An Introduction to eCos, the Embedded Configurable Operating System

by

Dr Nick Garnett

nickg@eCosCentric.com

#404 at the Embedded Systems Conference – San Francisco 2004

Introduction

eCos is an open source, royalty-free, real-time operating system intended for embedded applications. The highly configurable nature of eCos allows the operating system to be customized to precise application requirements, delivering the best possible run-time performance and an optimized hardware resource footprint. A thriving net community has grown up around the operating system ensuring ongoing technical innovation and wide platform support.

This tutorial gives a broad overview of the things that an engineer will need to know and do to evaluate eCos, develop applications, and port eCos to a new hardware platform. We will cover the following areas:

An overview of the functionality of eCos

How to configure eCos

How to build applications

How to port eCos to a new board

Overview

eCos is provided as an open source runtime system supported by the GNU open source development tools. Developers have full and unfettered access to all aspects of the runtime system. No parts of it are proprietary or hidden, and you are at liberty to examine, add to, and modify the code as you deem necessary. These rights are granted to you and protected by the eCos license. It also grants you the right to freely develop and distribute applications based on eCos. We welcome all contributions back to eCos such as board ports, device drivers and other components, as this helps the growth and development of eCos, and is of benefit to the entire eCos community.

One of the key technological innovations in eCos is the configuration system. The configuration system allows the application writer to impose their requirements on the run-time components, both in terms of their functionality and implementation, whereas traditionally the operating system has constrained the application's own implementation. Essentially, this enables eCos developers to create their own application-specific operating system and makes eCos suitable for a wide range of embedded uses. Configuration also ensures that the resource footprint of eCos is minimized as all unnecessary functionality and features are removed. The configuration system also presents eCos as a component architecture. This provides a standardized mechanism for component suppliers to extend the functionality of eCos and allows applications to be built from a wide set of optional configurable run-time components. Components can be provided from a variety of sources including the standard eCos release, commercial third party developers and open source contributors.

The royalty-free nature of eCos means that you can develop and deploy your application using the standard eCos release without incurring any royalty charges. In addition, there are no up-front license charges for the eCos runtime source code and associated tools. eCos delivers, without charge, everything necessary for basic embedded applications development.

eCos is designed to be portable to a wide range of target architectures and target platforms including 16, 32, and 64 bit architectures, MPUs, MCUs and DSPs. The eCos kernel, libraries and runtime components are layered on the Hardware Abstraction Layer (HAL), and thus will run on any target once the HAL and relevant device drivers have been ported to the target's processor architecture and board. Currently eCos supports thirteen different target architectures (ARM, Hitachi H8300, Intel x86, MIPS, Matsushita AM3x, Motorola 68k/Coldfire, PowerPC, SuperH, Fujitsu FR-V, Calmrisc, OpenRisc, SPARC and NEC V8xx) including many of the popular variants of these architectures and evaluation boards. Many new ports are in development and will be released as they become available.

eCos has been designed to support applications with real-time requirements, providing features such as full preemptability, minimal interrupt latencies, and all the necessary synchronization primitives, scheduling policies, and interrupt handling mechanisms needed for these type of applications. eCos also provides all the functionality required for general embedded application support including device drivers, memory management, exception handling, C, math libraries, etc. In addition to runtime support, the eCos system includes all the tools necessary to develop embedded applications, including eCos software configuration and build tools, and GNU based compilers, assemblers, linkers, debuggers, and simulators.

The following core functionality is provided:

- Hardware Abstraction Layer (HAL)
- Real-time kernel
    - Interrupt handling
    - Exception handling
    - Choice of schedulers
    - Thread support
    - Rich set of synchronization primitives
    - Timers, counters and alarms
    - Choice of memory allocators
    - Debug and instrumentation support
- µITRON 3.0 compatible API
- POSIX compatible API
- ISO C and math libraries
- Serial, Ethernet, LCD, PCMCIA, wallclock and watchdog device drivers
- USB slave support
- ROM, RAM, FAT16 and JFFS2 filesystems
- TCP/IP networking stacks
    - Bootp/DHCP
    - DNS
    - TFTP/FTP
    - SNMP
    - IPv6
    - HTTPD
    - PPP
    - VNC
- GDB debug support
- *RedBoot* ROM Monitor
    - Flash management
    - Serial and Ethernet download
    - Serial and Ethernet debugging

Configuring eCos

The configuration system is a major feature of eCos. In the large scale this is enabled by dividing eCos into a number of packages. There is a separate package for each subsystem, for example the kernel, HALs, network stack and each device driver are all separate packages. Packages can also be versioned, and a number of different versions of the same package may be present in the source repository simultaneously.

Each package contains a number of directories for the source files, include files, CDL scripts and other files that may be part of it. The CDL (Component Description Language) script describes the package to the configuration system. Information in the CDL script includes the set of source files to be compiled and any internal configuration options that may be needed.

The CDL file also contains dependency constraints on other packages and their option values. In addition to dependencies on the presence or absence of other packages, the CDL script can include requirements on the values, or ranges, of configuration options exported by other packages. For example, the POSIX package not only requires that the kernel package be present, but also requires it to implement prioritized threads, timeslicing, and priority inheritance.

Since the requirements expressed in the CDL files of different packages may conflict, and the user is also able to make his/her own changes to the configuration, it is possible for an invalid configuration to be generated. For this reason the configuration tools contain an inference engine which attempts to resolve any conflicts automatically.

eCos contains two versions of the configuration tool, a GUI tool, *configtool*, and a command line version, *ecosconfig*. Both tools contain the same CDL interpreter and inference engine, they just present a different user interface. They also both work on the same save files, so the tools may be used interchangeably. The GUI configuration tool is built using *wxWindows*, a portable windowing toolkit, and runs on both Windows and Linux hosts. It presents a tree view of the configuration space, see Figure 1 for an example. Large scale configuration changes such as adding or removing whole packages are supported by dialogs. Smaller scale changes, such as altering the values of specific options, can be done directly in the tree view. This tool also includes a subsystem to build the configured eCos library and to build and run the test programs that are part of the distribution.

The command line tool, *ecosconfig*, runs in both Linux and Windows under Cygwin (a free UNIX compatible porting layer for Windows). It is useful for automated build and testing systems, or for users who prefer this mode of working. The tool itself only directly supports large scale configuration actions such as applying a template or adding and removing packages. Small scale configuration of individual option values is handled by editing the human-readable save file and then re-executing *ecosconfig* to run the inference engine and update any dependencies.

The end result of running either configuration tool is a build directory tree containing a set of synthesized makefiles. These makefiles can be run from *configtool* directly, or, for command-line users, by executing *make* in the top-level of the build directory. The output of the build process is an install directory containing a tree of include files plus a lib directory containing libtarget.a – the compiled eCos system – some object files and a linker script. These are used by the application to generate a complete executable image which can then be downloaded to the target.

