL_DEBUG_GDB_THREAD_SUPPORT {
    display      "Include GDB multi-threading debug support"
    requires     CYGDBG_KERNEL_DEBUG_GDB_THREAD_SUPPORT
    default_value CYGDBG_KERNEL_DEBUG_GDB_THREAD_SUPPORT
    ...
}
```

The `legal_values` property imposes a constraint on the possible values of the data part of an option. Hence it is only applicable to options with the `data` or `booldata` flavors. It cannot be used for a package since the only valid value for a package is its version number. The arguments to the `legal_values` property should constitute a [CDL list expression](#).

```
cdl_option CYGNUM_LIBC_TIME_STD_DEFAULT_OFFSET {
    display      "Default Standard Time offset"
    flavor       data
    legal_values -- -90000 to 90000
    default_value -- 0
    ...
}
```

The `active_if` property does not relate directly to an option's value, but rather to its active state. Usually this is controlled via the configuration hierarchy: if the `CYGPKG_LIBC_STDIO` component is disabled then all options below it are inactive and do not have any consequences. In some cases the hierarchy does not provide sufficient control, for example an option should only be active if two disjoint sets of conditions are satisfied: the hierarchy could be used for one of these conditions, and an additional `active_if` property could be used for the other one. The arguments to `active_if` should constitute a [CDL goal expression](#).

```
# Do not provide extra semaphore debugging if there are no semaphores
cdl_option CYGDBG_KERNEL_INSTRUMENT_BINSEM {
    active_if CYGPKG_KERNEL_SYNC
    ...
}
```

The `implements` property is related to the concept of [CDL interfaces](#). If an option is active and enabled and it implements a particular interface then it contributes 1 to that interface's value.

```
cdl_package CYGPKG_NET_EDB7XXX_ETH_DRIVERS {
    display      "Cirrus Logic ethernet driver"
    implements   CYGHWR_NET_DRIVERS
    implements   CYGHWR_NET_DRIVER_ETH0
    ...
}
```

The `requires` property is used to impose constraints on the user's choices. For example it is unreasonable to expect the C library to provide thread-safe implementations of certain functions if the underlying kernel support has been disabled, or even if the kernel is not being used at all.

```
cdl_option CYGSEM_LIBC_PER_THREAD_ERRNO {
    display      "Per-thread errno"
    doc          ref/ecos-ref.15.html
    requires     CYGVAR_KERNEL_THREADS_DATA
    default_value 1
    ...
}
```

The arguments to the `requires` property should be a [CDL goal expression](#).

## Generating the Configuration Header Files

When creating or updating a build tree the component framework will also generate configuration header files, one per package. By default it will generate a `#define` for each option, component or interface that is active and enabled. For options with the `data` or `booldata` flavors the `#define` will use the option's data part, otherwise it will use the constant 1. Typical output would include:

```
#define CYGFUN_LIBC_TIME_POSIX 1
#define CYGNUM_LIBC_TIME_DST_DEFAULT_STATE -1
```

There are six properties which can be used to control the header file generation process: [define\\_header](#), [no\\_define](#), [define\\_format](#), [define](#), [if\\_define](#), and [define\\_proc](#).

By default the component framework will generate a configuration header file for each package based on the package's name: everything up to and including the first underscore is discarded, the rest of the name is lower-cased, and a `.h` suffix is appended. For example the configuration header file for the kernel package `CYGPKG_KERNEL` is `pkgconf/kernel.h`. The `define_header` property can be used to specify an alternative filename. This applies to all the components and options within a package, so it can only be used in the body of a `cdl_package` command. For example the following specifies that the configuration header file for the SPARClite HAL package is `pkgconf/hal_sparclite.h`.

```
cdl_package CYGPKG_HAL_SPARCLITE {
    display "SPARClite architecture"
    parent      CYGPKG_HAL
    hardware
    define_header hal_sparclite.h
    ...
}
```

**Note:** At present the main use for the `define_header` property is related to hardware packages, see the [reference pages](#) for more details.

The `no_define` property is used to suppress the generation of the default `#define`. This can be useful if an option's consequences are all related to the build process or to constraints, and the option is never actually checked in any



source code. It can also be useful in conjunction with the `define`, `if_define` or `define_proc` properties. The `no_define` property does not take any arguments.

```
cdl_component CYG_HAL_STARTUP {
    display      "Startup type"
    flavor       data
    legal_values { "RAM" "ROM" }
    default_value {"RAM"}
    no_define
    define -file system.h CYG_HAL_STARTUP
    ...
}
```

This example also illustrates the `define` property, which can be used to generate a `#define` in addition to the default one. It takes a single argument, the name of the symbol to be defined. It also takes options to control the configuration header file in which the symbol should be defined and the format to be used.

The `define_format` property can be used to control how the value part of the default `#define` gets formatted. For example a format string of `"0x%04x"` could be used to generate a four-digit hexadecimal number.

The `if_define` property is intended for use primarily to control assertions, tracing, and similar functionality. It supports a specific implementation model for these, allowing control at the grain of packages or even individual source files. The [reference pages](#) provide additional information.

The `define_proc` property provides an escape mechanism for those cases where something special has to happen at configuration header file generation time. It takes a single argument, a fragment of Tcl code, which gets executed when the header file is generated. This code can output arbitrary data to the header file, or perform any other actions that might be appropriate.

## Controlling what gets Built

There are six properties which affect the build process: [compile](#), [make](#), [make\\_object](#), [library](#), [include\\_dir](#), and [include\\_files](#). The last three apply to a package as a whole, and can only occur in the body of a `cdl_package` command.

Most of the source files that go into a package should simply be compiled with the appropriate compiler, selected by the target architecture, and with the appropriate flags, with an additional set defined by the target hardware and possible modifications on a per-package basis. The resulting object files will go into the library `libtarget.a`, which can then be linked against application code. The `compile` property is used to list these source files:

```
cdl_package CYGPKG_ERROR {
    display      "Common error code support"
    compile      strerror.cxx
    include_dir   cyg/error
    ...
}
```

The arguments to the `compile` property should be one or more source files. Typically most of the sources will be needed for the package as a whole, and hence they will be listed in one or more `compile` properties in the body of the `cdl_package`. Some sources may be specific to particular configuration options, in other words there is no point in compiling them unless that option is enabled, in which case the sources should be listed in a `compile` property in the corresponding `cdl_option`, `cdl_component` or `cdl_interface` body.

Some packages may have more complicated build requirements, for example they may involve a special target such as a linker script which should not end up in the usual library, or they may involve special build steps for generating an object file. The `make` and `make_object` properties provide support for such requirements, for example:

```
cdl_package CYGPKG_HAL_MN10300_AM33 {
  display      "MN10300 AM33 variant"
  ...
  make {
    <PREFIX>/lib/target.ld: <PACKAGE>/src/mn10300_am33.ld
    $(CC) -E -P -Wp,-MD,target.tmp -DEXTRAS=1 -xc $(INCLUDE_PATH) \
      $(CFLAGS) -o $@ $<
    @echo $@ ": \\" > $(notdir $@).deps
    @tail +2 target.tmp >> $(notdir $@).deps
    @echo >> $(notdir $@).deps
    @rm target.tmp
  }
}
```

For full details of custom build steps and the build process generally, see [Chapter 4](#).

By default all object files go into the library `libtarget.a`. It is possible to override this at the package level using the `library` property, but this should be avoided since it complicates application development: instead of just linking with a single library for all eCos-related packages, it suddenly becomes necessary to link with several libraries.

The `include_dir` and `include_files` properties relate to a package's exported header files. By default a package's header files will be exported to the `install/include` directory. This is the desired behavior for some packages like the C library, since headers like `stdio.h` should exist at that level. However if all header files were to end up in that directory then there would be a significant risk of a name clash. Instead it is better for packages to specify some sub-directory for their exported header files, for example:

```
cdl_package CYGPKG_INFRA {
  display      "Infrastructure"
  include_dir  cyg/infra
  ...
}
```

The various header files exported by the infrastructure, for example `cyg_ass.h` and `cyg_trac.h` will now end up in the `install/include/cyg/infra` sub-directory, where a name clash is very unlikely.

For packages which follow the [directory layout conventions](#) the component framework will assume that the package's `include` sub-directory contains all exported header files. If this is not the case, for example because the package is sufficiently simple that the layout convention is inappropriate, then the exported header files can be listed explicitly in an `include_files` property.

## Miscellaneous Properties

The [hardware](#) property is only relevant to packages. Some packages such as device drivers and HAL packages are hardware-specific, and generally it makes no sense to add such packages to a configuration unless the corresponding hardware is present on your target system. Typically hardware package selection happens automatically when you select your target. The `hardware` property should be used to identify a hardware-specific package, and does not take any arguments.

```

cdl_package CYGPKG_HAL_MIPS {
    display "MIPS architecture"
    parent      CYGPKG_HAL
    hardware
    include_dir  cyg/hal
    define_header hal_mips.h
    ...
}

```

At present the hardware property is largely ignored by the component framework. This may change in future releases.

## Option Naming Convention

All the options in a given configuration live in the same namespace. Furthermore it is not possible for two separate options to have the same name, because this would make any references to those options in CDL expressions ambiguous. A naming convention exists to avoid problems. It is recommended that component writers observe some or all of this convention to reduce the probability of name clashes with other packages.

There is an important restriction on option names. Typically the component framework will output a `#define` for every active and enabled option, using the name as the symbol being defined. This requires that all names are valid C preprocessor symbols, a limitation that is enforced even for options which have the `no_define` property. Preprocessor symbols can be any sequence of lower case letters a-z, upper case letters, A-Z, the underscore character `_`, and the digits 0-9. The first character must be a non-digit. Using an underscore as the first character is discouraged, because that may clash with reserved language identifiers. In addition there is a convention that preprocessor symbols only use upper case letters, and some component writers may wish to follow this convention.

A typical option name could be something like `CYGSEM_KERNEL_SCHED_BITMAP`. This name consists of several different parts:

1. The first few characters, in this case the three letters `CYG`, are used to identify the organization that produced the package. For historical reasons packages produced by Red Hat tend to use the prefix `CYG` rather than `RHAT`. Component writers should use their own prefix: even when cutting and pasting from an existing CDL script the prefix should be changed to something appropriate to their organization.

It can be argued that a short prefix, often limited to upper case letters, is not sufficiently long to eliminate the possibility of name clashes. A longer prefix could be used, for example one based on internet domain names. However the C preprocessor has no concept of namespaces or `import` directives, so it would always be necessary to use the full option name in component source code which gets tedious - option names tend to be long enough as it is. There is a small increased risk of name clashes, but this risk is felt to be acceptable.

2. The next three characters indicate the nature of the option, for example whether it affects the interface or just the implementation. A list of common tags is given below.
3. The `KERNEL_SCHED` part indicates the location of the option within the overall hierarchy. In this case the option is part of the scheduling component of the kernel package. Having the hierarchy details as part of the option name can help in understanding configurable code and further reduces the probability of a name clash.

4. The final part, `BITMAP`, identifies the option itself.

The three-character tag is intended to provide some additional information about the nature of the option. There are a number of pre-defined tags. However for many options there is a choice: options related to the platform should normally use `HWR`, but numerical options should normally use `NUM`; a platform-related numerical option such as the size of an interrupt stack could therefore use either tag. There are no absolute rules, and it is left to component writers to interpret the following guidelines:

`xxxARC_`

The `ARC` tag is intended for options related to the processor architecture. Typically such options will only occur in architectural or variant HAL packages.

`xxxHWR_`

The `HWR` tag is intended for options related to the specific target board. Typically such options will only occur in platform HAL packages.

`xxxPKG_`

This tag is intended for packages or components, in other words options which extend the configuration hierarchy. Arguably a `COM` tag would be more appropriate for components, but this could be confusing because of the considerable number of computing terms that begin with `com`.

`xxxGLO_`

This is intended for global configuration options, especially preferences.

`xxxDBG_`

The `DBG` tag indicates that the option is in some way related to debugging, for example it may enable assertions in some part of the system.

`xxxTST_`

This tag is for testing-related options. Typically these do not affect actual application code, instead they control the interaction between target-side test cases and a host-side testing infrastructure.

`xxxFUN_`

This is for configuration options which affect the interface of a package. There are a number of related tag which are also interface-related. `xxxFUN_` is intended primarily for options that control whether or not one or more functions are provided by the package, but can also be used if none of the other interface-related tags is applicable.

`xxxVAR_`

This is analogous to `FUN` but controls the presence or absence of one or more variables or objects.

`xxxCLS_`

The `CLS` tag is intended only for packages that provide an object-oriented interface, and controls the presence or absence of an entire class.

`xxxMFN_`

This is also for object-orientated interfaces, and indicates the presence or absence of a member function rather than an entire class.

`xxxSEM_`

A `SEM` option does not affect the interface (or if it does affect the interface, this is incidental). Instead it is used for options which have a fundamental effect on the semantic behavior of a package. For example the choice of kernel schedulers is semantic in nature: it does not affect the interface, in particular the function `cyg_thread_create` exists irrespective of which scheduler has been selected. However it does have a major impact on the system's behavior.

`xxxIMP_`

`IMP` is for implementation options. These do not affect either the interface or the semantic behavior (with the possible exception of timing-related changes). A typical implementation option controls whether or not a particular function or set of functions should get inlined.

`xxxNUM_`

This tag is for numerical options, for example the number of scheduling priority levels.

`xxxDAT_`

This is for data items that are not numerical in nature, for example a device name.

`xxxBLD_`

The `BLD` tag indicates an option that affects the build process, for example compiler flag settings.

`xxxINT_`

This should normally be used for CDL interfaces, which is a language construct that is largely independent from the interface exported by a package via its header files. For more details of CDL interfaces see [the Section called \*Interfaces\*](#).

`xxxPRI_`

This tag is not normally used for configuration options. Instead it is used by CDL scripts to pass additional private information to the source code via the configuration header files, typically inside a `define_proc` property.

`xxxSRC_`

This tag is not normally used for configuration options. Instead it can be used by package source code to interact with such options, especially in the context of the `if_define` property.

There is one special case of a potential name clash that is worth mentioning here. When the component framework generates a configuration header file for a given package, by default it will use a name derived from the package name (the `define_header` property can be used to override this). The file name is constructed from the package name by removing everything up to and including the first underscore, converting the remainder of the name to lower case, and appending a `.h` suffix. For example the kernel package `CYGPKG_KERNEL` will involve a header file `pkgconf/kernel.h`. If a configuration contained some other package `XYZPKG_KERNEL` then this would attempt to use the same configuration header file, with unfortunate effects. Case sensitivity could introduce problems as well, so a package `xyzpkg_kernel` would involve the same problem. Even if the header file names preserved the case

of the package name, not all file systems are case sensitive. There is no simple solution to this problem. Changing the names of the generated configuration header files would involve a major incompatible change to the interface, to solve a problem which is essentially hypothetical in nature.

## An Introduction to Tcl

All CDL scripts are implemented as Tcl scripts, and are read in by running the data through a standard Tcl interpreter, extended with a small number of additional commands such as `cdl_option` and `cdl_component`. Often it is not necessary to know the full details of Tcl syntax. Instead it is possible to copy an existing script, perform some copy and paste operations, and make appropriate changes to names and to various properties. However there are also cases where an understanding of Tcl syntax is very desirable, for example:

```
cdl_option CYGDAT_UITRON_MEMPOOLFIXED_EXTERNS {
    display      "Externs for initialization"
    flavor       data
    default_value {"static char fpool1[ 2000 ], \\n\
                  fpool2[ 2000 ], \\n\
                  fpool3[ 2000 ];"}
    ...
}
```

This causes the `cdl_option` command to be executed, which in turn evaluates its body in a recursive invocation of the Tcl interpreter. When the `default_value` property is encountered the braces around the value part are processed by the interpreter, stopping it from doing further processing of the braced contents (except for backslash processing at the end of a line, that is special). In particular it prevents command substitution for `[ 2000 ]`. A single argument will be passed to the `default_value` command which expects a CDL expression, so the expression parsing code is passed the following:

```
"static char fpool1[ 2000 ], \\n fpool2[ 2000 ], \\n fpool3[ 2000 ];"
```

The CDL expression parsing code will treat this as a simple string constant, as opposed to a more complicated expression involving other options and various operators. The string parsing code will perform the usual backslash substitutions so the actual default value will be:

```
static char fpool1[ 2000 ], \
    fpool2[ 2000 ], \
    fpool3[ 2000 ];
```

If the user does not modify the option's value then the following will be generated in the appropriate configuration header file:

```
#define CYGDAT_UITRON_MEMPOOLFIXED_EXTERNS static char fpool1[ 2000 ], \
    fpool2[ 2000 ], \
    fpool3[ 2000 ];
```

Getting this desired result usually requires an understanding of both Tcl syntax and CDL expression syntax. Sometimes it is possible to substitute a certain amount of trial and error instead, but this may prove frustrating. It is also worth pointing out that many CDL scripts do not involve this level of complexity. On the other hand, some of the

more advanced features of the CDL language involve fragments of Tcl code, for example the `define_proc` property. To use these component writers will need to know about the full Tcl language as well as the syntax.

Although the current example may seem to suggest that Tcl is rather complicated, it is actually a very simple yet powerful scripting language: the syntax is defined by just eleven rules. On occasion this simplicity means that Tcl's behavior is subtly different from other languages, which can confuse newcomers.

When the Tcl interpreter is passed some data such as `puts Hello`, it splits this data into a command and its arguments. The command will be terminated by a newline or by a semicolon, unless one of the quoting mechanisms is used. The command and each of its arguments are separated by white space. So in the following example:

```
puts Hello
set x 42
```

This will result in two separate commands being executed. The first command is `puts` and is passed a single argument, `Hello`. The second command is `set` and is passed two arguments, `x` and `42`. The intervening newline character serves to terminate the first command, and a semi-colon separator could be used instead:

```
puts Hello;set x 42
```

Any white space surrounding the semicolon is just ignored because it does not serve to separate arguments.

Now consider the following:

```
set x Hello world
```

This is not valid Tcl. It is an attempt to invoke the `set` command with three arguments: `x`, `Hello`, and `world`. The `set` only takes two arguments, a variable name and a value, so it is necessary to combine the data into a single argument by quoting:

```
set x "Hello world"
```

When the Tcl interpreter encounters the first quote character it treats all subsequent data up to but not including the closing quote as part of the current argument. The quote marks are removed by the interpreter, so the second argument passed to the `set` command is just `Hello world` without the quote characters. This can be significant in the context of CDL scripts. For example:

```
cdl_option CYG_HAL_STARTUP {
    ...
    default_value "RAM"
}
```

The Tcl interpreter strips off the quote marks so the CDL expression parsing code sees `RAM` instead of `"RAM"`. It will treat this as a reference to some unknown option `RAM` rather than as a string constant, and the expression evaluation code will use a value of 0 when it encounters an option that is not currently loaded. Therefore the option `CYG_HAL_STARTUP` ends up with a default value of 0. Either braces or backslashes should be used to avoid this, for example `default_value { "RAM" }`.

**Note:** There are long-term plans to implement some sort of CDL validation utility `cdllint` which could catch common errors like this one.

A quoted argument continues until the closing quote character is encountered, which means that it can span multiple lines. Newline or semicolon characters do not terminate the current command in such cases. description properties usually make use of this:

```
cdl_package CYGPKG_ERROR {
    description    "
        This package contains the common list of error and
        status codes. It is held centrally to allow
        packages to interchange error codes and status
        codes in a common way, rather than each package
        having its own conventions for error/status
        reporting. The error codes are modelled on the
        POSIX style naming e.g. EINVAL etc. This package
        also provides the standard strerror() function to
        convert error codes to textual representation."
    ...
}
```

The Tcl interpreter supports much the same forms of backslash substitution as other common programming languages. Some backslash sequences such as `\n` will be replaced by the appropriate character. The sequence `\\` will be replaced by a single backslash. A backslash at the very end of a line will cause that backslash, the newline character, and any white space at the start of the next line to be replaced by a single space. Hence the following two Tcl commands are equivalent:

```
puts "Hello\nworld\n"
puts \
"Hello
world
"
```

If a description string needs to contain quote marks or other special characters then backslash escapes can be used. In addition to quote and backslash characters, the Tcl interpreter treats square brackets, the `$` character, and braces specially. Square brackets are used for command substitution, for example:

```
puts "The answer is [expr 6 * 9]"
```

When the Tcl interpreter encounters the square brackets it will treat the contents as another command that should be executed first, and the result of executing that is used when continuing to process the script. In this case the Tcl interpreter will execute the command `expr 6 * 9`, yielding a result of 42<sup>1</sup> and then the Tcl interpreter will execute `puts "The answer is 42"`. It should be noted that the interpreter performs only one level of substitution: if the result of performing command substitution performs further special characters such as square brackets then these will not be treated specially.

Command substitution will not prove useful for many CDL scripts, except for e.g. a `define_proc` property which involves a fragment of Tcl code. Potentially there are some interesting uses, for example to internationalize display strings. However care does have to be taken to avoid unexpected command substitution, for example if an option description involves square brackets then typically these would require backslash-escapes.

The `$` character is used in Tcl scripts to perform variable substitution:

```
set x [expr 6 * 9]
puts "The answer is $x"
```



Variable substitution, like command substitution, is unlikely to prove useful for many CDL scripts except in the context of Tcl fragments. If it is necessary to have a `$` character then a backslash escape may have to be used.

Braces are used to collect a sequence of characters into a single argument, just like quotes. The difference is that variable, command and backslash substitution do not occur inside braces (with the sole exception of backslash substitution at the end of a line). Therefore given a line in a CDL script such as:

```
default_value { "RAM" }
```

The braces are stripped off by the Tcl interpreter, leaving `"RAM"` which will be handled as a string constant by the expression parsing code. The same effect could be achieved using one of the following:

```
default_value \"RAM\"
default_value \" \"RAM\" \"
```

Generally the use of braces is less confusing. At this stage it is worth noting that the basic format of CDL data makes use of braces:

```
cdl_option <name> {
    ...
};
```

The `cdl_option` command is passed two arguments, a name and a body, where the body consists of everything inside the braces but not the braces themselves. This body can then be executed in a recursive invocation of the Tcl interpreter. If a CDL script contains mismatched braces then the interpreter is likely to get rather confused and the resulting diagnostics may be difficult to understand.

Comments in Tcl scripts are introduced by a hash character `#`. However, a hash character only introduces a comment if it occurs where a command is expected. Consider the following:

```
# This is a comment
puts "Hello" # world
```

The first line is a valid comment, since the hash character occurs right at the start where a command name is expected. The second line does not contain a comment. Instead it is an attempt to invoke the `puts` command with three arguments: `Hello`, `#` and `world`. These are not valid arguments for the `puts` command so an error will be raised. If the second line was rewritten as:

```
puts "Hello"; # world
```

then this is a valid Tcl script. The semicolon identifies the end of the current command, so the hash character occurs at a point where the next command would start and hence it is interpreted as the start of a comment.

This handling of comments can lead to subtle behavior. Consider the following:

