

Debugging embedded systems with a real-time operating system

By Colin Walls, Mentor Graphics

This article provides an overview of the issues, scope and limitations of debugging when an RTOS is in use.

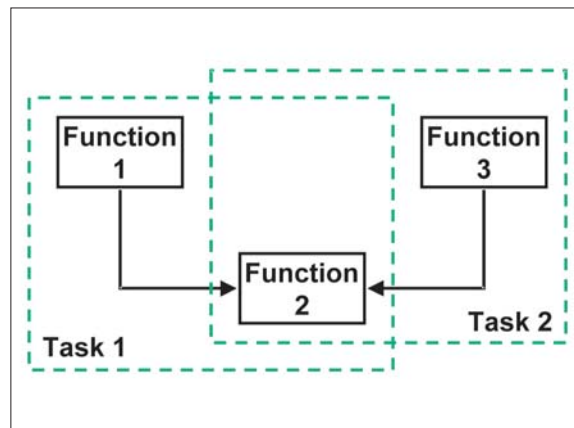


Figure 1. When using an RTOS, the calling functions could be executing in the context of a different task.

■ Most modern embedded system designs make use of a real-time operating system. This permits the software designer to employ the multi-task paradigm to distribute the available processor resources across the required functionality. At the same time, the opportunity arises to distribute the detailed design, programming and debugging effort across a team. An RTOS may be developed in-house or it may be a commercial product, licensed for use in the design at hand. In either case, the multi-tasking environment introduces some new challenges. Not the least of these is debugging. This article aims to provide a complete overview of the issues, scope and limitations of debugging when an RTOS is in use. The term “task” is used throughout. This may variously be translated into “thread”, “process” or “program”, depending upon the specific RTOS in use. The precise semantics are, for the most part, not relevant in this context.

The key function of an RTOS is to allocate time to each task in the system in a rational way. There are a variety of mechanisms, whereby this is achieved. The simplest being a round-robin, where each task simply passes control on to the next. The next possibility is some kind of time-slicing, where the RTOS allocates an equal-sized period of time to each task. Commonly, an RTOS has a much more complex facility, where each task has a priority and is

allocated resources accordingly. The perception of this allocation of processor time is important in the use and design of debug solutions.

Superficially, it would seem logical to keep in mind the mechanism by which the RTOS works: quickly swapping between tasks, so that they give the appearance of executing simultaneously. After all, this is a reflection of what is really happening. However, this does not prove to be very useful. With the possible exception of simplistic round-robin schedulers, it is never really possible to predict which task will run next. So a programmer cannot profit from this model. A debug solution, built with this concept in mind, is likely to be “task-aware” - primarily focused on the current task and the needs to isolate it from the rest of the system. This is rather inflexible.

Even though it is not a true reflection of reality, it proves much more useful to conceive a model where all the tasks are running simultaneously. This is what most programmers practice, as it is the only way to write secure code, with proper management of shared resources. A debug solution, designed with this mind-set, lends itself to multi-task debug - viewing a number of tasks and their interaction - which is much more flexible. Furthermore, the model is scalable upwards to a system that contains multiple processors, which really do execute

concurrently. The conventional, and only really viable, configuration, for debugging an embedded system with an RTOS, is to connect a host computer (PC or UNIX workstation, running the debugger) to the target device. The target may be real hardware or perhaps a simulation, which also runs on the host.

A simulation of the target device, running on the host computer, may be useful in a number of circumstances: before hardware is ready; before the available hardware is reliable; before hardware is available in volume. There are broadly two types of simulation available when debugging with an RTOS:

Instruction set simulation - where the simulator accepts the real code, built for a real target and simulates its execution instruction by instruction. The execution speed is relatively slow, but it is a very precise environment, with great opportunities for non-intrusive “instrumentation” of the code (i.e. monitoring the detailed execution behaviour of the code, without actually changing it). Although slower than real time, the time model may be accurate and has the unique possibility for “stopping real time”. Since the entire functionality of the processor is simulated, perhaps along with some of the surrounding hardware, there is no reason why a complete system - RTOS, drivers and application code - cannot be run.

