HW-RTOS

Real Time OS in Hardware

Issues with Real Time Performance in Conventional RTOS and Performance Improvements through HW-RTOS

Many embedded systems use an RTOS to implement a multi-task environment. However, the adoption of conventional software RTOS often results in insufficient real time performance. This is due to the high overhead and long interrupt disabled periods of the RTOS. In this white paper, we identify the mechanisms causing overhead and interrupt disabled periods and verify them with direct measurements. Then, using the same tests on HW-RTOS, we show how it delivers extremely beneficial results.

R70WP0002EJ0100 September 2018

1. Executive Summary

When real time embedded systems include a real time operating system (RTOS), the influence of RTOS overhead can become a serious issue. This isn't usually a problem in systems tolerant of overhead over one millisecond, but when overhead of several microseconds is a problem, often users must resort to tuning or even give up on RTOS altogether. This white paper identifies the causes of real time performance degradation due to conventional RTOS, and verifies them with measurement data. Our results show that the impact on real time performance is much greater than most software engineers think, and found that it is incredibly difficult to guarantee worst-case values. On the other hand, carrying out the same measurements on HW-RTOS shows that the impact on real time performance is minimal. We explain that since HW-RTOS allows for worst-case values to be defined in the design stage, it reduces some of the design-related burdens for software developers, thus making real time system development easier, and at the same time allowing guaranteed performance.

2. Primary Deciding Factors in RTOS Performance

2.1. Pros and Cons of RTOS

An RTOS is often used in embedded systems for several reasons. Among them are the facts that it's easier to create a multi-task environment using RTOS, and using semaphores and event flags makes it's simple to implement inter-task synchronization and communication. This results in easier software modularization and reusability, thus improving software development productivity as well as improved system reliability.

However, adopting RTOS often results in failure to achieve the target performance of the system itself. One issue is that as soon as the RTOS is installed, systems often seem to slow down. As a product the functions are fine, but response is slow, or movements seem sluggish. Another issue is that systems become unable to satisfy required time conditions. To put it simply, the system is unable to satisfy the need for real time performance.

After investigating the causes of these issues, we settled on two. The cause of the former is RTOS overhead, and the latter is the interrupt disabled period generated by RTOS. More simply, these two issues are the primary factors deciding RTOS performance. The following is a discussion of these two issues in a bit more detail.

2.2. Overhead from RTOS

RTOS is just one part of the software executed on the CPU. In other words, RTOS and application software share the CPU. As a result, it seems like the more the RTOS occupies the CPU, the more slower the application performance. The amount of load the RTOS puts on the system as a whole depends on the type of application. For example, network protocol control requires frequent use of RTOS functions. That's because implementing multitask network protocol control requires frequent use of the RTOS inter-task communication and synchronization functions. As a result, the RTOS occupancy rate of the CPU during network protocol control is high. Engineers have likely felt that network throughput seems to stay low when using a low-end processor. This is due to RTOS overhead.

So you can see, then, that it's always best to keep RTOS overhead as low as possible. In section "3. Conventional Software RTOS Performance," we describe measurement results to show just how much RTOS overhead there is.

2.3. RTOS-generated Interrupt Disable Period

First, let's discuss just what "real time" means. When “real time” is used to describe a system, it indicates that "the system guarantees a response to a given event within a specified time constraint." The "specified time constraint" can differ depending on the application. For example, a response time of 100ms is fine for a TV remote, but a lag of 100ms between hitting an electric organ key and the sound being emitted is considered too slow. And of course, servo motor control requires extremely fast response speed. For high speed servo control, response speeds below 1 microsecond are often required.

As the name states, RTOS is an OS to be used in real time systems. However, the truth is use of RTOS is not beneficial for systems requiring very fast response times. Typically, the response speed at which systems with RTOS installed can operate without issue is around 100 microseconds.

The reason for this is the RTOS interrupt disabled period. RTOS processing requires consistency. As a result, almost all of the periods when the RTOS is running are interrupt disabled periods. So installing an RTOS leads to a degradation in response speeds. If these interrupt disabled periods were fixed, it would be possible to construct a system that anticipates and compensates for them. However, the length of interrupt disabled periods can vary depending on the RTOS internal state. In addition, it's incredibly difficult to guarantee worst-case values for interrupt disabled periods. In section "3. Conventional Software RTOS Performance," we will show measured values for interrupt disabled periods, and explain why it's so difficult to guarantee worst-case values by examining the RTOS structure.

3. Conventional Software RTOS Performance

Overhead from RTOS corresponds to the RTOS run time. RTOS run time refers here to the time when the RTOS is running between when a system call executes or an interrupt occurs and the interrupt

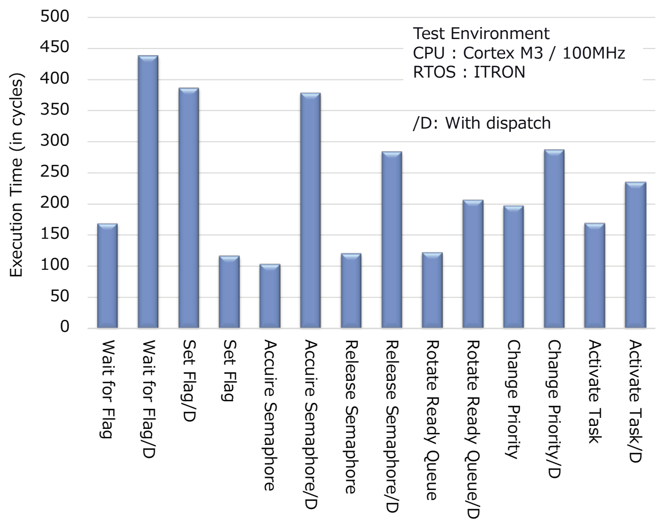

Figure 3-1 API Execution time

handler starts up, or the time used for Tick handling, These all result in overhead from RTOS.

