RENESAS

HW-RTOS

Real Time OS in Hardware

Additional Functions in HW-RTOS Offering the Low Interrupt Latency

In this white paper, we introduce two HW-RTOS functions that offer the lowest interrupt latency available and help create a more software-friendly environment. One of these is ISR implemented in hardware, which improves responsiveness when activating a task from an interrupt and eliminates the need for developing a handler in software. The other is a function allowing the use of non-OS managed interrupt handlers in a multitasking environment. This makes it easier to migrate from a non-RTOS environment to a multitasking one.

R70WP0003EJ0100 September, 2018

1. Executive Summary

In this white paper, we introduce two functions special to HW-RTOS that improve interrupt performance.

The first is the HW ISR function. Renesas stylized the ISR (Interrupt Service Routine) process and implemented it in hardware to create their HW ISR. With this function, the task corresponding to the interrupt signal can be activated directly and in real time. And, since the ISR is implemented in the hardware, application software engineers are relieved of the burden of developing a handler.

The second is called Direct Interrupt Service. This function is equivalent to allowing a non-OS managed interrupt handler to invoke an API. This function enables synchronization and communication between the non-OS managed interrupt handler and tasks, a benefit not available in conventional software. In other words, it allows the use of non-OS managed interrupt handlers in a multitasking environment. With this function, it's possible to develop software in multitasking environments even when applications demand extremely low interrupt jitter, such as with high precision servo motors. In sum, you can migrate from previously non-RTOS systems to multitasking environments and achieve high software development efficiency and system reliability.

2. Interrupt Basics

First, to ensure a firm foundation, let's discuss some general information relating to interrupts and RTOS.

2.1. Reducing Interrupt Disabled Periods

The upper part of Figure 1 shows what happens when an interrupt occurs. First, when an interrupt occurs, the corresponding ISR is activated. In this example, when interrupt 1 occurs, ISR1 is activated, and when interrupt 2 occurs, ISR2 is activated. The

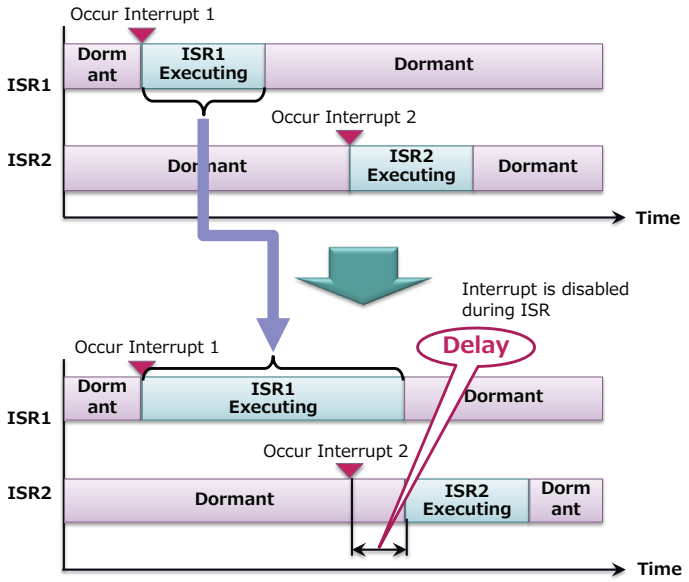

Figure 1 Activate ISR

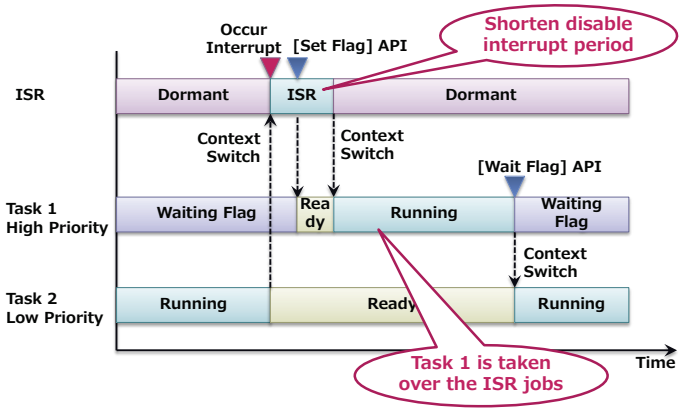

Figure 2 Taking over ISR jobs

problem is that generally, ISRs run with interrupts disabled. As a result, as the lower part of Figure 1 shows, if processing for ISR1 is prolonged, the start of the ISR2 process is delayed. This is not optimal for real time systems.

