How to Select a Reliable RTOS

By Matt Gordon

Selecting a reliable RTOS should be of the utmost importance to developers

What criteria should embedded systems developers use to select a real-time operating system (RTOS)? Reasonable answers to this question include performance, features, and price, all of which are front and center in the marketing material produced by RTOS vendors. In catalogs, brochures, and countless Web pages, each vendor claims to offer a fast, feature-filled solution at an appealing price.

Seldom mentioned in most RTOS vendors' literature is the issue of reliability. Nonetheless, selecting a reliable RTOS should be of the utmost importance to developers. Whereas an RTOS that is not blazingly fast or that offers a modest set of features may still prove beneficial to many projects, a piece of software that fails regularly has little value in any situation. Such unreliable software can nullify the efforts of an entire development team.

When useful information on reliability is not available, purchasing software becomes a gamble; developers can only hope that their vendor will provide a reliable product. Unfortunately, in today's dynamic RTOS landscape, new, unproven vendors with shaky products abound. Developers who purchase what initially seems to be a well-designed RTOS may discover that they are actually among the software's first users.

Perhaps one of the reasons that reliability, despite its undeniable importance, receives little attention from RTOS vendors is that it is relatively difficult to quantify. While descriptions of an RTOS's speed can be distilled into measurements of context-switch time and interrupt latency, software reliability does not lend itself to numerical expression. In other words, an RTOS provider can't simply state that, "on a scale of 1 to 10, our software has a reliability level of 7.4."

Despite vendors' tendencies to avoid explicitly discussing reliability, information on the topic is not impossible to find. An RTOS's source code can provide valuable insight into the product's reliability, as can the documentation accompanying the code. In place of actually reviewing documentation and source code themselves, many developers choose to seek the advice of colleagues who are familiar with a particular RTOS. An even better course of action for developers who are concerned with reliability is to select a certified RTOS, meaning one that has undergone the rigorous testing required of software in safety-critical fields. Although none of these options are tantamount to a guarantee that an RTOS will perform exactly as expected, they certainly represent an improvement over selecting an RTOS without any knowledge of its reliability.

RTOS Evaluation

An obvious path toward making a judgment on the reliability of an RTOS is to actually use the software. Through trial usage of an RTOS, a developer can begin to formulate an opinion on practically every aspect of the product, including reliability. Any data gathered during such an evaluation can serve as a helpful supplement to the information provided by the software's vendor.

Even though first-hand experience brings such clear benefits, many developers make RTOS purchases without having ever test driven the software that they've chosen. Oftentimes, arranging an evaluation of an RTOS is too much of a hassle. RTOS providers, understandably, are keenly interested in protecting their intellectual property. Thus, they typically limit access to their software. A developer seeking to evaluate a popular RTOS may only be able to obtain a severely limited version of the software.

There are a few notable exceptions to the rule of closely guarded software. Clearly, an open-source RTOS can be downloaded by anyone who is seeking to evaluate it. However, even some commercial software components can easily be evaluated. Micriμm's highly popular real-time kernels, μC/OS-II and μC/OS-III, are two such products.

μC/OS-II, which was introduced to thousands of developers via the book MicroC/OS-II, The Real-Time Kernel, is provided on Micriμm's Web site in source code form. For developers who would prefer to use Micriμm's newer kernel, μC/OS-III, an object code version of this product is likewise provided on the site. Included with the μC/OS-III download are several example projects that are described in μC/OS-III, The Real-Time Kernel, which is the follow-up to the µC/OS-II book. Conveniently, readers of the new book receive both the hardware and the software needed to evaluate µC/OS-III; the book is accompanied by an evaluation board that features a highly capable Cortex-M3-based microcontroller from Micriµm's partner ST Microelectronics.

Developers who have obtained an evaluation version of an RTOS, whether from Micriµm or elsewhere, should begin their investigations into reliability by reviewing the product's documentation. Although documentation does not, of course, directly affect how the code functions, a comprehensive, well-written user's manual often implies a carefully designed RTOS. Developers should be wary of any product that is provided with a hastily prepared manual or that lacks documentation altogether.