And at the same time, most of the periods when the RTOS is running result in RTOS interrupt disabled periods. However, since interrupts are allowed here and there, many RTOS incorporate ideas to help shorten interrupt disabled periods. Still, the rather long Tick handling process and internal queue control periods must be interrupt disabled.

In the next section section we discuss in detail the results of measuring API execution time, as well as the overhead and interrupt disabled periods during Tick handling and queue processing.

3.1. API Execution time

Figure 3-1 shows measured API execution times for ITRON, the most commonly used RTOS in Japan . The APIs that include “/D” are API processes that are executed with a dispatch. The measurements for Figure 3-1 were made under static conditions. However, since RTOS execution times depend on the current internal state, they fluctuate dynamically. Figure 3-2 shows an example of this. The figure displays all Set Flag API execution times. However, the more tasks that are waiting for the same event,

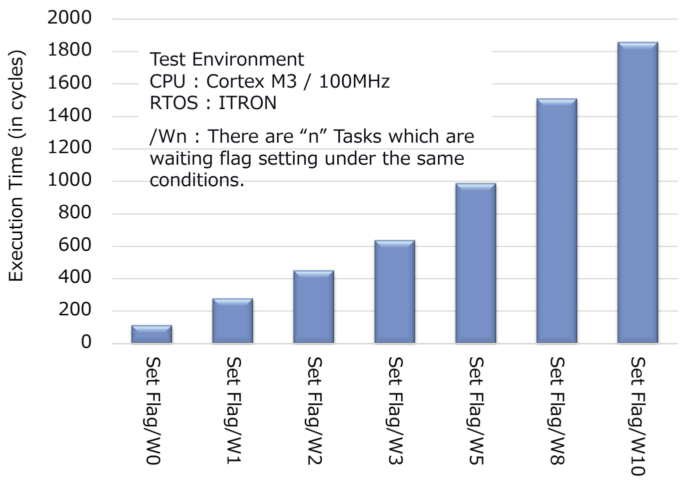

Figure 3-2 API Execution time

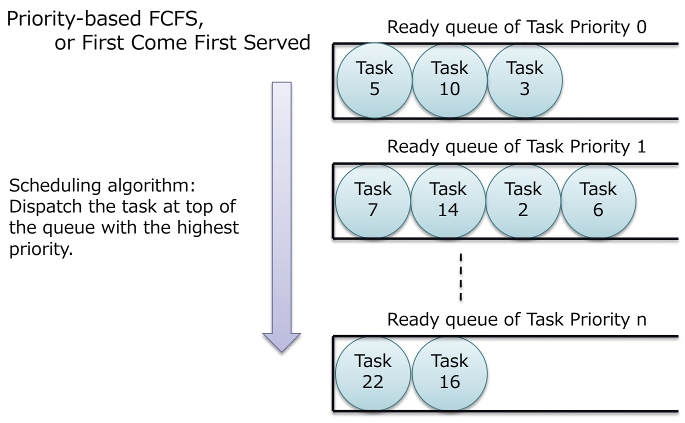
Priority-based FCFS, or First Come First Served
Scheduling algorithm: Dispatch the task at top of the queue with the highest priority.
Ready queue of Task Priority 0
Task 5
Task 10
Task 3
Ready queue of Task Priority 1
Task 7
Task 14
Task 2
Task 6
Ready queue of Task Priority n
Task 22
Task 16

Figure 3-3 Wait Queue

the longer the execution time. In other words, execution time fluctuates according to the internal state of RTOS., So what is the reason for this?

Figure 3-3 shows the Wait queue used in the RTOS, running on a priority level based FCFS (First Come, First Served) algorithm. Specifically, every object has a queue for each task priority level (in this example, priority level 0 is designated the highest priority level, and priority level n is the lowest). For example, to implement Semaphore, there must be n queues for each semaphore identifier. Each task that wishes to acquire a semaphore must wait in a queue. When a task releases its semaphore, the task at the head of the queue with the highest priority level is selected, and the semaphore is acquired by that task.

Additionally, the following key processes are also required for the Set Flag API.,p For flag control, the system must compare the waiting flag pattern and the flag pattern in the event table after it's set to determine if the conditions for release waiting have been met. In other words, the RTOS must go through each task, in order, from the head of the queues in Figure 3-3 and compare flag patterns. As a result, the processing time for Set Flag system calls increases dramatically in proportion to the number of tasks awaiting the same flag, as seen in Figure 3-2. So the RTOS overhead occurring with API execution changes dynamically in response to the internal state of the RTOS at the time of execution.

