CMP
United Business Media

Embedded Systems Design

Using certified operating systems effectively in safety critical embedded designs

How to select and use a certified RTOS in your safety critical application

By Michael Medoff, Exida
03/27/2007

In today's world many potentially dangerous pieces of equipment are controlled by embedded software. This equipment includes cars, trains, airplanes, oil refineries, chemical processing plants, nuclear power plants and medical devices.

As embedded software becomes more pervasive, so, too, do the risks associated with it. As a result, the issue of software safety has become a very hot topic in recent years. The leading international standard in this area is **IEC 61508**: Functional safety of electrical/electronic/programmable electronic safety-related systems.

This standard is generic and not specific to any industry, but has already spun off a number of industry specific derived standards, and can be applied to any industry that does not have its own standard in place. Several industry specific standards such as **EN50128** (Railway), **DO-178B** (Aerospace), **IEC 60880** (Nuclear) and **IEC 601-1-4** (Medical Equipment), are already in place.

Debra Herrmann (Herrmann, 1999) has found a total of 19 standards related to software safety and reliability cut across industrial sectors and technologies. These standards' popularity is on the rise, and more and more embedded products are being developed that conform to them.

Since an increasing number of embedded products also use an embedded real time operating system (**RTOS**), it has become inevitable that products with an RTOS are being designed to conform to such standards. This creates an important question for designers: how is my RTOS going to effect my certification?

This article will attempt to explore the challenges and advantages of using an RTOS in products that will undergo certification.

Making RTOSes Safer

Today, using a commercial RTOS is standard practice in many organizations. The benefits of using a commercial RTOS vs. rolling your own or going without one are many.

For starters, it will help you structure your project into encapsulated software parts (tasks and memory blocks) and save your own development team a lot of time and effort from creating such infrastructure. With today's tight budget and schedules, it is preferable to have your team work on application code for your product rather than infrastructure that supports your application.

With the commercial RTOS, not only has this work already been done for you, and is available to you, but it has presumably undergone tens of thousands of hours of actual field usage making it much more mature and likely more reliable than any new code that would be written.

In addition, using a commercial RTOS typically includes a rich development toolset that will help you to debug and monitor your applications. Having tools that are aware of the operating system and how it works is a tremendous advantage when trying to resolve complex design and debug issues.

In addition to the aforementioned reasons for using a commercial RTOS in any product, there are other advantages that a commercial RTOS can bring to safety applications. These include memory protection, safe communications and secure task scheduling.

These services allow application programmers to readily include safety measures in their application. Operating systems that have these features will save work and reduce risk for their users. If operating systems do not have these features available, then the application developer must build these features into their application.

The importance of protecting memory

Memory protection provides application and operating system data protection against hardware faults and illegal writes. Many operating systems will isolate the address spaces of different processes running on the OS. This protection, combined with a hardware **memory management unit** (**MMU**) will prevent one process from illegally writing over the data of another's.

This is a key feature that supports applications of different safety integrity levels to run on the same processor. Doing so has tremendous advantages when going through a certification.

First off, it allows you to identify non-safety critical tasks that do not need to be developed with the same level of rigor in terms of process and on-line diagnostics. This can greatly simplify the effort involved in the certification.

Secondly, it allows you to use third party software in non safety critical areas without having to certify that software. Having to certify third party software creates extra challenges if you do not have the source code, lifecycle documentation, and/or cooperation from the vendor.

Without this memory protection, you must either use other measures to assure this isolation, or consider all software to be safety critical and therefore subject to the requirements of the highest safety integrity level of your product. You must essentially assume that all software can corrupt the memory space of all other software and therefore must be treated as safety critical.

When using this type of memory protection, there is one concern that should be considered. If you are using a microcontroller that sends data between the core processor and on-chip I/O devices via direct memory access (**DMA**), then these transactions are not protected by the MMU.

If any of the I/O devices are safety critical, then the non safety critical applications either must not access these devices or additional protection measures will have to be added to the code.

Other types of memory protection available include **cyclic redundancy checking** (**CRC**) on static areas of data and code, duplicate storage, monitored heap management, and stack overflow detection.

CRC checking will detect hardware failures as well as illegal data writes. While a MMU may protect illegal data writes from another process, illegal data writes within a process may still occur and should be protected against. For static data that does not change at all or changes very infrequently, a CRC should be used if the data is safety critical.

Another alternative for protection of safety critical data is duplicate storage with comparisons every time the data is accessed. If duplicate storage is used, one copy should be inverted in order to detect specific failures of bits being stuck at one or zero.

