Smaller, faster, open source, free: the eCos RTOS

Jonathan Larmour

jifl@eCosCentric.com

http://www.ecoscentric.com

Barnwell House
Barnwell Drive
Cambridge
CB5 8UU
United Kingdom
+44 (0)1223 245571

Abstract

"*The freely available open source eCos RTOS is designed to efficiently operate in both small 32-bit embedded systems where memory is measured in kilobytes not megabytes, and large systems with multiple protocol stacks, network services and peripheral devices. In this paper, we look at the benefits of eCos, such as its size, speed and configurability, how it achieves them, and how best to exploit these in your embedded applications*."

Introduction

eCos, the embedded Configurable operating system, started life ten years ago but is now one of the most popular embedded real-time operating systems being deployed – CMP's Embedded Systems Programming magazine ranked it in the top ten based on world-wide market share.

The main differentiating factors of eCos from its competition that have made it successful are fairly obvious:

- *Open Source*: eCos is publically available and freely downloadable from its project web site http://ecos.sourceware.org
- *Zero cost*: There are no licence fees, and no royalties
- *Highly configurable:* The functionality, sizing, implementation choices, and semantics are configurable to an extremely fine grain.

The latter aspect is the central focus of this paper. We shall explore ways in which an eCos-based system can be configured to be smaller and faster than the well-known alternatives. By exploiting eCos' high degree of configurability, eCos can scale from a from a large SMP network processing system, with full BSD TCP/IP stack over Ethernet, DNS, SNMP, flash file system, POSIX threads, watchdog, real-time clock, serial drivers, occupying several hundred kilobytes, right down to a minimal system, occupying just 1Kb of ROM and less than 600 bytes of RAM, solely using the eCos configuration system.

Configurability background

First, it is necessary to explain some of the concepts around the eCos configuration system, so as to better understand its capabilities.

Many operating systems incorporate some degree of configurability. However usually this is only at quite a coarse grain, such as the incorporation of a network stack, or a whole file system, or a compatibility layer. While such flexibility is welcomed in general, it is still ill-suited to the demands of the embedded systems developer who requires a much greater degree of control over the behaviour, speed, complexity and ROM or RAM footprint of their application. Coarse-grain control of large high-level modules is insufficient.

Some operating systems give a mirage of configurability by allowing some features to be enabled or disabled using special system calls or by setting global variables. However the operating system must still retain all the code, and must perform additional run-time comparisons to check the values of the configurable settings.

Using a more resource-hungry operating system can result in wasting hundreds or even thousands of kilobytes due to either unused code, or code that is unnecessarily complex for its intended use. This translates to having to fit more Flash or RAM on the board, or using a higher speed processor – factors that can greatly affect both profit margin for the product, and the ability to sell it at a competitive price, especially for non-trivial product volumes. There is therefore a huge incentive to control the operating system's resources more carefully, and fine-tune them to the specific requirements of the application.

eCos Configuration Architecture

eCos provides a configuration system that allows fine-grained control of specific configuration points. The heart of the system is the *C*omponent *D*efinition *L*anguage (CDL) which allows the software within eCos to provide information on and dependencies between configuration points.

Configuration points can be at a coarse-grain with *Packages*. Examples of packages may be a TCP/IP networking stack or 16550 serial UART driver. Or at the finest level, they may be *Options*, examples of options being the size of the kernel idle thread's stack, whether assertion support is enabled, or whether a particular C library function is thread-safe. eCos presently has literally thousands of options!

To manage the potential complexity of having so many configuration points in the system, they are arranged hierarchically, with related options being able to be grouped into *Components*, which may themselves be configuration points. Components may contain further components, and are usually then grouped within a package, although components can even be set to act as parents to other packages in the displayed hierarchy if this may be the most appropriate way to present the grouping of functionality.

Configuration points have a type and a value. The type is usually boolean or data. In the case of data, it may a string or a number, and may in turn be constrained to a selection from an enumeration, or to a particular numeric range. They may be used to indicate that one or more source files need to be compiled if that option is enabled, and not otherwise. Most usefully, they can express constraints on other configuration points in the system, so that for example thread-safety is only relevant when the eCos kernel thread scheduler is enabled within the configuration.