```
cdl_option WHATEVER {
# This is a comment }
    default_value 0
    ...
}
```

Consider the way the Tcl interpreter processes this. The command name and the first argument do not pose any special difficulties. The opening brace is interpreted as the start of the next argument, which continues until a closing brace is encountered. In this case the closing brace occurs on the second line, so the second argument passed

to cdl\_option is \n # This is a comment. This second argument is processed in a recursive invocation of the Tcl interpreter and does not contain any commands, just a comment. Top-level script processing then resumes, and the next command that is encountered is default\_value. Since the parser is not currently processing a configuration option this is an error. Later on the Tcl interpreter would encounter a closing brace by itself, which is also an error.

For component writers who need more information about Tcl, especially about the language rather than the syntax, various resources are available. A reasonable starting point is the Scriptics developer web site (<http://www.tcl.tk/scripting/>).

## Values and Expressions

It is fairly reasonable to expect that enabling or disabling a configuration option such as CYGVAR\_KERNEL\_THREADS\_DATA in some way affects its *value*. This will have an effect on any expressions that reference this option such as requires CYGVAR\_KERNEL\_THREADS\_DATA. It will also affect the consequences of that option: how it affects the build process and what happens to any constraints that CYGVAR\_KERNEL\_THREADS\_DATA may impose (as opposed to constraints on this option imposed by others).

In a language like C the handling of variables is relatively straightforward. If a variable *x* gets referenced in an expression such as if (*x* != 0), and that variable is not defined anywhere, then the code will fail to build, typically with an unresolved error at link-time. Also in C a variable *x* does not live in any hierarchy, so its value for the purposes of expression evaluation is not affected by anything else. C variables also have a clear type such as int or long double.

In CDL things are not so straightforward.

### Option Values

There are four factors which go into an option's value:

1. An option may or may not be loaded.
2. If the option is loaded, it may or may not be active.
3. Even if the option is active, it may or may not be enabled.
4. If the option is loaded, active and enabled then it will have some associated data which constitutes its value.

### Is the Option Loaded?

At any one time a configuration will contain only a subset of all possible packages. In fact it is impossible to combine certain packages in a single configuration. For example architectural HAL packages should contain a set of options defining endianness, the sizes of basic data types and so on (many of which will of course be constant for any given architecture). Any attempt to load two architectural HAL packages into a configuration will fail because of the resulting name clash. Since CDL expressions can reference options in other packages, and often need to do so, it is essential to define the resulting behavior.

One complication is that the component framework does not know about every single option in every single package. Obviously it cannot know about packages from arbitrary third parties which have not been installed. Even for packages which have been installed, the current repository database does not hold details of

every option, only of the packages themselves. If a CDL expression contains a reference to some option `CYGSEM_KERNEL_SCHED_TIMESLICE` then the component framework will only know about this option if the kernel package is actually loaded into the current configuration. If the package is not loaded then theoretically the framework might guess that the option is somehow related to the kernel by examining the option name but this would not be robust: the option could easily be part of some other package that violates the naming convention.

Assume that the user is building a minimal configuration which does not contain the kernel package, but does have other packages which contain the following constraints:

```
requires CYGPKG_KERNEL
requires CYGPKG_KERNEL_THREADS_DATA
requires !CYGSEM_KERNEL_SCHED_TIMESLICE
```

Clearly the first constraint is not satisfied because the kernel is not loaded. The second constraint is also not satisfied. The third constraint is trivially satisfied: if there is no kernel then the kernel's timeslicing support cannot possibly be enabled.

Any options which are not in the current configuration are handled as follows:

1. Any references to that option will evaluate to 0, so `requires !CYGSEM_KERNEL_SCHED_TIMESLICE` will be satisfied but `requires CYGSEM_KERNEL_THREADS_DATA` will not be satisfied.
2. An option that is not loaded has no consequences on the build process. It cannot directly result in any `#define`'s in a configuration header file, nor in any files being compiled. This is only reasonable: if the option is not loaded then the component framework has no way of knowing about any compile or similar properties. An option that is not loaded can have indirect consequences by being referenced in CDL expressions.
3. An option that is not loaded cannot impose any constraints on the rest of the configuration. Again this is the only reasonable behavior: if the option is not loaded then any associated `requires` or `legal_values` properties will not be known.

## Is the Option Active

The next issue to consider is whether or not a particular option is active. Configuration options are organized in a hierarchy of components and sub-components. For example the C library package contains a component `CYGPKG_LIBC_STDIO` containing all the options related to standard I/O. If a user disables the component as a whole then all the options below it become inactive: it makes no sense to disable all stdio functionality and then manipulate the buffer sizes.

Inactive is not quite the same as disabled, although the effects are similar. The value of an inactive option is preserved. If the user modifies a buffer size option, then disables the whole stdio component, the buffer size value remains in case the stdio component is re-enabled later on. Some tools such as the graphical configuration tool will treat inactive options specially, for example such options may be grayed out.

The active or inactive state of an option may affect other packages. For example a package may use the `sprintf` function and require support for floating point conversions, a constraint that is not satisfied if the relevant option is inactive. It is necessary to define exactly what it means for an option to be inactive:

1. An option is inactive if its parent is either inactive or disabled. For example if `CYGPKG_LIBC_STDIO` is disabled then all the options and sub-components become inactive; since `CYGPKG_LIBC_STDIO_FLOATING_POINT` is now inactive, `CYGSEM_LIBC_STDIO_PRINTF_FLOATING_POINT` is inactive as well.

- Options may also be inactive as a result of an `active_if` property. This is useful if a particular option is only relevant if two or more disjoint sets of conditions need to be satisfied, since the hierarchical structure can only cope with at most one such set.
- If an option is inactive then any references to that option in CDL expressions will evaluate to 0. Hence a constraint of the form `requires CYGSEM_LIBC_STDIO_PRINTF_FLOATING_POINT` is not satisfied if the entire stdio component is disabled.
- An option that is inactive has no consequences on the build process. No `#define` will be generated. Any compile or similar properties will be ignored.
- An option that is inactive cannot impose any constraints on the rest of the configuration. For example `CYGSEM_LIBC_STDIO_PRINTF_FLOATING_POINT` has a dependency `requires CYGPKG_LIBM`, but if all of the stdio functionality is disabled then this constraint is ignored (although of course there may be other packages which have a dependency on `CYGPKG_LIBM`).

### Is the Option Enabled? What is the Data?

The majority of configuration options are boolean in nature, so the user can either enable or disable some functionality. Some options are different. For example `CYGNUM_LIBC_STDIO_BUFSIZE` is a number, and `CYGDAT_LIBC_STDIO_DEFAULT_CONSOLE` is a string corresponding to a device name. A few options like `CYGDAT_UITRON_TASK_EXTERNS` can get very complicated. CDL has to cope with this variety, and define the exact behavior of the system in terms of constraints and build-time consequences.

In CDL the value of an option consists of two parts. There is a boolean part, controlling whether or not the option is enabled. There is also a data part, providing additional information. For most options one of these parts is fixed, as controlled by the option's flavor property:

Flavor	Enabled	Data
none	Always enabled	1, not modifiable
bool	User-modifiable	1, not modifiable
data	Always enabled	User-modifiable
booldata	User-modifiable	User-modifiable

The effects of the boolean and data parts are as follows:

- If an option is disabled, in other words if the boolean part is false, then any references to that option in CDL expressions will evaluate to 0. This is the same behavior as for inactive options. The data part is not relevant. The `none` and `data` flavors specify that the option is always enabled, in which case this rule is not applicable.
- If an option is enabled then any references to that option in CDL expressions will evaluate to the option's data part. For two of the flavors, `none` and `bool`, this data part is fixed to the constant 1 which generally has the expected result.
- If a component or package is disabled then all sub-components and options immediately below it in the hierarchy are inactive. By a process of recursion this will affect all the nodes in the subtree.
- If an option is disabled then it can impose no constraints on the rest of the configuration, in particular `requires` and `legal_values` properties will be ignored. If an option is enabled then its constraints should be satisfied, or the component framework will report various conflicts. Note that the `legal_values` constraint only applies to

the data part of the option's value, so it is only useful with the `data` and `booldata` flavors. Options with the `none` and `data` flavors are always enabled so their constraints always have to be satisfied (assuming the option is active).

5. If an option is disabled then it has no direct consequences at build-time: no `#define` will be generated, no files will get compiled, and so on. If an option is active and enabled then all the consequences take effect. The option name and data part are used to generate the `#define` in the appropriate configuration header file, subject to various properties such as `no_define`, but the data part has no other effects on the build system.

By default all options and components have the `bool` flavor: most options are boolean in nature, so making this the default allows for slightly more compact CDL scripts. Packages have the `booldata` flavor, where the data part always corresponds to the version of the package that is loaded into the configuration: changing this value corresponds to unloading the old version and loading in a different one.

**CDL Flavors:** The concept of CDL flavors tends to result in various discussions about why it is unnecessarily complicated, and would it not have been easier to do . . . However there are very good reasons why CDL works the way it does.

The first common suggestion is that there is no need to have separate flavors `bool`, `data`, and so on. A boolean option could just be handled as a data option with legal values 0 and 1. The counter arguments are as follows:

1. It would actually make CDL scripts more verbose. By default all options and components have the `bool` flavor, since most options are boolean in nature. Without a `bool` flavor it would be necessary to indicate explicitly what the legal values are somehow, e.g. with a `legal_values` property.
2. The boolean part of an option's value has a very different effect from the data part. If an option is disabled then it has no consequences at build time, and can impose no constraints. A `data` option always has consequences and can impose constraints. To get the desired effect it would be necessary to add CDL data indicating that a value of 0 should be treated specially. Arguably this could be made built-in default behavior, although that would complicate options where 0 is a perfectly legal number, for example `CYGNUM_LIBC_TIME_STD_DEFAULT_OFFSET`.
3. There would be no replacement for a `booldata` option for which 0 is a valid value. Again some additional CDL syntax would be needed to express such a concept.

Although initially it may seem confusing that an option's value has both a boolean and a data part, it is an accurate reflection of how configuration options actually work. The various alternatives would all make it harder to write CDL scripts.

The next common suggestion is that the data part of a value should be typed in much the same way as C or C++ data types. For example it should be possible to describe `CYGNUM_LIBC_STDIO_BUFSIZE` as an integer value, rather than imposing `legal_values` constraints. Again there are very good reasons why this approach was not taken:

1. The possible legal values for an integer are rarely correct for a CDL option. A constraint such as `1 to 0x7fffffff` is a bit more accurate, although if this option indicates a buffer size it is still not particularly good — very few targets will have enough memory for such a buffer. Forcing CDL writers to list the `legal_values` constraints explicitly should make them think a bit more about what values are actually sensible. For example `CYGNUM_LIBC_TIME_DST_DEFAULT_OFFSET` has legal values in the range `-90000 to 90000`, which helps the user to set a sensible value.
2. Not all options correspond to simple data types such as integers. `CYGDAT_LIBC_STDIO_DEFAULT_CONSOLE` is a C string, and would have to be expressed using something like `char []`. This introduces plenty of opportunities for confusion, especially since square brackets may get processed by the Tcl interpreter for command substitution.

3. Some configuration options can get very complicated indeed, for example the default value of `CYGDAT_UITRON_TASK_INITIALIZERS` is:

```
CYG_UIT_TASK( "t1", 1, task1, &stack1, CYGNUM_UITRON_STACK_SIZE ), \  
CYG_UIT_TASK( "t2", 2, task2, &stack2, CYGNUM_UITRON_STACK_SIZE ), \  
CYG_UIT_TASK( "t3", 3, task3, &stack3, CYGNUM_UITRON_STACK_SIZE ), \  
CYG_UIT_TASK( "t4", 4, task4, &stack4, CYGNUM_UITRON_STACK_SIZE )
```

This would require CDL knowing about C macros, structures, arrays, static initializers, and so on. Adding such detailed knowledge about the C language to the component framework is inappropriate.

4. CDL needs to be usable with languages other than C. At present this includes C++, in future it may include languages such as Java. Each language adds new data types and related complications, for example C++ classes and inheritance. Making CDL support a union of all data types in all possible languages is not sensible.

The CDL approach of treating all data as a sequence of characters, possibly constrained by a `legal_values` property or other means, has the great advantage of simplicity. It also fits in with the Tcl language that underlies CDL.

## Some Examples

The following excerpt from the C library's CDL scripts can be used to illustrate how values and flavors work in practice:

```
cdl_component CYGPKG_LIBC_RAND {  
    flavor          none  
    compile         stdlib/rand.cxx  
  
    cdl_option CYGSEM_LIBC_PER_THREAD_RAND {  
        requires      CYGVAR_KERNEL_THREADS_DATA  
        default_value 0  
    }  
  
    cdl_option CYGNUM_LIBC_RAND_SEED {  
        flavor        data  
        legal_values  0 to 0x7fffffff  
        default_value 1  
    }  
  
    cdl_option CYGNUM_LIBC_RAND_TRACE_LEVEL {  
        flavor        data  
        legal_values  0 to 1  
        default_value 0  
    }  
}
```

If the application does not require any C library functionality then it is possible to have a configuration where the C library is not loaded. This can be achieved by starting with the minimal template, or by starting with another template such as the default one and then explicitly unloading the C library package. If this package is not loaded then any references to the `CYGPKG_LIBC_RAND` component or any of its options will have a value of 0 for the

purposes of expression evaluation. No `#define`'s will be generated for the component or any of its options, and the file `stdlib/rand.cxx` will not get compiled. There is nothing special about the C library here, exactly the same would apply for say a device driver that does not correspond to any of the devices on the target hardware.

Assuming the C library is loaded, the next thing to consider is whether or not the component and its options are active. The component is layered immediately below the C library package itself, so if the package is loaded then it is safe to assume that the package is also enabled. Therefore the parent of `CYGPKG_LIBC_RAND` is active and enabled, and in the absence of any `active_if` properties `CYGPKG_LIBC_RAND` will be active as well.

The component `CYGPKG_LIBC_RAND` has the flavor `none`. This means the component cannot be disabled. Therefore all the options in this component have an active and enabled parent, and in the absence of any `active_if` properties they are all active as well.

The component's flavor `none` serves to group together all of the configuration options related to random number generation. This is particularly useful in the context of the graphical configuration tool, but it also helps when it comes to naming the options: all of the options begin with `CYGxxx_LIBC_RAND`, giving a clear hint about both the package and the component within that package. The flavor means that the component is always enabled and has the value 1 for the purposes of expression evaluation. There will always be a single `#define` of the form:

```
#define CYGPKG_LIBC_RAND 1
```

In addition the file `stdlib/rand.cxx` will always get built. If the component had the default `bool` flavor then users would be able to disable the whole component, and one less file would need to be built. However random number generation is relatively simple, so the impact on eCos build times are small. Furthermore by default the code has no dependencies on other parts of the system, so compiling the code has no unexpected side effects. Even if it was possible to disable the component, the sensible default for most applications would still leave it enabled. The net result is that the flavor `none` is probably the most sensible one for this component. For other components the default `bool` flavor or one of the other flavors might be more appropriate.

Next consider option `CYGSEM_LIBC_PER_THREAD_RAND` which can be used to get a per-thread random number seed, possibly useful if the application needs a consistent sequence of random numbers. In the absence of a flavor property this option will be boolean, and the `default_value` property means that it is disabled by default — reasonable since few applications need this particular functionality, and it does impose a constraint on the rest of the system. If the option is left disabled then no `#define` will be generated, and if there were any compile or similar properties these would not take effect. If the option is enabled then a `#define` will be generated, using the option's data part which is fixed at 1:

```
#define CYGSEM_LIBC_PER_THREAD_RAND 1
```

The `CYGSEM_LIBC_PER_THREAD_RAND` option has a `requires` constraint on `CYGVAR_KERNEL_THREADS_DATA`. If the C library option is enabled then the constraint should be satisfied, or else the configuration contains a conflict. If the configuration does not include the kernel package then `CYGVAR_KERNEL_THREADS_DATA` will evaluate to 0 and the constraint is not satisfied. Similarly if the option is inactive or disabled the constraint will not be satisfied.

`CYGNUM_LIBC_RAND_SEED` and `CYGNUM_LIBC_RAND_TRACE_LEVEL` both have the `data` flavor, so they are always enabled and the component framework will generate appropriate `#define`'s:

```
#define CYGNUM_LIBC_RAND_SEED 1
#define CYGNUM_LIBC_RAND_SEED_1
#define CYGNUM_LIBC_RAND_TRACE_LEVEL 0
#define CYGNUM_LIBC_RAND_TRACE_LEVEL_0
```

Neither option has a compile or similar property, but any such properties would take effect. Any references to these options in CDL expressions would evaluate to the data part, so a hypothetical constraint of the form { requires CYGNUM\_LIBC\_RAND\_SEED > 42 } would not be satisfied with the default values. Both options use a simple constant for the default\_value expression. It would be possible to use a more complicated expression, for example the default for CYGNUM\_LIBC\_RAND\_TRACE\_LEVEL could be determined from some global debugging option or from a debugging option that applies to the C library as a whole. Both options also have a legal\_values constraint, which must be satisfied since the options are active and enabled.

**Note:** The value 0 is legal for both CYGNUM\_LIBC\_RAND\_SEED and CYGNUM\_LIBC\_RAND\_TRACE\_LEVEL, so in a CDL expression there is no easy way of distinguishing between the options being absent or having that particular value. This will be addressed by future enhancements to the expression syntax.

## Ordinary Expressions

Expressions in CDL follow a conventional syntax, for example:

```
default_value CYGGLO_CODESIZE > CYGGLO_SPEED
default_value { (CYG_HAL_STARTUP == "RAM" &&
                !CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS &&
                !CYGINT_HAL_USE_ROM_MONITOR_UNSUPPORTED &&
                !CYGSEM_HAL_POWERPC_COPY_VECTORS) ? 1 : 0 }
default_value { "\"/dev/ser0\" " }
```

However there is a complication in that the various arguments to a default\_value property will first get processed by a Tcl interpreter, so special characters like quotes and square brackets may get processed. Such problems can be avoided by enclosing non-trivial expressions in braces, as in the second example above. The way expression parsing actually works is as follows:

1. The Tcl interpreter splits the line or lines into a command and its arguments. In the first default\_value expression above the command is default\_value and there are three arguments, CYGGLO\_CODESIZE, > and CYGGLO\_SPEED. In the second and third examples there is just one argument, courtesy of the braces.
2. Next option processing takes place, so any initial arguments that begin with a hyphen will be interpreted as options. This can cause problems if the expression involves a negative number, so the special argument -- can be used to prevent option processing on the subsequent arguments.
3. All of the arguments are now concatenated, with a single space in between each one. Hence the following two expressions are equivalent, even though they will have been processed differently up to this point.

```
default_value CYGGLO_CODESIZE > CYGGLO_SPEED
default_value {CYGGLO_CODESIZE > CYGGLO_SPEED}
```

4. The expression parsing code now has a single string to process.

CDL expressions consist of four types of element: references to configuration options, constant strings, integers, and floating point numbers. These are combined using a conventional set of operators: the unary operators -, ~ and !; the arithmetic operators +, -, \*, / and %; the shift operators << and >>; the comparison operators ==, !=, <, >, <=, >=, and <=.



`<=`, `>` and `>=`; the bitwise operators `&`, `^` and `|`; the logical operators `&&` and `||`; the string concatenation operator `.`; and the ternary conditional operator `A ? B : C`. There is also support for some less widely available operators for logical equivalence and implication, and for a set of function-style operations. Bracketed sub-expressions are supported, and the operators have the usual precedence:

Priority	Operators	Category
16	references, constants	basic elements
15	<code>f(a, b, c)</code>	function calls
14	<code>~</code>	bitwise not
14	<code>!</code>	logical not
14	<code>-</code>	arithmetic negation
13	<code>*</code> <code>/</code> <code>%</code>	multiplicative arithmetic
12	<code>+</code> <code>-</code> <code>.</code>	additive arithmetic and string concatenation
11	<code>&lt;&lt;</code> <code>&gt;&gt;</code>	bitwise shifts
10	<code>&lt;=</code> <code>&lt;</code> <code>&gt;</code> <code>&gt;=</code>	inequality
9	<code>==</code> <code>!=</code>	comparison
8	<code>&amp;</code>	bitwise and
7	<code>^</code>	bitwise xor
6	<code> </code>	bitwise or
5	<code>&amp;&amp;</code>	logical and
4	<code>  </code>	logical or
3	<code>xor</code> , <code>eqv</code>	logical equivalence
2	<code>implies</code>	logical implication
1	<code>?</code> <code>:</code>	conditional

Function calls have the usual format of a name, an opening bracket, one or more arguments separated by commas, and a closing bracket. For example:

```
requires { !is_substr(CYGBLD_GLOBAL_CFLAGS, " -fno-rtti") }
```

Functions will differ in the number of arguments and may impose restrictions on some or all of their arguments. For example it may be necessary for the first argument to be a reference to a configuration option. The available functions are described in [the Section called \*Functions\*](#).

The logical `xor` operator evaluates to true if either the left hand side or the right hand side but not both evaluate to true. The logical `eqv` operator evaluates to true if both the left and right hand sides evaluate to true, or if both evaluate to false. The `implies` operator evaluates to true either if the left hand side is false or if the right hand side is true, in other words `A implies B` has the same meaning as `!A || B`. An example use would be:

```
requires { is_active(CYGNUM_LIBC_MAIN_DEFAULT_STACK_SIZE) implies
  (CYGNUM_LIBC_MAIN_DEFAULT_STACK_SIZE >= (16 * 1024)) }
```

This constraint would be satisfied if either the support for a main stack size is disabled, or if that stack is at least 16K. However if such a stack were in use but was too small, a conflict would be raised.

A valid CDL identifier in an expression, for example `CYGGLO_SPEED`, will be interpreted as a reference to a configuration option by that name. The option does not have to be loaded into the current configuration. When the component framework evaluates the expression it will substitute in a suitable value that depends on whether or not the option is loaded, active, and enabled. The exact rules are described in [the Section called \*Option Values\*](#).

A constant string is any sequence of characters enclosed in quotes. Care has to be taken that these quotes are not stripped off by the Tcl interpreter before the CDL expression parser sees them. Consider the following:

```
default_value "RAM"
```

The quote marks will be stripped before the CDL expression parser sees the data, so the expression will be interpreted as a reference to a configuration option `RAM`. There is unlikely to be such an option, so the actual default value will be 0. Careful use of braces or other Tcl quoting mechanisms can be used to avoid such problems.

String constants consist of the data inside the quotes. If the data itself needs to contain quote characters then appropriate quoting is again necessary, for example:

```
default_value { "\"/dev/ser0\" " }
```

An integer constant consists of a sequence of digits, optionally preceeded with the unary `+` or `-` operators. As usual the sequence `0x` or `0X` can be used for hexadecimal data, and a leading `0` indicates octal data. Internally the component framework uses 64-bit arithmetic for integer data. If a constant is too large then double precision arithmetic will be used instead. Traditional syntax is also used for double precision numbers, for example `3.141592` or `-3E6`.

Of course this is not completely accurate: CDL is not a typed language, all data is treated as if it were a string. For example the following two lines are equivalent:

```
requires CYGNUM_UITRON_SEMAS > 10
requires { CYGNUM_UITRON_SEMAS > "10" }
```

When an expression gets evaluated the operators will attempt appropriate conversions. The `>` comparison operator can be used on either integer or double precision numbers, so it will begin by attempting a string to integer conversion of both operands. If that fails it will attempt string to double conversions. If that fails as well then the component framework will report a conflict, an evaluation exception. If the conversions from string to integer are successful then the result will be either the string `0` or the string `1`, both of which can be converted to integers or doubles as required.

It is worth noting that the expression `CYGNUM_UITRON_SEMAS >10` is not ambiguous. CDL identifiers can never begin with a digit, so it is not possible for `10` to be misinterpreted as a reference to an identifier instead of as a string.

Of course the implementation is slightly different again. The CDL language definition is such that all data is treated as if it were a string, with conversions to integer, double or boolean as and when required. The implementation is allowed to avoid conversions until they are necessary. For example, given `CYGNUM_UITRON_SEMAS > 10` the expression parsing code will perform an immediate conversion from string to integer, storing the integer representation, and there is no need for a conversion by the comparison operator when the expression gets evaluated. Given `{ CYGNUM_UITRON_SEMAS > "10" }` the parsing code will store the string representation and a conversion happens the first time the expression is evaluated. All of this is an implementation detail, and does not affect the semantics of the language.

Different operators have different requirements, for example the bitwise `or` operator only makes sense if both operands have an integer representation. For operators which can work with either integer or double precision numbers, integer arithmetic will be preferred.

The following operators only accept integer operands: unary `~` (bitwise not), the shift operators `<<` and `>>`, and the bitwise operators `&`, `|` and `^`.

The following operators will attempt integer arithmetic first, then double precision arithmetic: unary `-`, the arithmetic operators `+`, `-`, `*`, `/`, and `%`; and the comparison operators `<`, `<=`, `>` and `>=`.

The equality `==` and inequality `!=` operators will first attempt integer conversion and comparison. If that fails then double precision will be attempted (although arguably using these operators on double precision data is not sensible). As a last resort string comparison will be used.

The operators `!`, `&&` and `||` all work with boolean data. Any string that can be converted to the integer 0 or the double 0.0 is treated as false, as is the empty string or the constant string `false`. Anything else is interpreted as true. The result is either 0 or 1.

The conditional operator `? :` will interpret its first operand as a boolean. It does not perform any processing on the second or third operands.

In practice it is rarely necessary to worry about any of these details. In nearly every case CDL expressions just work as expected, and there is no need to understand the full details.

**Note:** The current expression syntax does not meet all the needs of component writers. Some future enhancements will definitely be made, others are more controversial. The list includes the following:

1. An option's value is determined by several different factors: whether or not it is loaded, whether or not it is active, whether or not it is enabled, and the data part. Currently there is no way of querying these individually. This is very significant in the context of options with the `bool` or `booldata` flavors, because there is no way of distinguishing between the option being absent/inactive/disabled or it being enabled with a data field of 0. There should be unary operators that allow any of the factors to be checked.
2. Only the `==` and `!=` operators can be used for string data. More string-related facilities are needed.
3. An implies operator would be useful for many goal expression, where `A implies B` is equivalent to `!A || B`.
4. Similarly there is inadequate support for lists. On occasion it would be useful to write expressions involving say the list of implementors of a given CDL interface, for example a sensible default value could be the first implementor. Associated with this is a need for an indirection operator.
5. Arguably extending the basic CDL expression syntax with lots of new operators is unnecessary, instead expressions should just support Tcl command substitution and then component writers could escape into Tcl scripts for complicated operations. This has some major disadvantages. First, the inference engine would no longer have any sensible way of interpreting an expression to resolve a conflict. Second, the component framework's value propagation code keeps track of which options get referenced in which expressions and avoids unnecessary re-evaluation of expressions; if expressions can involve arbitrary Tcl code then there is no simple way to eliminate unnecessary recalculations, with a potentially major impact on performance.

**Note:** The current implementation of the component framework uses 64 bit arithmetic on all host platforms. Although this is adequate for current target architectures, it may cause problems in future. At some stage it is likely that an arbitrary precision integer arithmetic package will be used instead.

## Functions

CDL expressions can contain calls to a set of built-in functions using the usual syntax, for example;

