ARM-based SoC with Loosely coupled type hardware RTOS for industrial network systems

Naotaka Maruyama*†, Takuya Ishikawa†, Shinya Honda†, Hiroaki Takada† and Katsunobu Suzuki‡
*KERNELON Silicon Inc., 520-6, Fujisawa, Fujisawa-shi, 251-0052, Japan
Email: maruyama_na@kernelon.com
†Graduate School of Information Science, Nagoya University, Furo-cho, Chikusa-ku, Nagoya, 464-8603, Japan
Email: {t_ishikawa, honda, hiro}@ertl.jp
‡2nd Solution Business Unit, Renesas Electronics Corp., 1753, Simonumabe, Nakahara-ku, Kawasaki-shi, 211-8668, Japan
Email: katsunobu.suzuki.fn@renesas.com

***Abstract*—Many different types of high-speed networks are employed in industrial systems, which affect real-time processing, such as motor control of industrial controllers, because of the increased CPU load due to network protocol processing. In this study, we propose a system-on-a-chip (SoC) architecture for industrial controllers to reduce the network protocol processing load. The proposed architecture adopts a RTOS in hardware in order to accelerate RTOS execution because the RTOS is frequently called in protocol processing and commonly used in many protocols. Existing hardware RTOSs require a purpose-built core which has a special interface to connect with the hardware RTOS in a tightly coupled manner. However, ARM processors are required in many industrial systems. We manufactured a SoC for Industrial controllers using the proposed architecture and it is the world's first SoC with hardware RTOS, and it is commercially available. The results of our experimental evaluations showed that the API execution time of the proposed architecture was 1.4–2.9 times faster and the UDP/IP throughput of the proposed architecture was 1.67 times faster compared with that when using a conventional software RTOS.**

I. Introduction

In factories, low speed networks such as controller area networks (CANs) have been used as industrial networks. However, high-speed networks such as 10/100/1000 Mbps Ethernet have been employed recently. Industrial controllers (IndCntlrs), which control industrial devices such as motors, require real-time processing. However, the deployment of high-speed network systems affects the real-time processing of IndCntlrs. High-speed network protocol processing occupies a large amount of CPU time and generates frequent interrupts, thus it is necessary to decrease the network protocol processing load of IndCntlrs.

In contrast to office networks, industrial networks are required to meet real-time constraints. This requires periodic and deterministic data transfer to ensure synchronization among the IndCntlrs connected by the network. An Ethernet is adopted as a physical layer in industrial network protocol stacks but many protocols, such as PROcess FIeld NETwork (PROFINET), Ethernet industrial protocol (EtherNet/IP), ModbusTCP, and Ethernet for control automation technology (EtherCAT), have been proposed for use as the upper layer of industrial network protocol stacks to satisfy real-time constraints[14]. Therefore, multi-protocol support is required for IndCntlrs. Furthermore, improved real-time processing is also required to satisfy the real time constraints.

The adoption of ARM cores is required for IndCntlr because users want to utilize the properties of software that have already been developed and they need to retain the same integrated development environments. Furthermore, ARM cores provide scalability because they have many lineups and they are also reliable due to the fact they have been implemented in a huge number of embedded systems throughout the world. For these reasons, major semiconductor manufacturers have moved their products from proprietary cores to ARM cores.

Low power consumption and low costs are important because industrial systems comprise a large number of IndCntlrs, which are connected via a network.

As mentioned above, the requirements for industrial network systems are: (1) decreasing the load of network protocol processing, (2) improved real-time processing, (3) multi-protocol support, (4) adoption of ARM cores, and (5) low costs and low power consumption.

One approach that satisfies requirements (1) and (2) is to adopt a network protocol offload engine. However, to satisfy requirement (3), the engines have to be designed and implemented for all protocol types. Furthermore, this approach lacks flexibility, needs a greater silicon area in the SoC, and is more expensive; thus, it is impractical. Another approach is to increase the clock rate of the core. However, requirement (5) is not satisfied because the system has high costs and high power consumption.

The purpose of the present study is to propose an architecture for a SoC for IndCntlrs that satisfy the requirements mentioned above and to manufacture the SoC. We satisfied the requirements by implementing a RTOS in hardware. The RTOS APIs are invoked frequently during protocol processing in any type of protocol. Thus, a large amount of CPU time is consumed by RTOS execution during the protocol processing.

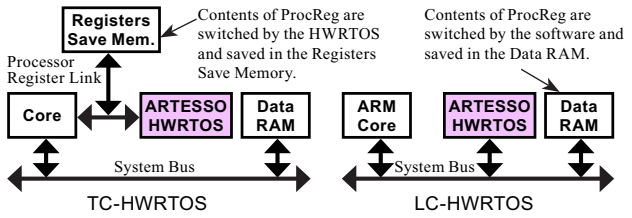


Fig. 1. **Coupling types for the HWRTOS**

Therefore, if a RTOS is implemented in hardware and the CPU time occupation by RTOS is reduced, the protocol processing load can be reduced. Thus, requirement (1) is satisfied. Requirement (2) is also satisfied because reducing the RTOS overheads also decreases the interrupt response time. This method also satisfies requirement (3) because the hardware RTOS is commonly used in many type of protocols, thus it decreases the processing load for many type of protocols. Furthermore, the method satisfies requirement (5) because it does not require an increase in the core performance. In this study, we refer to "RTOS implemented in hardware" as HWRTOS and "RTOS implemented in software" as SWRTOS.

