Embedded application development using eCos

John L Dallaway
eCosCentric Limited
Cambridge UK
jld@ecoscentric.com

INTRODUCTION

eCos [1] is a portable, open source, royalty-free, real-time operating system (RTOS) that is well suited for resource-constrained embedded applications. The advent of low cost 32-bit system-on-chip solutions incorporating limited-capacity Flash and RAM components has led to renewed demand for small-footprint RTOS components. eCos addresses this requirement by providing a rich set of run-time features which may be configured to match application requirements precisely. One of the key distinguishing features of eCos is its advanced configuration system that provides the infrastructure necessary to define the configurable features of the operating system, specify dependencies between these features and infer configuration changes that may be required to produce a self-consistent version of the operating system.

This paper provides an illustration of embedded application development using eCos. It serves to guide the reader through each of the engineering activities involved in working with eCos including operating system configuration using the eCos Configuration Tool, cross-platform debugging via an Eclipse-based IDE and application deployment in Flash memory. The ease with which eCos may be configured for a specific application and subsequently optimised is emphasised.

CONFIGURATION

Terminology

The eCos run-time code is made available as a set of building blocks referred to as *eCos packages*. Each package provides C/C++ source code, header files, documentation and a configuration script which describes the package and its configurable features. The configuration script is written in a language developed specifically for this purpose - the *Component Definition Language* (CDL). Each eCos package exports a well-defined API and has an associated version identifier. This allows for the updating of an individual package within an eCos repository independently of other packages. Configuration of eCos at a macroscopic level is achieved by selecting a set of hardware-specific eCos packages appropriate for a particular hardware platform and a set of hardware-independent eCos packages appropriate for a particular application area.

Each package provides one or more configurable *components* which may be closely linked via shared data or other internal interfaces. Each component provides a number of configurable *options* to control operating system behaviour at a microscopic level. These options may be Boolean, allowing a specific behaviour to be enabled/disabled, they may provide a numeric or string value, defining the length of a data buffer for example, or they may combine the two, providing a value when enabled but also allowing the associated feature to be disabled. Configurable options are implemented in the eCos source code as C pre-processor macros. The values of these macros are defined in header files generated by eCos-specific configuration tools.

Use of the pre-processor for configuration in this way allows for code size optimisation by reducing the semantics of specific source code functions, providing gains beyond those offered by linker garbage collection.

Host tools

eCos packages and their associated configuration options are manipulated by the application developer using a graphical *eCos Configuration Tool*. This tool runs on a development PC (Windows or Linux) and provides a view of the entire configuration space in the form of a tree. This view allows the developer to comprehend the large number of configurable options provided by the operating system. The properties of the currently selected configuration option are presented, including default value, value constraints and a short description of the option which assists the developer in selecting an appropriate value. The value itself may be modified using either a check box, a text box, a drop-down list box or increment/decrement buttons as appropriate. When a configuration option is modified, an inference engine evaluates the change and determines whether all logical dependencies between configuration options are still satisfied. If necessary, the inference engine will propose additional configuration changes to maintain the integrity of the configuration space. The developer is at liberty to accept the proposed changes or to make alternative changes to resolve the conflict.

A command line version of the tool (*ecosconfig*) is also available. This tool allows for scripted configuration of eCos or may be used for interactive configuration by developers with a preference for command-line tools.

As one would expect, eCos configurations generated using these tools may be saved to file for later retrieval. Other facilities allow for the exporting of configuration changes relative to one of the pre-defined *template* configurations included within each eCos source code repository. These changes may then be imported into another eCos configuration, allowing for the merging of configuration data.

COMPILATION

Following configuration, the eCos host tools may be used to generate an *eCos build tree* which is populated with a number of GNU makefiles, one for each eCos package included in the configuration. Compilation of eCos is effected by invoking GNU *make* on a top-level makefile that references each of the package makefiles in turn. The source code is compiled using a version of *GCC* built for cross-compilation to the appropriate processor architecture. The resulting object code is placed in an *eCos install tree* in the form of a library archive file and a set of ancillary object files suitable for static linking with application code. A number of linker scripts are provided, allowing application code to be linked for execution from RAM for debugging purposes or to be linked for execution directly from Flash memory for deployment in the field. A third option involves compiling eCos with optional startup code that relocates application code from Flash to RAM during early initialisation to achieve higher performance. A number of header files are also emitted by the eCos build and placed alongside the library in the install tree. These header files provide the API definitions by which application code can make calls into the library. They also define the value of each configuration option within the library, allowing application code to adapt its own behaviour to accommodate the configuration of eCos against which it is being built.