As a basis for predicting reliability, an RTOS's source code may be more useful than any documentation. Accordingly, developers who have access to source code should certainly not take this resource for granted. The basic traits that developers should seek to identify when reviewing an RTOS's code are summarized below. An RTOS that does not exhibit these attributes was likely not written with reliability in mind:

- Spacing – There should be ample white space. The code should be indented consistently.
- Naming – Functions and variables should be named in a logical and consistent fashion.
- Comments – The code should be heavily commented. The comments should make the intent of the programmer clear.
- Data types – Portable data types should be defined. Standard C data types (char, short, etc.) are not portable.
- Simple expressions – The code should be devoid of unnecessary complexity. As much as possible, simple and intuitive expressions should be used.

Varying degrees of scrutiny can be applied to both source code and documentation. However, a developer can begin forming hypotheses about an RTOS's reliability after reviewing just a few source files or pages of a user's manual. Confirming these hypotheses is not nearly so easy.

True measures of an RTOS's reliability can only be made by exhaustively testing the software. Many developers' first step toward performing such testing is to run a simple application, one that incorporates only a handful of tasks. However, even an application that just blinks an LED can pose problems. One potential obstacle to running a basic, RTOS-based application is hardware availability. Developers who don't have access to the hardware for which an evaluation package was developed will certainly have trouble running code.

To eliminate hardware as a concern for developers, many vendors provide PC-based evaluation versions of their software. However, there is a separate issue associated with these offerings: significant differences may exist between the evaluation version of an RTOS and the actual product. If ported to an embedded microcontroller, software that previously ran well on a PC might fail. For this reason, when any sort of testing is performed using PC-based evaluation software, the results must be viewed with skepticism.

Even when developers are able to run evaluation software on an appropriate hardware platform, there are plenty of roadblocks to prevent accurate assessments of an RTOS from being reached. Of these barriers, time may be the most difficult to overcome. In the short amount of time normally allotted to developers for evaluating software, gathering any meaningful data is simply impossible.

Finding the time to merely confirm that a simple, RTOS-based application correctly blinks a development board's LEDs might not be an issue for every developer. However, such a test offers little assurance that an RTOS will run well when used in a larger application. Thus, thoroughgoing developers must go beyond blinking LEDs. For such developers, the next step is to integrate the RTOS and a relatively complex application.

The application used to test an RTOS must be chosen carefully. A standard test suite, provided by the RTOS vendor is one poFssible option. Such applications typically exercise multiple different RTOS services and then report the results.

Since an RTOS is unlikely to fail any tests performed by vendor-furnished software, the use of custom test suites is preferable. The ideal test suite would be comprehensive, exercising practically every part of an RTOS. However, writing such a suite would be a monumental undertaking. Thus, some developers opt to avoid formal test packages altogether and instead use application code from a previous project to test an RTOS. Although these developers are freed of the burden of writing test code, they might spend months of their time integrating their application and the RTOS. Such lengthy evaluation periods are highly undesirable, especially for developers planning to try out more than one RTOS.

Considering the amount of time often required for testing, the results can be disappointing. The tests that developers usually run indicate whether or not an RTOS's basic services function correctly, but they do little to measure reliability. An RTOS that successfully completes such tests could still contain hidden bugs. Such bugs can go undetected for years.

History and Reputation

Developers who have used a particular RTOS successfully in one or more of their previous projects can be relatively confident that the product lacks hidden bugs. These developers, unlike those who have simply run a few tests on an RTOS, actually have evidence of reliability. However, reusing an RTOS, rather than selecting a new one, is not an option for every project. A new project may demand features that a previously used RTOS lacks, or the project's budget may not accommodate the licensing costs of the old RTOS.

For projects that necessitate the use of an unfamiliar RTOS, developers can sometimes turn to their colleagues for information on the product's reliability. Consulting with co-workers is a quick means of obtaining data that might otherwise take years to collect. With such data, developers can avoid selecting an RTOS that has a history of poor performance.

Discussions with colleagues can also help developers determine how long a particular RTOS has been in use. Finding an RTOS with a lengthy history is important, because even a bug-ridden product can be improved over time. When the only feasible alternative is a relatively new product, developers should not be quick to discount an RTOS that has a checkered past.

Evidence of an RTOS's history can oftentimes be found in the software's documentation. Any carefully maintained and documented RTOS should be accompanied by release notes indicating how the software has evolved overtime. A typical set of release notes for an RTOS includes a list of all the bugs that have been uncovered in the software. Descriptions of how these bugs were eliminated are also provided in the notes, as are summaries of additional improvements that have been made to the software. A snippet from the actual release notes for Micriμm's μC/OS-II is shown below.