As shown in Fig.1, there are two methods for connecting a core with a HWRTOS. One is a tightly coupled type of HWRTOS (TC-HWRTOS) and the other is a loosely coupled type of HWRTOS (LC-HWRTOS). Requirement (4) is satisfied with LC-HWRTOS, as mentioned in Section II, thus we adopted the LC-HWRTOS.

The main contributions of this study are as follows.

1. Proposal of a LC-HWRTOS architecture for IndCntlr.
2. Proposal of improvement of performance by parallel execution of a core and a HWRTOS.
3. Design and manufacture of a SoC for IndCntlrs.
4. Evaluation of the RTOS performance and network throughputs of both LC-HWRTOS and SWRTOS on the SoC.

The remainder of this paper is organized as follows. The HWRTOS architecture for IndCntlrs is described in Section II. Section III provides details of the LC-HWRTOS. Section IV explains the architecture of the SoC and the performance evaluations are presented in Section V. Related work is presented in Section VI and Section VII concludes this study. The Appendix describes our previous development system, original ARTESSO system[12]. The HWRTOS that we developed is referred to as ARTESSO HWRTOS.

II. HWRTOS ARCHITECTURE FOR INDCNTLR

This section describes why the LC-HWRTOS is adopted to satisfy requirement (4).

The following are the combinations of four cores and RTOS types for the IndCntlr SoC.

(A) HWRTOS + purpose-built core (TC-HWRTOS)
(B) HWRTOS + modified ARM core (TC-HWRTOS)
(C) HWRTOS + ARM core (LC-HWRTOS)
(D) SWRTOS + ARM core (SWRTOS)

The following explains the TC-HWRTOS and LC-HWRTOS as shown in Fig.1. In TC-HWRTOS, the core has the Processor Register Link, which is a special interface for calling an API and for switching the contents between the Register Save Memory and the internal registers of the core. The internal registers comprise a program counter, stack pointer, flag register, and general-purpose registers. In the present study, the core internal registers are referred to as ProcReg. The contents of the ProcReg are kept in the Register Save Memory on a task-by-task basis. Thus, all API functions, including context switching, are implemented in hardware. Therefore, in TC-HWRTOS, the execution time is quite fast, although a purpose-built core has to be used. As mentioned in the Appendix, the original ARTESSO system is configured as a TC-HWRTOS and it belongs to type (A). (B) is configured as a TC-HWRTOS with an ARM core, thus the ARM core needs to be modified to provide a special interface with ARTESSO HWRTOS. However, the scalability and reliability of the core are lower than formal ARM products since it is modified.

TABLE I
Different combinations of RTOSs and cores

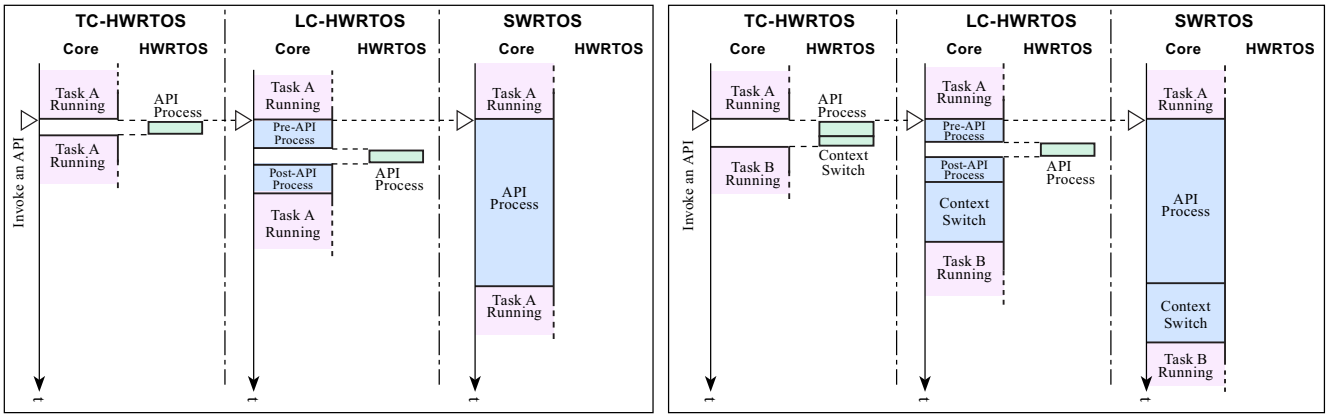
Combination Type	(A)	(B)	(C)	(D)
RTOS	TC-HWRTOS		LC-HWRTOS	SWRTOS
Core	Purpose-build	ARM (modified)	ARM	
API execution time	Low		Middle	High
Tick management offloading	Available			Un-available
IIA offloading	Available			Un-available
Core Reliability	Middle		High	
Core Scalability	No	Low	Yes	
Core verification cost	High	Middle	Low	
Standard Tools	Unusable	Usable		
LSI development cost	Middle	High	Middle	Low

In LC-HWRTOS, the HWRTOS is implemented on the system bus of the core, thus the core can invoke an API through the system bus and the core does not need to be modified with a special interface. Therefore, the API execution is quite fast, although context switching is executed by software in the same manner as SWRTOS.

Table I shows the advantages and disadvantages of the each combination. (C) is configured as a LC-HWRTOS with an ARM core. (D) is a conventional SWRTOS system.