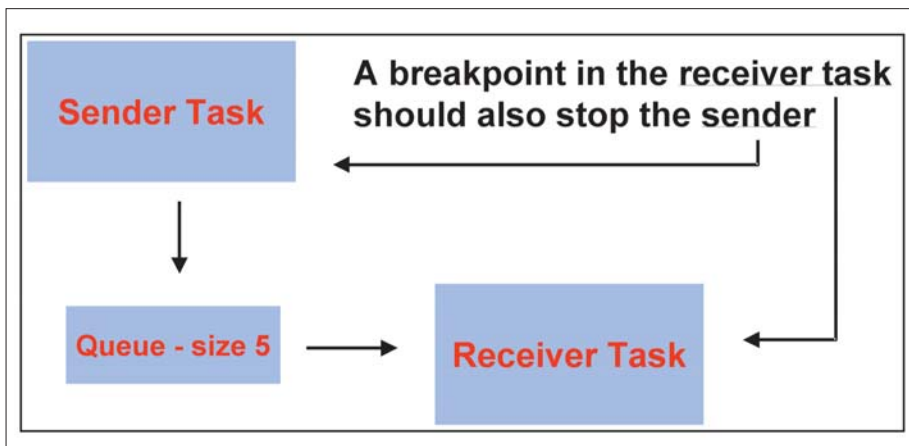


Figure 2. It is very common for task interdependencies to occur in an RTOS-based application.

Native system execution - where the code is built to actually run on the host computer. This results in a very good execution speed - probably similar to real time - but with a different real time profile. A couple of alternative approaches may be taken to facilitating host execution of the target code: there may be a special version of the RTOS that runs as a host process; alternatively, the RTOS calls may be mapped onto calls to the host OS.

At some point, real hardware becomes available and/or limitations of simulation are reached. The hardware may be: prototype hardware, production hardware or evaluation boards (similar architecture to final target). Although apparently better than simulation, using actual hardware is not without problems - hardware glitches and faults and visibility of execution are among them. So, even when hardware is to hand, simulation may remain useful to address certain types of bug. If target hardware is to be used, a suitable connection to the host computer must be established. In broad terms, there are two types of host/target connection that may be employed for debugging: dedicated debug connection - where the link has no additional function, other than debug; communications link - which may be used for other purposes, as well as debug. In some cases, both may be useful.

The use of a host-target connection, which is dedicated to debugging, is becoming quite common. Typically, this is based upon the JTAG standard, which, whilst never intended for this purpose, has been found to be ideal for many such applications. Almost all new processor designs incorporate a JTAG interface for debug. A JTAG connection is quite cheap and easy to incorporate into a design. There are only a small number of signals - it is a synchronous serial protocol - and low-cost connectors are used. Some kind of adapter is required to interface to the host computer. This may be a simple "intelligent cable", costing \$100-\$1000; it may be

an interface box, which includes other functionality [trace, for example], costing \$1000-\$10000. A big advantage of this type of debug connection is that it will even function with a target that is barely working. So long as the processor has power and a valid clock, a JTAG connection may normally be established. Of course, little progress may be made until the memory system is also functional. Typically, a target processor is "frozen" (not executing code) which it is communicating with the host over a JTAG connection.

Historically, the use of a conventional communications link for debugging was very common. Generally this would be a serial line like RS232, RS422 or Ethernet. However, there are many other viable possibilities: USB, PCI Bus, Bluetooth, IRDA, 802.11. A key, but obvious, requirement is a working target. The hardware must be fully functional - the processor must be capable of executing code and the communications interfaces working. The software must, of course, be working to the extent that the communications protocols can be supported.

This leads to an interesting "chicken and egg" issue: if this is the debug interface, how can the drivers and protocol stacks be debugged? The answer is a combination of simulation and the use of a dedicated debug connection, which is ideal for such low level debugging. This type of RTOS-aware debugging is necessarily more complex. This need not be a great concern to the user of a commercial RTOS product, but would probably discourage the user of an in-house kernel. If the communications link is required for the application, sharing it for debugging purposes may be very attractive. Otherwise, the overhead of the additional hardware and software may be disproportionate to the benefits gained. The big advantage of RTOS-aware debugging, over such a link, is flexibility. The target may continue to run, while the debugger communicates with it. This is called run-mode debugging.

There are broadly two modes of RTOS-aware debug: stop mode and run mode. Stop mode is almost always an option. The availability of run mode as well is advantageous.

This mode, which is sometimes also called freeze mode, is when the entire system is stopped - no code execution at all - when a breakpoint is hit or the operator intervenes.

Stop mode is a totally host-resident debug implementation, with no overheads on the target at all. Typically, the target connection is JTAG. Stop mode debug is quite satisfactory for most situations and is only problematic when the bug being sought is closely tied in with the dynamic interaction between tasks, or when halting the system as a whole would cause disadvantageous side-effects (that may hamper locating the bug). This debug mode is ideal for use with an in-house RTOS, as no target support is required and the implementation may be quite straightforward. With simulation, stop mode debug is sufficient for any conceivable multi-tasking debug scenario.