One way to solve this problem is to hand ISR processing over to a task. That is, the ISR only runs processes that are of the highest importance for real time performance, and for other processes a task is activated in response to the ISR, and the task takes over processing. This is shown in Figure 2. First, Task 2 is running. Then, an interrupt occurs and the ISR is activated. The ISR then activates Task 1,

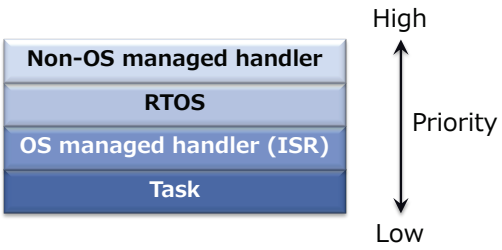

Figure 3 OS / non-OS managed interrupt

which is currently awaiting a flag. So, the Set Flag API is invoked. Task 1 meets the conditions required for release from the Wait state, and then moves to the Ready state. The ISR no longer has any work to do and terminates itself. Then, since Task 1 has a higher priority level than the previously running Task 2, Task 1 is run. Accordingly, while Task 1 is running, if a new interrupt occurs, the start of the corresponding ISR is not delayed. t Thus, by handing ISR processing over to a task, the only thing the ISR needs to do is invoke an API, and all the other processing can be left to the task.

In addition to Set Flag, other APIs used for this goal include Semaphore Release and Wait Release.

Here we are simply using the ISR to invoke an API, but of course it's also acceptable to execute processes with high real time performance within the ISR. It's optimal, though, to execute only the most urgent processes in ISR, and execute all others with Task 1.

As described above, skillful use of the RTOS in interrupt handling allows timely running of high priority processes, as well as interrupt disabled periods made as short as possible.

2.2. OS managed Interrupt and Non-OS managed Interrupt

There are interrupts managed by the OS and those managed outside the OS. As discussed in 2.1, the interrupt-activated handler itself is an interrupt managed by the RTOS. This interrupt-activated handler is called an OS-managed interrupt handler (in this text, we call this OS-managed interrupt handler “ISR”, or “Interrupt Service Routine”). The priority level relationship between the RTOS, the ISR, and tasks is shown in Figure 3.

When we say that the RTOS manages an interrupt, it means that the OS activates the ISR. Namely, when an interrupt occurs, the RTOS activates the ISR corresponding to that interrupt. As you can see from the figure, tasks have the lowest priority level, then the ISR, and next comes the RTOS. Since, as the figure shows, the RTOS is higher priority than the ISR, if an interrupt occurs while the RTOS is in the middle of a process, ISR activation is delayed. As a result, the RTOS creates overhead between the occurrence of an interrupt and ISR activation. Generally, since RTOS processes tend to be critical ones, they are run with interrupts disabled. It's vital to be aware of these delays in ISR activation by OS managed interrupts as described above.

Now we'll discuss non-OS managed interrupts . The handler activated by non-OS managed interrupts is called a non-OS managed interrupt handler. As you can see in Figure 3, non-OS managed interrupt handlers are even higher priority than the RTOS process. As a result, even if the RTOS is in the middle of a process, that process can be interrupted and the handler activated immediately. Since the software has absolutely no role in this activation, the latency up until the handler is activated is the interrupt latency as defined by the CPU. This non-OS managed interrupt handler is often used in applications which simply cannot allow much RTOS overhead, such as high-speed servo motor control.

