Priority Inversion
Use Case Example

INTRODUCTION

Priority Inversion can occur within embedded systems when using an RTOS configured for priority based, preemptive scheduling. Priority Inversion is a term used to describe a situation when a higher priority task cannot execute because it is waiting for a low priority task to complete. This typically occurs when attempting to gain control of a semaphore or other resource. Priority Inversion can be a significant problem, and can lead to serious consequences. However, in small embedded systems it can often be avoided at system design time by considering how resources are accessed and designing the system to avoid contention.

This article describes how Priority Inversion can be detected using a profiling tool, how its effects can be minimized, and how to design out Priority Inversion in the first place.

We can see that the SamplerTask is running, but it does not clear the watchdog timer in the last execution of the Task, and therefore allows a watchdog reset to occur. So why didn't SamplerTask reset the watchdog timer? Let's enable Kernel Service calls in figure 2 to see what the task was doing.

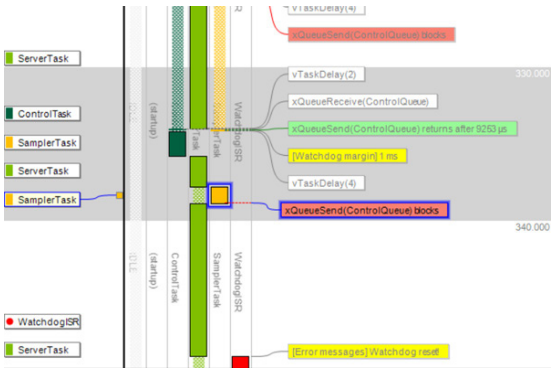

Figure 2. *An Example of Kernal Service calls*

DETECTING PRIORITY INVERSION USING TRACEALYZER

Tracealyzer is a profiling tool that provides engineers with visibility into the sequence of events occurring within their software designs. This example from Percepio illustrates how to detect a Priority Inversion with Tracealyzer, and is about an engineer who had an issue with a randomly occurring reset. By placing a breakpoint in the reset exception handler, they figured out that it was the watchdog timer that had expired. The watchdog timer was supposed to be reset in a high priority task that executed periodically.

The ability to insert custom Tracealyzer User Events allowed the engineer to gain greater visibility. 'User Events' are similar to a classic "printf()" call and in this example were added when the watchdog timer was reset, and when it expired. User Events also support data arguments, used to log the timer value (just before resetting it) to see the watchdog "margin", i.e., remaining time. The result can be seen below, in the yellow text labels of figure 1.

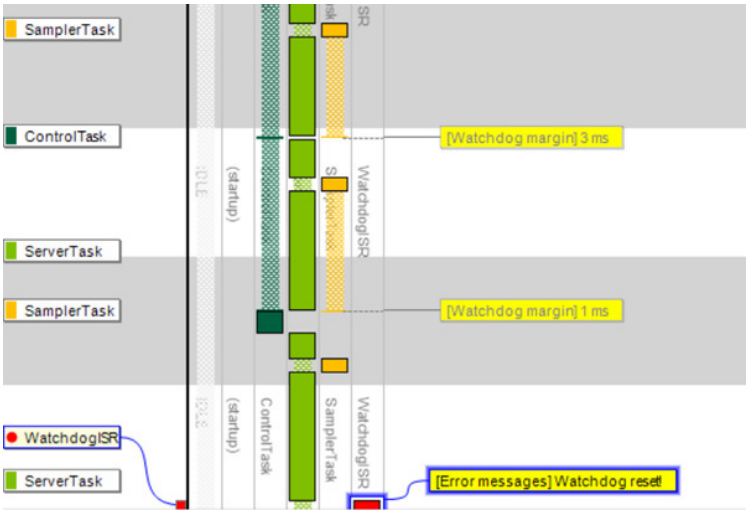

Figure 1. *An Example of User Events*

The last event of SamplerTask is a call to xQueueSend, an OPEN**RTOS** function that puts a message in a message queue. Note that the label is red, meaning that the xQueueSend call blocked the task, which caused a contextswitch to ServerTask before the watchdog timer had been reset, which caused it to expire and reset the system.