CHANGES IN V2.86

os_core.c	`OS_EventPendMulti()` was added. Optimized `OS_EventTaskRdy()`, and added support for multi-pend. Optimized `OS_EventTaskWait()`. Removed `OS_EventTOAbort()` and added `OS_EventTaskWaitMulti()`, `OS_EventTaskRemove()`, and `OS_EventTaskRemoveMulti()` Optimized `OS_TaskStat()`.
os_mbox.c	Rearranged `OSMboxPend()` to support multi-pend.
os_mutex.c	Rearranged `OSMutexPend()` for consistency.
os_q.c	Rearranged `OSQPend()` to support multi-pend.
os_sem.c	Rearranged `OSSemPend()` to support multi-pend.
os_task.c	Made cosmetoc changes to `OSTaskChangePrio()`, and added support for multi-pend.

Developers who plan on researching the history of an RTOS before selecting the software for use in a new project should also look into the history of the RTOS's vendor. Reliable software is rarely provided by disreputable vendors. Prior to purchasing an RTOS, developers should attempt to confirm that the software's vendor does not regularly discontinue products, and that the company does not have a history of financial or legal problems. The quality of support provided by a vendor should also be of keen interest to developers who may purchase from that company.

Evidence of an RTOS vendor's past successes and failures can be difficult to obtain. The vendors themselves are hardly the ideal source of such information. Not surprisingly, RTOS providers are quick to praise their own records and to slight the accomplishments of competitors. Although industry publications can sometimes be used to track the reputations of RTOS vendors, developers who rely on these sources of information must be careful to distinguish unbiased reporting from opinions and cleverly veiled advertisements.

The best means of learning about an RTOS vendor may be to query that company's customers. Having dealt with a particular vendor, these individuals normally have no problems judging whether or not that company is worthy of additional business. A company that is attentive to its existing customers' needs is likely to treat future customers similarly.

Tracking down customers of a particular vendor can be difficult. Lucky developers may find co-workers who are customers. However, many companies use the same RTOS for all of their projects. The employees of such companies typically don't have experience with a large number of RTOS vendors.

Even developers whose co-workers are customers of a particular RTOS vendor may have trouble obtaining honest advice concerning that vendor and its products. Like the aforementioned trade publications, an RTOS vendor's customers do not always provide unbiased information. A developer whose previous project was ultimately unsuccessful, for example, might be hesitant to recommend the RTOS that was used in that project, even if the software was not the source of any problems.

Certification

Ideally, developers wouldn't need to bother their co-workers for RTOS advice. Reliable software would simply be given a seal of approval from an unbiased third party, and developers could make their choices accordingly. This mark of reliability would be awarded only after a thorough audit of both the software and the environment in which it was written.

Such a clear-cut system for identifying reliable software might seem too good to be true. However, in the avionics and medical fields, granting a seal of approval to carefully designed software is common practice. For those safety-critical industries, in fact, the seal of approval, which is achieved through a process officially known as certification, is required of practically all software. In the United States, software used in airplane parts and other avionics products must be certified by the Federal Aviation Administration (FAA), while software for medical devices requires certification from the Food and Drug Administration (FDA).

Since software used outside of safety-critical fields is typically not subject to the scrutiny that avionics and medical products receive, many developers have little knowledge of the certification process. Developers of all types of software, though, can benefit from the procedures that are used in the safety-critical world. The existence of software certification makes the job of selecting a reliable RTOS much easier. Rather than attempting to assess reliability by performing a litany of tests themselves, developers can simply select an RTOS that has undergone the rigorous testing involved in certification.

The certification process is not standardized across the safety-critical fields. The process that must be followed for FAA certification differs slightly from its FDA counterpart. Even within a particular industry, various types of certification can exist. DO-178B, Software Considerations in Airborne Systems and Equipment Certification, the document that contains the guidelines for FAA-certified software, describes five different levels of certification: A, B, C, D, and E. Which of these levels is required for a particular product depends on the potential results of a failure of that product. Level A certification, which is the most stringent, is required for any product that could bring about the catastrophic failure of an aircraft. Lower levels of certification are reserved for products that could fail without significantly impacting overall aircraft functionality.

Such nuances aside, the overall goal of every type of certification is the same: ensuring that the software under consideration does what it is supposed to do. Presumably, any developer who is evaluating an RTOS also has this objective. Certification, though, involves much more thorough examination than could realistically be performed by a time-strapped developer with an evaluation version of an RTOS.