Another difference between the ISR and non-OS managed interrupt handler is whether or not an API can be invoked during the handler process. Since the ISR is run under RTOS management, it's possible to invoke an API. However, since a non-OS managed

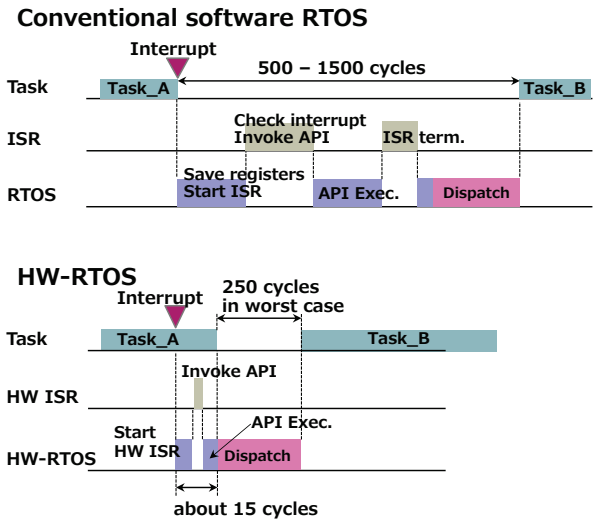

Figure 4 Interrupt Processing (without dispatch)

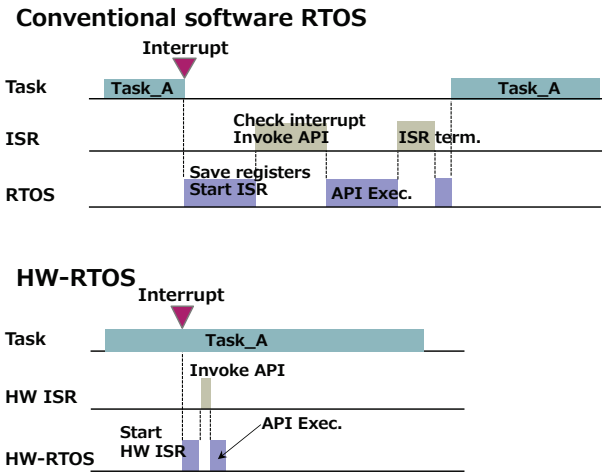

Figure 5 Interrupt Processing (with dispatch)

interrupt handler can be activated while the OS is executing critical processes, naturally it's impossible to use any other RTOS functions. As a result, as Figure 2 shows, it's not possible to hand over a part of the handler process to a task using an API. Not being able to invoke an API during the interrupt handler is a huge disadvantage. Typically, if you trace any software process back to its origin, you'll find an interrupt. To put it the other way around, when a single interrupt occurs, a certain process is activated, and that process then activates another process, so the system is like a chain. As a result, when you can't invoke an API from the handler, particularly synchronization or communication APIs, that's a fatal problem.

Now, having explained a bit about OS-managed interrupts and non-OS managed interrupts, we can sum them up as follows. OS-managed interrupts cause overhead until the ISR is activated, but it is possible to invoke an API while the handler is running. With non-OS managed interrupts, the handler activation time is as fast as the CPU hardware performance, but it's not possible to invoke an API while the handler is running.

3. HW ISR

3.1. HW ISR Function and Operation

The HW ISR function is a hardware implementation of the ISR process. More specifically, this function invokes an API in response to an interrupt signal inside the HW-RTOS. As explained for Figure 2, APIs like Set Flag invoked by the ISR which activate other tasks can be invoked in hardware. By doing so, the ISR from Figure 2 can be implemented completely in hardware.

This is easier to understand with Figure 4. The upper section of Figure 4, "Conventional Software RTOS" is a timing chart for processing interrupts in a conventional software RTOS. When an interrupt occurs, the currently running Task A is interrupted, and processing transfers to the RTOS. The context used for Task A in the RTOS is evacuated, and the RTOS activates the ISR. The interrupt is checked in the ISR, and an API corresponding to the interrupt signal is invoked. In Figure 2, the invoked API was Set Flag. When an API is invoked, processing moves back to the RTOS, and the invoked API is executed in the RTOS. When the API ends, the returned values are sent to the ISR, and the ISR terminates itself. When Task B is in the Ready state based on the

