

HOW TO WRITE AN RTOS

A Guide for Anyone Pondering The "Make-or-Buy" Decision

This article considers four of the most frequently asked questions about RTOS. Firstly, it tries to answer the problem of performance versus code size and portability. Secondly, it deals with the problem of testing and verifying the RTOS and the application. Thirdly, it deals with the problem of characterizing the performance of a given RTOS. Finally, it presents the problem of tools interoperability (a RTOS alone is almost of no use).

As a vendor of the RTXC real-time kernel, Embedded System Products, Inc. routinely receives inquiries about its products. We receive not just requests for product information and literature but also questions along the nature of "how" and "why". Some inquiries come from prospects that have decided to use a commercial product. Others are still tossing around the hot potato of the "make-or-buy" decision. The questions they ask run the gamut from the very technical to the very mundane. Some questions are so probable that an article dealing with them may be useful to anyone contemplating either acquiring a commercial RTOS or building a custom kernel.

Fortunately, the points of their inquiries have as much to do with RTOS design in general as with the specifics of the RTXC implementation. So, they are probably fitting subjects for presentation in a publication where it is desirable to reduce commercial hype. In that vein, and taking into consideration the space available, a distillation yields four main subject areas that are the commonly asked. Ranking them in the order of their seeming importance to the inquirers, the issues are as follows:

- Performance, Code Size, and Portability
- Testing and Verification
- Characterization
- Tool Interoperability

Before beginning, let's agree on at least one definition. Whether one agrees with it or not (and I do not) the embedded systems industry seems to accept the following definition:

real-time kernel = real-time executive = real-time operating system (RTOS).

We could probably spend the space allocated for this article debating the correctness of that definition but I would prefer to move on to more important matters. This article will make use of the multiple equality just for simplicity's sake. So, please, don't inundate me with argumentative e-mail. This is another issue like the number of angels that can dance on a pinhead, it will never be decided.

PERFORMANCE, CODE SIZE, AND PORTABILITY

Performance, code size, and portability of an operating system are very interrelated. When one designs an RTOS, one of the basic design policies is that of whether the design will focus on performance or code size. They are often (but not always) mutually exclusive. If one designs for performance, the policies about RAM and ROM utilization often go out the window. The designer usually makes design choices on how fast a procedure can occur rather on how much memory it consumes. Code tends to have repetitive segments because it is faster to execute straight-line code than to call functions or subroutines that increase system overhead.

On the other hand, a design policy of minimizing code size will make extensive use of subroutines and functions to eliminate repetitious code segments. That saves ROM but it also degrades performance compared to straight-line code due to the overhead of setting up the stack on function entry and cleaning it up at the time of exit. Then, there is the question of scalability and how it affects code size. A scalable RTOS permits the user to configure it to contain only the services needed by the application, thereby minimizing the amount of code space required. If the scaling process can eliminate unneeded code, it is expected that the code will have improved performance over an unscaled version. Having source code to the RTOS greatly improves the prospects for scalability as opposed to trying to scale the kernel from an object library.

There are also considerations about RAM utilization that also weigh on performance. The single largest influence on an RTOS' use of RAM is the way it treats stack space for the various processes. For example, does an interrupt service routine (ISR) run on the stack of the interrupted process or does it have a separate stack? If one chooses the former, the RAM implications can be significant because each process' stack must be able to accommodate the worst case possibility of interrupt nesting. That design makes the

Can the RTOS on the target processor operate more effectively with a performance design or a memory size design for a given application?

ISR fast but it consumes a great deal of RAM if the number of processes is large. But, if the processor supports a large address space and the application has lots of RAM, the design is fast and works quite well.

On a small system without much RAM or a large address space, more prudent use of RAM for stacks can prevent exceeding the application's RAM budget. Consider the second design choice. If the designer chooses to operate ISRs on a separate stack, each one, at minimum, requires enough RAM for only one processor context. In addition, there must be a common stack large enough to handle N-1 processor contexts (N is the number of possible simultaneous interrupts) to accommodate ISRs. This idea saves a lot of RAM per process and for the application as a whole, but it requires execution of code to switch between stacks. The performance is lower compared to that of using the stack of the interrupted process, but only by the amount of time to effect the stack switch. Even on a processor having a larger address space, saving RAM can be an economically sound objective.

The issue of portability involves the code size and performance issues in that portability is largely a function of the language used to implement the RTOS. Assembly language usually consumes less code space than an equivalent operation coded in a higher level language such as C. Similarly, the assembly language code probably executes faster and is, thus, a higher performance design than the C code. Unfortunately, the assembly language code is not at all portable and it will probably take longer to develop than the same functionality implemented in C. (We haven't discussed the cost issues here)

On the other hand, RTOS code written in C is very portable to other processors, even those from different manufacturers. If portability is a concern of the user, a C implementation may be preferable even at the expense of lower performance and larger code size. However, by selecting a good C compiler with a good optimizer, it is possible to minimize code size and obtain performance that is almost as good as assembly language.

This topic boils down to one major question: Can the RTOS on the target processor operate more effectively with a performance design or a memory size design for a given application? The answer gets into considerations of processor choice, nature of the application, evolution of the product, and ultimately, system cost. Remember, high performance often means more RAM usage and that extra RAM costs money. If it's a small processor, that could be a problem. If a larger processor, it is probably less so. Remember also that the portability decision when making a custom kernel has a significant effect on development time, and consequently, on cost as well.

