eCos™ User's Guide

August 2000

Copyright © 1998, 1999, 2000 Red Hat Inc.

Copying terms

Trademarks

Java™, Sun®, and Solaris™ are trademarks or registered trademarks of Sun Microsystems, Inc.

SPARC® is a registered trademark of SPARC International, Inc.

UNIX™ is a trademark of The Open Group.

Microsoft®, Windows NT®, Windows 95®, Windows 98® and Windows 2000® are registered trademarks of Microsoft Corporation.

Linux® is a registered trademark of Linus Torvalds.

Intel® is a registered trademark of Intel Corporation.

eCos™ is a trademark of Red Hat, Inc.

Red Hat® is a registered trademark of Red Hat, Inc.

300-400-1010049-03

Contents

Part I: The eCos Configuration Tool

1

Getting Started

Introduction

The **eCos Configuration Tool** is used to tailor eCos at source level, prior to compilation or assembly, and provides a configuration file and a set of files used to build user applications. The sources and other files used for building a configuration are provided in a *component repository*, which is loaded when the eCos Configuration Tool is invoked. The component repository includes a set of files defining the structure of relationships between the Configuration Tool and other components, and is written in a *Component Definition Language* (CDL). For a description of the concepts underlying component configuration, see "CDL Concepts" on page 45.

Invoking the eCos Configuration Tool

There are two ways in which to invoke the eCos Configuration Tool:

- from the desktop explorer or program set up at installation time (by default **Start**->**Programs**->**Red Hat eCos**->**Configuration Tool**).
- type (at a command prompt or in the **Start** menu's **Run** item): <foldername>\ConfigTool.exe where <foldername> is the full path of the directory in which you installed the eCos Configuration Tool**.**
  - ❑ The **Configuration Tool** will be displayed (see Figure 1).

Figure 1: **Configuration Tool**
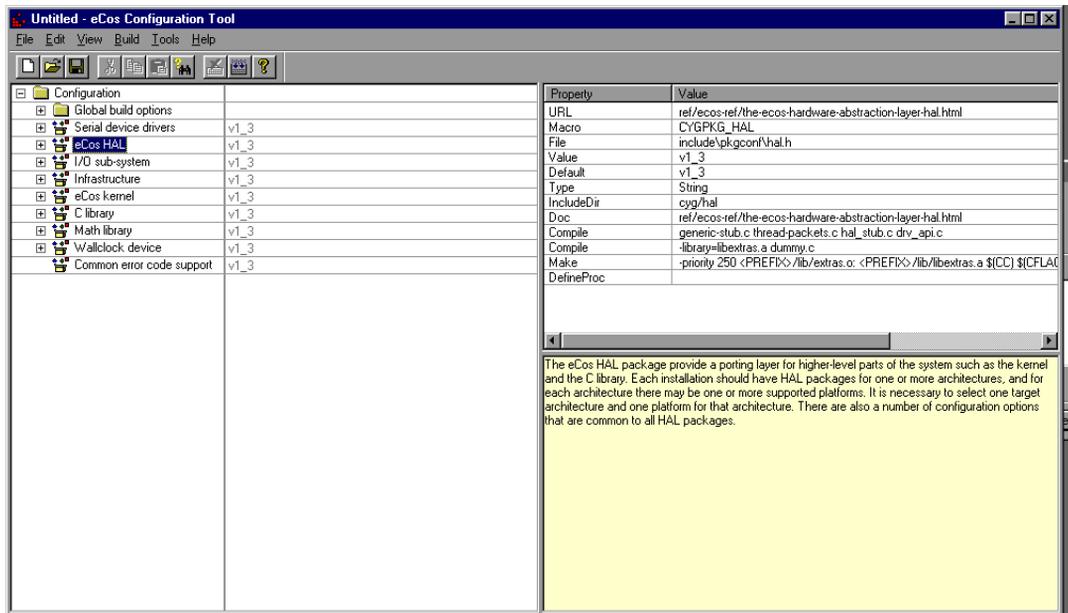
The Component Repository

When you invoke the eCos Configuration Tool, it accesses the *Component Repository*, a read-only location of configuration information. For an explanation of "Component Repository" see "CDL Concepts" on page 45.

The eCos Configuration Tool will look for a component repository using (in descending order of preference):

- The component repository most recently used by the current user
- A default location set by the installation procedure
- User input

The final case above will normally only occur if the previous repository has been moved or installation information stored in the NT registry has been modified; it will result in a dialog box being displayed that allows you to specify the repository location:

Figure 2: Repository relocation dialog box
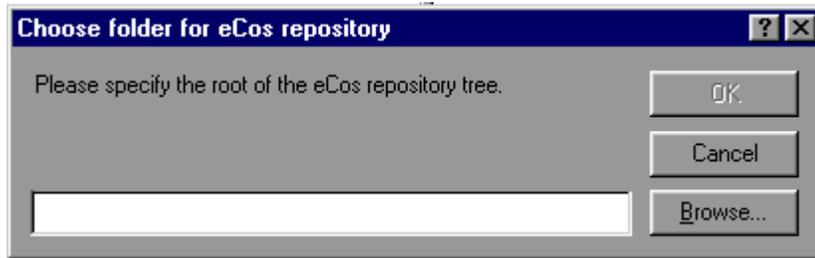
Note that in order to use the eCos Configuration Tool you are obliged to provide a valid repository location.

In the rare event that you subsequently wish to change the component location, select **Build**->**Repository** and the above dialog box will then be displayed.

eCos Configuration Tool Documents

Configuration Save File

eCos configuration settings and other information (such as disabled conflicts) that are set using the eCos Configuration Tool are saved to a file between sessions. By default, when the eCos Configuration Tool is first invoked, it reads and displays information from the Component Registry and displays the information in an untitled blank document. You can perform the following operations on a document:

Save the currently active document

Use the "**File->Save**" menu item or click the **Save Document** icon on the toolbar; if the current document is unnamed, you will be prompted to supply a name for the configuration save file.

Figure 3: **Save As dialog box**
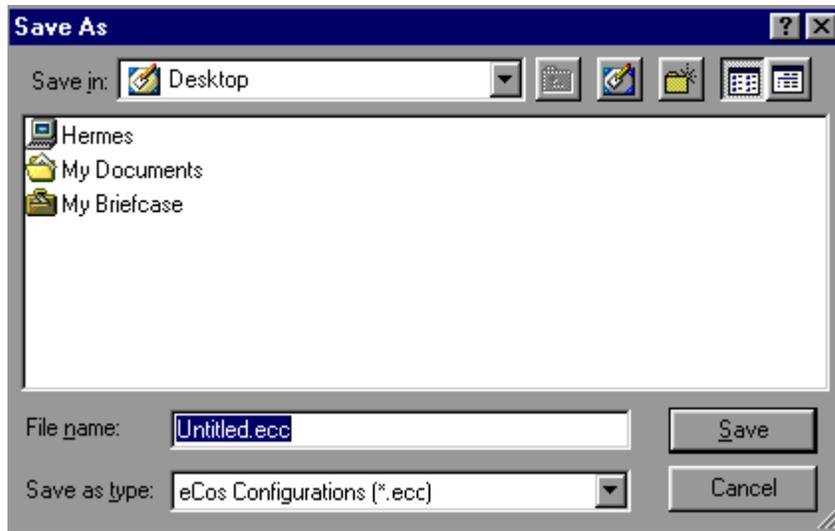
Open an existing document

Select **File**->**Open**, or click the **Open Document** icon on the toolbar. You will be prompted to supply a name for the configuration save file.

Open dialog box
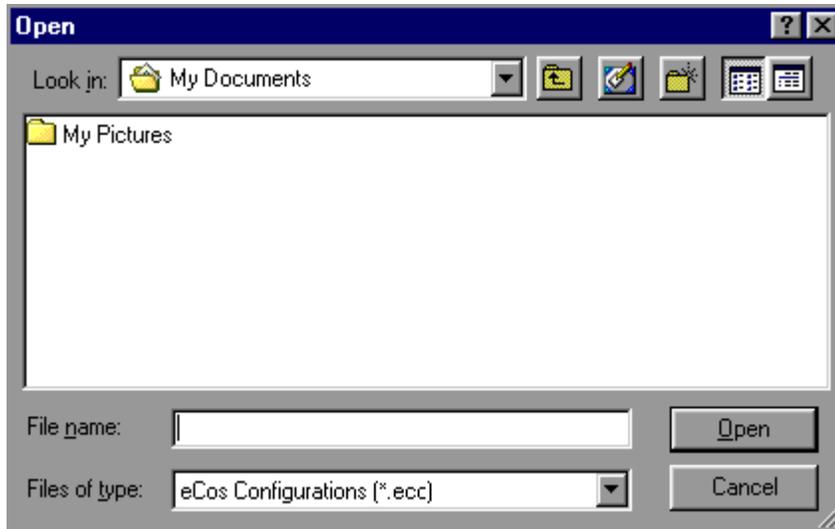
Open a document you have used recently

Click its name at the bottom of the **File** menu.

Documents may also be opened by:

- dragging and dropping a Configuration Save File from the desktop explorer to the eCos Configuration Tool
- double-clicking a Configuration Save File in the desktop explorer
- invoke the eCos Configuration Tool with the name of a Configuration File as command-line argument, or by creating a shortcut to the eCos Configuration Tool with such an argument.

Create a new blank document based on the Component Registry

Select **File**->**New**, or click the **New Document** icon on the toolbar.

Save to a different file name

Select **File**->**Save As**. You will be prompted to supply a new name for the configuration save file.

Build and Install Trees

The location of the build and install trees are derived from the eCos save file name as illustrated in the following example:

Save file name = "c:\My eCos\config1.ecc"

Install tree folder = "c:\My eCos\config1_install"

Build tree folder = "c:\My eCos\config1_build"

These names are automatically generated from the name of the save file.

See also "CDL Concepts" on page 45.

2

Getting Help

The eCos Configuration Tool contains several methods for accessing online help.

Context-sensitive Help for Dialogs

Most dialogs displayed by the eCos Configuration Tool are supplied with context-sensitive help. You can then get help relating to any control within the current dialog box by

- Right-clicking the control (or pressing **F1**)

  A "What's This?" popup menu will be displayed. Click the menu to display a brief description of the function of the selected control.

- Clicking the help (question mark) icon in the dialog caption bar

  A question mark cursor will be displayed. Click on any control to display a brief description of its function.

Some dialogs may have a **Help** button. You can press this to display a more general description of the function of the dialog box as a whole. This help will be in HTML form; for more information, see below.

Context-sensitive Help for Other Windows

In the **Help** menu, click **eCos Configuration Tool Help** (or press **F1**). A HTML page describing the general operation of the currently active window will be displayed. This help will normally be in HTML format; for more information, see "Methods of Displaying HTML Help".

Context-sensitive Help for Configuration Items

In the configuration window, right-click on a configuration item (or use **Shift+F10**). A context menu will be displayed; select **Visit Documentation** to display the page in the eCos documentation that most closely corresponds to the selected item.

Methods of Displaying HTML Help

By default, help in HTML form is displayed using an *HTML Help* viewer built in to the eCos Configuration Tool. This form of help will be familiar to Windows 98 or Windows 2000 users: it takes the form of a 3-pane floating window comprising **Toolbar**, **Navigation** and **Topic** windows. The **Navigation Window** provides access to Table of Contents (TOC), Index, and Search facilities. A toolbar is provided to allow quick access to related internet sites, including the **Red Hat** home page and net distribution sites.

Figure 5: **HTML Help viewer**
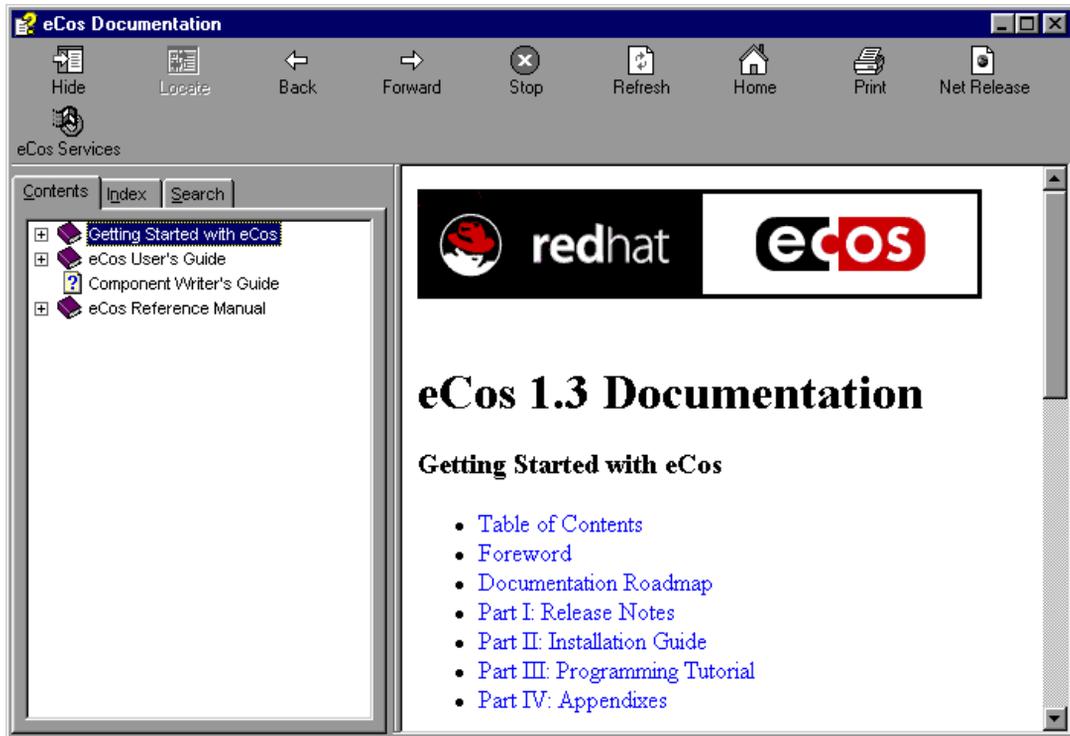
If you wish, you may choose to have *HTML Help* displayed in a browser of your choice. To do this, select **View**->**Settings** and use the controls in the View Documentation group to select the replacement browser. Note that the Navigation facilities of the built-in *HTML Help* system will be unavailable if you choose this method of displaying help.

3

Customization

The following visual aspects of the eCos Configuration Tool can be changed to suit individual preferences. These aspects are saved on a per-user basis, so that when the eCos Configuration Tool is next invoked by the same user, the appearance will be as set in the previous session.

Window Placement

The relative sizes of all windows in the eCos Configuration Tool may be adjusted by dragging the splitter bars that separate the windows. The chosen sizes will be used the next time the eCos Configuration Tool is invoked by the current user.

All windows except the **Configuration Window** may be shown or hidden by using

the commands under the **View** menu (for example, **View->Memory Layout**) or the corresponding keyboard accelerators (**Alt+1** to **Alt+5**). By default the memory layout and conflicts window are hidden.

Your chosen set of windows (and their relative sizes) will be preserved between invocations of the eCos Configuration Tool.

Toolbars

Select **View**->**Toolbars**: each of the standard and Memory Layout toolbars may be hidden or shown.

Settings

To change other visual aspects, select **View**->**Settings**. The **Settings** dialog box will be displayed that allows you to customize the following options:

Displaying Header Files

You can change the viewer used to display header files.

View Documentation

You can change the browser used to display *HTML Help* (see "Methods of Displaying HTML Help" on page 9).

Configuration Item Labels

In the configuration window, you can choose to have either *descriptive names* (the default) or *macro names* displayed as tree item labels. Descriptive names are generally more comprehensible, but macro names are used in some contexts such as conflict resolution and may be directly related to the source code of the configuration. Note that it is possible to search for an item in the configuration view by selecting **Find**->**Edit** (see "Searching" on page 31). Both descriptive names and macro names can be searched.

Configuration Item Integer Format

You can choose to have integer items in the Configuration Window displayed in decimal or hexadecimal format.

Fonts

The font used in each window of the eCos Configuration Tool may be changed independently. To use this feature, select the window whose font is to be changed in the drop-list labeled "Window" and press the "**Change Font**" button.

Settings

View header files using
○ Associated viewer
● This viewer: Browse...

[   ]

View documentation using
● Built-in viewer
○ Associated browser (netscape.exe)
○ This browser: Browse...

[   ]

Configuration Pane

Labels
○ Use macro names   ● Use descriptive names

Integer items
● Decimal   ○ Hexadecimal

Font
Window: [Configuration ▼]   Change Font...

[OK]   [Cancel]
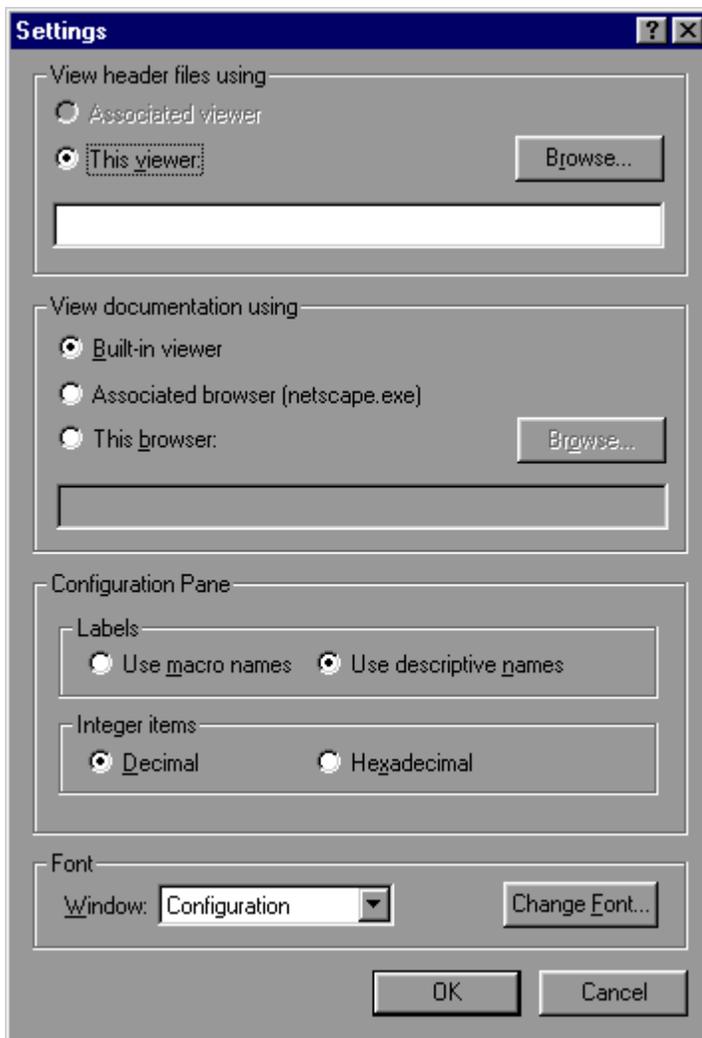
4

Screen Layout

The following windows are available within the eCos Configuration Tool:

■ Configuration Window

■ Properties Window

■ Short Description

■ Memory Layout

■ Conflicts

■ Output

The layout of the windows my be adjusted to suit your preferences: see "Settings" on page 12.

Configuration Window

This is the principal window used to configure eCos. It takes the form of a tree-based representation of the *configuration items* within the currently loaded eCos packages.

In the case of items whose values may be changed, controls are available to set the item values. These either take the form of check boxes or radio buttons within the tree itself or *cells* to the right of the thin vertical splitter bar. Controls in the tree may be used in the usual way; cells, however, must first be activated.

To activate a cell, simply click on it: it will assume a sunken appearance and data can then be edited in the cell. To terminate in-cell editing, click elsewhere in the configuration window or press **ENTER**. To discard the partial results of in-cell editing and revert to the previous value, press **ESCAPE**. Note that an asterisk appears

against configuration items which have changed since the configuration was last saved.
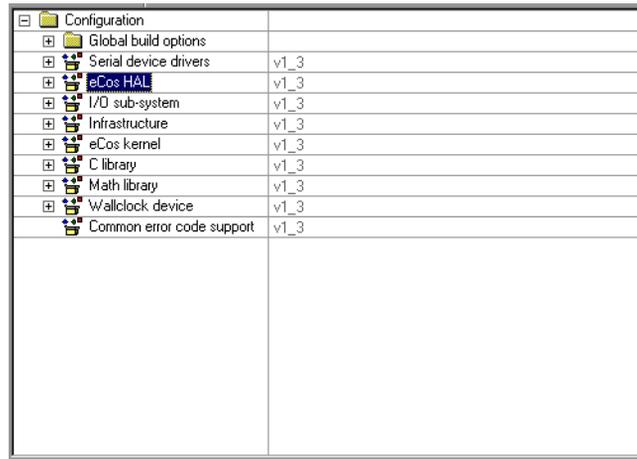
Cells come in three varieties, according to the type of data they accept:

Table 1:

Cell Type	Data Accepted
Integer	Decimal or hexadecimal values
Floating Point	Floating point values
String	Any

In the case of string cells, you can double-click the cell to display a dialog box containing a larger region in which to edit the string value. This is useful in the case of long strings, or those spanning multiple lines.

Disabled items

Some items will appear *disabled*. In this case the item label and any associated controls and cells will be grayed. It is not be possible to change the values of disabled items.