Monitored **heap management** will ensure that enough memory is available for dynamic memory allocation and can include some checks for corruption. Stack overflow detection will detect the situation where the stack has run out of space and has started overwriting other memory areas.

Communication of safety critical data is another area where a commercial RTOS can make things easier. This includes both inter-processor communication and intra-process communication. The use of the inter-processor communication is becoming more critical when the system and hardware designers decide to partition the system in multiple processors.

This is a trend that can be seen even in smaller electronic control units such as smart transmitters as the price for a single processor drops below the price of specific ICs. Both of these types of communication can be risky when you have tasks of varying priorities that can interrupt each other.

If you are not extremely careful with the implementation it becomes possible to send inconsistent data that was interrupted before gathering a complete set.

These types of problems are often very difficult to find and fix because they only occur rarely and it is not easy to capture data when they do occur. However, when they do occur, the system response could be an unsafe undetected failure. If the underlying operating system uses queue based messaging between specific tasks with protected sender/receiver information and message body then this problem can be solved at the operating system level.

Similarly, if the OS uses shared memory areas with controlled read/write access with protected ownership information and content, the OS can take care of this issue for you as well. Otherwise, you would have to include similar measures in your application code which could create quite a bit of extra work.

A secure task scheduling and monitoring mechanism that ensures that safety critical tasks run when needed is another advantage that can be offered by a commercial RTOS. This can be done using either deterministic scheduling, logical flow control monitoring as per IEC 61508-2 or a time fence which will terminate the execution of a task if it over-runs its allotted execution or deadline.

These methods, when combined with a windowed external hardware watchdog are highly effective in assuring that critical functions run at the rate that they are required.

IEC 61508 Certification

Currently the IEC 61508 standard does not make any reference to RTOS software or COTS (**Commercial off the shelf software**). This will be changing in the upcoming second edition of the standard which will explicitly require that COTS software shall meet the same requirements as newly developed software.

This is essentially implied by the current standard by not being mentioned, but now it will be specifically called out. Therefore, when certifying your product, you must treat the OS just like any of your components.

Software certification consists of several different phases. First, the development process used to create the software is analyzed. Then the software design is analyzed to determine potential failure modes and measures implemented in the software. Herein lies the major challenge of using a commercial operating system. Since the development process and safety measures of the operating system is out of your control, how can you possibly hope to get this product certified? Fortunately, there are several options here.

The simplest option is to use a certified operating system. There are real time operating systems on the market have been certified to IEC 61508. Choosing one of these operating systems can take a lot of headaches out of the process. The second option is to use a non-certified operating system and include it as a component in your certification process. This option is more difficult, but it can and has been done many times.

Certified Operating Systems

The major advantage of using a certified operating system is the reduction of risk, cost, and time to market. Doing so eliminates the risk that the operating system component is not able to be certified without changes that may be outside of your control.

It gets rid of the cost and time involved in certifying the operating system portion of your design. It eliminates the cost and time involved in creating additional measures in your application code to avoid potential faults in the operating system. And, it rules out the need to gather proven in use data on the operating system.

Another advantage of using a certified RTOS is that it will provide a safety manual, which provides guidance on how to safely use the operating system. This will include information about which features and functions can and can't be used safely as well as any procedures that must be put in place to ensure safety. Also, a certified RTOS will provide some of the features such as memory protection that will make it easier to design safety into your application.

These reasons make it quite attractive to use a certified operating system in your device if at all possible. Of course, there are cases when this is just not practical. A common example is the case where you have an already existing product that you are trying to certify. This product may use a non-certified operating system, and the effort to switch operating systems could be quite large. In addition, doing so could disrupt the reliability of a product that has many hours of field proven runtime to its credit. In this case, switching to a new operating system may add more risk and cost than it saves.

Non-certified Operating Systems

If switching operating systems is not an option, then you must follow the path of implementing measures in your application code to ensure the safety of the operating system. The best way to go about this is to do a software hazard analysis.

This process essentially consists of analyzing all of the components of the operating system to determine what failure modes are possible. For each failure mode found, a measure must exist in the product to ensure that the failure is safe.