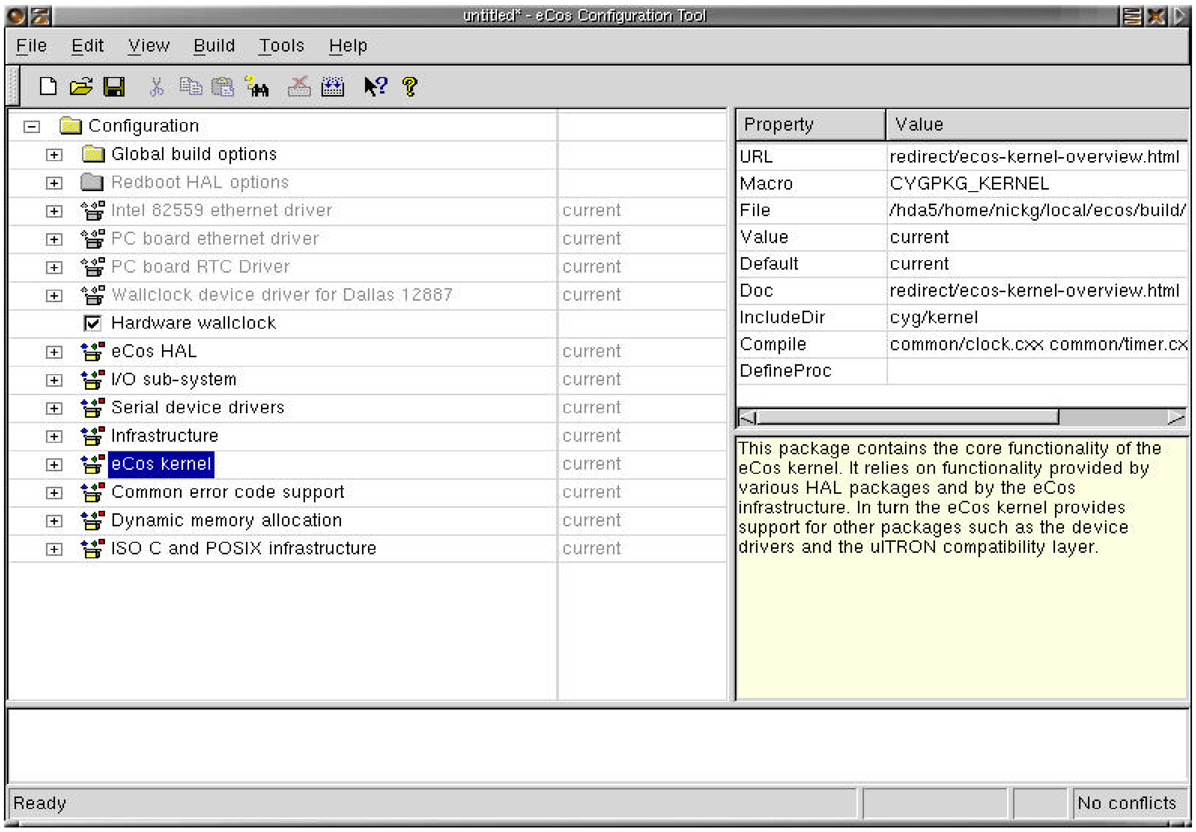
Example Configuration Session

This section runs through an example configuration session that shows how eCos is configured and built.

Figure 1 shows the startup screen of the *configtool*. On the left is a tree view of the configuration space with each major package listed. To the right of each entry in the tree view is its current value, in this case it is the version number of the entries that are packages.

The panel to the top right shows the properties of the item currently selected in the tree view. Here it shows the properties of the kernel package. The most important part here is the *Macro* property which will be defined as a C preprocessor macro in a header (.h) file when this package is present. The panel immediately below it shows a description of the package derived from the package's CDL file.

The panel at the bottom of the screen is for command output when a build is being done.



Configuration	
Global build options	
Redboot HAL options	
Intel 82559 ethernet driver	current
PC board ethernet driver	current
PC board RTC Driver	current
Wallclock device driver for Dallas 12887	current
Hardware wallclock	
eCos HAL	current
I/O sub-system	current
Serial device drivers	current
Infrastructure	current
eCos kernel	current
Common error code support	current
Dynamic memory allocation	current
ISO C and POSIX infrastructure	current

Property	Value
URL	redirect/ecos-kernel-overview.html
Macro	CYGPKG_KERNEL
File	/hda5/home/nickg/local/ecos/build/
Value	current
Default	current
Doc	redirect/ecos-kernel-overview.html
IncludeDir	cyg/kernel
Compile	common/clock.cxx common/timer.cx
DefineProc	

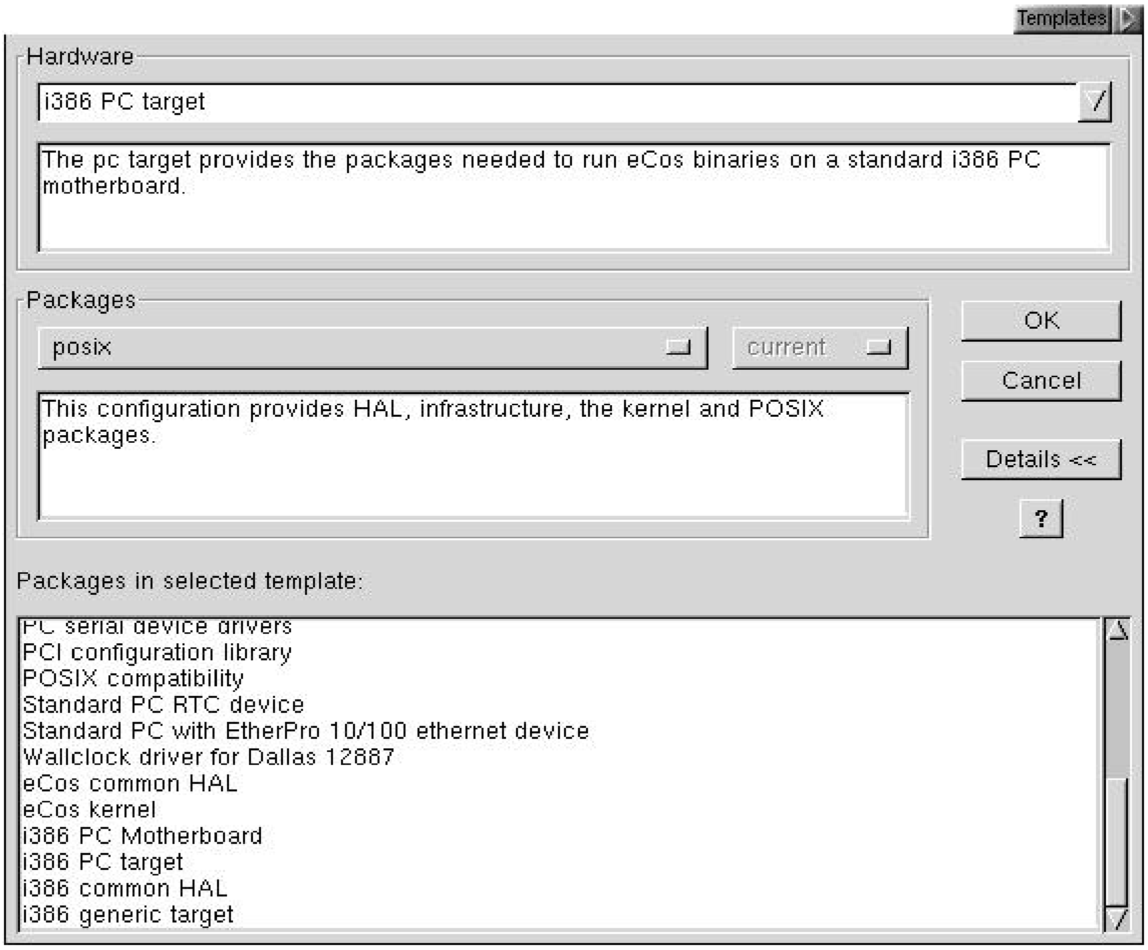
This package contains the core functionality of the eCos kernel. It relies on functionality provided by various HAL packages and by the eCos infrastructure. In turn the eCos kernel provides support for other packages such as the device drivers and the uITRON compatibility layer.