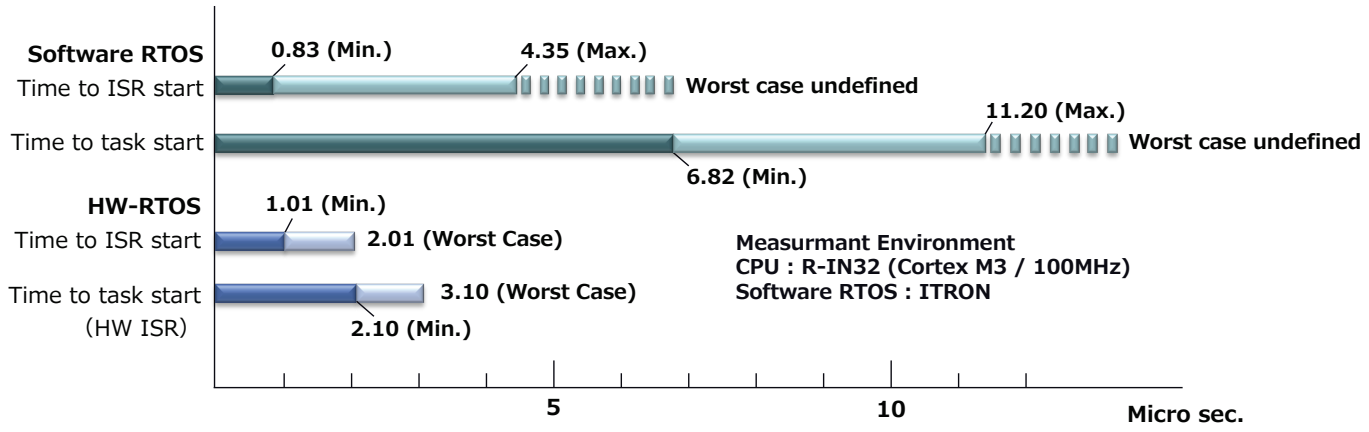

Figure 6 Interrupt responsiveness

result of the executed API, if Task B has a higher priority level than the currently running task, a Task Switch will be necessary. The RTOS then executes a dispatch, and activates Task B.

That's the process for handling interrupts in conventional software. When a single interrupt occurs and the resulting processes are executed as just described, it can consume 500 to 1,500 cycles of CPU time.

The bottom of Figure 4 shows an example of the HW-RTOS HW ISR function in use. When an interrupt occurs while Task A is running, the HW-RTOS is activated. The HW ISR function inside the HW-RTOS is called up, and the API corresponding to the interrupt signal is then invoked inside the HW ISR. The HW-RTOS receives the invocation and executes the API. When Task B is in the Ready state based on the result of the executed API, if the HW-RTOS finds that Task B has a higher priority level than the currently running task, it signals a task switch to the Library Software, and the Library Software executes a dispatch as ordered to activate Task B.

So that's how HW-RTOS uses HW ISR to handle interrupts. As the figure shows, the HW-RTOS processing time is about 15 cycles, and most of the rest of the time used is for the Library Software's dispatch execution. One other point to focus on is that during the period between when the interrupt occurs and the Dispatch begins, the CPU is still running Task A. This is possible because the CPU and the HW-RTOS are running on different hardware, and so can run in parallel.

The upper section of Figure 5 shows, just like in Figure 4, a timing chart for processing interrupts in a conventional software RTOS, but in this example, after the ISR invokes an API there is no need for a context switch based on the execution results. After the ISR completes, processing returns to Task A.

The lower section of Figure 5 shows the same situation, with no task switch needed, in HW-RTOS. As the figure shows, the CPU continues processing Task A. This processing is made possible because the HW-RTOS and CPU can run in parallel. The surprising fact is that even though an interrupt occurs, the CPU processing can continue uninterrupted, and there is no overhead placed on the CPU.

3.2. Interrupt Responsiveness

Figure 6 shows the results of interrupt responsiveness measurements. The CPU used was a 100MHz Cortex M3, and the Software RTOS was an ITRON, which is one of the most commonly used RTOS in Japan. The HW-RTOS was rated at 100MHz, as well.