```
requires { !is_substr(CYGBLD_GLOBAL_CFLAGS, "-fno-rtti") }
```

The available function calls are as follows:

`get_data(option)`

This function can be used to obtain just the data part of a loaded configuration option, ignoring other factors such as whether or not the option is active and enabled. It takes a single argument which should be the name of a configuration option. If the specified option is not loaded in the current configuration then the function returns 0, otherwise it returns the data part. Typically this function will only be used in conjunction with `is_active` and `is_enabled` for fine-grained control over the various factors that make up an option's value.

`is_active(option)`

This function can be used to determine whether or not a particular configuration option is active. It takes a single argument which should be the name of an option, and returns a boolean. If the specified option is not loaded then the function will return false. Otherwise it will consider the state of the option's parents and evaluate any `active_if` properties, and return the option's current active state. A typical use might be:

```
requires { is_active(CYGNUM_LIBC_MAIN_DEFAULT_STACK_SIZE) implies  
          (CYGNUM_LIBC_MAIN_DEFAULT_STACK_SIZE >= (16 * 1024)) }
```

In other words either the specified configuration option must be inactive, for example because the current application does not use any related C library or POSIX functionality, or the stack size must be at least 16K.

The configuration system's inference engine can attempt to satisfy constraints involving `is_active` in various different ways, for example by enabling or disabling parent components, or by examining `active_if` properties and manipulating terms in the associated expressions.

`is_enabled(option)`

This function can be used to determine whether or not a particular configuration option is enabled. It takes a single argument which should be the name of an option, and returns a boolean. If the specified option is not loaded then the function will return false. Otherwise it will return the current boolean part of the option's value. The option's active or inactive state is ignored. Typically this function will be used in conjunction with `is_active` and possibly `get_data` to provide fine-grained control over the various factors that make up an option's value.

`is_loaded(option)`

This function can be used to determine whether or not a particular configuration option is loaded. It takes a single argument which should be the name of an option, and returns a boolean. If the argument is a package then the `is_loaded` function provides little or no extra information, for example the following two constraints are usually equivalent:

```
requires { CYGPKG_KERNEL }  
requires { is_loaded(CYGPKG_KERNEL) }
```

However if the specified package is loaded but re-parented below a disabled component, or inactive as a result of an `active_if` property, then the first constraint would not be satisfied but the second constraint would. In

other words the `is_loaded` makes it possible to consider in isolation one of the factors that are considered when CDL expressions are evaluated.

The configuration system's inference engine will not automatically load or unload packages to satisfy `is_loaded` constraints.

```
is_substr(haystack, needle)
```

This can be used to check whether or not a particular string is present in another string. It is used mainly for manipulating compiler flags. The function takes two arguments, both of which can be arbitrary expressions, and returns a boolean.

`is_substr` has some understanding of word boundaries. If the second argument starts with a space character then that will match either a real space or the start of the string. Similarly if the second argument ends with a space character then that will match a real space or the end of the string. For example, all of the following conditions are satisfied:

```
is_substr("abracadabra", "abra")
is_substr("abracadabra", " abra")
is_substr("hocus pocus", " pocus")
is_substr("abracadabra", "abra ")
```

The first is an exact match. The second is a match because the leading space matches the start of the string. The third is an exact match, with the leading space matching an actual space. The fourth is a match because the trailing space matches the end of the string. However, the following condition is not satisfied.

```
is_substr("abracadabra", " abra ")
```

This fails to match at the start of the string because the trailing space is not matched by either a real space or the end of the string. Similarly it fails to match at the end of the string.

If a constraint involving `is_substr` is not satisfied and the first argument is a reference to a configuration option, the inference engine will attempt to modify that option's value. This can be achieved either by appending the second argument to the current value, or by removing all occurrences of that argument from the current value.

```
requires { !is_substr(CYGBLD_GLOBAL_CFLAGS, " -fno-rtti ") }
requires { is_substr(CYGBLD_GLOBAL_CFLAGS, " -frtti ") }
```

When data is removed the leading and trailing spaces will be left. For example, given an initial value of `<CYGBLD_GLOBAL_CFLAGS` of `-g -fno-rtti -O2` the result will be `-g -O2` rather than `-g-O2`.

If exact matches are needed, the function `is_xsubstr` can be used instead.

```
is_xsubstr(haystack, needle)
```

This function checks whether or not the pattern string is an exact substring of the string being searched. It is similar to `is_substr` but uses exact matching only. In other words, leading or trailing spaces have to match exactly and will not match the beginning or end of the string being searched. The function takes two arguments, both of which can be arbitrary expressions, and returns a boolean. The difference between `is_substr` and `is_xsubstr` is illustrated by the following examples:

```
cdl_option MAGIC {
    flavor data
    default_value { "abracadabra" }
```

```

    }
    ...
    requires { is_substr(MAGIC, " abra") }
    requires { is_xsubstr(MAGIC, " abra") }

```

The first goal will be satisfied because the leading space in the pattern matches the beginning of the string. The second goal will not be satisfied initially because there is no exact match, so the inference engine is likely to update the value of `MAGIC` to `abracadabra abra` which does give an exact match.

```
version_cmp(A, B)
```

This function is used primarily to check that a sufficiently recent [version](#) of some other package is being used. It takes two arguments, both of which can be arbitrary expressions. In practice usually one of the arguments will be a reference to a package and the other will be a constant version string. The return value is -1 if the first argument is a more recent version than the second, 0 if the two arguments correspond to identical versions, and 1 if the first argument is an older version. For example the following constraint can be used to indicate that the current package depends on kernel functionality that only became available in version 1.3:

```
requires { version_cmp(CYGPKG_KERNEL, "v1.3") <= 0 }
```

**Note:** At this time it is not possible to define new functions inside a CDL script. Instead functions can only be added at the C++ level, usually by extending `libcdl` itself. This is partly because there is more to CDL functions than simple evaluation: associated with most functions is support for the inference engine, so that if a constraint involving a function is not currently satisfied the system may be able to find a solution automatically.

## Goal Expressions

The arguments to certain properties, notably `requires` and `active_if`, constitute a goal expression. As with an ordinary expression, all of the arguments get combined and then the expression parser takes over. The same care has to be taken with constant strings and anything else that may get processed by the Tcl interpreter, so often a goal expression is enclosed entirely in braces and the expression parsing code sees just a single argument.

A goal expression is basically just a sequence of ordinary expressions, for example:

```
requires { CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS
           !CYGDBG_HAL_DEBUG_GDB_BREAK_SUPPORT
           !CYGDBG_HAL_DEBUG_GDB_CTRL_C_SUPPORT }

```

This consists of three separate expressions, all of which should evaluate to a non-zero result. The same expression could be written as:

```
requires { CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS  &&
           !CYGDBG_HAL_DEBUG_GDB_BREAK_SUPPORT &&
           !CYGDBG_HAL_DEBUG_GDB_CTRL_C_SUPPORT }

```

Alternatively the following would have much the same effect:

```
requires CYGDBG_HAL_DEBUG_GDB_INCLUDE_STUBS
```

```
requires !CYGDBG_HAL_DEBUG_GDB_BREAK_SUPPORT
requires !CYGDBG_HAL_DEBUG_GDB_CTRL_C_SUPPORT
```

Selecting between these alternatives is largely a stylistic choice. The first is slightly more concise than the others. The second is more likely to appeal to mathematical purists. The third is more amenable to cutting and pasting.

The result of evaluating a goal expression is a boolean. If any part of the goal expression evaluates to the integer 0 or an equivalent string then the result is false, otherwise it is true.

The term “goal expression” relates to the component framework’s inference engine: it is a description of a goal that should be satisfied for a conflict-free configuration. If a requires constraint is not satisfied then the inference engine will examine the goal expression: if there is some way of changing the configuration that does not introduce new conflicts and that will cause the goal expression to evaluate to true, the conflict can be resolved.

The inference engine works with one conflict and hence one goal expression at a time. This means that there can be slightly different behavior if a constraint is specified using a single requires property or several different ones. Given the above example, suppose that none of the three conditions are satisfied. If a single goal expression is used then the inference engine might be able to satisfy only two of the three parts, but since the conflict as a whole cannot be resolved no part of the solution will be applied. Instead the user will have to resolve the entire conflict. If three separate goal expressions are used then the inference engine might well find solutions to two of them, leaving less work for the user. On the other hand, if a single goal expression is used then the inference engine has a bit more information to work with, and it might well find a solution to the entire conflict where it would be unable to find separate solutions for the three parts. Things can get very complicated, and in general component writers should not worry about the subtleties of the inference engine and how to manipulate its behavior.

It is possible to write ambiguous goal expressions, for example:

```
requires CYGNUM_LIBC_RAND_SEED -CYGNUM_LIBC_RAND_TRACE_LEVEL > 5
```

This could be parsed in two ways:

```
requires ((CYGNUM_LIBC_RAND_SEED - CYGNUM_LIBC_RAND_TRACE_LEVEL) > 5)
requires CYGNUM_LIBC_RAND_SEED && ((-CYGNUM_LIBC_RAND_TRACE_LEVEL) > 5)
```

The goal expression parsing code will always use the largest ordinary expression for each goal, so the first interpretation will be used. In such cases it is a good idea to use brackets and avoid possible confusion.

## List Expressions

The arguments to the legal\_values property constitute a goal expression. As with an ordinary and goal expressions, all of the arguments get combined and then the expression parser takes over. The same care has to be taken with constant strings and anything else that may get processed by the Tcl interpreter, so often a list expression is enclosed entirely in braces and the expression parsing code sees just a single argument.

Most list expressions take one of two forms:

```
legal_values <expr1> <expr2> <expr3> ...
legal_values <expr1> to <expr2>
```

expr1, expr2 and so on are ordinary expressions. Often these will be constants or references to calculated options in the architectural HAL package, but it is possible to use arbitrary expressions when necessary. The first syntax indicates a list of possible values, which need not be numerical. The second syntax indicates a numerical range:

both sides of the `to` must evaluate to a numerical value; if either side involves a floating point number then any floating point number in that range is legal; otherwise only integer values are legal; ranges are inclusive, so 4 is a valid value given a list expression `1 to` ; if one or both sides of the `to` does not evaluate to a numerical value then this will result in a run-time conflict. The following examples illustrate these possibilities:

```
legal_values { "red" "green" "blue" }
legal_values 1 2 4 8 16
legal_values 1 to CYGARC_MAXINT
legal_values 1.0 to 2.0
```

It is possible to combine the two syntaxes, for example:

```
legal_values 1 2 4 to CYGARC_MAXINT -1024 -20.0 to -10
```

This indicates three legal values 1, 2 and -1024, one integer range 4 to CYGARC\_MAXINT, and one floating point range -20.0 to -10.0. In practice such list expressions are rarely useful.

The identifier `to` is not reserved, so it is possible to have a configuration option with that name (although it violates every naming convention). Using that option in a list expression may however give unexpected results.

The graphical configuration tool uses the `legal_values` list expression to determine how best to let users manipulate the option's value. Different widgets will be appropriate for different lists, so { "red" "green" "blue" } might involve a pull-down option menu, and `1 to 16` could involve a spinner. The exact way in which `legal_values` lists get mapped on to GUI widgets is not defined and is subject to change at any time.

As with goal expressions, list expressions can be ambiguous. Consider the following hypothetical example:

```
legal_values CYGNUM_LIBC_RAND_SEED -CYGNUM_LIBC_RAND_TRACE_LEVEL
```

This could be parsed in two ways:

```
legal_values (CYGNUM_LIBC_RAND_SEED - CYGNUM_LIBC_RAND_TRACE_LEVEL)
legal_values (CYGNUM_LIBC_RAND_SEED) (-CYGNUM_LIBC_RAND_TRACE_LEVEL)
```

Both are legal. The list expression parsing code will always use the largest ordinary expression for each element, so the first interpretation will be used. In cases like this it is a good idea to use brackets and avoid possible confusion.

## Interfaces

For many configurability requirements, options provide sufficient expressive power. However there are times when a higher level of abstraction is appropriate. As an example, suppose that some package relies on the presence of code that implements the standard kernel scheduling interface. However the requirement is no more stringent than this, so the constraint can be satisfied by the `mlqueue` scheduler, the `bitmap` scheduler, or any additional schedulers that may get implemented in future. A first attempt at expressing the dependency might be:

```
requires CYGSEM_KERNEL_SCHED_MLQUEUE || CYGSEM_KERNEL_SCHED_BITMAP
```

This constraint will work with the current release, but it is limited. Suppose there is a new release of the kernel which adds another scheduler such as a deadline scheduler, or suppose that there is a new third party package which



adds such a scheduler. The package containing the limited constraint would now have to be updated and another release made, with possible knock-on effects.

CDL interfaces provide an abstraction mechanism: constraints can be expressed in terms of an abstract concept, for example “scheduler”, rather than specific implementations such as `CYGSEM_KERNEL_SCHED_MLQUEUE` and `CYGSEM_KERNEL_SCHED_BITMAP`. Basically an interface is a calculated configuration option:

```
cdl_interface CYGINT_KERNEL_SCHEDULER {
    display "Number of schedulers in this configuration"
    ...
}
```

The individual schedulers can then implement this interface:

```
cdl_option CYGSEM_KERNEL_SCHED_MLQUEUE {
    display      "Multi-level queue scheduler"
    default_value 1
    implements   CYGINT_KERNEL_SCHEDULER
    ...
}

cdl_option CYGSEM_KERNEL_SCHED_BITMAP {
    display      "Bitmap scheduler"
    default_value 0
    implements   CYGINT_KERNEL_SCHEDULER
    ...
}
```

Future schedulers can also implement this interface. The value of an interface, for the purposes of expression evaluation, is the number of active and enabled options which implement this interface. Packages which rely on the presence of a scheduler can impose constraints such as:

```
requires CYGINT_KERNEL_SCHEDULER
```

If none of the schedulers are enabled, or if the kernel package is not loaded, then `CYGINT_KERNEL_SCHEDULER` will evaluate to 0. If at least one scheduler is active and enabled then the constraint will be satisfied.

Because interfaces have a calculated value determined by the implementors, the `default_value` and calculated properties are not applicable and should not appear in the body of a `cdl_interface` command. Interfaces have the data flavor by default, but the `bool` and `booldata` flavors may be specified instead. A `bool` interface is disabled if there are no active and enabled implementors, otherwise it is enabled. A `booldata` interface is disabled if there are no active and enabled implementors, otherwise it is enabled and has a value corresponding to the number of these implementors. Other properties such as `requires` and `compile` can be used as normal.

Some component writers will not want to use interfaces in this way. The reasoning is that their code will only have been tested with the existing schedulers, so the `requires` constraint needs to be expressed in terms of those schedulers; it is possible that the component will still work with a new scheduler, but there are no guarantees. Other component writers may take a more optimistic view and assume that their code will work with any scheduler until proven otherwise. It is up to individual component writers to decide which approach is most appropriate in any given case.

One common use for interfaces is to describe the hardware functionality provided by a given target. For example the CDL scripts for a TCP/IP package might want to know whether or not the target hardware has an ethernet

interface. Generally it is not necessary for the TCP/IP stack to know exactly which ethernet hardware is present, since there should be a device driver which implements the appropriate functionality. In CDL terms the device drivers should implement an interface `CYGHWR_NET_DRIVERS`, and the CDL scripts for the TCP/IP stack can use this in appropriate expressions.

**Note:** Using the term *interface* for this concept is sometimes confusing, since the term has various other meanings as well. In practice, it is often correct. If there is a configuration option that implements a given CDL interface, then usually this option will enable some code that provides a particular interface at the C or C++ level. For example an ethernet device driver implements the CDL interface `CYGHWR_NET_DRIVERS`, and also implements a set of C functions that can be used by the TCP/IP stack. Similarly `CYGSEM_KERNEL_SCHED_MLQUEUE` implements the CDL interface `CYGINT_KERNEL_SCHEDULER` and also provides the appropriate scheduling functions.

## Updating the ecos.db database

The current implementation of the component framework requires that all packages be present in a single component repository and listed in that repository's `ecos.db` database. This is not generally a problem for application developers who can consider the component repository a read-only resource, except when adding or removing packages via the administration tool. However it means that component writers need to do their development work inside a component repository as well, and update the database with details of their new package or packages. Future enhancements to the component framework may allow new components to be developed outside a repository.

Like most files related to the component framework, the `ecos.db` database is actually a Tcl script. Typical package entries would look like this:

```
package CYGPKG_LIBC {
  alias { "C library" libc clib clibrary }
  directory language/c/libc
  script libc.cdl
  description "
This package enables compatibility with the ISO C standard - ISO/IEC
9899:1990. This allows the user application to use well known standard
C library functions, and in eCos starts a thread to invoke the user
function main()"
}

package CYGPKG_IO_PCI {
  alias { "PCI configuration library" io_pci }
  directory io/pci
  script io_pci.cdl
  hardware
  description "
    This package contains the PCI configuration library."
}
```

The `package` command takes two arguments, a name and a body. The name must be the same as in the `cdl_package` command in the package's top-level CDL script. The body can contain the following five commands: `alias`, `directory`, `script`, `hardware` and `description`.

**alias**

Each package should have one or more aliases. The first alias is typically used when listing the known packages, because a string like `C library` is a bit easier to read and understand than `CYGPKG_LIBC`. The other aliases are not used for output, but are accepted on input. For example the `ecosconfig` command-line tool will accept `add libc` as an option, as well as `add CYGPKG_LIBC`.

**directory**

This is used to specify the location of the package relative to the root of the component repository. It should be noted that in the current component framework this location cannot be changed in subsequent releases of the package: if for some reason it is desirable to install a new release elsewhere in the repository, all the old versions must first be uninstalled; the database cannot hold two separate locations for one package.

**script**

The `script` command specifies the location of the package's top-level CDL script, in other words the one containing the `cdl_package` definition. If the package follows the [directory layout conventions](#) then this script will be in the `cdl` sub-directory, otherwise it will be relative to the package's top-level directory. Again once a release has been made this file should not change in later releases. In practice the top-level script is generally named after the package itself, so changing its name is unlikely to be useful.

**hardware**

Packages which are tied to specific hardware, for example device drivers and HAL packages, should indicate this in both the `cdl_package` command of the CDL script and in the database entry.

**description**

This should give a brief description of the package. Typically the text for the description property in the `cdl_package` command will be re-used.

**Note:** Most of the information in the `ecos.db` file could be obtained by a relatively simple utility. This would be passed a single argument identifying a package's top-level CDL script. The directory path relative to the component repository root could be determined from the filename. The `name`, `description` and `hardware` fields could be obtained from the script's `cdl_package` command. The `display` property would supply the first alias, additional aliases could be obtained by extending the syntax of that property or by other means. Something along these lines may be provided by a future release of the component framework.

Currently the `ecos.db` database also holds information about the various targets. When porting to a new target it will be necessary to add information about the target to the database, as well as the details of the new platform HAL package and any related packages.

## Notes

1. It is possible that some versions of the Tcl interpreter will instead produce a result of 54 when asked to multiply six by nine. Appropriate reference documentation (<http://www.douglasadams.com/creations/hhgg.html>) should be consulted for more information on why 42 is in fact the correct answer.



# Chapter 4. The Build Process

Some CDL properties describe the consequences of manipulating configuration options. There are two main types of consequences. Typically enabling a configuration option results in one or more `#define`'s in a configuration header file, and properties that affect this include `define`, `define_proc` and `no_define`. Enabling a configuration option can also affect the build process, primarily determining which files get built and added to the appropriate library. Properties related to the build process include `compile` and `make`. This chapter describes the whole build process, including details such as compiler flags and custom build steps.

Part of the overall design of the eCos component framework is that it can interact with a number of different build systems. The most obvious of these is GNU make: the component framework can generate one or more makefiles, and the user can then build the various packages simply by invoking `make`. However it should also be possible to build eCos by other means: the component framework can be queried about what is involved in building a given configuration, and this information can then be fed into the desired build system. Component writers should be aware of this possibility. Most packages will not be affected because the `compile` property can be used to provide all the required information, but care has to be taken when writing custom build steps.

## Build Tree Generation

It is necessary to create an eCos configuration before anything can be built. With some tools such as the graphical configuration tool this configuration will be created in memory, and it is not essential to produce an `ecos.ecc` savefile first (although it is still very desirable to generate such a savefile at some point, to allow the configuration to be re-loaded later on). With other tools the savefile is generated first, for example using `ecosconfig new`, and then a build tree is generated using `ecosconfig tree`. The savefile contains all the information needed to recreate a configuration.

An eCos build actually involves three separate trees. The component repository acts as the source tree, and for application developers this should be considered a read-only resource. The build tree is where all intermediate files, especially object files, are created. The install tree is where the main library `libtarget.a`, the exported header files, and similar files end up. Following a successful build it is possible to take just the install tree and use it for developing an application: none of the files in the component repository or the build tree are needed for that. The build tree will be needed again only if the user changes the configuration. However the install tree does not contain copies of all of the documentation for the various packages, instead the documentation is kept only in the component repository.

By default the build tree, the install tree, and the `ecos.ecc` savefile all reside in the same directory tree. This is not a requirement, both the install tree and the savefile can be anywhere in the file system.

It is worth noting that the component framework does not separate the usual `make` and `make install` stages. A build always populates the install tree, and any `make install` step would be redundant.

The install tree will always begin with two directories, `include` for the exported header files and `lib` for the main library `libtarget.a` and other files such as the linker script. In addition there will be a subdirectory `include/pkgconf` containing the configuration header files, which are generated or updated at the same time the build tree is created or updated. More details of header file generation are given below. Additional `include` subdirectories such as `sys` and `cyg/kernel` will be created during the first build, when each package's exported header files are copied to the install tree. The install tree may also end up with additional subdirectories during a build, for example as a result of custom build steps.

The component framework does not define the structure of the build tree, and this may vary between build systems. It can be assumed that each package in the configuration will have its own directory in the build tree, and that this directory will be used for storing the package's object files and as the current directory for any build steps for that package. This avoids problems when custom build steps from different packages generate intermediate files which happen to have the same name.

Some build systems may allow application developers to copy a source file from the component repository to the build tree and edit the copy. This allows users to experiment with small changes, for example to add a couple of lines of debugging to a package, without having to modify the master copy in the component repository which could be shared by several projects or several people. Functionality such as this is transparent to component writers, and it is the responsibility of the build system to make sure that the right thing happens.

**Note:** There are some unresolved issues related to the build tree and install tree. Specifically, when updating an existing build or install tree, what should happen to unexpected files or directories? Suppose the user started with a configuration that included the math library, and the install tree contains header files `include/math.h` and `include/sys/ieee754.h`. The user then removed the math library from the configuration and is updating the build tree. It is now desirable to remove these header files from the install tree, so that if any application code still attempts to use the math library this will fail at compile time rather than at link time. There will also be some object files in the existing `libtarget.a` library which are no longer appropriate, and there may be other files in the install tree as a result of custom build steps. The build tree will still contain a directory for the math library, which no longer serves any purpose.

However, it is also possible that some of the files in the build tree or the install tree were placed there by the user, in which case removing them automatically would be a bad idea.

At present the component framework does not keep track of exactly what should be present in the build and install trees, so it cannot readily determine which files or library members are obsolete and can safely be removed, and which ones are unexpected and need to be reported to the user. This will be addressed in a future release of the system.

## Configuration Header File Generation

Configuration options can affect a build in two main ways. First, enabling a configuration option or other CDL entity can result in various files being built and added to a library, thus providing functionality to the application code. However this mechanism can only operate at a rather coarse grain, at the level of entire source files. Hence the component framework also generates configuration header files containing mainly C preprocessor `#define` directives. Package source code can then `#include` the appropriate header files and use `#if`, `#ifdef` and `#ifndef` directives to adapt accordingly. In this way configuration options can be used to enable or disable entire functions within a source file or just a single line, whichever is appropriate.

The configuration header files end up in the `include/pkgconf` subdirectory of the install tree. There will be one header file for the system as a whole, `pkgconf/system.h`, and there will be additional header files for each package, for example `pkgconf/kernel.h`. The header files are generated when creating or updating the build and install trees, which needs to happen after every change to the configuration.

The component framework processes each package in the configuration one at a time. The exact order in which the packages are processed is not defined, so the order in which `#define`'s will end up in the global `pkgconf/system.h` header may vary. However for any given configuration the order should remain consistent until packages are added to or removed from the system. This avoids unnecessary changes to the global header

file and hence unnecessary rebuilds of the packages and of application code because of header file dependency handling.

Within a given package the various components, options and interfaces will be processed in the order in which they were defined in the corresponding CDL scripts. Typically the data in the configuration headers consists only of a sequence of `#define`'s so the order in which these are generated is irrelevant, but some properties such as `define_proc` can be used to add arbitrary data to a configuration header and hence there may be dependencies on the order. It should be noted that re-parenting an option below some other package has no effect on which header file will contain the corresponding `#define`: the preprocessor directives will always end up in the header file for the package that defines the option, or in the global configuration header.

There are six properties which affect the process of generating header files: `define_header`, `no_define`, `define_format`, `define`, `if_define`, and `define_proc`.

The `define_header` property can only occur in the body of a `cdl_package` command and specifies the name of the header file which should contain the package's configuration data, for example:

