An Adaptive Approach for Implementing RTOS in Hardware

Tetsuo Miyauchi[1,a)] Kiyofumi Tanaka[1,b)]

Abstract: In recent years, along with a growth of IoT (Internet of Things), many embedded devices are equipped with processors/controllers, where a real-time OS (RTOS) is accommodated to make full use of complicated functions of the devices. It is desired that RTOS runs fast with as small memory usage as possible since it is overhead from an application program's viewpoint. Therefore, it is expected that providing hardware for a part of RTOS processing reduces memory usage while it makes the processing fast. Under the circumstances where several examples of hardware implementations of RTOS are found, we implement functions of the μITRON[12] specification in FPGA hardware. In addition, we propose an approach to adapting it to applications' requirement.

Keywords: RTOS, adaptive approach, μITRON, FPGA

1. Introduction

Along with the popularization of IoT (Internet of Things), micro processors are more than ever being embedded in lots of appliances, which have complicated functions with communication. In order to implement micro processors in various things, cost is one of the most important factors. For reducing the cost, it is desirable that processing resources which software and hardware use should be reduced as much as possible while functions to be provided and performance are maintained.

From the viewpoint of embedded system development, RTOS (Real-Time Operating System) is commonly used to make developing a system with strict time constraint more efficient. As using RTOS makes it possible to divide an application software into multiple tasks and develop each task separately, software module independency increases and parallel development and verification of tasks can be easily performed. Additionally, hardware abstraction, synchronization and communication functions through kernel objects, and real time scheduling can be utilized. That makes it possible to expedite a system development in a short period.

Nevertheless, as an RTOS kernel itself is just overhead for an application program, the smaller footprints of an RTOS kernel mean the better implementation and it is desirable that execution time is short enough.

μITRON4.0 specification is one of the most commonly used RTOS specifications [12]. As an RTOS kernel which follows μITRON4.0 specification is in most cases provided as a library, only specified system calls which are actually used in an application program are linked with the application program, which means that processor memory space for a program code is not wasted with object codes of unused system calls. Still, lots of software resources are used for typical operations such as queue operation and task control block operation, which are commonly used in various RTOS kernel functions. Essentially, these operations should use less memory space and the execution time should be as fast as possible.

It is expected that Implementing RTOS functions in hardware can make it possible to reduce software code size and shorten system call execution time. In our approach, RTOS kernel functions for μITRON4.0 specification are implemented in an FPGA. Compared with a full software RTOS, we aim at reducing software code size and execution time.

While there are several studies for implementing RTOS functions in hardware as described in the next section, characteristics of our approach are: building RTOS functions with error checking in FPGA hardware, removing error checking functions in RTOS system calls if possible and deleting hardware functions for unused system calls. We show that this approach can be implemented in an FPGA and evaluate the number of hardware resources used in an FPGA, software code sizes, and execution time for system calls, when the system is adapted to an application program so that unused functions are eliminated.

The organization of this paper is as follows. In the next section, related works for hardware implementations of RTOS are shown. In Section 3, we describe the features of the hardware structure and software structure of our implementation. Section 4 shows results of evaluation in terms of the number of hardware resources, software sizes and execution time of system calls, and discuss the results. Finally, we summarize this paper and issues for the next step in Section 5.

[1] Japan Advanced Institute of Science and Technology, Asahidai 1-1, Nomi, Ishikawa, 923–1292, Japan
[a)] t-miyauc@jaist.ac.jp
[b)] kiyofumi@jaist.ac.jp

2. Related Work

As cost reduction is an important issue for embedded systems, it is desirable that the system code size including RTOS and used hardware resources should be as small as possible to lower the total cost. Implementing RTOS as hardware to reduce the amount of the software code size and improve the execution efficiency has been studied in several decades.

The literature [10] and [11] are studies for implementing an RTOS based on μITRON specification in hardware, in which a task scheduler and system calls such as semaphore and eventflag are implemented.

In the literature [4], a dedicated processor and hardware system, ARTESSO (Advanced Real Time Embedded SiliconSystem Operator), which provides the same RTOS functions as those in a widely used software RTOS kernel, is described. In the literature [6], in addition, the processor core of this system is modified to an ARM core. Moreover, in the literature [5], this system is enhanced to a multiprocessor system.

Similarly, in the literature [3], a system in which general purpose RTOS functions with API interfaces and a dedicated CISC processor are implemented in an FPGA is proposed.