Right-Clicking

You can right-click on an item in the configuration window item to display a pop-up menu which (depending on the type of the item selected) allows you to:

- **Properties** – information relating to the currently selected item is displayed. The information is equivalent to that displayed in the Properties Window.
- **Restore Defaults** - the default value of the currently selected item is restored.
- **Visit Documentation** - causes the HTML page most closely relating to the currently selected item to be displayed. This has the same effect as double-clicking the URL property in the Properties Window.
- **View Header File** – this causes the file containing the items to be displayed. This is equivalent to double-clicking on the File property in the Properties Window. The viewer used for this purpose may be changed using the **View->Settings** menu item (see "Settings" on page 12). Note that this operation is only possible when the current configuration is saved, in order to avoid the possibility of changing the source repository.
- **Unload Package** - this is equivalent to using the **Build->Packages** menu item to select and unload the package in question.

Conflicts Window

This window exists to display any configuration item *conflicts*. Conflicts are the result of failures to meet the requirements between configuration items expressed in the CDL. See "Conflicts" in "CDL Concepts" on page 45.
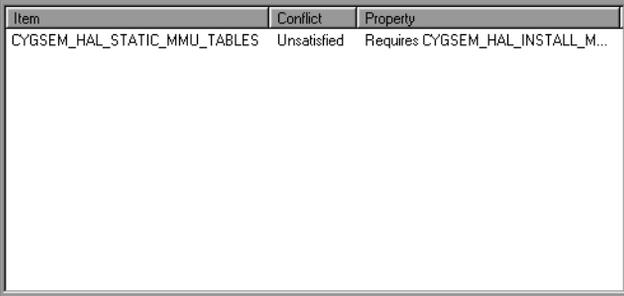
The window comprises three columns:

- **Item**

  This is the macro name of the first item involved in the conflict.

- **Conflict**

  This is a description of the conflict type. The currently supported types are "unresolved", "illegal value", "evaluation exception", "goal unsatisfied" and "bad data".

- **Property**

  This contains a description of the configuration item's property that caused the

conflict.

Within the conflicts window you can right-click on any item to display a context menu which allows you to choose from one of the following options:

■   **Locate the item involved in the conflict** – this will cause the configuration window to display the item relating most closely to the selected conflict.

You can use the **Tools->Resolve Conflicts** menu item to resolve conflicts – see "Resolving conflicts" on page 27.

Output Window

This window displays any output generated by execution of external tools and any error messages that are not suitable for display in other forms (for example, as message boxes).

Within the output window you can right-click to display a context menu which allows you to:

■   Save the contents of the window to a file
■   Clear the contents of the window

Properties Window

This window displays the CDL properties of the item currently selected in the configuration window. The same information may be displayed by right-clicking the item and selecting "properties".
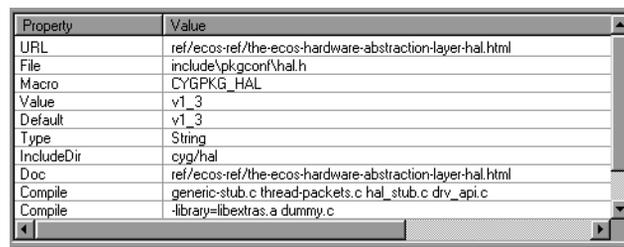
Property	Value
URL	ref/ecos-ref/the-ecos-hardware-abstraction-layer-hal.html
File	include\pkgconf\hal.h
Macro	CYGPKG_HAL
Value	v1_3
Default	v1_3
Type	String
IncludeDir	cyg/hal
Doc	ref/ecos-ref/the-ecos-hardware-abstraction-layer-hal.html
Compile	generic-stub.c thread-packets.c hal_stub.c drv_api.c
Compile	-library=libextras.a dummy.c

Two properties may be double-clicked as follows:

■   **URL** – double-clicking on a URL property causes the referenced HTML page to be displayed. This has the same effect as right-clicking on the item and choosing "Visit Documentation".

■ **File** – double-clicking on a File property in a saved configuration causes the File to be displayed. The viewer used for this purpose may be changed using the **View->Settings** menu item. Note that this operation is only possible when the current configuration is saved, in order to avoid the possibility of changing the source repository.

Short Description Window

This window displays a short description of the item currently selected in the configuration window. More extensive documentation may be available by right-clicking on the item and choosing "Visit Documentation".

Memory Layout Window

The memory layout window presents a graphical view of the memory layout of the currently selected combination of target architecture, platform and start-up type. Each memory region is represented by a horizontal bar within the window. Each bar is further divided into a number of blocks representing memory sections. Unused parts of a memory section are represented using hatching. All numeric information is presented in hexadecimal format:

Default memory layouts are provided for all supported platforms; you do not need to edit these layouts in order to begin development on standard, supported, eCos platforms. However, you may need to modify the memory layouts at certain times, for example when additional memory is installed on an evaluation board. When the memory layout is modified, a new linker script fragment is generated to allow the linker to make use of the new memory.

Layout Manipulation

The memory layout window includes controls to create, delete and modify the properties of both memory regions and memory sections (collectively referred to as *memory items*). These manipulation functions are accessible from both memory items and from the memory layout toolbar, which may be shown or hidden by the **View->Toolbars->Memory Layout** menu item. When modifying or deleting an item, it is necessary to first select it with your mouse. The currently selected item is

displayed with a focus rectangle (as section *rodata* above). Creation and modification of a memory item is achieved using a property sheet. The property sheet for a memory item may also be accessed by double-clicking on the item in the memory window.

Memory Regions

Details of a memory region may be specified using the region properties sheet, displayed by double-clicking on the name of the memory region in the memory layout display. The general settings page of this sheet allows editing of the region parameters:
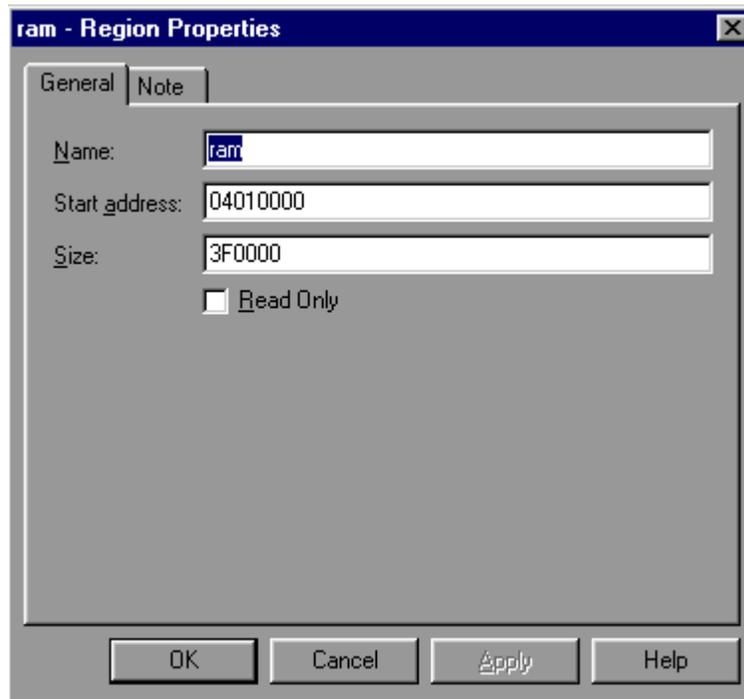
The name of each memory region is arbitrary, but should not contain spaces or punctuation characters. The start address and size of each memory region is specified in bytes and entered as hexadecimal numbers. The **Read Only** check box should be checked where the memory region represents a block of read-only memory. This information is used to verify that the initial and final locations of any relocating memory sections are within appropriate memory regions.

The **Note** page of the region properties sheet may be used to keep notes concerning the memory regions. These notes are saved with the memory layout in the build tree.

Memory sections

Details of a memory section may be specified or modified using the section properties sheet. The general settings page of this sheet allows editing of the parameters which are common to all sections:
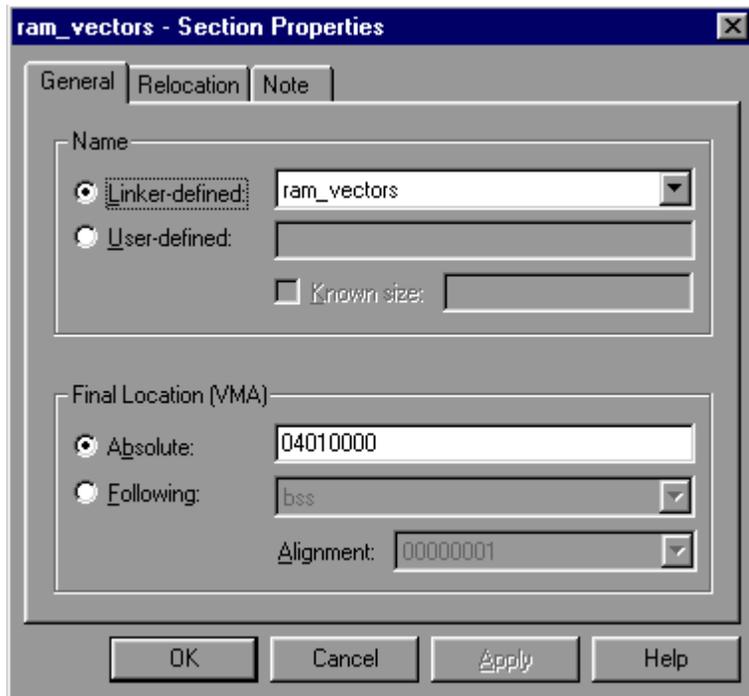
Each memory section is either linker-defined or user-defined. The name of a linker-defined section is selected from a drop-down list appropriate for the currently selected target architecture. Only those names which are not currently in use are presented. The name of a user-defined section must not contain spaces or punctuation characters. The size of a user-defined section may also be specified by checking the **Known Size** check box. The size should then be entered as a hexadecimal number. User-defined sections of unknown size are assumed to occupy all available space up to the next section or the end of the memory region.

The final memory location after relocation (also known as VMA) of a memory section may be defined using an absolute start address or by specifying another section which it follows in the memory map. Where an absolute address is required, this should be entered as a hexadecimal number.

Alternatively, the preceding section may be selected from a drop-down list of appropriate existing sections. In this case, the alignment of the section in terms of an n-byte boundary should also be selected.

The relocation settings page allows editing of the parameters which are specific to relocating sections:
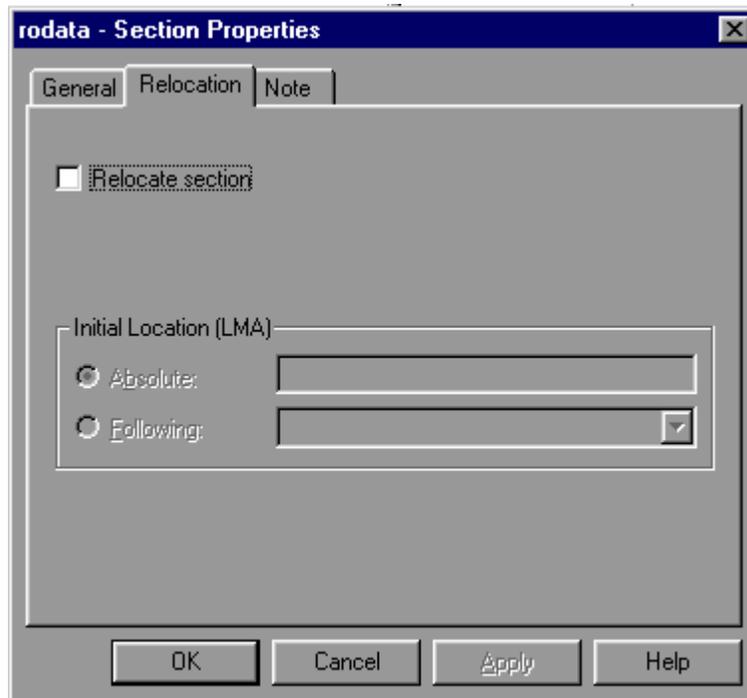
The relocation of a memory section at system start-up is enabled by checking the **Relocate Section** check box. The initial size to which the memory section is loaded (also known as the LMA) may be defined using an absolute start address or by specifying another section which it follows in the memory map. Where an absolute address is required, this should be entered as a hexadecimal number. The address must lie within a read-only memory region. Alternatively, the preceding section may be selected from a drop-down list of appropriate existing sections. The initial location of the preceding section must be a location in a read-only memory region.

The note page of the section properties sheet may be used to keep notes concerning the memory section. These notes are saved with the memory layout in the build tree.

Memory access

User-defined memory sections may be accessed using C preprocessor macros defined in a memory layout header file exported by the eCos Configuration Tool. The name of the memory layout header file appropriate for the current configuration is defined by the *CYGHWR_MEMORY_LAYOUT* configuration item.

Macros specifying the start address and size are defined for each user-defined memory section and may be accessed as demonstrated in the following example:

Table 2: Accessing a user-defined memory section named example

```
#include <pkgconf/system.h>
#include CYGHWR_MEMORY_LAYOUT_H

int main ()
{
 // use the memory section as an integer array
 int * array = (int *) CYGMEM_SECTION_example;
 unsigned int array_size = CYGMEM_SECTION_example_SIZE / sizeof (int);

 // initialize each array element
 unsigned int count;
 for (count = 0; count < array_size; ++count)
 array [count] = 0;

 return 0;
}
```

5

Updating the Configuration

Adding and Removing Packages

To add or remove packages from the configuration, select **Build**->**Packages**. The
following dialog box will be displayed:

Packages dialog box
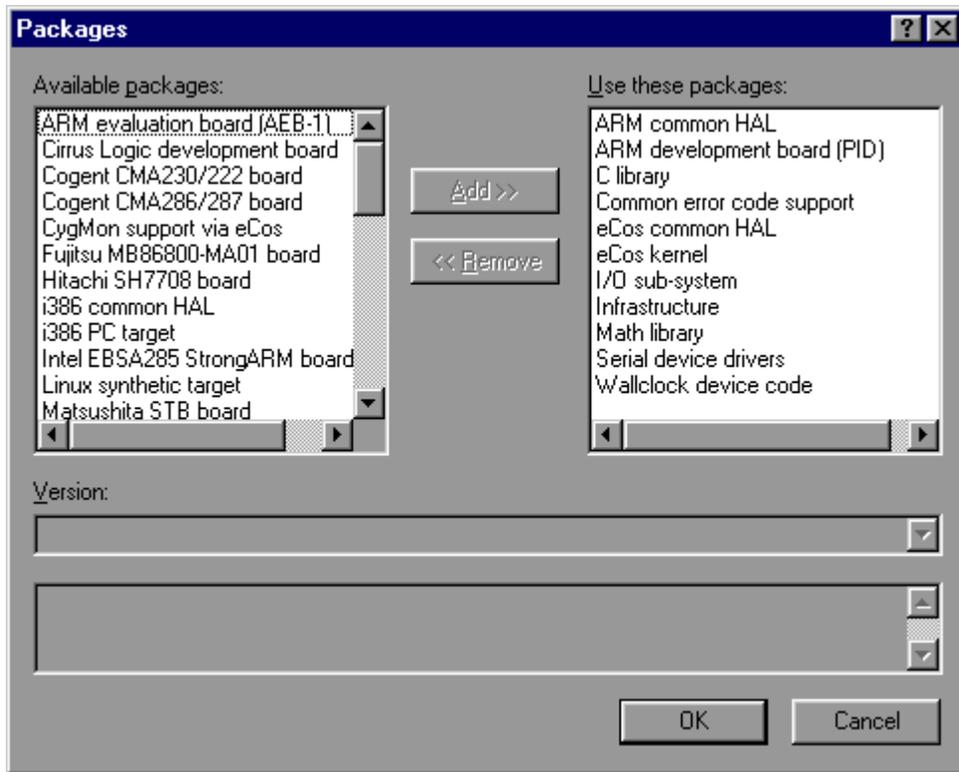
The left list shows those packages that are available to be loaded. The right-hand list shows those that are currently loaded. In order to transfer packages from one list to another (that is, to load or unload packages) double-click the selection or click the **Add** or **Remove** buttons.

The version drop-list displays the versions of the selected packages. When loading packages, this control may be used to load versions other than the most recent (current). Note that if more than one package is selected, the version drop-list will display only the versions in common to all the selected packages.

The bottommost window in the dialog displays a brief description of the selected package. If more than one package is selected, this window will be blank.

Platform Selection

To add, modify or remove entries in the list of platforms used for running tests, select **Tools->Platforms**. The following dialog will be displayed:
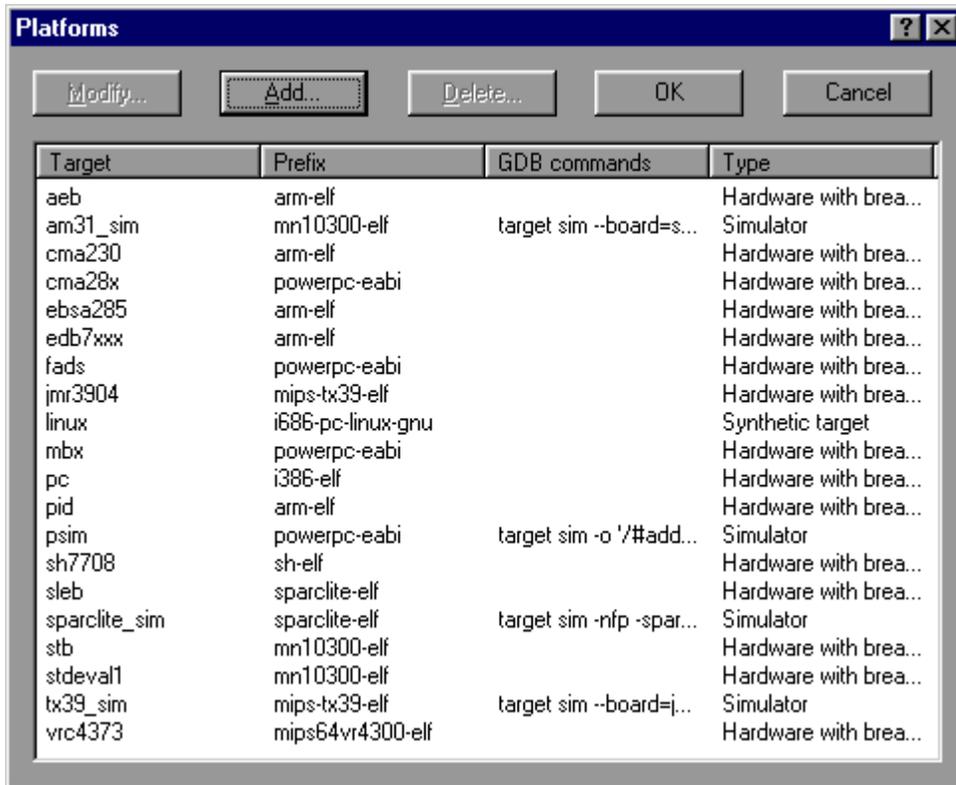
You may add, modify or remove platform entries as you wish, but in order to run tests, a platform must be defined to correspond to the currently loaded hardware template. The information associated with each platform name is used to run tests.

To modify a platform, click the **Modify** button with the appropriate platform selected, or double-click on an entry in the list. A dialog will be displayed that allows you to change the command prefix, platform type and arguments for **GDB**.

Figure 8: **Platform Modify dialog box**

To add a new platform, click the **Add** button. A similar dialog will be displayed that allows you to define a new platform. To remove a platform, click the **Delete** button or press the **DEL** key with the appropriate platform selected.

Figure 9: **New Platform dialog box**

The command prefix is used when running tests in order to determine the names of the executables (such as gdb) to be used. For example, if the gdb executable name is "arm-elf-gdb.exe" the prefix should be set to "arm-elf".

The platform type indicates the capabilities of the platform - whether it is hardware or a simulator, and whether breakpoints are supported.

The arguments for the **GDB** field allow additional arguments to be passed to gdb when it is used to run a test. This is typically used in the case of simulators linked to gdb in order to define memory layout.

Using Templates

To load a configuration based on a template, select **Build**->**Templates**.

The following dialog box will be displayed:

Figure 10: **Templates dialog box**
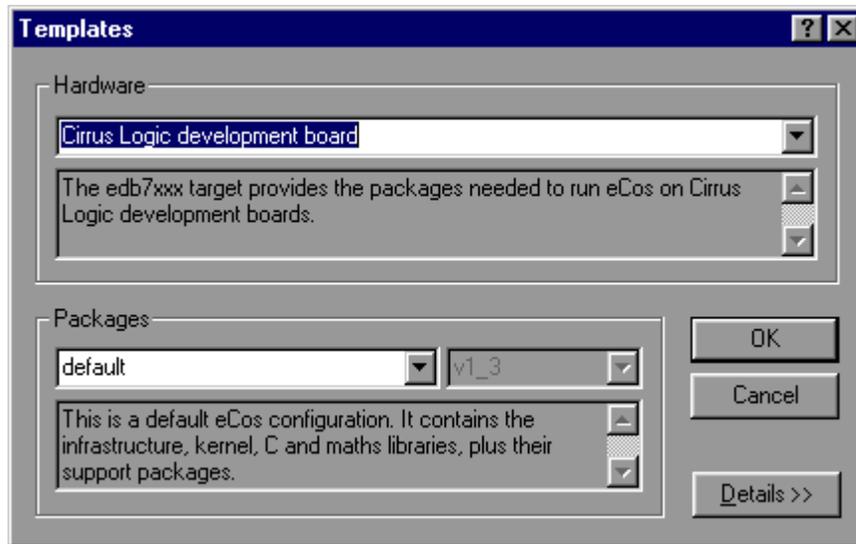
Change the hardware template, the packages template, or both. To select a hardware template, choose from the first drop-list. To choose a packages template, choose from the second. Brief descriptions of each kind of template are provided in the corresponding edit boxes.