Typically, most designers expect results from system call execution in several hundred clock cycles. However, right now many don't realize that the overlap of certain conditions can make it take much more time than that. And so, when conditions like those discussed above pile up, overhead can suddenly spike, meaning certain processes don't get completed within their specified time constraints, resulting in overall failure as a real time system.

3.2. Influence of Queue Handling

As explained in Figure 3-3 with the Wait queue's logical structure, an RTOS needs priority level queues for every object. Here, we'll explain how queues are implemented by focusing on one particular object.

Usually, queues implemented in software use a list structure. Within the RTOS, the TCBs are connected to each other with a list architecture to implement a Wait queue (TCB: Task Control Block, a structure used to aggregate information for every task. One exists for each task identifier.). The queue

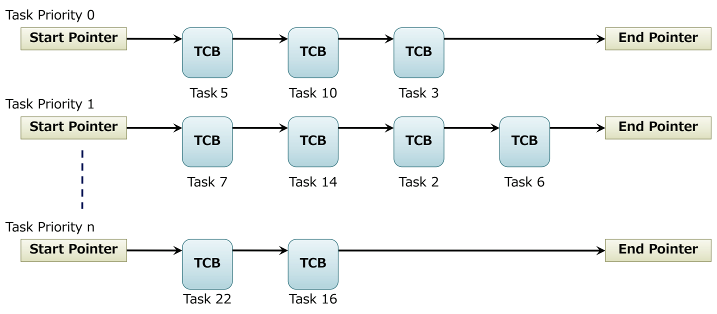

Figure 3-4 Architecture of Wait Queue (1)

from Figure 3-3 is implemented as show in Figure 3-4.

When a task is taken from the Wait queue, the task at the head of the queue with the highest priority level is taken first. In this figure, that's Task 5. In the figure, a task is appended to the queue at "priority level 0," but sometimes no task is in the queue. Thus, the software must check if tasks are appended to the queue starting from the highest priority level down, in order, to find the highest priority level queue where tasks are waiting. As a result, the processing time required can vary greatly depending on queue conditions. Another problem occurs with the pointers. There is an enormous number of queues within the RTOS. For example, if there are 64 semaphore identifiers, 64 event identifiers, and 16 priority levels, then the total number will be

$$64 \times 2 \times 16 = 2{,}048$$

queues. As a result, even with this relatively low number of objects, an enormous amount of resources. (i.e. queues) is needed to handle pointers alone.

One possible improvement to this is shown in Figure 3-5. The tail of each queue is connected to the TCB at the head of the next queue that follows in descending priority (i.e. next lower queue). This structure allows for very easily removing a task from

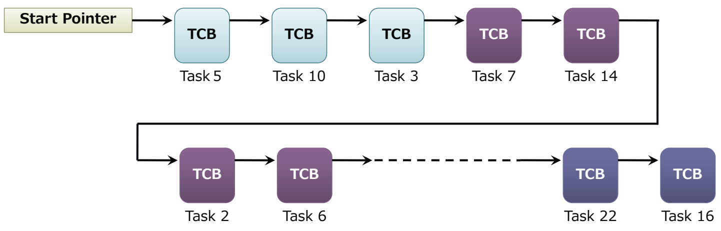

Figure 3-5 Architecture of Wait Queue (2)

the Wait queue by simply selecting whichever task the pointer is indicating. It also greatly reduces the number of pointers. However, upon more careful consideration, this improvement idea results in substantial processing when appending tasks to the queue. For example, when appending a task with priority level 7 to the queue, the search begins from the head, then when the search reaches the priority level 7 tasks, it appends the new task to the end of the last level 7 task.

Either way, it's clear that queue handling requires a lot of processing time, and that time will vary depending on the conditions of the queues. In addition, queue handling is such a critical process that interrupts are disabled during each process. In the following, we will explore just how much time queue handling takes, and evaluate how much impact that has on real time performance.

The RTOS queue structure being evaluated is the form shown in Figure 3-5. Thus we can predict that the processing time required when appending a new task to the queue will vary greatly depending on how many tasks are already in the queue. We can also predict that the number of tasks appended to the queue at each priority level will affect the processing time. So we ran measurements in three patterns, as shown in Figure 3-6. In pattern 1, one task each of priority levels 2-14 was already appended. In pattern 2 there were two of each, and in pattern 3, four of each. For each pattern, we changed the priority level of the task to be connected and measured the

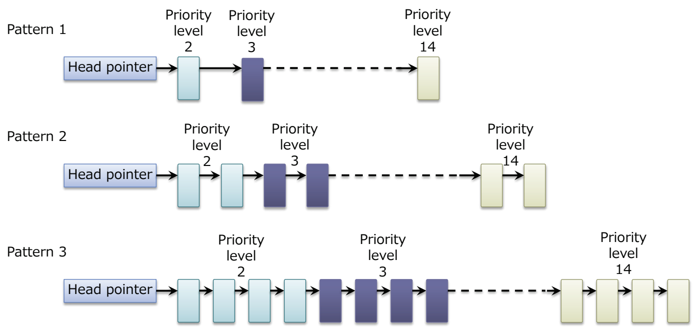

Figure 3-6 Measurement Patterns of Queue Processing Time

difference in processing time. More specifically, we designated a semaphore identifier, and by repeatedly invoking the Acquire Semaphore API, generated the patterns in Figure 3-6 in advance. Then at the end, we invoked the Acquire Semaphore API one final time and measured the API execution time.