Ready

No conflicts

Figure 1 -- Configtool Initial Screen

The first thing we need to do is select the target hardware and a template to base our configuration on. In the *Build* menu we can select the *Templates* entry and we get the dialog shown in Figure 2. The top *Hardware* section allows the user to select a specific target board. In this case we are selecting the PC target. The *Packages* section allows us to select a specific template to base our configuration on. Templates are predefined sets of packages with any conflicts between them already resolved. Templates exist to set up the configuration for particular purposes, such as running POSIX or ìITRON applications, or with particular functionality such as networking or building RedBoot. In this example we are going to select the POSIX template. The packages that will be included in the configuration are listed at the bottom of the dialog.



i386 PC target

The pc target provides the packages needed to run eCos binaries on a standard i386 PC motherboard.

posix

current

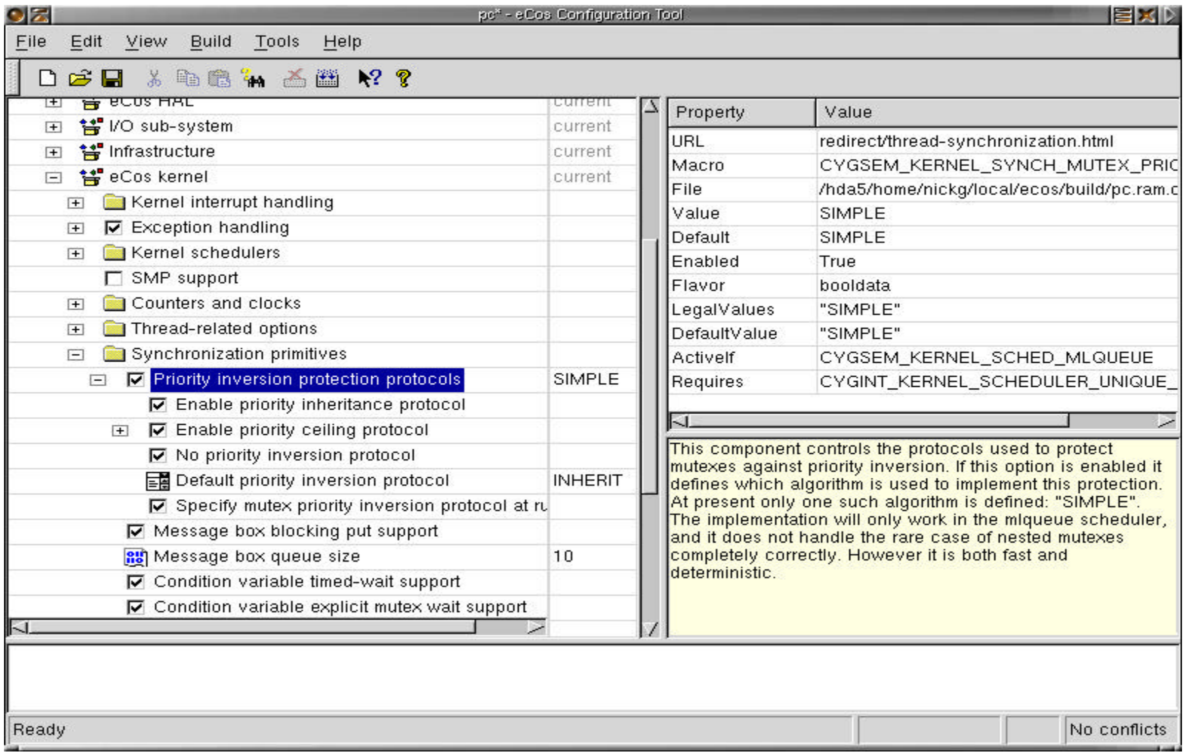
This configuration provides HAL, infrastructure, the kernel and POSIX packages.

PC serial device drivers
PCI configuration library
POSIX compatibility
Standard PC RTC device
Standard PC with EtherPro 10/100 ethernet device
Wallclock driver for Dallas 12887
eCos common HAL
eCos kernel
i386 PC Motherboard
i386 PC target
i386 common HAL
i386 generic target

Figure 2 -- Templates Dialog

Clicking the *OK* button takes us back to the main screen. Here we can inspect and maybe change the configuration by opening up the tree view. Figure 3 shows the view opened up to show the configuration options that control the kernel support for priority inversion protection. The POSIX package requires that this option be enabled, so as an experiment let's disable this option in the kernel. Figure 4 shows what happens when we do that, we get a dialog from the inference engine showing us a conflict. The POSIX configuration options _POSIX_THREAD_PRIO_INHERIT and _POSIX_THREAD_PRIO_PROTECT depend on this option being defined. The inference engine is proposing a solution, which is to disable the options in the POSIX package. If we accept that proposed solution by clicking the *Continue* button we will return to the main screen with the kernel support now disabled, as shown in Figure 5.

Having configured eCos we are now ready to build the eCos library and tests. If we select the *Build->Tests* menu item, *configtool* will generate the build and install directories, synthesize the makefiles and invoke *make*. Output from the build process will appear in the output panel at the bottom of the main *configtool* window. Once the build completes successfully, we can run some tests.