The following describes comparison of the API execution sequence for each RTOS type. Fig.2 (a) shows the sequence without context switching. In TC-HWRTOS, after invoking an API, the HWRTOS executes the API function and the task restarts after completion. In contrast to the TC-HWRTOS, the LC-HWRTOS requires a pre-procedure to call APIs and a post-procedure to obtain the result value, because arguments and a return value of the API are handed over using registers in LC-HWRTOS, which are deployed between the core and the ARTESSO HWRTOS, as mentioned in Section III, whereas they can be handed over using ProcReg in TC-HWRTOS. Fig.2 (b) shows the API function sequence with context switching. The LC-HWRTOS requires a context-switching process based on software in addition to the pre- and post-procedures. The software used to call the RTOS, such as pre- and post-procedures, is called a RTOS driver.

To satisfy requirement (4), (B) or (C) must be selected. The execution of APIs by (B) is faster than that by (C). However, the scalability and the reliability are lower than formal ARM,



(a) API without context switch (b) API with context switch

Fig. 2. **API function sequence**

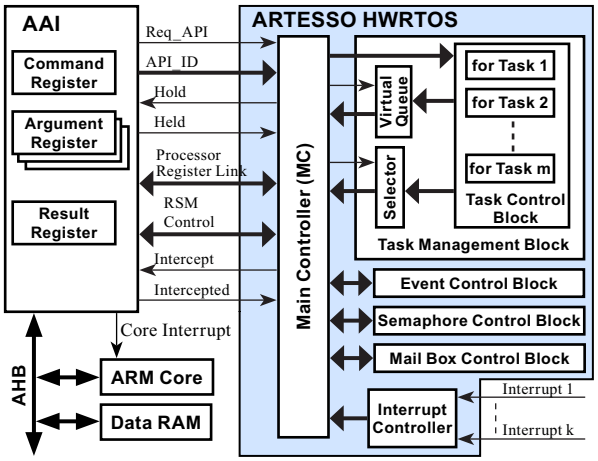


Fig. 3. **LC-HWRTOS for the IndCntlr SoC**

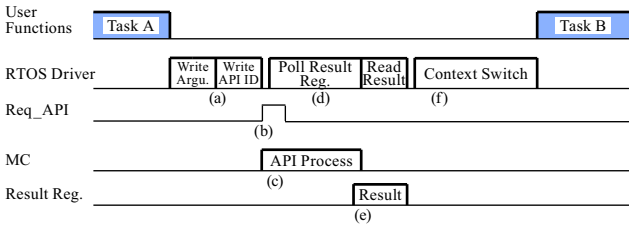


Fig. 4. **API call sequence in LC-HWRTOS**

and development cost is high in (B). The core availability and real-time response in (C) are improved compared with (D) because all of the RTOS functions are executed by hardware, except for context switching. Based on these considerations, we selected (C), the LC-HWRTOS, as the RTOS for the IndCntlr SoC.

III. LC-HWRTOS for the IndCntlr SoC

A. Architecture of the LC-HWRTOS

Fig.3 shows the architecture of the IndCntlr SoC based on the LC-HWRTOS method. The original ARTESSO system was designed based on the TC-HWRTOS, and the LC-HWRTOS of this study was modified the original ARTESSO system. The modifications are as follows. A new module, ARTESSO AHB Interface (AAI) is implemented, which permits the ARM core to access the ARTESSO HWRTOS through the AHB. AHB stands for advanced high-performance bus and is ARM system bus. The AAI includes the "Command Register," "Argument Registers," and "Result Register." The core accesses these registers through the AHB to invoke an API and to obtain the return value of the API. The Argument Registers and Result Register are also accessed from the Main Controller (MC). The MC is modified to connect with the AAI and to implement new functions, as described in subsection C.

The following modules are same as those in the original ARTESSO system. The Task Control Block maintains the management data related to all of the tasks, such as the "current state" and "task priority." The Virtual Queue implements a great deal of queues in small hardware and provides fast queue operations. The Event Control Block, the Semaphore Control Block, and the Mail Box Control Block maintain and manage the information required by event functions, semaphore functions, and mailbox functions, respectively.

B. Procedure for Calling an API

Fig.4 shows the API call sequence. The core writes the argument into the Argument Registers and writes an API identifier in the Command Register, (a). Next, the AAI sends a Req_API signal to the MC with an API_ID signal which indicates the API identifier, (b). When the signals are detected, the MC begins the execution of the API, (c). After the MC completes the execution of the API, the MC writes the return value into the Result Register, (e). The core polls the Result Register after the API call and it reads the value until a valid return value is written into the Result Register by the MC, (d).

If the API execution requires context switching, information that indicates the context switching request and the dispatched task identifier are written in the Result Register with the return value. The core then executes context switching using software, (f). Specifically, the contents of the current task in the ProcReg in the core are saved to the Data RAM and the contents of the next task are loaded into the ProcReg.