A method of constructing processor functions adapted to an application program, called ASIP (Application-specific instruction-set processor), is described in the literature [1] and [2].

While studies for implementing RTOS functions in hardware, such as the ones above, have been conducted for several years, we have been studying to reduce runtime and resource overhead by adapting RTOS kernel functions and processor functions to an application program (in the literature [7] and [8]). As part of these researches, in this paper, we demonstrate possibility of adapting RTOS hardware to an application program for reducing code size of software RTOS kernel and improving efficiency in executing system calls. The proposed methods are evaluated in terms of hardware resources used, the maximum frequency, and execution time. This evaluation is performed by using an actual FPGA board.

Since the method of implementing primitive RTOS kernel operations is described in [9] in detail, we omit the way of implementing them in this paper. We focus on how to implement the RTOS kernel functions with error checking in hardware and how to remove the error checking parts as a result of analyzing application source codes. We also discuss the effect of reducing unused RTOS kernel functions including error checking.

3. Implementation

3.1 Hardware Structure

Hardware structure which we have implemented is explained below. We have implemented principal functions of the μITRON[12] standard profile specification. Fig 1 shows

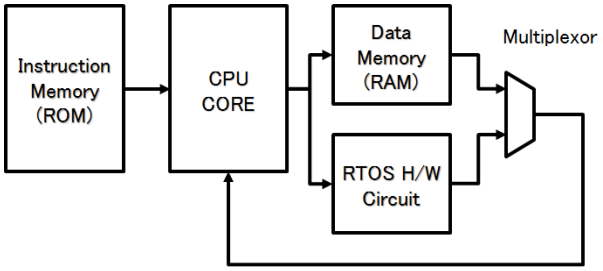

Fig. 1 Processor structure with RTOS hardware.

a structure of a processor core and RTOS hardware circuit we implemented. In the literature [9], we described effectiveness of our implementation of RTOS primitive functions such as RTOS queue operations. Fig 1 is cited from the literature [9].

RTOS hardware proposed in this paper consists of not only primitive functions but also all functions including error checking in system calls. Therefore, compared with software-only RTOS implementation, execution time of an RTOS system call can be reduced and the software code size can be decreased.

Fig 2 shows an RTOS hardware structure. In this figure, the part of the "RTOS Hardware Core", which is described in detail in [9], is the fundamental functions related to TCB (Task Control Block) queue operations. An RTOS hardware operation command and data are delivered to the "RTOS Hardware Wrapper" part by an software program in a processor. RTOS hardware operation command is designated with a memory address of a memory reference instruction and an RTOS hardware operation is decided with a pair of an address and data.

Interfaces between RTOS Hardware Wrapper and RTOS Hardware Core, which is shown in Fig 2, are explained as follows. RTOS Hardware Core has input signals and output signals as shown in Table 1. CLK is the input clock signal from the system. In this implementation, as CPU operates at 50MHz, RTOS Hardware Core works with the same clock cycle. RST indicates the reset signal from the system. When RST signal is high, register values in RTOS Hardware Core are initialized. Addr is an input command for RTOS Hardware Core. Data is an input data for the corresponding input command. When these signals are delivered from RTOS Hardware Wrapper to RTOS Hardware Core, RTOS Hardware Core manipulates resources in RTOS Hardware Core and the results are presented via the output port, HighestTask. The main information in this output signal is the highest task id which is generated after manipulating RTOS resources.

Operations of RTOS Hardware Core are described in Table 2. RTOS Hardware Wrapper issues these operations to RTOS Hardware Core with several data (if any).

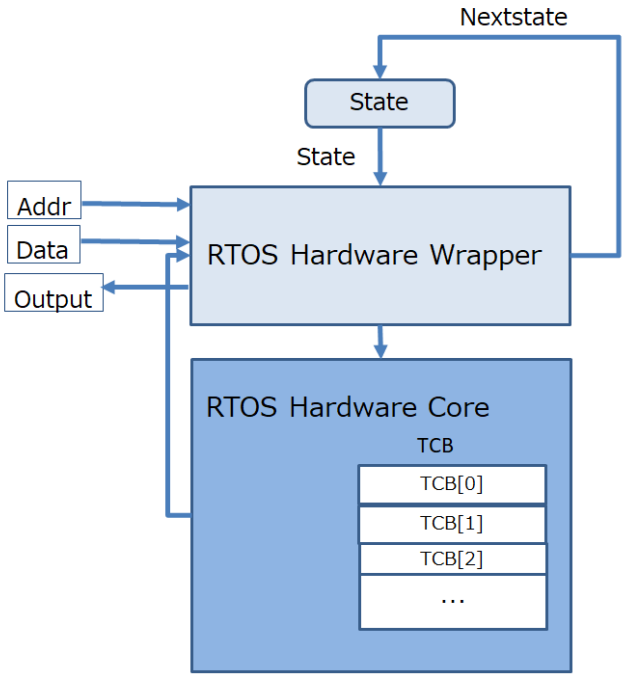

Fig. 2 RTOS Hardware Structure.