pc* - eCos Configuration Tool
File Edit View Build Tools Help

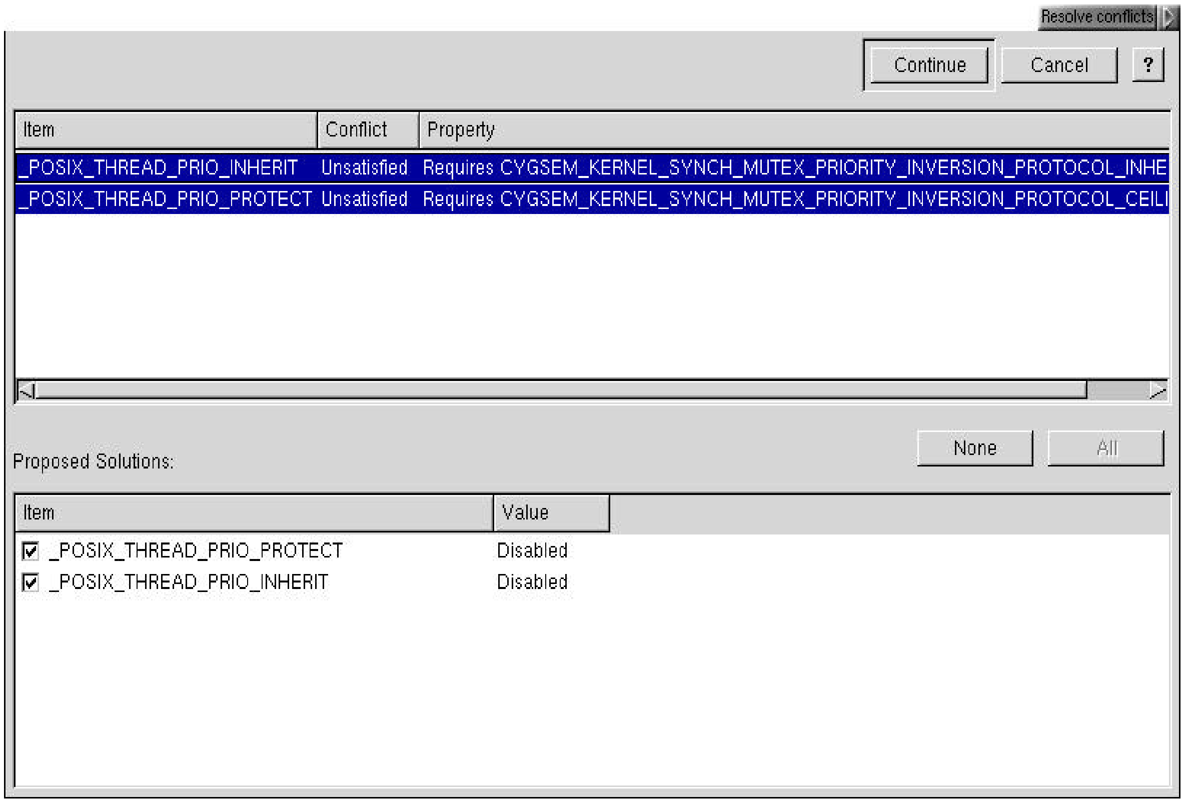
eCos HAL	current
I/O sub-system	current
Infrastructure	current
eCos kernel	current
Kernel interrupt handling	
☑ Exception handling	
Kernel schedulers	
☐ SMP support	
Counters and clocks	
Thread-related options	
Synchronization primitives	
☑ Priority inversion protection protocols	SIMPLE
☑ Enable priority inheritance protocol	
☑ Enable priority ceiling protocol	
☑ No priority inversion protocol	
Default priority inversion protocol	INHERIT
☑ Specify mutex priority inversion protocol at ru	
☑ Message box blocking put support	
Message box queue size	10
☑ Condition variable timed-wait support	
☑ Condition variable explicit mutex wait support	

Property	Value
URL	redirect/thread-synchronization.html
Macro	CYGSEM_KERNEL_SYNCH_MUTEX_PRIC
File	/hda5/home/nickg/local/ecos/build/pc.ram.c
Value	SIMPLE
Default	SIMPLE
Enabled	True
Flavor	booldata
LegalValues	"SIMPLE"
DefaultValue	"SIMPLE"
ActiveIf	CYGSEM_KERNEL_SCHED_MLQUEUE
Requires	CYGINT_KERNEL_SCHEDULER_UNIQUE_

This component controls the protocols used to protect mutexes against priority inversion. If this option is enabled it defines which algorithm is used to implement this protection. At present only one such algorithm is defined: "SIMPLE". The implementation will only work in the mlqueue scheduler, and it does not handle the rare case of nested mutexes completely correctly. However it is both fast and deterministic.

Ready | No conflicts

Figure 3 -- Configuration Options

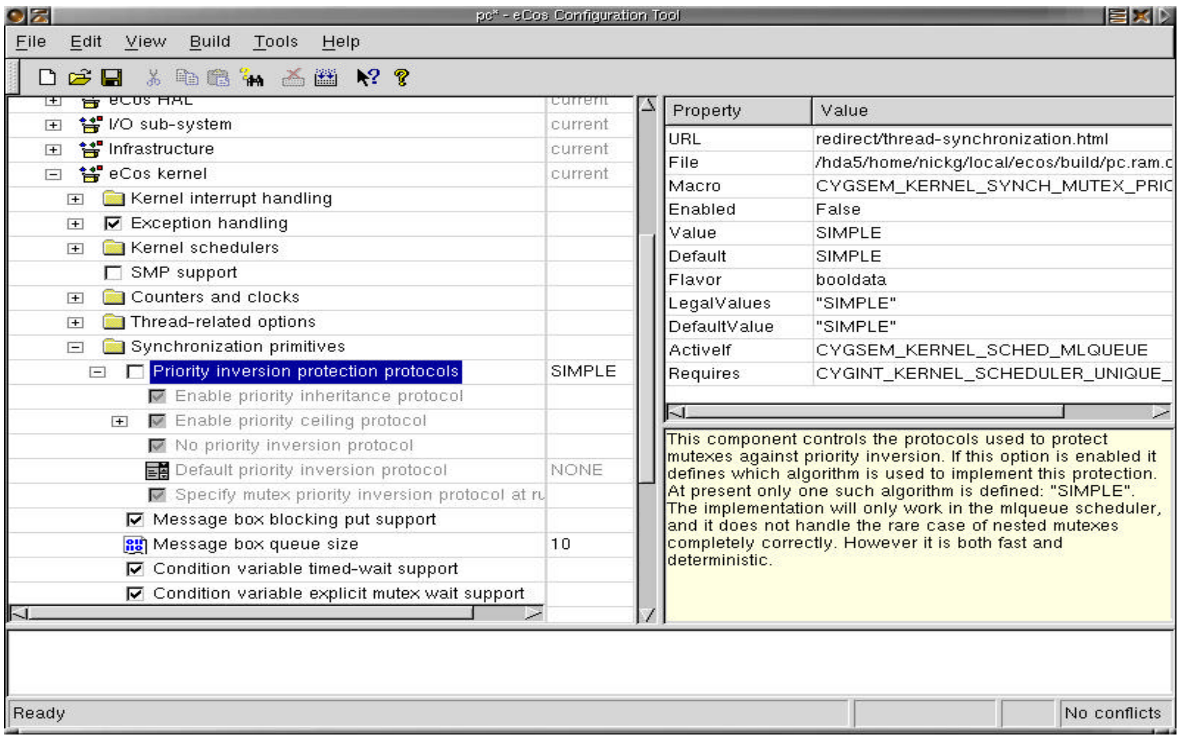


Item	Conflict	Property
_POSIX_THREAD_PRIO_INHERIT	Unsatisfied	Requires CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_INHE
_POSIX_THREAD_PRIO_PROTECT	Unsatisfied	Requires CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILI

Proposed Solutions:

Item	Value
☑ _POSIX_THREAD_PRIO_PROTECT	Disabled
☑ _POSIX_THREAD_PRIO_INHERIT	Disabled