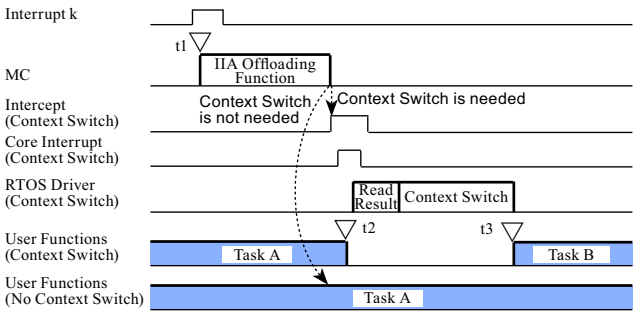


Fig. 5. **IIA sequence in POHC**

C. Parallel execution of core and ARTESSO HWRTOS

As mentioned in Appendix, the original ARTESSO HWRTOS implements not only API functions but also other RTOS functions such as interrupt invoked API (IIA) and tick management to offload their works from the core. When an interrupt occurs, the IIA function invokes a selected API, which is determined by the interrupt cause. After the IIA function the MC executes task switch if needed. Fig.11 in the Appendix shows the IIA offloading in the TC-HWRTOS and Fig.10 (1) shows that the SWRTOS executes the same procedure as IIA offloading. In the tick management function, the ARTESSO HWRTOS implements hardware timeout-timers for each task and if the MC detects a timer expiration, the MC moves the task from the wait queue to the ready queue, and then the MC executes task switch if needed.

In the SWRTOS, the user function is suspended while executing the RTOS function. Since the RTOS functions and user functions are both executed on a core and the user function has to wait for return value of called API. In the original ARTESSO HWRTOS, the user function was suspended during the MC working in the same manner as the SWRTOS. This method is called serial operation of a HWRTOS and a core (SOHC). However the IIA and the tick management functions are invoked by interrupts therefore they can be started without invocation by user functions executing on the core, thus the IIA and the tick management can execute in parallel with user functions in the HWRTOS. Proposed architecture allows parallel operation by modifying the MC. This method is called parallel operation of a HWRTOS and a core (POHC). If the RTOS decides that a task switching is not needed at the end of the IIA or tick management function, the currently executing user function continues to run, as shown in Fig.5. The POHC improves the system performance, especially in systems where interrupts are generated frequently such as network protocol processing.

IV. SoC Implementation: R-IN32M3

This section provides a summary of the R-IN32M3 SoC[15], which we developed as an IndCntlr to satisfy the requirements specified in Section I. Fig.6 shows the overall configuration of R-IN32M3. The R-IN32M3 is commercially available and the libraries for R-IN32M3, such as the RTOS driver, and protocol stacks, are available from Renesas Electronics web site.

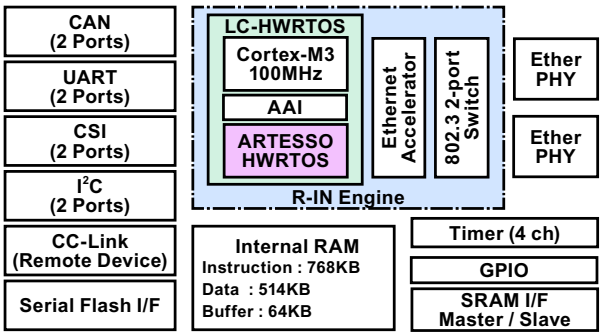


Fig. 6. **Overall configuration of R-IN32M3 SoC**

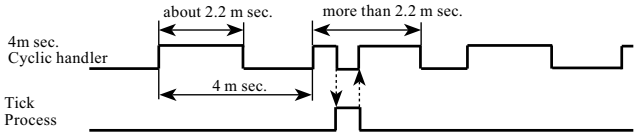


Fig. 7. **Test of the effect of the time tick**

The R-IN32M3 provides various types of communication ports, such as CAN and CC-Link, while two Ethernet ports are provided for use by industrial Ethernet systems.

The R-IN Engine is a module that implements industrial network functions. It is not a conventional simple processing unit but it provides high performance and has high added value, as follows.

1) LC-HWRTOS: The LC-HWRTOS comprises the AAI, the ARTESSO HWRTOS, and the core, as mentioned in section III. A Cortex-M3 is adopted as an ARM core.

2) 802.3 Two-port Switch: The module operates with a daisy chain configuration using a two-port PHY in industrial Ethernet. Two different hardware configurations are possible, EtherCAT/slave and CC-Link IE/Field.

3) Ethernet Accelerator: The Ethernet Accelerator has three functions for accelerating protocol processing. (i) Checksum execution for TCP and IP. (ii) A protocol header rearrangement function, which rearranges the compressed header format into a format that the core can handle easily, and vice versa. (iii) A buffer management function that comprises buffer allocation and release functions. They are also implemented by hardware logic.

V. Performance Evaluations

This section presents comparisons of the performance obtained using TC-HWRTOS, LC-HWRTOS, and SWRTOS.

A. Evaluation Items

The RTOS performance and network performance are evaluated. *1)* to *5)* present evaluations of the RTOS performance and *6)* describes the network performance, as follows.

1) API execution time: The improvement in the execution time with HWRTOS is evaluated. The API execution time is measured with and without context switching for each of the ”start task,” ”release semaphore” and ”wake up task” APIs.