```
cdl_package <some_package> {
    ...
    define_header xyzzy.h
}
```

Given such a `define_header` property the component framework will use the file `pkgconf/xyzzy.h` for the package's configuration data. If a package does not have a `define_header` property then a suitable file name is constructed from the package's name. This involves:

1. All characters in the package name up to and including the first underscore are removed. For example `CYGPKG_KERNEL` is converted to `KERNEL`, and `CYGPKG_HAL_ARM` is converted to `HAL_ARM`.
2. Any upper case letters in the resulting string will be converted to lower case, yielding e.g. `kernel` and `hal_arm`.
3. A `.h` suffix is appended, yielding e.g. `kernel.h` and `hal_arm.h`.

Because of the naming restrictions on configuration options, this should result in a valid filename. There is a small possibility of a file name class, for example `CYGPKG_PLUGH` and `CYGPKG_plugh` would both end up trying to use the same header file `pkgconf/plugh.h`, but the use of lower case letters for package names violates the naming conventions. It is not legal to use the `define_header` property to put the configuration data for several packages in a single header file. The resulting behaviour is undefined.

Once the name of the package's header file has been determined and the file has been opened, the various components, options and interfaces in the package will be processed starting with the package itself. The following steps are involved:

1. If the current option or other CDL entity is inactive or disabled, the option is ignored for the purposes of header file generation. `#define`'s are only generated for options that are both active and enabled.
2. The next step is to generate a default `#define` for the current option. If this option has a `no_define` property then the default `#define` is suppressed, and processing continues for `define`, `if_define` and `define_proc` properties.
  - a. The header file appropriate for the default `#define` is determined. For a `cdl_package` this will be `pkgconf/system.h`, for any other option this will be the package's own header file. The intention here is that packages and application code can always determine which packages are in the configuration by

`#include'ing pkgconf/system.h`. The C preprocessor lacks any facilities for including a header file only if it exists, and taking appropriate action otherwise.

- b. For options with the flavors `bool` or `none`, a single `#define` will be generated. This takes the form:

```
#define <option> 1
```

For example:

```
#define CYGFUN_LIBC_TIME_POSIX 1
```

Package source code can check whether or not an option is active and enabled by using the `#ifdef`, `#ifndef` or `#if defined(...)` directives.

- c. For options with the flavors `data` or `booldata`, either one or two `#define`'s will be generated. The first of these may be affected by a `define_format` property. If this property is not defined then the first `#define` will take the form:

```
#define <option> <value>
```

For example:

```
#define CYGNUM_LIBC_ATEXIT_HANDLERS 32
```

Package source code can examine this value using the `#if` directive, or by using the symbol in code such as:

```
for (i = 0; i < CYGNUM_LIBC_ATEXIT_HANDLERS; i++) {
    ...
}
```

It must be noted that the `#define` will be generated only if the corresponding option is both active and enabled. Options with the `data` flavor are always enabled but may not be active. Code like the above should be written only if it is known that the symbol will always be defined, for example if the corresponding source file will only get built if the containing component is active and enabled. Otherwise the use of additional `#ifdef` or similar directives will be necessary.

- d. If there is a `define_format` property then this controls how the option's value will appear in the header file. Given a format string such as `%08x` and a value 42, the component framework will execute the Tcl command `format %08x 42` and the result will be used for the `#define`'s value. It is the responsibility of the component writer to make sure that this Tcl command will be valid given the format string and the legal values for the option.

- e. In addition a second `#define` may or may not be generated. This will take the form:

```
#define <option>_<value>
```

For example:

```
#define CYGNUM_LIBC_ATEXIT_HANDLERS_32
```

The `#define` will be generated only if it would result in a valid C preprocessor symbol. If the value is a string such as `"/dev/ser0"` then the `#define` would be suppressed. This second `#define` is not particularly useful for numerical data, but can be valuable in other circumstances. For example if the legal values for an option `XXX_COLOR` are `red`, `green` and `blue` then code like the following can be used:

```
#ifdef XXX_COLOR_red
    ...
#endif
```



```

#ifdef XXX_COLOR_green
    ...
#endif
#ifdef XXX_COLOR_blue
    ...
#endif

```

The expression syntax provided by the C preprocessor is limited to numerical data and cannot perform string comparisons. By generating two `#define`'s in this way it is possible to work around this limitation of the C preprocessor. However some care has to be taken: if a component writer also defined a configuration option `XXX_COLOR_green` then there will be confusion. Since such a configuration option violates the naming conventions, the problem is unlikely to arise in practice.

3. For some options it may be useful to generate one or more additional `#define`'s or, in conjunction with the `no_define` property, to define a symbol with a name different from the option's name. This can be achieved with the `define` property, which takes the following form:

```
define [-file=<filename>] [-format=<format>] <symbol>
```

For example:

```
define FOPEN_MAX
```

This will result in something like:

```

#define FOPEN_MAX 8
#define FOPEN_MAX_8

```

The specified symbol must be a valid C preprocessor symbol. Normally the `#define` will end up in the same header file as the default one, in other words `pkgconf/system.h` in the case of a `cdl_package`, or the package's own header file for any other option. The `-file` option can be used to change this. At present the only legal value is `system.h`, for example:

```
define -file=system.h <symbol>
```

This will cause the `#define` to end up in the global configuration header rather than in the package's own header. Use of this facility should be avoided since it is very rarely necessary to make options globally visible.

The `define` property takes another option, `-format`, to provide a format string.

```
define -format=%08x <symbol>
```

This should only be used for options with the `data` or `booldata` flavor, and has the same effect as the `define_format` property has on the default `#define`.

`define` properties are processed in the same way the default `#define`. For options with the `bool` or `none` flavors a single `#define` will be generated using the value 1. For options with the `data` or `booldata` flavors either one or two `#define`'s will be generated.

4. After processing all `define` properties, the component framework will look for any `if_define` properties. These take the following form:

```
if_define [-file=<filename>] <symbol1> <symbol2>
```

For example:

```
if_define CYGSRG_KERNEL CYGDBG_USE_ASSERTS
```

The following will be generated in the configuration header file:

```
#ifdef CYGSRG_KERNEL
# define CYGDBG_USE_ASSERTS
#endif
```

Typical kernel source code would begin with the following construct:

```
#define CYGSRG_KERNEL 1
#include <pkgconf/kernel.h>
#include <cyg/infra/cyg_ass.h>
```

The infrastructure header file `cyg/infra/cyg_ass.h` only checks for symbols such as `CYGDBG_USE_ASSERTS`, and has no special knowledge of the kernel or any other package. The `if_define` property will only affect code that defines the symbol `CYGSRG_KERNEL`, so typically only kernel source code. If the option is enabled then assertion support will be enabled for the kernel source code only. If the option is inactive or disabled then kernel assertions will be disabled. Assertions in other packages are not affected. Thus the `if_define` property allows control over assertions, tracing, and similar facilities at the level of individual packages, or at finer levels such as components or even single source files if desired.

**Note:** Current eCos packages do not yet make use of this facility. Instead there is a single global configuration option `CYGDBG_USE_ASSERTS` which is used to enable or disable assertions for all packages. This issue should be addressed in a future release of the system.

As with the `define` property, the `if_define` property takes an option `-file` with a single legal value `system.h`. This allows the output to be redirected to `pkgconf/system.h` if and when necessary.

5. The final property that is relevant to configuration header file generation is `define_proc`. This takes a single argument, a Tcl fragment that can add arbitrary data to the global header `pkgconf/system.h` and to the package's own header. When the `define_proc` script is invoked two variables will be set up to allow access to these headers: `cdl_header` will be a channel to the package's own header file, for example `pkgconf/kernel.h`; `cdl_system_header` will be a channel to `pkgconf/system.h`. A typical `define_proc` script will use the Tcl `puts` command to output data to one of these channels, for example:

```
cdl_option <name> {
    ...
    define_proc {
        puts $::cdl_header "#define XXX 1"
    }
}
```

**Note:** In the current implementation the use of `define_proc` is limited because the Tcl script cannot access any of the configuration data. Therefore the script is limited to writing constant data to the configuration headers. This is a major limitation which will be addressed in a future release of the component framework.

**Note:** Generating C header files with `#define's` for the configuration data suffices for existing packages written in some combination of C, C++ and assembler. It can also be used in conjunction with some other languages,

for example by first passing the source code through the C preprocessor and feeding the result into the appropriate compiler. In future versions of the component framework additional programming languages such as Java may be supported, and the configuration data may also be written to files in some format other than C preprocessor directives.

**Note:** At present there is no way for application or package source code to get hold of all the configuration details related to the current hardware. Instead that information is spread over various different configuration headers for the HAL and device driver packages, with some of the information going into `pkgconf/system.h`. It is possible that in some future release of the system there will be another global configuration header file `pkgconf/hardware.h` which either contains the configuration details for the various hardware-specific packages or which `#include`'s all the hardware-specific configuration headers. The desirability and feasibility of such a scheme are still to be determined. To avoid future incompatibility problems as a result of any such changes, it is recommended that all hardware packages (in other packages containing the hardware property) use the `define_header` property to specify explicitly which configuration header should be generated.

## The `system.h` Header

Typically configuration header files are `#include`'d only by the package's source code at build time, or by a package's exported header files if the interface provided by the package may be affected by a configuration option. There should be no need for application code to know the details of individual configuration options, instead the configuration should specifically meet the needs of the application.

There are always exceptions. Application code may want to adapt to configuration options, for example to do different things for ROM and RAM booting systems, or when it is necessary to support several different target boards. This is especially true if the code in question is really re-usable library code which has not been converted to an eCos package, and hence cannot use any CDL facilities.

A major problem here is determining which packages are in the configuration: attempting to `#include` a header file such as `pkgconf/net.h` when it is not known for certain that that particular package is part of the configuration will result in compilation errors. The global header file `pkgconf/system.h` serves to provide such information, so application code can use techniques like the following:

```
#include <pkgconf/system.h>
#ifdef CYGPKG_NET
# include <pkgconf/net.h>
#endif
```

This will compile correctly irrespective of the eCos configuration, and subsequent code can use `#ifdef` or similar directives on `CYGPKG_NET` or any of the configuration options in that package.

In addition to determining whether or not a package is present, the global configuration header file can also be used to find out the specific version of a package that is being used. This can be useful if a more recent version exports additional functionality. It may also be necessary to adapt to incompatible changes in the exported interface or to changes in behaviour. For each package the configuration system will typically `#define` three symbols, for example for a V1.3.1 release:

```
#define CYGNUM_NET_VERSION_MAJOR 1
#define CYGNUM_NET_VERSION_MINOR 3
```

```
#define CYGNUM_NET_VERSION_RELEASE 1
```

There are a number of problems associated with such version `#define`'s. The first restriction is that the package must follow the standard naming conventions, so the package name must be of the form `xxxPKG_yyy`. The three characters immediately preceding the first underscore must be `PKG`, and will be replaced with `NUM` when generating the version `#define`'s. If a package does not follow the naming convention then no version `#define`'s will be generated.

Assuming the package does follow the naming conventions, the configuration tools will always generate three version `#define`'s for the major, minor, and release numbers. The symbol names are obtained from the package name by replacing `PKG` with `NUM` and appending `_VERSION_MAJOR`, `_VERSION_MINOR` and `_VERSION_RELEASE`. It is assumed that the resulting symbols will not clash with any configuration option names. The values for the `#define`'s are determined by searching the version string for sequences of digits, optionally preceded by a minus sign. It is possible that some or all of the numbers are absent in any given version string, in which case `-1` will be used in the `#define`. For example, given a version string of `V1.12beta`, the major version number is 1, the minor number is 12, and the release number is `-1`. Given a version string of `beta` all three numbers would be set to `-1`.

There is special case code for the version `current`, which typically corresponds to a development version obtained via anonymous CVS or similar means. The configuration system has special built-in knowledge of this version, and will assume it is more recent than any specific release number. The global configuration header defines a special symbol `CYGNUM_VERSION_CURRENT`, and this will be used as the major version number when version `current` of a package is used:

```
#define CYGNUM_VERSION_CURRENT 0x7fffffff00
...
#define CYGNUM_INFRA_VERSION_MAJOR CYGNUM_VERSION_CURRENT
#define CYGNUM_INFRA_VERSION_MINOR -1
#define CYGNUM_INFRA_VERSION_RELEASE -1
```

The large number used for `CYGNUM_VERSION_CURRENT` should ensure that major version comparisons work as expected, while still allowing for a small amount of arithmetic in case that proves useful.

It should be noted that this implementation of version `#define`'s will not cope with all version number schemes. However for many cases it should suffice.

## Building eCos

The primary goal of an eCos build is to produce the library `libtarget.a`. A typical eCos build will also generate a number of other targets: `extras.o`, startup code `vectors.o`, and a linker script. Some packages may cause additional libraries or targets to be generated. The basic build process involves a number of different phases with corresponding priorities. There are a number of predefined priorities:

Priority	Action
0	Export header files
100	Process compile properties and most <code>make_object</code> custom build steps
200	Generate libraries

Priority	Action
300	Process make custom build steps

Generation of the `extras.o` file, the startup code and the linker script actually happens via make custom build steps, typically defined in appropriate HAL packages. The component framework has no special knowledge of these targets.

By default custom build steps for a `make_object` property happen during the same phase as most compilations, but this can be changed using a `-priority` option. Similarly custom build steps for a `make` property happen at the end of a build, but this can also be changed with a `-priority` option. For example a priority of 50 can be used to run a custom build step between the header file export phase and the main compilation phase. Custom build steps are discussed in more detail below.

Some build systems may run several commands of the same priority in parallel. For example files listed in compile properties may get compiled in parallel, concurrently with `make_object` custom build steps with default priorities. Since most of the time for an eCos build involves processing compile properties, this allows builds to be speeded up on suitable host hardware. All build steps for a given phase will complete before the next phase is started.

## Updating the Build Tree

Some build systems may involve a phase before the header files get exported, to update the build and install trees automatically when there has been a change to the configuration savefile `ecos.ecc`. This is useful mainly for application developers using the command line tools: it would allow users to create the build tree only once, and after any subsequent configuration changes the tree would be updated automatically by the build system. The facility would be analogous to the `--enable-maintainer-mode` option provide by the `autoconf` and `automake` programs. At present no eCos build system implements this functionality, but it is likely to be added in a future release.

## Exporting Public Header Files

The first compulsory phase involves making sure that there is an up to date set of header files in the install tree. Each package can contain some number of header files defining the exported interface. Applications should only use exported functionality. A package can also contain some number of private header files which are only of interest to the implementation, and which should not be visible to application code. The various packages that go into a particular configuration can be spread all over the component repository. In theory it might be possible to make all the exported header files accessible by having a lengthy `-I` header file search path, but this would be inconvenient both for building eCos and for building applications. Instead all the relevant header files are copied to a single location, the `include` subdirectory of the install tree. The process involves the following:

1. The install tree, for example `/usr/local/ecos/install`, and its `include` subdirectory `/usr/local/ecos/install/include` will typically be created when the build tree is generated or updated. At the same time configuration header files will be written to the `pkgconf` subdirectory, for example `/usr/local/ecos/include/pkgconf`, so that the configuration data is visible to all the packages and to application code that may wish to examine some of the configuration options.
2. Each package in the configuration is examined for exported header files. The exact order in which the packages are processed is not defined, but should not matter.

- a. If the package has an `include_files` property then this lists all the exported header files:

```
cdl_package <some_package> {
    ...
    include_files header1.h header2.h
}
```

If no arguments are given then the package does not export any header files.

```
cdl_package <some_package> {
    ...
    include_files
}
```

The listed files may be in an `include` subdirectory within the package's hierarchy, or they may be relative to the package's toplevel directory. The `include_files` property is intended mainly for very simple packages. It can also be useful when converting existing code to an eCos package, to avoid rearranging the sources.

- b. If there is no `include_files` property then the component framework will look for an `include` subdirectory in the package, as per the layout conventions. All files, including those in subdirectories, will be treated as exported header files. For example, the math library package contains files `include/math.h` and `include/sys/ieeefp.h`, both of which will be exported to the install tree.
- c. As a last resort, if there is neither an `include_files` property nor an `include` subdirectory, the component framework will search the package's toplevel directory and all of its subdirectories for files with one of the following suffixes: `.h`, `.hxx`, `.inl` or `.inc`. All such files will be interpreted as exported header files.

This last resort rule could cause confusion for packages which have no exported header files but which do contain one or more private header files. For example a typical device driver simply implements an existing interface rather than define a new one, so it does not need to export a header file. However it may still have one or more private header files. Such packages should use an `include_files` property with no arguments.

3. If the package has one or more exported header files, the next step is to determine where the files should end up. By default all exported header files will just end up relative to the install tree's `include` subdirectory. For example the math library's `math.h` header would end up as `/usr/local/ecos/include/math.h`, and the `sys/ieeefp.h` header would end up as `/usr/local/ecos/include/sys/ieeefp.h`. This behaviour is correct for packages like the C library where the interface is defined by appropriate standards. For other packages this behaviour can lead to file name clashes, and the `include_dir` property should be used to avoid this:

```
cdl_package CYGPKG_KERNEL {
    include_dir cyg/kernel
}
```

This means that the kernel's exported header file `include/kapi.h` should be copied to `/usr/local/ecos/include/cyg/kernel/kapi.h`, where it is very unlikely to clash with a header file from some other package.

4. For typical application developers there will be little or no need for the installed header files to change after the first build. Changes will be necessary only if packages are added to or removed from the configuration. For component writers, the build system should detect changes to the master copy of the header file source code and update the installed copies automatically during the next build. The build system is expected to perform a header file dependency analysis, so any source files affected should get rebuilt as well.
5. Some build systems may provide additional support for application developers who want to make minor changes to a package, especially for debugging purposes. A header file could be copied from the component repository (which for application developers is assumed to be a read-only resource) into the build tree and edited there. The build system would detect a more recent version of such a header file in the build tree and install it. Care would have to be taken to recover properly if the modified copy in the build tree is subsequently removed, in order to revert to the original behaviour.
6. When updating the install tree's `include` subdirectory, the build tree may also perform a clean-up operation. Specifically, it may check for any files which do not correspond to known exported header files and delete them.

**Note:** At present there is no defined support in the build system for defining custom build steps that generate exported header files. Any attempt to use the existing custom build step support may fall foul of unexpected header files being deleted automatically by the build system. This limitation will be addressed in a future release of the component framework, and may require changing the priority for exporting header files so that a custom build step can happen first.

## Compiling

Once there are up to date copies of all the exported header files in the build tree, the main build can proceed. Most of this involves compiling source files listed in `compile` properties in the CDL scripts for the various packages, for example:

```
cdl_package CYGPKG_ERROR {
    display      "Common error code support"
    compile      strerror.cxx
    ...
}
```

`compile` properties may appear in the body of a `cdl_package`, `cdl_component`, `cdl_option` or `cdl_interface`. If the option or other CDL entity is active and enabled, the property takes effect. If the option is inactive or disabled the property is ignored. It is possible for a `compile` property to list multiple source files, and it is also possible for a given CDL entity to contain multiple `compile` properties. The following three examples are equivalent:

```
cdl_option <some_option> {
    ...
    compile file1.c file2.c file3.c
}
```

```
cdl_option <some_option> {
    ...
    compile file1.c
    compile file2.c
}
```

```

    compile file3.c
}

cdl_option <some_option> {
    ...
    compile file1.c file2.c
    compile file3.c
}

```

Packages that follow the directory layout conventions should have a subdirectory `src`, and the component framework will first look for the specified files there. Failing that it will look for the specified files relative to the package's root directory. For example if a package contains a source file `strerror.cxx` then the following two lines are equivalent:

```

compile strerror.cxx
compile src/strerror.cxx

```

In the first case the component framework will find the file immediately in the packages `src` subdirectory. In the second case the framework will first look for a file `src/src/strerror.cxx`, and then for `src/strerror.cxx` relative to the package's root directory. The result is the same.

The file names may be relative paths, allowing the source code to be split over multiple directories. For example if a package contains a file `src/sync/mutex.cxx` then the corresponding CDL entry would be:

```

compile sync/mutex.cxx

```

All the source files relevant to the current configuration will be identified when the build tree is generated or updated, and added to the appropriate makefile (or its equivalent for other build systems). The actual build will involve a rule of the form:

```

<object file> : <source file>
    $(CC) -c $(INCLUDE_PATH) $(CFLAGS) -o $@ $<

```

The component framework has built-in knowledge for processing source files written in C, C++ or assembler. These should have a `.c`, `.cxx` and `.s` suffix respectively. The current implementation has no simple mechanism for extending this with support for other languages or for alternative suffixes, but this should be addressed in a future release.

The compiler command that will be used is something like `arm-elf-gcc`. This consists of a command prefix, in this case `arm-elf`, and a specific command such as `gcc`. The command prefix will depend on the target architecture and is controlled by a configuration option in the appropriate HAL package. It will have a sensible default value for the current architecture, but users can modify this option when necessary. The command prefix cannot be changed on a per-package basis, since it is usually essential that all packages are built with a consistent set of tools.

The `$(INCLUDE_PATH)` header file search path consists of at least the following:

1. The `include` directory in the install tree. This allows source files to access the various header files exported by all the packages in the configuration, and also the configuration header files.
2. The current package's root directory. This ensures that all files in the package are accessible at build time.



3. The current package's `src` subdirectory, if it is present. Generally all files to be compiled are located in or below this directory. Typically this is used to access private header files containing implementation details only.

The compiler flags `$(CFLAGS)` are determined in two steps. First the appropriate HAL package will provide a configuration option defining the global flags. Typically this includes flags that are needed for the target processor, for example `-mcpu=arm9`, various flags related to warnings, debugging and optimization, and flags such as `-finit-priority` which are needed by eCos itself. Users can modify the global flags option as required. In addition it is possible for existing flags to be removed from and new flags to be added to the current set on a per-package basis, again by means of user-modifiable configuration options. More details are given below.

Component writers can assume that the build system will perform full header file dependency analysis, including dependencies on configuration headers, but the exact means by which this happens is implementation-defined. Typical application developers are unlikely to modify exported or private header files, but configuration headers are likely to change as the configuration is changed to better meet the needs of the application. Full header file dependency analysis also makes things easier for the component writers themselves.

The current directory used during a compilation is an implementation detail of the build system. However it can be assumed that each package will have its own directory somewhere in the build tree, to prevent file name clashes, that this will be the current directory, and that intermediate object files will end up here.

## Generating the Libraries

Once all the compile and `make_object` properties have been processed and the required object files have been built or rebuilt, these can be collected together in one or more libraries. The archiver will be the `ar` command corresponding to the current architecture, for example `powerpc-eabi-ar`. By default all of the object files will end up in a single library `libtarget.a`. This can be changed on a per-package basis using the `library` property in the body of the corresponding `cdl_package` command, for example:

```
cdl_package <SOME_PACKAGE> {
    ...
    library libSomePackage.a
}
```

However using different libraries for each package should be avoided. It makes things more difficult for application developers since they now have to link the application code with more libraries, and possibly even change this set of libraries when packages are added to or removed from the configuration. The use of a single library `libtarget.a` avoids any complications.

It is also possible to change the target library for individual files, using a `-library` option with the corresponding `compile` or `make_object` property. For example:

```
compile -library=libSomePackage.a hello.c
make_object -library=libSomePackage.a {
    ...
}
```

Again this should be avoided because it makes application development more difficult. There is one special library which can be used freely, `libextras.a`, which is used to generate the `extras.o` file as described below.

The order in which object files end up in a library is not defined. Typically each library will be created directly in the install tree, since there is little point in generating a file in the build tree and then immediately copying it to the install tree.

## The `extras.o` file

Package sources files normally get compiled and then added to a library, by default `libtarget.a`, which is then linked with the application code. Because of the usual rules for linking with libraries, augmented by the use of link-time garbage collection, this means that code will only end up in the final executable if there is a direct or indirect reference to it in the application. Usually this is the desired behaviour: if the application does not make any use of say kernel message boxes, directly or indirectly, then that code should not end up in the final executable taking up valuable memory space.

In a few cases it is desirable for package code to end up in the final executable even if there are no direct or indirect references. For example, device driver functions are often not called directly. Instead the application will access the device via the string `"/dev/xyzzy"` and call the device functions indirectly. This will be impossible if the functions have been removed at link-time.

Another example involves static C++ objects. It is possible to have a static C++ object, preferably with a suitable constructor priority, where all of the interesting work happens as a side effect of running the constructor. For example a package might include a monitoring thread or a garbage collection thread created from inside such a constructor. Without a reference by the application to the static object the latter will never get linked in, and the package will not function as expected.

A third example would be copyright messages. A package vendor may want to insist that all products shipped using that package include a particular message in memory, even though many users of that package will object to such a restriction.

To meet requirements such as these the build system provides support for a file `extras.o`, which always gets linked with the application code via the linker script. Because it is an object file rather than a library everything in the file will be linked in. The `extras.o` file is generated at the end of a build from a library `libextras.a`, so packages can put functions and variables in suitable source files and add them to that library explicitly:

```
compile -library=libextras.a xyzzy.c
compile xyzzy_support.c
```

In this example `xyzzy.o` will end up in `libextras.a`, and hence in `extras.o` and in the final executable. `xyzzy_support.o` will end up in `libtarget.a` as usual, and is subject to linker garbage collection.

## Compilers and Flags

### Caution

Some of the details of compiler selection and compiler flags described below are subject to change in future revisions of the component framework, although every reasonable attempt will be made to avoid breaking backwards compatibility.

The build system needs to know what compiler to use, what compiler flags should be used for different stages of the build and so on. Much of this information will vary from target to target, although users should be able to override this when appropriate. There may also be a need for some packages to modify the compiler flags. All platform HAL packages should define a number of options with well-known names, along the following lines (any existing platform HAL package can be consulted for a complete example):

```
cdl_component CYGBLD_GLOBAL_OPTIONS {
    flavor none
    parent CYGPKG_NONE
    ...

    cdl_option CYGBLD_GLOBAL_COMMAND_PREFIX {
        flavor data
        default_value { "arm-elf" }
        ...
    }
    cdl_option CYGBLD_GLOBAL_CFLAGS {
        flavor data
        default_value "-Wall -g -O2 ..."
        ...
    }

    cdl_option CYGBLD_GLOBAL_LDFLAGS {
        flavor data
        default_value "-g -nostdlib -Wl,--gc-sections ..."
        ...
    }
}
```

The `CYGBLD_GLOBAL_OPTIONS` component serves to collect together all global build-related options. It has the flavor `none` since disabling all of these options would make it impossible to build anything and hence is not useful. It is parented immediately below the root of the configuration hierarchy, thus making sure that it is readily accessible in the graphical configuration tool and, for command line users, in the `ecos.ecc` save file.

**Note:** Currently the parent property lists a parent of `CYGPKG_NONE`, rather than an empty string. This could be unfortunate if there was ever a package with that name. The issue will be addressed in a future release of the component framework.

The option `CYGBLD_GLOBAL_COMMAND_PREFIX` defines which tools should be used for the current target. Typically this is determined by the processor on the target hardware. In some cases a given target board may be able to support several different processors, in which case the `default_value` expression could select a different toolchain depending on some other option that is used to control which particular processor. `CYGBLD_GLOBAL_COMMAND_PREFIX` is modifiable rather than calculated, so users can override this when necessary.

Given a command prefix such as `arm-elf`, all C source files will be compiled with `arm-elf-gcc`, all C++ sources will be built using `arm-elf-g++`, and `arm-elf-ar` will be used to generate the library. This is in accordance with the usual naming conventions for GNU cross-compilers and similar tools. For the purposes of custom build steps, tokens such as `$(CC)` will be set to `arm-elf-gcc`.

The next option, `CYGBLD_GLOBAL_CFLAGS`, is used to provide the initial value of `$(CFLAGS)`. Some compiler flags such as `-Wall` and `-g` are likely to be used on all targets. Other flags such as `-mcpu=arm7tdmi` will be

target-specific. Again this is a modifiable option, so the user can switch from say `-O2` to `-Os` if desired. The option `CYGBLD_GLOBAL_LDFLAGS` serves the same purpose for `$(LDFLAGS)` and linking. It is used primarily when building test cases or possibly for some custom build steps, since building eCos itself generally involves building one or more libraries rather than executables.

Some packages may wish to add certain flags to the global set, or possibly remove some flags. This can be achieved by having appropriately named options in the package, for example:

```
cdl_component CYGPKG_KERNEL_OPTIONS {
    display "Kernel build options"
    flavor none
    ...

    cdl_option CYGPKG_KERNEL_CFLAGS_ADD {
        display "Additional compiler flags"
        flavor data
        default_value { "" }
        ...
    }

    cdl_option CYGPKG_KERNEL_CFLAGS_REMOVE {
        display "Suppressed compiler flags"
        flavor data
        default_value { "" }
        ...
    }

    cdl_option CYGPKG_KERNEL_LDFLAGS_ADD {
        display "Additional linker flags"
        flavor data
        default_value { "" }
        ...
    }

    cdl_option CYGPKG_KERNEL_LDFLAGS_REMOVE {
        display "Suppressed linker flags"
        flavor data
        default_value { "" }
        ...
    }
}
```

In this example the kernel does not modify the global compiler flags by default, but it is possible for the users to modify the options if desired. The value of `$(CFLAGS)` that is used for the compilations and custom build steps in a given package is determined as follows:

1. Start with the global settings from `CYGBLD_GLOBAL_CFLAGS`, for example `-g -O2`.
2. Remove any flags specified in the per-package `CFLAGS_REMOVE` option, if any. For example if `-O2` should be removed for this package then `$(CFLAGS)` would now have a value of just `-g`.
3. Then concatenate the flags specified by the per-package `CFLAGS_ADD` option, if any. For example if `-Os` should be added for the current package then the final value of `$(CFLAGS)` will be `-g -Os`.

`$(LDFLAGS)` is determined in much the same way.

**Note:** The way compiler flags are handled at present has numerous limitations that need to be addressed in a future release, although it should suffice for nearly all cases. For the time being custom build steps and in particular the `make_object` property can be used to work around the limitations.

Amongst the issues, there is a specific problem with package encapsulation. For example the math library imposes some stringent requirements on the compiler in order to guarantee exact IEEE behavior, and may need special flags on a per-architecture basis. One way of handling this is to have `CYGPKG_LIBM_CFLAGS_ADD` and `CYGPKG_LIBM_CFLAGS_REMOVE` default\_value expressions which depend on the target architecture, but such expressions may have to be updated for each new architecture. An alternative approach would allow the architectural HAL package to modify the default\_value expressions for the math library, but this breaks encapsulation. A third approach would allow some architectural HAL packages to define one or more special options with well-known names, and the math library could check if these options were defined and adjust the default values appropriately. Other packages with floating point requirements could do the same. This approach also has scalability issues, in particular how many such categories of options would be needed? It is not yet clear how best to resolve such issues.

**Note:** When generating a build tree it would be desirable for the component framework to output details of the tools and compiler flags in a format that can be re-used for application builds, for example a makefile fragment. This would make it easier for application developers to use the same set of flags as were used for building eCos itself, thus avoiding some potential problems with incompatible compiler flags.

## Custom Build Steps

### Caution

Some of the details of custom build steps as described below are subject to change in future revisions of the component framework, although every reasonable attempt will be made to avoid breaking backwards compatibility.

For most packages simply listing one or more source files in a `compile` property is sufficient. These files will get built using the appropriate compiler and compiler flags and added to a library, which then gets linked with application code. A package that can be built in this way is likely to be more portable to different targets and build environments, since it avoids build-time dependencies. However some packages have special needs, and the component framework supports custom build steps to allow for these needs. There are two properties related to this, `make` and `make_object`, and both take the following form:

```
make {
    <target_filepath> : <dependency_filepath> ...
    <command>
    ...
}
```

Although this may look like makefile syntax, and although some build environments will indeed involve generating makefiles and running `make`, this is not guaranteed. It is possible for the component framework to be integrated

with some other build system, and custom build steps should be written with that possibility in mind. Each custom build step involves a target, some number of dependency files, and some number of commands. If the target is not up to date with respect to one or more of the dependencies then the commands need to be executed.

- a. Only one target can be specified. For a `make_object` property this target must be an object file. For a `make` property it can be any file. In both cases it must refer to a physical file, the use of phony targets is not supported. The target should not be an absolute path name. If the generated file needs to end up in the install tree then this can be achieved using a `<PREFIX>` token, for example:

```
make {
    <PREFIX>/lib/mytarget : ...
    ...
}
```

When the build tree is generated and the custom build step is added to the makefile (or whatever build system is used) `<PREFIX>` will be replaced with the absolute path to the install tree.

- b. All the dependencies must also refer to physical files, not to phony targets. These files may be in the source tree. The `<PACKAGE>` token can be used to indicate this: when the build tree is generated this token will be replaced with the absolute path to the package's root directory in the component repository, for example:

```
make_object {
    xyzzy.o : <PACKAGE>/src/xyzzy.c
    ...
}
```

If the component repository was installed in `/usr/local/ecos` and this custom build step existed in version 1\_5 of the kernel, `<PACKAGE>` would be replaced with `/usr/local/ecos/packages/kernel/v1_5`.

Alternatively the dependencies may refer to files that are generated during the build. These may be object files resulting from compile properties or other `make_object` properties, or they may be other files resulting from a `make` property, for example:

```
compile plugh.c
make_object {
    xyzzy.o : plugh.o
    ...
}
```

- c. No other token or makefile variables may be used in the target or dependency file names. Also conditionals such as `ifneq` and similar makefile functionality must not be used.
- d. Similarly the list of commands must not use any makefile conditionals or similar functionality. A number of tokens can be used to provide access to target-specific or environmental data. Note that these tokens look like makefile variables, unlike the `<PREFIX>` and `<PACKAGE>` tokens mentioned earlier:

Token	Purpose	Example value
<code>\$(AR)</code>	the GNU archiver	<code>mips-tx39-elf-ar</code>
<code>\$(CC)</code>	the GNU compiler	<code>sh-elf-gcc</code>
<code>\$(CFLAGS)</code>	compiler flags	<code>-O2 -Wall</code>
<code>\$(COMMAND_PREFIX)</code>	the triplet prefix	<code>mn10300-elf-</code>
<code>\$(INCLUDE_PATH)</code>	header file search path	<code>-I. -Isrc/misc</code>

Token	Purpose	Example value
<code>\$(LDFLAGS)</code>	linker flags	<code>-nostdlib -Wl,-static</code>
<code>\$(OBJCOPY)</code>	the objcopy utility	<code>arm-elf-objcopy</code>
<code>\$(PREFIX)</code>	location of the install tree	<code>/home/fred/ecos-install</code>
<code>\$(REPOSITORY)</code>	location of the component repository	<code>/home/fred/ecos/packages</code>

In addition commands in a custom build step may refer to the target and the dependencies using `$$`, `$<`, `$^` and `$*`, all of which behave as per GNU make syntax. The commands will execute in a suitable directory in the build tree.

- e. The current directory used during a custom build step is an implementation detail of the build system. However it can be assumed that each package will have its own directory somewhere in the build tree, to prevent file name clashes, and that this will be the current directory. In addition any object files generated as a result of compile properties will be located here as well, which is useful for custom build steps that depend on a `.o` file previously generated.

Any temporary files created by a custom build step should be generated in the build tree (in or under the current directory). Such files should be given a `.tmp` file extension to ensure that they are deleted during a `make clean` or equivalent operation.

If a package contains multiple custom build steps with the same priority, it is possible that these build steps will be run concurrently. Therefore these custom build steps must not accidentally use the same file names for intermediate files.

- f. Care has to be taken to make sure that the commands in a custom build step will run on all host platforms, including Windows NT as well as Linux and other Unix systems. For example, all file paths should use forward slashes as the directory separator. It can be assumed that Windows users will have a full set of CygWin tools installed and available on the path. The GNU coding standards (<http://www.gnu.org/prep/standards.html>) provide some useful guidelines for writing portable build rules.
- g. A custom build step must not make any assumptions concerning the version of another package. This enforces package encapsulation, preventing one package from accessing the internals of another.
- h. No assumptions should be made about the target platform, unless the package is inherently specific to that platform. Even then it is better to use the various tokens whenever possible, rather than hard-coding in details such as the compiler. For example, given a custom build step such as:

```
arm-elf-gcc -c -mcpu=arm7di -o $$ $<
```

Even if this build step will only be invoked on ARM targets, it could cause problems. For example the toolchain may have been installed using a prefix other than `arm-elf`. Also, if the user changes the compiler flags then this would not be reflected in the build step. The correct way to write this rule would be:

```
$(CC) -c $(CFLAGS) -o $$ $<
```

Some commands such as the compiler, the archiver, and objcopy are required sufficiently often to warrant their own tokens, for example `$(CC)` and `$(OBJCOPY)`. Other target-specific commands are needed only rarely and the `$(COMMAND_PREFIX)` token can be used to construct the appropriate command name, for example:

```
$(COMMAND_PREFIX)size $< > $$
```

- i. Custom build steps should not be used to build host-side executables, even if those executables are needed to build parts of the target side code. Support for building host-side executables will be added in a future version of the component framework, although it will not necessarily involve these custom build steps.

By default custom build steps defined in a `make_object` property have a priority of 100, which means that they will be executed in the same phase as compilations resulting from a `compile` property. It is possible to change the priority using a property option, for example:

```
make_object -priority 50 {  
    ...  
}
```

Specifying a priority smaller than a 100 means that the custom build step happens before the normal compilations. Priorities between 100 and 200 happen after normal compilations but before the libraries are archived together. `make_object` properties should not specify a priority of 200 or later.

Custom build steps defined in a `make` property have a default priority of 300, and so they will happen after the libraries have been built. Again this can be changed using a `-priority` property option.

## Startup Code

Linking an application requires the application code, a linker script, the eCos library or libraries, the `extras.o` file, and some startup code. Depending on the target hardware and how the application gets booted, this startup code may do little more than branching to `main()`, or it may have to perform a considerable amount of hardware initialization. The startup code generally lives in a file `vectors.o` which is created by a custom build step in a HAL package. As far as application developers are concerned the existence of this file is largely transparent, since the linker script ensures that the file is part of the final executable.

This startup code is not generally of interest to component writers, only to HAL developers who are referred to one of the existing HAL packages for specific details. Other packages are not expected to modify the startup in any way. If a package needs some work performed early on during system initialization, before the application's main entry point gets invoked, this can be achieved using a static object with a suitable constructor priority.

**Note:** It is possible that the `extras.o` support, in conjunction with appropriate linker script directives, could be used to eliminate the need for a special startup file. The details are not yet clear.

## The Linker Script

### Caution

This section is not finished, and the details are subject to change in a future release. Arguably linker script issues should be documented in the HAL documentation rather than in this guide.



Generating the linker script is the responsibility of the various HAL packages that are applicable to a given target. Developers of components other than HAL packages need not be concerned about what is involved. Developers of new HAL packages should use an existing HAL as a template.

**Note:** It may be desirable for some packages to have some control over the linker script, for example to add extra alignment details for a particular section. This can be risky because it can result in subtle portability problems, and the current component framework has no support for any such operations. The issue may be addressed in a future release.

## Building Test Cases

### Caution

The support in the current implementation of the component framework for building and running test cases is limited, and should be enhanced considerably in a future version. Compatibility with the existing mechanisms described below will be maintained if possible, but this cannot be guaranteed.

Whenever possible packages should be shipped with one or more test cases. This allows users to check that all packages function correctly in their particular configuration and on their target, which may be custom hardware unavailable to the package developer. The component framework needs to provide a way of building such test cases. For example, if a makefile system is used then there could be a `make tests` target to build the test cases, or possibly a `make check` target to build and run the test cases and process all the results. Unfortunately there are various complications.

Not every test case will be applicable to every configuration. For example if the user has disabled the C library's `CYGPKG_LIBC_STDIO` component then there is no point in building or running any of the test cases for that component. This implies that test cases need to be associated with configuration options somehow. It is possible for the test case to use one or more `#ifdef` statements to check whether or not it is applicable in the current configuration, and compile to a null program when not applicable. This is inefficient because the test case will still get built and possibly run, even though it will not provide any useful information.

Many packages involve direct interaction with hardware, for example a serial line or an ethernet interface. In such cases it is only worthwhile building and running the test if there is suitable software running at the other end of the serial line or listening on the same ethernet segment, and that software would typically have to run on the host. Of course the serial line in question may be hooked up to a different piece of hardware which the application needs to talk to, so disconnecting it and then hooking it up to the host for running some tests may be undesirable. The decision as to whether or not to build the test depends not just on the eCos configuration but also on the hardware setup and the availability of suitable host software.

There are different kinds of tests, and it is not always desirable to run all of them. For example a package may contain a number of stress tests intended to run for long periods of time, possibly days or longer. Such tests should certainly be distinguished somehow from ordinary test cases so that users will not run them accidentally and wonder how long they should wait for a `pass` message before giving up. Stress tests may also have dependencies on the hardware configuration and on host software, for example a network stress test may require lots of ethernet packets.

In the current implementation of the component framework these issues are not yet addressed. Instead there is only very limited support for building test cases. Any package can define a calculated configuration option of the form `CYGPKG_<package-name>_TESTS`, whose value is a list of test cases. The calculated property can involve an expression so it is possible to adapt to a small number of configuration options, but this quickly becomes unwieldy. A typical example would be:

```
cdl_option CYGPKG_UITRON_TESTS {
    display "uITRON tests"
    flavor data
    no_define
    calculated { "tests/test1 tests/test2 tests/test3 \
        tests/test4 tests/test5 tests/test6 tests/test7 \
        tests/test8 tests/test9 tests/testcxx tests/testcx2 \
        tests/testcx3 tests/testcx4 tests/testcx5 \
        tests/testcx6 tests/testcx7 tests/testcx8 \
        tests/testcx9 tests/testintr" }
    description "
This option specifies the set of tests for the uITRON compatibility layer."
}
```

This implies that there is a file `tests/test1.c` or `tests/test1.cxx` in the package's directory. The commands that will be used to build the test case will take the form:

```
$(CC) -c $(INCLUDE_PATH) $(CFLAGS) -o <build path>/test1.o \
    <source path>/tests/test1.c
$(CC) $(LDFLAGS) -o <install path>/tests/test1 <build path>/test1.o
```

The variables `$(CC)` and so on are determined in the same way as for custom build steps. The various paths and the current directory will depend on the exact build system being used, and are subject to change. As usual the sources in the component repository are treated as a read-only resources, intermediate files live in the build tree, and the desired executables should end up in the install tree.

Each test source file must be self-contained. It is not possible at present to build a little per-package library that can be used by the test cases, or to link together several object files to produce a single test executable. In some cases it may be possible to `#include` source code from a shared file in order to avoid unnecessary code replication. There is no support for manipulating compiler or linker flags for individual test cases: the flags that will be used for all files are `$(CFLAGS)` and `$(LDFLAGS)`, as per custom build steps. Note that it is possible for a package to define options of the form `CYGPKG_<PACKAGE-NAME>_LDFLAGS_ADD` and `CYGPKG_<PACKAGE-NAME>_LDFLAGS_REMOVE`. These will affect test cases, but in the absence of custom build steps they will have no other effect on the build.

# Chapter 5. CDL Language Specification

This chapter contains reference information for the main CDL commands `cdl_option`, `cdl_component`, `cdl_package` and `cdl_interface`, followed by the various properties such as `active_if` and `compile` in alphabetical order.

## `cdl_option`

### Name

Command `cdl_option` — Define a single configuration option

### Synopsis

```
cdl_option <name> {  
    ...  
}
```

### Description

The option is the basic unit of configurability. Generally each option corresponds to a single user choice. Typically there is a certain amount of information associated with an option to assist the user in manipulating that option, for example a textual description. There will also be some limits on the possible values that the user can choose, so an option may be a simple yes-or-no choice or it may be something more complicated such as an array size or a device name. Options may have associated constraints, so if that option is enabled then certain conditions have to be satisfied elsewhere in the configuration. Options usually have direct consequences such as preprocessor `#define` symbols in a configuration header file.

`cdl_option` is implemented as a Tcl command that takes two arguments, a name and a body. The name must be a valid C preprocessor identifier: a sequence of upper or lower case letters, digits or underscores, starting with a non-digit character; identifiers beginning with an underscore should normally be avoided because they may clash with system packages or with identifiers reserved for use by the compiler. Within a single configuration, names must be unique. If a configuration contained two packages which defined the same entity `CYGIMP_SOME_OPTION`, any references to that entity in a `requires` property or any other expression would be ambiguous. It is possible for a given name to be used by two different packages if those packages should never be loaded into a single configuration. For example, architectural HAL packages are allowed to re-use names because a single configuration cannot target two different architectures. For a recommended naming convention see [the Section called \*Package Contents and Layout\* in Chapter 2](#).

The second argument to `cdl_option` is a body of properties, typically surrounded by braces so that the Tcl interpreter treats it as a single argument. This body will be processed by a recursive invocation of the Tcl interpreter, extended with additional commands for the various properties that are allowed inside a `cdl_option`. The valid properties are:

*cdl\_option*

**active\_if**

Allow additional control over the active state of this option.

**calculated**

The option's value is not directly user-modifiable, it is calculated using a suitable CDL expression.

**compile**

List the source files that should be built if this option is active and enabled.

**default\_value**

Provide a default value for this option using a CDL expression.

**define**

Specify additional `#define` symbols that should go into the owning package's configuration header file.

**define\_format**

Control how the option's value will appear in the configuration header file.

**define\_proc**

Use a fragment of Tcl code to output additional data to configuration header files.

**description**

Provide a textual description for this option.

**display**

Provide a short string describing this option.

**doc**

The location of on-line documentation for this option.

**flavor**

Specify the nature of this option.

**if\_define**

Output a common preprocessor construct to a configuration header file.

**implements**

Enabling this option provides one instance of a more general interface.

**legal\_values**

Impose constraints on the possible values for this option.

**make**

An additional custom build step associated with this option, resulting in a target that should not go directly into a library.

[make\\_object](#)

An additional custom build step associated with this option, resulting in an object file that should go into a library.

[no\\_define](#)

Suppress the normal generation of a preprocessor `#define` symbol in a configuration header file.

[parent](#)

Control the location of this option in the configuration hierarchy.

[requires](#)

List constraints that the configuration should satisfy if this option is active and enabled.

## Example