DEBUGGING

The downloading and debugging of eCos applications is typically achieved using *GDB* running on the development PC and communicating over a suitable channel such as RS232 serial using the *GDB remote protocol*. This approach relies on a GDB stub which must be installed on the target for execution at system reset. The eCos repository includes thread-aware GDB stub code for all the major embedded processor architectures and this code may also be included in the *RedBoot* debug and bootstrap firmware. RedBoot provides additional facilities for debugging over TCP/IP Ethernet and for managing code, data and filesystem images in Flash memory. Both RedBoot and the GDB stub are based on the eCos hardware abstraction layer, allowing rapid porting of this software to any eCos-enabled platform.

Hardware-based debug solutions using a JTAG or BDM interface may also be used with eCos. Such solutions rely on hardware initialisation via an appropriate hardware setup file to mirror the low-level board initialisation which would otherwise be performed by startup code in the GDB stub. A hardware-based debug solution is essential where the target hardware platform provides insufficient RAM to accommodate the application code. In such cases, the application must be downloaded to Flash for debugging purposes.

A number of graphical front-ends for GDB are available and many developers find such tools more convenient than the GDB command line. The well-known Eclipse IDE initiative includes a C/C++ Development Tools project (CDT) [2] and the debug support provided by the CDT plug-ins may be adapted for cross-development purposes. CDT retrieves target state, such as the current value of local variables, on a just-in-time basis and is therefore able to retain a responsive user interface when working with remote targets.

DEVELOPMENT ILLUSTRATION

Example application

For the purpose of illustration, we can envisage a small networked appliance which is capable of serving a hierarchy of digital image files to a remote web browser. The appliance connects to a local network via Ethernet. Images are stored in Flash memory. This appliance will be prototyped on a low-cost processor evaluation board which incorporates both an Ethernet port and a Flash memory part with sufficient capacity for holding multiple images.

The significant run-time components for this application are as follows:

- A TCP/IP stack
- An HTTP server
- A Flash-resident filesystem
- A file transfer mechanism to populate the filesystem

eCos packages are available which provide each of these features and the first stage of the eCos configuration process is to select the packages that will be necessary to support this application. Of those filesystems currently ported to eCos, the Journalling Flash FileSystem (JFFS2) is the most appropriate. JFFS2 is a log-structured filesystem specifically designed to reside in Flash and survive power failure during file write operations without corruption.

Several TCP/IP stacks have been ported to eCos and the developer has the luxury of selecting either a stack based on the FreeBSD sources for high performance or a stack based on lwIP sources to minimise code and data requirements. In this example, the FreeBSD stack is used. A TFTP client is included with the eCos networking infrastructure and this can be used for transferring images to the board.

Macroscopic configuration

The initial selection of eCos packages is simplified by the availability of a number of configuration *templates*. Each template defines a set of hardware-independent packages appropriate for a specific application area. In the case of a networked application requiring POSIX-like compatibility, the *net* template might be appropriate, but in this example the *default* template is used. The *default* template provides the following eCos packages:

eCos package name	Description
CYGPKG_HAL	eCos common HAL
CYGPKG_IO	I/O sub-system
CYGPKG_IO_SERIAL	Generic serial I/O support
CYGPKG_INFRA	Infrastructure

eCos package name	Description
CYGPKG_KERNEL	eCos kernel
CYGPKG_MEMALLOC	Dynamic memory allocation
CYGPKG_ISOINFRA	ISO C and POSIX infrastructure
CYGPKG_LIBC	C library
CYGPKG_LIBC_I18N	ISO C library internationalization
CYGPKG_LIBC_SETJMP	ISO C library non-local jumps
CYGPKG_LIBC_SIGNALS	ISO C library signals
CYGPKG_LIBC_STARTUP	ISO environment startup/termination
CYGPKG_LIBC_STDIO	ISO C library standard input/output functions
CYGPKG_LIBC_STDLIB	ISO C library general utility functions
CYGPKG_LIBC_STRING	ISO C library string functions
CYGPKG_LIBC_TIME	ISO C library date/time functions
CYGPKG_LIBM	Math library
CYGPKG_IO_WALLCLOCK	Wallclock I/O support
CYGPKG_ERROR	Common error code support