Run mode is a more sophisticated RTOS-aware debugging set-up, as it allows just parts of a system to be stopped; for example: just the current task stops, others continue execution; a defined group of tasks stop, others continue execution; all tasks stop, but interrupts are still serviced. This facilitates some subtle debugging, which may not be possible with stop mode. Run mode needs sophisticated software support on the target - a debug monitor. This is most likely to preclude its use with an in-house RTOS. A memory overhead is also incurred.

Run mode requires a working communications link to the host computer. This may be a further overhead and a debug challenge in itself (to get the communications working).

RTOS awareness may mean different things to different people and is very dependent upon the tools and RTOS in use. The two key areas of functionality are the ability to view data, in an RTOS-aware fashion, and the control of RTOS functions and objects.

All the information concerning the status of a running system is contained in its memory. Meaningful viewing of this data is simply a matter of knowing where it is located and how it is structured and formatted.

For tasks, it is necessary to be able to view a list of tasks in the system, with their running state (typically "current", "ready", "blocked" or "suspended") and priority. For each individual task, it is desirable to be able to view local variables, registers (including the program counter) and stack. Lists and information about other RTOS

objects-e.g. mailboxes, queues, semaphores, mutexes, and messages- are also required.

A task-aware debugger may also exercise some control over the target system. There are various possibilities for controlling tasks:

- a task may be suspended - not available to be scheduled
- a new instance of a task may be spawned (in RTOS that supports dynamic task creation)
- a task's priority may be adjusted
- a task may be paused, while others continue to execute (if run mode debug is available - see above)

Additionally, it is useful to be able to stop the whole system (suspend the scheduler), when using run mode. Breakpoints may also be applied with task-awareness. This is covered below.

In C, it is very common for a function to be called from several other functions. When using an RTOS, the calling functions could be executing in the context of a different task (figure 1). Thus, the called function may be in use more than once "simultaneously". This is not really a problem. The only precaution to be taken is to ensure that the function is re-entrant - i.e.

all its data space is dynamically allocated - no static variables. If a system requires multiple tasks, all of which perform the same operations on different data, there is no need for multiple copies of the code. A single copy will suffice. The task may be instantiated multiple times (resulting in multiple task control block entries), each with its own data. An embedded developer contacted their RTOS/tools supplier with a problem: his design had a fixed amount of code memory, which was all used, and further enhancements were required to the application. The support engineer looked at what was being done. He tried improving compiler optimisation, which was helpful, but nowhere near enough extra memory was freed. Then he took a closer look at the application, made a fairly small change to the structure and reduced the code memory requirements by nearly 90%. The application was some kind of data router, which handled 10 channels of data. Each channel was handled identically by its own task. Originally, each task had its own copy of the code. The modification, that saved so much memory, was to share one copy of the code between all 10 tasks. This introduced a new problem, however. Previously, a breakpoint, placed in the code of one task, would only take effect in the context of that task. The code was

not shared and, hence, only executed in the single task context. Now, the breakpoint would trip regardless of which task utilised the line of code. The solution was a debugger with task-aware breakpoints (more on this below), which the RTOS/tools vendor could readily supply. So the story had a happy ending for all concerned. If any code in an RTOS-based application is shared, then debugging is greatly eased by the availability of task-aware breakpoints. Such a breakpoint is similar to any other, except that it is qualified by the identifier of the task (or maybe tasks) in which it is required to be active.

In fact, a breakpoint can (almost) never be truly "task-aware". The apparent awareness is the result of the action taken by the debugger, when a breakpoint is hit. A decision is made as to whether execution should continue (the current task is not the one in which the breakpoint is required) or stop. This task-context decision always represents a time overhead, usually small, which has various levels of impact, depending upon its implementation. This overhead is called breakpoint latency.

Broadly, the task context decision may be made on the target (by a monitor program running there) or on the host (by the debugger

itself). Target: If the task context is evaluated on the target, the breakpoint latency is reduced. The delay may actually (only) occur at the time a breakpoint is hit or it may impact the task context switch, each time the task is scheduled. This is an implementation detail. In either case, a monitor program is required in the target, which may be quite complex to implement. However, the resulting facility is most flexible. Host: The alternative is for every breakpoint to halt the target and for the task context decision to be made by the debugger. Even if this decision is made very quickly, the breakpoint latency will tend to be higher because of the host-target communications. With this approach, no support is required at all on the target and the coupling between the debugger and the RTOS is much looser. Since the implementation is quite simple, it is an ideal choice for an in-house RTOS.