```
cdl_option CYGDBG_INFRA_DEBUG_PRECONDITIONS {
    display      "Preconditions"
    default_value 1
    description  "
        This option allows individual control of preconditions.
        A precondition is one type of assert, which it is
        useful to control separately from more general asserts.
        The function is CYG_PRECONDITION(condition,msg)."
}
```

## See Also

Command [cdl\\_component](#), command [cdl\\_package](#), command [cdl\\_interface](#).

*cdl\_option*

## cdl\_component

### Name

Command `cdl_component` — Define a component, a collection of configuration options

### Synopsis

```
cdl_component <name> {  
    ...  
}
```

### Description

A component is a configuration option that can contain additional options and sub-components. The body of a `cdl_component` can contain the same properties as that of a `cdl_option`. There is an additional property, `script` which allows configuration data to be split into multiple files. It is also possible for a component body to include `cdl_component`, `cdl_option` and `cdl_interface` entities that should go below this component in the configuration hierarchy.

`cdl_component` is implemented as a Tcl command that takes two arguments, a name and a body. The name must be a valid C preprocessor identifier: a sequence of upper or lower case letters, digits or underscores, starting with a non-digit character; identifiers beginning with an underscore should normally be avoided because they may clash with system packages or with identifiers reserved for use by the compiler. Within a single configuration, names must be unique. If a configuration contained two packages which defined the same entity `CYGIMP_SOME_OPTION`, any references to that entity in a `requires` property or any other expression would be ambiguous. It is possible for a given name to be used by two different packages if those packages should never be loaded into a single configuration. For example, architectural HAL packages are allowed to re-use certain names because a single configuration cannot target two different architectures. For a recommended naming convention see [the Section called \*Package Contents and Layout\* in Chapter 2](#).

The second argument to `cdl_component` is a body of properties and other commands, typically surrounded by braces so that the Tcl interpreter treats it as a single argument. This body will be processed by a recursive invocation of the Tcl interpreter, extended with additional commands for the various properties that are allowed inside a `cdl_component`. The valid commands are:

#### [active\\_if](#)

Allow additional control over the active state of this component.

#### [calculated](#)

The component's value is not directly user-modifiable, it is calculated using a suitable CDL expression.

#### [cdl\\_component](#)

Define a sub-component.

*cdl\_component*

**cdl\_interface**

Define an interface which should appear immediately below this component in the configuration hierarchy.

**cdl\_option**

Define a configuration option which should appear immediately below this component in the configuration hierarchy.

**compile**

List the source files that should be built if this component is active and enabled.

**default\_value**

Provide a default value for this component using a CDL expression.

**define**

Specify additional `#define` symbols that should go into the owning package's configuration header file.

**define\_format**

Control how the component's value will appear in the configuration header file.

**define\_proc**

Use a fragment of Tcl code to output additional data to configuration header files.

**description**

Provide a textual description for this component.

**display**

Provide a short string describing this component.

**doc**

The location of on-line documentation for this component.

**flavor**

Specify the nature of this component.

**if\_define**

Output a common preprocessor construct to a configuration header file.

**implements**

Enabling this component provides one instance of a more general interface.

**legal\_values**

Impose constraints on the possible values for this component.



**make**

An additional custom build step associated with this component, resulting in a target that should not go directly into a library.

**make\_object**

An additional custom build step associated with this component, resulting in an object file that should go into a library.

**no\_define**

Suppress the normal generation of a preprocessor `#define` symbol in a configuration header file.

**parent**

Control the location of this component in the configuration hierarchy.

**requires**

List constraints that the configuration should satisfy if this component is active and enabled.

**script**

Include additional configuration information from another CDL script

## Example

```
cdl_component CYGDBG_USE_ASSERTS {
    display      "Use asserts"
    default_value 1
    description  "
        If this component is enabled, assertions in the code are
        tested at run-time. Assert functions (CYG_ASSERT()) are
        defined in 'include/cyg/infra/cyg_ass.h' within the 'install'
        tree. If the component is disabled, these result in no
        additional object code and no checking of the asserted
        conditions."
    script      assert.cdl
}
```

## See Also

Command [cdl\\_option](#), command [cdl\\_package](#), command [cdl\\_interface](#).

*cdl\_component*

## cdl\_package

### Name

Command `cdl_package` — Define a package, a component that can be distributed

### Synopsis

```
cdl_package <name> {  
    ...  
}
```

### Description

A package is a unit of distribution. It is also a configuration option in that users can choose whether or not a particular package is loaded into the configuration, and which version of that package should be loaded. It is also a component in that it can contain additional components and options in a hierarchy.

The top-level CDL script for a package should begin with a `cdl_package` command. This can contain most of the properties that can be used in a `cdl_option` command, and a number of additional ones which apply to a package as a whole. It is also possible to include `cdl_component`, `cdl_interface` and `cdl_option` commands in the body of a package. However all configuration entities that occur at the top level of the script containing the `cdl_package` command are automatically placed below that package in the configuration hierarchy, so putting them inside the body has no effect.

The following properties cannot be used in the body of a `cdl_package` command:

`flavor`

Packages always have the `flavor booldata`.

`default_value`

The value of a package is its version number. This is specified at the time the package is loaded into the configuration, and cannot be calculated afterwards. Typically the most recent version of the package will be loaded.

`legal_values`

The legal values list for a given package is determined by which versions of that package are installed in the component repository, and cannot be further constrained in the CDL scripts.

`calculated`

The value of a package is always selected at the time that it is loaded into the configuration, and cannot be re-calculated afterwards.

`script`

This would be redundant since the CDL script containing the `cdl_package` command acts as that package's script.

`cdl_package` is implemented as a Tcl command that takes two arguments, a name and a body. The name must be a valid C preprocessor identifier: a sequence of upper or lower case letters, digits or underscores, starting with a non-digit character; identifiers beginning with an underscore should normally be avoided because they may clash with system packages or with identifiers reserved for use by the compiler. Packages should always have unique names within a given component repository. For a recommended naming convention see [the Section called \*Package Contents and Layout\* in Chapter 2](#).

The second argument to `cdl_package` is a body of properties and other commands, typically surrounded by braces so that the Tcl interpreter treats it as a single argument. This body will be processed by a recursive invocation of the Tcl interpreter, extended with additional commands for the various properties that are allowed inside a `cdl_package`. The valid commands are:

#### `active_if`

Allow additional control over the active state of this package.

#### `cdl_component`

Define a component which should appear immediately below this package in the configuration hierarchy.

#### `cdl_interface`

Define an interface which should appear immediately below this package in the configuration hierarchy.

#### `cdl_option`

Define an option which should appear immediately below this package in the configuration hierarchy.

#### `compile`

List the source files that should be built for this package.

#### `define`

Specify additional `#define` symbols that should go into the package's configuration header file.

#### `define_format`

Control how the package's value will appear in the global configuration header file `pkgconf/system.h`

#### `define_header`

Specify the configuration header file that will be generated for this package.

#### `define_proc`

Use a fragment of Tcl code to output additional data to configuration header files.

#### `description`

Provide a textual description for this component.

#### `display`

Provide a short string describing this component.

**doc**

The location of on-line documentation for this component.

**hardware**

This package is tied to specific hardware.

**if\_define**

Output a common preprocessor construct to a configuration header file.

**implements**

Enabling this component provides one instance of a more general interface.

**include\_dir**

Specify the desired location of this package's exported header files in the install tree.

**include\_files**

List the header files that are exported by this package.

**library**

Specify which library should contain the object files generated by building this package.

**make**

An additional custom build step associated with this component, resulting in a target that should not go directly into a library.

**make\_object**

An additional custom build step associated with this component, resulting in an object file that should go into a library.

**no\_define**

Suppress the normal generation of the package's `#define` in the global configuration header file `pkgconf/system.h`.

**parent**

Control the location of this package in the configuration hierarchy.

**requires**

List constraints that the configuration should satisfy if this package is active.

## Example

```
cdl_package CYGPKG_INFRA {
    display      "Infrastructure"
    include_dir  cyg/infra
    description  "
```

*cdl\_package*

Common types and useful macros.  
Tracing and assertion facilities.  
Package startup options."

```
compile startup.cxx prestart.cxx pkgstart.cxx userstart.cxx      \  
        dummyxxmain.cxx null.cxx simple.cxx fancy.cxx buffer.cxx \  
        diag.cxx tcdiag.cxx memcpy.c memset.c delete.cxx  
}
```

## See Also

Command [cdl\\_option](#), command [cdl\\_component](#), command [cdl\\_interface](#).

## cdl\_interface

### Name

Command `cdl_interface` — Define an interface, functionality that can be provided by a number of different implementations.

### Synopsis

```
cdl_interface <name> {  
    ...  
}
```

### Description

An interface is a special type of calculated configuration option. It provides an abstraction mechanism that is often useful in CDL expressions. As an example, suppose that some package relies on the presence of code that implements the standard kernel scheduling interface. However the requirement is no more stringent than this, so the constraint can be satisfied by the `mlqueue` scheduler, the `bitmap` scheduler, or any additional schedulers that may get implemented in future. A first attempt at expressing the dependency might be:

```
requires CYGSEM_KERNEL_SCHED_MLQUEUE || CYGSEM_KERNEL_SCHED_BITMAP
```

This constraint is limited, it may need to be changed if a new scheduler were to be added to the system. Interfaces provide a way of expressing more general relationships:

```
requires CYGINT_KERNEL_SCHEDULER
```

The interface `CYGINT_KERNEL_SCHEDULER` is *implemented* by both the `mlqueue` and `bitmap` schedulers, and may be implemented by future schedulers as well. The value of an interface is the number of implementors that are active and enabled, so in a typical configuration only one scheduler will be in use and the value of the interface will be 1. If all schedulers are disabled then the interface will have a value 0 and the `requires` constraint will not be satisfied.

Some component writers may prefer to use the first `requires` constraint on the grounds that the code will only have been tested with the `mlqueue` and `bitmap` schedulers and cannot be guaranteed to work with any new schedulers. Other component writers may take a more optimistic view and assume that their code will work with any scheduler until proven otherwise.

Interfaces must be defined in CDL scripts, just like options, components and packages. This involves the command `cdl_interface` which takes two arguments, a name and a body. The name must be a valid C preprocessor identifier: a sequence of upper or lower case letters, digits or underscores, starting with a non-digit character; identifiers beginning with an underscore should normally be avoided because they may clash with system packages or with identifiers reserved for use by the compiler. Within a single configuration, names must be unique. If a configuration contained two packages which defined the same entity `CYGIMP_SOME_OPTION`, any references to that entity in a `requires` property or any other expression would be ambiguous. It is possible for a given name to be used by two different packages if those packages should never be loaded into a single configuration. For example, architectural

## `cdl_interface`

HAL packages are allowed to re-use names because a single configuration cannot target two different architectures. For a recommended naming convention see [the Section called \*Package Contents and Layout\* in Chapter 2](#).

The second argument to `cdl_interface` is a body of properties, typically surrounded by braces so that the Tcl interpreter treats it as a single argument. This body will be processed by a recursive invocation of the Tcl interpreter, extended with additional commands for the various properties that are allowed inside a `cdl_interface`. The valid properties are a subset of those for a `cdl_option`.

### `active_if`

Allow additional control over the active state of this interface.

### `compile`

List the source files that should be built if this interface is active.

### `define`

Specify additional `#define` symbols that should go into the owning package's configuration header file.

### `define_format`

Control how the interface's value will appear in the configuration header file.

### `define_proc`

Use a fragment of Tcl code to output additional data to configuration header files.

### `description`

Provide a textual description for this interface.

### `display`

Provide a short string describing this interface.

### `doc`

The location of on-line documentation for this interface.

### `flavor`

Interfaces have the `data` flavor by default, but they can also be given the `bool` or `booldata` flavor when necessary. A `bool` interface is disabled if there are no active and enabled implementors, otherwise it is enabled. A `booldata` interface is also disabled if there are no active and enabled implementors, otherwise it is enabled and the data is a number corresponding to the number of these implementors.

### `if_define`

Output a common preprocessor construct to a configuration header file.

### `implements`

If this interface is active it provides one instance of a more general interface.



### legal\_values

Interfaces always have a small numerical value. The legal\_values can be used to apply additional constraints such as an upper limit.

### make

An additional custom build step associated with this option, resulting in a target that should not go directly into a library.

### make\_object

An additional custom build step associated with this option, resulting in an object file that should go into a library.

### no\_define

Suppress the normal generation of a preprocessor #define symbol in a configuration header file.

### parent

Control the location of this option in the configuration hierarchy.

### requires

List constraints that the configuration should satisfy if this option is active and enabled.

A number of properties are not applicable to interfaces:

### calculated

Interfaces are always calculated, based on the number of active and enabled entities that implement the interface.

### default\_value

Interface values are calculated so a default\_value property would be meaningless.

Interfaces are not containers, so they cannot hold other entities such as options or components.

A commonly used constraint on interface values takes the form:

```
requires CYGINT_KERNEL_SCHEDULER == 1
```

This constraint specifies that there can be only one scheduler in the system. In some circumstances it is possible for the configuration tools to detect this pattern and act accordingly, so for example enabling the bitmap scheduler would automatically disable the mlqueue scheduler.

## Example

```
cdl_interface CYGINT_KERNEL_SCHEDULER {
    display "Number of schedulers in this configuration"
    requires 1 == CYGINT_KERNEL_SCHEDULER
}
```

*cdl\_interface*

## See Also

Property [implements](#), command [cdl\\_option](#), command [cdl\\_component](#), command [cdl\\_package](#).

# active\_if

## Name

Property `active_if` — Allow additional control over the active state of an option or other CDL entity.

## Synopsis

```
cdl_option <name> {  
    active_if <condition>  
    ...  
}
```

## Description

Configuration options or other entities may be either active or inactive. Typically this is controlled by the option's location in the overall hierarchy. Consider the option `CYGDBG_INFRA_DEBUG_PRECONDITIONS`, which exists below the component `CYGDBG_USE_ASSERT`. If the whole component is disabled then the options it contains are inactive: there is no point in enabling preconditions unless there is generic assertion support; any requires constraints associated with preconditions are irrelevant; any compile property or other build-related property is ignored.

In some cases the hierarchy does not provide sufficient control over whether or not a particular option should be active. For example, the math library could have support for floating point exceptions which is only worthwhile if the hardware implements appropriate functionality, as specified by the architectural HAL. The relevant math library configuration options should remain below the `CYGPKG_LIBM` package in the overall hierarchy, but should be inactive unless there is appropriate hardware support. In cases like this an `active_if` property is appropriate.

Another common use of `active_if` properties is to avoid excessive nesting in the configuration hierarchy. If some option B is only relevant if option A is enabled, it is possible to turn A into a component that contains B. However adding another level to the hierarchy for a component which will contain just one entry may be considered excessive. In such cases it is possible for B to have an `active_if` dependency on A.

`active_if` takes a goal expression as argument. For details of goal expression syntax see [the Section called \*Goal Expressions\* in Chapter 3](#). In most cases the goal expression will be very simple, often involving just one other option, but more complicated expressions can be used when appropriate. It is also possible to have multiple `active_if` conditions in a single option, in which case all of the conditions have to be satisfied if the option is to be active.

The `active_if` and `requires` properties have certain similarities, but they serve a different purpose. Suppose there are two options A and B, and option B relies on functionality provided by A. This could be expressed as either `active_if A` or as `requires A`. The points to note are:

- If `active_if A` is used and A is disabled or inactive, then graphical tools will generally prevent any attempt at modifying B. For example the text for B could be grayed out, and the associated checkbox (if B is a boolean option) would be disabled. If the user needs the functionality provided by option B then it is necessary to go to option A first and manipulate it appropriately.

- If `requires A` is used and A is disabled or inactive, graphical tools will still allow B to be manipulated and enabled. This would result in a new conflict which may get resolved automatically or which may need user intervention.
- If there are hardware dependencies then an `active_if` condition is usually the preferred approach. There is no point in allowing the user to manipulate a configuration option if the corresponding functionality cannot possibly work on the currently-selected hardware. Much the same argument applies to coarse-grained dependencies, for example if an option depends on the presence of a TCP/IP stack then an `active_if CYGPKG_NET` condition is appropriate: it may be possible to satisfy the condition, but it requires the fairly drastic step of loading another package; further more, if the user wanted a TCP/IP stack in the configuration then it would probably have been loaded already.
- If option B exists to provide additional debugging information about the functionality provided by A then again an `active_if` constraint is appropriate. There is no point in letting users enable extra debugging facilities for a feature that is not actually present.
- The configuration system's inference engine will cope equally well with `active_if` and `requires` properties. Suppose there is a conflict because some third option depends on B. If B is `active_if A` then the inference engine will attempt to make A active and enabled, and then to enable B if necessary. If B `requires A` then the inference engine will attempt to enable B and resolve the resulting conflict by causing A to be both active and enabled. Although the inference occurs in a different order, in most cases the effect will be the same.

## Example

```
# Do not provide extra semaphore debugging if there are no semaphores
cdl_option CYGDBG_KERNEL_INSTRUMENT_BINSEM {
    active_if CYGPKG_KERNEL_SYNC
    ...
}

# Avoid another level in the configuration hierarchy
cdl_option CYGSEM_KERNEL_SYNC_MUTEX_PRIORITY_INHERITANCE_SIMPLE_RELAY {
    active_if CYGSEM_KERNEL_SYNC_MUTEX_PRIORITY_INHERITANCE_SIMPLE
    ...
}

# Functionality that is only relevant if another package is loaded
cdl_option CYGSEM_START_UITRON_COMPATIBILITY {
    active_if CYGPKG_UITRON
    ...
}

# Check that the hardware or HAL provide the appropriate functionality
cdl_option CYGDBG_HAL_DEBUG_GDB_BREAK_SUPPORT {
    active_if CYGINT_HAL_DEBUG_GDB_STUBS_BREAK
    ...
}
```

## See Also

Property [requires](#).

*active\_if*

# calculated

## Name

Property `calculated` — Used if the current option's value is not user-modifiable, but is calculated using a suitable CDL expression.

## Synopsis

```
cdl_option <name> {  
    calculated <expression>  
    ...  
}
```

## Description

In some cases it is useful to have a configuration option whose value cannot be modified directly by the user. This can be achieved using a `calculated`, which takes a CDL expression as argument (see [the Section called \*Ordinary Expressions\* in Chapter 3](#) for a description of expression syntax). The configuration system evaluates the expression when the current package is loaded and whenever there is a change to any other option referenced in the expression. The result depends on the option's flavor:

flavor none

Options with this flavor have no value, so the calculated property is not applicable.

flavor bool

If the expression evaluates to a non-zero result the option is enabled, otherwise it is disabled.

flavor booldata

If the result of evaluating the expression is zero then the option is disabled, otherwise the option is enabled and its value is the result.

flavor data

The value of the option is the result of evaluating the expression.

There are a number of valid uses for calculated options, and there are also many cases where some other CDL facility would be more appropriate. Valid uses of calculated options include the following:

- On some target hardware a particular feature may be user-configurable, while on other targets it is fixed. For example some processors can operate in either big-endian or little-endian mode, while other processors do not provide any choice. It is possible to have an option `CYGARC_BIGENDIAN` which is calculated in some architectural HAL packages but user-modifiable in others.
- Calculated options can provide an alternative way for one package to affect the behavior of another one. Suppose a package may provide two possible implementations, a preferred one involving self-modifying code and a

slower alternative. If the system involves a ROM bootstrap then the slower alternative must be used, but it would be inappropriate to modify the startup option in every HAL to impose constraints on this package. Instead it is possible to have a calculated option whose value is { `CYG_HAL_STARTUP == "ROM"` }, and which has appropriate consequences. Arguably this is a spurious example, and it should be a user choice whether or not to use self-modifying code with a default\_value based on `CYG_HAL_STARTUP`, but that is for the component writer to decide.

- Sometimes it should be possible to perform a particular test at compile-time, for example by using a C preprocessor `#if` construct. However the preprocessor has only limited functionality, for example it cannot perform string comparisons. CDL expressions are more powerful.
- Occasionally a particular sub-expression may occur multiple times in a CDL script. If the sub-expression is sufficiently complex then it may be worthwhile to have a calculated option whose value is the sub-expression, and then reference that calculated option in the appropriate places.

Alternatives to using calculated options include the following:

- CDL [interfaces](#) are a form of calculated option intended as an abstraction mechanism. An interface can be used to express the concept of *any scheduler*, as opposed to a specific one such as the bitmap scheduler.
- If a calculated option would serve only to add additional information to a configuration header file, it may be possible to achieve the same effect using a [define\\_proc](#) property or one of the other properties related to header file generation.

**Tip:** If the first entry in a calculated expression is a negative number, for example `calculated -1` then this can be misinterpreted as an option instead of as part of the expression. Currently the calculated property does not take any options, but this may change in future. Option processing halts at the sequence `--`, so the desired value can be expressed safely using `calculated -- -1`

### Warning

Some of the CDL scripts in current eCos releases make excessive use of calculated options. This is partly because the recommended alternatives were not always available at the time the scripts were written. It is also partly because there is still some missing functionality, for example `define_proc` properties cannot yet access the configuration data so it may be necessary to use calculated properties to access the data and perform the desired manipulation via a CDL expression. New scripts should use calculated options only in accordance with the guidelines.

**Note:** For options with the booldata flavor the current CDL syntax does not allow the enabled flag and the value to be calculated separately. Functionality to permit this may be added in a future release.

**Note:** It has been suggested that having options which are not user-modifiable is potentially confusing, and that a top-level `cdl_constant` command should be added to the language instead of or in addition to the calculated property. Such a change is under consideration. However because the value of a calculated option can depend on other options, it is not necessarily constant.



## Example

```
# A constant on some target hardware, perhaps user-modifiable on other
# targets.
cdl_option CYGNUM_HAL_RTC_PERIOD {
    display      "Real-time clock period"
    flavor       data
    calculated    12500
}
```

## See Also

Properties [default\\_value](#), [flavor](#) and [legal\\_values](#),

*calculated*

# compile

## Name

Property `compile` — List the source files that should be built if this option is active and enabled.

## Synopsis

```
cdl_option <name> {  
    compile [-library=libxxx.a] <list of files>  
    ...  
}
```

## Description

The `compile` property allows component developers to specify source files which should be compiled and added to one of the target libraries. Usually each source file will end up the library `libtarget.a`. It is possible for component writers to specify an alternative library for an entire package using the [library](#) property. Alternatively the desired library can be specified on the `compile` line itself. For example, to add a particular source file to the `libextras.a` library the following could be used:

```
cdl_package CYGPKG_IO_SERIAL {  
    ...  
    compile -library=libextras.a common/tty.c  
}
```

Details of the build process including such issues as compiler flags and the order in which things happen can be found in [Chapter 4](#).

`compile` properties can occur in any of `cdl_option`, `cdl_component`, `cdl_package` or `cdl_interface`. A `compile` property has effect if and only if the entity that contains it is active and enabled. Typically the body of a `cdl_package` will define any source files that need to be built irrespective of individual options, and each `cdl_component`, `cdl_option`, and `cdl_interface` will define source files that are more specific. A single `compile` property can list any number of source files, all destined for the same library. A `cdl_option` or other entity can contain multiple `compile` properties, each of which can specify a different library. It is possible for a given source file to be specified in `compile` properties for several different options, in which case the source file will get built if any of these options are active and enabled.

If the package follows the [directory layout conventions](#) then the configuration tools will search for the specified source files first in the `src` subdirectory of the package, then relative to the package directory itself.

**Note:** A shortcoming of the current specification of `compile` properties is that there is no easy way to specify source files that should be built unless an option is enabled. It would sometimes be useful to be able to say: “if option `A` is enabled then compile file `x.c`, otherwise compile file `y.c`. There are two simple ways of achieving this:

- Always compile `y.c`, typically by listing it in the body of the `cdl_package`, but use `#ifndef A` to produce an empty object file if option `A` is not enabled. This has the big disadvantage that the file always gets compiled and hence for some configurations builds will take longer than necessary.
- Use a calculated option whose value is `!A`, and have a `compile y.c` property in its body. This has the big disadvantage of adding another calculated option to the configuration.

It is likely that this will be resolved in the future, possibly by using some sort of expression as the argument to a compile property.

**Note:** Currently it is not possible to control the priority of a compile property, in other words the order in which a file gets compiled relative to other build steps. This functionality might prove useful for complicated packages and should be added.

## Example