The results are shown in Figure 3-7. As the chart shows, processing time for queues with large numbers of tasks already appended is high, and it also takes more time to append tasks with low priority levels.

Figure 3-8 deals with pattern 3, and shows the results of measuring the length of interrupt disabled periods during processing. This figure clearly shows that API execution periods are almost all interrupt disabled.

The above all helps clarify the following points: API execution time accompanying queue handling fluctuates dramatically based on the state of the queues in RTOS. Also, the interrupt disabled periods resulting from that are almost as long as the APIexecution time, so interrupt disabled periods can also fluctuate dynamically depending on the RTOS queue internal states.

So when a large number of tasks are appended to queues, queue handling can create unexpected overhead and unexpectedly long interrupt disabled periods, possibly resulting in unexpected faults in real time systems. In all likelihood, application and software designers don't often consider the number of tasks connected to queues inside the RTOS, but it really is important to be aware in advance of possible issues they could cause.

3.3. Influence of Tick Handling

Next we'll discuss the impact of Tick handling. Tick handling is a process connected to RTOS time management. More specifically,

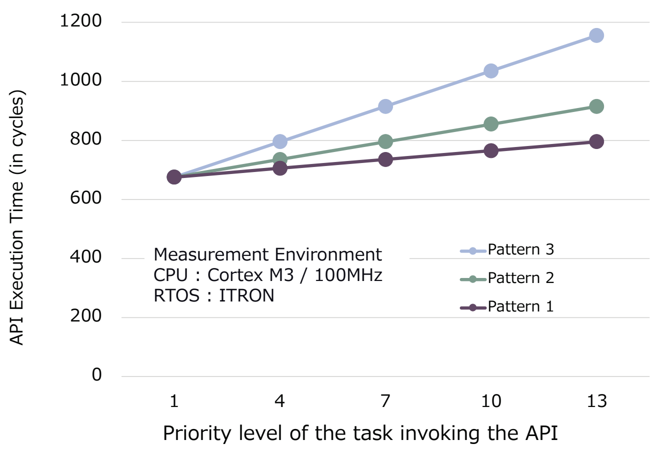

Figure 3-7 Execution Time of Acquire Semaphore API

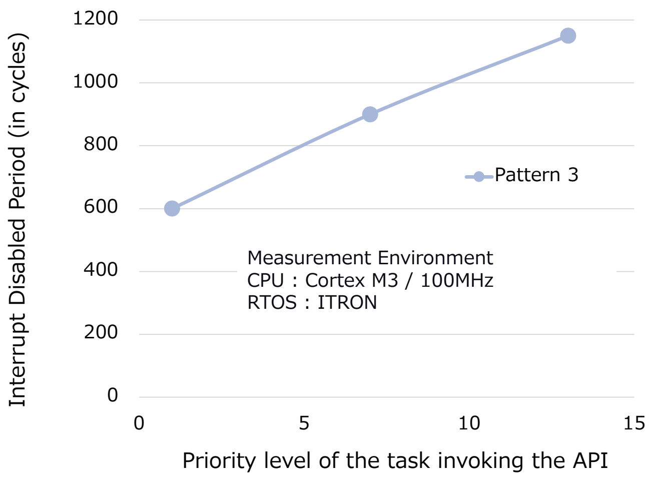

Figure 3-8 Interrupt Disabled Period by Queue Processing

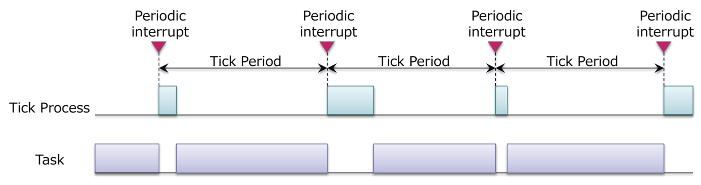

Figure 3-9 Tick Process

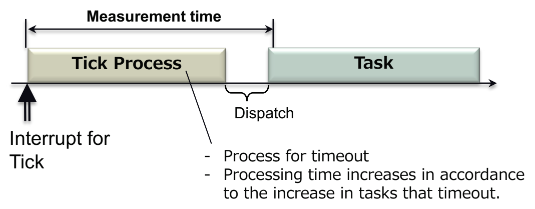
Measurement time
Tick Process
Task
Dispatch
Interrupt for Tick
- Process for timeout
- Processing time increases in accordance to the increase in tasks that timeout.

Figure 3-10 Measurement of Tick

- RTOS check s if tasks in Wait status have timed out, and if they have then it moves the task from Wait to Ready.
- RTOS checks if the Cyclic Handler activation time has arrived, and if it has, then it activates the Cyclic Handler.

are two of the processes it controls. As shown in Figure 3-9, Tick handling runs the processes above through activating periodic interrupts. The interval of those periodic interrupts is called the Tick period, and that period is the basic time unit of Tick handling. As a result, the time out period or cyclic handler interval are all integer multiples of the Tick period.