2) Interrupt response: The improvement in the interrupt response with the IIA offloading is evaluated. The times are

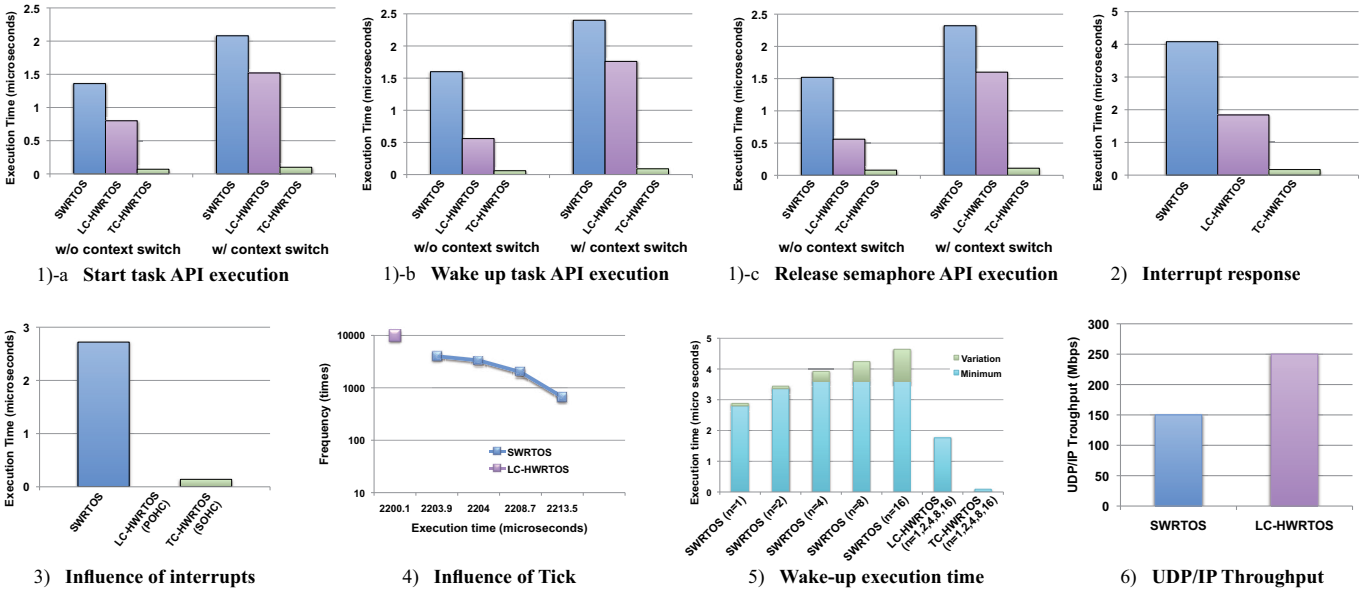


Fig. 8. **Evaluation Result**

measured from t1 to t3 in Fig.10 (1) with SWRTOS, in Fig.11 with TC-HWRTOS and in Fig.5 with LC-HWRTOS.

3) IIA overheads: The improvement in IIA offloading with the POHC is evaluated. In this case, the task is not switched as the result of the API execution, which is invoked during the IIA. The times are measured from t1 to t2 in Fig.10 (1) for SWRTOS, and in Fig.11 for TC-HWRTOS. In LC-HWRTOS, the time is zero is verified.

4) Influence of tick: The improvement in the real-time response with the tick management offloading is evaluated, i.e., the variation in the interrupt response with the tick function. Three periodic tasks are performed, where each cycle is set to 4, 10, or 15 ms. Each task process executes 20,000 loops, which execution time is about 2.2 ms. As shown in Fig.7, the execution time of the 4-ms periodic task is measured. If there are no influences on the tick, the execution time is about 2.2 ms and fixed.

5) Wake-up execution time: The improvement in the real-time performance in hardware implementation is evaluated. The execution time of the API is measured if n tasks are waiting for their timeout in the queue and one of them is woken by the wake-up task API. The measurements are executed 1 million times for each n, and the maximum and minimum times for each n are obtained.

6) UDP/IP throughput: The UDP/IP throughput is evaluated using each LC-HWRTOS and SWRTOS.

B. Experimental Setup

The performance is evaluated with SWRTOS, LC-HWRTOS, and TC-HWRTOS which are API-compatible. With SWRTOS and the LC-HWRTOS, the R-IN32M3 evaluation board is used. The operational clock of both the core and the HWRTOS are set to 100 MHz. In the SWRTOS evaluation, the HWRTOS function of the R-IN32M3 is disabled. In the TC-HWRTOS, the Verilog source code of the original ARTESSO system shown in the Appendix is used and the time is obtained by the Verilog RTL simulator.

In the network performance evaluation related to the SWRTOS and the LC-HWRTOS, the evaluation environment described above is employed and the UDP/IP protocol stack is implemented on it, where the UDP/IP throughput is measured in both environments.

C. Evaluation Results

The following section presents the evaluation results for the experiments described in the previous section.

For *1)*, Fig.8 1) a–c show the execution time results with the three APIs. Without context switching, the execution times with the LC-HWRTOS are 1.7 to 2.9 times faster compared with the SWRTOS. With context switching, however, the execution times with the LC-HWRTOS are 1.4 to 1.5 times faster compared with the SWRTOS. The results are better ”without context switching” with the LC-HWRTOS because only pre- and post-API software processing are needed in ”without” case, but context-switching software is added process in ”with” case. As mentioned in Section II, execution time in the TC-HWRTOS is the fastest.

For *2)*, Fig.8 2) shows the results of the interrupt response evaluation. The execution time with the LC-HWRTOS is 2.3 times faster compared with that with the SWRTOS.

For *3)*, Fig.8 3) shows the IIA overheads. With the LC-HWRTOS, the IIA function is offloaded from the core and context switch is not executed thus the IIA overheads are zero. The results demonstrate the system performance is improved by POHC method, because the core can keep to execute in case of a context switch is not needed, as shown in Fig.5.