Resolving conflicts

During the process of configuring eCos it is possible that conflicts will be created. For more details of the meaning of conflicts, see *"CDL Concepts" on page 45.*

The Conflicts Window displays all conflicts in the current configuration. Additionally, a window in the status bar displays a count of the conflicts. Because the resolution of conflicts can be time-consuming, a mechanism exists whereby conflicts can be resolved automatically.

You can choose to have a conflicts resolution dialog box displayed by means of the **Tools->Options** menu item.

Figure 11: **Options**
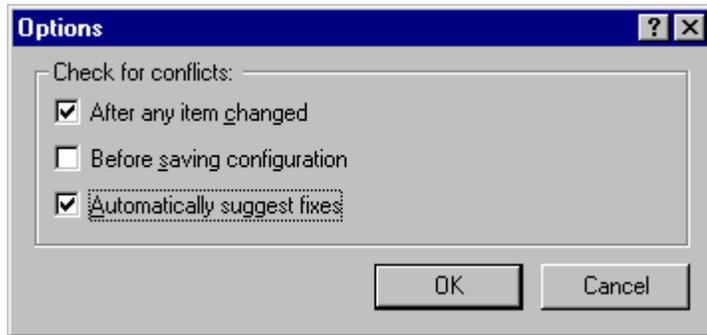
You can choose to have conflicts checked under the following circumstances:

- After any item is changed (in other words, as soon as the conflict is created)
- Before saving the configuration (including building)
- Never

The method you chose depends on how much you need your configuration to be free of conflicts. You may want to avoid having to clean up all the conflicts at once, or you may want to keep the configuration consistent at all times. If you have major changes to implement, which may resolve the conflicts, then you might want to wait until after you have completed these changes before you check for conflicts.

NOTE   If you choose to check conflicts after any item is changed, only newly arising conflicts are displayed. If you choose to check for conflicts before saving the configuration, the complete set is displayed.

Automatic resolution

If you check the "Automatically suggest fixes" check box, a conflicts resolution dialog box will be displayed whenever new conflicts are created. The same dialog box may be displayed at any stage by means of the **Tools->Resolve Conflicts** menu item.

The conflicts resolution dialog box contains two major windows.

Figure 12: **Resolve conflicts window**
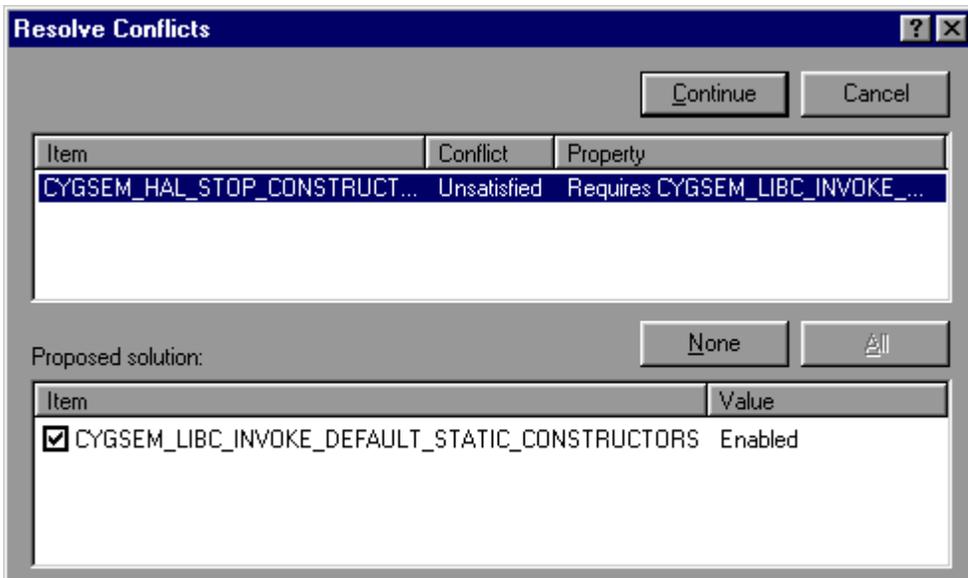
The upper contains the set of conflicts to be addressed; the format of the data being as that of the Conflicts Window. The lower window contains a set of proposed resolutions – each entry is a suggested configuration item value change that as a whole may be expected to lead to the currently selected conflict being resolved.

Note that there is no guarantee:

- that automatic resolutions will be determinable for every conflict.
- that the resolutions for separate conflicts will be independent. In other words, the resolution of one conflict may serve to prevent the resolution of another.
- that the resolution conflicts will not create further conflicts.

The above warnings are, however, conservative. In practice (so long as the number and extent of conflicts are limited) automatic conflict resolution may be used to good effect to correct problems without undue amounts of programmer intervention.

In order to select the conflicts to be applied, select or clear the check boxes against the resolutions for each proposed resolution. By default all resolutions are selected; you can return to the default state (in other words, cause all check boxes for each conflict to again become checked) by pressing the "Reset" button. Note that multiple selection may be used in the resolutions control to allow ranges of check boxes to be toggled in one gesture.

When you are happy to apply the selected resolutions for each conflict displayed, click **Apply**; this will apply the resolutions. Alternatively you may cancel from the dialog box without any resolutions being applied.

6

Searching

Select **Edit** --> **Find**. You will be presented with a Find dialog box:

Figure 13: Find dialog box

Using this dialog box you can search for an exact text string in any one of three ways, as specified by your selection in the "Search in" drop-list:

- Macro names - the search is for a text match within configuration item macro names
- Item names - the search is for a text match within configuration item descriptive names
- Short descriptions - the search is for a text match within configuration item short descriptions

Note that to invoke **Find** you can also click the **Find** icon on the toolbar.

7

Building

When you have configured eCos, you may build the configuration.

On the **Build** menu, click:

- **Library** (or click the Build Library icon on the toolbar) – this causes the eCos configuration to be built. The result of a successful build will be (among other things) a library against which user code can be linked
- **Tests** – this causes the eCos configuration to be built, and additionally builds the relevant test cases linked against the eCos library
- **Clean** – this removes all intermediate files, thus causing a subsequent build/library or build/tests operation to cause recompilation of all relevant files.
- **Stop** – this causes a currently executing build (any of the above steps) to be interrupted

Build options may be displayed by using the **Build->Options** menu item. This displays a dialog box containing a drop-list control and two windows. The drop-list control allows you to select the type of build option to be displayed (for example "LDFLAGS" are the options applied at link-time. The left-hand window is a tree view of the packages loaded in the current configuration. The right-hand window is a list of the build options that will be used for the currently selected package.

Note that this dialog box currently affords only read-only access to the build options. In order to change build options you must edit the relevant string configuration item.

A single level of inheritance is supported: each package's build options are combined with the global options (these are to be found in the "Global build options" folder in the configuration view).
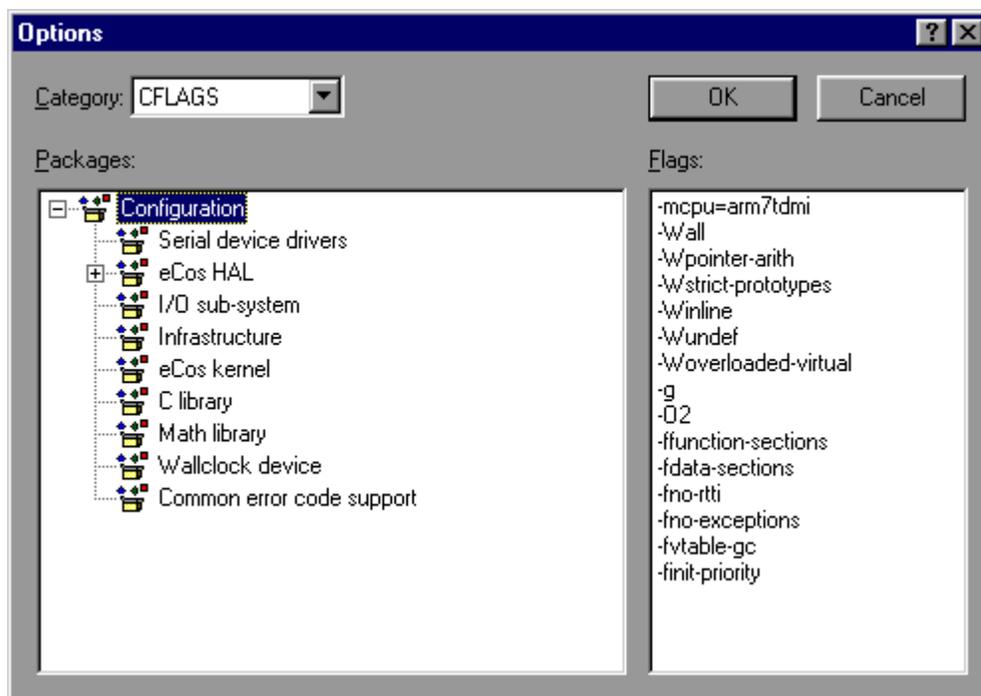
Selecting Build Tools

Normally the installation process will supply the information required for the eCos Configuration Tool to locate the build tools (compiler, linker, etc…) necessary to perform a build. However if this information is not registered, or it is necessary to specify the location manually (for example, when a new toolchain installation has been made), select **Tools**->**Paths**->**Build Tools**. The following dialog box will be displayed:

Figure 14: Build tools
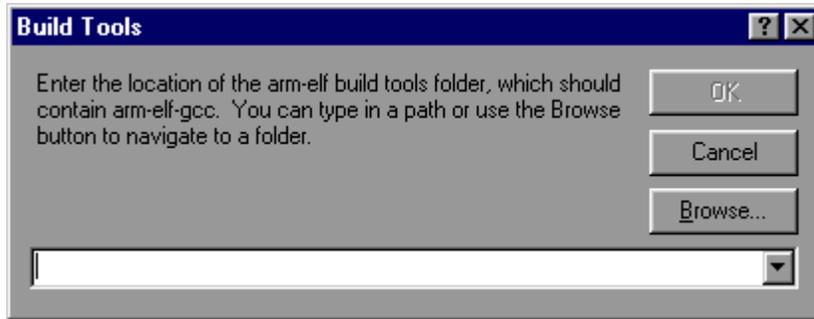
This dialog box allows you to locate the folder containing the build tools.

Selecting User Tools

Normally the installation process will supply the information required for the eCos Configuration Tool to locate the user tools (cat, ls, etc…) necessary to perform a build. However if this information is not registered, or it is necessary to specify the location manually (for example, when a new toolchain installation has been made), select **Tools**->**Paths**->**User Tools**. The following dialog box will be displayed:

Figure 15: User tools

8
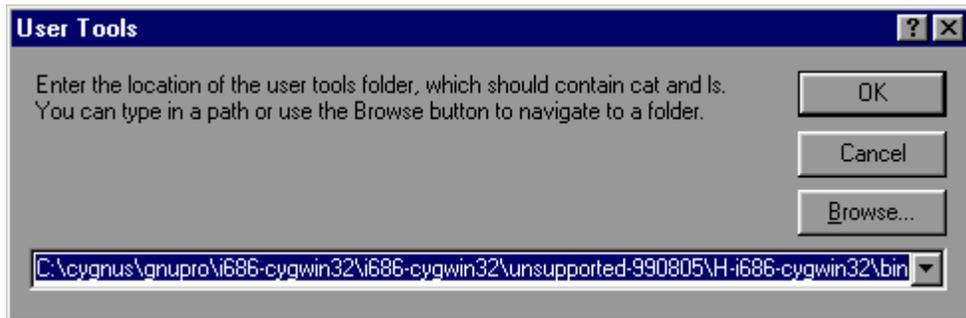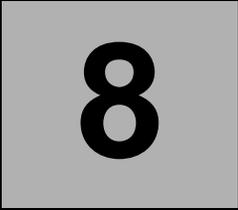
Execution

Test executables that have been linked using the Build/Tests operation against the current configuration can be executed by selecting **Tools**->**Run Tests**.

When tests are run, the **Configuration Tool** looks for a platform name corresponding to the currently loaded hardware template. If no such platform is found, a dialog will be displayed for you to define one; this dialog is similar to that displayed by the **Add** function in the **Tools->Platforms** dialog, but in this case the platform name cannot be changed.

When a test run is invoked, a property sheet is displayed, comprising three tabs: **Executables**, **Output** and **Summary**.

Note that the property sheet is resizable.

Three buttons appear on the property sheet itself: **Run/Stop**, **Close** and **Properties**.

The **Run** button is used to initiate a test run. Those tests selected on the **Executables** tab are run, and the output recorded on the **Output** and **Summary** tabs. During the course of a run, the **Run** button changes to "Stop". The button may be used to interrupt a test run at any point.

Properties

The **Properties** button is used to change the connectivity properties for the test run.

Figure 16: **Properties dialog box**
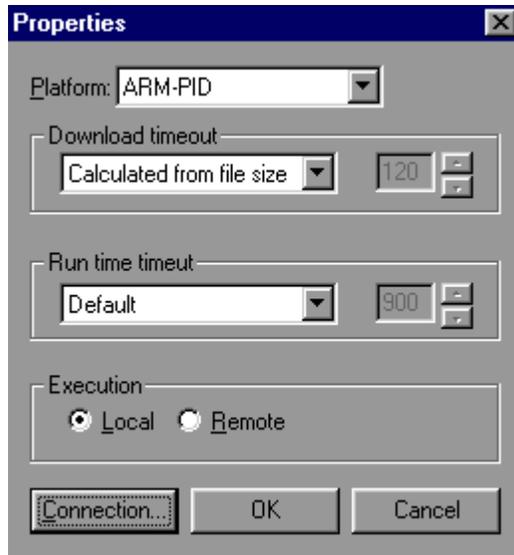
Download Timeout

This group of controls serves to set the maximum time that is allowed for downloading a test to the target board. If the time is exceeded, the test will be deemed to have failed for reason of "Download Timeout" and the execution of that particular test will be abandoned. This option only applies to tests run on hardware, not to those executed in a simulator. Times are in units of elapsed seconds.

Three options are available using the drop-down list:

■ Calculated from file size - an estimate of the maximum time required for download is made using the (stripped) executable size and the currently used baud rate

■ Specified - a user-specified value may be entered in the adjacent edit box

■ None - no maximum download time is to be applied.

Run time Timeout

This group of controls serves to set the maximum time that is allowed for executing a test on the target board or in a simulator. If the time is exceeded, the test will be deemed to have failed for reason of "Timeout" and the execution of that particular test will be abandoned. In the case of hardware, the time is measured in elapsed seconds: in the case of a simulator it is in CPU seconds.

Three options are available using the drop-down list:

- None - no maximum download time is to be applied.
- Specified - a user-specified value may be entered in the adjacent edit box
- Default - a default value of 30 seconds is used

Connection

The **Connection** button may be used to specify how the target board is to be accessed:

Figure 17: Connection dialog box
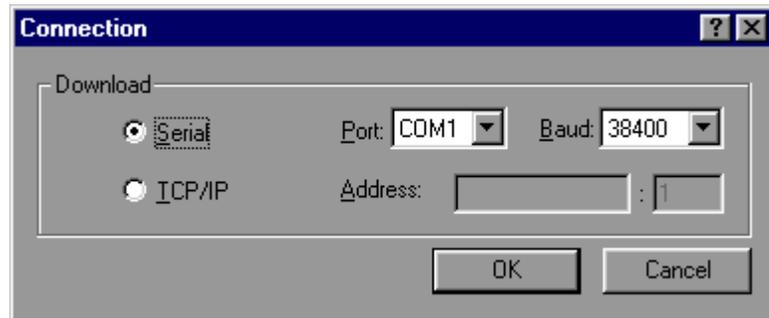
If the target board is connected using a serial cable, the **Serial** radio button should be checked. In this case you can select a port (COM1, COM2, …) and an appropriate baud rate using drop-list boxes.

If the target board is accessed remotely using GDB remote protocol, the "TCP/IP" radio button should be checked. In this case you can select a host name and TCP/IP port number using edit boxes.

Executables Tab

This is used to adjust the set of tests available for execution. A check box against each executable name indicates whether that executable will be included when the **Run** button is pressed. The **Check All** and **Uncheck All** buttons may be used to check or uncheck all items.

When the property sheet is first displayed, it will be pre-populated with those test executables that have been linked using the Build/Tests operation against the current configuration.

Figure 18: **Run tests**
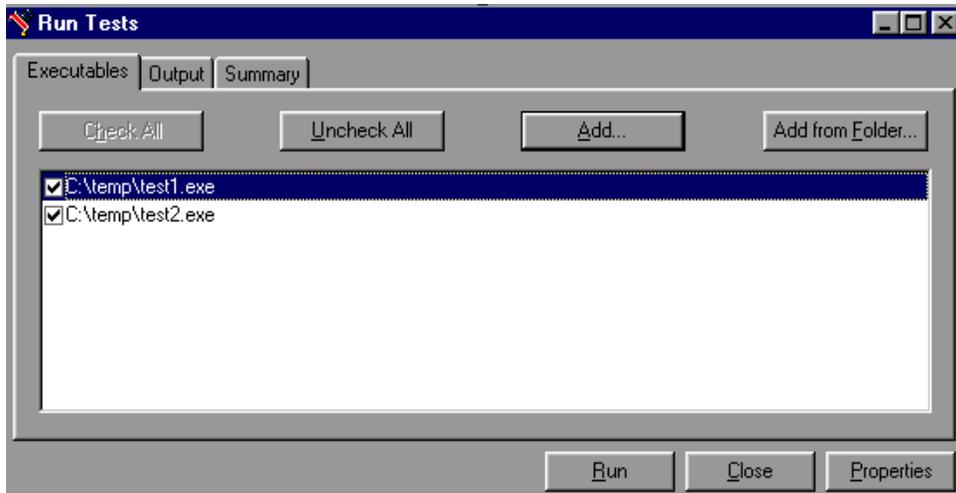
You can right-click in the window to display a context menu containing **Add** and **Remove** items. Clicking **Remove** will remove those executables selected. Clicking **Add** will display a dialog box that allows you to add to the set of items. Equivalently the **Add** button may be used to add executables, and the **DEL** key may be used to remove them.

You can use the **Add from Folder** button to add a number of executables in a specified folder (optionally including subfolders).

Figure 19: **Add files from folder**
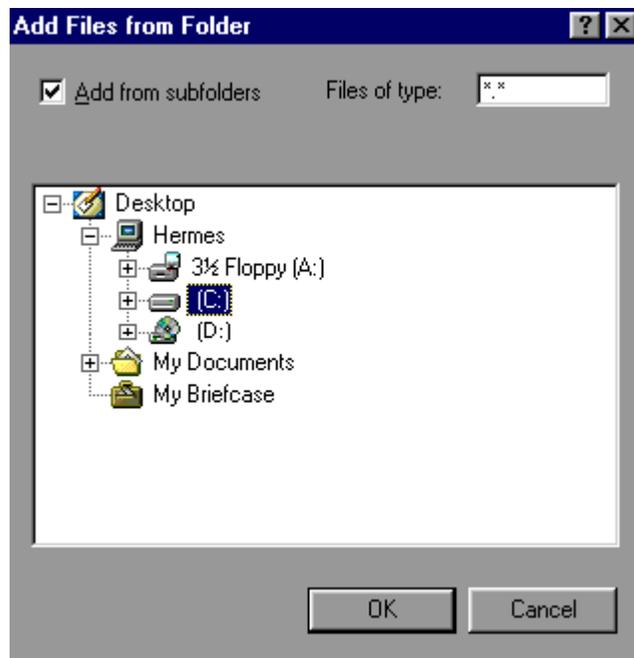
The "Add from subfolders" check box should be checked if you wish the search for executables to descend into subfolders (in the example above the whole of the C drive would be searched).

The "Files of type" edit box should be used to specify the extension of those files to be matched [for example, "*.exe"].

Output Tab

This tab is used to display the output from running tests. The output can be saved to a file or cleared by means of the popup menu displayed when you right-click in the window.

Summary Tab

This tab is used to display a record, in summary form, of those tests executed. For each execution, the following information is displayed:

- *Time* - the date and time of execution
- *Host* - the host name of the machine from which the test was downloaded
- *Platform* - the platform on which the test was executed
- *Executable* - the executable (file name) of the test executed

- *Status* - the result of executing the test. This will be one of the following:
  - ❑ Not started
  - ❑ No result
  - ❑ Inapplicable
  - ❑ Pass
  - ❑ DTimeout
  - ❑ Timeout
  - ❑ Cancelled
  - ❑ Fail
  - ❑ Assert fail
- *Size* - the size [stripped/unstripped] of the test executed
- *Download* - the download time [mm:ss/mm:ss] used. The first of the two times displayed represents the actual time used: the second the limit time.
- *Elapsed* - the elapsed time [mm:ss] used.
- *Execution* - the execution time [mm:ss/mm:ss] used. The first of the two times displayed represents the actual time used: the second the limit time.

The output can be saved to a file or cleared by means of the popup menu displayed when you right-click in the window.