This all means that the Tick process is an essential function of RTOS, but below we will see that it has its problems, as well.

1. As Figure 3-9 shows, applications are periodically interrupted, resulting in lower CPU use efficiency.
2. As Tick processing is a critical process, interrupts must be disabled during processing. As a result, interrupt responsiveness deteriorates.
3. Since the Tick interval is the basic unit of time outs and the cyclic handler, reducing the length of the Tick interval results in more precise time management. However, shortening the Tick interval increases the amount of CPU time used by the Tick process, reducing CPU efficiency for applications.

Thus, in terms of RTOS performance, Tick handling is a cause of overhead as well as having a major influence on interrupt disabled periods. Below, we'll show the results of measuring the overhead time accompanying Tick handling, as well as the interrupt disabled periods it causes.

As discussed before, the Tick process checks if tasks in Wait status have timed out or not. Thus, it can be assumed that processing needs increase when a large number of tasks time out simultaneously. So we adjusted the timing of task time outs, and designed test software so that n tasks would time out simultaneously within the same Tick process to measure the time involved in Tick handling. Specifically, we measured the time from when the the Tick activation interrupt is generated until the next task is dispatched after the Tick completes, as shown in Figure 3-10. For this example, we set n to 1, 8, and 16. The results are shown in Figure 3-11. We also measured the interrupt disabled periods from Tick handling. The results are shown in Figure 3-12.

The conclusions we can reach from these figures are as follows: First, since Figures 3-11 and 3-12 are basically identical, we can see that Tick handling is run with interrupts disabled. Also, we can see that for every additional task that times out we create about 567 cycles of additional overhead (as well as interrupt disabled time).

These results show that to guarantee real time processing, software engineers using RTOS must be

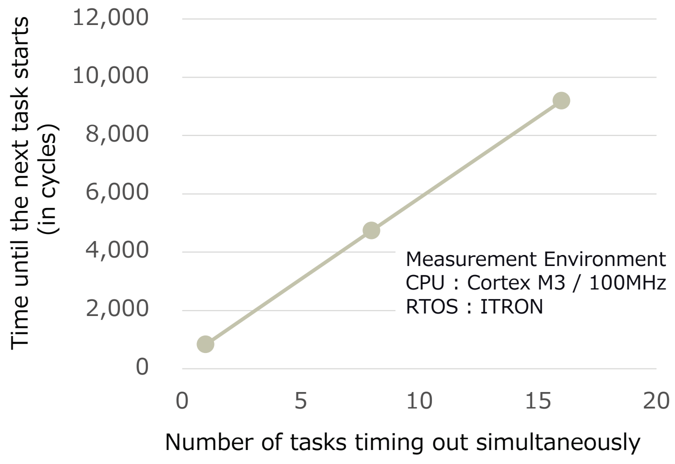

Figure 3-11 Overhead caused by Tick Processing

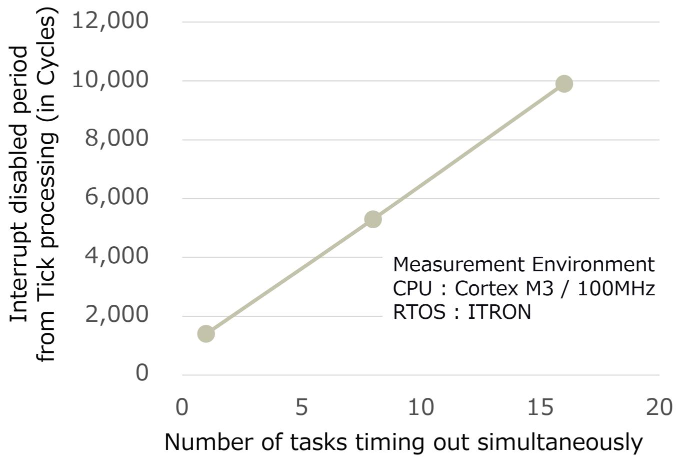

Figure 3-12 Interrupt disabled period from Tick processing

incredibly careful in development. In actuality, there is very little chance that 16 tasks will time out simultaneously. However, typical software engineers probably never consider the impact of the number of simultaneous time outs. If, for example, a number of anomalous conditions pile up and multiple tasks time out simultaneously, normally it is necessary to guarantee real time performance. However, real time performance cannot be guaranteed in this case due to the longer interrupt-disabled period caused by so many time outs for the multiple tasks occurring simultaneously.

So we've discussed overhead and interrupt disabled periods in conventional software RTOS. Execution time with conventional RTOS for even simple APIs, which usually might not involve much overhead at all, might end up taking a very long time to execute depending on the internal state of the RTOS, and the interrupt disabled periods can vary dynamically as well. Tick handling can also have a serious impact on overhead and interrupt disabled periods in much the same way. With a 100MHz CPU, sometimes interrupt disabled periods can last several tens of microseconds, which is cause for serious concern in real time processing.

The RTOS measured in this section was just one of those on the market, and each RTOS will have its own structure, so results may vary. However, without some kind of radical solution, using another RTOS will most likely follow the same trends.

4. Improvement Measures and Performance with HW-RTOS

4.1. API Execution Time Improvement

In this section, we discuss how using HW-RTOS can offer solutions to all the problems shown so far.