When the *eCos Configuration Tool* is invoked, a configuration based on this default template and the drivers associated with the default target hardware platform is automatically created in memory and presented to the developer for further manipulation (figure 1). If an alternative template or target is required, these may be selected via the templates dialog box. In addition to the eCos packages provided by the default template, the following packages are required:

Additional eCos package name	Description
CYGPKG_IO_FILEIO	Generic file I/O support
CYGPKG_IO_FLASH	Generic Flash support
CYGPKG_FS_JFFS2	JFFS2 filesystem
CYGPKG_CRC	CRC support
CYGPKG_COMPRESS_ZLIB	Zlib compress/decompress support
CYGPKG_LINUX_COMPAT	Linux compatibility functions
CYGPKG_NET	Generic networking support
CYGPKG_NET_FREEBSD_STACK	FreeBSD TCP/IP stack
CYGPKG_IO_ETH_DRIVERS	Generic ethernet driver support
CYGPKG_HTTPD	HTTP server

These packages are added via the packages dialog box. Note that some of these packages are not obvious direct requirements for this application but are required by other packages which are to be added. For example, the JFFS2 filesystem package requires generic file I/O, generic Flash I/O, CRC, Zlib and Linux compatibility functions provided by other eCos packages. If a developer were to add the JFFS2 filesystem package to the eCos configuration without these supporting packages, then the eCos Configuration Tool would indicate that the required packages are missing and should also be added.

Microscopic configuration

When adding new eCos packages to an existing eCos configuration, a number of conflicts may be reported which concern individual configuration options within the packages. For example, the JFFS2 filesystem requires that the generic file I/O package provides *inode* support. Such support is not required by all filesystems and so the generic file I/O package does not provide this support by

default. In this case, the eCos Configuration Tool can use the information provided within the CDL scripts of the various eCos packages to infer that the generic inode support option (CYGPKG_IO_FILEIO_INODE) must be enabled. A conflict resolution dialog box is presented to the developer, advising which options should be modified and allowing the developer to resolve the conflicts with a single mouse click.

In addition to those configuration changes necessary to resolve configuration conflicts, there will be a number of configuration changes necessary to support specific application requirements. In the case of this example application, a web browser may attempt to retrieve many image files concurrently and the maximum number of open files permitted by eCos (CYGNUM_FILEIO_NFILE) should be increased from the default value of 16. A number of options which provide an enhanced debugging experience may also be beneficial in an initial configuration. eCos assertion and tracing support should be enabled (CYGPKG_INFRA_DEBUG) and compiler optimisation should be disabled (CYGBLD_GLOBAL_CFLAGS).

Having manipulated the eCos configuration to a point where it is expected to serve all application requirements, the configuration is saved. At this point, the eCos build tree is generated by the eCos Configuration Tool and populated with makefiles. The eCos library can then be built directly from the host tool.

Application project creation

The Eclipse platform with C/C++ Development Tooling (CDT) plug-ins provides a powerful environment for C/C++ application development. Although CDT is not specifically designed for cross-platform development, modified CDT plug-ins are available to simplify the process of downloading code to a remote target when launching a debug session.

CDT provides options for creating both *Standard Make* and *Managed Make* projects. Standard Make projects incorporate a regular GNU makefile that the developer must maintain. With Managed Make projects, CDT generates and maintains an appropriate GNU makefile automatically as source files are added to and removed from the project.

An eCosCentric plug-in [3] may be used to simplify the configuration of a Managed Make project suitable for building an eCos application. The developer creates a new C/C++ Managed Make project via a *New Project* wizard and specifies the *eCos Executable* project type. Following creation of the project, it is necessary to provide the name of the GCC cross compiler and the location of the *eCos install tree* containing the eCos library and header files to be used within the project. These parameters are specified using project macros and may have different values for *Debug* and *Release* configurations if required. The name of the GCC cross compiler must be provided separately to allow CDT to index the project using header files provided by the compiler. Such indexing occurs outside the context of code compilation.