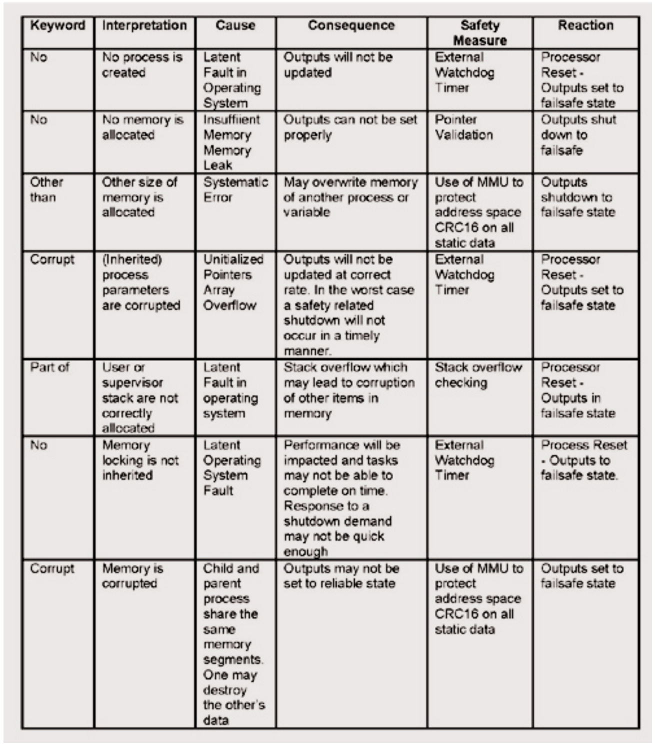
A safe failure is defined as one where the outputs can be placed in the state that shuts down the process which is normally de-energized. A hazard analysis is done by going through attributes of each component one by one and applying guidewords to determine possible deviations.

Possible causes and consequences are then analyzed and possible safety measures are considered. **Figure 1**, **top right**, shows an example hazard analysis for one attribute of an operating system. Note that the purpose of this example is just to give you a feel of how the hazard analysis is done and should not be considered complete or correct.

The result of the hazard analysis will be a list of safety measures such as those shown in **Column 5** of **Figure 1**, **top right**. These safety measures may be items that are already implemented in your application or they may be new measures that you must add to your application.

The disadvantage of having to do this analysis on the operating system is obvious; it could be a lot of work especially if it is discovered that many safety measures must be implemented. However, it is not as bad as it might seem at first.

Many of the safety measures that you would need to implement for the operating system would also be beneficial to your application and would end up being required anyway once you performed the hazard analysis on your own application.



Keyword	Interpretation	Cause	Consequence	Safety Measure	Reaction
No	No process is created	Latent Fault in Operating System	Outputs will not be updated	External Watchdog Timer	Processor Reset - Outputs set to failsafe state
No	No memory is allocated	Insuffiient Memory Memory Leak	Outputs can not be set properly	Pointer Validation	Outputs shut down to failsafe
Other than	Other size of memory is allocated	Systematic Error	May overwrite memory of another process or variable	Use of MMU to protect address space CRC16 on all static data	Outputs shutdown to failsafe state
Corrupt	(Inherited) process parameters are corrupted	Unitialized Pointers Array Overflow	Outputs will not be updated at correct rate. In the worst case a safety related shutdown will not occur in a timely manner.	External Watchdog Timer	Processor Reset - Outputs set to failsafe state
Part of	User or supervisor stack are not correctly allocated	Latent Fault in operating system	Stack overflow which may lead to corruption of other items in memory	Stack overflow checking	Processor Reset - Outputs in failsafe state
No	Memory locking is not inherited	Latent Operating System Fault	Performance will be impacted and tasks may not be able to complete on time. Response to a shutdown demand may not be quick enough	External Watchdog Timer	Process Reset - Outputs to failsafe state.
Corrupt	Memory is corrupted	Child and parent process share the same memory segments. One may destroy the other's data	Outputs may not be set to reliable state	Use of MMU to protect address space CRC16 on all static data	Outputs set to failsafe state

Figure 1. Example Hazard Analysis

In conclusion, there are many well known advantages for using a commercial RTOS in your product and as a result their use is quite widespread today. When using an RTOS in a safety critical application there are some significant advantages and challenges in doing so.

A good operating system will actually have features that make it much easier to implement safety functions and can be a big help in reducing the total amount of work required. However, using a third party operating system introduces an area of risk that may be out of your control.

Fortunately, there are certified operating systems on the market that mitigate most of this risk and there are accepted methods available for including a non-certified operating system if necessary.

Michael Medoff *is a senior safety engineer with* ***Exida.com****. He has over 18 years of software development experience, including work on embedded safety-critical and high-availability applications.*

References

Herrmann, Debra S, Software Safety and Reliability, IEEE Computer Society Press, Los Alamitos, CA, 1999.