9

Creating a Shell

To call up a shell window, select **Tools**->**Shell**:
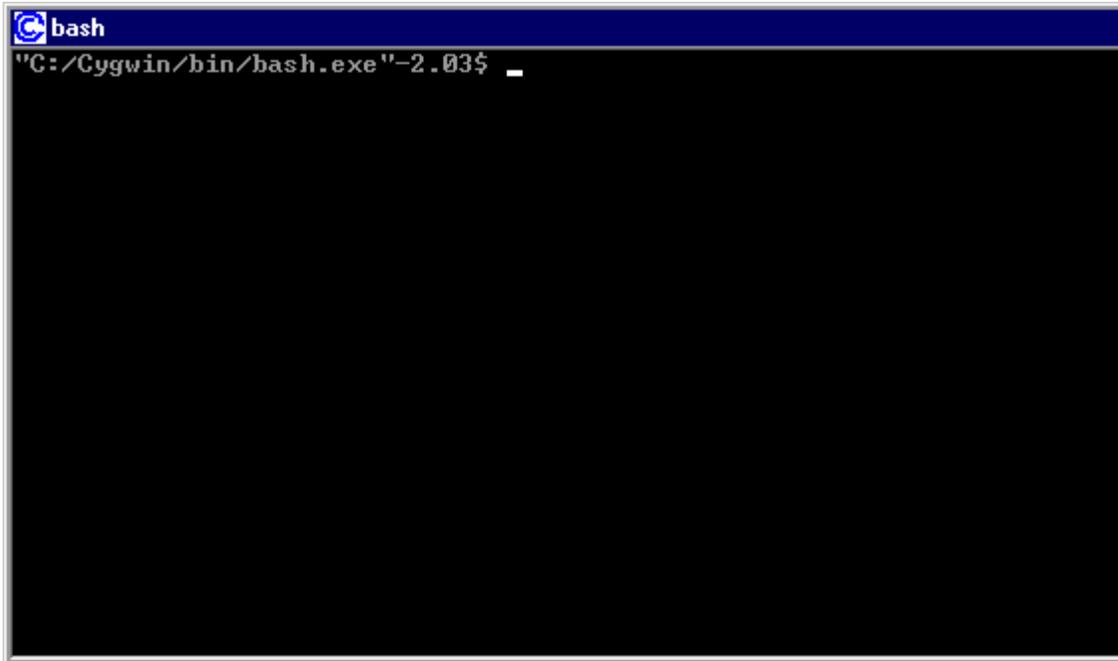

```
C bash
"C:/Cygwin/bin/bash.exe"-2.03$ _
```

Keyboard Accelerators

The following table presents the list of keyboard accelerators that can be used with the **Configuration Tool**.

Table 3: Keyboard accelerators

Accelerator	*Action*	*Remarks*
Alt+1	hide/show properties window	
Alt+2	hide/show documentation window	
Alt+3	hide/show short description window	
Alt+4	hide/show memory layout window	
Alt+5	hide/show output window	
Ctrl+A	select all	output window and in-cell editing
Ctrl+C	copy	output window and in-cell editing
Ctrl+F	Edit->Find	
Ctrl+N	File->New	
Ctrl+O	File->Open	
Ctrl+S	File->Save	
Ctrl+V	Paste	in-cell editing only
Ctrl+X	Cut	in-cell-editing only
Ctrl+Z	Undo	in-cell editing only
F1	Context-sensitive help	
F3	Find next	
F7	Build->Library	
Shift+F7	Build->Tests	
Alt+F6	View->Next window	
Shift+Alt+0	View->Previous window	
Shift+Ins	Paste	in-cell editing only

Accelerator	*Action*	*Remarks*
Shift+**F10**	Display context menu	Configuration window
Alt+**Enter**	Display properties dialog box	Configuration window
>	Increment item value	Configuration window
<	Decrement item value	Configuration window
Space	Toggle item value	Configuration window

Part II: eCos Programming Concepts and Techniques

Programming with eCos is somewhat different from programming in more traditional environments. eCos is a configurable open source system, and you are able to configure and build a system specifically to meet the needs of your application.

Various different directory hierarchies are involved in configuring and building the system: the *component repository*, the *build tree*, and the *install tree*. These directories exist in addition to the ones used to develop applications.

10

CDL Concepts

About this Chapter

This chapter serves as a brief introduction to the concepts involved in eCos (Embedded Configurable Operating System). It describes the configuration architecture and the underlying technology to a level required for the embedded systems developer to configure eCos. It does not describe in detail aspects such as how to write reusable components for eCos: this information is given in the *CDL Writer's Guide*.

Background

Software solutions for the embedded space place particularly stringent demands on the developer, typically represented as requirements for small memory footprint, high performance and robustness. These demands are addressed in eCos by providing the ability to perform compile-time specialization: the developer can tailor the operating system to suit the needs of the application. In order to make this process manageable, eCos is built in the context of a Configuration Infrastructure: a set of tools including a *Configuration Tool* and a formal description of the process of configuration by means of a *Component Definition Language*.

Configurations

eCos is tailored at source level (that is, before compilation or assembly) in order to create an eCos *configuration*. In concrete terms, an eCos configuration takes the form of a configuration save file (with extension .ecc) and set of files used to build user applications (including, when built, a library file against which the application is linked).

Component Repository

eCos is shipped in source in the form of a *component repository* - a directory hierarchy that contains the sources and other files which are used to build a configuration. The component repository can be added to by, for example, downloading from the net.

Component Definition Language

Part of the component repository is a set of files containing a definition of its structure. The form used for this purpose is the *Component Definition Language* (CDL). CDL defines the relationships between components and other information used by tools such as the eCos Configuration Tool. CDL is generally formulated by the writers of components: it is not necessary to write or understand CDL in order for the embedded systems developer to construct an eCos configuration.

Packages

The building blocks of an eCos configuration are called *packages*. Packages are the units of software distribution. A set of core packages (such as kernel, C library and math library) is provided by Red Hat: additional third-party packages will be available in future.

A package may exist in one of a number of *versions*. The default version is the *current* version. Only one version of a given package may be present in the component repository at any given time.

Packages are organized in a tree hierarchy. Each package is either at the top-level or is the child of another package.

The eCos Administration Tool can be used to add or remove packages from the component repository. The eCos Configuration Tool can be used to include or exclude packages from the configuration being built.

Configuration Items

Configuration items are the individual entities that form a configuration. Each item corresponds to the setting of a C pre-processor macro (for example, CYGHWR_HAL_ARM_PID_GDB_BAUD). The code of eCos itself is written to test such preprocessor macros so as to tailor the code. User code can do likewise.

Configuration items come in the following flavors:

- *None*: such entities serve only as placeholders in the hierarchy, allowing other entities to be grouped more easily.
- *Boolean* entities are the most common flavor; they correspond to units of functionality that can be either enabled or disabled. If the entity is enabled then there will be a #define; code will check the setting using, for example, #ifdef
- *Data* entities encapsulate some arbitrary data. Other properties such as a set or range of legal values can be used to constrain the actual values, for example to an integer or floating point value within a certain range.
- *Booldata* entities combine the attributes of *Boolean* and *Data*: they can be enabled or disabled and, if enabled, will hold a data value.

Like packages, configuration items exist in a tree-based hierarchy: each configuration item has a parent which may be another configuration item or a package. Under some conditions (such as when packages are added or removed from a configuration), items may be "re-parented" such that their position in the tree changes.

Expressions

Expressions are relationships between CDL items. There are three types of expression in CDL:

Table 4: CDL Expressions

Expression Type	Result	Common Use [see Table 2]
Ordinary	A single value	legal_values property
List	A range of values (for example "1 to 10")	legal_values property
Goal	True or False	requires and active_if properties

Properties

Each configuration item has a set of properties. The following table describes the most commonly used:

A complete description of properties is contained in the *CDL Writer's Guide*.

Table 5: Configuration properties

Property	Use
Flavor	The "type" of the item, as described above
Enabled	Whether the item is enabled
Current_value	The current value of the item
Default_value	An ordinary expression defining the default value of the item
Legal_values	A list expression defining the values the item may hold (for example, 1 to10)
Active_if	A goal expression denoting the requirement for this item to be active (see below: *Inactive Items*)
Requires	A goal expression denoting requirements this item places on others (see below: *Conflicts*)
Calculated	Whether the item as non-modifiable
Macro	The corresponding C pre-processor macro
File	The C header file in which the macro is defined
URL	The URL of a documentation page describing the item
Hardware	Indicates that a particular package is related to specific hardware

Inactive Items

Descendants of an item that is disabled are inactive: their values may not be changed. Items may also become *inactive* if an active_if expression is used to make the item dependent on an expression involving other items.

Conflicts

Not all settings of configuration items will lead to a coherent configuration; for example, the use of a timeout facility might require the existence of timer support, so if the one is required the other cannot be removed. Coherence is policed by means of consistency rules (in particular, the goal expressions that appear as CDL items *requires* and *active_if* attributes [see above]). A violation of consistency rules creates a *conflict*, which must be resolved in order to ensure a consistent configuration. Conflict resolution can be performed manually or with the assistance of the eCos tools. Conflicts come in the following flavors:

■ An *unresolved* conflict means that there is a reference to an entity that is not yet in the current configuration

■ An *illegal value* conflict is caused when a configuration item is set to a value that is not permitted (that is, a *legal_values* goal expression is failing)

- An *evaluation exception* conflict is caused when the evaluation of an expression would fail (for example, because of a division by zero)
- An *unsatisfied goal* conflict is caused by a failing *requires* goal expression
- A *bad data* conflict arises only rarely, and corresponds to badly constructed CDL. Such a conflict can only be resolved by reference to the CDL writer.

Templates

A *template* is a saved configuration - that is, a set of packages and configuration item settings. Templates are provided with eCos to allow you to get started quickly by instantiating (copying) a saved configuration corresponding to one of a number of common scenarios; for example, a basic eCos configuration template is supplied that contains the infrastructure, kernel, C and math libraries, plus their support packages.

11

The Component Repository and Working Directories

Each of the file trees involved in eCos development has a different role.

Component Repository

The eCos component repository contains directories for all the packages that are shipped with eCos or provided by third parties.

The component repository should not be modified as part of application development.

Figure 20: **Component repository**
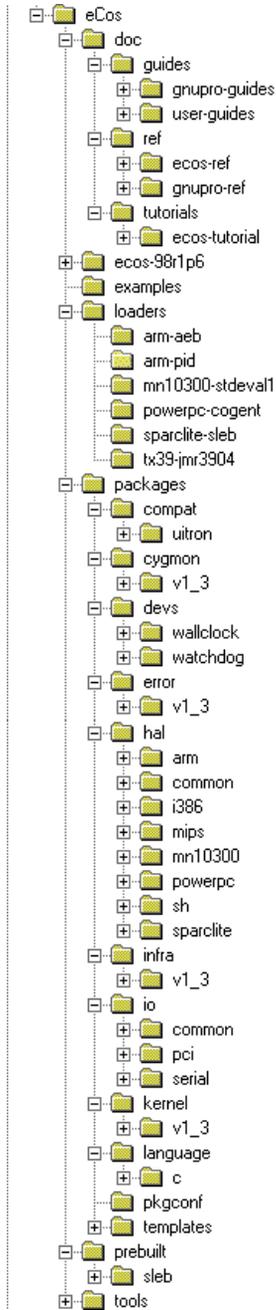
Purpose

The component respository is the master copy of source code for all system and third party components. It also contains some files needed to administer and build the system, such as ecosadmin.tcl.

How is it modified?

You modify it by importing new versions of packages from a distribution or removing existing packages. These activities are undertaken using the **eCos Package Administration Tool**.

When is it edited manually?

Files in the component repository should only be edited manually as determined by the component maintainer.

User applications

User application source code should *not* go into the component repository.

Examples of files in this hierarchy:

BASE_DIR/doc/ref/ecos-ref.html

> The top level HTML file for the *eCos Reference Manual*.

BASE_DIR/prebuilt/pid/tests/kernel/v1_3_x/tests/thread_gdb.exe

BASE_DIR/prebuilt/linux/tests/kernel/v1_3_x/tests/thread_gdb.exe

> Prebuilt tests for the supported platforms, and the synthetic Linux target.

BASE_DIR/examples/twothreads.c

> One of the example programs.

BASE_DIR/ecosadmin.tcl

> The Tcl program which is used to import new versions of packages from a distribution or remove existing packages.

BASE_DIR/packages/language/c/libm/v1_3_x/src/double/portable-api/s_tanh.c

> Implementation of the hyperbolic tangent function in the standard math library.

BASE_DIR/pkgconf/rules.mak

> A file with make rules, used by the makefile.

Build Tree

The build tree is the directory hierarchy in which all *generated* files are placed. Generated files consist of the makefile, the compiled object files, and a dependency file (with a .d extension) for each source file.

Purpose

The build tree is where all intermediate object files are placed.

How is it modified?

Recompiling can modify the object files.

User applications

User application source or binary code should *not* go in the build tree.

Examples of files in this hierarchy

*ecos-work/language/c/libc/*v1_3_x/src

    The directory in which object files for the C library are built.

Install Tree

The install tree is the location for all files needed for application development. The `libtarget.a` library, which contains the custom-built eCos kernel and other components, is placed in the install tree, along with all packages' public header files. If you build the tests, the test executable programs will also be placed in the install tree.

By default, the install tree is created by `ecosconfig` in a subdirectory of the build tree called `install`. This can be modified with the `--prefix` option (see "Manual Configuration" on page 64).

Purpose

The install tree is where the custom-built `libtarget.a` library, which contains the eCos kernel and other components, is located. The install tree is also the location for all the header files that are part of a published interface for their component.

How is it modified?

Recompiling can replace `libtarget.a` and the test executables.

When is it edited manually?

Where a memory layout requires modification without use of the eCos Configuration Tool, the memory layout files must be edited directly in the install tree. These files are located at `install/include/pkgconf/mlt_*.*`. Note that subsequent modification of the install tree using the Configuration Tool will result in such manual edits being lost.

User applications

User application source or binary code should *not* go in the install tree.

Examples of files in this hierarchy

install/lib/libtarget.a

> The library containing the kernel and other components.

install/include/cyg/kernel/kapi.h

> The header file for the kernel C language API.

install/include/pkgconf/mlt_arm_pid_ram.ldi

> The linker script fragment describing the memory layout for linking applications intended for execution on an ARM PID development board using RAM startup.

install/include/stdio.h

> The C library header file for standard I/O.

Application Build Tree

This tree is not part of eCos itself: it is the directory in which eCos end users write their own applications.

Example applications and their `Makefile` are located in the component repository, in the directory *BASE_DIR*/examples.

There is no imposed format on this directory, but there are certain compiler and linker flags that must be used to compile an eCos application. The basic set of flags is shown in the example `Makefile`, and additional details can be found in "Compiler and Linker Options" on page 55.

12

Compiler and Linker Options

eCos is built using the GNU C and C++ compilers. The versions of the tools **Red Hat** has prepared for this release have some enhancements, such as constructor priority ordering and selective linking, which will eventually become part of the standard distribution.

Some **GCC** options are required for eCos, and others can be useful. This chapter gives a brief description of the required options as well as some recommended eCos-specific options. All other **GCC** options (described in the **GNUPro** manuals) are available.

Compiling a C Application

The following command lines demonstrate the *minimum* set of options required to compile and link an eCos program written in C.

NOTE   Remember that when this manual shows `gcc` you should type the full name of the cross compile,e.g. `mn10300-elf-gcc`, `mips-tx39-elf-gcc`, `powerpc-eabi-gcc`, `sparclite-elf-gcc`, `arm-elf-gcc`, `mips64vr4300-elf-gcc`, or `sh-elf-gcc`. When compiling for the synthetic Linux target, use the native gcc which must have the features required by eCos.

```
$ gcc -c  -IINSTALL_DIR/include file.c
$ gcc -o program file.o -LINSTALL_DIR/lib -Ttarget.ld -nostdlib
```

NOTE

■   Certain targets may require extra options, for example the SPARClite
    architectures require the option `-mcpu=sparclite`. Examine the
    BASE_DIR`/packages/targets` file or *BASE_DIR*`/examples/Makefile` or the
    "Global compiler flags" option (CYGBLD_GLOBAL_CFLAGS) in your
    generated eCos configuration) to see if any extra options are required, and if so,
    what they are.

The following command lines use some other options which are recommended
because they use the selective linking feature:

```
$ gcc -c  -IINSTALL_DIR/include -I. -ffunction-sections -fdata-sections -g -O2
file.c
$ gcc -o program file.o -ffunction-sections -fdata-sections -Wl,--gc-sections
-g -O2 -LINSTALL_DIR/lib -Ttarget.ld -nostdlib
```

Compiling a C++ Application

The following command lines demonstrate the *minimum* set of options required to
compile and link an eCos program written in C++.

NOTE

■   Remember that when this manual shows `g++` you should type the full name of the
    cross compiler: `mn10300-elf-g++`, `mips-tx39-elf-g++`, `powerpc-eabi-g++`,
    `sparclite-elf-g++`, `arm-elf-g++`, `mips64vr4300-elf-g++`, or `sh-elf-g++`.
    When compiling for the synthetic Linux target, use the native `g++` which must
    have the features required by eCos.

```
$ g++ -c  -IINSTALL_DIR/include -fno-rtti -fno-exceptions file.cxx
$ g++ -o program file.o -LINSTALL_DIR/lib -Ttarget.ld -nostdlib
```

NOTE

■   Certain targets may require extra options, for example the SPARClite
    architectures require the option `-mcpu=sparclite`. Examine the
    BASE_DIR`/packages/targets` file or *BASE_DIR*`/examples/Makefile` or the
    "Global compiler flags" option (CYGBLD_GLOBAL_CFLAGS) in your
    generated eCos configuration) to see if any extra options are required, and if so,
    what they are.

The following command lines use some other options which are recommended
because they use the selective linking feature:

```
$ g++ -c  -IINSTALL_DIR/include -I. -ffunction-sections -fdata-sections
-fno-rtti -fno-exceptions -fvtable-gc -finit-priority -g -O2 file.cxx
$ g++ -o program file.o -Wl,--gc-sections -g -O2 -LINSTALL_DIR/lib -Ttarget.ld
-nostdlib
```

13

Debugging Techniques

eCos applications and components can be debugged in traditional ways, with printing statements and debugger single-stepping, but there are situations in which these techniques cannot be used. One example of this is when a program is getting data at a high rate from a real-time source, and cannot be slowed down or interrupted.

eCos's infrastructure module provides a *tracing* formalism, allowing the kernel's tracing macros to be configured in many useful ways. eCos's kernel provides *instrumentation buffers* which also collect specific (configurable) data about the system's history and performance.

Tracing

To use eCos's tracing facilities you must first configure your system to use tracing. You should enable the *Asserts and Tracing* component (*CYGPKG_INFRA_DEBUG*) and the *Use tracing* component within it (*CYGDBG_USE_TRACING*). These options can be enabled with the **Configuration Tool** or by editing the file *BUILD_DIR*/pkgconf/infra.h manually.

You should then examine all the tracing-related options in the *Package: Infrastructure* chapter of the *eCos Reference Manual*. One useful set of configuration options are: *CYGDBG_INFRA_DEBUG_FUNCTION_REPORTS* and *CYGDBG_INFRA_DEBUG_TRACE_MESSAGE*, which are both enabled by default when tracing is enabled.

The following "Hello world with tracing" shows the output from running the hello world program (from the programming tutorial in *Getting Started with eCos*) that was built with tracing enabled:

Table 6: Hello world with tracing

```
$ mips-tx39-elf-run --board=jmr3904 hello
Hello, eCos world!
ASSERT FAIL: <2>cyg_trac.h          [ 623]
Cyg_TraceFunction_Report_::set_exitvoid()
exitvoid used in typed function
TRACE:<1>mlqueue.cxx [395]Cyg_ThreadQueue_Implementation::enqueue()
{{enter
TRACE:<1>mlqueue.cxx  [395]Cyg_ThreadQueue_Implementation::enqueue()
}}RETURNING UNSET!
TRACE:<1>mlqueue.cxx  [126]Cyg_Scheduler_Implementation::add_thread()
}}RETURNING UNSET!
TRACE:<1>thread.cxx    [654]Cyg_Thread::resume()
}}return void
TRACE:<1>cstartup.cxx   [160]cyg_iso_c_start()
}}return void
TRACE:<1>startup.cxx    [142]cyg_package_start()
}}return void
TRACE:<1>startup.cxx    [150]cyg_user_start()
{{enter
TRACE:<1>startup.cxx    [150]cyg_user_start()
(((void)))
TRACE:<1>startup.cxx    [153]cyg_user_start()
'This is the system default cyg_user_start()'
TRACE:<1>startup.cxx    [157]cyg_user_start()
}}return void
TRACE:<1>sched.cxx    [212]Cyg_Scheduler::start()
{{enter
TRACE:<1>mlqueue.cxx [102]Cyg_Scheduler_Implementation::schedule()
{{enter
TRACE:<1>mlqueue.cxx [437]Cyg_ThreadQueue_Implementation::highpri()
{{enter
TRACE:<1>mlqueue.cxx [437]Cyg_ThreadQueue_Implementation::highpri()
}}RETURNING UNSET!
TRACE:<1>mlqueue.cxx [102]Cyg_Scheduler_Implementation::schedule()
}}RETURNING UNSET!
TRACE:<2>intr.cxx    [450]Cyg_Interrupt::enable_interrupts()
{{enter
TRACE:<2>intr.cxx    [450]Cyg_Interrupt::enable_interrupts()
}}RETURNING UNSET!
TRACE:<2>thread.cxx   [69]Cyg_HardwareThread::thread_entry()
{{enter
TRACE:<2>cstartup.cxx   [127]invoke_main()
{{enter
TRACE:<2>cstartup.cxx   [127]invoke_main()
((argument is ignored))
TRACE:<2>dummyxxmain.cxx  [60]__main()
{{enter
TRACE:<2>dummyxxmain.cxx  [60]__main()
(((void)))
TRACE:<2>dummyxxmain.cxx  [63]__main()
'This is the system default __main()'
TRACE:<2>dummyxxmain.cxx  [67]__main()
}}return void
```

```
TRACE: <2>memcpy.c        [ 112] _memcpy()
{{enter
TRACE: <2>memcpy.c        [ 112] _memcpy()
((dst=80002804, src=BFC14E58, n=19))
TRACE: <2>memcpy.c        [ 164] _memcpy()
}}returning 80002804
TRACE: <2>cstartup.cxx    [ 137] invoke_main()
'main() has returned with code 0. Calling exit()'
TRACE: <2>exit.cxx       [ 71] __libc_exit()
{{enter
TRACE: <2>exit.cxx       [ 71] __libc_exit()
((status=0 ))
TRACE: <2>atexit.cxx    [ 84] cyg_libc_invoke_atexit_handlers()
{{enter
TRACE: <2>atexit.cxx    [ 84] cyg_libc_invoke_atexit_handlers()
(((void)))

Scheduler:

Lock:                  0
Current Thread:        <null>

Threads:

Idle Thread            pri =  31 state = R     id =   1
                       stack base = 800021F0 ptr = 80002510 size = 00000400
                       sleep reason NONE     wake reason NONE
                       queue = 80000C54      wait info = 00000000

<null>                 pri =   0 state = R     id =   2
                       stack base = 80002A48 ptr = 8000A968 size = 00008000
                       sleep reason NONE     wake reason NONE
                       queue = 80000BD8      wait info = 00000000
```

Kernel Instrumentation

Instrument buffers can be used to find out how many events of a given type happened in the kernel during execution of a program.