In the software RTOS, the time until the ISR activated was 0.83 to 4.35 microseconds, in the HW-RTOS it was 1.01 to 2.01 microseconds. Also, the time between when the ISR invoked an API and the next task activated in software RTOS was 6.82 to 11.2 microseconds, while the HW-RTOS time was 2.10 to 3.10 microseconds. The HW-RTOS was using the HW ISR when the time was measured until the task activated. As the figure shows, the maximum measured values for software RTOS depend on the measurement environment, and the actual worst-case values remain undefined. On the other hand, the maximum values for HW-RTOS are the maximum theoretical worst-case values, and the fact that they can never be greater is a major benefit.

So, you can see how using HW-RTOS and HW ISR can result in major improvements in interrupt responsiveness. In addition, this figure doesn't show it but, as described with Figure 5, overhead is dramatically reduced.

3.3. Advantages of HW-ISR

The four APIs that HW ISR can invoke are Set Flag, Release Semaphore, Wakeup Task, and Wait Release. Each one is set to be programmable for every interrupt signal line.

With HW-RTOS, it's possible to use both HW ISR and a software ISR simultaneously. Since the HW ISR is run inside the HW-RTOS, no software processes can be executed in the ISR. As a result, if a developer truly wants to run software processing in the ISR, it's best to activate the software ISR. However, most software developers using HW ISR choose not to also use software ISR since, for the overwhelming majority of applications, HW ISR offers more than sufficient real time performance. Another advantage to using HW ISR is that there's no need to design a handler. Since all the processing operations

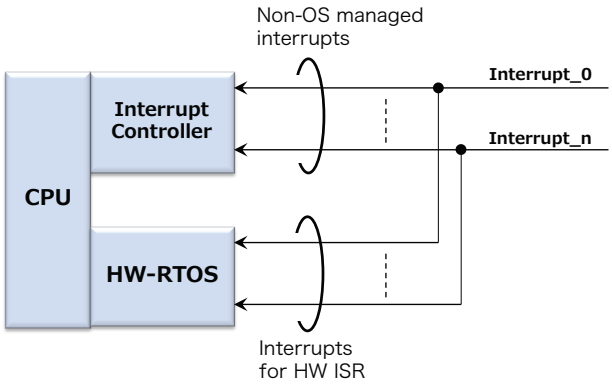

Figure 7 Configuration of Interrupt Acceptance

are done with tasks, most of that pressure is taken off of software engineers.

4. Direct Interrupt Service

4.1. Overview

As explained in 2.2, with OS-managed interrupts, the handler can invoke an API and the process can be handed over to a task, but the interrupt response is slow. On the other hand, when handling non-OS managed interrupts, interrupt response is very fast but it's not possible to hand processing over to a task, so there remains a serious disadvantage for software systems. On the other hand, processing speed is very fast in HW-RTOS, so if HW ISR activates a task directly from an interrupt, sufficient real time performance is guaranteed. However, there may still be a need for even more real time performance. For example, in order to achieve interrupt jitter in the tens to hundreds of nanoseconds range, HW ISR performance is just not enough.

With HW RTOS, there is a function that not only allows similar performance to non-OS managed interrupts, but also allows handover to tasks for processing. We call it Direct Interrupt Service.

Figure 7 is a diagram of interrupts in HW-RTOS. Each interrupt signal is input into both the interrupt controller and the HW-RTOS. The signal input into the interrupt controller becomes a non-OS managed