Figure 4-1 shows how to invoke APIs in the HW-RTOS. As the figure shows, the API invoked by an application is conveyed as a hardware signal via library software to the HW-RTOS, and the returned value is also received via the library software. The

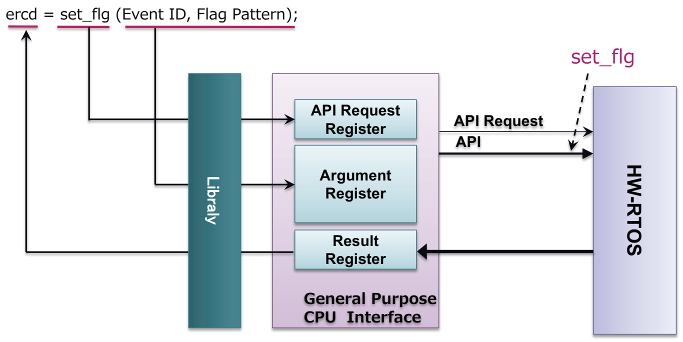

Figure 4-1 How to invoke an API in HW-RTOS

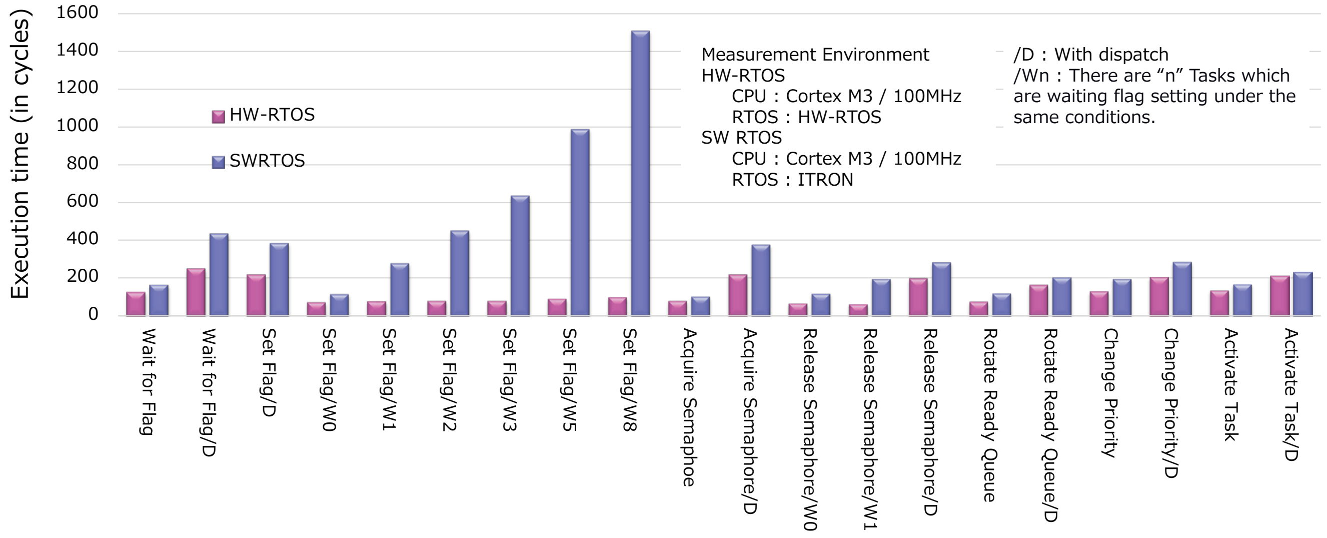

Figure 4-2 API Execution time

library software also executes dispatch processes according to instructions from the HW-RTOS. In Figure 4-2, we can see a comparison of execution time measurements between HW-RTOS and conventional software RTOS (referred to as SW RTOS from now on).

API execution time in HW-RTOS is incredibly fast, and most system calls can be executed within 10 cycles. However, overhead from the library software does lead to some increased execution times as shown in Figure 4-2. With HW-RTOS, API execution times for identical APIs simply don't change depending on internal state, as was shown with Set Flag in SW RTOS. Another huge advantage is that system call worst-case values can be specified in the data sheet. We go over the reasons for that below.

4.2. Queue Handling Improvement

HW-RTOS implements queues with hardware using Renesas' unique technology called "Virtual Queue," [1] [3]. Virtual Queue can append a task to the queue, remove a task from a queue, and also remove a task from the middle of the queue, each within 2