Figure 4 -- Conflict Dialog

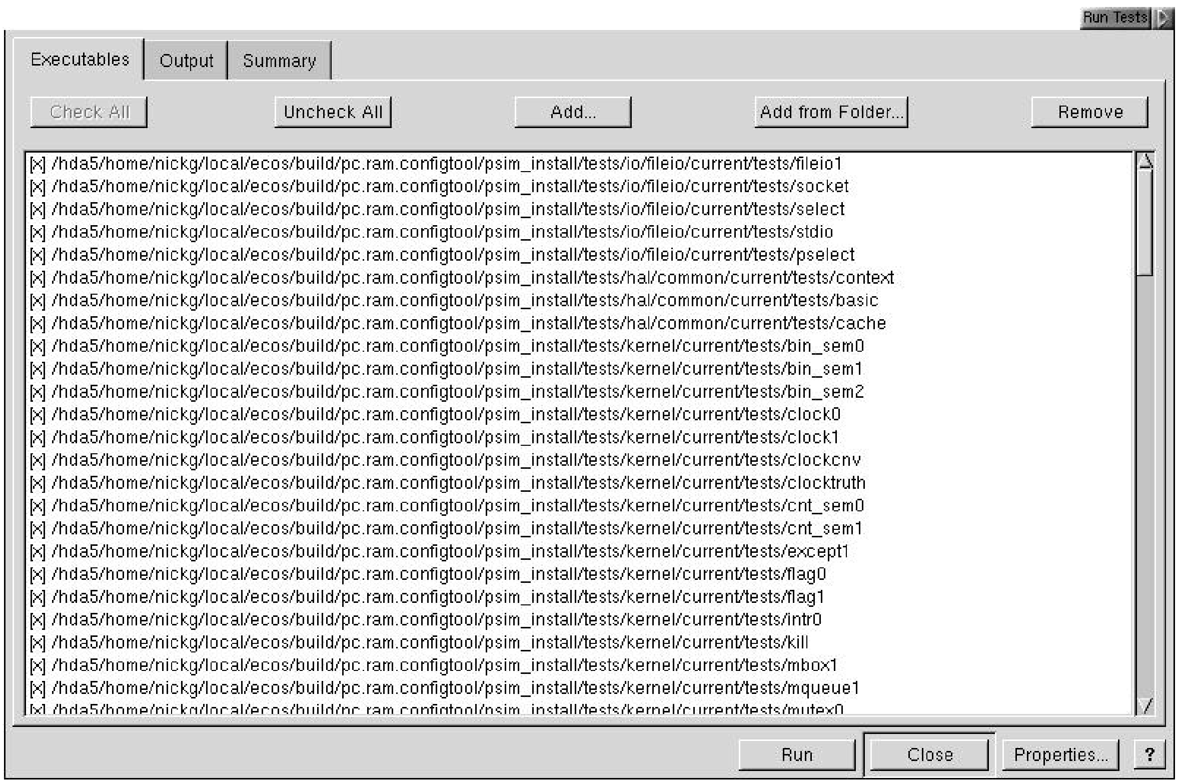


```
[+] eCos HAL                                            current
[+] I/O sub-system                                      current
[+] Infrastructure                                      current
[-] eCos kernel                                         current
    [+] Kernel interrupt handling
    [+] [✓] Exception handling
    [+] Kernel schedulers
        [ ] SMP support
    [+] Counters and clocks
    [+] Thread-related options
    [-] Synchronization primitives
        [-] [ ] Priority inversion protection protocols SIMPLE
                [✓] Enable priority inheritance protocol
            [+] [✓] Enable priority ceiling protocol
                [✓] No priority inversion protocol
                    Default priority inversion protocol NONE
                [✓] Specify mutex priority inversion protocol at ru
            [✓] Message box blocking put support
                Message box queue size                  10
            [✓] Condition variable timed-wait support
            [✓] Condition variable explicit mutex wait support
```

Property	Value
URL	redirect/thread-synchronization.html
File	/hda5/home/nickg/local/ecos/build/pc.ram.c
Macro	CYGSEM_KERNEL_SYNCH_MUTEX_PRIC
Enabled	False
Value	SIMPLE
Default	SIMPLE
Flavor	booldata
LegalValues	"SIMPLE"
DefaultValue	"SIMPLE"
ActiveIf	CYGSEM_KERNEL_SCHED_MLQUEUE
Requires	CYGINT_KERNEL_SCHEDULER_UNIQUE_

This component controls the protocols used to protect mutexes against priority inversion. If this option is enabled it defines which algorithm is used to implement this protection. At present only one such algorithm is defined: "SIMPLE". The implementation will only work in the mlqueue scheduler, and it does not handle the rare case of nested mutexes completely correctly. However it is both fast and deterministic.

Figure 5 -- Resolved Conflict

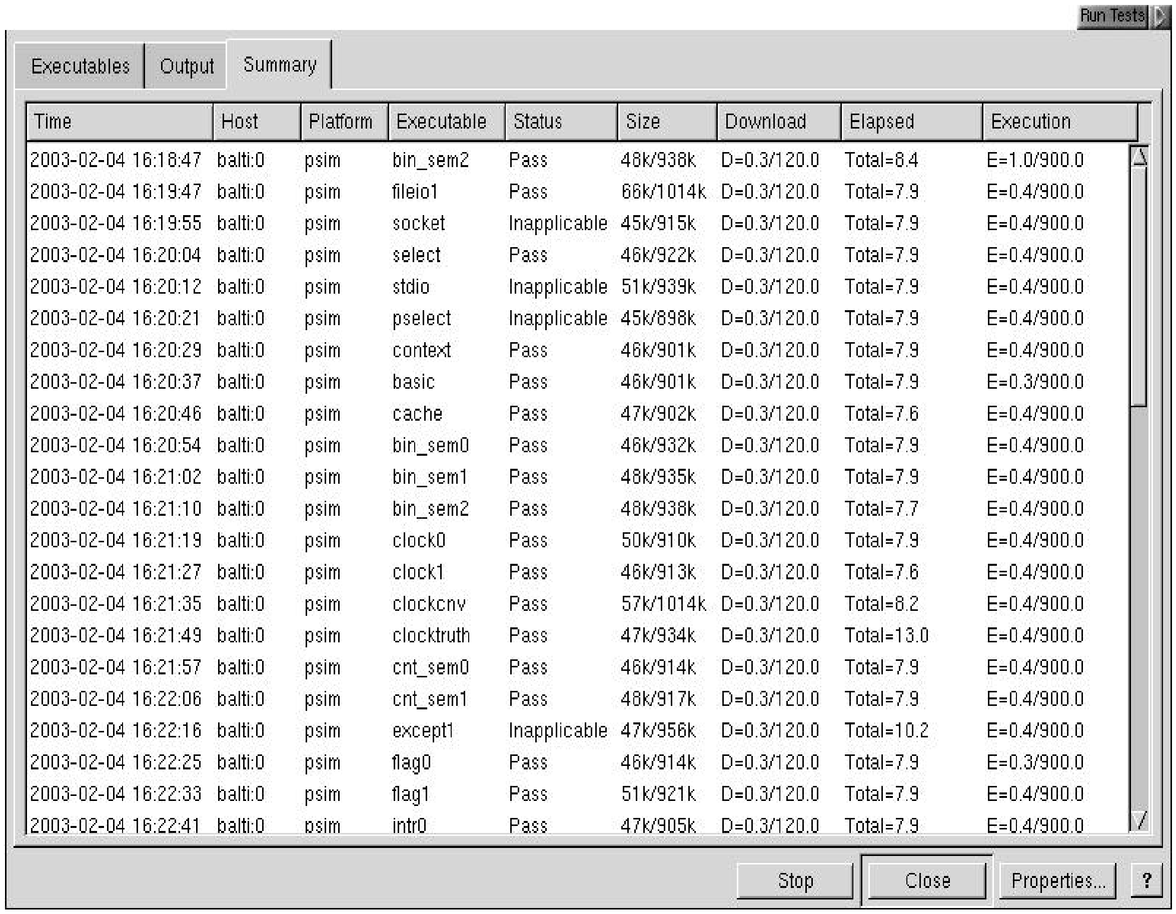


```
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/io/fileio/current/tests/fileio1
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/io/fileio/current/tests/socket
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/io/fileio/current/tests/select
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/io/fileio/current/tests/stdio
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/io/fileio/current/tests/pselect
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/hal/common/current/tests/context
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/hal/common/current/tests/basic
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/hal/common/current/tests/cache
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/bin_sem0
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/bin_sem1
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/bin_sem2
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/clock0
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/clock1
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/clockcnv
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/clocktruth
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/cnt_sem0
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/cnt_sem1
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/except1
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/flag0
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/flag1
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/intr0
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/kill
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/mbox1
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/mqueue1
[x] /hda5/home/nickg/local/ecos/build/pc.ram.configtool/psim_install/tests/kernel/current/tests/mutex0
```