It is important to differentiate between the scope of a breakpoint and the action that it will take. A breakpoint's scope is the task or tasks in the context of which it will be activated. This is only relevant if code is shared between tasks. Typically, a breakpoint may have the scope of a single task, several instances or all instantiations of the shared code. The action of the breakpoint is what happens when an active breakpoint is hit. Possibilities include: stop the system, which is the only option if stop mode debug is used; stop just the scoped task; stop a number of tasks, normally including the scoped one; stop all the tasks, but continue to service interrupts.

It is very common for task interdependencies to occur in an RTOS-based application (figure 2). A simple case is when one task is generating data, which is sent to another for processing. These tasks are interdependent. The receiver depends upon the sender for data to process; the sender is dependent upon the receiver to accept the data that it has generated, otherwise its buffer will fill up. This second case is very relevant to debugging. If a breakpoint results in (only) the receiver task being stopped, the sender task will rapidly reach an unstable state.

What is required is a means by which a breakpoint on the receiver task will also result in the sender task being stopped. This is accommodated by a synchronised breakpoint. The idea is simple: when a task-aware breakpoint is set in one task, a list of other tasks is provided; these should also be halted, when the breakpoint is hit. The additional tasks may be other instances of the scoped task, or they may be different code entirely, or both.

Synchronised breakpoints can only be implemented when run mode debug is in use, as the remainder of the system continues to execute. It is possible to implement this facility of the

host or on the target. A host implementation is relatively simple, but results in a small delay (latency), while the synchronised task list is processed; this results in each of the additional tasks over-running slightly, which is rarely a problem. A target implementation may be much more complex, but latency is reduced.

The use of a memory management unit (MMU) in some form is common with many modern microprocessors. This may be just to implement some simple inter-task memory protection or for the full implementation of a process model. The advantages and disadvantages from the design and implementation perspective are widely documented, but the debug implications are rarely considered. An MMU may be used in a relatively simple way to afford memory protection between tasks. Each task is only permitted to access specific areas of memory. Thus, each task is protected from interference by others and its own erroneous behaviour, e.g. illegal memory references, stack violations etc. From the debug perspective, this use of an MMU is not a significant problem. The debugger simply needs to deactivate the MMU in order to access all the memory. This has no impact upon integrity of the software itself.

An MMU may be used to remap the memory address space for each task so that it appears to have control of a complete processor. Such a task is commonly referred to as a process. It offers greater flexibility and protection, but complicates debug. The debugger needs access to the remapping information so that it can use the MMU to access the memory space of each task. It is possible to have a "hybrid" memory model, where each process itself contains multiple threads of execution, as in Windows and UNIX. This, in turn, provides the opportunity for a hybrid debug capability: a debugger, which is associated with a single process in the system and provides thread-aware debug of its code.

The deployment of multiple processors in embedded systems is increasingly common. This may be by means of multiple boards in the system, multiple devices on a board, multiple processor cores on a chip or any combination of these. From the debug perspective, architectural details are relatively unimportant. The processors may be all the same type - an array of processors may be a useful solution to certain types of problem. Alternatively, there may be a mixture of architectures, where different processor capacities (8/16/32 bit) or capabilities (e.g. DSP) are useful.

Debugging with multiple processors may be approached in various ways. Multiple debuggers could be employed - one for each processor - but this could be complex, as each is likely to be

a different tool. There may also be connection capacity issues. A single debugger is possible, if the debug architecture of choice accommodates the "true simultaneity" paradigm. This results in a less steep learning curve (only one tool) and opens the possibility for advanced debug functionality, like synchronised breakpoints. Of course, multiple processor systems are just as likely to use an RTOS and, hence, have particular debug needs. This can get more interesting, as there may actually be multiple RTOS products deployed in a single system. This would be a challenge for any debug technology. A challenge, but not an insurmountable one.

It is clear that debugging, when an RTOS is in use, may be a complex business and the development of suitable tools non-trivial. As systems become increasingly complex, the capabilities of the debug tools need to track the needs of embedded software developers.

Understanding RTOS debugging is important for both developers and managers. Even if the functionality is transparent, because it has been addressed by the RTOS/tools vendor, the developer can use this knowledge to appreciate the possibilities and limitations of the technology. A manager can efficiently deploy resources and make informed purchasing decisions. ■