You can monitor a class of several types of events, or you can just look at individual events.

Examples of events that can be monitored are:

- scheduler events
- thread operations
- interrupts
- mutex operations
- binary semaphore operations

- counting semaphore operations
- clock ticks and interrupts

Examples of fine-grained scheduler event types are:

- scheduler lock
- scheduler unlock
- rescheduling
- time slicing

Information about the events is stored in an *event record*. The structure that defines this record has type **struct Instrument_Record**:

The list of records is stored in an array called `instrument_buffer`, which you can let the kernel provide or you can provide yourself by setting the configuration option *CYGVAR_KERNEL_INSTRUMENT_EXTERNAL_BUFFER*.

To write a program that examines the instrumentation buffers:

1. Enable instrumentation buffers in the eCos kernel configuration. The component macro is *CYGPKG_KERNEL_INSTRUMENT*.

2. To allocate the buffers yourself, enable the configuration option *CYGVAR_KERNEL_INSTRUMENT_EXTERNAL_BUFFER*.

3. Include the header file `cyg/kernel/instrmnt.h`.
```
#include <cyg/kernel/instrmnt.h>
```

4. The `Instrumentation_Record` structure is not published in the kernel header file. In the future there will be a cleaner mechanism to access it, but for now you should paste into your code in the following lines:
```
struct Instrument_Record
{
 CYG_WORD16 type; // record type
 CYG_WORD16 thread; // current thread id
 CYG_WORD timestamp; // 32 bit timestamp
 CYG_WORD arg1; // first arg
 CYG_WORD arg2; // second arg
};
```

5. Enable the events you want to record using **cyg_instrument_enable()**, and disable them later. Look at `cyg/kernel/instrmnt.h` and the examples below to see what events can be enabled.

6. Place the code you want to debug between the matching functions **cyg_instrument_enable()** and **cyg_instrument_disable()**.

7. Examine the buffer. For now you need to look at the data in there (the example program below shows how to do that), and future versions of eCos will include a host-side tool to help you understand the data.

Table 7: Using instrument buffers

This program is also provided in the `examples` directory.

```
/* this is a program which uses eCos instrumentation buffers; it needs
 to be linked with a kernel which was compiled with support for
 instrumentation */

#include <stdio.h>
#include <pkgconf/kernel.h>
#include <cyg/kernel/instrmnt.h>
#include <cyg/kernel/kapi.h>

#ifndef CYGVAR_KERNEL_INSTRUMENT_EXTERNAL_BUFFER
# error You must configure eCos with CYGVAR_KERNEL_INSTRUMENT_EXTERNAL_BUFFER
#endif

struct Instrument_Record
{
 CYG_WORD16 type; // record type
 CYG_WORD16 thread; // current thread id
 CYG_WORD timestamp; // 32 bit timestamp
 CYG_WORD arg1; // first arg
 CYG_WORD arg2; // second arg
};

struct Instrument_Record instrument_buffer[20];
cyg_uint32 instrument_buffer_size = 20;

int main(void)
{
 int i;

 cyg_instrument_enable(CYG_INSTRUMENT_CLASS_CLOCK, 0);
 cyg_instrument_enable(CYG_INSTRUMENT_CLASS_THREAD, 0);
 cyg_instrument_enable(CYG_INSTRUMENT_CLASS_ALARM, 0);

 printf("Program to play with instrumentation buffer\n");

 cyg_thread_delay(2);

 cyg_instrument_disable(CYG_INSTRUMENT_CLASS_CLOCK, 0);
 cyg_instrument_disable(CYG_INSTRUMENT_CLASS_THREAD, 0);
 cyg_instrument_disable(CYG_INSTRUMENT_CLASS_ALARM, 0);

 for (i = 0; i < instrument_buffer_size; ++i) {
 printf("Record %02d: type 0x%04x, thread %d, ",
              i, instrument_buffer[i].type, instrument_buffer[i].thread);
 printf("time %5d, arg1 0x%08x, arg2 0x%08x\n",
              instrument_buffer[i].timestamp, instrument_buffer[i].arg1,
              instrument_buffer[i].arg2);
 }
 return 0;
}
```

Here is how you could compile and run this program in the `examples` directory, using (for example) the MN10300 compiler:

```
$ make XCC=mn10300-elf-gcc   PKG_INSTALL_DIR=/tmp/ecos-work-mn10300/install
instrument-test
mn10300-elf-gcc -c -o instrument-test.o -g -Wall
-I/tmp/ecos-work-mn10300/install/include -ffunction-sections -fdata-sections
instrument-test.c
mn10300-elf-gcc -nostartfiles -L/tmp/ecos-work-mn10300/install/lib
-W1,--gc-sections -o instrument-test instrument-test.o -Ttarget.ld -nostdlib
$ mn10300-elf-run --board=stdeval1 instrument-test
```

Table 8: Instrument buffer output

Here is the output of the `instrument-test` program. Notice that in little over 2 seconds, and with very little activity, and with few event types enabled, it gathered 17 records. In larger programs it will be necessary to select very few event types for debugging.

```
Program to play with instrumentation buffer
Record 00: type 0x0207, thread 2, time  6057, arg1 0x48001cd8, arg2 0x00000002
Record 01: type 0x0202, thread 2, time  6153, arg1 0x48001cd8, arg2 0x00000000
Record 02: type 0x0904, thread 2, time  6358, arg1 0x48001d24, arg2 0x00000000
Record 03: type 0x0905, thread 2, time  6424, arg1 0x00000002, arg2 0x00000000
Record 04: type 0x0906, thread 2, time  6490, arg1 0x00000000, arg2 0x00000000
Record 05: type 0x0901, thread 2, time  6608, arg1 0x48009d74, arg2 0x48001d24
Record 06: type 0x0201, thread 2, time  6804, arg1 0x48001cd8, arg2 0x480013e0
Record 07: type 0x0803, thread 1, time    94, arg1 0x00000000, arg2 0x00000000
Record 08: type 0x0801, thread 1, time   361, arg1 0x00000000, arg2 0x00000000
Record 09: type 0x0802, thread 1, time   548, arg1 0x00000001, arg2 0x00000000
Record 10: type 0x0803, thread 1, time    94, arg1 0x00000000, arg2 0x00000000
Record 11: type 0x0801, thread 1, time   361, arg1 0x00000001, arg2 0x00000000
Record 12: type 0x0903, thread 1, time   513, arg1 0x48009d74, arg2 0x48001d24
Record 13: type 0x0208, thread 1, time   588, arg1 0x00000000, arg2 0x00000000
Record 14: type 0x0203, thread 1, time   697, arg1 0x48001cd8, arg2 0x480013e0
Record 15: type 0x0802, thread 1, time   946, arg1 0x00000002, arg2 0x00000000
Record 16: type 0x0201, thread 1, time  1083, arg1 0x480013e0, arg2 0x48001cd8
Record 17: type 0x0000, thread 0, time     0, arg1 0x00000000, arg2 0x00000000
Record 18: type 0x0000, thread 0, time     0, arg1 0x00000000, arg2 0x00000000
Record 19: type 0x0000, thread 0, time     0, arg1 0x00000000, arg2 0x00000000
```

Part III: Configuration and the Package Repository

The following chapters contain information on running ecosconfig (the command line tool that manipulates configurations and constructs build trees) and on managing a source repository across multiple versions of eCos.

14

Manual Configuration

eCos developers using a Windows NT host will generally use the graphical Configuration Tool for configuring an eCos system and building the target library. At present there is no equivalent to this tool available for developers using a UNIX host, so command line tools have to be used instead. These command line tools can also be used for batch operations on all platforms, for example as part of a nightly rebuild procedure.

In the current release of the system the command line tools do not provide exactly the same functionality as the graphical tool. Most importantly, there is no facility to resolve configuration conflicts interactively.

The eCos configuration system, both graphical and command line tools, are under constant development and enhancement. Developers should note that the procedures described may change considerably in future releases.

Directory Tree Structure

When building eCos there are three main directory trees to consider: the source tree, the build tree, and the install tree.

The source tree, also known as the component repository, is read-only. It is possible to use a single component repository for any number of different configurations, and it is also possible to share a component repository between multiple users by putting it on a network drive.

The build tree contains everything that is specific to a particular configuration, including header and other files that contain configuration data, and the object files that result from compiling the system sources for this configuration.

The install tree is usually located in the `install` subdirectory of the build tree. Once an eCos system has been built, the install tree contains all the files needed for application development including the header files and the target library. By making copies of the install tree after a build it is possible to separate application development and system configuration, which may be desirable for some organizations.

Creating the Build Tree

Generating a build tree is a non-trivial operation and should not be attempted manually. Instead, eCos is shipped with a tool called `ecosconfig` that should be used to create a build tree.

Usually `ecosconfig` will be run inside the build tree itself. If you are creating a new build tree then typically you will create a new empty directory using the `mkdir` command, `cd` into that directory, and then invoke `ecosconfig` to create a configuration. By default, the configuration is stored in a file `ecos.ecc` in the current directory. The configuration may be modified by editing this file directly. `ecosconfig` itself deals with a number of coarse-grained configuration options such as the target platform and the packages that should be used.

The `ecosconfig` tool is also used subsequently to generate a build tree for a configuration. Once a build tree exists, it is possible to run `ecosconfig` again inside the same build tree. This will be neccessary if your wish to change some of the configuration options.

`ecosconfig` does not generate the top-level directory of the build tree; you must do this yourself.

```
$ mkdir ecos-work
$ cd ecos-work
```

The next step is to run `ecosconfig`:

```
$ ecosconfig <qualifiers> <command>
```

ecosconfig qualifiers

The available command line qualifiers for `ecosconfig` are as follows. Multiple qualifiers may be used on the command line:

`--help`

Provides basic usage guidelines for the available commands and qualifiers. All other qualifiers and commands are ignored if --help is used.

`--config=<file>`

> Specifies an eCos configuration save file for use by the tool. By default, the file `ecos.ecc` in the current directory is used. Developers may prefer to use a common location for all their eCos configurations rather than keep the configuration information in the base of the build tree.

`--prefix=<dir>`

> Specifies an alternative location for the install tree. By default, the install tree resides inside the `install` directory in the build tree. Developers may prefer to locate the build tree in a temporary file hierarchy but keep the install tree in a more permanent location.

`--srcdir=<dir>`

> Specifies the location of the component repository. By default, the tool uses the location specified in the *ECOS_REPOSITORY* environment variable. Developers may prefer to use of this qualifier if they are working with more than one repository.

`--no-resolve`

> Disables the implicit resolution of conflicts while manipulating the configuration data. developers may prefer to reslove conflicts by editing the eCos configuration save file manually The check command implicitly uses --no-resolve.

`--ignore-errors`
> `-i`

> By default, ecosconfig will exit with an error code if the current configuration contains any conflicts, and it is not possible to generate or update a build tree for such configurations. This qualifier causes ecosconfig to ignore such problems, and hence it is possible to generate a build tree even if there are still conflicts. Of course, there are no guarantees that the resulting system will actually do anything useful.

`--verbose`
> `-v`

Display more information.

`-q`

Display less information.

The --config, --prefix and srcdir qualifiers can also be written with two arguments, for example:

```
  ecosconfig --srcdir <dir> ...
```

This simplifies filename completion with some shells.

ecosconfig commands

The available commands for `ecosconfig` are as follows:

`list`

> Lists the available packages, targets and templates as installed in the eCos repository. Aliases and package versions are also reported.

`new <target> [<template> [<version>]]`

> Creates a new eCos configuration for the specified target hardware and saves it. A software template may also be specified. By default, the template named 'default' is used. If the template version is not specified, the latest version is used.

`target <target>`

> Changes the target hardware selection for the eCos configuration. This has the effect of unloading packages supporting the target selected previously and loading the packages which support the new hardware. This command will be used typically when switching between a simulator and real hardware.

`template <template> [<version>]`

> Changes the template selection for the eCos configuration. This has the effect of unloading packages specified by the template selected previously and loading the packages specified by the new template. By default, the latest version of the specified template is used.

`remove <packages>`

> Removes the specified packages from the eCos configuration. This command will be used typically when the template on which a configuration is based contains packages which are not required.

`add <packages>`

> Adds the specified packages to the eCos configuration. This command will be used typically when the template on which a configuration is based does not contain all the packages which are required. For example, add-on packages provided by third parties will not be known to the standard templates so they will have to be added explicitly.

`version <version> <packages>`

> Selects the specified version of a number of packages in the eCos configuration. By default, the most recent version of each package is used. This command will be used typically when an older version of a package is required.

`check`

> Presents the following information concerning the current configuration:
>
> **1.** the selected target hardware

2. the selected template

3. additional packages

4. removed packages

5. the selected version of packages where this is not the most recent version

6. conflicts in the current configuration

The check command will never perform automatic conflict resolution, so there is no need to use --no-resolve.

resolve

Resolves conflicts identified in the current eCos configuration by invoking an inference capability. Resolved conflicts are reported, but not all conflicts may be resolvable. This command will be used typically following manual editing of the configuration.

export <file>

Exports a minimal eCos configurastion save file with the specified name. This file contains only those options which do not have their default value. Such files are used typically to transfer option values from one configuration to another.

import <file>

Imports a minimal eCos configuration save file with the specified name. The values of those options specified in the file are applied to the current configuration.

tree

Generates a build tree based on the current eCos configuration. This command will be used typically just before building eCos. Normally a build tree can only be generated if the configuration has no unresolved conflicts, but --ignore-errors can be used to override this.

Conflicts and Constraints

Configuration options are not completely independent. For example the C library's strtod() and atof() functions rely on the math library package to provide certain functionality. If the math library package is removed then the C library can no longer provide these functions. Each package describes constraints like these in CDL "requires" properties. If a constraint is not satisfied then the configuration contains a conflict. For any given conflict there can be several ways of resolving a conflict. For example, it would be possible to add the math library package back to the configuration, or to disable the strtod() and atof() functions.

The eCos configuration tools will report any conflicts in the current configuration. If there are any such conflicts then the configuration is usually unsafe and it makes no sense to build and run eCos in such circumstances, in fact any attempt at building eCos is likely to fail. In exceptional cases it is possible to override this by using e.g. the --ignore-errors qualifier with ecosconfig.

Many constraints are fairly simple in nature, and the configuration tools contain an inference engine which can resolve the associated conflicts automatically. For example, if the math library package is removed then the inference engine can resolve the resulting conflict by disabling the configuration option for strtod() and atof(). All such changes will be reported. Sometimes the inference engine cannot resolve a conflict, for example it is not allowed to override a change that has been made explicitly by the user. Sometimes it will find a solution which does not match the application's requirements.

A typical session involving conflicts would look something like this:

```
$ ecosconfig new <target>
```

This creates a new configuration with the default template. For most targets this will not result in any conflicts, because the default settings for the various options meet the requirements of the default template. For some targets there may be conflicts, and the inference engine would come into play.

```
$ ecosconfig remove libm
U CYGSEM_LIBC_STDIO_SCANF_FLOATING_POINT, new inferred value 0
U CYGFUN_LIBC_strtod, new inferred value 0
U CYGSEM_LIBC_STDIO_PRINTF_FLOATING_POINT, new inferred value 0
```

ecosconfig reports that this change caused three conflicts, all in the C library. The inference engine was able to resolve all the conflicts and update the relevant configuration options accordingly.

To suppress the inference engine --no-resolve can be used:

```
$ ecosconfig new <target>
$ ecosconfig --no-resolve remove libm
C CYGSEM_LIBC_STDIO_SCANF_FLOATING_POINT, "requires" constraint not
    satisfied: CYGPKG_LIBM
C CYGSEM_LIBC_STDIO_PRINTF_FLOATING_POINT, "requires" constraint
    not satisfied: CYGPKG_LIBM
C CYGFUN_LIBC_strtod, "requires" constraint not satisfied:
    CYGPKG_LIBM
```

Three unresolved conflicts are reported. The check command can be used to get the current state of the configuration, and the --verbose qualifier will provide additional information:

```
$ ecosconfig --srcdir /home/bartv/ecc/ecc --verbose check
Target: pid
Template: default
Removed:
    CYGPKG_LIBM
3 conflict(s):
```

```
    C CYGFUN_LIBC_strtod, "requires" constraint not satisfied:
    CYGPKG_LIBM
    Possible solution:
        CYGFUN_LIBC_strtod -> 0
        CYGSEM_LIBC_STDIO_SCANF_FLOATING_POINT -> 0
    C CYGSEM_LIBC_STDIO_PRINTF_FLOATING_POINT, "requires" constraint
    not satisfied: CYGPKG_LIBM
    Possible solution:
        CYGSEM_LIBC_STDIO_PRINTF_FLOATING_POINT -> 0
    C CYGSEM_LIBC_STDIO_SCANF_FLOATING_POINT, "requires" constraint not
    satisfied: CYGPKG_LIBM
    Possible solution:
        CYGSEM_LIBC_STDIO_SCANF_FLOATING_POINT -> 0
```

If the proposed solutions are acceptable, the resolve command can be used to apply them:

```
$ ecosconfig resolve
U CYGSEM_LIBC_STDIO_SCANF_FLOATING_POINT, new inferred value 0
U CYGFUN_LIBC_strtod, new inferred value 0
U CYGSEM_LIBC_STDIO_PRINTF_FLOATING_POINT, new inferred value 0
```

The current configuration is again conflict-free and it is possible to generate a build tree. The --quiet qualifier can be used to suppress the change messages, if desired.

When changing individual configuration options by editing the ecos.ecc file (as described below), the resulting system should be checked and any problems should be resolved. For example, if CYGFUN_LIBC_strtod is explicitly enabled in the savefile:

```
$ <edit ecos.ecc>
$ ecosconfig check
Target: pid
Template: default
Removed:
    CYGPKG_LIBM
1 conflict(s):
C CYGFUN_LIBC_strtod, "requires" constraint not satisfied:
CYGPKG_LIBM
$ ecosconfig resolve
C CYGFUN_LIBC_strtod, "requires" constraint not satisfied:
CYGPKG_LIBM
```

In this case the inference engine cannot resolve the conflict automatically because that would involve changing a user setting. Any attempt to generate a build tree will fail:

```
$ ecosconfig --srcdir /home/bartv/ecc/ecc tree
C CYGFUN_LIBC_strtod, "requires" constraint not satisfied:
CYGPKG_LIBM

Unable to generate build tree, this configuration still contains
conflicts.   Either resolve the conflicts or use --ignore-errors
```

It is still possible to generate a build tree:

```
$ ecosconfig --srcdir /home/bartv/ecc/ecc --ignore-errors tree
C CYGFUN_LIBC_strtod, "requires" constraint not satisfied:
CYGPKG_LIBM
$ make
```

In this case eCos will fail to build. In other cases of unresolved conflicts eCos may build, but may not run. In general all conflicts should be resolved by editing the ecos.ecc file, by letting the inference engine make appropriate changes, or by other means, before any attempt is made to build or run eCos.

Building the System

Once a build tree has been generated with `ecosconfig`, building eCos is straightforward:

```
$ make
```

The build tree contains the subdirectories, makefiles, and everything else that is needed to generate the default configuration for the selected architecture and platform. The only requirement is that the tools needed for that architecture, for example `powerpc-eabi-g++`, are available using the standard search path. If this is not the case then the `make` will fail with an error message. If you have a multiprocessor system then it may be more efficient to use:

$$\$ \text{ make } -j \; \boldsymbol{n}$$

where **n** is equal to the number of processors on your system.

Once the `make` process has completed, the install tree will contain the header files and the target library that are needed for application development.

It is also possible to build the system's test cases for the current configuration:

```
$ make tests
```

The resulting test executables will end up in a `tests` subdirectory of the install tree.

If disk space is scarce then it is possible to make the copy of the install tree for application development purposes, and then use:

```
$ make clean
```

The build tree will now use up a minimum of disk space — the bulk of what is left consists of configuration header files that you may have edited and hence should not be deleted automatically. However, it is possible to rebuild the system at any time without reinvoking `ecosconfig`, just by running `make` again.

Under exceptional circumstances it may be necessary to run `make clean` for other reasons, such as when a new release of the toolchain is installed. The toolchain includes a number of header files which are closely tied to the compiler, for example `limits.h`, and these header files are not and should not be duplicated by eCos. The makefiles perform header file dependency analysis, so that when a header file is changed all affected sources will be rebuilt during the next `make`. This is very useful when the configuration header files are changed, but it also means that a build tree containing information about the locations of header files must be rebuilt. If a new

version of the toolchain is installed and the old version is removed then this location information is no longer accurate, and `make` will complain that certain dependencies cannot be satisfied. Under such circumstances it is necessary to do a `make clean` first.