Figure 6 -- Executables Dialog

Clicking the *Tools->Run Tests* menu item brings up the dialog shown in Figure 6. This shows all the tests that have been built. If we click the *Check All* button and then hit the *Run* button we will start executing the tests.

In this example we have chosen the PowerPC simulator target, which does not need real hardware to execute, so the programs start running immediately. The *Output* tab shows the full output of the test execution, but the configuration tool is able to parse the output and generate a summary, which is shown in the *Summary* tab, Figure 7. Here we can see for each test whether it passed or failed. Inapplicable tests are ones that rely on packages that have not been included in the configuration or which require specific values of certain options that are not true for this configuration.



Time	Host	Platform	Executable	Status	Size	Download	Elapsed	Execution
2003-02-04 16:18:47	balti:0	psim	bin_sem2	Pass	48k/938k	D=0.3/120.0	Total=8.4	E=1.0/900.0
2003-02-04 16:19:47	balti:0	psim	fileio1	Pass	66k/1014k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:19:55	balti:0	psim	socket	Inapplicable	45k/915k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:20:04	balti:0	psim	select	Pass	46k/922k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:20:12	balti:0	psim	stdio	Inapplicable	51k/939k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:20:21	balti:0	psim	pselect	Inapplicable	45k/898k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:20:29	balti:0	psim	context	Pass	46k/901k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:20:37	balti:0	psim	basic	Pass	46k/901k	D=0.3/120.0	Total=7.9	E=0.3/900.0
2003-02-04 16:20:46	balti:0	psim	cache	Pass	47k/902k	D=0.3/120.0	Total=7.6	E=0.4/900.0
2003-02-04 16:20:54	balti:0	psim	bin_sem0	Pass	46k/932k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:21:02	balti:0	psim	bin_sem1	Pass	48k/935k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:21:10	balti:0	psim	bin_sem2	Pass	48k/938k	D=0.3/120.0	Total=7.7	E=0.4/900.0
2003-02-04 16:21:19	balti:0	psim	clock0	Pass	50k/910k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:21:27	balti:0	psim	clock1	Pass	46k/913k	D=0.3/120.0	Total=7.6	E=0.4/900.0
2003-02-04 16:21:35	balti:0	psim	clockcnv	Pass	57k/1014k	D=0.3/120.0	Total=8.2	E=0.4/900.0
2003-02-04 16:21:49	balti:0	psim	clocktruth	Pass	47k/934k	D=0.3/120.0	Total=13.0	E=0.4/900.0
2003-02-04 16:21:57	balti:0	psim	cnt_sem0	Pass	46k/914k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:22:06	balti:0	psim	cnt_sem1	Pass	48k/917k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:22:16	balti:0	psim	except1	Inapplicable	47k/956k	D=0.3/120.0	Total=10.2	E=0.4/900.0
2003-02-04 16:22:25	balti:0	psim	flag0	Pass	46k/914k	D=0.3/120.0	Total=7.9	E=0.3/900.0
2003-02-04 16:22:33	balti:0	psim	flag1	Pass	51k/921k	D=0.3/120.0	Total=7.9	E=0.4/900.0
2003-02-04 16:22:41	balti:0	psim	intr0	Pass	47k/905k	D=0.3/120.0	Total=7.9	E=0.4/900.0

Figure 7 -- Summary Dialog

Program Development

The eCos tools merely generate a set of include files, a library, some object files and a linker script for combining with an application to make an executable. How the application is developed is up to the user, eCos imposes no methods of its own.

The simplest approach is to use the command line tools that form part of the GCC toolchain together with GNU make. An example makefile is included as part of the eCos system so this is very easy to do. The user is also able to use his/her own favorite editor for program development. For debugging the command line version of GDB can be used, or its GUI mode, Insight. Most of the eCos developers use *emacs* which provides support for invoking make and running GDB from within it, to provide many of the useful features of an IDE (Integrated Development Environment).

For those users who want to use a full IDE, eCos is well supported by the Eclipse/CDT IDE. Eclipse is an open source IDE framework developed by a consortium that includes IBM, Borland. Red Hat, SuSE and many others. CDT is a set of C/C++ development plugins for Eclipse that have been further modified by eCosCentric to support embedded eCos development. Eclipse includes an integrated editor and source browser and an integrated debugger. CDT interfaces to the GCC toolchain and uses GNU make to execute builds and interfaces to GDB to debug programs.

Application development is not confined to these approaches, any development environment may be used. The only restriction is that the GCC toolchain should be used to compile and link applications. Debugging with GDB is recommended since other debuggers may not be able to interpret the debug information included in the object file by the GCC tools.

eCos has a number of facilities for debugging applications. RedBoot supports the GDB remote debug protocol which will allow programs to be downloaded and debugged over either serial or, where available, Ethernet connections. The protocol stub may also be included into any executable so that it may be debugged without RedBoot present – although this supports serial communications only. GDB can also support hardware debug mechanisms such as JTAG, BDM and some ICEs. Within eCos itself there are debug mechanisms such as an execution trace buffer and kernel instrumentation. If none of these debug mechanisms are available, there is always the fallback of printing out messages. The `diag_printf()` function uses very low level HAL facilities to emit messages through a diagnostic serial channel.

Porting eCos

This section covers the structure of the eCos Hardware Abstraction Layer (HAL) and describes the general steps needed to port eCos to a new platform.

HAL Structure

eCos is designed to be highly portable. Most of eCos is target independent and is layered on top of a Hardware Abstraction Layer (HAL). In addition to isolating hardware specific code from the rest of the system, the HAL is itself structured to simplify the porting process and to maximize the amount of code that can be reused between similar targets.

