Implementation of Interrupt Handlers in Full Hardware Implementation of RTOS-Based Systems

Yuki NAKATANI[†] Nagisa ISHIURA[††]
[†] Graduate School of Science and Technology [††] School of Engineering
Kwansei Gakuin University
1 Gakuen Uegahara, Sanda, Hyogo, 669-1330, Japan

Hiroyuki TOMIYAMA Hiroyuki KANBARA
College of Science and Engineering,
Ritsumeikan University
1-1-1 Noji-Higashi, Kusatsu, Shiga, 525-8577 Japan

Abstract—This paper proposes a method for implementing interrupt handling in the context of a fully hardware-implemented RTOS-based system. To enhance the responsiveness of real-time systems, previous research by Oosako et al. has explored the complete hardware implementation of RTOS functionalities and tasks. Ando has proposed an architecture that enables task hardwareization using general-purpose high-level synthesis; however, this approach does not address interrupt handling. In this paper, we present a hardware implementation of alarm handlers, cyclic handlers, and interrupt handlers. Each handler is associated with a dedicated timer that counts down and triggers the handler when the counter reaches zero, thereby simplifying the control logic. The interrupt handler is designed to accelerate activation by evaluating invocation conditions in parallel. Service calls related to handlers are implemented by updating or referencing specific status registers. Experimental results demonstrate that, in the proposed RTOS hardware, alarm and interrupt handlers can be triggered within 1 and 2 cycles, respectively, after their conditions are satisfied. Furthermore, all service calls related to handlers can be executed within 5 cycles.

I. Introduction

Recent advances in information and communication technology have led to the rapid development of new services and devices. As a result, embedded systems are increasingly required to support more advanced and sophisticated functionalities. In particular, systems used in automotive equipment and unmanned aerial vehicles demand not only functionality but also strict real-time performance. Such systems are typically designed using a real-time operating system (RTOS), which provides mechanisms to ensure that processing in response to input events is completed within a defined time frame. However, achieving real-time performance is becoming increasingly challenging as system complexity grows.

One common approach to improving the responsiveness of RTOS-based systems is to implement RTOS functionalities in hardware. For example, references [1, 2] propose hardware implementations of the RTOS scheduler, while [3, 4] implement the entire RTOS in hardware. However, in these approaches, tasks and handlers remain as software components, and therefore issues such as CPU waiting and context-switching overhead are not fully addressed.

In contrast, the approach presented in [5] proposes implementing not only the RTOS functionalities but also all tasks in hardware. In this method, tasks are executed in parallel as independent hardware modules, eliminating CPU waiting and context-switching overhead. Ando et al. [6] further proposed an architecture and method for generating task hardware modules from source code using general-purpose high-level synthesis tools. Additionally, [7] presents hardware implementations of RTOS services such as mutexes and data queues. However, these studies focus solely on user tasks, and do not address support for external or timer-based interrupts.

This paper extends the architecture proposed in [6] by introducing methods to implement alarm handlers, cyclic handlers, and interrupt handlers entirely in hardware. By assigning a dedicated timer to each handler, the control logic for alarm and cyclic handlers is simplified. Service calls related to handlers are implemented through updates or references to associated status registers.

The extended RTOS hardware, designed based on the proposed method, achieves fast response times: alarm and interrupt handlers are triggered in 1 and 2 cycles, respectively, after their activation conditions are met. Furthermore, all service calls related to handlers can be executed within 5 cycles. Logic synthesis targeting the Xilinx Artix-7 FPGA revealed that a configuration with four tasks and one instance of each handler type requires 1,727 LUTs—approximately a 1.52 times increase compared to a configuration without handling support.

The remainder of this paper is organized as follows. Chapter 2 first discusses the hardware implementation and architectural design of RTOS-based systems, then provides an overview of handlers in RTOS. Chapter 3 details the implementation methods for alarm, cyclic, and interrupt handlers, along with the hardware realization of service calls related to these handlers. Chapter 4 presents an evaluation of the proposed approach in terms of circuit area, handler response cycles, and the execution cycles of service calls.