Configuration points usually have descriptions, links to on-line documentation, they may indicate abstract properties, override default build rules with make fragments, contain a wide variety of expressions (CDL is an extension of a TCL interpreter) and there are many more useful properties. Clearly CDL is a very flexible and powerful tool for managing the configuration space.

Developers use the eCos Configuration Tool to navigate the options within its easy-to-use graphical interface (a command-line based tool is also available). When saving the configuration, the tool generates a set of C header files which contain C preprocessor defines corresponding to every configuration option that was enabled in the configuration as specified by the user.

Using this mechanism, the source files within eCos are able to use simple "#ifdef" or "#if" tests to adjust behaviour according to the configuration settings. While this approach is simple and familiar to developers, it is nevertheless flexible enough to fulfil the requirements of most configuration options.[1]

Each configuration point has an associated macro name used for evaluating the presence (tested with "#ifdef") or value (referenced with "#if" or just by direct use) of the configuration data. This flexibility is not limited to C and C++ source files, but is readily usable in assembler source files and linker scripts, or indeed any file capable of being preprocessed.

As well as generating these C header files when saving, the Configuration Tool will generate a directory tree structure with makefiles to allow the chosen configuration of eCos to be built, either within the graphical tool or on the command-line. The result of compiling eCos within this *build tree* is a new tree, the *install tree*, which includes header files, a library and some support object files, as well as a linker script.

The application may then reference the eCos APIs supplied by the header files and use the linker script to generate their final application. The linker script automatically causes the relevant libraries and support object files to be employed in the link.

---

1 There are some example of configuration points in eCos which have more demanding requirements in terms of the effect they have on the rest of the system. For those, it is possible to exploit CDL's flexibility as a scripting language, and by using custom make rules associated with the option.

eCos contains a large test suite containing hundreds of test programs, each program testing a wide range of functionality. As well as building eCos, the developer can build the test suite that goes along with that configuration. Not every test is built – only those potentially applicable to the chosen configuration. And for those tests that are built, each test can adjust what it tests and the expected behaviour in response to the settings in the configuration. This means that the developer can generate a test suite that is specific to their configuration, and can be used to validate the choices in the configuration and verify it works as expected.

Size efficiencies in the eCos design

From the outset of its design, eCos has adopted general principles which inherently improve the resource footprint.

The first and most obvious one results from eCos being open source, permitting the complete elimination of unused code and data using the C preprocessor as described earlier. A closed source binary-only operating system can never achieve the same savings, although nothing in eCos prevents the use of binary-only packages if they were still required, for example due to licensing constraints.

Perhaps the next greatest contributor to resource footprint minimisation is the design principle of linking the application and eCos operating system together into a single image. This approach is significantly different from most other operating systems, where the operating system kernel is a single monolithic image, forming an executive under which the separately linked application may be run.

Instead, by linking the application and eCos together, unused code files within the eCos libraries that are not (directly or indirectly) referenced by the application are simply never included in the final image using the well-known principles of library linking.

Furthermore, eCos exploits a feature in the GNU compiler and linker that was originally written specifically for eCos called linker garbage collection, sometimes known as selective linking. Linker GC takes the principles of library linking further using properties of ELF sections to ensure that not only are unused object files within libraries ignored in the final image, but that even for those object files that are referenced, any unused functions and data within them can be discarded.

eCos allows these benefits to be extended to the application itself as well. To allow a module to have linker GC operate on it, it should be compiled with the “-ffunction-sections -fdata-sections” flags to GCC. When linking the application, to indicate that the linker should perform a garbage collection pass GCC must be passed the flag “-Wl,--gc-sections”.

Finally, eCos was designed from the outset to support e*X*ecute *I*n *P*lace (XIP) allowing code to run directly from Flash or other directly addressable ROM, without needing to be copied to RAM first. Efficient design of Flash drivers ensures that even Flash erase and write operations need not interfere with the ability to run XIP as long as it does not directly affect the running image.

Speed efficiencies in the eCos design

A secondary benefit of fully linking applications and the eCos operating system is that there is no longer any real user-mode/system-mode divide. Instead system calls are now straightforward function calls, with a consequent significant reduction in overhead. Moreover, on processors with MMUs, making these “system calls” no longer requires any potentially expensive page table switching.

eCos has also been designed from the outset to have improved real-time response compared to many other real-time systems, by separating the interrupt handling sequence into multiple stages.

