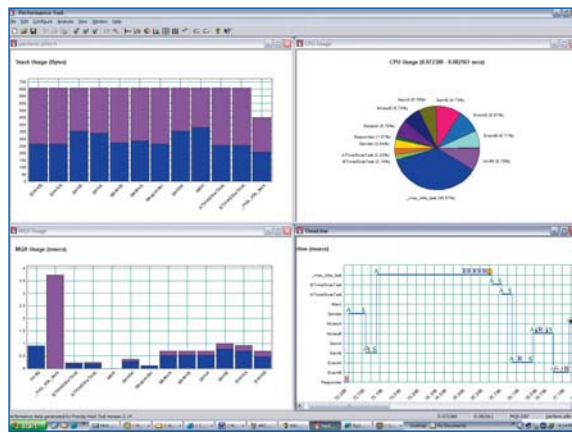


Debugging multi-threaded applications with RTOS-aware tools

by Brian Fellowes, MQX Embedded (a division of ARC International)

MANY MULTI-THREADED DEBUGGING ISSUES ARE ALMOST BEYOND THE CAPABILITIES OF THE STANDARD SOURCE-LEVEL DEBUGGER. ADDITIONAL RTOS-AWARE TOOLS ARE REQUIRED SO THAT DEBUGGING CAN BE DONE AT A HIGHER LEVEL.



RTOS-aware profiling tool

As embedded applications are becoming more and more complex, their use of a real-time operating system (RTOS) is increasing. The use of an RTOS does however create new and unique problems when debugging an application. Many multi-threaded debugging issues are almost beyond the capabilities of the standard source level debugger. Additional RTOS-aware tools are required so that debugging can be done at a higher level. If these are not available, the increased time spent on debugging may almost outweigh the benefits that inclusion of an operating system should provide.

Typically, most source-level cross-debuggers can only display information such as local variables, registers and call stack that are related to the thread state or context for the active thread, that is the task currently running on the CPU. There is often no way to see the state of the other threads at the same time. A function may be called by each of several different threads, so setting a standard breakpoint in that function is not very useful when we want to debug its use in the context of one particular execution thread.

Using only the information available from a standard debugger, to follow the interaction of multiple threads and their synchronisation with semaphores, events and message passing etc would require a detailed knowledge of the inner workings of the RTOS and its data struc-

tures. If source code is not provided for the RTOS then this will prove even more difficult and time consuming. In any case, using an operating system should make the development of application software faster and easier, so it rather defeats this objective if you have to acquire a detailed knowledge of how it works internally in order to use it. What is required is a set of tools that can be integrated with the source-level debugger to interpret status information from the RTOS and clearly display it in additional debugging windows. These should include features such as a summary of threads and their current status, task stack usage, semaphores, mutexes and events, message buffers, queues and memory pools assigned by the operating system.

In addition, we need the ability in the debugger itself to set task or thread-specific breakpoints and to have context switching for viewing the context of different tasks at any given time. For example, in the MetaWare SeeCode debugger from MQX Embedded, task specific breakpoints can be selected by an option in the breakpoint window and execution switched between threads. It is also possible to have multiple source code windows, each one locked to a particular thread. RTOS trace and profiling tools are also very useful to give a high-level view of system activity both for debugging and to optimise performance.

Finally, some means of gathering such RTOS-aware information from the target CPU while it is still running is extremely useful, especially in the later stages of debugging where problems may occur intermittently and infrequently. An example of this is the Embedded Debug Server (EDS) tool supplied with MQX, which allows you to do this via serial, Ethernet or custom connections from the host to your target board.