highintegritysystems
email: sales@highintegritysystems.com
web: www.highintegritysystems.com

WITTENSTEIN

FIXING PRIORITY INVERSION BY TURNING TASK PRIORITY LEVELS

A solution could be to change the scheduling priorities, so that ControlTask gets higher priority than ServerTask. Figure 5 shows the result of switching the task scheduling priorities between ServerTask and ControlTask. The system now shows a much more stable behavior. The CPU load of SamplerTask (here red) is quite steady around 20%, indicating a stable periodic behavior, and the watchdog margin is a perfect "line", always at 10 ms. It does not expire anymore – problem solved! (Note that the task colours have changed due to the change in relative priority levels.)

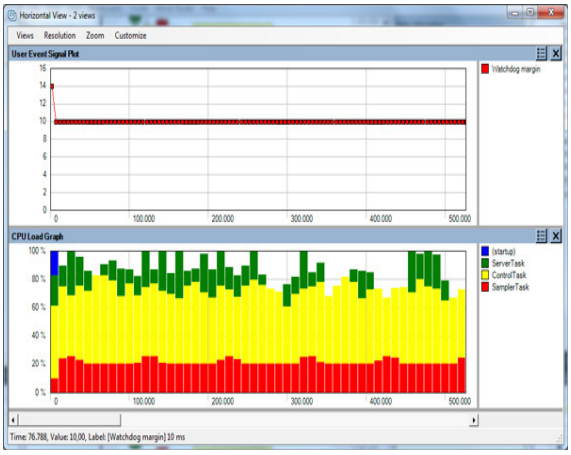

Figure 5. *Switching between ServerTask & ControlTask*

So why was xQueueSend blocking the task? By doubleclicking on this Event Label, we open the Object History View, showing all operations in this particular queue, "ControlQueue", as illustrated in figure 3. The far right column shows a visualization of the buffered messages. We can see that the message queue already contains five messages and is probably full, hence the blocking. But the ControlTask is supposed to read the queue and make room, why hasn't this worked as expected?

Timestamp	Actor	Event	Block time	Status	Size	Queue
142.255	ControlTask	xQueueReceive		Received post #27	1	27 28
143.805	SamplerTask	xQueueSend		Sent post #29	2	28 29
144.684	ControlTask	xQueueReceive		Received post #28	1	28 29
147.391	ControlTask	xQueueReceive		Received post #29	0	29
148.805	SamplerTask	xQueueSend		Sent post #30	1	30
149.913	ControlTask	xQueueReceive		Received post #30	0	30
152.141	ControlTask	xQueueReceive	1.772	Trying to receive...	0	Empty
153.805	SamplerTask	xQueueSend		Sent post #31	1	31
153.913	ControlTask	xQueueReceive		Received post #31	0	31
156.295	ControlTask	xQueueReceive	19.307	Trying to receive...	0	Empty
158.815	SamplerTask	xQueueSend		Sent post #32	1	32
163.805	SamplerTask	xQueueSend		Sent post #33	2	32 33
168.805	SamplerTask	xQueueSend		Sent post #34	3	32 33 34
173.805	SamplerTask	xQueueSend		Sent post #35	4	32 33 34 35
175.602	ControlTask	xQueueReceive		Received post #32	3	32 33 34 35
177.664	ControlTask	xQueueReceive		Received post #33	2	33 34 35
178.805	SamplerTask	xQueueSend		Sent post #36	3	34 35 36
183.805	SamplerTask	xQueueSend		Sent post #37	4	34 35 36 37
188.805	SamplerTask	xQueueSend		Sent post #38	5	34 35 36 37 38
193.812	SamplerTask	xQueueSend		Trying to send...	5	34 35 36 37 38

Figure 3. *Object History View*

To investigate this, it would be interesting to see how the watchdog margin varies over time. We have this information in User Event Logging, and by using the User Event Signal Plot, we can plot the watchdog margin over time. By adding a CPU Load Graph on the same timeline, we can see how the task execution affects the watchdog margin, as shown in figure 4.