The HAL for a particular target is divided into a number of packages. The most generic package is the Architecture HAL, which encapsulates the features of a particular CPU architecture. This includes starting up the system from the reset vector as well as handling exceptions and interrupts. Other pieces of hardware that may be handled at this level include the Memory Management Unit (MMU), the Floating Point Unit (FPU) and the CPU caches. The Architecture HAL also provides code to support thread context switching and architecture specific debugging operations such as planting breakpoints or single stepping.

The Variant HAL provides support for a particular variant of an architecture. These variations range from instruction set differences to the collection of on-chip peripherals available in different packages. Each variant usually corresponds to a particular physical device – a single processor chip or microcontroller for example. Depending on where in the architecture they are defined, some or all of the support for the MMU, FPU and caches may be located here rather than in the architecture HAL. Other things that are often handled at this level are memory and interrupt controllers, UARTs and PCI bus interfaces.

The Platform HAL covers the features specific to a particular board. Its main job is to initialize the board and define the position and size of memory and peripherals that are not already handled by the variant HAL. If the variant does not represent a highly integrated device then the memory and interrupt controllers may need to be defined here, together with UARTs, bus interfaces and any other devices such as LED and LCD displays.

In addition to these three main HAL levels there may be others. The Common HAL provides code that needs to live at HAL level, but is portable to all targets. This includes the GDB remote protocol stub and the rest of the target-end code for debugging. It also implements the virtual vector service interface that is used for communication between RedBoot and eCos applications. There are also some auxiliary HALs which provide support for functionality that is common to many different targets. These HALs may operate at either the variant or platform level. Examples are the XScale common core HAL that provides common XScale functionality to all XScale variants and the QUICC device support HAL that provides QUICC device functionality to various PowerPC platforms.

All HALs export a set of well defined interfaces to the rest of eCos. Internally, the interfaces between the packages that make up the HAL are more fluid and are under the control of the HAL designer; however most HALs follow a standard pattern.

The public interfaces are all exported from the Architecture HAL through a set of standard header files. These headers in turn include standard headers from the variant and platform HALs. All the public interfaces are defined as C preprocessor macros. This approach allows for a variety of different implementations for the HAL interfaces such as code fragments, function calls or inline assembler. It also allows the variant and platform HALs to override or modify the base implementation by simply redefining the appropriate macro.

Code for each HAL package is included in a variety of assembler and C source files which are compiled and added to the system library. As with the headers, there are standard names and conventions about what each file will contain. In addition to the sources, HAL packages, like all packages, contain a CDL file. This describes the package, listing the source files and expressing any dependencies it may have on other packages. Where necessary the variant and platform packages may define things that reconfigure the architecture HAL to support a different sub-variant. Specific to platform HALs are a set of MLT (Memory Layout Tool) files which define the layout of RAM, ROM and other memory regions on the target board, and the allocation of program sections to them. These are used to generate the linker script for building executables. There is a separate MLT file for each startup type (ROM, RAM etc.) that the platform supports.

Porting Process

There is no simple recipe for porting an eCos HAL. The exact set of steps and the order in which they must be done will vary from one target to another. This section will therefore only give some general hints about how to go about it and describe the general areas in which work is needed. The most common porting activity is to support a new board. This section will therefore concentrate on doing a platform port, with some references to variant ports. Architecture ports are rare and somewhat more involved.

The best approach to generating a new HAL is to copy an existing HAL and modify it to match the new hardware. This provides the HAL author with a framework for the new HAL. The preferable approach is to choose a platform HAL that uses the same variant HAL as the target board. Some files in each HAL are named for the target, so the first job is to rename those files to match the new HAL. The most prominent such files are the CDL script and the MLT memory layout files, but some of the source files may also have platform specific names. Within the CDL file it will also be necessary to rename the component names to match the new platform, this can usually be done very easily with an editor's search and replace operation.

The next step is to update the package database to refer to the new HAL. There are two entries that need to be added. The first is a *package* entry for the new platform HAL, which can usually be cloned from an existing entry. The second is a *target* entry for the board. It is in this target entry that the various HAL packages are brought together to form a complete HAL for any given board.

After doing this we now essentially have a complete copy of the original HAL under another name. We now need to start modifying the code to support the new target. This can range from simply changing a few constants for the memory controller, to a wholesale rewrite, depending on how close the current code is to the target board.

The first thing to change is the initialization code. This is nearly always done in the platform HAL since this is where detailed knowledge of the layout of memory devices such as RAM and ROM are known, and where device registers or PCI bus windows should be mapped. Initialization is divided between a number of assembler level macros and C functions. The assembler macros need only get the system up to a level where it can execute C code safely. This usually means initializing the memory controller and maybe the MMU and caches. The remaining initialization can usually be left until we reach C code, where it is easier to do the more complex initializations required by things such as PCI bus interfaces.

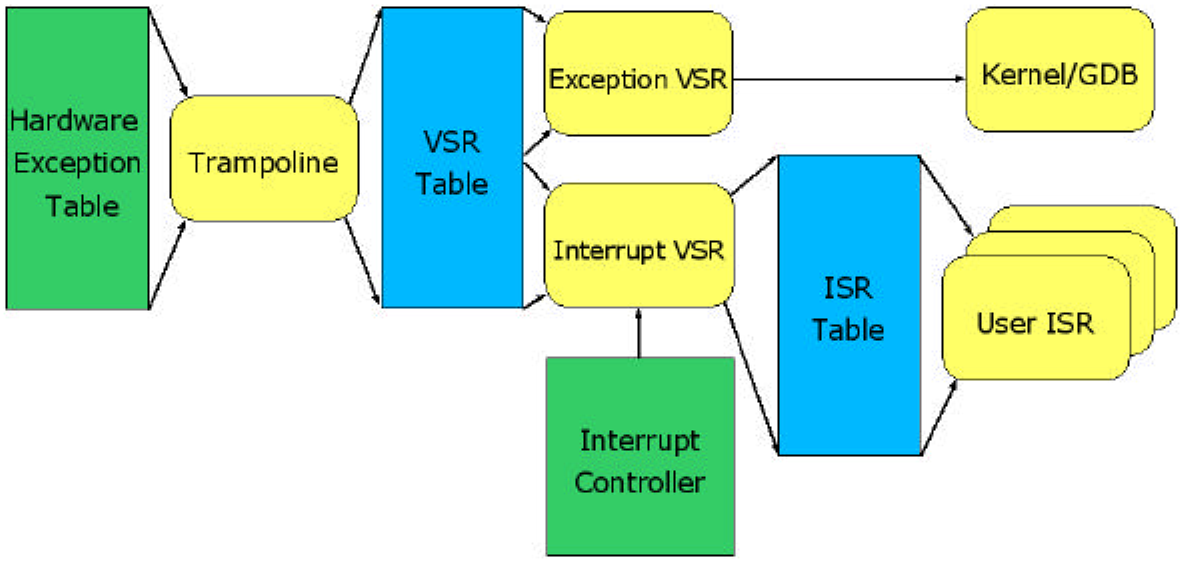


Figure 8 -- Exception and Interrupt Architecture

Figure 8 shows the structure of the eCos exception and interrupt architecture. The mechanisms used to enter exception handlers in different architectures vary enormously. The purpose of the eCos system is to provide a common, portable mechanism that can be applied to all architectures and platforms. Attached to the hardware interrupt vectors are a set of *Trampoline* routines that translate the hardware entry mechanism into an indirect jump through the *VSR Table*. This trampoline may be as simple as a single instruction, as on the ARM, or may be a handful of instructions that interrogate a CPU register and decode it into a table index, as on the MIPS. The VSR table entries point to an assembly code Vector Service Routines (VSRs) which know how to handle that kind of exception or interrupt. Exception VSRs usually just pass the event up to the kernel, or to the GDB remote protocol stubs, if present, which will usually result in GDB returning with a break event. The interrupt VSRs need to interrogate the interrupt controller and decode what it tells them into an *ISR Table* index, from which a pointer to a (usually C) Interrupt Service Routine (ISR) is fetched and executed. It is this translation that the platform HAL needs to implement, which it does by defining the mapping between the interrupt controller and the ISR table, and providing an assembler level macro to do the decoding.