Interrupts are initially handled by a low level assembler routine termed a “Vectored Service Routine” (VSR) that is specific to the Hardware Abstraction Layer (HAL) for the architecture. Unless overridden by the developer, this will decode the interrupt source and call a C function previously registered as the Interrupt Service Routine (ISR) for the interrupt. This routine is usually executed with global interrupts disabled, although some architectures do provide for nesting of interrupts, permitting higher priority interrupts to interrupt lower priority ISRs. This approach will be familiar to most embedded developers from other operating systems.

But while it’s possible to handle the interrupt fully within the ISR, eCos allows for part of the interrupt handling to be processed in a “Deferred Service Routine” (DSR). This part of the interrupt handling sequence runs with global interrupts enabled. This ensures that ISRs can still handle events from potentially higher priority sources. DSRs do not pre-empt each other and are simply run in sequence, although if a single source generates multiple interrupts, its DSR need only be called once. Unlike ISRS, DSRs can perform operations like wake up threads waiting on kernel synchronisation objects.

In some cases, there are four levels of interrupt handling in total. An example of this in eCos is RX packet interrupt handling in the BSD network stack port. The HAL’s default interrupt VSR will handle the low level aspects, calling an ISR, which will in turn request that its associated DSR be called. That DSR wakes up a helper thread which will push the packet into the bottom of the network stack, and it will be in that thread’s context that most of the network stack processing itself (defragmentation, TCP/UDP protocol handling, routing, etc.) occurs. This helper thread does not need to run at high priority – the priority is in fact configurable – and so unlike many network stack implementations, the internal processing of the network stack can be pre-empted by higher priority application threads as needed. This ensures a greatly improved real-time response.

This design principle has been adopted across the eCos device driver implementations, namely that as much as possible of the interrupt processing and device handling is performed with global interrupts enabled. Furthermore, where possible it is done in the context of the thread which requested the driver operation, or failing that a helper thread, in order to ensure real-time behaviour. This then guarantees high priority to those threads that need it. Many other operating systems have a “single-threaded” kernel where once a system call is made, all pre-emption is impossible until the system call completes and returns from the system mode to user mode.

Keeping the code size down

Despite the inherent design features of eCos to minimise code and RAM usage, there is only so much that these techniques can achieve, even when features like selective linking are removing unused code and data. Instead eCos allows the developer to fine-tune the operating system configuration to fit perfectly with the requirements of the application.

There are various techniques that can be used to determine what can be trimmed. The easiest and most logical approach is to examine the eCos configuration with a hierarchical top-down approach where the developer removes unnecessary features from the configuration. First the developer can remove whole packages that are not used by the application. Then individual components, and then options within the remaining packages can be similarly analysed and treated, either being disabled entirely, or in the case of non-boolean options, having their values adjusted. The hierarchical display of the eCos Configuration Tool makes it well suited to the task of analysing the configuration.

Documentation for each option is displayed in the Configuration Tool and clarifies the impact the option has on program operation.

However the wealth of configuration options can make verifying the selection of each configuration option daunting. There are other techniques that can assist the developer in finding where resources are going, and therefore where the focus for streamlining the configuration should be.

The most useful tool to analyse code and data use is the ability of the development tools to generate a linker map. This file can be generated by the linker by passing GCC the flag “-Wl ,--Map,mapfile” where “mapfile” is the file name to use for the linker map. The linker map contains a precise list of what functions and data were incorporated by the linker into the program, their precise footprints, and what object files were used, which in the case of eCos are named conveniently to quickly identify which package they originated from. It also gives dependency information to show not just that a function was pulled in to the program image, but from where it was referenced that caused it to be pulled in. Note that some of the generated information is only identifiable if selective linking is used, but that is the default for eCos.