Figure 4. *User Event Signal Plot*

In the CPU Load Graph, we can see that the ServerTask is executing a lot in the second half of the trace, and this seems to impact the watchdog margin. ServerTask (bright green) has higher priority than ControlTask (dark green), so when it is executing a lot in the end of the trace, we see that ControlTask is getting less CPU time. Most likely, this could cause the full message queue, since ControlTask might not be able to read messages fast enough when the higher priority ServerTask is using most of the CPU time. This is an example of a Priority Inversion problem, as the SamplerTask is blocked by an unrelated task of lower priority.

FIXING PRIORITY INVERSION BY USING A GATEKEEPER

To avoid Priority Inversion when accessing system resources we recommend using a 'Gatekeeper' Task. A basic Gatekeeper Task, figure 6, consisting of a Task that controls the 'resource', a queue for receiving data/ command, and a call back function.

Application Tasks write data/commands to the queue Instead of directly accessing the resource. The Gatekeeper Task processes the data/commands and the resource is updated accordingly. When the resource changes state an ISR is triggered (say it's a new network message) which is placed in the Queue, the Gatekeeper Task will then execute the registered callback function to pass back the new data to the Application Task.

This method prevents Priority Inversion when accessing system resources and is our recommended solution.

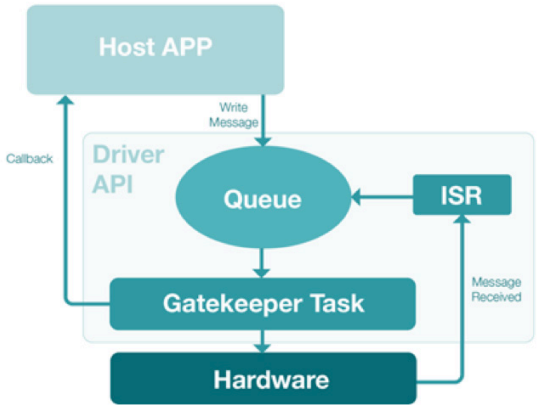

Figure 6. *A Gatekeeper Task*

FIXING PRIORITY INVERSION BY USING CRITICAL SECTIONS

A basic but effective way of avoiding Priority Inversion is to enter a critical section. This is where the RTOS scheduling algorithm is suspended during the time a Priority Inversion could occur. Although an appropriate solution in some systems, it may affect the responsiveness of the system. It's an option open to designers, but we would recommend instead the use of a Gatekeeper Task in most situations.

FIXING PRIORITY INVERSION BY MUTEXES

Priority Inversion can occur when multiple tasks are accessing a single resource such as a network driver or graphics display. In this case it may be appropriate to use a Mutex to minimize the effect of a Priority Inversion.

Mutexes are binary semaphores that include a priority inheritance mechanism. Whereas binary semaphores are the better choice for implementing synchronization (between tasks or between tasks and an interrupt), Mutexes are the better choice for implementing simple mutual exclusion (hence 'MUT'ual 'EX'clusion).

When used for mutual exclusion the Mutex acts like a token that is used to guard a resource. When a task wishes to access the resource it must first obtain ('take') the token. When it has finished with the resource it must 'give' the token back – allowing other tasks the opportunity to access the same resource.

Mutexes also permit a block time to be specified. The block time indicates the maximum number of 'ticks' that a task should enter the Blocked state when attempting to 'take' a Mutex if the Mutex is not immediately available. However, unlike binary semaphores, Mutexes employ priority inheritance. This means that if a high priority task blocks while attempting to obtain a Mutex (token) that is currently held by a lower priority task, then the priority of the task holding the token is temporarily raised to that of the blocking task. This mechanism is designed to ensure the higher priority task is kept in the blocked state for the shortest time possible, and in so doing minimise the 'Priority Inversion' that has already occurred.

Priority inheritance does not cure Priority Inversion. It just minimises its effect in some situations. Hard real time applications should be designed such that Priority Inversion does not happen in the first place.

WITTENSTEIN high **integrity** systems

Worldwide Sales and Support
Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@high**integrity**systems.com
Web: www.high**integrity**systems.com

WITTENSTEIN