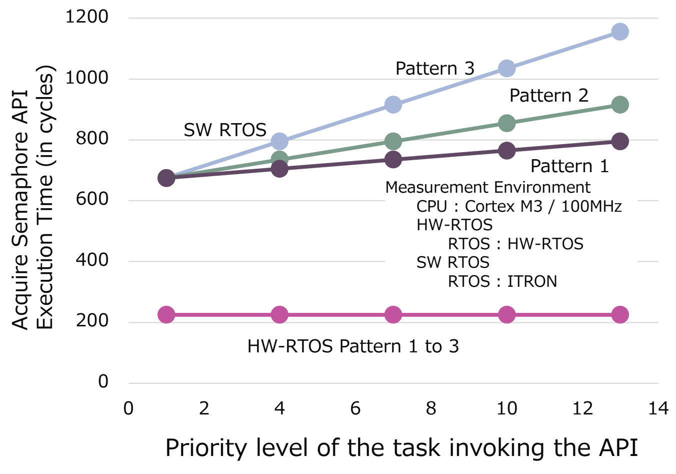

Figure 4-3 Queue Handling and its overhead

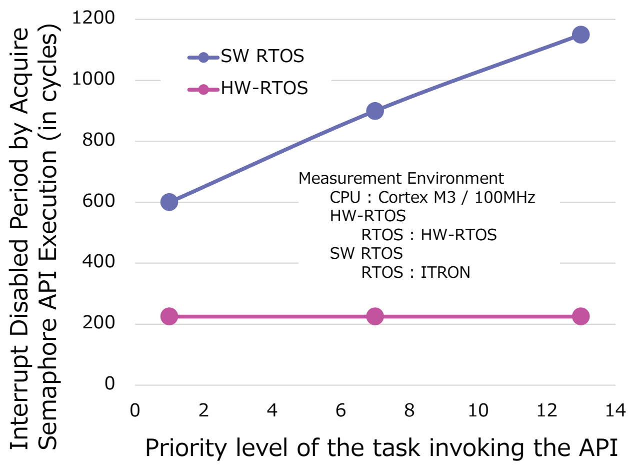

Figure 4-4 Interrupt disabled time by Queue Handling

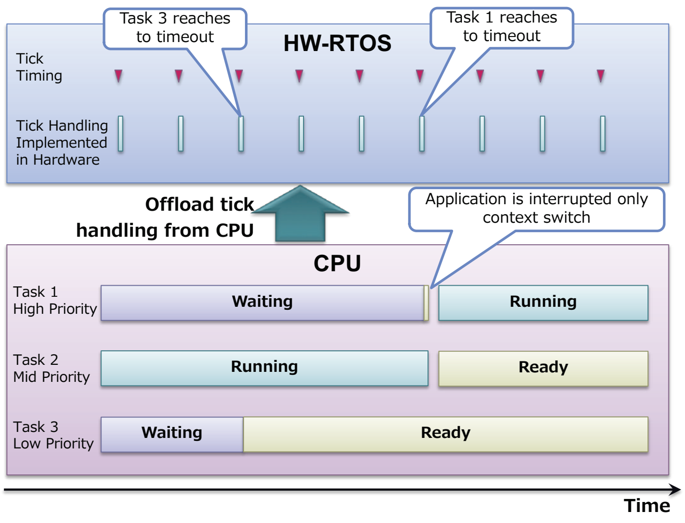
HW-RTOS
Task 3 reaches to timeout
Task 1 reaches to timeout
Tick Timing
Tick Handling Implemented in Hardware
Offload tick handling from CPU
Application is interrupted only context switch
CPU
Task 1 High Priority
Waiting
Running
Task 2 Mid Priority
Running
Ready
Task 3 Low Priority
Waiting
Ready
Time

Figure 4-5 Tick Offloading

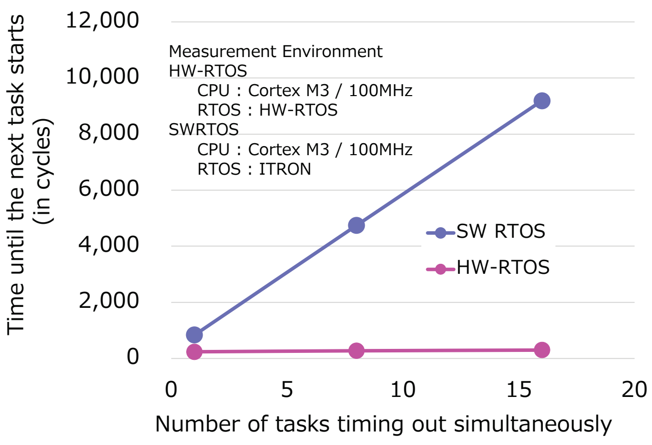

Figure 4-6 Overhead caused by Tick Processing

cycles. As a result, no matter what the state of the queue is, processing can be completed within a specified time.

Figure 4-3 shows the HW-RTOS execution time for the system call Acquire Semaphor, as measured in 3.2. The measurements confirm that, no matter the state of the queues, in HW-RTOS API execution time is fixed. We also measured interrupt disabled periods for pattern 3, as shown in Figure 4-4, and as you can see these are also fixed in HW-RTOS. These all go to show how the HW-RTOS' virtual queue allows for fixed execution times no matter the queue's internal state, and thus fixed overhead and interrupt disabled periods as well. In other words, application software designers can be sure there is no chance of unexpected overhead or interrupt disabled periods. What's more, we can specify the worst-case values for execution time and interrupt disabled periods for the APIs discussed above.

4.3. Tick Handling Improvements

Next we discuss improvements to Tick handling found in HW-RTOS. Tick handling is completely implemented in hardware in HW-RTOS.[2][5] We call this function Tick Offloading. Figure 4-5 shows how Tick Offloading works. As the diagram shows, the Tick function is completely implemented in hardware and installed on the HW-RTOS. As you saw in Figure 3-9, in the past Tick handling was activated by periodic interrupts, but Figure 4-5 demonstrates how on HW-RTOS Tick handling has no need for periodic interrupts, and since there is no need to run the Tick process on the CPU, application software can run uninterrupted even during the Tick. The only time the application will be interrupted (and overhead generated) is when task switch is executed at time out.