II. Full Hardware Implementation of RTOS-Based Systems

A. Full Hardware Implementation of RTOS-Based Systems

Oosako et al. proposed a method for implementing both RTOS functionalities and all tasks/handlers entirely in hardware [5]. The concept is illustrated in Fig. 1. The upper part of the figure shows a system where an RTOS and tasks (`TSK`i) are implemented in software, with tasks executed by a CPU under the management of the RTOS. The lower part shows the entire system implemented in hardware. Each task (`TSK`i) is synthesized as an independent hardware module using high-level synthesis. The *manager* module provides the RTOS functions as hardware and offers task execution control, as well as inter-task synchronization and communication mechanisms such as mutexes and data queues.

In this system, since all tasks in the ready state are executed in parallel, there is no CPU waiting nor context-switching overhead. Since the scheduling function is reduced to the simple task of sending execution signals to tasks in the ready state, the scheduling overhead is minimized. Furthermore, by implementing tasks in hardware, they can be executed at high speed, significantly improving the system's responsiveness.

Although this approach is generally applicable only to systems with up to approximately 16 tasks due to the impracticality of implementing a large number of tasks in hardware in terms of circuit area, it can dramatically improve the system's response performance.

B. Architecture by Ando and Muguruma

In contrast to the approach in [5], where the RTOS service routines linked to tasks were synthesized into hardware using a proprietary binary synthesis system, Ando and Muguruma proposed an architecture where the RTOS service routines are implemented within the *manager* hardware module, significantly reducing circuit size and improving execution speed [6]. They also developed an architecture that enables the hardware synthesis of tasks using general-purpose high-level synthesis tools.

This paper adopts the architecture proposed in [6] as a premise. Fig. 2 shows this architecture. `TSK0` to `TSK2`

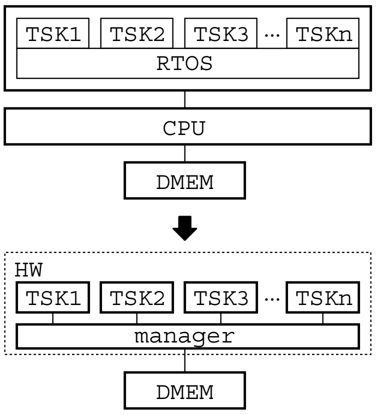

Fig. 1. Full hardware implementation of RTOS-based system [5]

represent the tasks, and the *manager* module implements the RTOS functionalities in hardware. Located below the *manager* are the service modules including *control*, *shared variable*, *data queue*, *event flag*, and *mutex*, which provide various services. Among these, the *control* module handles service calls for updating task states and priorities. In this architecture, accesses to shared variables are treated as services and processed by the *shared variable* module. The *tsk status* register holds the states and priorities of each task (the states of the kernel are kept in the *global status* register in the *control* module).

While all ready tasks are executed in parallel, service calls are processed sequentially to avoid interference between services. In such cases, the task with the highest priority is selected by the Request Arbiter (RA) and executed. The *wait* register manages which task is waiting for which service.

A task (`TSK`i) requests a service by writing the service number to the control register `TF`i and the required arguments to `TA`i. The RA references the states of the requesting tasks and performs arbitration, writing the selected task number, service number, and necessary arguments to the `XT`, `XF`, and `XA`j registers, respectively. The service module responsible for the service number written to `XF` processes the request and writes the return value to the `XA0` register. Once the return value is written, the *manager* writes the value from `XA0` to `TA`i and notifies the task of service completion. Upon receiving this notification, the task resumes its processing. The implementation of the service modules is described in detail in [7, 8].

C. Handlers

Handlers are routines executed when specific events or conditions occur. In this paper, we focus on alarm handlers, cyclic handlers, and interrupt handlers.

An alarm handler is executed once after a specified time has elapsed since it was started. A cyclic handler is executed repeatedly at a specified time interval after being started.