```
cdl_package CYGPKG_INFRA {
  display      "Infrastructure"
  include_dir  cyg/infra
  description  "
    Common types and useful macros.
    Tracing and assertion facilities.
    Package startup options."

  compile startup.cxx prestart.cxx pkgstart.cxx userstart.cxx \
    dummyxxmain.cxx memcpy.c memset.c delete.cxx \
    diag.cxx tcdiag.cxx
}
```

## See Also

Properties [make](#), [make\\_object](#) and [library](#).

# default\_value

## Name

Property `default_value` — Provide a default value for this option using a CDL expression.

## Synopsis

```
cdl_option <name> {  
    default_value <expression>  
    ...  
}
```

## Description

The `default_value` property allows the initial value of a configuration option to depend on other configuration options. The arguments to the property should be a CDL expression, see [the Section called \*Ordinary Expressions\* in Chapter 3](#) for the syntactic details. In many cases a simple constant value will suffice, for example:

```
cdl_component CYGPKG_KERNEL_EXCEPTIONS {  
    ...  
    default_value 1  
  
    cdl_option CYGSEM_KERNEL_EXCEPTIONS_DECODE {  
        ...  
        default_value 0  
    }  
}
```

However it is also possible for an option’s default value to depend on other options. For example the common HAL package provides some support functions that are needed by the eCos kernel, but are unlikely to be useful if the kernel is not being used. This relationship can be expressed using:

```
cdl_option CYGFUN_HAL_COMMON_KERNEL_SUPPORT {  
    ...  
    default_value CYGPKG_KERNEL  
}
```

If the kernel is loaded then this HAL option is automatically enabled, although the user can still disable it explicitly should this prove necessary. If the kernel is not loaded then the option is disabled, although it can still be enabled by the user if desired. `default_value` expressions can be more complicated than this if appropriate, and provide a very powerful facility for component writers who want their code to “just do the right thing” in a wide variety of configurations.

The CDL configuration system evaluates the `default_value` expression when the current package is loaded and whenever there is a change to any other option referenced in the expression. The result depends on the option’s flavor:

## *default\_value*

flavor none

Options with this flavor have no value, so the `default_value` property is not applicable.

flavor bool

If the expression evaluates to a non-zero result the option is enabled by default, otherwise it is disabled.

flavor booldata

If the result of evaluating the expression is zero then the option is disabled, otherwise the option is enabled and its value is the result.

flavor data

The default value of the option is the result of evaluating the expression.

A `cdl_option` or other entity can have at most one `default_value` property, and it is illegal to have both a calculated and a `default_value` property in one body. If an option does not have either a `default_value` or a calculated property and it does not have the flavor `none` then the configuration tools will assume a default value expression of 0.

On occasion it is useful to have a configuration option `A` which has both a `requires` constraint on some other option `B` and a `default_value` expression of `B`. If option `B` is not enabled then `A` will also be disabled by default and no conflict arises. If `B` is enabled then `A` also becomes enabled and again no conflict arises. If a user attempts to enable `B` but not `A` then there will be a conflict. Users should be able to deduce that the two options are closely interlinked and should not be manipulated independently except in very unusual circumstances.

**Tip:** If the first entry in a `default_value` expression is a negative number, for example `default_value -1` then this can be misinterpreted as an option instead of as part of the expression. Currently the `default_value` property does not take any options, but this may change in future. Option processing halts at the sequence `--`, so the desired value can be expressed safely using `default_value -- -1`

**Note:** In many cases it would be useful to calculate default values using some global preferences, for example:

```
cdl_option CYGIMP_LIBC_STRING_PREFER_SMALL_TO_FAST {  
    ...  
    default_value CYGGLO_CODESIZE > CYGGLO_SPEED  
}
```

Such global preference options do not yet exist, but are likely to be added in a future version.

**Note:** For options with the `booldata` flavor the current syntax does not allow the default values of the enabled flag and the value to be controlled separately. Functionality to permit this may be added in a future release.

## Example

```
cdl_option CYGDBG_HAL_DEBUG_GDB_THREAD_SUPPORT {
```

```
display      "Include GDB multi-threading debug support"
requires     CYGDBG_KERNEL_DEBUG_GDB_THREAD_SUPPORT
default_value CYGDBG_KERNEL_DEBUG_GDB_THREAD_SUPPORT
description  "
    This option enables some extra HAL code which is needed
    to support multi-threaded source level debugging."
}
```

## See Also

Properties [calculated](#), [flavor](#) and [legal\\_values](#).

*default\_value*



# define

## Name

Property `define` — Specify additional `#define` symbols that should go into the owning package's configuration header file.

## Synopsis

```
cdl_option <name> {  
    define [-file=<filename>] [-format=<format>] <symbol>  
    ...  
}
```

## Description

Normally the configuration system generates a single `#define` for each option that is active and enabled, with the defined symbol being the name of the option. These `#define`'s go to the package's own configuration header file, for example `pkgconf/kernel.h` for kernel configuration options. For the majority of options this is sufficient. Sometimes it is useful to have more control over which `#define`'s get generated.

The `define` property can be used to generate an additional `#define` if the option is both active and enabled, for example:

```
cdl_option CYGNUM_LIBC_STDIO_FOPEN_MAX {  
    ...  
    define FOPEN_MAX  
}
```

If this option is given the value 40 then the following `#define`'s will be generated in the configuration header `pkgconf/libc.h`:

```
#define CYGNUM_LIBC_STDIO_FOPEN_MAX 40  
#define FOPEN_MAX 40
```

The default `#define` can be suppressed if desired using the `no_define` property. This is useful if the symbol should only be defined in `pkgconf/system.h` and not in the package's own configuration header file. The value that will be used for this `#define` is the same as for the default one, and depends on the option's flavor as follows:

`flavor none`

Options with this flavor are always enabled and have no value, so the constant 1 will be used.

`flavor bool`

If the option is disabled then no `#define` will be generated. Otherwise the constant 1 will be used.

`flavor booldata`

If the option is disabled then no `#define` will be generated. Otherwise the option's current value will be used.

## *define*

flavor data

The option's current value will be used.

For active options with the `data` flavor, and for active and enabled options with the `booldata` flavor, either one or two `#define`'s will be generated. These take the following forms:

```
#define <symbol> <value>
#define <symbol>_<value>
```

For the first `#define` it is possible to control the format used for the value using a `-format=<format>` option. For example, the following can be used to output some configuration data as a C string:

```
cdl_option <name> {
    ...
    define -format="\\\\"<symbol>
}
```

The implementation of this facility involves concatenating the Tcl command `format`, the format string, and the string representation of the option's value, and evaluating this in a Tcl interpreter. Therefore the format string will be processed twice by a Tcl parser, and appropriate care has to be taken with quoting.

The second `#define` will be generated only if is a valid C preprocessor macro symbol. By default the symbols generated by define properties will end up in the package's own configuration header file. The `-file` option can be used to specify an alternative destination. At the time of writing the only valid alternative definition is `-file=system.h`, which will send the output to the global configuration header file `pkgconf/system.h`.

### Caution

Care has to be taken with the `-format` option. Because the Tcl interpreter's `format` command is used, this property is subject to any problems with the implementation of this in the Tcl library. Generally there should be no problems with string data or with integers up to 32 bits, but there may well be problems if 64-bit data is involved. This issue may be addressed in a future release.

## Example

```
cdl_component CYG_HAL_STARTUP {
    display      "Startup type"
    flavor       data
    legal_values {"RAM" "ROM" }
    default_value {"RAM"}
    no_define
    define -file=system.h CYG_HAL_STARTUP
    ...
}
```

## See Also

Properties [define\\_format](#), [define\\_header](#), [define\\_proc](#), [if\\_define](#) and [no\\_define](#).

*define*

# define\_format

## Name

Property `define_format` — Control how an option's value will appear in the configuration header file.

## Synopsis

```
cdl_option <name> {  
    define_format <format string>  
    ...  
}
```

## Description

For active options with the `data` flavor, and for active and enabled options with the `booldata` flavor, the configuration tools will normally generate two `#define`'s in the package's configuration header file. These take the following forms:

```
#define <name> <value>  
#define <name>_<value>
```

The `define_format` property can be used to control exactly what appears as the value for the first of these `#define`'s. For example, the following can be used to output some configuration data as a C string:

```
cdl_option <name> {  
    ...  
    define -format="\\\\"<symbol>  
}
```

The implementation of `define_format` involves concatenating the Tcl command `format`, the format string, and the string representation of the option's value, and evaluating this in a Tcl interpreter. Therefore the format string will be processed twice by a Tcl parser, and appropriate care has to be taken with quoting.

The second `#define` will be generated only if is a valid C preprocessor macro symbol, and is not affected by the `define_format` property. Also, the property is only relevant for options with the `data` or `booldata` flavor, and cannot be used in conjunction with the `no_define` property since it makes no sense to specify the format if no `#define` is generated.

### Caution

Because the Tcl interpreter's `format` command is used, this property is subject to any problems with the implementation of this in the Tcl library. Generally there should be no problems with string data or with integers up to 32 bits, but there may well be problems if 64-bit data is involved. This issue may be addressed in a future release.

## Example

```
cdl_option CYGNUM_UITRON_VER_ID    {
    display      "OS identification"
    flavor       data
    legal_values 0 to 0xFFFF
    default_value 0
    define_format "0x%04x"
    description  "
        This value is returned in the 'id'
        field of the T_VER structure in
        response to a get_ver() system call."
}
```

## See Also

Properties [define](#), [define\\_header](#), [define\\_proc](#), [if\\_define](#) and [no\\_define](#).

# define\_header

## Name

Property `define_header` — Specify the configuration header file that will be generated for a given package.

## Synopsis

```
cdl_package <name> {  
    define_header <file name>  
    ...  
}
```

## Description

When the configuration tools generate a build tree, one of the steps is to output each package's configuration data to a header file. For example the kernel's configuration data gets output to `pkgconf/kernel.h`. This allows each package's source code to `#include` the appropriate header file and adapt to the choices made by the user.

By default the configuration tools will synthesize a file name from the package name. This involves removing any prefix such as `CYGPKG_`, up to and including the first underscore, and then converting the remainder of the name to lower case. In some cases it may be desirable to use a different header file, for example an existing package may have been ported to eCos and the source code may already `#include` a particular file for configuration data. In such cases a `define_header` property can be used to specify an alternative filename.

The `define_header` property can only be used in the body of a `cdl_package` command. It applies to a package as a whole and cannot be used at a finer grain. The name specified in a `define_header` property will always be interpreted as relative to the `include/pkgconf` sub-directory of the install tree.

**Note:** For hardware-specific packages such as device drivers and HAL packages, the current scheme of generating a configuration header file name based on the package name may be abandoned. Instead all hardware packages would send their configuration data to a single header file, `pkgconf/hardware.h`. This would make it easier for code to obtain details of the current hardware, but obviously there are compatibility issues. For now it is recommended that all hardware packages specify their configuration header file explicitly.

## Example

```
cdl_package CYGPKG_HAL_ARM {  
    display      "ARM architecture"  
    parent      CYGPKG_HAL  
    hardware  
    include_dir  cyg/hal  
    define_header hal_arm.h  
    ...  
}
```

*define\_header*

}

## See Also

Properties [define](#), [define\\_format](#), [define\\_proc](#), [if\\_define](#), [no\\_define](#) and [hardware](#),



# define\_proc

## Name

Property `define_proc` — Use a fragment of Tcl code to output additional data to configuration header files.

## Synopsis

```
cdl_option <name> {  
    define_proc <Tcl script>  
    ...  
}
```

## Description

For most configuration options it is sufficient to have an entry in the configuration header file of the form:

```
#define <name> <value>
```

In some cases it is desirable to perform some more complicated processing when generating a configuration header file. There are a number of CDL properties for this, including `define_format` and `if_define`. The most flexible is `define_proc`: this allows the component writer to specify a Tcl script that gets invoked whenever the configuration system generates the header file for the owning package. The script can output anything to the header file, for example it could generate a C data structure based on various configuration values.

At the point that the `define_proc` script is invoked there will be two channels to open files, accessible via Tcl variables: `cdl_header` is a channel to the current package's own header file for example `pkgconf/kernel.h`; `cdl_system_header` is a channel to the global configuration file `pkgconf/system.h`. A typical `define_proc` script will use the `puts` command to output data to one of these channels.

`define_proc` properties only take effect if the current option is active and enabled. The default behavior of the configuration system for an option with the `bool` flavor corresponds to the following `define_proc`:

```
cdl_option XXX {  
    ...  
    define_proc {  
        puts $cdl_header "#define XXX 1"  
    }  
}
```

### Warning

In the current implementation it is not possible for a `define_proc` property to examine the current values of various configuration options and adapt accordingly. This is a major limitation, and will be addressed in future versions of the configuration tools.

## Example

```
cdl_package CYGPKG_HAL_ARM_PID {
  display      "ARM PID evaluation board"
  parent       CYGPKG_HAL_ARM
  define_header hal_arm_pid.h
  include_dir   cyg/hal
  hardware

  define_proc {
    puts $::cdl_system_header "#define CYGBLD_HAL_TARGET_H    <pkgconf/hal_arm.h>"
    puts $::cdl_system_header "#define CYGBLD_HAL_PLATFORM_H  <pkgconf/hal_arm_pid.h>"
    puts $::cdl_header " "
    puts $::cdl_header "#define HAL_PLATFORM_CPU            \"ARM 7TDMI\""
    puts $::cdl_header "#define HAL_PLATFORM_BOARD         \"PID\""
    puts $::cdl_header "#define HAL_PLATFORM_EXTRA        \"\""
    puts $::cdl_header " "
  }
  ...
}
```

## See Also

Properties [define](#), [define\\_format](#), [define\\_header](#), [if\\_define](#) and [no\\_define](#).

# description

## Name

Property description — Provide a textual description for an option.

## Synopsis

```
cdl_option <name> {  
    description <text>  
    ...  
}
```

## Description

Users can only be expected to manipulate configuration options sensibly if they are given sufficient information about these options. There are three properties which serve to explain an option in plain text: the display property gives a textual alias for an option, which is usually more comprehensible than something like `CYGPKG_LIBC_TIME_ZONES`; the description property gives a longer description, typically a paragraph or so; the doc property specifies the location of additional on-line documentation related to a configuration option. In the context of a graphical tool the display string will be the primary way for users to identify configuration options; the description paragraph will be visible whenever the option is selected; the on-line documentation will only be accessed when the user explicitly requests it.

At present there is no way of providing any sort of formatting mark-up in a description. It is possible that future versions of the configuration tools will provide some control over the way the description text gets rendered.

## Example

```
cdl_option CYGDBG_INFRA_DEBUG_TRACE_MESSAGE {  
    display      "Use trace text"  
    default_value 1  
    description  "  
        All trace calls within eCos contain a text message  
        which should give some information about the circumstances.  
        These text messages will end up being embedded in the  
        application image and hence there is a significant penalty  
        in terms of image size.  
        It is possible to suppress the use of these messages by  
        disabling this option.  
        This results in smaller code size, but there is less  
        human-readable information available in the trace output,  
        possibly only filenames and line numbers."  
}
```

*description*

## **See Also**

Properties [display](#) and [doc](#).

# display

## Name

Property `display` — Provide a short string describing this option.

## Synopsis

```
cdl_option <name> {  
    display <string>  
    ...  
}
```

## Description

Users can only be expected to manipulate configuration options sensibly if they are given sufficient information about these options. There are three properties which serve to explain an option in plain text: the `display` property gives a textual alias for an option, which is usually more comprehensible than something like `CYGPKG_LIBC_TIME_ZONES`; the `description` property gives a longer description, typically a paragraph or so; the `doc` property specifies the location of additional on-line documentation related to a configuration option. In the context of a graphical tool the `display` string will be the primary way for users to identify configuration options; the `description` paragraph will be visible whenever the option is selected; the on-line documentation will only be accessed when the user explicitly requests it.

## Example

```
cdl_option CYGNUM_KERNEL_SYNCH_MBOX_QUEUE_SIZE {  
    display      "Message box queue size"  
    flavor      data  
    legal_values 1 to 65535  
    default_value 10  
    description  "  
        This configuration option controls the number of messages  
        that can be queued in a message box before a non-blocking  
        put() operation will fail or a blocking put() operation will  
        block. The cost in memory is one pointer per message box for  
        each possible message."  
}
```

## See Also

Properties [description](#) and [doc](#).

*display*

# doc

## Name

Property `doc` — The location of online-documentation for a configuration option.

## Synopsis

```
cdl_option <name> {  
    doc <URL;>  
    ...  
}
```

## Description

Users can only be expected to manipulate configuration options sensibly if they are given sufficient information about these options. There are three properties which serve to explain an option in plain text: the `display` property gives a textual alias for an option, which is usually more comprehensible than something like `CYGPKG_LIBC_TIME_ZONES`; the `description` property gives a longer description, typically a paragraph or so; the `doc` property specifies the location of additional on-line documentation related to a configuration option. In the context of a graphical tool the display string will be the primary way for users to identify configuration options; the description paragraph will be visible whenever the option is selected; the on-line documentation will only be accessed when the user explicitly requests it.

The documentation may be an absolute URL, but more generally the on-line documentation will be shipped with the package and can be accessed via a relative URL. If the package follows the [directory layout conventions](#) then the configuration tools will search for the specified html file first in the `doc` subdirectory of the package, then relative to the package directory itself. The URL may contain a `#` character to specify an anchor within a page.

### Warning

At the time of writing the eCos packages in the standard distribution do not conform to the directory layout conventions when it comes to the documentation. Instead of organizing the documentation on a per-package basis and placing it in the corresponding `doc` sub-directories, all the documentation is kept in a central location. This should get addressed in a future release of the system. Third party component writers should follow the layout conventions.

## Example

```
cdl_package CYGPKG_KERNEL {  
    display      "eCos kernel"  
    doc          ref/ecos-ref.4.html  
    include_dir  cyg/kernel  
    description  "
```

*doc*

```
    This package contains the core functionality of the eCos
    kernel. It relies on functionality provided by various HAL
    packages and by the eCos infrastructure. In turn the eCos
    kernel provides support for other packages such as the device
    drivers and the uITRON compatibility layer."
    ...
}
```

## See Also

Properties [description](#) and [display](#).



# flavor

## Name

Property `flavor` — Specify the nature of a configuration option.

## Synopsis

```
cdl_option <name> {  
    flavor <flavor>  
    ...  
}
```

## Description

The state of a CDL configuration option is a somewhat complicated concept. This state determines what happens when a build tree is generated: it controls what files get built and what `#define`'s end up in configuration header files. The state also controls the values used during expression evaluation. The key concepts are:

1. An option may or may not be loaded into the current configuration. However it is still possible for packages to reference options which are not loaded in a `requires` constraint or other expression. If an option is not loaded then it will have no direct effect on the build process, and 0 will be used for expression evaluation.
2. Even if an option is loaded it may still be inactive. Usually this is controlled by the option's location in the configuration hierarchy. If an option's parent is active and enabled then the option will normally be active. If the parent is either inactive or disabled then the option will be inactive. For example, if kernel timeslicing is disabled then the option `CYGNUM_KERNEL_SCHED_TIMESLICE_TICKS` is irrelevant and must have no effect. The `active_if` property can be used to specify additional constraints. If an option is inactive then it will have no direct effect on the build process, in other words it will not cause any files to get built or `#define`'s to be generated. For the purposes of expression evaluation an inactive option has a value of 0.
3. An option may be enabled or disabled. Most options are boolean in nature, for example a particular function may get inlined or it may involve a full procedure call. If an option is disabled then it has no direct effect on the build process, and for the purposes of expression evaluation it has a value of 0.
4. An option may also have additional data associated with it, for example a numerical value used to control the size of an array.

Most options are boolean in nature and do not have any additional associated data. For some options only the data part makes sense and users should be unable to manipulate the enabled/disabled part of the state. For a comparatively small number of options it makes sense to have the ability to disable that option or to enable it and associate data as well. Finally, when constructing an option hierarchy it is occasionally useful to have entities which serve only as placeholders. The `flavor` property can be used to control all this. There are four possible values. It should be noted that the active or inactive state of an option takes priority over the flavor: if an option is inactive then no `#define`'s will be generated and any build-related properties such as `compile` will be ignored.

**flavor none**

The `none` is intended primarily for placeholder components in the hierarchy, although it can be used for other purposes. Options with this flavor are always enabled and do not have any additional data associated with them, so there is no way for users to modify the option. For the purposes of expression evaluation an option with flavor `none` always has the value 1. Normal `#define` processing will take place, so typically a single `#define` will be generated using the option name and a value of 1. Similarly build-related properties such as `compile` will take effect.

**flavor bool**

Boolean options can be either enabled or disabled, and there is no additional data associated with them. If a boolean option is disabled then no `#define` will be generated and any build-related properties such as `compile` will be ignored. For the purposes of expression evaluation a disabled option has the value 0. If a boolean option is enabled then normal `#define` processing will take place, all build-related properties take effect, and the option's value will be 1.

**flavor data**

Options with this flavor are always enabled, and have some additional data associated with them which can be edited by the user. This data can be any sequence of characters, although in practice the `legal_values` property will often be used to impose constraints. In appropriate contexts such as expressions the configuration tools will attempt to interpret the data as integer or floating point numbers. Since an option with the `data` flavor cannot be disabled, normal `#define` processing takes place and the data will be used for the value. Similarly all build-related properties take effect, and the option's value for the purposes of expression evaluation is the data.

**flavor booldata**

This combines the `bool` and `data` flavors. The option may be enabled or disabled, and in addition the option has some associated data. If the option is disabled then no `#define` will be generated, the build-related properties have no effect, and for the purposes of expression evaluation the option's value is 0. If the option is enabled then a `#define` will be generated using the data as the value, all build-related properties take effect, and the option's value for the purposes of expression evaluation is the data. If 0 is legal data then it is not possible to distinguish this case from the option being disabled or inactive.

Options and components have the `bool` flavor by default, but this can be changed as desired. Packages always have the `booldata` flavor, and this cannot be changed. Interfaces have the `data` flavor by default, since the value of an interface is a count of the number of active and enabled interfaces, but they can be given the `bool` or `booldata` flavors.

**Note:** The expression syntax needs to be extended to allow the loaded, active, enabled and data aspects of an option's state to be examined individually. This would allow component writers to distinguish between a disabled `booldata` option and an enabled one which has a value of 0. Such an enhancement to the expression syntax may also prove useful in other circumstances.

## Example

```
cdl_component CYGPKG_LIBM_COMPATIBILITY {
```

```
cdl_component CYGNUM_LIBM_COMPATIBILITY {
    flavor booldata
    ...

    cdl_option CYGNUM_LIBM_COMPAT_DEFAULT {
        flavor data
        ...
    }
}

...
}

cdl_component CYGPKG_LIBM_TRACE {
    flavor      bool
    ...
}
```

## See Also

Properties [calculated](#), [default\\_value](#) and [legal\\_values](#),

*flavor*

# hardware

## Name

Property `hardware` — Specify that a package is tied to specific hardware.

## Synopsis

```
cdl_option <name> {  
    active_if <condition>  
    ...  
}
```

## Description

Some packages such as device drivers and HAL packages are hardware-specific, and generally it makes no sense to add such packages to a configuration unless the corresponding hardware is present on your target system. Typically hardware package selection happens automatically when you select your target. The hardware property can be used in the body of a `cdl_package` command to indicate that the package is hardware-specific.

**Note:** At the time of writing the hardware property is largely ignored by the configuration tools, but this may change in future. Amongst other possible changes, for hardware-specific packages such as device drivers and HAL packages, the current scheme of generating a configuration header file name based purely on the package name may be abandoned. Instead all hardware packages would send their configuration data to a single header file, `pkgconf/hardware.h`. This would make it easier for code to obtain details of the current hardware, but obviously there are compatibility issues. For now it is recommended that all hardware packages specify their configuration header file explicitly.

## Example

```
cdl_package CYGPKG_HAL_ARM {  
    display      "ARM architecture"  
    parent      CYGPKG_HAL  
    hardware  
    include_dir  cyg/hal  
    define_header hal_arm.h  
    ...  
}
```

## See Also

Property [define\\_header](#), and command `cdl_package`.

*hardware*

# if\_define

## Name

Property `if_define` — Output a common preprocessor construct to a configuration header file.

## Synopsis