The most commonly encountered problems with multithreaded debugging are: stack overflow, memory corruption, semaphore deadlock, memory leaks and thread starvation. Most real-time operating systems allocate a separate memory stack to each thread, rather than share one large common stack. This has the benefit of reducing the context switching time (that is the time taken to switch execution from one thread to another thread) and is more robust. However, it can be difficult to estimate a safe size for each stack and overflow can occur when too many local variables are allocated, there are many nested function calls or recursive functions are called. When stack overflow does occur, the resulting memory corruption may manifest itself in many different ways, depending on what use that memory was intended for. The problem may be difficult to diagnose at first as it could occur in the context of another thread, by the corruption of a message buffer and so on. In the worst case a problem

Task	Stack Base	Stack Limit	Stack Used	Overflow?	% Used
Idle	0xe10e8	0xe0f9c	0xe1048	No	48 %
Task1	0xe15dc	0xe11dc	0xe1464	No	36 %
Task0	0xe1800	0xe16d0	0xe16d4	Yes	100 %
Interrupt	0xe089c	0xe049c	0xe0854	No	7 %

Stack usage summary showing overflow of Task 0

may only be noticed under certain rare conditions, which are hard to reproduce as a test case. This might happen for example when the stack overflow goes into another task's stack area that is seldom actually used.

If a stack overflow is suspected, then using the standard debugger, it would be necessary first to find the beginning and end of each task stack in memory, fill each area with a known value and after running the code, check each area to see how much has been overwritten. If there are many threads in the application this will clearly become very tedious. (The terms "thread" and "task" are considered as being identical for the purposes of this discussion). Using an RTOS aware plug-in for the debugger allows display of this information instantly. Ideally, the stack overflow should be flagged as an error in other debug windows too. For example, in the MQX RTOS, the task summary window will also display the stack overflow for the particular task as will the general "check for errors" window. This is a good place to look when you don't have a clue where the problem is!

In some RTOS the task stacks are also used to save the context for interrupts. Although this may save the time it takes to switch to a separate interrupt stack it can increase the possibility of stack overflow, as each thread must have a stack

that is large enough to cope with the worst-case scenario for nested interrupts. Where there is a separate interrupt stack, as in MQX, it is much easier to set and monitor a safe stack size. This also facilitates debugging of nested interrupts. In most cases, this type of error checking is used purely for debugging purposes and will be removed in the release build. However, it is worth noting that all, or parts of it, can be included in the final product to provide run-time error checking and recovery.

Memory corruption of course can occur in any system, for example where a pointer is incremented out of bounds and even the use of a hardware MMU can sometimes only localise such problems. This type of problem can be hard to diagnose and track down using a standard debugger, so again we should hope for some help from the RTOS-aware tools. In the case of MQX for instance, each memory block allocated by the RTOS has a small header containing a checksum. If this header is corrupted an error will be displayed for that block in the Memory Block Window or can be picked up in the Check for Errors Window. Then the debugger can be used to track down the source of the erroneous writes by setting a breakpoint in the debugger for writes to that memory block for example.

In multi-threaded applications memory cor-

ruption may happen when more than one thread can access the same memory and is often caused by a context switch to another thread whilst the first is in the middle of accessing the memory. Usually, this problem is avoided by using a semaphore or mutex to ensure that read or write operations to shared memory by one thread are completed before another can gain access to it. However, first you need to recognise the need to protect specific memory locations. Consider the example where TaskA reads a shared global variable, such as a large structure and TaskB writes to it. If TaskB is in the act of writing to the global variable when a context switch to TaskA occurs, then TaskA reads corrupted data because TaskB had not completed its update of the variable. To help find such problems we need to be able to set a thread-specific breakpoint in one of the threads while it is reading or writing the variable and then use a thread context-switching feature to see what other threads are doing at the same time.

Semaphore deadlock — When we do use semaphores to protect access to resources that are shared between threads, there is always the potential for another type of bug, known as semaphore deadlock. This occurs when a thread is waiting for a semaphore, which will never become free because the thread that owns the semaphore is also blocked. An example is the following. Task0 tries to lock Sem0 and then lock Sem1. Task1 tries to lock Sem1 and then lock Sem0. If, between Task0 locking Sem0 and Task0 locking Sem1, a context switch occurs to Task1, then Task0 will have locked Sem0 but not Sem1. Task1 will run and lock Sem1, and then block when trying to lock Sem0. Task0 will become the active task again and will try to lock Sem1, but Task1 has already locked it. The result is deadlock, in which Task0 is blocked while waiting for Sem1, which is owned by Task1, and Task1 is blocked waiting for Sem0, which is owned by Task0. Neither task can run to free up the semaphore, which the other task is waiting for.