For *4)*, Fig.8 4) shows the effects of the time tick management function on real-time processing. The evaluation is only executed in the SWRTOS and the LC-HWRTOS because the performance of TC-HWRTOS is almost same as that of

LC-HWRTOS. The horizontal axis indicates the execution time required for the 4-ms periodic task. The execution time required for the 4-ms periodic task in the SWRTOS varies according to the periodic interrupt and the maximum range of variation is 9.6 ms. By contrast, the execution time in the LC-HWRTOS is always 2.2001 ms. These results show that real-time processing is delayed by a maximum of 9.6 ms by tick management function with the SWRTOS, whereas the process is not delayed with the LC-HWRTOS. Therefore, the LC-HWRTOS is advantageous during real-time processing because a task that is invoked by an interrupt can start at a precise time.

For *5)*, Fig.8 5) shows the execution time of the wake up task when some tasks are waiting for timeouts. With the SWRTOS, the maximum execution time and its variation increase according to n. When n is 16, the variation is 1.04 μs. By contrast, the execution time is always fixed regardless of n with the LC-HWRTOS. The execution time with the LC-HWRTOS is 2.6 times faster compared with the SWRTOS when n is 16. These results show that the wake up task execution time varies according to the internal conditions in the SWRTOS, whereas the time does not depend on the internal conditions in the LC-HWRTOS. Therefore, the LC-HWRTOS is advantageous for real-time processing because a task can wake up at a precise time in the LC-HWRTOS.

For *6)*, Fig.8 6) shows the UDP/IP throughput results where both platforms are exactly the same, except for the RTOS. The results indicate that the performance is improved by 1.67 times only when the SWRTOS is replaced by the LC-HWRTOS. The results also show that the network load decreased by 40% with the LC-HWRTOS, thus LC-HWRTOS would be effective in industrial network systems.

VI. Related Work

Various techniques have been proposed for improving the performance of RTOSs, some of which implement the RTOS functions partially in hardware [6-11]. Others implement all of the functions of the RTOS in hardware [1-5]. Previous studies have shown that the performance of the HWRTOS was several times faster than that of the SWRTOS. Some of them adopt ARM core, however, they implemented non-standardized and limited number of APIs, and they only provided several or several tens of queues because they could not implement a large number of queues in small volume hardware, therefore insufficient objects could be provided. Consequently, their methods did not satisfy the requirements of industrial network systems. Furthermore, they have not been commercially available as a SoC. By contrast, the proposed architecture provides 41 ITRON[13] standard APIs, which are sufficient for utilization in the IndCntlrs. ITRON is a RTOS standard and widely used in Japan. The proposed architecture also provides several thousand queues at low cost using the novel Virtual Queue technology, as described in the Appendix. Therefore, the proposed architecture satisfies the requirements of industrial network systems. The SoC based on the proposed architecture is the world's first commercial product which implements ARM and RTOS in hardware.

VII. Conclusion

Recently, faster network systems have been deployed for industrial networks using Ethernet. However, the increased protocol processing load affects real-time processes such as motor control. Thus, we developed R-IN32M3 SoC to overcome this problem, which can be used in industrial network systems. The requirements of industrial network systems are: (1) reducing the load of network protocol processing, (2) improved real-time capability, (3) multi-protocol support, (4) adoption of ARM cores, and (5) low costs and low power consumption. To satisfy these requirements, Corex-M3 was adopted as the core and the LC-HWRTOS was adopted as the RTOS. Our evaluations showed that the LC-HWRTOS operated faster than the SWRTOS. Our experimental results showed that the UDP/IP throughput was increased by 1.67 times by replacing the SWRTOS with the LC-HWRTOS and the network load decreased by 40%.

References

[1] Lindh L., "Fastchart – a fast time deterministic CPU and hardware based real-time kernel," in Proc. of Euromicro Workshop on Real Time Systems, pp. 36–40, Jun, 1991

[2] Adomat J., Furunas J., Lindh L., Starner J., "Real-time kernel in hardware RTU: a step towards deterministic and high-performance real-time systems," in Proc. of the 8th Euromicro Workshop, pp. 164–168, Jun 1996

[3] Nordstrom S., Lindh L., Johansson L., Skoglund T., "Application specific real-time microkernel in hardware," in Proc. of Real Time Conference, 2005.

[4] Samuelsson T., Åkerholm M., Nygren P., Johan Stärner J., Lindh L., "Comparison of multiprocessor real-time operating systems implemented in hardware and software," in Proc. of Int'l Workshop on Advanced Real-Time Operating System Services (ARTOSS'03), 2003.

[5] Nakano T., Utama A., Itabashi M., Shiomi A., Imai M., "Hardware implementation of a real-time operating system," in Proc. of 12th TRON Project International Symposium (TORN'95), pp. 34044, 1995.