Once the project has been configured correctly, application source files are added to the project and edited as required. In the case of the networked appliance under discussion, this will include handler functions for the various HTTP requests that are used to navigate image directories and serve thumbnail images. The application code may be compiled and linked against the eCos library on demand. The build system refers to compiler and linker options defined in the eCos install tree to ensure that application code is built in an appropriate manner for the intended target hardware. Any errors and warnings issued by the compiler are recognised by CDT and are used to generate a problem list and to provide annotations in the source code editor.

Application Debugging

Debugging using CDT is achieved by creating a *debug configuration* that links a project and a generated executable file within the project with launch parameters including the name of the GDB tool and the I/O port to be used for communication with the remote hardware. A debug session may then be launched with reference to this debug configuration. On launch of a debug session, the executable code is downloaded to the remote target and CDT switches to an alternative arrangement of views within the Eclipse workbench that is more appropriate for the manipulation

and monitoring of target state (figure 2). Breakpoints may be specified at the source code level. Variables, registers and memory regions may be monitored as the developer steps through code at the source code level or machine instruction level. Diagnostic output from the application is passed up the debug channel and presented within the CDT user interface by default.

Debugging may reveal issues that must be addressed in the eCos configuration. For example, a diagnostic message indicating that the TCP/IP stack is unable to allocate sufficient memory from its own memory pool may be observed. The eCos configuration would then be modified to increase the size of this memory pool (CYGPKG_NET_MEMPOOL_SIZE).

Configuration optimisation

Following debugging and the successful execution of the application code, further optimisation of the underlying eCos configuration is usually possible. It should now be clear precisely which pieces of functionality are required by the application and which can be discarded. In the case of the networked appliance used as an example, several eCos packages that form part of the default template on which the eCos configuration is based are not required and may be removed:

Unused eCos package name	Description
CYGPKG_IO_SERIAL	Generic serial I/O support
CYGPKG_LIBC_SETJMP	ISO C library non-local jumps
CYGPKG_LIBC_SIGNALS	ISO C library signals
CYGPKG_LIBM	Math library
CYGPKG_IO_WALLCLOCK	Wallclock I/O support

Further optimisation involves navigating through the remaining nodes of the configuration tree presented by the eCos Configuration Tool and making appropriate adjustments. Some examples of the optimisations that may be employed for this application include disabling the HTTP server's system monitor (CYGPKG_HTTPD_MONITOR), disabling the Zlib stdio utility functions (CYGFUN_COMPRESS_ZLIB_GZIO), disabling diagnostic support within the generic Ethernet support package (CYGDBG_IO_ETH_DRIVERS_DEBUG) and disabling the *long long* variants of utility functions within the C library (CYGFUN_LIBC_STDLIB_CONV_LONGLONG).

By optimising the eCos configuration in this way, a 46kB reduction in compiler-optimised code size may be realised for the example application. Further optimisation is possible by examining the available eCos configuration options in more detail.

Following eCos configuration optimisation and re-testing, eCos assertion and tracing support should be disabled and compiler optimisation re-enabled. Finally, it is advisable to build the eCos test suite against the optimised eCos library and execute all tests using the test execution facility of the eCos Configuration Tool to verify correct behaviour.

Application deployment

Once an optimised build of an eCos application is running correctly from RAM, the eCos configuration may be modified to allow for booting directly from Flash memory (CYG_HAL_STARTUP). Configuring eCos for ROM startup results in the following changes:

- An alternative linker script is selected that places the code in Flash memory
- Machine initialisation code is introduced in the hardware abstraction layer to eliminate any dependency on a bootloader
- Startup code is introduced to copy pre-initialised static data into RAM for subsequent modification by the application
- Diagnostic output is redirected to a serial port

When the application is rebuilt against the revised eCos configuration, the resulting application image is of a form suitable for programming into the base of Flash memory for immediate

execution on system reset. A version of RedBoot executing from RAM may be used to download the application image into RAM and then write it to Flash, replacing the Flash-resident bootloader or GDB stub.

An alternative deployment option involves using RedBoot to write a version of the application image intended for RAM startup into an uncommitted region of Flash. RedBoot may then be configured to copy the application image from Flash back into RAM and execute it at system reset. This may be achieved by defining a RedBoot *boot script*. An advantage of this approach is that it allows for in-field updating of the application and/or associated data by attaching a terminal to the board and using it to disable the boot script. RedBoot may also provide power-on self test (POST) functionality where appropriate.