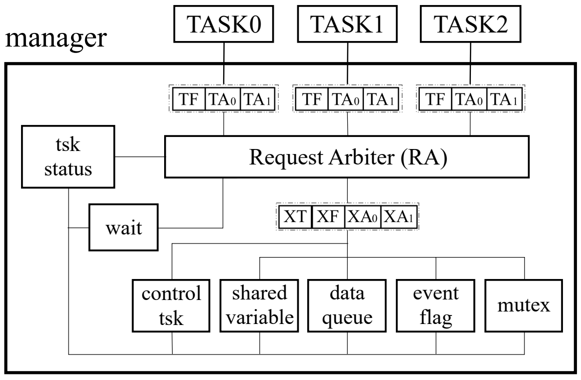

Fig. 2. Architecture proposed in [6]

An interrupt handler is executed when a signal is sent to the IRQ (Interrupt Request) port or when it is explicitly triggered by a task. However, the execution of an interrupt handler is delayed if the kernel has globally disabled interrupts (interrupt lock flag is set), or if the specific interrupt has been disabled (interrupt lock flag for that interrupt is set). Additionally, each interrupt is assigned a priority, and if this priority is lower than the system's interrupt priority mask, the execution is also delayed.

Handlers, like tasks, have execution priorities, but they always have higher priorities than tasks. If multiple time events or interrupts occur simultaneously, the handler or event with the highest priority is executed first.

The control and status referencing of handlers are performed via service calls, which can be invoked from tasks or other handlers. For example, in TOPPERS/ASP3[1], the service calls listed in Table I are defined. For alarm and cyclic handlers, as shown in (a) and (b), service calls allow the initiation with a specified time, termination, and status referencing.

For interrupt handlers, as shown in (c), service calls allow tasks to request or clear interrupts, enable or disable interrupts, and set or reference interrupt priorities.

III. Hardware Implementation of Interrupt Handling

A. Overview

In this paper, we present implementation of interrupt handling mechanisms within the RTOS hardware architecture proposed in [6]. The target handlers include alarm handlers, cyclic handlers, and interrupt handlers. Similar to tasks, each handler is synthesized as an independent hardware module using a high-level synthesis (HLS) system. The execution of these handlers is handled in the same manner as tasks.

For the alarm and cyclic handlers, each handler is equipped with its own dedicated timer, which simplifies the control logic for their activation. This approach ensures that the management of the handler's activation can be performed independently, without the need for complex centralized control.

For interrupt handlers, the evaluation of activation conditions is performed in parallel, allowing for faster invocation of the handler upon the occurrence of an interrupt. The activation control for these handlers is managed by both the *manager* module and the *control* module.

Furthermore, the processing of service calls related to handlers—such as setting the activation conditions for alarm, cyclic, and interrupt handlers from tasks—is implemented in hardware by directly updating or referencing the status registers and timer values associated with each handler. This method ensures efficient and deterministic execution of handler-related service calls.

TABLE I
Service calls related to handlers

(a) Service calls for alarm handlers

`sta_alm(ID alm, RELTIM tim)`	Starts alarm notification `alm` so that the handler is triggered after `tim` has elapsed.
`stp_alm(ID alm)`	Stops alarm notification `alm`.
`ref_alm(ID alm)`	References the status of alarm notification `alm`.

(b) Service calls for cyclic handlers

`sta_cyc(ID cyc)`	Starts cyclic notification `cyc`.
`stp_cyc(ID cyc)`	Stops cyclic notification `cyc`.
`ref_cyc(ID cyc)`	References the status of cyclic notification `cyc`.

(c) Service calls for interrupt handlers

`ras_int(ID it)`	Requests interrupt `it`.
`clr_int(ID it)`	Clears the request for interrupt `it`.
`dis_int(ID it)`	Disables interrupt `it`.
`ena_int(ID it)`	Enables interrupt `it`.
`prb_int(ID it)`	References the request status of interrupt `it`.
`chg_ipm(PRI pri)`	Updates the interrupt priority mask to `pri`.
`get_ipm(PRI *p_pri)`	References the interrupt priority mask.

B. Control of Alarm Handlers

In this paper, unlike conventional software implementations, each alarm handler is equipped with a dedicated timer. The timer is initialized by a service call that activates the alarm. The *manager* module decrements the timer at every clock cycle, and when the timer reaches zero, the handler's state is transitioned to an executable state.

Table II shows the information stored in the status registers of the alarm handler. The `almstat` register is a status flag indicating whether the alarm is active; it is set to 1 when the alarm is triggered. The `almpri` register holds the priority of the handler. The `almtim` register is a countdown timer representing the relative time until notification, which is decremented every clock cycle while `almstat` is set to 1. The `alm_handlerstat` register indicates the execution state of the handler.