For example, the developer may be using JFFS2, the Journalling Flash File System, in their configuration, and may notice that various routines from the zlib package are being used to support built-in compression and decompression. The developer knows that his or her application does not want or need this ability. The linker map file can identify that the JFFS2 package was the reason for the use of zlib, and the configuration option enabling built-in support for compression in JFFS2 can then be quickly identified and disabled, eliminating the dependency. An alternative approach may have been to have scanned the packages loaded in the configuration, notice the unnecessary presence of zlib, and remove it from the configuration. At that point the configuration tool would detect a conflict due to the constraints expressed in the JFFS2 CDL, indicate to the user that JFFS2 is presently configured to use compression, and automatically suggest disabling the option controlling compression within JFFS2.

Perhaps in some cases it may not be clear what option needs to be adjusted to remove the dependencies, but the configuration tool allows the developer to examine an option and see not only what constraints it has on other CDL properties, but what other options have constraints on it. In the very worst case, as eCos is open source, the module could be identified from the linker map and the source file examined to determine if there are any options that may remove the dependency.

A good example of where the dependency may be hidden is with the printf() and scanf() family of functions. The default implementation conforms to the ISO standard, allowing use of floating point format specifiers. But these result in a large body of code being included, both in the printf()/scanf() functions themselves, but also as a result of functions from the floating point math library needing to be called to support them. Furthermore, on CPUs with no FPU fitted, floating point operations must be emulated requiring several more kilobytes of code. This may be entirely needless when a developer is fully aware they never intend to display floating point using these functions, just print occasional text output, possibly only for diagnostics.

Some embedded systems provide non-standard functions like iprintf() which is an integer-only printf. However, being non-standard, all callers need to use it. If just one caller does not, then the situation is worse as there is now both the full printf() implementation *and* the iprintf() implementation in the program image. Guaranteeing exclusive use of iprintf() can be tedious especially when reusing existing code from elsewhere, which is something good software engineers should be seeking to maximise.

Instead eCos provides a configuration option to disable floating point support within printf() and scanf(). This allows existing code to work without change. This is not the only case of code where the underlying implementation can be streamlined, even though the visible API remains the same. For example, it is worth examining the ISO C and math libraries' configuration for functions that can be configured for thread-safe behaviour. The developer may be aware that the functions are only ever called from one thread, or one thread at once.

Think of the alternatives

Sometimes it may not be a straightforward case of adjusting the configuration options visibly presented. An eCos developer needs to be aware of what alternative implementations exist within eCos.

A good example of this is TCP/IP networking stacks, where there is a fully featured network stack derived from the FreeBSD project. The BSD stacks are well known for their performance, reliability and standards conformance. They have been well tested and designed for maximum throughput. However the stack is very resource hungry. All the packet buffer pools and other buffers are configurable values, although reducing these will affect the performance of the stack. Below a certain point the developer will need to be aware of exactly what the network activity and demand is likely to be, otherwise there is the risk of the stack dropping packets simply due to insufficient buffer capacity. But there is not much that can be done to reduce the code footprint, other than disabling IPv6 if it is not to be used.

Instead eCos was designed with a plug-in network stack infrastructure, and it is possible to use alternative TCP/IP implementations. In particular, there is a port of Adam Dunkel's lightweight IP (lwIP) stack. This stack was designed from the outset for embedded systems, and while not as featureful as the larger BSD implementation, is capable of fulfilling most developer's requirements using a greatly reduced footprint, while still maintaining the use of the well known BSD socket API.

A second example where it is possible to get away with a simpler implementation of the same functionality is signals. eCos provides a POSIX compatibility package, which includes fully POSIX compliant support for signals, signal handling, signal actions, masks and synchronization with POSIX threads. But it may be that the developer only wishes to use signals in a simple way, and is content to do so from standard eCos threads instead of POSIX threads. In this scenario, a developer can remove the POSIX package from their configuration and replace it with a much simpler signal handling mechanism provided by a different eCos package which implements signals only to the level of compliance of the ISO C standard, rather than the much more heavyweight POSIX standard.

There are numerous other examples in eCos where different implementations with different size versus speed versus functionality trade-offs can be chosen.

Reducing code size in the toolchain

There are other steps that can be taken to reduce code size other than by adjustment of the eCos configuration. Most obviously, compiler optimisation should be enabled for both eCos and the application linked against it. Although there are a wealth of compiler optimisations, taking the broad brush approach, compiling with "-O2" will produce a good compromise of small optimised code. Optimisation levels of 3 and above will tend to increase code size. But in addition, appending "-Os" will make further optimisation changes that will reduce the space usage of the program, albeit possibly at the expense of performance. eCos itself defaults to building with just "-O2".