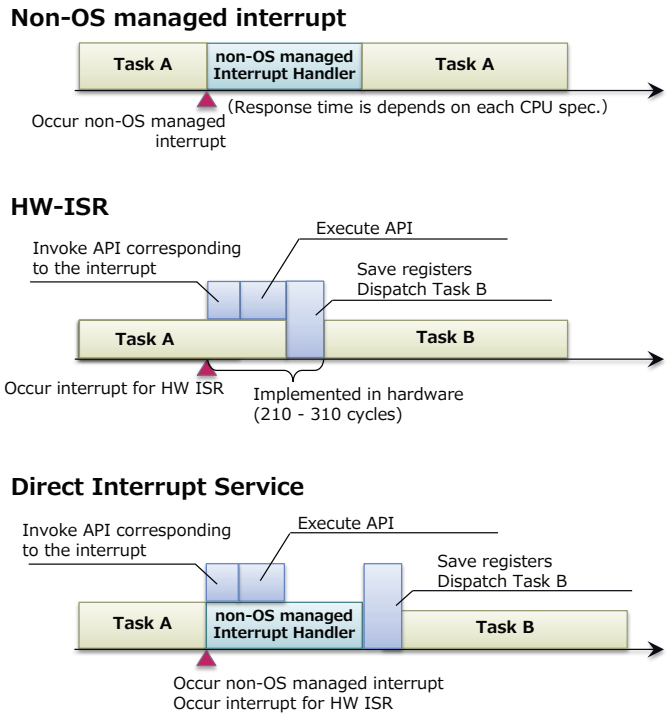

Figure 8 Direct Interrupt Service

interrupt. The interrupt signal input to the HW-RTOS is used to activate the HW ISR. Each input section has its own Mask Register to select the input signal. The typical setting is that when one is configured to Through, the other is set to Mask. However, by setting both to Through, you can implement a Direct Interrupt Service. Figure 8 explains this in detail.

The timing chart at the top of Figure 8 shows what happens when interrupt signals are set to Through for the interrupt controller alone. In this situation, when an interrupt occurs, the non-OS managed interrupt handler activates. The center section of Figure 8 is a timing chart for when interrupt signals are set to Through for the HW-RTOS alone. In this case, the HW ISR is activated.

The bottom, then, shows what happens if we set interrupt signals to Through for both the Interrupt Controller and the HW-RTOS. Namely, when an interrupt signal is generated, the non-OS managed interrupt handler activates, and at the same time the HW-RTOS invokes the corresponding API internally. That API executes in parallel with the non-OS managed interrupt handler, and in this example Task B is released from the Wait state. Task B activates at the same time as the non-OS managed interrupt handler ends.

4.2. Advantages of Direct Interrupt Service

Using the Direct Interrupt Service like this allows both the non-OS managed Interrupt Process and the task corresponding to the interrupt to activate. This is equivalent to achieving the non-OS managed interrupt handler's ability to invoke an API and synchronize and communicate between other tasks. In short, this means it's possible to implement the interrupt responsiveness of the non-OS managed interrupt management, while still being able to invoke an API from within the handler at the same time.

In the past, to keep interrupt response jitter below the tens of microseconds range, we had to forfeit a multitasking environment. But now, using Direct Interrupt Service makes it possible. It's now possible to run multitasking processes like network control on a single CPU along with high precision servo motor control. As a result, not only does CPU use-efficiency increase, but we can expect the software development environment to improve as well. The reason for that is, since we could not use RTOS in conventional servo motor control, software engineers needed expert skills to construct a system with multiple functions. Not only did that keep development productivity down, but also reduced system reliability. But using Direct Interrupt Service offers solutions to all of those problems.

5. Conclusion

Typically, interrupt handlers run with interrupts disabled. In order to keep this interrupt disabled period as short as possible, it's important to hand processing over to tasks. To that end, the interrupt

handler can invoke APIs like Set Flag or Semaphore Release, and by activating waiting tasks, let them take over processing. Also, the fundamental truth that "basically, all processes can be traced back to an interrupt" is vital for understanding the succession from interrupt handler to task. As described above, HW-RTOS has made it possible to implement processes from interrupt occurrence to releasing the waiting task from the Wait state in hardware. This is what we call HW ISR. This function has enabled extremely high-speed activating of tasks corresponding to interrupt signals.

And for those applications requiring even less interrupt response jitter, HW-RTOS also offers the Direct Interrupt Service function. Using it allows both the non-OS managed interrupt handler and the task corresponding to the interrupt to activate. This completely solves the past problems of synchronization and communication between tasks and non-OS managed interrupt handlers, and the inability of the non-OS managed interrupt handler to hand processing over to tasks.

And so, we can see that HW-RTOS is more than merely a hardware implementation of API processes, but also achieves the fastest possible interrupt responsiveness as well as offering ease of use for software systems.

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).