```
cdl_option <name> {  
    if_define [-file=<filename>] <symbol1> <symbol2>  
    ...  
}
```

## Description

The purpose of the `if_define` property is best explained by an example. Suppose you want finer-grained control over assertions, say on a per-package or even a per-file basis rather than globally. The assertion macros can be defined by an exported header file in an infrastructure package, using code like the following:

```
#ifndef CYGDBG_USE_ASSERTS  
# define CYG_ASSERT( _bool_, _msg_ )    \  
    CYG_MACRO_START                    \  
    if ( ! ( _bool_ ) )                \  
        CYG_ASSERT_DOCALL( _msg_ ); \  
    CYG_MACRO_END  
#else  
# define CYG_ASSERT( _bool_, _msg_ ) CYG_EMPTY_STATEMENT  
#endif
```

Assuming this header file is `#include`'d directly or indirectly by any code which may need to be built with assertions enabled, the challenge is now to control whether or not `CYGDBG_USE_ASSERTS` is defined for any given source file. This is the purpose of the `if_define` property:

```
cdl_option CYGDBG_KERNEL_USE_ASSERTS {  
    ...  
    if_define CYGSRG_KERNEL CYGDBG_USE_ASSERTS  
    requires CYGDBG_INFRA_ASSERTION_SUPPORT  
}
```

If this option is active and enabled then the kernel's configuration header file would end up containing the following:

```
#ifndef CYGSRG_KERNEL  
# define CYGDBG_USE_ASSERTS 1  
#endif
```

Kernel source code can now begin with the following construct:

```
#define CYGSRG_KERNEL 1  
#include <pkgconf/kernel.h>
```

## *if\_define*

```
#include <cyg/infra/cyg_ass.h>
```

The configuration option only affects kernel source code, assuming nothing else `#define`'s the symbol `CYGSRC_KERNEL`. If the per-package assertion option is disabled then `CYGDBG_USE_ASSERTS` will not get defined. If the option is enabled then `CYGDBG_USE_ASSERTS` will get defined and assertions will be enabled for the kernel sources. It is possible to use the same mechanism for other facilities such as tracing, and to apply it at a finer grain such as individual source files by having multiple options with `if_define` properties and multiple symbols such as `CYGSRC_KERNEL_SCHED_BITMAP_CXX`.

The `if_define` property takes two arguments, both of which must be valid C preprocessor symbols. If the current option is active and enabled then three lines will be output to the configuration header file:

```
#ifdef <symbol1>
# define <symbol2>
#endif
```

If the option is inactive or disabled then these lines will not be output. By default the current package's configuration header file will be used, but it is possible to specify an alternative destination using a `-file` option. At present the only legitimate alternative destination is `system.h`, the global configuration header. `if_define` processing happens in addition to, not instead of, the normal `#define` processing or the handling of other header-file related properties.

**Note:** The infrastructure in the current eCos release does not yet work this way. In future it may do so, and the intention is that suitable configuration options get generated semi-automatically by the configuration system rather than having to be defined explicitly.

**Tip:** As an alternative to changing the configuration, updating the build tree, and so on, it is possible to enable assertions by editing a source file directly, for example:

```
#define CYGSRC_KERNEL 1
#define CYGDBG_USE_ASSERTS 1
#include <pkgconf/kernel.h>
#include <cyg/infra/cyg_ass.h>
```

The assertion header file does not care whether `CYGDBG_USE_ASSERTS` is `#define`'d via a configuration option or by explicit code. This technique can be useful to component writers when debugging their source code, although care has to be taken to remove any such `#define`'s later on.

## Example

```
cdl_option CYGDBG_KERNEL_USE_ASSERTS {
    display "Assertions in the kernel package"
    ...
    if_define CYGSRC_KERNEL CYGDBG_USE_ASSERTS
    requires CYGDBG_INFRA_ASSERTION_SUPPORT
}
```



## See Also

Properties [define](#), [define\\_format](#), [define\\_header](#), [define\\_proc](#) and [no\\_define](#).

*if\_define*

# implements

## Name

Property `implements` — Enabling this option provides one instance of a more general interface.

## Synopsis

```
cdl_option <name> {  
    implements <interface>  
    ...  
}
```

## Description

The CDL interface concept provides an abstraction mechanism that can be useful in many different circumstances. Essentially an interface is a calculated option whose value is the number of active and enabled options which implement that interface. For example the interface `CYGINT_KERNEL_SCHEDULER` has a value corresponding to the number of schedulers in the system, typically just one.

The `implements` property takes a single argument, which should be the name of an interface. This interface may be defined in the same package as the implementor or in some other package. In the latter case it may sometimes be appropriate for the implementor or the implementor's package to have a `requires` property for the package containing the interface. An option may contain multiple `implements` properties. It is possible for an option to implement a given interface multiple times, and on occasion this is actually useful.

## Example

```
cdl_option CYGSEM_KERNEL_SCHED_MLQUEUE {  
    display      "Multi-level queue scheduler"  
    default_value 1  
    implements   CYGINT_KERNEL_SCHEDULER  
    ...  
}
```

## See Also

Command [cdl\\_interface](#).

*implements*

# include\_dir

## Name

Property `include_dir` — Specify the desired location of a package's exported header files in the install tree.

## Synopsis

```
cdl_package <name> {  
    include_dir <sub-directory>  
    ...  
}
```

## Description

Most packages export one or more header files defining their public interface. For example the C library exports header files such as `stdio.h` and `ctype.h`. If the package follows the [directory layout conventions](#) then the exported header files will normally be found in the package's `include` sub-directory. Alternatively the `include_files` property can be used to specify which header files should be exported.

By default a package's exported header files will be copied to the `include` sub-directory of the install tree. This is correct for packages like the C library because that is the correct location for files such as `stdio.h`. However to reduce the probability of name clashes it is desirable for packages to use different sub-directories, for example infrastructure header files get copied to `include/cyg/infra` rather than to the top-level `include` directory itself.

It would be possible to replicate these sub-directories in each package's source tree, such that the infrastructure header file sources lived in `include/cyg/infra` in the source tree as well as in the install tree. This would make things more difficult for the package developers. Instead it is possible to specify the desired install tree sub-directory using an `include_dir` property, for example `include_dir cyg/infra`.

The `include_dir` property can only be used in the body of a `cdl_package` command, since it applies to all of the header files exported by a package, and only one `include_dir` property can be used. If there is no `include_dir` property then exported header files will end up in the top-level `include` directory of the install tree.

## Example

```
cdl_package CYGPKG_INFRA {  
    display      "Infrastructure"  
    include_dir  cyg/infra  
    description  "  
        Common types and useful macros.  
        Tracing and assertion facilities.  
        Package startup options."  
    ...  
}
```

*include\_dir*

## See Also

Property [include\\_files](#), and command [cdl\\_package](#).

# include\_files

## Name

Property `include_files` — List the header files that are exported by a package.

## Synopsis

```
cdl_package <name> {  
    include_files <file1> ...  
    ...  
}
```

## Description

Most packages export one or more header files defining their public interface. For example the C library exports header files such as `stdio.h` and `ctype.h`. If the package follows the [directory layout conventions](#) then the exported header files will normally be found in the package's `include` sub-directory. For packages which do not follow these conventions, typically simple ones for which a complicated sub-directory hierarchy is undesirable, there has to be an alternative way of specifying which header file or files define the public interface. The `include_files` property provides support for this.

By default, if a package does not have an `include` subdirectory and it does not have an `include_files` property then all files with a suffix of `.h`, `.hxx`, `.inl` or `.inc` will be treated as public header files. However some of these may be private files containing implementation details. If there is an `include_files` property then only the files listed in that property will be exported.

If a package should not export any header files but does contain private implementation headers, an `include_files` property with no arguments should be used.

## Example

```
cdl_package <SOME_PACKAGE> {  
    ...  
    include_dir    <some directory>  
    include_files  interface.h  
}  
  
cdl_package <ANOTHER_PACKAGE> {  
    ...  
    include_files  
}
```

*include\_files*

## See Also

Property [include\\_dir](#), and command [cdl\\_package](#).



# legal\_values

## Name

Property `legal_values` — Impose constraints on the possible values for an option.

## Synopsis

```
cdl_option <name> {  
    legal_values <list expression>  
    ...  
}
```

## Description

Options with the `data` or `booldata` flavors can have an arbitrary sequence of characters as their data. In nearly all cases some restrictions have to be imposed, for example the data should correspond to a number within a certain range, or it should be one of a small number of constants. The `legal_values` property can be used to impose such constraints. The arguments to the property should be a CDL list expression, see [the Section called \*List Expressions\* in Chapter 3](#) for the syntactic details. Common examples include:

```
legal_values 0 to 0x7fff  
legal_values 9600 19200 38400  
legal_values { "RAM" "ROM" }
```

The `legal_values` property can only be used for options with the `data` or `booldata` flavors, since it makes little sense to further constrain the legal values of a boolean option. An option can have at most one `legal_values` property.

**Tip:** If the first entry in a `legal_values` list expression is a negative number, for example `legal_values -1 to 1` then this can be misinterpreted as an option instead of as part of the expression. Currently the `legal_values` property does not take any options, but this may change in future. Option processing halts at the sequence `--`, so the desired range can be expressed safely using `legal_values -- -1 to 1`

**Note:** Architectural HAL packages should provide constants which can be used in `legal_values` list expressions. For example it should be possible to specify a numeric range such as `0 to CYGARC_MAXINT`, rather than hard-wiring numbers such as `0x7fffffff` which may not be valid on all targets. Current HAL packages do not define such constants.

**Note:** The `legal_values` property is restricted mainly to numerical ranges and simple enumerations, and cannot cope with more complicated data items. Future versions of the configuration system will provide additional data validation facilities, for example a `check_proc` property which specifies a Tcl script that can be used to perform the validation.

*legal\_values*

## Example

```
cdl_option CYGNUM_LIBC_TIME_STD_DEFAULT_OFFSET {
  display      "Default Standard Time offset"
  flavor       data
  legal_values -- -90000 to 90000
  default_value -- 0
  description  "
    This option controls the offset from UTC in
    seconds when in local Standard Time. This
    value can be positive or negative. It
    can also be set at run time using the
    cyg_libc_time_setzoneoffsets() function."
}
```

## See Also

Properties [calculated](#), [default\\_value](#), and [flavor](#).

# library

## Name

Property `library` — Specify which library should contain the object files generated by building this package.

## Synopsis

```
cdl_package <name> {  
    library <library name>  
    ...  
}
```

## Description

By default all object files that get built for all packages end up in a single library, `libtarget.a`. This makes things easier for the typical application developer because it is only necessary to link with a single library, rather than with separate libraries for each package. It is possible to specify an alternative library for specific files as an option to the `compile` and `make_object` properties, and there is one library `libextras.a` which serves a specific purpose in the build system. The `library` property allows an alternative library to be specified for all the object files that will be generated for a given package.

The use of the `library` property should be avoided, since it makes things more difficult for application developers. The property is intended only for special cases, for example if there are legal objections to mingling object files from different packages in a single library. It could also be used to work around name clash problems if two packages happen to define an exported symbol with the same name, but any attempt to use multiple libraries in this way is error-prone and should be avoided.

The `library` property takes a single argument, the name of a library, which should follow the standard naming convention of `lib<something>.a`. A `library` property can only occur in the body of a `cdl_package` command and applies to all object files generated for that package (except where explicitly overwritten with a `-library=` option to one of the build-related properties). A `cdl_package` body can contain at most one `library` property.

## Example

```
cdl_package <SOME_PACKAGE> {  
    ...  
    library libSomePackage.a  
}
```

## See Also

Properties `compile`, `make`, and `make_object`, command `cdl_package`.

*library*

# make

## Name

Property `make` — Define an additional custom build step associated with an option, resulting in a target that should not go directly into a library.

## Synopsis

```
cdl_option <name> {  
    make [-priority=<pri>] {  
        <custom build step>  
    }  
    ...  
}
```

## Description

When building an eCos configuration the primary target is a single library, `libtarget.a`. In some cases it is necessary to build some additional targets. For example architectural HAL packages typically build a linker script and some start-up code. Such additional targets can be specified by a `make` property. Any option can have one or more `make` properties, although typically such properties only occur in the body of a `cdl_package` command.

The `make` property takes a single argument, which resembles a makefile rule: it consists of a target, a list of dependencies, and one or more commands that should be executed. However the argument is not a makefile fragment, and custom build steps may get executed in build environments that do not involve `make`. For full details of custom build steps see [the Section called \*Custom Build Steps\* in Chapter 4](#).

### Warning

The exact syntax and limitations of custom build steps have not yet been finalized, and are subject to change in future versions of the configuration tools.

The `make` property takes an optional priority argument indicating the order in which build steps take place. This priority complements the dependency list, and avoids problems with packages needing to know details of custom build steps in other packages (which may change between releases). The defined order is:

#### Priority 0

The header files exported by the current set of packages are copied to the appropriate places in the `include` subdirectory of the install tree. Any unnecessary copies are avoided, to prevent rebuilds of package and application source modules caused by header file dependencies.

**Note:** A possible future enhancement of the build system may result in the build and install trees being updated automatically if there has been a change to the `ecos.ecc` configuration savefile.

### Priority 100

All files specified in compile properties will get built, producing the corresponding object files. In addition any custom build steps defined by `make_object` properties get executed, unless there is a `-priority=` option.

### Priority 200

The libraries now get built using the appropriate object files.

### Priority 300

Any custom build steps specified by make properties now get executed, unless the priority for a particular build step is changed from its default.

For example, if a custom build step needs to take place before any of the normal source files get compiled then it should be given a priority somewhere between 0 and 100. If a custom build step involves post-processing an object file prior to its incorporation into a library then a priority between 100 and 200 should be used.

## Example

```
cdl_package CYGPKG_HAL_MN10300_AM33 {
    display      "MN10300 AM33 variant"
    parent       CYGPKG_HAL_MN10300
    implements   CYGINT_HAL_MN10300_VARIANT
    hardware
    include_dir   cyg/hal
    define_header hal_mn10300_am33.h
    description   "
        The MN10300 AM33 variant HAL package provides generic
        support for this processor architecture. It is also
        necessary to select a specific target platform HAL
        package."

    make {
        <PREFIX>/lib/target.ld: <PACKAGE>/src/mn10300_am33.ld
        $(CC) -E -P -Wp,-MD,target.tmp -DEXTRAS=1 -xc $(INCLUDE_PATH) $(CFLAGS) -o $$ $<
        @echo $$ ": \\" > $(notdir $@).deps
        @tail +2 target.tmp >> $(notdir $@).deps
        @echo >> $(notdir $@).deps
        @rm target.tmp
    }
}
```

## See Also

Properties [compile](#), [make\\_object](#) and [library](#).

# make\_object

## Name

Property `make_object` — Define a custom build step, resulting in an object file that should go into a library.

## Synopsis

```
cdl_option <name> {  
    make_object [-library=<library>] [-priority=<pri>] {  
        <custom build step>  
    }  
    ...  
}
```

## Description

When building an eCos configuration the primary target is a single library, `libtarget.a`. Most of the object files which go into this library will be generated as a result of compile properties. Occasionally it may be necessary to have special build steps for a given object file, and this can be achieved with a `make_object` property. The use of this property should be avoided whenever possible because it greatly increases the risk of portability problems, both on the host side because of possible problems with the tools, and on the target side because a custom build step may not allow adequately for the wide variety of architectures supported by eCos.

The `make_object` property takes a single argument, which resembles a makefile rule: it consists of a target, a list of dependencies, and one or more commands that should be executed. The target should be an object file. However the `make_object` argument is not a makefile fragment, and custom build steps may get executed in build environments that do not involve make. For full details of custom build steps see [the Section called \*Custom Build Steps\* in Chapter 4](#).

### Warning

The exact syntax and limitations of custom build steps have not yet been finalized, and may change in future versions of the configuration tools.

The `make_object` property takes an optional library argument. If no library is specified then the default library for the current package will be used, which will be `libtarget.a` unless the `cdl_package` command contains a library property.

The `make_object` property also takes an optional priority argument indicating the order in which build steps take place. This priority complements the dependency list, and avoids problems with packages needing to know details of custom build steps in other packages (which may change between releases). The defined order is:

#### Priority 0

The header files exported by the current set of packages are copied to the appropriate places in the `include` subdirectory of the install tree. Any unnecessary copies are avoided, to prevent rebuilds of package and application source modules caused by header file dependencies.

**Note:** A possible future enhancement of the build system may result in the build and install trees being updated automatically if there has been a change to the `ecos.ecc` configuration savefile.

#### Priority 100

All files specified in `compile` properties will get built, producing the corresponding object files. In addition any custom build steps defined by `make_object` properties get executed, unless there is a `-priority=` option.

#### Priority 200

The libraries now get built using the appropriate object files.

#### Priority 300

Any custom build steps specified by `make` properties now get executed, unless the priority for a particular build step is changed from its default.

For example, if a custom build step needs to take place before any of the normal source files get compiled then it should be given a priority somewhere between 0 and 100. If a custom build step involves post-processing an object file prior to its incorporation into a library then a priority between 100 and 200 should be used. It is not sensible to have a priority above 200, since that would imply building an additional object file for a library that has already been created.

## Example

```
cdl_option XXX {
    ...
    make_object {
        parser.o: parser.y
                yacc $<
                $(CC) $(CFLAGS) -o $@ y.tab.c
    }
}
```

## See Also

Properties [compile](#), [make](#) and [library](#).



# no\_define

## Name

Property `no_define` — Suppress the normal generation of a preprocessor `#define` symbol in a configuration header file.

## Synopsis

```
cdl_option <name> {  
    no_define  
    ...  
}
```

## Description

By default all active and enabled properties result in either one or two `#define`'d symbols in the package's configuration header file, and this is one of the main ways in which options can affect packages at build-time. It is possible to suppress the default `#define`'s by specifying a `no_define` property in the body of an option or other CDL entity. This property takes no arguments and should occur only once in a given body.

The `no_define` property is frequently used in conjunction with one of the other header-file related properties such as `define`. If one of the other properties is used to export the required information to a configuration header file then often there is little point in exporting the default `#define` as well — in fact there could be a name clash. The `no_define` property can also be useful if the sole purpose of an option is to affect which files get built, and the default `#define` would never get tested in any source code. However in such cases the default `#define` is mostly harmless and there is little to be gained by suppressing it.

## Example

```
cdl_component CYG_HAL_STARTUP {  
    display      "Startup type"  
    flavor       data  
    legal_values { "RAM" "ROM" }  
    default_value {"RAM"}  
    no_define  
    define -file system.h CYG_HAL_STARTUP  
    ...  
}
```

## See Also

Properties [define](#), [define\\_format](#), [define\\_header](#), [define\\_proc](#) and [if\\_define](#).

*no\_define*

# parent

## Name

Property `parent` — Control the location of an option in the configuration hierarchy.

## Synopsis

```
cdl_option <name> {  
    parent <component or package>  
    ...  
}
```

## Description

Configuration options live in a hierarchy of packages and components. By default a given option's position in the hierarchy is a simple consequence of its position within the CDL scripts. Packages are generally placed at the top-level of the configuration. Any components or options that are defined at the same level as the `cdl_package` command in a package's top-level CDL script are placed immediately below that package in the hierarchy. Any options or components that are defined in the body of a `cdl_package` or `cdl_component` command, or that are read in as a result of processing a component's script property, will be placed immediately below that package or component in the hierarchy.

In some circumstances it is useful to specify an alternative position in the hierarchy for a given option. For example it is often convenient to re-parent device driver packages below `CYGPKG_IO` in the configuration hierarchy, thus reducing the number of packages at the top level of the hierarchy and making navigation easier. The `parent` property can be used to achieve this.

The `parent` property takes a single argument, which should be the name of a package or component. The body of a `cdl_option` or other CDL entity can contain at most one `parent` property.

Although the `parent` property affects an option's position in the overall hierarchy and hence whether or not that option is active, a re-parented option still belongs to the package that defines it. By default any `#define`'s will be exported to that package's configuration header file. Any compile properties can only reference source files present in that package, and it is not directly possible to cause some file in another package to be built by re-parenting.

As a special case, if an empty string is specified for the `parent` then the option is placed at the top of the hierarchy, ahead of any packages which are not explicitly re-parented in this way. This facility is useful for configuration options such as global preferences and default compiler flags.

**Tip:** If an option is re-parented somewhere below another package and that other package is not actually loaded, the option is an orphan and its active/inactive state is undefined. In such cases it is a good idea for the owning package to require the presence of the other one. Unfortunately this technique does not work if a package as a whole is re-parented below another one that has not been loaded: the package is orphaned so it may be automatically inactive, and hence any requires properties would have no effect.

*parent*

## Example

```
cdl_package CYGPKG_HAL_I386 {
    display      "i386 architecture"
    parent      CYGPKG_HAL
    hardware
    include_dir  cyg/hal
    define_header hal_i386.h
    ...
}

cdl_component CYGBLD_GLOBAL_OPTIONS {
    display      "Global build options"
    parent      " "
    ...
}
```

## See Also

Property [script](#), commands [cdl\\_component](#) and [cdl\\_package](#).

# requires

## Name

Property `requires` — List constraints that the configuration should satisfy if a given option is active and enabled..

## Synopsis

```
cdl_option <name> {  
    requires <goal expression>  
    ...  
}
```

## Description

Configuration options are not independent. For example the C library can provide thread-safe implementations of certain functions, but only if the kernel is present, if the kernel provides multi-threading, and if the kernel options related to per-thread data are enabled. It is possible to express such constraints using `requires` properties.

The arguments to a `requires` property should constitute a goal expression, as described in [the Section called \*List Expressions\* in Chapter 3](#). Most goal expressions are relatively simple because the constraints being described are simple, but complicated expressions can be used when necessary. The body of an option or other CDL entity can contain any number of `requires` constraints. If the option is active and enabled then all these constraints should be satisfied, and any goal expressions which evaluate to 0 will result in conflicts being raised. It is possible for users to ignore such conflicts and attempt to build the current configuration anyway, but there is no guarantee that anything will work. If an option is inactive or disabled then its `requires` constraints will be ignored.

The configuration system contains an inference engine which can resolve many types of conflicts automatically. For example, if option `A` is enabled and requires an option `B` that is currently disabled then the inference engine may attempt to resolve the conflict by enabling `B`. However this will not always be possible, for example there may be other constraints in the configuration which force `B` to be disabled at present, in which case user intervention is required.

## Example

```
cdl_component CYGPKG_IO_SERIAL_POWERPC_COGENT_SERIAL_A {  
    display      "Cogent PowerPC serial port A driver"  
    flavor       bool  
    default_value 0  
    requires     (CYGIMP_KERNEL_INTERRUPTS_CHAIN || \  
                 !CYGPKG_IO_SERIAL_POWERPC_COGENT_SERIAL_B)  
    ...  
}
```

*requires*

## **See Also**

Property [active\\_if](#).

# script

## Name

Property `script` — Include additional configuration information from another CDL script.

## Synopsis

```
cdl_component <name> {  
    script <filename>  
    ...  
}
```

## Description

It is possible to define all the configuration options and sub-components for a given package in a single CDL script, either by nesting them in the appropriate command bodies, by extensive use of the parent property, or by some combination of these two. However for large packages this is inconvenient and it is better to split the raw configuration data over several different files. The `script` property can be used to achieve this. It takes a single filename as argument. If the package follows the [directory layout conventions](#) then the configuration tools will look for the specified file in the `cdl` sub-directory of the package, otherwise it will look for the file relative to the package's top-level directory.

The `script` property can only occur in the body of a `cdl_component` command, and only one `script` property per body is allowed.

## Example

```
cdl_component CYGPKG_UITRON_TASKS {  
    display      "Tasks"  
    flavor       none  
    description   "  
        uITRON Tasks are the basic blocks of multi-tasking  
        in the uITRON world; they are threads or lightweight  
        processes, sharing the address space and the CPU.  
        They communicate using the primitives outlined above.  
        Each has a stack, an entry point (a C or C++ function),  
        and (where appropriate) a scheduling priority."  
  
    script        tasks.cdl  
}
```

*script*

## See Also

Command [cdl\\_component](#), and property [parent](#).



# Chapter 6. Templates, Targets and Other Topics

## Templates

This section is still under construction.

## Targets

This section is still under construction.