We have already mentioned the benefits of linker GC, and the developer should remember to supply the necessary flags on their own application's compile and link lines.

C++ users should ensure that C++ exceptions, and run-time type identification (RTTI) are disabled if unused as these will increase code (and data) use. The compiler flags "-fno-exceptions -fno-rtti" may be used for this.

Finally, there is one special case worth of note: for those using the ARM architecture, most ARM chips in use today support Thumb instructions, which occupy 16-bits instead of 32-bits. The resulting code may occasionally take more instructions as the instruction set is less powerful, but the greater code density will reduce the overall code footprint. eCos itself can be built in Thumb mode, and this can be readily achieved by changing just a single configuration option.

Reducing the RAM footprint

The first step to reducing the RAM footprint is largely to adopt most of the techniques in the previous chapter to reduce the code size. By removing code modules, the RAM footprint is also likely to diminish. Furthermore, examination of the linker script works equally well for data as code.

Many eCos device drivers, as well as the networking stack, use buffers with sizes that can be set in the configuration. Buffer size can be traded off against performance depending on the expected usage and throughput. Again choosing lwIP over the BSD stack will allow the developer to configure a reduced buffer size. lwIP can perform better when buffer sizes are smaller, due to having been designed from the outset for embedded systems.

But one key contributor to RAM usage in an RTOS is the thread stacks. It can be difficult to determine the maximum possible stack use by a thread so that stack has not been allocated unnecessarily. Fortunately, eCos provides some assistance in determining a good choice of thread stack size.

The eCos kernel contains a configuration option which, when enabled, will allow the stack use by each thread to be measured. The maximum amount of stack used can then be obtained either by calling a kernel API function for a particular thread, or by enabling a further option allowing the stack usage to be printed on the console when the thread terminates.

However some further input from the developer is still required as the kernel by itself cannot guarantee that every thread reaches the most stack hungry part of its execution path. Experienced developers will have a good insight as to which code path that should involve, and create the conditions for that code path to be followed. Subsequently the stack use can be measured.

There are additional factors to account for, beyond the measured stack use. The most important is that there must be additional room for the CPU context to be saved on the stack should there be an interrupt. As this may include a clock interrupt for a timeslice, this is easily possible. The HAL defines a structure named HAL_Saved_Registers in the hal_arch.h header file which is used to hold the CPU context. Room must be left for this structure on the stack. Indeed the size of that structure may be affected by the configuration. For example, some HALs can be configured to not save or restore the floating point registers. This allows these elements within the saved context structure to be omitted resulting in lower stack use. The fact they do not have to be saved or restored will also reduce context switch time and interrupt latency.

By default, when handling an interrupt, eCos will switch to a separate "interrupt stack" before executing ISRs and DSRs. This is so that space does not have to be reserved on each individual thread stack to allow for the possibility of running ISRs and DSRs, and thus makes thread stack use more deterministic. Use of the interrupt stack can be disabled as it does

slightly increase interrupt latency. The size of the interrupt stack is also configurable, and it should be large enough to execute the most stack hungry DSR as well as the most stack hungry ISR simultaneously. Fortunately since ISRs and DSRs are meant to be short, this is usually easy to determine.

Code in eCos sources may also create threads, each of which will obviously require a stack. The stack sizes can be set in the configuration, and the defaults are generally very conservative giving ample opportunity for a developer to reduce RAM footprint. Examples of system threads include the aforementioned BSD network stack helper thread used for the bulk of received packet and time-out handling, the kernel idle thread which is the lowest priority thread in the system, and threads used to provide network services such as an SNMP agent, TFTP server, HTTP server and so on.

Improving performance

Although we have already mentioned a number of ways in which it is possible to configure eCos for greater performance, such as disabling HAL floating point support, ensuring compiler optimisation is enabled, or choosing the BSD network stack over the lwIP network stack, there are a few other aspects worth covering that may aid developers looking for a performance boost.

One first step is to follow the suggestions in the preceding sections to reduce the code and data footprints. In many cases extra unnecessary features enabled in the configuration will result in extra code having to run to support those features, even if the features are themselves never used.