And since there are practically no interrupt disabled periods from Tick handling, the only time interrupts are disabled is during task switching at time out, just like overhead. What's more, since Tick handling is extremely fast, it's possible to shorten the Tick interval, and so greatly improve the precision of time outs and the cyclic handler.

So you can clearly see that with the Tick Offloading function, we have solved all three of the problems with conventional Tick handling identified earlier.

In Figure 4-6, we show the HW-RTOS results of the same Tick overhead test shown in Figure 3.3. Namely, we measured the overhead generated by the

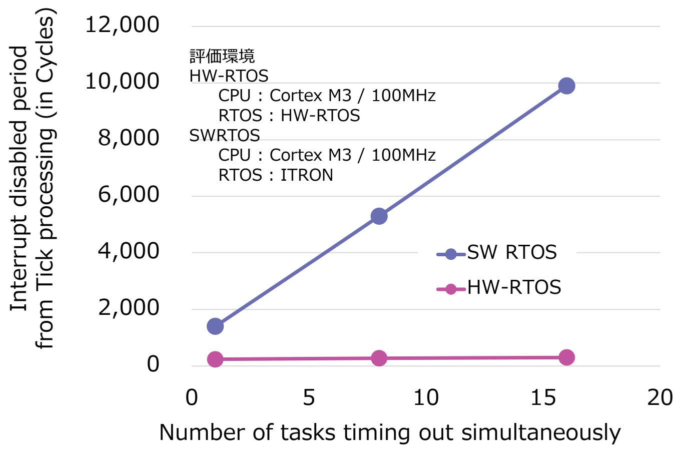

Figure 4-7 Interrupt disabled period from Tick processing

Time Out process when 1, 8 and 16 tasks time out during the same Tick. The chart clearly shows that the overhead in HW-RTOS is 225 cycles, regardless of the number of tasks timing out.

Figure 4-7 shows interrupt disabled periods in the same environment, and just as with overhead, interrupt disabled periods are fixed in HW-RTOS, regardless of the number of tasks timing out.

As we discussed, in SW RTOS Tick handling of the time out process for a Wait task generates lots of overhead, and disables interrupts for this period, so compared to the SW RTOS, HW-RTOS creates less and more stable overhead and interrupt disabled periods.

It's rare for multiple tasks to time out simultaneously, however it's difficult for designers to know in advance if simultaneous time outs will occur, and if they do, to predict how much overhead or how long an interrupt disabled period will occur. That difficulty is just what makes HW-RTOS, with its fixed overhead and interrupt disabled periods even when simultaneous multiple time outs occur, so incredibly effective in real time systems.

5. Conclusion

In conventional software RTOS, it is difficult to guarantee worst-case values for overhead and interrupt disabled periods. The reason for that is that queue handling time and execution time for the Tick process can vary dramatically depending on the RTOS' internal state, because the number of tasks in the Wait queue or tasks timing out simultaneously is always changing. This makes it nearly impossible for the designer to know what to expect. When these issues with conventional software RTOS pile up, it's entirely possible that the system will be unable to meet real time performance needs, resulting in a fatal failure as a real time system.

On the other hand, with HW-RTOS it's possible to define worst-case values for overhead and interrupt disabled periods in advance, so application software designers can define real time performance at the design stage. Using HW-RTOS can greatly reduce the development load on software designers, and allows easy installation of highly reliable real time systems. And with the extremely short interrupt disabled periods, it's also easier to construct a more precise real time system.

6. References

[1] N. Maruyama, T. Ishihara, H. Yasuura,“ An RTOS in Hardware for Energy Efficient Software-based TCP/IP Processing” in Proc. of IEEE Symposium on Application Specific Processors (SASP), 2010, pp. 13-18.

[2] N. Maruyama, T. Ishikawa, S. Honda, H. Takada, K. Suzuki, ”ARM-based SoC with Loosely coupled type hardware RTOS for industrial network systems,” in Proc. of Operating Systems Platforms for Embedded Real-Time applications (OSPERT’14), 2014, pp. 9-16.

[3] Naotaka Maruyama, Tohru Ishihara, Hiroto Yasuura: ”An Energy Efficient Software-Based TCP/IP Processing Method Using an RTOS in Hardware”, The IEICE Transactions on Fundamentals of Electronics,

Communications and Computer Sciences (Japanese Edition), A Vol.J94-A No.9 pp.692-701 (2011).

[4] Naotaka Maruyama, Toshiyuki Ichiba, Shinya Honda, Hiroaki Takada: "A Hardware RTOS for Multicore Systems", The IEICE transactions on information and systems (Japanese edition) D, Vol. J96-D No.10 pp.2150-2162 (2013)

[5] Naotaka Maruyama, Takuya Ishikawa, Shinya Honda, Hiroaki Takada, Katsunobu Suzuki: "Loosely Coupled Hardware RTOS Equipped Production Network SoC",The IEICE transactions on information and systems (Japanese edition)　D, Vol.J98-D No.4 pp.661-673 (2015).