[1]https://www.toppers.jp/

TABLE II
Status registers of alarm handler

Variable Name	Description
`almstat`	Alarm status flag
`almpri`	Alarm handler priority
`almtim`	Relative time until notification
`alm_handlerstat`	Execution state of the alarm handler

The operational flow of the alarm handler is as follows. The *manager* module controls the alarm and alarm handler, while the *control* module processes service calls related to the handler.

0) In the initial state, `almstat` is set to 0, `alm_handlerstat` is in the Dormant (DMT) state, and `almpri` stores the handler's priority.
1) When a task calls `sta_alm(alm, tim)` to start the alarm, the *control* module sets `almstat` to 1 and initializes `almtim` with the specified `tim` value.
2) The *manager* module decrements `almtim` at every clock cycle while `almstat` is 1.
3) When `almtim` reaches zero, the *manager* transitions `alm_handlerstat` to the Running state and resets `almstat` to 0.
4) Upon completion of the handler's execution, the alarm handler sets a completion flag to notify it to the *manager*.
5) The *manager* transitions `alm_handlerstat` back to the Dormant state.
6) The alarm returns to its initial state.

C. Control of Cyclic Handlers

The control of cyclic handlers is similar to that of alarm handlers. The primary difference is that when the countdown timer reaches zero, the handler not only transitions to the running state but also has its timer automatically reloaded with the predefined cycle period. This reloading process continues repeatedly until a service call to stop the cyclic handler (`stp_cyc`) is issued, at which point both the status flag and the timer reset flag are cleared.

Table III lists the information stored in the status registers of each cyclic handler. Similar to alarm handlers, these registers include the status flag, priority, relative time, and execution state of the handler. Additionally, a flag named `cyc_reset` is introduced to control the automatic reloading of the cycle period. When `cyc_reset` is set to 1 and `cyctim` reaches zero, the `cyc_handlerstat` is transitioned to the running state, and `cyctim` is reloaded with the predefined cycle period.

TABLE III
Status registers of cyclic handler

Variable Name	Description
`cycstat`	Status flag for cyclic notification
`cycpri`	Priority of the cyclic handler
`cyctim`	Relative time until the next cyclic notification
`cyc_reset`	Timer reset flag
`cyc_handlerstat`	Execution state of the cyclic handler

D. Control of Interrupt Handlers

The interrupt handler transitions to the running state either upon receiving an IRQ (interrupt request) signal from the external port or upon an interrupt request service call by a task. The status registers associated with each interrupt handler are summarized in Table IV.

The priority and execution state of the interrupt handler are managed similarly to other types of handlers. The `intstat` register serves as the interrupt request flag, which is set when an interrupt request is issued. The `int_locked` register is an interrupt lock flag; when this flag is set to 1, the interrupt handler cannot transition to the ready or running state.

The operational flow of the interrupt handler is as follows:

0) In the initial state, `intstat` is 0, `int_locked` is 0, and `int_handlerstat` is set to Dormant (DMT). The `intpri` register stores the priority of the interrupt handler.
1) When the IRQ signal is received or a task issues an interrupt request by invoking `ras_int(intid)`, `intstat` is set to 1.
2) The *manager* module evaluates whether all of the following conditions are satisfied:
   - `int_locked` is 0.
   - The priority of the interrupt handler is higher than the interrupt priority mask value.
   - The global interrupt lock flag is 0.
   - The CPU lock flag is 0.
3) If all the above conditions are met, the *manager* transitions `int_handlerstat` to the running state and resets `intstat` to 0.
4) Upon completion of the handler's execution, the handler sets a completion flag and notifies the *manager*.
5) The *manager* transitions `int_handlerstat` back to the dormant state.
6) The handler returns to the initial state.

TABLE IV
Status registers of interrupt handler

Variable Name	Description
`intstat`	Interrupt request flag
`intpri`	Interrupt priority
`int_locked`	Interrupt lock flag
`int_handlerstat`	Execution state of the interrupt handler

E. *Hardware Implementation of Service Calls*

Service calls related to alarm handlers, cyclic handlers, and interrupt handlers are processed by the *control* module, which is responsible for the execution control of tasks. The basic processing involves simple referencing or updating of the handlers' status registers, allowing all service calls to be completed within a single cycle (including service call invocation overhead, the total processing time is five cycles).