Packages

eCos is a component architecture. The system comes as a number of packages which can be enabled or disabled as required, and new packages can be added as they become available. Unfortunately, the packages are not completely independent: for example the μITRON compatibility package relies almost entirely on functionality provided by the kernel package, and it would not make sense to try to build μITRON if the kernel was disabled. The C library has fewer dependencies: some parts of the C library rely on kernel functionality, but it is possible to disable these parts and thus build a system that has the C library but no kernel. The `ecosconfig` tool has the capability of checking that all the dependencies are satisfied, but it may still be possible to produce configurations that will not build or (conceivably) that will build but not run. Developers should be aware of this and take appropriate care.

By default, `ecosconfig` will include all packages that are appropriate for the specified hardware in the configuration. The common HAL package and the eCos infrastructure must be present in every configuration. In addition, it is always necessary to have one architectural HAL package and one platform HAL package. Other packages are optional, and can be added or removed from a configuration as required.

The application may not require all of the packages; for example, it might not need the μITRON compatibility package, or the floating point support provided by the math library. There is a slight overhead when eCos is built because the packages will get compiled, and there is also a small disk space penalty. However, any unused facilities will get stripped out at link-time, so having redundant packages will not affect the final executable.

Coarse-grained Configuration

Coarse-grained configuration of an eCos system means making configuration changes using the `ecosconfig` tool. These changes include:

1. switching to different target hardware
2. switching to a different template
3. adding or removing a package
4. changing the version of a package

Whenever `ecosconfig` generates or updates an eCos configuration, it generates a configuration save file.

Suppose that the configuration was first created using the following command line:

```
$ ecosconfig new stdeval1
```

To change the target hardware to the Cogent CMA28x PowerPC board, the following command would be needed:

```
$ ecosconfig target cma28x
```

To switch to the PowerPC simulator instead:

```
$ ecosconfig target psim
```

As the hardware changes, hardware-related packages such as the HAL packages and device drivers will be added to and removed from the configuration as appropriate.

To remove any package from the current configuration, use the `remove` command:

```
$ ecosconfig remove uitron
```

You can disable multiple packages using multiple arguments, for example:

```
$ ecosconfig remove uitron libm
```

If this turns out to have been a mistake then you can reenable one or more packages with the `add` command:

```
$ ecosconfig add libm
```

Changing the desired version for a package is also straightforward:

```
$ ecosconfig version v1_3_x kernel
```

where x may be a one or two digit number.

It is necessary to regenerate the build tree and header files following any changes to the configuration before rebuilding eCos:

```
$ ecosconfig tree
```

Fine-grained Configuration

`ecosconfig` only provides coarse-grained control over the configuration: the hardware, the template and the packages that should be built. Unlike the Configuration Tool, `ecosconfig` does not provide any facilities for manipulating finer-grained configuration options such as how many priority levels the scheduler should support. There are hundreds of these options, and manipulating them by means of command line arguments would not be sensible.

In the current system fine-grained configuration options may be manipulated by manual editing of the configuration file. When a file has been edited in this way, the `ecosconfig` tool should be used to check the configuration for any conflicts which may have been introduced:

```
$ ecosconfig check
```

The `check` command will list all conflicts and will also rewrite the configuration file, propogating any changes which affect other options. The user may choose to resolve the conflicts either by re-editing the configuration file manually or by invoking the inference engine using the `resolve` command:

```
$ ecosconfig resolve
```

The `resolve` command will list all conflicts which can be resolved and save the resulting changes to the configuration.

It is necessary to regenerate the build tree and header files following any changes to the configuration before rebuilding eCos:

```
$ ecosconfig tree
```

All the configuration options and their descriptions are listed in the *eCos Reference Manual*.

Editing an eCos Savefile

The eCos configuration information is held in a single savefile, typically `ecos.ecc`, which can be generated by either the GUI configuration tool or by the command line *ecosconfig* tool. The file normally exists at the toplevel of the build tree. It is a text file, allowing the various configurations options to be edited inside a suitable text editor or by other programs or scripts, as well as in the GUI config tool.

An eCos savefile is actually a script in the *Tcl* programming language, so any modifications to the file need to preserve Tcl syntax. For most configuration options, any modifications will be trivial and there is no need to worry about Tcl syntax. For example, changing a 1 to a 0 to disable an option. For more complicated options, for example *CYGDAT_UITRON_TASK_EXTERNS ,* which involves some lines of C code, more care has to be taken. If an edited savefile is no longer a valid Tcl script then the configuration tools will be unable to read back the data for further processing, for example to generate a build tree. An outline of Tcl syntax is given below. One point worth noting here is that a line that begins with a "#" is usually a comment, and the bulk of an eCos savefile actually consists of such comments, to make it easier to edit.

Header

An eCos savefile begins with a header, which typically looks something like this:

```
# eCos saved configuration
# ---- commands ---------------------------------------------------
# This section contains information about the savefile format.
# It should not be edited. Any modifications made to this section
# may make it impossible for the configuration tools to read
# the savefile.
```

```
cdl_savefile_version 1;
cdl_savefile_command cdl_savefile_version {};
cdl_savefile_command cdl_savefile_command {};
cdl_savefile_command
cdl_configuration { description hardware template package };
cdl_savefile_command cdl_package { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_component { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_option { value_source user_value wizard_value inferred_value };
cdl_savefile_command cdl_interface { value_source user_value wizard_value inferred_value };
```

This section of the savefile is intended for use by the configuration system, and should not be edited. If this section is edited then the various configuration tools may no longer be able to read in the modified savefile.

Toplevel Section

The header is followed by a section that defines the configuration as a whole. A typical example would be:

```
# ---- toplevel ---------------------------------------------------
# This section defines the toplevel configuration object. The only
# values that can be changed are the name of the configuration and
# the description field. It is not possible to modify the target,
# the template or the set of packages simply by editing the lines
# below because these changes have wide-ranging effects. Instead
# the appropriate tools should be used to make such modifications.

cdl_configuration eCos {
description "" ;

# These fields should not be modified.
hardware    pid ;
template    uitron ;
package -hardware CYGPKG_HAL_ARM current ;
package -hardware CYGPKG_HAL_ARM_PID current ;
package -hardware CYGPKG_IO_SERIAL current ;
package -template CYGPKG_HAL current ;
package -template CYGPKG_IO current ;
package -template CYGPKG_INFRA current ;
package -template CYGPKG_KERNEL current ;
package -template CYGPKG_UITRON current ;
package -template CYGPKG_LIBC current ;
package -template CYGPKG_LIBM current ;
package -template CYGPKG_DEVICES_WALLCLOCK current ;
package -template CYGPKG_ERROR current ;
};
```

This section allows the configuration tools to reload the various packages that make up the configuration. Most of the information should not be edited. If it is necessary to add a new package or to remove an existing one then the appropriate tools should be used for this, for example:

```
$ ecosconfig remove CYGPKG_LIBM
```

There are two fields which can be edited. Configurations have a name; in this case eCos. They can also have a description, which is some arbitrary text. The configuration tools do not make use of these fields, they exist so that users can store additional information about a configuration.

Conflicts Section

The toplevel section is followed by details of all the conflicts (if any) in the configuration, for example:

```
# ---- conflicts ------------------------------------------------------
# There are 2 conflicts.
#
# option CYGNUM_LIBC_TIME_DST_DEFAULT_OFFSET
#   Property LegalValues
#   Illegal current value 100000
#   Legal values are: -90000 to 90000
#
# option CYGSEM_LIBC_TIME_CLOCK_WORKING
#   Property Requires
#   Requires constraint not satisfied: CYGFUN_KERNEL_THREADS_TIMER
```

When editing a configuration you may end up with something that is invalid. Any problems in the configuration will be reported in the conflicts section. In this case there are two conflicts. The option *CYGNUM_LIBC_TIME_DST_DEFAULT_OFFSET* has been given an illegal value: typically this would be fixed by searching for the definition of that option later on in the savefile and modifying the value. The second conflict is more interesting, an unsatisfied *requires* constraint. Configuration options are not independent: disabling some functionality in, say, the kernel, can have an impact elsewhere; in this case the C library. The various dependencies between the options are specified by the component developers and checked by the configuration system. In this case there are two obvious ways in which the conflict could be resolved: re-enabling *CYGFUN_KERNEL_THREADS_TIMER*, or disabling *CYGSEM_LIBC_TIME_CLOCK_WORKING*. Both of these options will be listed later on in the file.

Some care has to be taken when modifying configuration options, to avoid introducing new conflict. For instance it is possible that there might be other options in the system which have a dependency on *CYGSEM_LIBC_TIME_CLOCK_WORKING*, so disabling that option may not be the best way to resolve the conflict. Details of all such dependencies are provided in the appropriate places in the savefile.

It is not absolutely required that a configuration be conflict-free before generating a build tree and building eCos. It is up to the developers of each component to decide what would happen if an attempt is made to build eCos while there are still conflicts.

In serious cases there is likely to be a compile-time failure, or possibly a link-time failure. In less serious cases the system may build happily and the application can be linked with the resulting library, but the component may not quite function as intended - although it may still be good enough for the specific needs of the application. It is also possible that everything builds and links, but once in a while the system will unaccountably crash. Using a configuration that still has conflicts is done entirely at the user's risk.

Data Section

The bulk of the savefile lists the various packages, components, and options, including their values and the various dependencies. A number of global options come first, especially those related to the build process such as compiler flags. These are followed by the various packages, and the components and options within those packages, in order.

Packages, components and options are organized in a hierarchy. If a particular component is disabled then all options and sub-components below it will be inactive: any changes made to these will have no effect. The savefile contains information about the hierarchy in the form of comments, for example:

```
cdl_package CYGPKG_KERNEL ...
# >
cdl_component CYGPKG_KERNEL_EXCEPTIONS ...
# >
cdl_option CYGSEM_KERNEL_EXCEPTIONS_DECODE ...
cdl_option CYGSEM_KERNEL_EXCEPTIONS_GLOBAL ...
# <
cdl_component CYGPKG_KERNEL_SCHED ...
# >
cdl_option CYGSEM_KERNEL_SCHED_MLQUEUE ...
cdl_option CYGSEM_KERNEL_SCHED_BITMAP ...
# <
# <
```

This corresponds to the following hierarchy:

```
 CYGPKG_KERNEL
   CYGPKG_KERNEL_EXCEPTIONS
     CYGSEM_KERNEL_EXCEPTIONS_DECODE
     CYGSEM_KERNEL_EXCEPTIONS_GLOBAL
   CYGPKG_KERNEL_SCHED
     CYGSEM_KERNEL_SCHED_MLQUEUE
     CYGSEM_KERNEL_SCHED_BITMAP
```

Providing the hierarchy information in this way allows programs or scripts to analyze the savefile and readily determine the hierarchy. It could also be used by a sufficiently powerful editor to support structured editing of eCos savefiles. The information is not used by the configuration tools themselves since they obtain the hierarchy from the original CDL scripts.

Each configurable entity is preceded by a comment, of the following form:

```
# Kernel schedulers
# doc: ref/ecos-ref/ecos-kernel-overview.html#THE-SCHEDULER
# The eCos kernel provides a choice of schedulers. In addition
# there are a number of configuration options to control the
# detailed behaviour of these schedulers.
cdl_component CYGPKG_KERNEL_SCHED {
...
};
```

This provides a short textual alias `Kernel schedulers` for the component. If online documentation is available for the configurable entity then this will come next. Finally there is a short description of the entity as a whole. All this information is provided by the component developers.

Each configurable entity takes the form:

```
<type> <name> {
      <data>
};
```

Configurable entities may not be active. This can be either because the parent is disabled or inactive, or because there are one or more *active_if* properties. Modifying the value of an inactive entity has no effect on the configuration, so this information is provided first:

```
cdl_option CYGSEM_KERNEL_EXCEPTIONS_DECODE {
# This option is not active
# The parent CYGPKG_KERNEL_EXCEPTIONS is disabled
...
};

...

cdl_option CYGIMP_IDLE_THREAD_YIELD {
# This option is not active
# ActiveIf constraint: (CYGNUM_KERNEL_SCHED_PRIORITIES == 1)
#     CYGNUM_KERNEL_SCHED_PRIORITIES == 32
#   --> 0
...
};
```

For *CYGIMP_IDLE_THREAD_YIELD* the savefile lists the expression that must be satisfied if the option is to be active, followed by the current value of all entities that are referenced in the expression, and finally the result of evaluating that expression.

Not all options are directly modifiable in the savefile. First, the value of packages (which is the version of that package loaded into the configuration) cannot be modified here.

```
cdl_package CYGPKG_KERNEL {
# Packages cannot be added or removed, nor can their version be changed,
# simply by editing their value. Instead the appropriate configuration
# should be used to perform these actions.     .
..;
}
```

The version of a package can be changed using e.g.:

```
  $ ecosconfig version 1.3 CYGPKG_KERNEL
```

Even though a package's value cannot be modified here, it is still important for the savefile to contain the details. In particular packages may impose constraints on other configurable entities and may be referenced by other configurable entities. Also it would be difficult to understand or extract the configuration's hierarchy if the packages were not listed in the appropriate places in the savefile.

Some components (or, conceivably, options) do not have any associated data. Typically they serve only to introduce another level in the hierarchy, which can be useful in the context of the GUI configuration tool.

```
cdl_component CYGPKG_HAL_COMMON_INTERRUPTS {
# There is no associated value.
};
```

Other components or options have a calculated value. These are not user-modifiable, but typically the value will depend on other options which can be modified. Such calculated options can be useful when controlling what gets built or what ends up in the generated configuration header files. A calculated value may also effect other parts of the configuration, for instance, via a *requires* constraint.

```
cdl_option BUFSIZ {
# Calculated value: CYGSEM_LIBC_STDIO_WANT_BUFFERED_IO ? CYGNUM_LIBC_STDIO_BUFSIZE : 0
#     CYGSEM_LIBC_STDIO_WANT_BUFFERED_IO == 1
#     CYGNUM_LIBC_STDIO_BUFSIZE == 256
# Current_value: 256
};
```

A special type of calculated value is the *interface*. The value of an interface is the number of active and enabled options which *implement* that interface. Again the value of an interface cannot be modified directly; only by modifying the options which implement the interface. However, an interface can be referenced by other parts of the configuration.

```
cdl_interface CYGINT_KERNEL_SCHEDULER {
# Implemented by CYGSEM_KERNEL_SCHED_MLQUEUE, active, enabled
# Implemented by CYGSEM_KERNEL_SCHED_BITMAP, active, disabled
# This value cannot be modified here.
# Current_value: 1
# Requires: 1 == CYGINT_KERNEL_SCHEDULER
#     CYGINT_KERNEL_SCHEDULER == 1
#   --> 1

# The following properties are affected by this value
# interface CYGINT_KERNEL_SCHEDULER
#     Requires: 1 == CYGINT_KERNEL_SCHEDULER
};
```

If the configurable entity is modifiable then there will be lines like the following:

```
cdl_option CYGSEM_KERNEL_SCHED_MLQUEUE {
...
# Flavor: bool
# No user value, uncomment the following line to provide one.
# user_value 1
# value_source default
# Default value: 1
...
};
```

Configurable entities can have one of four different flavors: none, bool, data and booldata. Flavor none indicates that there is no data associated with the entity, typically it just acts as a placeholder in the overall hierarchy. Flavor bool is the most common, it is a simple yes-or-no choice. Flavor data is for more complicated configuration choices, for instance the size of an array or the name of a device. Flavor booldata is a combination of bool and data: the option can be enabled or disabled, and there is some additional data associated with the option as well.

In the above example the user has not modified this particular option, as indicated by the comment and by the commented-out `user_value` line. To disable this option the file should be edited to:

```
cdl_option CYGSEM_KERNEL_SCHED_MLQUEUE {
...
# Flavor: bool
# No user value, uncomment the following line to provide one.
user_value 0
# value_source default
# Default value: 1
...
}
```

The comment preceding the `user_value 0` line can be removed if desired, otherwise it will be removed automatically the next time the file is read and updated by the configuration tools.

Much the same process should be used for options with the data flavor, for example:

```
cdl_option CYGNUM_LIBC_TIME_DST_DEFAULT_OFFSET {
# Flavor: data
# No user value, uncomment the following line to provide one.
# user_value 3600
# value_source default
# Default value: 3600
# Legal values: -90000 to 90000
};
```

can be changed to:

```
cdl_option CYGNUM_LIBC_TIME_DST_DEFAULT_OFFSET {
# Flavor: data
user_value 7200
# value_source default
# Default value: 3600
# Legal values: -90000 to 90000 };
```

Note that the original text provides the default value in the `user_value` comment, on the assumption that the desired new value is likely to be similar to the default value. The `value_source` comment does not need to be updated, it will be fixed up if the savefile is fed back into the configuration system and regenerated.

For options with the booldata flavor, the `user_value` line needs take two arguments. The first argument is for the boolean part, the second for the data part. For example:

```
cdl_component CYGNUM_LIBM_COMPATIBILITY {
# Flavor: booldata
# No user value, uncomment the following line to provide one.
# user_value 1 POSIX
# value_source default
# Default value: 1 POSIX
# Legal values:  "POSIX" "IEEE" "XOPEN" "SVID"
...
};
```

could be changed to:

```
cdl_component CYGNUM_LIBM_COMPATIBILITY {
# Flavor: booldata
user_value 1 IEEE
# value_source default
# Default value: 1 POSIX
# Legal values:  "POSIX" "IEEE" "XOPEN" "SVID"
...
```

```
};
```

or alternatively, if the whole component should be disabled, to:

```
cdl_component CYGNUM_LIBM_COMPATIBILITY {
# Flavor: booldata
user_value 0 POSIX
# value_source default
# Default value: 1 POSIX
# Legal values:  "POSIX" "IEEE" "XOPEN" "SVID"
...
};
```

Some options take values that span multiple lines. An example would be:

```
cdl_option CYGDAT_UITRON_MEMPOOLVAR_INITIALIZERS {
# Flavor: data
# No user value, uncomment the following line to provide one.
# user_value \
# "CYG_UIT_MEMPOOLVAR( vpool1, 2000 ), \\
#  CYG_UIT_MEMPOOLVAR( vpool2, 2000 ), \\
#  CYG_UIT_MEMPOOLVAR( vpool3, 2000 ),"
# value_source default
# Default value: \
#     "CYG_UIT_MEMPOOLVAR( vpool1, 2000 ), \\
#       CYG_UIT_MEMPOOLVAR( vpool2, 2000 ), \\
#       CYG_UIT_MEMPOOLVAR( vpool3, 2000 ),"
};
```

Setting a user value for this option involves uncommenting and modifying all relevant lines, for example:

```
cdl_option CYGDAT_UITRON_MEMPOOLVAR_INITIALIZERS {
# Flavor: data
user_value \
"CYG_UIT_MEMPOOLVAR( vpool1, 4000 ), \\
CYG_UIT_MEMPOOLVAR( vpool2, 4000 ),"
# value_source default
# Default value: \
#     "CYG_UIT_MEMPOOLVAR( vpool1, 2000 ), \\
#       CYG_UIT_MEMPOOLVAR( vpool2, 2000 ), \\
#       CYG_UIT_MEMPOOLVAR( vpool3, 2000 ),"
};
```

In such cases appropriate care has to be taken to preserve Tcl syntax, as discussed below.

The configuration system has the ability to keep track of several different values for any given option. All options start off with a default value, in other words their value source is set to `default`. If a configuration involves conflicts and the configuration system's inference engine is allowed to resolve these automatically then it may provide an `inferred` value instead, for example:

```
cdl_option CYGMFN_KERNEL_SYNCH_CONDVAR_TIMED_WAIT {
# Flavor: bool
# No user value, uncomment the following line to provide one.
# user_value 1
# The inferred value should not be edited directly.
inferred_value 0
# value_source inferred
# Default value: 1
...
};
```

Inferred values are calculated by the configuration system and should not be edited by the user. If the inferred value is not correct then a user value should be substituted instead:

```
cdl_option CYGMFN_KERNEL_SYNCH_CONDVAR_TIMED_WAIT {
# Flavor: bool
user_value 1
# The inferred value should not be edited directly.
inferred_value 0
# value_source inferred
# Default value: 1
...
};
```

The inference engine will not override a user value with an inferred one. Making a change like this may well re-introduce a conflict, since the inferred value was only calculated to resolve a conflict. Another run of the inference engine may find a different and more acceptable way of resolving the conflict, but this is not guaranteed and it may be up to the user to examine the various dependencies and work out an acceptable solution.

Inferred values are listed in the savefile because the exact inferred value may depend on the order in which changes were made and conflicts were resolved. If the inferred values were absent then it is possible that reloading a savefile would not exactly restore the configuration. Default values on the other hand are entirely deterministic so there is no actual need for the values to be listed in the savefile. However, the default value can be very useful information so it is provided in a comment.