DISCUSSION

This paper has illustrated the steps involved in working with eCos from the perspective of an embedded application developer. The host tools and underlying configuration technology employed by eCos allow the developer to comprehend the large number of configurable options provided within the operating system source code. The developer is therefore able to customise the operating system to precise application requirements and achieve optimised use of available hardware resources. Multi-threaded applications may be readily deployed from Flash on hardware platforms featuring as little as 32kB RAM using these techniques. Where more memory is available, RedBoot bootstrap and debug firmware may be deployed for in-field updating of application and data images.

REFERENCES

[1] http://ecos.sourceware.org
[2] http://www.eclipse.org/cdt
[3] http://www.ecoscentric.com/ecos/eclipse.shtml

Presented at the Embedded World 2007 Conference, Nűrnberg, Germany

Copyright © eCosCentric Limited 2006

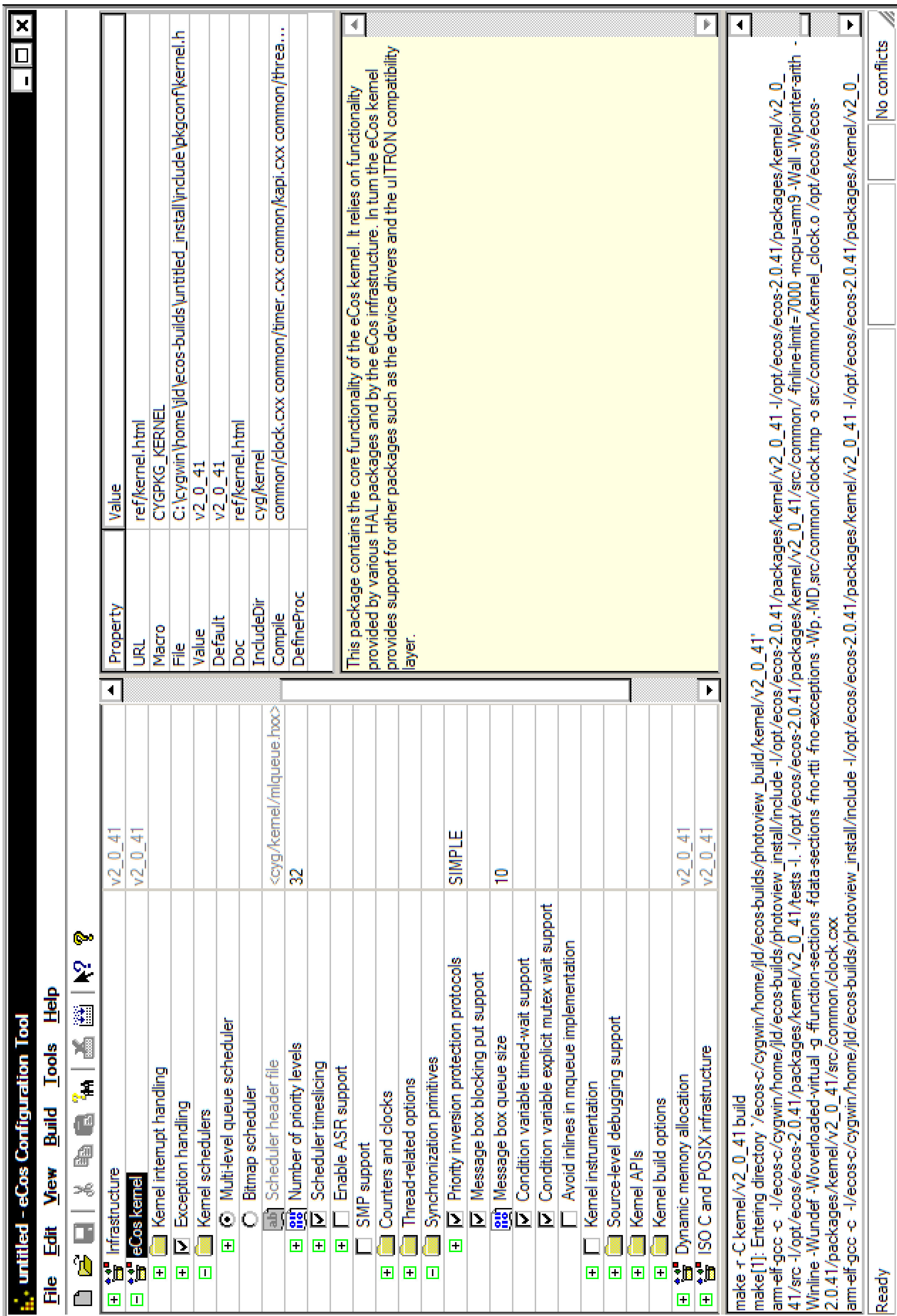

Figure 1. eCos Configuration Tool

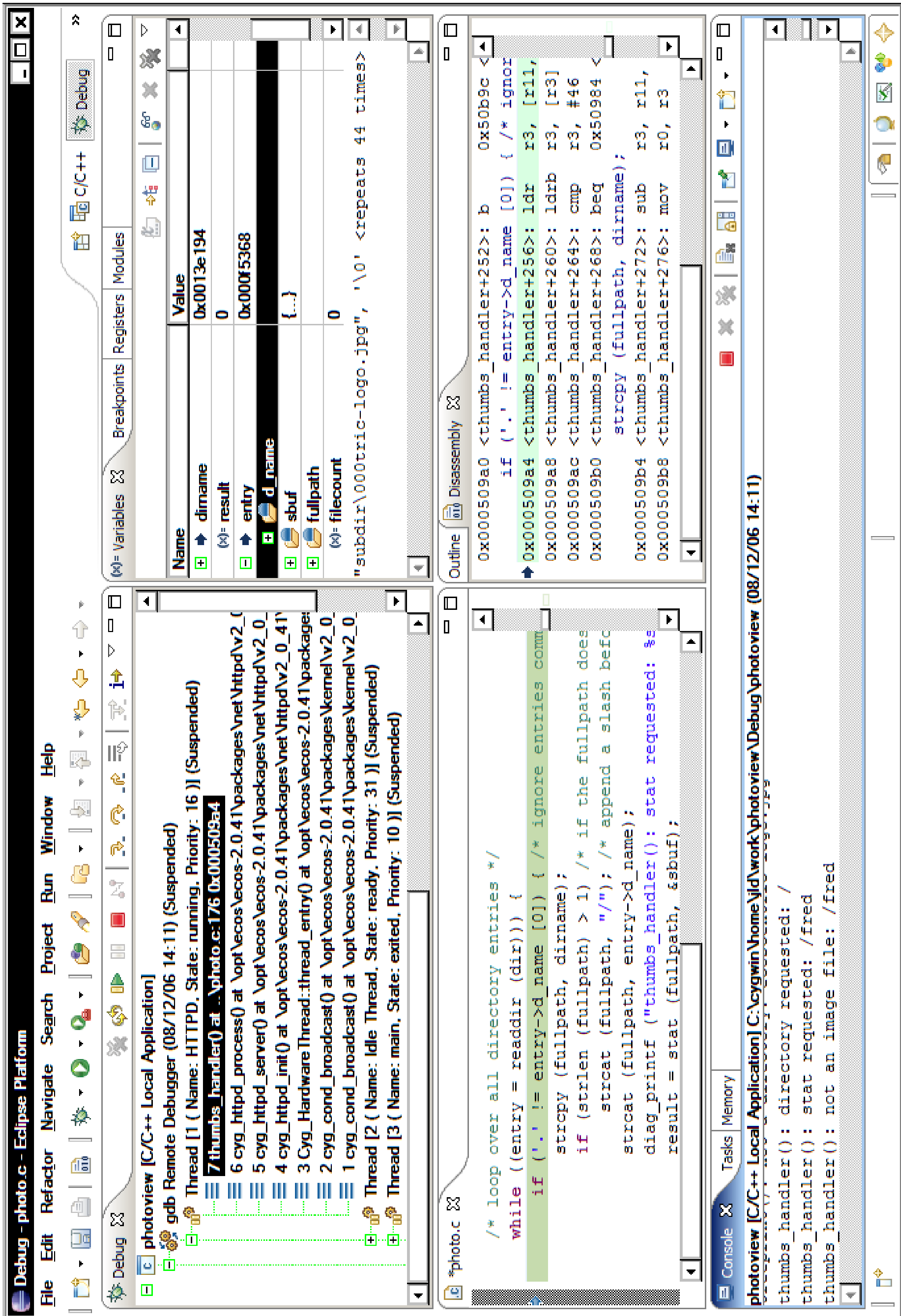

Figure 2. CDT debug perspective