[6] Kohout P., Ganesh B., Jacob B., "Hardware support for real-time operating systems," in Proc. of the 1st International Conference on Hardware/Software Codesign and System Synthesis, pp. 45–51, Oct. 2003

[7] Chandra S., Regazzoni F., Lajolo M., "Hardware/software partitioning of operating systems: a behavioral synthesis approach," in Proc. of the 16th ACM Great Lakes Symposium on VLSI, pp. 324–329, 2006

[8] Parisoto A., Souza A. Jr, Carro L., Pontremoli M., Pereira C., Suzim A., "F-Timer: dedicated FPGA to real-time systems design support," in Proc. of 9th Euromicro Workshop on Real-Time Systems, pp. 35–40, 1997

[9] Mooney III V., Lee J., Daleby A., Ingstrom K., Klevin T., Lindth L., "A comparison of the RTU hardware RTOS with a hardware/software RTOS," in Proc. of Design Automation Conference, 2003, pp. 683–688.

[10] Mooney III V.J., Blough D.M., "A hardware-software real-time operating system framework for SoCs," IEEE Design & Test of Computers, 44–51, 2002.

[11] Mooney III. V., "Hardware/software partitioning of operating systems," in Proc. of Design Automation and Test in Europe Conference (DATE'03), 2003, pp. 338–339.

[12] Maruyama N., Ishihara T, Yasuura H., "An RTOS in hardware for energy efficient software-based TCP/IP," Proc. of IEEE Symposium on Application Specific Processors (SASP), 2010, pp. 13–18.

[13] TRON ASSOCIATION, "μITRON4.0 Specification," 1999.

[14] Felser M., "Real-time Ethernet – industry prospective," Proceedings of the IEEE, Volume 93, Issue 6, June 2005, pp. 1118–1129.

[15] Renesas Electronics, "R-IN32M3-Series Data Sheet," Dec 9, 2013.

APPENDIX

This appendix describes the original ARTESSO system [12], which we developed using the ARTESSO HWRTOS, as shown in Fig.9.

A. Summary of the original ARTESSO system

The original ARTESSO system comprises an ARTESSO HWRTOS, a Register Save Memory, and an ARTESSO core, as shown in Fig.9. They are configured as a TC-HWRTOS. The ARTESSO HWRTOS conforms with ITRON specifications [13] and it supports 41 ITRON APIs. The supported APIs are listed in Table II. The Register Save Memory is used to maintain the contents of the ProcReg on a task-by-task basis. The ARTESSO Core is a proprietary 32-bit RISC processor, which has a special interface that connects with the ARTESSO HWRTOS in a tightly coupled manner.

B. Features of the original ARTESSO system

The following are the features of the original ARTESSO system.

1. Configuration of TC-HWRTOS.
2. Several thousand queues in hardware at a low cost based on an innovative idea called Virtual Queue.
3. An interrupt invoked API (IIA) offloading function that executes interrupt processing in hardware, which improves the performance of the interrupt response.
4. A tick management offloading function that removes the software tick process, which also improves the interrupt response.

1) TC-HWRTOS: Fig.9 shows the architecture of the original ARTESSO system, which is configured as a TC-HWRTOS. The Main Controller (MC) is implemented by a hardware state machine that executes all of the API call processes. The Task Control Block maintains the management information related to all of the tasks, such as the "current state" and "task priority." The Virtual Queue module implements all of the queues used by the RTOS. The Event Control Block, the Semaphore Control Block, and the Mail Box Control Block maintain and manage the information required by event functions, semaphore functions, and mailbox functions, respectively.

Next, we describe the API call procedure. The arguments and return values of the APIs are handed over using some of the general purpose registers in the ProcReg. When the core decodes an API call assembler instruction, the core changes the Req_API signal to 1 and indicates an API identifier by sending the API_ID signal to the MC. If the MC detects this signal, it changes the hold signal to 1 and commences API processing according to the API_ID signal. If the API has arguments, the MC refers to the specific registers in the ProcReg that are assigned to the arguments. When the MC completes the execution of the API process, it writes the return value in the specific register of the ProcReg that is assigned to the return value and it then changes the hold signal from 1 to 0, thereby indicating the completion of the process to the core.

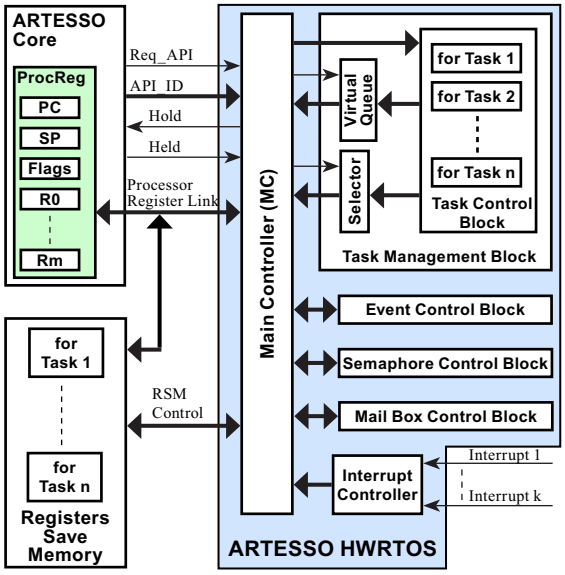


Fig. 9. **Original ARTESSO system (TC-HWRTOS)**

TABLE II
Supported APIs

task	start, exit, terminate, change priority, get task ID, sleep, wakeup, release wait
flag	create, delete, wait, set, clear, poll
semaphore	create, delete, wait, release
mailbox	create, delete, send, receive
cpu	lock, unlock
dispatch	disable, enable
ready queue	rotate
system timer	set, get

If the core recognizes the hold signal transition, it restarts and fetches from the next program counter.