Occasionally the user will want to do some experimentation, and temporarily switch an option from a user value back to a default or inferred one to see what the effect would be. This could be achieved by simply commenting out the user value. For instance, if the current savefile contains:

```
cdl_option CYGMFN_KERNEL_SYNCH_CONDVAR_TIMED_WAIT {
# Flavor: bool
user_value 1
# The inferred value should not be edited directly.
inferred_value 0
# value_source user
```

```
# Default value: 1
...
};
```

then the inferred value could be restored by commenting out or removing the `user_value` line:

```
cdl_option CYGMFN_KERNEL_SYNCH_CONDVAR_TIMED_WAIT {
# Flavor: bool
# user_value 1
# The inferred value should not be edited directly.
inferred_value 0
# value_source user
# Default value: 1
...
};
```

This is fine for simple values. However if the value is complicated then there is a problem: commenting out the `user_value` line or lines means that the user value becomes invisible to the configuration system, so if the savefile is loaded and then regenerated the information will be lost. An alternative approach is to keep the `user_value` but explicitly set the `value_source` line, for example:

```
cdl_option CYGMFN_KERNEL_SYNCH_CONDVAR_TIMED_WAIT {
# Flavor: bool
user_value 1
# The inferred value should not be edited directly.
inferred_value 0
value_source inferred
# Default value: 1
...
};
```

In this case the configuration system will use the inferred value for the purposes of dependency analysis etc., even though a user value is present. To restore the user value the `value_source` line can be commented out again. If there is no explicit `value_source` then the configuration system will just use the highest priority one: the user value if it exists; otherwise the inferred value if it exists; otherwise the default value which always exists.

The default value for an option can be a simple constant, or it can be an expression involving other options. In the latter case the expression will be listed, together with the values for all options referenced in the expression and the current result of evaluating that expression. This is for informational purposes only, the default value is always recalculated deterministically when loading in a savefile.

```
cdl_option CYGBLD_GLOBAL_COMMAND_PREFIX {
# Flavor: data
# No user value, uncomment the following line to provide one.
```

```
# user_value arm-elf
# value_source default
# Default value:  CYGHWR_THUMB ? "thumb-elf" : "arm-elf"
#      CYGHWR_THUMB == 0
#   --> arm-elf
};
```

For options with the data or booldata flavor, there are likely to be constraints on the possible values. If the value is supposed to be a number in a given range and a user value of "`hello world`" is provided instead then there are likely to be compile-time failures. Component developers can specify constraints on the legal values, and these will be listed in the savefile.

```
cdl_option X_TLOSS {
# Flavor: data
# No user value, uncomment the following line to provide one.
# user_value 1.41484755040569E+16
# value_source default
# Default value: 1.41484755040569E+16
# Legal values: 1 to 1e308
};

cdl_component CYGNUM_LIBM_COMPATIBILITY {
# Flavor: booldata
# No user value, uncomment the following line to provide one.
# user_value 1 POSIX
# value_source default
# Default value: 1 POSIX
# Legal values:  "POSIX" "IEEE" "XOPEN" "SVID"
...
};
```

In some cases the legal values list may be an expression involving other options. If so then the current values of the referenced options will be listed, together with the result of evaluating the list expression, just as for default value expressions.

If an illegal value is provided then this will result in a conflict, listed in the conflicts section of the savefile. For more complicated options a simple legal values list is not sufficient to allow the current value to be validated, and the configuration system will be unable to flag conflicts. This issue will be addressed in future releases of the configuration system.

Following the value-related fields for a given option, any *requires* constraints belonging to this option will be listed. These constraints are only effective if the option is active and, for bool and booldata flavors, enabled. If some aspect of eCos functionality is inactive or disabled then it cannot impose any constraints on the rest of the system. As usual, the full expression will be listed followed by the current values of all options that are referenced and the result of evaluating the expression:

```
cdl_option CYGSEM_LIBC_TIME_TIME_WORKING {
```

```
...
# Requires: CYGPKG_DEVICES_WALLCLOCK
#     CYGPKG_DEVICES_WALLCLOCK == current
#   --> 1
};
```

When modifying the value of an option it is useful to know not only what constraints the option imposes on the rest of the system but also what other options in the system depend in some way on this one. The savefile provides this information:

```
cdl_option CYGFUN_KERNEL_THREADS_TIMER {
...
# The following properties are affected by this value
# option CYGMFN_KERNEL_SYNCH_CONDVAR_TIMED_WAIT
#     Requires: CYGFUN_KERNEL_THREADS_TIMER
# option CYGIMP_UITRON_STRICT_CONFORMANCE
#     Requires: CYGFUN_KERNEL_THREADS_TIMER
# option CYGSEM_LIBC_TIME_CLOCK_WORKING
#     Requires: CYGFUN_KERNEL_THREADS_TIMER
};
```

Tcl Syntax

eCos savefiles are implemented as Tcl scripts, and are read in by running the data through a standard Tcl interpreter that has been extended with a small number of additional commands such as `cdl_option` and `cdl_configuration`. In many cases this is an implementation detail that can be safely ignored while editing a savefile: simply replacing a `1` with a `0` to disable some functionality is not going to affect whether or not the savefile is still a valid Tcl script and can be processed by a Tcl interpreter. However, there are more complicated cases where an understanding of Tcl syntax is at least desirable, for example:

```
cdl_option CYGDAT_UITRON_MEMPOOLVAR_EXTERNS {
    # Flavor: data
    user_value \
     "static char vpool1\[ 2000 \], \\
    vpool2\[ 2000 \], \\
     vpool3\[ 2000 \];"
# value_source default
# Default value: \
    #     "static char vpool1\[ 2000 \], \\
    #      vpool2\[ 2000 \], \\
    #      vpool3\[ 2000 \];"
};
```

The backslash at the end of the `user_value` line is processed by the Tcl interpreter as a line continuation character. The quote marks around the user data are also interpreted by the Tcl interpreter and serve to turn the entire data field into a single argument. The backslashes preceding the opening and closing square brackets prevent

the Tcl interpreter from treating these characters specially, otherwise there would be an attempt at *command substitution* as described below. The double backslashes at the end of each line of the data will be turned into a single backslash by the Tcl interpreter, rather than escaping the newline character, so that the actual data seen by the configuration system is:

```
static char vpool1[ 2000 ], \
       vpool2[ 2000 ], \
       vpool3[ 2000 ];
```

This is of course the data that should end up in the µITRON configuration header file. The opening and closing braces surrounding the entire body of the option data are also significant and cause this body to be passed as a single argument to the `cdl_option` command. The closing semicolon is optional in this example, but provides a small amount of additional robustness if the savefile is edited such that it is no longer a valid Tcl script. If the data contained any `$` characters then these would have to be treated specially as well, via a backslash escape.

In spite of what all the above might seem to suggest, Tcl 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 behaviour 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
```

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 eCos savefiles. For instance, consider the following configuration option:

```
cdl_option CYGDAT_LIBC_STDIO_DEFAULT_CONSOLE {
# Flavor: data
# No user value, uncomment the following line to provide one.
# user_value "\"/dev/ttydiag\""
# value_source default
# Default value: "\"/dev/ttydiag\""
};
```

The desired value of the configuration option should be a valid C string, complete with quote characters. If the savefile was edited to:

```
cdl_option CYGDAT_LIBC_STDIO_DEFAULT_CONSOLE {
# Flavor: data
user_value "/dev/ttydiag"
# value_source default
# Default value: "\"/dev/ttydiag\""
};
```

then the Tcl interpreter would remove the quote marks when the savefile is read back in, so the option's value would not have any quote marks and would not be a valid C string. The configuration system is not yet able to perform the required validation so the following `#define` would be generated in the configuration header file:

```
  #define CYGDAT_LIBC_STDIO_DEFAULT_CONSOLE /dev/ttydiag
```

This is likely to cause a compile-time failure when building eCos.

A quoted argument continues until the closing quote character is encountered, which means that it can span multiple lines. This can also be encountered in eCos savefiles, for instance, in the *CYGDAT_UITRON_MEMPOOLVAR_EXTERNS* example mentioned earlier. Newline or semicolon characters do not terminate the current command in such cases.

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
          "
```

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 54, and then the Tcl interpreter will execute `puts "The answer is 54"`. It should be noted that the interpreter contains 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 line substitution is very unlikely to prove useful in the context of an eCos savefile, but it is part of the Tcl language and hence cannot be easily suppressed while reading in a savefile. As a result care has to be taken when savefile data involves square brackets. Consider the following:

```
cdl_option CYGDAT_UITRON_MEMPOOLFIXED_EXTERNS {
     ...
     user_value \
"static char fpool1[ 2000 ],
fpool2[ 2000 ];"
     ...
};
```

The Tcl interpreter will interpret the square brackets as an attempt at command substitution and hence it will attempt to execute the command `2000` with no arguments. No such command is defined by the Tcl language or by the savefile-related extensions provided by the configuration system, so this will result in an error when an attempt is made to read back the savefile. Instead it is necessary to backslash-escape the square brackets and thus suppress command substitution:

```
cdl_option CYGDAT_UITRON_MEMPOOLFIXED_EXTERNS {
     ...
     user_value \
"static char fpool1\[ 2000 \],
fpool2\[ 2000 \];"
     ...
};
```

Similarly 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 very unlikely to prove useful in the context of an eCos savefile. Should it be necessary to have a $ character in configuration data then again a backslash escape needs to be used.

```
cdl_option FOODAT_MONITOR_PROMPT {
     ...
     user_value "\$ "
     ...
};
```

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). So, for example, the *CYGDAT_UITRON_MEMPOOL_EXTERNFIXED_EXTERNS* value could be written as:

```
cdl_option CYGDAT_UITRON_MEMPOOLFIXED_EXTERNS {
     ...
     user_value \
{static char fpool1[ 2000 ],
fpool2[ 2000 ];}
     ...
};
```

The configuration system does not use this when generating savefiles because for simple edits of a savefile by inexperienced users the use of brace characters is likely to be a little bit more confusing than the use of quotes.

At this stage it is worth noting that the basic format of each configuration option in the savefile makes use of braces:

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

The configuration system extends the Tcl language with a small number of additional commands such as `cdl_option`. These commands take two arguments, a name and a body, where the body consists of all the text between the braces. First a check is made that the specified option is actually present in the configuration. Then the body is executed in a recursive invocation of the Tcl interpreter, this time with additional commands such as `user_value` and `value_source`. If, after editing, the braces are not correctly matched up then the savefile will no longer be a valid Tcl script and errors will be reported when the savefile is loaded again.

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 behaviour. Consider the following:

```
cdl_option WHATEVER {
     # This is a comment }
     user_value 42
     ...
}
```

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. Toplevel savefile processing then resumes, and the next command that is encountered is `user_value`. Since the relevant savefile code 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. Fortunately this sequence of events is very unlikely to occur when editing generated savefiles.

This should be sufficient information about Tcl to allow for safe editing of eCos savefiles. Further information is available from a wide variety of sources, for example the book *Tcl and the Tk Toolkit* by John K Ousterhout.

Editing the Sources

For many users, controlling the packages and manipulating the available configuration options will be sufficient to create an embedded operating system that meets the application's requirements. However, since eCos is shipped entirely in source form, it is possible to go further when necessary: you can edit the eCos sources themselves. This requires some understanding of the way the eCos build system works.

The most obvious place to edit the source code is directly in the component repository. For example, you could edit the file `kernel/v1_3_x/src/sync/mutex.cxx` to change the way kernel mutexes work, or possibly just to add some extra diagnostics or assertions. Once the file has been edited, it is possible to invoke `make` at the top level of the build tree and the target library will be rebuilt as required. A small optimization is possible: the build tree is largely a mirror of the component repository, so it too will contain a subdirectory `kernel/v1_3_x`; if `make` is invoked in this directory then it will only check for changes to the kernel sources, which is a bit more efficient than checking for changes throughout the component repository.

Editing a file in the component repository is fine if this tree is used for only one eCos configuration. If the repository is used for several different configurations, however, and especially if it is shared by multiple users, then making what may be experimental changes to the master sources would be a bad idea. The build system provides an alternative. It is possible to make a copy of the file in the build tree, in other words copy `mutex.cxx` from the `kernel/v1_3_x/src/sync` directory in the component repository to `kernel/v1_3_x/src/sync` in the build tree, and edit the file in the build tree. When `make` is invoked it will pick up local copies of any of the sources in preference to the master versions in the component repository. Once you have finished modifying the eCos sources you can install the final version back in the component repository. If the changes were temporary in nature and only served to aid the debugging process, then you can discard the modified version of the sources.

The situation is slightly more complicated for the header files that a package may export, such as the C library's `stdio.h` header file, which can be found in the directory `language/c/libc/v1_3_x/include`. If such a header file is changed, either directly in the component repository or after copying it to the build tree, then `make` must be invoked at the top level of the build tree. In cases like this it is not safe to rebuild just the C library because other packages may depend on the contents of `stdio.h`.

Modifying the Memory Layout

Each eCos platform package is supplied with linker script fragments which describe the location of memory regions on the evaluation board and the location of memory sections within these regions. The correct linker script fragment is selected and included in the eCos linker script `target.ld` when eCos is built.

It is not necessary to modify the default memory layouts in order to start development with eCos. However, it will be necessary to edit a linker script fragment when the memory map of the evaluation board is changed. For example, if additional memory is added, the linker must be notified that the new memory is available for use. As a

minimum, this would involve modifying the length of the corresponding memory region. Where the available memory is non-contiguous, it may be necessary to declare a new memory region and reassign certain linker output sections to the new region.

Linker script fragments and memory layout header files should be edited within the eCos install tree. They are located at `include/pkgconf/mlt_*.*`. Where multiple start-up types are in use, it will be necessary to edit multiple linker script fragments and header files. The information provided in the header file and the corresponding linker script fragment must always match. A typical linker script fragment is shown below:

Table 9: eCos linker script fragment

```
MEMORY
{
 rom : ORIGIN = 0x40000000, LENGTH = 0x80000
 ram : ORIGIN = 0x48000000, LENGTH = 0x200000
}

SECTIONS
{
 SECTIONS_BEGIN
 SECTION_rom_vectors (rom, 0x40000000, LMA_EQ_VMA)
 SECTION_text (rom, ALIGN (0x1), LMA_EQ_VMA)
 SECTION_fini (rom, ALIGN (0x1), LMA_EQ_VMA)
 SECTION_rodata (rom, ALIGN (0x1), LMA_EQ_VMA)
 SECTION_rodata1 (rom, ALIGN (0x1), LMA_EQ_VMA)
 SECTION_fixup (rom, ALIGN (0x1), LMA_EQ_VMA)
 SECTION_gcc_except_table (rom, ALIGN (0x1), LMA_EQ_VMA)
 SECTION_data (ram, 0x48000000, FOLLOWING (.gcc_except_table))
 SECTION_bss (ram, ALIGN (0x4), LMA_EQ_VMA)
 SECTIONS_END
}
```

The file consists of two blocks, the MEMORY block contains lines describing the address (ORIGIN) and the size (LENGTH) of each memory region. The MEMORY block is followed by the SECTIONS block which contains lines describing each of the linker output sections. Each section is represented by a macro call. The arguments of these macros are ordered as follows:

1. The memory region in which the section will finally reside.

2. The final address (VMA) of the section. This is expressed using one of the following forms:

n

at the absolute address specified by the unsigned integer *n*

ALIGN (*n*)

following the final location of the previous section with alignment to the next *n*-byte boundary

3. The initial address (LMA) of the section. This is expressed using one of the following forms:

LMA_EQ_VMA

the LMA equals the VMA (no relocation)

AT (*n*)

at the absolute address specified by the unsigned integer *n*

FOLLOWING (*.name*)

following the initial location of section *name*

In order to maintain compatibility with linker script fragments and header files exported by the eCos Configuration Tool, the use of other expressions within these files is not recommended.

Note that the names of the linker output sections will vary between target architectures. A description of these sections can be found in the specific *GNUPro Toolkit Reference manual* for your architecture.

15

Managing the Package Repository

A source distribution of eCos consists of a number of packages, such as the kernel, the C library, and the µITRON subsystems. These are individually versioned in the tree structure of the source code, to support distribution on a per-package basis and to support third party packages whose versioning systems might be different. The **eCos Package Administration Tool** is used to manage the installation and removal of packages from a variety of sources with potentially multiple versions.

The presence of the version information in the source tree structure might be a hindrance to the use of a separate source control system such as **CVS** or **SourceSafe**™. To work in this way, you can rename all the version components to some common name (such as "current") thus unifying the structure of source trees from distinct eCos releases.

The eCos build system will treat any such name as just another version of the package(s), and support building in exactly the same way. However, performing this rename invalidates any existing build trees that referred to the versioned source tree, so do the rename first, before any other work, and do a complete rebuild afterwards.

Package Installation

Package installation and removal is performed using the **eCos Package Adminstration Tool**. This tool is a Tcl script named `ecosadmin.tcl` which allows the user to add new eCos packages and new versions of existing packages to an eCos

repository. Such packages must be distributed as a single file in the eCos package distribution format. Unwanted packages may also be removed from the repository using this tool. A graphical version of the tool is provided as part of the **eCos Developer's Kit**.

Using the Administration Tool

The graphical version of the **eCos Package Administration Tool**, provided as part of the **eCos Developer's Kit**, provides functions equivalent to the command-line version for those who prefer a Windows-based interface.

It may be invoked in one of two ways:

- from the start menu (by default **Start->Programs->Red Hat eCos->Package Administration Tool**)
- from the eCos Configuration Tool via the **Tools->Administration** menu item
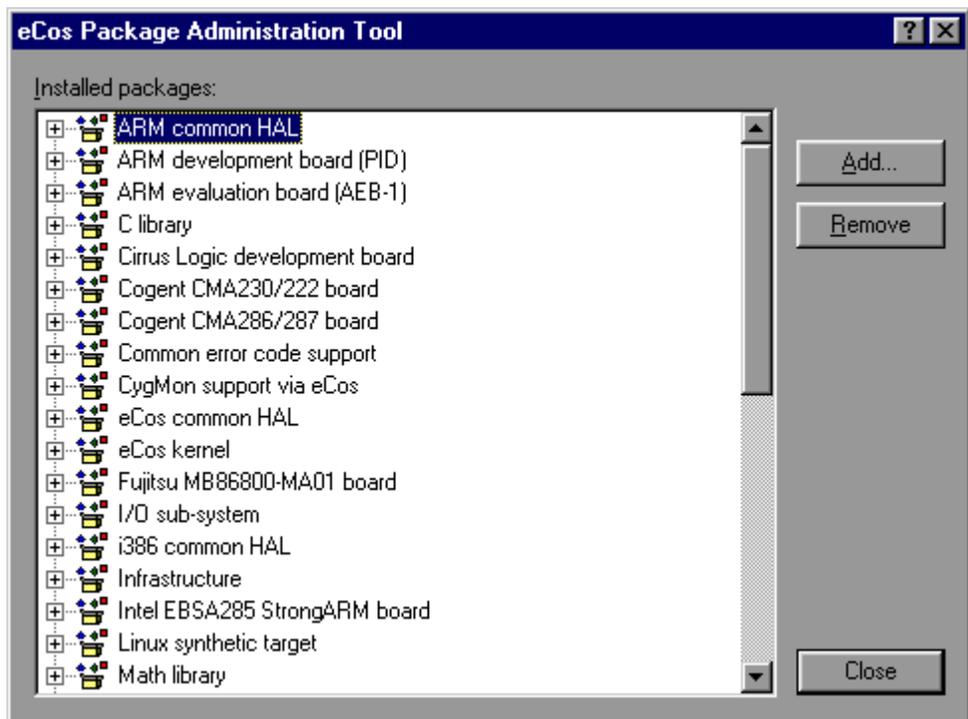
The main window of the tool displays the packages which are currently installed in the form of a tree. The installed versions of each package may be examined by expanding the tree.

Packages may be added to the eCos repository by clicking on the **Add** button. The eCos package distribution file to be added is then selected via a **File Open** dialog box.

Packages may be removed by selecting a package in the tree and then clicking on the **Remove** button. If a package node is selected, all versions of the selected package will be removed. If a package version node is selected, only the selected version of the package will be removed.

Using the command line

The `ecosadmin.tcl` script is located in the base of the eCos repository. Use a command of the following form under versions of UNIX:

```
$ tclsh ecosadmin.tcl <command>
```

Under Windows, a command of the following form may be used at the Cygwin command line prompt:

```
$ cygtclsh80 ecosadmin.tcl <command>
```

The following commands are available:

`add <file>`

Adds the packages contained with the specified package distribution file to the eCos repository and updates the package database accordingly. By convention, eCos package distribution files are given the `.epk` suffix.

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

Removes the specified package from the eCos repository and updates the package database accordingly. Where the optional version qualifier is used, only the specified version of the package is removed.

`list`

Produces a list of the packages which are currently installed and their versions. The available templates and hardware targets are also listed.

Note that is is possible to remove critical packages such as the common HAL package using this tool. Users should take care to avoid such errors since core eCos packages may only be re-installed in the context of a complete re-installation of eCos.

Package Structure

The files in an installed eCos source tree are organized in a natural tree structure, grouping together files which work together into *Packages*. For example, the kernel files are all together in:

```
BASE_DIR/kernel/v1_3_x/include/
BASE_DIR/kernel/v1_3_x/src/
BASE_DIR/kernel/v1_3_x/tests/
```

and μITRON compatibility layer files are in:

```
BASE_DIR/compat/uitron/v1_3_x/include/
BASE_DIR/compat/uitron/v1_3_x/src/
BASE_DIR/compat/uitron/v1_3_x/tests/
```

The feature of these names which is of interest here is the v1_3_x near the end. If you start using eCos after the Version 1.2 release, you might see a different name here, e.g. v1_4 for version 1.4; if you received pre-release Beta versions you might have seen v0_2 or v0_3 for versions 0.2 and 0.3.

It may seem odd to place a version number deep in the path, rather than having something like `BASE_DIR/v1_3_x/...everything...`

or leaving it up to you to choose a different install-place when a new release of the system arrives.