The processing for service calls related to alarm handlers is shown in Table V(a). Here, `out_`*i* denotes the output port of the *control* module, where the returned values are sent back to the task. The `sta_alm` call sets `almstat` to 1 and assigns the specified value to `almtim`. The `stp_alm` call resets `almstat` to 0. The `ref_alm` call outputs the current values of `almstat` and `almtim` to the output ports for reporting to the *manager* module.

Similarly, Table V(b) shows the processing for service calls related to cyclic handlers. The behavior is mostly the same as that of alarm handlers; however, in the cases of `sta_cyc` and `stp_cyc`, the `cyc_reset` flag is set to 1 and 0, respectively, to control the automatic resetting of the cycle timer.

Table V(c) illustrates the processing of service calls related to interrupt handlers. The `ras_int` and `clr_int` calls set and reset `intstat`, respectively. The `dis_int` and `ena_int` calls set and reset `int_locked`, respectively. The `prb_int` call outputs the value of `intstat` to the output port for reporting to the *manager*. The `chg_ipm` call assigns the specified priority value to the global interrupt priority mask register `intpri_mask`, while the `get_ipm` call outputs the current value of `intpri_mask` to the output port.

Upon receiving a service call request, the *control* module determines the appropriate output and the required updates to the status registers of each handler. It then communicates these changes to the *manager*, which applies them to the respective status registers, thereby completing the service call processing in hardware.

TABLE V
Service call processing for handlers

(a) Service calls for alarm handlers

`sta_alm(ID alm, RELTIM tim)`	`almstat <= 1,  almtim <= tim`
`stp_alm(ID alm)`	`almstat <= 0`
`ref_alm(ID alm)`	`out_1 <= almstat,  out_2 <= almtim`

(b) Service calls for cyclic handlers

`sta_cyc(ID cyc)`	`cycstat <= 1,  cyctim <= tim` `cyc_reset <= 1`
`stp_cyc(ID cyc)`	`cycstat <= 0,  cyc_reset <= 0`
`ref_cyc(ID cyc)`	`out_1 <= cycstat,  out_2 <= cyctim`

(c) Service calls for interrupt handlers

`ras_int(ID it)`	`intstat <= 1`
`clr_int(ID it)`	`intstat <= 0`
`dis_int(ID it)`	`int_locked <= 1`
`ena_int(ID it)`	`int_locked <= 0`
`prb_int(ID it)`	`out_1 <= intstat`
`chg_ipm(PRI pri)`	`intpri_mask <= pri`
`get_ipm(PRI *p_pri)`	`out_1 <= intpri_mask`

IV. Implementation and Evaluation Results

Based on the proposed method, the *manager* module was designed using Verilog HDL and synthesized targeting the Xilinx FPGA Artix-7 (xc7a100tcsg324-3) using Xilinx Vivado 2023.2.

The logic synthesis results of the *manager* module are shown in Table VI. In the column #task/handler, `tsk`, `alm`, `cyc`, and `int` represent the number of tasks, alarm handlers, cyclic handlers, and interrupt handlers, respectively. #LUT denotes the number of lookup tables, and *delay* indicates the critical path delay. In this experiment, *manager* module only implements the *control* module as the service module and does not include modules such as mutexes or event flags. The circuit size is approximately proportional to the total number of tasks and handlers (232.1 LUTs per task/handler). Although the critical path delay slightly increases as the number of tasks and handlers increases, it remains under 10 ns.

The response cycles of the handlers are presented in Table VII. For alarm and cyclic handlers, the response cycle refers to the number of cycles from the timer reaching zero until the handler begins execution. For interrupt handlers, it refers to the number of cycles from when an interrupt signal is input to the IRQ port until the handler begins execution. Alarm and cyclic handlers can be invoked in 1 clock cycle, while interrupt handlers are invoked in 2 clock cycles, demonstrating extremely low-latency handler activation.

Table VIII shows the response cycles for service calls related to handlers. These represent the number of cycles from when a task or handler issues a service call to the point it receives a return value, assuming the call is processed without being delayed by other tasks or handlers. All service calls can be completed within 5 clock cycles, achieving very high-speed execution.