Before a product can be certified by either the FAA or FDA, its developers (or, in many cases, consultants hired by those developers) must generate reams of documentation. Normally, the first document to be prepared is a master plan that outlines how the software will be proven worthy of certification. In DO-178B, this document is called the "Plan for Software Aspects of Certification."

Amongst the other documents that must be prepared in support of certification are multiple sets of standards. Coding conventions and other rules that were followed during the development of the product being certified are described in these documents. Software that was not developed according to strict coding conventions stands no chance of achieving certification.

Also precluded from certification is software that is not accompanied by exhaustive requirements documentation. The purpose of this documentation is to describe the intended functionality of a particular piece of software. In certified software, every line of code can be traced to a requirement; nothing is superfluous.

Determining whether a set of requirements has been successfully implemented necessitates extensive testing. The team involved in certifying an avionics or medical product may need to write tens of thousands of lines of test code. This code cannot be carelessly prepared; it must thoroughly exercise the software that is to be certified. Generally, test code is written for every function comprising the software.

Test code that is capable of verifying that a piece of software meets all of its requirements can be said to have achieved requirements coverage. In addition to mandating this sort of coverage, the top levels of certification described in DO-178B also call for structural coverage. This extra stipulation helps ensure that test code is exhaustive. There are different degrees of structural coverage, but the key to attaining any of them is to write tests that cause every portion of a program to be executed. A comparison of different types of structural coverage is provided below.

EXAMPLE CODE:

```
if ((spd != 0) || (rpm != 0))
{
    ctr++;
}
```

Type of Coverage	Definition in DO-178B	Test Cases for Example Code*
Statement Coverage	Every statement in the program has been executed at least once.	(spd = 1, rpm = 0)
Decision Coverage	Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken on all possible outcomes at least once.	(spd = 1, rpm = 0), (spd = 0, rpm = 0)
Modified Condition/Decision Coverage	Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.	(spd = 1, rpm = 0), (spd = 0, rpm = 1), (spd = 0, rpm = 0)

If every line of code in a program corresponds to a requirement, then structural coverage can be attained by way of requirements coverage. DO-178B's insistence that both types of coverage must be met for higher levels of certification gives some indication as to how rigorously certified software is actually tested. Although no testing methodology is perfect, the likelihood of a bug surviving the certification process is incredibly small.

By using certification as a criterion for selecting an RTOS, developers can help to keep their own products bug-free. One detail that developers should keep in mind is that an RTOS is rarely at the focus of certification efforts. Instead, certification normally involves application code. When an RTOS is used in a certified application, though, it is subject to all of the testing that the certification process entails.

Only a portion of the many RTOS vendors in business today offer products that have survived the certification process. Micriμm has been providing such software since 2000, when μC/OS-II was first used in an FAA-certified application. Today, μC/OS-II is one of the most popular kernels amongst developers whose products must be certified. In addition to being well suited for avionics and medical products, it is a sound choice for developers in the rail transportation and industrial control fields. It has even been used in applications certified to the FAA's toughest standard, DO-178B, Level A.

μC/OS-II is part of a product lineup that also includes a file system module, graphics software, and a variety of protocol stacks. When developing such products, Micriμm's engineers follow stringent coding standards and are mindful of the requirements of the various safety-critical fields. Accordingly, all of the products are highly reliable.

Such reliable software is appropriate for all sorts of different applications, not just those requiring certification. However, developers in safety-critical fields receive an added benefit from Micriμm's software. By taking advantage of Validation Suites from Micriμm's partner Validated Software (http://www.validatedsoftware.com), these developers can significantly reduce the amount of work required to certify their own products. A Validation Suite is a package that contains the majority of the artifacts required for certification. Validated Software has prepared such packages for several of Micriµm's software modules. For most of the modules, multiple Validation Suites are available, each corresponding to a different safety critical field.

Conclusion

The numerous planning documents, standards, and test results that are necessary for certification offer proof that eliminating bugs from software can be exceedingly difficult. Writing reliable software is a challenge for all developers, including those employed by RTOS vendors. Accordingly, there is risk involved in purchasing an RTOS.

No matter what precautions RTOS buyers take, they cannot eliminate this risk entirely. By choosing a well-tested RTOS from a reputable vendor, though, they can substantially reduce the chances of any problems occurring. The ideal choice is an RTOS that has withstood the certification process, a product that meets the standards of the most demanding applications. For many projects, in the safety-critical world and beyond, such an RTOS can be the difference between success and failure.