There is a rationale for this organization: as indicated, the kernel and the μITRON compatibility subsystem are examples of software packages. For the first few releases of eCos, all the Red Hat packages will move along in step, i.e. Release 1.3.x will feature Version 1.3.x of every package, and so forth. But in future, especially when third party packages become available, it is intended that the package be the unit of software distribution, so it will be possible to build a system from a selection of packages with different version numbers, and even differing versioning *schemes*. A Tcl script `ecosadmin.tcl` is provided in the eCos repository to manage the installation and removal of packages in this way.

Many users will have their own source code control system, version control system or equivalent, and will want to use it with eCos sources. In that case, since a new release of eCos comes with different pathnames for all the source files, a bit of work is necessary to import a new release into your source repository.

One way of handling the import is to rename all the version parts to some common name, for example "current", and continue to work. "current" is suggested because `ecosconfig` recognizes it and places it first in any list of versions. In the future, Red Hat may provide a tool to help with this, or an option in the install wizard. Alternatively, in a POSIX shell environment (Linux or Cygwin on Windows) use the following command:

```
find . -name v1_3_x -type d -printf 'mv %p %h/current\n' | sh
```

Having carried out such a renaming operation, your source tree will now look like this:

```
BASE_DIR/kernel/current/include/
BASE_DIR/kernel/current/src/
BASE_DIR/kernel/current/tests/
  ...
```

```
BASE_DIR/compat/uitron/current/include/
BASE_DIR/compat/uitron/current/src/
BASE_DIR/compat/uitron/current/tests/
```

which is a suitable format for import into your own source code control system. When you get a subsequent release of eCos, do the same thing and use your own source code control system to manage the new source base, by importing the new version from

```
NEW_BASE_DIR/kernel/current/include/
```

and so on.

The eCos build tool will now offer only the "current" version of each package; select this for the packages you wish to use.

Making such a change has implications for any build trees you already have in use. A configured build tree contains information about the selected packages and their selected versions. Changing the name of the "versioning" folder in the source tree invalidates this information, and in consequence it also invalidates any local configuration options you have set up in this build tree. So if you want to change the version information in the source tree, do it first, before investing any serious time in configuring and building your system. When you create a new build tree to deal with the new source layout, it will contain default settings for all the configuration options, just like the old build tree did before you configured it. You will need to redo that configuration work in the new tree.

Moving source code around also invalidates debugging information in any programs or libraries built from the old tree; these will need to be rebuilt.

Part IV: Special Topics

16

Real-time Characterization

When building a real-time system, care must be taken to ensure that the system will be able to perform properly within the constraints of that system. One of these constraints may be how fast certain operations can be performed. Another might be how deterministic the overall behavior of the system is. Lastly the memory footprint (size) and unit cost may be important.

One of the major problems encountered while evaluating a system will be how to compare it with possible alternatives. Most manufacturers of real-time systems publish performance numbers, ostensibly so that users can compare the different offerings. However, what these numbers mean and how they were gathered is often not clear. The values are typically measured on a particular piece of hardware, so in order to truly compare, one must obtain measurements for exactly the same set of hardware that were gathered in a similar fashion.

Two major items need to be present in any given set of measurements. First, the raw values for the various operations; these are typically quite easy to measure and will be available for most systems. Second, the determinacy of the numbers; in other words how much the value might change depending on other factors within the system. This value is affected by a number of factors: how long interrupts might be masked, whether or not the function can be interrupted, even very hardware-specific effects such as cache locality and pipeline usage. It is very difficult to measure the determinacy of any given operation, but that determinacy is fundamentally important to proper overall characterization of a system.

In the discussion and numbers that follow, three key measurements are provided. The first measurement is an estimate of the interrupt latency: this is the length of time from when a hardware interrupt occurs until its Interrupt Service Routine (ISR) is called.

The second measurement is an estimate of overall interrupt overhead: this is the length of time average interrupt processing takes, as measured by the real-time clock interrupt (other interrupt sources will certainly take a different amount of time, but this data cannot be easily gathered). The third measurement consists of the timings for the various kernel primitives.

Methodology

Key operations in the kernel were measured by using a simple test program which exercises the various kernel primitive operations. A hardware timer, normally the one used to drive the real-time clock, was used for these measurements. In most cases this timer can be read with quite high resolution, typically in the range of a few microseconds. For each measurement, the operation was repeated a number of times. Time stamps were obtained directly before and after the operation was performed. The data gathered for the entire set of operations was then analyzed, generating average (mean), maximum and minimum values. The sample variance (a measure of how close most samples are to the mean) was also calculated. The cost of obtaining the real-time clock timer values was also measured, and was subtracted from all other times.

Most kernel functions can be measured separately. In each case, a reasonable number of iterations are performed. Where the test case involves a kernel object, for example creating a task, each iteration is performed on a different object. There is also a set of tests which measures the interactions between multiple tasks and certain kernel primitives. Most functions are tested in such a way as to determine the variations introduced by varying numbers of objects in the system. For example, the mailbox tests measure the cost of a 'peek' operation when the mailbox is empty, has a single item, and has multiple items present. In this way, any effects of the state of the object or how many items it contains can be determined.

There are a few things to consider about these measurements. Firstly, they are quite micro in scale and only measure the operation in question. These measurements do not adequately describe how the timings would be perturbed in a real system with multiple interrupting sources. Secondly, the possible aberration incurred by the real-time clock (system heartbeat tick) is explicitly avoided. Virtually all kernel functions have been designed to be interruptible. Thus the times presented are typical, but best case, since any particular function may be interrupted by the clock tick processing. This number is explicitly calculated so that the value may be included in any deadline calculations required by the end user. Lastly, the reported measurements were obtained from a system built with all options at their default values. Kernel instrumentation and asserts are also disabled for these measurements. Any number of

configuration options can change the measured results, sometimes quite dramatically. For example, mutexes are using priority inheritance in these measurements. The numbers will change if the system is built with priority inheritance on mutex variables turned off.

The final value that is measured is an estimate of interrupt latency. This particular value is not explicitly calculated in the test program used, but rather by instrumenting the kernel itself. The raw number of timer ticks that elapse between the time the timer generates an interrupt and the start of the timer ISR is kept in the kernel. These values are printed by the test program after all other operations have been tested. Thus this should be a reasonable estimate of the interrupt latency over time.

Using these Measurements

These measurements can be used in a number of ways. The most typical use will be to compare different real-time kernel offerings on similar hardware, another will be to estimate the cost of implementing a task using eCos (applications can be examined to see what effect the kernel operations will have on the total execution time). Another use would be to observe how the tuning of the kernel affects overall operation.

Influences on Performance

A number of factors can affect real-time performance in a system. One of the most common factors, yet most difficult to characterize, is the effect of device drivers and interrupts on system timings. Different device drivers will have differing requirements as to how long interrupts are suppressed, for example. The eCos system has been designed with this in mind, by separating the management of interrupts (ISR handlers) and the processing required by the interrupt (DSR—Deferred Service Routine—handlers). However, since there is so much variability here, and indeed most device drivers will come from the end users themselves, these effects cannot be reliably measured. Attempts have been made to measure the overhead of the single interrupt that eCos relies on, the real-time clock timer. This should give you a reasonable idea of the cost of executing interrupt handling for devices.

Measured Items

This section describes the various tests and the numbers presented. All tests use the C kernel API (available by way of cyg/kernel/kapi.h). There is a single main thread in the system that performs the various tests. Additional threads may be created as part of the testing, but these are short lived and are destroyed between tests unless otherwise noted. The terminology "lower priority" means a priority that is less important, not necessarily lower in numerical value. A higher priority thread will run in preference to a lower priority thread even though the priority value of the higher priority thread may be numerically less than that of the lower priority thread.

Thread Primitives

Create thread

> This test measures the `cyg_thread_create()` call. Each call creates a totally new thread. The set of threads created by this test will be reused in the subsequent thread primitive tests.

Yield thread

> This test measures the `cyg_thread_yield()` call. For this test, there are no other runnable threads, thus the test should just measure the overhead of trying to give up the CPU.

Suspend [suspended] thread

> This test measures the `cyg_thread_suspend()` call. A thread may be suspended multiple times; each thread is already suspended from its initial creation, and is suspended again.

Resume thread

> This test measures the `cyg_thread_resume()` call. All of the threads have a suspend count of 2, thus this call does not make them runnable. This test just measures the overhead of resuming a thread.

Set priority

> This test measures the `cyg_thread_set_priority()` call. Each thread, currently suspended, has its priority set to a new value.

Get priority

> This test measures the `cyg_thread_get_priority()` call.

Kill [suspended] thread

> This test measures the `cyg_thread_kill()` call. Each thread in the set is killed. All threads are known to be suspended before being killed.

Yield [no other] thread

This test measures the `cyg_thread_yield()` call again. This is to demonstrate that the `cyg_thread_yield()` call has a fixed overhead, regardless of whether there are other threads in the system.

Resume [suspended low priority] thread

This test measures the `cyg_thread_resume()` call again. In this case, the thread being resumed is lower priority than the main thread, thus it will simply become ready to run but not be granted the CPU. This test measures the cost of making a thread ready to run.

Resume [runnable low priority] thread

This test measures the `cyg_thread_resume()` call again. In this case, the thread being resumed is lower priority than the main thread and has already been made runnable, so in fact the resume call has no effect.

Suspend [runnable] thread

This test measures the `cyg_thread_suspend()` call again. In this case, each thread has already been made runnable (by previous tests).

Yield [only low priority] thread

This test measures the `cyg_thread_yield()` call. In this case, there are many other runnable threads, but they are all lower priority than the main thread, thus no thread switches will take place.

Suspend [runnable->not runnable] thread

This test measures the `cyg_thread_suspend()` call again. The thread being suspended will become non-runnable by this action.

Kill [runnable] thread

This test measures the `cyg_thread_kill()` call again. In this case, the thread being killed is currently runnable, but lower priority than the main thread.

Resume [high priority] thread

This test measures the `cyg_thread_resume()` call. The thread being resumed is higher priority than the main thread, thus a thread switch will take place on each call. In fact there will be two thread switches; one to the new higher priority thread and a second back to the test thread. The test thread exits immediately.

Thread switch

This test attempts to measure the cost of switching from one thread to another. Two equal priority threads are started and they will each yield to the other for a number of iterations. A time stamp is gathered in one thread before the `cyg_thread_yield()` call and after the call in the other thread.

Scheduler Primitives

Scheduler lock

This test measures the **cyg_scheduler_lock()** call.

Scheduler unlock [0 threads]

This test measures the **cyg_scheduler_unlock()** call. There are no other threads in the system and the unlock happens immediately after a lock so there will be no pending DSR's to run.

Scheduler unlock [1 suspended thread]

This test measures the **cyg_scheduler_unlock()** call. There is one other thread in the system which is currently suspended.

Scheduler unlock [many suspended threads]

This test measures the **cyg_scheduler_unlock()** call. There are many other threads in the system which are currently suspended. The purpose of this test is to determine the cost of having additional threads in the system when the scheduler is activated by way of **cyg_scheduler_unlock()**.

Scheduler unlock [many low priority threads]

This test measures the **cyg_scheduler_unlock()** call. There are many other threads in the system which are runnable but are lower priority than the main thread. The purpose of this test is to determine the cost of having additional threads in the system when the scheduler is activated by way of **cyg_scheduler_unlock()**.

Mutex Primitives

Init mutex

This test measures the **cyg_mutex_init()** call. A number of separate mutex variables are created. The purpose of this test is to measure the cost of creating a new mutex and introducing it to the system.

Lock [unlocked] mutex

This test measures the **cyg_mutex_lock()** call. The purpose of this test is to measure the cost of locking a mutex which is currently unlocked. There are no other threads executing in the system while this test runs.

Unlock [locked] mutex

This test measures the **cyg_mutex_unlock()** call. The purpose of this test is to measure the cost of unlocking a mutex which is currently locked. There are no other threads executing in the system while this test runs.

Trylock [unlocked] mutex

This test measures the `cyg_mutex_trylock()` call. The purpose of this test is to measure the cost of locking a mutex which is currently unlocked. There are no other threads executing in the system while this test runs.

Trylock [locked] mutex

This test measures the `cyg_mutex_trylock()` call. The purpose of this test is to measure the cost of locking a mutex which is currently locked. There are no other threads executing in the system while this test runs.

Destroy mutex

This test measures the `cyg_mutex_destroy()` call. The purpose of this test is to measure the cost of deleting a mutex from the system. There are no other threads executing in the system while this test runs.

Unlock/Lock mutex

This test attempts to measure the cost of unlocking a mutex for which there is another higher priority thread waiting. When the mutex is unlocked, the higher priority waiting thread will immediately take the lock. The time from when the unlock is issued until after the lock succeeds in the second thread is measured, thus giving the round-trip or circuit time for this type of synchronizer.

Mailbox Primitives

Create mbox

This test measures the `cyg_mbox_create()` call. A number of separate mailboxes is created. The purpose of this test is to measure the cost of creating a new mailbox and introducing it to the system.

Peek [empty] mbox

This test measures the `cyg_mbox_peek()` call. An attempt is made to peek the value in each mailbox, which is currently empty. The purpose of this test is to measure the cost of checking a mailbox for a value without blocking.

Put [first] mbox

This test measures the `cyg_mbox_put()` call. One item is added to a currently empty mailbox. The purpose of this test is to measure the cost of adding an item to a mailbox. There are no other threads currently waiting for mailbox items to arrive.

Peek [1 msg] mbox

This test measures the `cyg_mbox_peek()` call. An attempt is made to peek the value in each mailbox, which contains a single item. The purpose of this test is to measure the cost of checking a mailbox which has data to deliver.

Put [second] mbox

This test measures the `cyg_mbox_put()` call. A second item is added to a mailbox. The purpose of this test is to measure the cost of adding an additional item to a mailbox. There are no other threads currently waiting for mailbox items to arrive.

Peek [2 msgs] mbox

This test measures the `cyg_mbox_peek()` call. An attempt is made to peek the value in each mailbox, which contains two items. The purpose of this test is to measure the cost of checking a mailbox which has data to deliver.

Get [first] mbox

This test measures the `cyg_mbox_get()` call. The first item is removed from a mailbox that currently contains two items. The purpose of this test is to measure the cost of obtaining an item from a mailbox without blocking.

Get [second] mbox

This test measures the `cyg_mbox_get()` call. The last item is removed from a mailbox that currently contains one item. The purpose of this test is to measure the cost of obtaining an item from a mailbox without blocking.

Tryput [first] mbox

This test measures the `cyg_mbox_tryput()` call. A single item is added to a currently empty mailbox. The purpose of this test is to measure the cost of adding an item to a mailbox.

Peek item [non-empty] mbox

This test measures the `cyg_mbox_peek_item()` call. A single item is fetched from a mailbox that contains a single item. The purpose of this test is to measure the cost of obtaining an item without disturbing the mailbox.

Tryget [non-empty] mbox

This test measures the `cyg_mbox_tryget()` call. A single item is removed from a mailbox that contains exactly one item. The purpose of this test is to measure the cost of obtaining one item from a non-empty mailbox.

Peek item [empty] mbox

This test measures the `cyg_mbox_peek_item()` call. An attempt is made to fetch an item from a mailbox that is empty. The purpose of this test is to measure the cost of trying to obtain an item when the mailbox is empty.

Tryget [empty] mbox

This test measures the `cyg_mbox_tryget()` call. An attempt is made to fetch an item from a mailbox that is empty. The purpose of this test is to measure the cost of trying to obtain an item when the mailbox is empty.

Waiting to get mbox

This test measures the `cyg_mbox_waiting_to_get()` call. The purpose of this test is to measure the cost of determining how many threads are waiting to obtain a message from this mailbox.

Waiting to put mbox

This test measures the `cyg_mbox_waiting_to_put()` call. The purpose of this test is to measure the cost of determining how many threads are waiting to put a message into this mailbox.

Delete mbox

This test measures the `cyg_mbox_delete()` call. The purpose of this test is to measure the cost of destroying a mailbox and removing it from the system.

Put/Get mbox

In this round-trip test, one thread is sending data to a mailbox that is being consumed by another thread. The time from when the data is put into the mailbox until it has been delivered to the waiting thread is measured. Note that this time will contain a thread switch.

Semaphore Primitives

Init semaphore

This test measures the `cyg_semaphore_init()` call. A number of separate semaphore objects are created and introduced to the system. The purpose of this test is to measure the cost of creating a new semaphore.

Post [0] semaphore

This test measures the `cyg_semaphore_post()` call. Each semaphore currently has a value of 0 and there are no other threads in the system. The purpose of this test is to measure the overhead cost of posting to a semaphore. This cost will differ if there is a thread waiting for the semaphore.

Wait [1] semaphore

This test measures the `cyg_semaphore_wait()` call. The semaphore has a current value of 1 so the call is non-blocking. The purpose of the test is to measure the overhead of "taking" a semaphore.

Trywait [0] semaphore

This test measures the `cyg_semaphore_trywait()` call. The semaphore has a value of 0 when the call is made. The purpose of this test is to measure the cost of seeing if a semaphore can be "taken" without blocking. In this case, the answer would be no.

Trywait [1] semaphore

This test measures the `cyg_semaphore_trywait()` call. The semaphore has a value of 1 when the call is made. The purpose of this test is to measure the cost of seeing if a semaphore can be "taken" without blocking. In this case, the answer would be yes.

Peek semaphore

This test measures the `cyg_semaphore_peek()` call. The purpose of this test is to measure the cost of obtaining the current semaphore count value.

Destroy semaphore

This test measures the `cyg_semaphore_destroy()` call. The purpose of this test is to measure the cost of deleting a semaphore from the system.

Post/Wait semaphore

In this round-trip test, two threads are passing control back and forth by using a semaphore. The time from when one thread calls `cyg_semaphore_post()` until the other thread completes its `cyg_semaphore_wait()` is measured. Note that each iteration of this test will involve a thread switch.

Counters

Create counter

This test measures the `cyg_counter_create()` call. A number of separate counters are created. The purpose of this test is to measure the cost of creating a new counter and introducing it to the system.

Get counter value

This test measures the `cyg_counter_current_value()` call. The current value of each counter is obtained.

Set counter value

This test measures the `cyg_counter_set_value()` call. Each counter is set to a new value.

Tick counter

This test measures the `cyg_counter_tick()` call. Each counter is "ticked" once.

Delete counter

This test measures the `cyg_counter_delete()` call. Each counter is deleted from the system. The purpose of this test is to measure the cost of deleting a counter object.

Alarms

Create alarm

This test measures the `cyg_alarm_create()` call. A number of separate alarms are created, all attached to the same counter object. The purpose of this test is to measure the cost of creating a new counter and introducing it to the system.

Initialize alarm

This test measures the `cyg_alarm_initialize()` call. Each alarm is initialized to a small value.

Disable alarm

This test measures the `cyg_alarm_disable()` call. Each alarm is explicitly disabled.

Enable alarm

This test measures the `cyg_alarm_enable()` call. Each alarm is explicitly enabled.

Delete alarm

This test measures the `cyg_alarm_delete()` call. Each alarm is destroyed. The purpose of this test is to measure the cost of deleting an alarm and removing it from the system.

Tick counter [1 alarm]

This test measures the `cyg_counter_tick()` call. A counter is created that has a single alarm attached to it. The purpose of this test is to measure the cost of "ticking" a counter when it has a single attached alarm. In this test, the alarm is not activated (fired).

Tick counter [many alarms]

This test measures the `cyg_counter_tick()` call. A counter is created that has multiple alarms attached to it. The purpose of this test is to measure the cost of "ticking" a counter when it has many attached alarms. In this test, the alarms are not activated (fired).

Tick & fire counter [1 alarm]

This test measures the `cyg_counter_tick()` call. A counter is created that has a single alarm attached to it. The purpose of this test is to measure the cost of "ticking" a counter when it has a single attached alarm. In this test, the alarm is activated (fired). Thus the measured time will include the overhead of calling the alarm callback function.

Tick & fire counter [many alarms]

This test measures the `cyg_counter_tick()` call. A counter is created that has

multiple alarms attached to it. The purpose of this test is to measure the cost of "ticking" a counter when it has many attached alarms. In this test, the alarms are activated (fired). Thus the measured time will include the overhead of calling the alarm callback function.

Alarm latency [0 threads]

This test attempts to measure the latency in calling an alarm callback function. The time from the clock interrupt until the alarm function is called is measured. In this test, there are no threads that can be run, other than the system idle thread, when the clock interrupt occurs (all threads are suspended).

Alarm latency [2 threads]

This test attempts to measure the latency in calling an alarm callback function. The time from the clock interrupt until the alarm function is called is measured. In this test, there are exactly two threads which are running when the clock interrupt occurs. They are simply passing back and forth by way of the `cyg_thread_yield()` call. The purpose of this test is to measure the variations in the latency when there are executing threads.

Alarm latency [many threads]

This test attempts to measure the latency in calling an alarm callback function. The time from the clock interrupt until the alarm function is called is measured. In this test, there are a number of threads which are running when the clock interrupt occurs. They are simply passing back and forth by way of the `cyg_thread_yield()` call. The purpose of this test is to measure the variations in the latency when there are many executing threads.

Sample Numbers

For sample results, see Appendix 1 of *Getting Started with eCos*

Index

g++   56
gcc   56
shell
  creating   41
short description window   18
summary tab   39
system performance
  influences on   103
  measuring   101
  sample numbers   112
  tests performed   104

T

templates   27
test execution   35
tests
  alarms   111

counters   110
mailbox primitives   107
mutex primitives   106
scheduler primitives   106
semaphore primitives   109
thread primitives   104
Toolbars   11
tracing   57

U

updating configuration   23
user tools   34

W

window placement   11