If a context switch is needed after the completion of API execution, the MC saves the contents of the ProcReg in the Register Save Memory and loads the next task contents into the ProcReg from the Register Save Memory. Next, the MC changes the Hold signal from 1 to 0, which indicates completion to the core and the core then restarts the fetch process again.

In the interrupt procedure, interrupt signals enter the Interrupt Controller in the ARTESSO HWRTOS. If the MC detects an interrupt through the Interrupt Controller, the MC changes the Hold signal to 1 to stop the core. When the core stops, the Held signal is changed to 1. If the MC detects the Held signal transition, it saves the contents of the ProcReg to the Register Save Memory and loads the ISR contents from the Registers Save Memory into the ProcReg. Next, the MC changes the Hold signal from 1 to 0 to indicate the completion of the context switch to the core and the then core restarts the fetch process according to the ISR register set.

As mentioned above, the original ARTESSO system implements the TC-HWRTOS using a special interface with a procedure between the ARTESSO HWRTOS and the ARTESSO

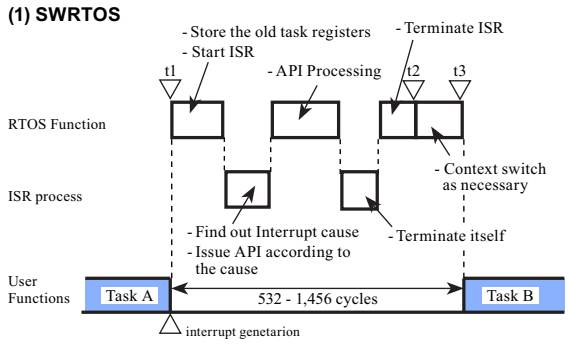


Fig. 10. **ISR and IIA operations**

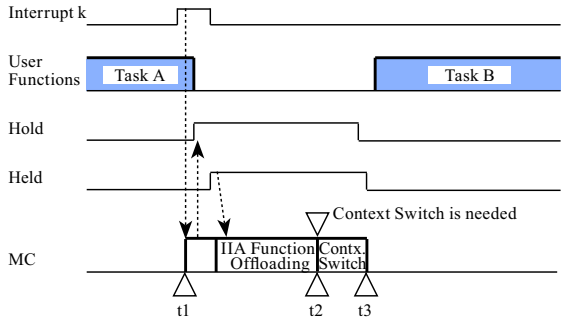


Fig. 11. **IIA offload execution**

core.

2) Virtual Queue: In general, RTOSs require many queues, i.e., several thousand queues, since they are needed for each instance of an object. To implement an RTOS in hardware, the queues should also be implemented in hardware. However, it is very expensive to implement many queues in hardware using conventional technology such as hardware FIFOs. Thus, we propose the innovative idea of a Virtual Queue, which is an information compression technique with reversibility that produces a large number of queues with a small hardware volume.

3) IIA Offloading: The IIA offloading function is implemented in addition to the conventional interrupt service routine (ISR) in the ARTESSO HWRTOS. The IIA offloading function invokes a selected API based on the cause of the interrupt. Fig.10 (1) shows that the SWRTOS executes the same procedure as IIA offloading. The following description provides specific details of the procedure. When an interrupt is generated, the SWRTOS starts and it saves the current task register contents in the ProcReg, before loading the ISR register contents into the ProcReg and starting the ISR. The ISR determines the cause of the interrupt and invokes an API based on the cause. After the completion of API execution, the SWRTOS terminates the ISR and the SWRTOS then replaces the ISR contents in the ProcReg with the new task register contents.

IIA offloading can replace the software ISR process, as mentioned above, therefore all of the interrupt processes can be implemented in hardware. Thus, the interrupt response is improved dramatically, as shown in Fig.10 (2) and Fig.11.

4) Tick management Offloading: The SWRTOS uses a tick process to implement timeout processing for tasks. The tick process is woken up by a periodic interrupt and it decrements each timeout counter. If it detects a timeout, it removes the task from the wait queue and appends it to the ready queue. As mentioned above, the tick process is a critical process and interrupts are inhibited during the process, which results in fluctuations in the interrupt latency. By contrast, the ARTESSO HWRTOS implements hardware timeout counters on a task-by-task basis and the counter value is decremented by its hardware. If the counter reaches zero, the MC is started, which removes the task from the wait queue and appends the task to the ready queue. Next, the MC executes context switching if necessary. Thus, the ARTESSO HWRTOS does not require the tick process in software. This results in a drastic reduction in fluctuations in the interrupt latency.

TABLE III
RTOS execution time

System Call	Dispatch	SWRTOS	ARTESSO HWRTOS (TC-HWRTOS)
Sleep Task	Yes	628	10
Wakeup Task	Yes	496	10
Change Priority	Yes	541	11
Receive from Mailbox	No	224	7
Receive from Mailbox	Yes	591	11
Send to Mailbox	No	360	8
Send to Mailbox	Yes	541	11
Wait Semaphore	No	216	6
Wait Semaphore	Yes	558	9
Release Semaphore	No	344	7
Release Semaphore	Yes	536	11

Unit : Cycles

C. Performance of the original ARTESSO system

Table III compares the performance of commercial SWRTOS and the ARTESSO HWRTOS, which has the TC-HWRTOS configuration. The RTOS performance of the original ARTESSO system was several tens times faster than that of SWRTOS. The interrupt response using IIA offloading was 38 to 104 times faster than that of the SWRTOS, as shown in Fig.10.