TABLE VI
Logic synthesis results of the manager module

#task/handler	#LUT	delay [ns]
tsk×8	1,828	5.367
tsk×8, cyc×1	1,857	5.154
tsk×8, cyc×2	2,310	5.228
tsk×8, cyc×4	2,731	5.282
tsk×8, alm×1	2,003	5.192
tsk×8, alm×2	2,705	5.142
tsk×8, alm×4	3,052	5.583
tsk×8, int×1	1,844	5.374
tsk×8, int×2	2,348	5.398
tsk×8, int×4	2,844	5.303
tsk×8, alm×1, cyc×1, int×1	2,354	5.152
tsk×8, alm×2, cyc×2, int×2	3,554	5.856

TABLE VII
Response cycles of handlers

Handler	#Cycle
Alarm handler	1
Cyclic handler	1
Interrupt handler	2

V. Conclusion

In this paper, we have proposed a method for implementing interrupt handling in the context of fully hardware-implemented RTOS-based systems. The proposed approach includes the hardware implementation of the control mechanisms for alarm handlers, cyclic handlers, and interrupt handlers, as well as the service calls associated with these handlers. Although the circuit size of the management hardware (*manager*) increases approximately in proportion to the number of tasks and handlers, both interrupt activation and service call processing can be executed with extremely low latency.

As future work, we plan to develop a system that automatically synthesizes the entire hardware from the source code of tasks and handlers, and to explore the application of the proposed method to practical use cases.

Acknowledgements

Authors would like to express their appreciation to Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their discussion and valuable comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin University. This work was partly supported by JSPS KAKENHI under Grant No. 24K14885.

TABLE VIII
Response cycles of service calls

Service Call	#Cycle
`sta_alm`	5
`stp_alm`	5
`sta_cyc`	5
`stp_cyc`	5
`ras_int`	5
`clr_int`	5
`dis_int`	5
`ena_int`	5
`chg_ipm`	5

References

[1] Y. Cho, S. Yoo, K. Choi, N-E Zergainoh, and A. A. Jerraya: "Scheduler implementation in MPSoC design," in *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC 2005)*, pp. 151–156 (Jan. 2005).

[2] M. Vetromille, L. Ost, C. A. M. Marcon, C. Reif, and F. Hessel: "RTOS scheduler implementation in hardware and software for real time applications," in *Proc. Proc. International Symposium on Rapid System Prototyping (RSP '06)* pp. 163–168 (June 2006).

[3] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai: "Performance evaluation of STRON: A hardware implementation of a real-time OS," in *IEICE Trans. Fundamentals*, vol. E82-A, no. 11 pp. 2375–2382 (Nov. 1999).

[4] N. Maruyama, T. Ishihara, and H. Yasuura: "An RTOS in hardware for energy efficient software-based TCP/IP processing," in *Proc. Symposium on Application Specific Pr ocessors (SASP 2010)*, pp. 58–63 (June 2010).

[5] Y. Oosako, N. Ishiura, H. Tomiyama, and H. Kanbara: "Synthesis of Full Hardware Implementation of RTOS-Based Systems," in *Proc. International Symposium on Rapid System Prototyping (RSP 2018)*, pp. 1–7 (Oct. 2018).

[6] T. Ando, I. Muguruma, Y. Ishii, N. Ishiura, H. Tomiyama, and H. Kambara: "Full Hardware Implementation of RTOS-Based Systems Using General High-Level Synthesizer," in *Proc. Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI 2022)*, pp. 2–7 (Oct. 2022).

[7] H. Minamiguchi, M. Nakahara, Y. Ishii, Y. Shinohara, I. Muguruma, and N. Ishiura: "Hardware RTOS Services for Full Hardware Implementation of RTOS-Based Systems," in *Proc. Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI 2022)*, pp. 14-19 (Oct. 2022).

[8] M. Nakahara and N. Ishiura: "Arrival Order Processing of Service Requests in Full Hardware Implementation of RTOS-Based Systems," in *Proc. International Technical Conf. on Circuit/Systems, Computers and Communications (ITC-CSCC 2023)*, pp. 467-472 (June 2023).