TESTING AND VERIFICATION

Regardless of whether one "makes" or "buys" an RTOS, one thing is certain: it is going to be a part of

the application. Consequently, it should be tested. How does one go about testing the RTOS and verifying its proper operation? If one chooses a commercial product, the expectation is that it is already tested. The expectation is often not reality because vendors of commercial kernels treat this important aspect of their products quite differently. Some are thoroughly tested according to a formal procedure and are highly reliable. Others cut short this process, preferring instead, to do some basic testing and then put the RTOS into the hands of their users for them to find and sort out any problems.

If a user has a tight schedule or must maximize productivity of the development personnel, a well-tested RTOS, though it may have a higher price, may be a cost-effective choice. The developer who can focus on implementing the application has a higher return on investment than the developer who spends valuable project time hacking on an untested commercial RTOS or an unproved custom kernel trying to make it work properly. Of course, if one has ample resources and no constraints as to time-to-market, the challenge of debugging an RTOS while trying to get a product finished can be a delightful programming experience. Unfortunately, it's not too rewarding.

While testing and verification are areas where commercial vendors sometimes cut corners, custom kernel developers almost always do so. That they choose not to do extensive testing is not surprising, given the costs, time, and labor involved. Can they do it? Certainly, so long as they realize that it is not unusual for good testing and verification to cost about 3-4 (or more) times that of developing the RTOS code. It all depends on how one chooses to test the code.

There are various kinds and degrees of testing methodologies available to an RTOS developer. The decision on which ones to use reflect not only the needs of the application and the budget, but also the willingness of the developer to put up with tracking down RTOS bugs when they occur. In general, one has to look at the complete testing process with respect to its various components.

At first, there are usually unit tests conducted by the developing programmer that confirm the RTOS can operate at some primitive level. Next would come API tests to prove that the application code can correctly invoke all kernel services. "Black Box" testing follows completion of API tests. "Black Box" tests use the specifications and user documentation to determine if the RTOS operates as specified. In order to prevent bias, "Black Box" tests require development by an individual or team that has not been involved in the implementation of the RTOS. The code is opaque, a "Black Box", hence the name. This step can be very important as it also proves out the correctness of the user documentation. Completion of this phase of testing is about the earliest one would release an RTOS for incorporation into an application.

The next phase of testing involves "White Box" tests. They derive their name from the openness of the code to the test developer, just the opposite of "Black

Box" tests. The purpose of "White Box" tests is to test all of the minor variations that are missed during "Black Box" tests and ensure proper code coverage. Lastly, there should be stress testing in which the limits of the RTOS are probed. Stress tests try to make the RTOS fail by putting it under heavy loading. The theory is that if one knows where the bridge collapses, one can take steps to ensure that a heavier load is not put on it.

If followed for a single RTOS configuration, the procedure just described can be fairly arduous, time consuming, and certainly expensive. But what if the RTOS is scalable? The magnitude of the testing and verification issue increases geometrically for a scalable RTOS because testing must include not only the unscaled version, but the scaled configurations as well. In a scalable RTOS, especially one where source code is available, it is quite possible to have, literally, millions of possible configurations. In most scalable kernels, it is virtually impossible to test all possible configurations. Consequently, thorough testing of a scalable RTOS requires a very sophisticated testing environment with a high degree of automation.

CHARACTERIZATION

Characterization of an RTOS is simply a term for measuring its performance. A user of the RTOS needs to know what the performance is for a given configuration. This is especially true with a scalable kernel where different configurations can yield different characterizations. With deterministic operation so important to many users of commercial or custom kernels, knowledge of how the kernel performs in a given situation is not only valuable, but also a necessity. The performance figures produced by a characterization suite will tell the user if the RTOS can handle the job required by the application.

Because of scalability, the RTOS characterization suite almost certainly must be automated to about the same extent as the test suite. It should be able to characterize any configuration of the RTOS and provide auditable results to the user or developer. The report of results is very important in that it assures the user not only that the configuration works but also how well it performs. Given that information, the developer can decide to use the kernel as configured or reconfigure it. There are also governmental entities in various countries, the Food and Drug Administration in the USA, for example, that have requirements about demonstrable proof that an operating system has been tested when used in a product falling under its purview. A characterization report coupled with a test and verification log should more than meet those requirements.

TOOL INTEROPERABILITY

Any user of an RTOS today would expect to have a suite of tools that are interoperable. That would include the obvious tools such as compilers, linkers, simulators, and debuggers and emulators. Additionally, the debuggers or emulators should have some kind

of awareness of the RTOS so that the user can determine how the system responds to the application's use of various kernel services. This is a money issue because it bears directly on the cost of usage of the tools. Most developers today do not have unused time in their work schedules that permit hacking with tools that don't work properly. If the tools don't work together, the user is going to spend many extra, unnecessary, hours of engineering time completing the product.

One way to ensure that the tools work together is to acquire the RTOS and the tools from the same source. A number of vendors of RTOS products have reseller's licenses for compilers, debuggers, etc. Conversely, some compiler, debugger, or emulator manufacturers also resell RTOSs. Another way to achieve the same result is to license the RTOS from a vendor that supports, but does not necessarily resell, more than one tool chain. Either way should produce the desired results. The latter probably offers the greater choice of tools.

SUMMARY

This article has been devoted to a brief discussion about four topics that seem to be important to those trying to decide whether to "make or buy" a real-time kernel. Each topic alone is suitable for an entire textbook but, unfortunately, there is only enough space here to bring out the main highlights and give the reader some basic information. Hopefully, the information will be of use in making what is becoming, to more and more developers, a critical decision.

Tom Barrett, founder and president of Embedded System Products, Inc. located in Houston, Texas (USA), designed the real-time operating system RTX. He holds a degree in mathematics from the University of Texas, and has over 30 years' experience in the embedded software industry.

**CONSULT THE
RTOS BUYER'S GUIDE
AT PAGE 79.**