In addition to decoding interrupts during interrupt processing, the platform HAL must also define macros that operate on the interrupt controller to mask and unmask each individual interrupt, acknowledge interrupts if necessary and configure them.

The single most important interrupt source for eCos is the clock interrupt. This provides the kernel with a time base by generating interrupts at a fixed frequency. In some targets it is possible to use an architecture defined timer (MIPS and PowerPC do this for example), and in others there is a common timer device integrated on-chip. However, for most platforms an external timer device must be programmed to generate these interrupts.

The clock can be set to tick at any frequency, depending on the ability of the CPU to handle the interrupt rate. The clock resolution, the number of nanoseconds per tick, is expressed as a fraction to allow frequencies which cannot be expressed as a whole number of nanoseconds, such as 60Hz, to be represented. It is the platform HAL's job to define the *numerator* and *denominator* of this fraction, together with the *period*, which is the initialization value for the timer device. In many HALs it is fairly easy to calculate the *period* from the resolution in the CDL script.

To program the clock, the HAL must define a macro to initialize the timer device. It must also define a macro to reset the device after an interrupt – where there is no need to do anything this macro may be left empty. A macro to read the number of ticks since the last interrupt must also be defined; some simple arithmetic on this value, together with the period and resolution allows accurate time intervals to be calculated. A final, optional, macro implements a microsecond busy delay loop, using the timer, for use in device drivers and other places where very short delays are needed.

eCos needs access to low level serial IO supplied by the HAL. This is used for diagnostic messages and for the GDB remote debug protocol. It differs from fully buffered interrupt driven serial I/O which is provided by a separate device driver. The low-level serial support is mostly handled by RedBoot, and is exported to the application either though the virtual vector interface or system call traps.

The HAL support for serial IO is slightly complicated by the need to support the virtual vector interface, but this is mostly boilerplate code and can just be copied. Apart from defining initial baud rates in the CDL, the HAL author simply needs to implement initialization, getc() and putc() routines. When a program is running, GDB can send a Ctrl-C character to interrupt it and to support this the HAL routines also define a function that is called from the interrupt subsystem to test for an incoming interrupt.

Cache support is mainly handled in the variant HAL rather than the platform, since caches are integrated into the CPU. In some architectures, where the main cache operations are part of the instruction set, all the variant HAL needs to do is define the cache dimensions: total size, line size and number of ways. Where there is no architectural definition for the cache, the variant HAL must also define the cache operations. These are split into two groups, those that operate on each cache as a whole, and those that only operate on groups of lines. Global cache operations are those that enable or disable the caches as a whole, invalidate or flush the entire cache. The line operations take an address range and apply an operation to just the group of cache lines that cover that range. Since not all hardware implements all of these operations, these macros are only defined if the underlying support is present.

The last thing that a platform HAL must define is the memory layout of linked executables. This is defined in the MLT files which consist of an `.ldi` file and a matching `.h` file. The `.ldi` file defines the location and size of the ROM and RAM memory that will be used by the application and the allocation of program sections (such as `.text`, `.data` and `.bss`) to them. The `.ldi` file is combined with a more generic `.ld` file, defined in the architecture or variant HAL, to generate a `target.ld` linker script file in the install directory, which is then used to link any application. The `.h` file is a header file that application and system code can include to access memory layout related definitions.

A separate set of MLT files are defined for each startup type supported by the platform, for example: ROM – applications that run out of ROM, RAM – applications that run entirely in RAM, or ROMRAM – applications that copy themselves from ROM to RAM before running. The memory layout for each of these startup types is usually arranged so that a ROM-resident application like RedBoot does not use the same memory as a RAM application, allowing them to coexist in the same machine.

Once all of these functional areas have been addressed it should be possible to configure and build either eCos or RedBoot for the target board. It is usually recommended that HAL authors start by trying to bring RedBoot up on any board first. The RedBoot environment is much easier to work in, being interrupt- and thread-free, and once RedBoot is running, the job of running and testing eCos is much easier.

The first problem to solve when porting RedBoot to any board is how to get it into memory in the first place. With luck there may be an existing ROM monitor that can be used for this purpose. Alternatively it may be necessary to use a JTAG or BDM debugger. Once the executable can be loaded the next thing to get working is the initialization code which sets up the memory controller and the serial IO code. Once these are working then RedBoot will generally start up and run.

Once a RAM version of RedBoot is running it is necessary to get a ROM-resident version that starts from the reset vector running. This will usually need programming to ROM or FLASH memory. If FLASH is present then the appropriate FLASH driver should be implemented (see later) and the ROM version of RedBoot programmed from the RAM version. However, care must be taken here to avoid rendering the board unbootable; this is where a JTAG/BDM interface is useful since the board can always be rescued.

Once RedBoot is running, it is then a fairly easy job to get eCos running. The main things that need doing are programming the interrupt controller and the timer. The eCos executable can be loaded over the serial line using GDB and debugged like any other program. It is normal to choose a subset of the standard tests that come with eCos for this initial bring-up: `tm_basic` for testing the functionality of the clock and the rest of eCos, `clocktruth` for testing that the clock period has been calculated correctly, `except1` to check that exception handling is working.

Additional Packages

In addition to the HAL, there are a number of other packages and subsystems that may need porting to provide full support for a given platform. These are mostly device drivers except for PCI bus support, which is an addition to the HAL. A device driver infrastructure exists for FLASH memory, serial and Ethernet device drivers. The infrastructure divides all drivers into a generic driver for a particular device, or class of devices, plus a platform specific package that customizes the generic driver to a particular platform. As with HALs, the standard porting technique is to copy an existing driver and modify it.

eCos Resources

The main eCos website is at http://ecos.sourceware.org. This site contains the latest stable release of eCos, as well as the CVS repository for ongoing development. Full documentation, a FAQ , lists of supported hardware and third party contributions are available. The site also hosts a number of mailing lists for users and developer interaction.

eCosCentric employs most of the original developers of eCos, and offers a number of products that build on the public releases. These include CD ROM versions of the stable releases which include pre-built toolchains; eCosPro, a tested and supported release for specific targets; and general support, and consultancy services. eCosCentric's website can be found at http://eCosCentric.com.