An obvious example is kernel timeslicing support. If timeslicing is enabled, this will result in the kernel clock interrupt handler needing to check if a new thread needs to be scheduled. However if the application will never create threads of the same priority, and as eCos is a real-time system, then in that scenario no thread can ever timeslice another. In that case, timeslicing may as well be disabled, resulting in reduced clock interrupt handling time. In fact, it would probably be desirable to select the bitmap scheduler implementation instead of the default multi-level queue based scheduler implementation. The bitmap scheduler is much simpler, smaller and faster but does not support threads of identical priority.

One further reason why reducing code and data footprint can be beneficial is because unused code and data may fill cache lines in the instruction cache or data cache, resulting in increased cache misses. By eliminating unused code, the used code is more likely to occupy the same cache line, i.e. increased cache locality. The overall effect will probably not be significant but may be non-trivial. Using higher levels of compiler optimisation in GCC such as -O3 can be tempting and will usually be beneficial, but there is a risk that performance may even be reduced, because at -O3 and higher, the compiler unrolls loops and performs automatic function inlining. Both these optimisations carry the risk of decreasing cache locality, unless using advanced features such as profile-directed optimisation as described below.

Cache locality can also be increased by using smaller instructions, such as ARM Thumb mode. The instruction set may be less powerful, but depending on the size of the inner loops in the program, the greater code density of Thumb mode on ARM may result in increased cache hits.

Certainly use of Thumb mode may be significantly faster than ARM mode if running directly (XIP) from a slow Flash or ROM. Whether it will be faster will depend on hardware properties such as the number of wait states, cache size, and the access width. If the access width is only 8-bits, it is highly likely that Thumb mode will be faster. Again, the value of eCos being highly portable and configurable is shown as switching eCos to Thumb mode requires

changing a single configuration option.

Another solution for poor performance when running code from a slow memory device is not to run XIP at all if sufficient RAM capacity is available. RAM is almost always faster. eCos terms this “ROMRAM” startup as the program is compiled for ROM but relocates itself to RAM at the very beginning of the boot sequence.

The classic approach to identifying regions of the program which would benefit from closer analysis for performance is to use a profiler. The public version of eCos supports timer-based profiling using the GNU profiler, gprof. Samples of the program counter are taken by a high frequency timer interrupt handler, allowing a statistical analysis of where the program spends its time to be built up.

More useful is the improved gprof support found in eCosCentric’s enhanced version of eCos called eCosPro. This allows basic block profiling, which is more accurate than timer-based profiling.

The data from the gprof support package within eCos can be uploaded to the host and then be analysed to identify areas where attention should be focussed for performance improvements.

Recent versions of the GNU compiler also support profile-directed compiler optimisations. The profiling data can be used to optimise code in two significant ways: branch prediction, so that in the common case a branch is not taken, which improves instruction cache hits from the current cache line; and for directing inlining, to determine when automatic inlining may or may not be beneficial at -O3 and above. As indicated earlier, inlining at -O3 may be detrimental to performance if it results in increased cache misses in the commonly executed case.

Conclusion

We have touched on many aspects of how eCos is able to avoid the overheads suffered by most alternative operating systems. The key advantage of eCos is the power of its unique configuration system. eCos can be configured to precisely fit the requirements of the application and no more. It is feature-rich, yet does not have to pay a penalty if the features are not used. Above all, it puts embedded developers in control.

Embedded developers want control over the code running in the products and loathe unnecessary bloat. Many would prefer to reinvent the wheel and write their own basic OS, rather than trust a traditional closed source operating system with little visibility; or even if the source is available, it may be too complex to work out how to customise it for the application’s requirements. With eCos there is no longer a reason to do this as the developer can maintain full control and visibility of the code, while also being able to control configuration choices in a simple way.

There are some disadvantages to the large degree of configurability: when talking to our customers we are often asked what eCos’ memory footprint is, or its interrupt latency, to which the answer can only be that it depends! But you can be confident that eCos is designed to provide precisely what your application needs.

The author is Chief Maintainer of eCos, and Chief Development Engineer at eCosCentric, the leading supplier of consultancy, training, development and support services for eCos. http://www.eCosCentric.com/