When using a standard debugger it can be very difficult to solve this type of problem as it requires a knowledge of the semaphore data structures within the RTOS and having to step through large amounts of code to find the point at which both threads block and do not return. Again, with RTOS awareness added to the debugger, you can stop execution and bring up a window that shows the status of threads and the semaphores they own, which should enable a quick diagnosis. Memory leaks are one of the most common problems that occur in multi-threaded applications. Passing memory among multiple threads increases the likelihood of forgetting to free a memory buffer or losing a pointer. This means that the memory that has not been freed back to the RTOS will, over time, exhaust some or all of its memory resources and cause a failure in

Task Summary, Processor 1

Task Name	Task ID	TD	State	Priority	Task Error Code
Idle	0x10001	0xe0eb8	Active	11	Ok
Task1	0x10002	0xe1118	Sem Blkd	9	Ok
Task0	0x10003	0xe160c	Sem Blkd	9	Ok

Examine Semaphore: 0xe1ce8, Processor 1

Semaphore ID: 0xe1ce8
 Name: sem.Sem1
 Valid? Yes
 Destroy? No
 Priority Q'ing? No
 Priority Inherit? Yes
 Count: 0
 Max Count: 1
 Strict? Yes

Own/Wait

Task	Task ID	TD
Own	Task0, ID:0x10003, TD:0xe160c	
Wait	Task1, ID:0x10002, TD:0xe1118	

Examine Semaphore: 0xe1d88, Processor 1

Semaphore ID: 0xe1d88
 Name: sem.Sem0
 Valid? Yes
 Destroy? No
 Priority Q'ing? No
 Priority Inherit? Yes
 Count: 0
 Max Count: 1
 Strict? Yes

Own/Wait

Task	Task ID	TD
Own	Task1, ID:0x10002, TD:0xe1118	
Wait	Task0, ID:0x10003, TD:0xe160c	

Screenshot from SeeCode debugger showing task and semaphore status view

Click-for-More

Interested in more information about Debugging multi-threaded applications in an embedded environment?

Visit our specific website with links to:

- Details on the MQX RTOS and MetaWare embedded development tools
- Technical information about the MQX Embedded RTOS and stack aware debugging tools
- Downloadable evaluation kits where you can experience this technology
- Information about MQX Embedded and its partners

Simply type-in Reader Service #: **605** at Embedded-Control-Europe.com/know-how



the application. The following is a simple example: TaskA allocates a memory block and sends a pointer to the block to TaskB, assuming that it will free the memory block when it is finished with it. TaskB receives the pointer to the memory block but does not free it. If this process is repeated, the RTOS will eventually run out of memory to allocate.

It is often difficult to determine the source of this type of problem without following the code line by line, in an attempt to match the memory allocation code with the corresponding code to free the memory. What is needed to accelerate this debugging process is an RTOS memory pool display window in your debugger that shows which threads own which memory blocks. If a large number of memory blocks are all being used by one thread and none of them are being freed, then it is likely that this thread will be the cause of the problem. This occurs when a thread is prevented from running for sufficient time to carry out the work it needs to do. This is usually because threads that were set at a higher priority level are taking up too much

of the total CPU time and may be because of a flawed selection of thread priorities or inefficiency in the synchronisation scheme used. In some cases it may not be obvious that this is a scheduling problem rather than an error in the coding of the thread's functionality. An RTOS-aware profiling tool that can provide a trace of thread activity and a display of the CPU usage by each thread will make it much easier to detect and rectify this type of error. It is also valuable for optimising the system performance.

The debugging of multithreaded applications is likely to be difficult and time consuming if only a standard source-level debugger is available, and will require a detailed knowledge of the RTOS that is used. A good set of RTOS-aware tools that are integrated with the debugger enables many types of error to be diagnosed quickly at a high level. They can also help to refine the system level design and optimise the performance. Therefore, the availability and capabilities of such tools should be an important factor in the selection of an RTOS and development environment. ■