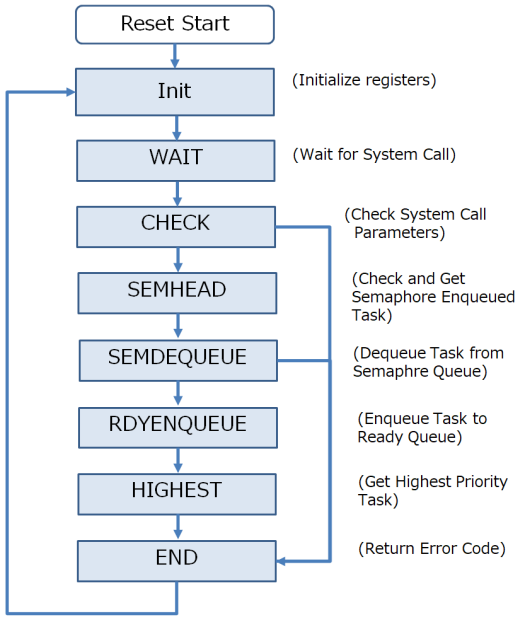

Fig. 3 Example of State Transition (sig_sem).

Table 3 shows output data for each operation. To refer to these output data, several data such as task id are input with respect to the operation in advance.

As the RTOS Hardware Wrapper part works with a state machine, the next state is decided by the current status of the RTOS hardware. For example, sig_sem() system call runs with state transition as Fig 3.

RTOS hardware waits for an issue of a system call in WAIT state. Software sets RTOS system call parameters and issues a system call to RTOS hardware. Issue of an RTOS system call is performed with a write access to a memory address which is assigned to issue of an RTOS hardware system call.

When the RTOS hardware detects write access to this address, it transits to CHECK state, which is the next state. In CHECK state, system call parameters are checked for validity. When a parameter error is found, a proper error code is set and the hardware transits to the END state.

In END state, an error code is passed to a processor core as an output from the RTOS hardware. Software can get an error code from the RTOS hardware by reading the error code passed from the RTOS hardware.

When the parameter error checking is cleared with no errors, in the case of sig_sem() system call, whether there is a task queued in the semaphore waiting queue is checked in SEMHEAD state, then the state transits to SEMDEQUEUE state. In case that any task is not queued in the semaphore waiting queue, if the semaphore count exceeds the maximum semaphore count, E_QVOR is set to the error code and the

Table 1 Inputs and Outputs of RTOS Hardware Core.

I/O	Signal	Size(bits)	Description
In	CLK	1	Clock
In	RST	1	Reset Signal
In	Addr	32	Command for RTOS Hardware Core
In	Data	32	Data for the command
Out	HighestTask	32	Output from RTOS Hardware Core (Mainly used for getting the highest task id)

Table 2 Operations for RTOS Hardware Core (input).

Operation	Input data
Connect a task to ready queue	Task id, Task priority
Remove a task from ready queue	Task id, Task priority
Change a task priority	Task id, Task priority
Refer to a task state	Task id
Refer to the first waiting task for a semaphore	Semaphore id
Connect a task to a semaphore waiting queue	Task id, Semaphore id
Remove a task from a semaphore waiting queue	Task id, Task priority, Semaphore id
Refer to the first waiting task for an eventflag	Eventflag id
Connect a task to an eventflag	Task id, Task priority
Remove a task from a eventflag	Eventflag id
Remove a task from any waiting queue	Task id

Table 3 Operations for RTOS Hardware Core (output).

Operation	Output data
Refer to the first priority task	Task id
Refer to a task state	Task status
Refer to the first waiting task for a semaphore	Task id
Refer to the first waiting task for an eventflag	Task id

hardware transits to END state. On the other hand, if the semaphore count does not exceed the limit, the semaphore count is increased and the hardware transits to END state. In case that a task is found in the semaphore waiting queue, the corresponding task is deleted from the semaphore waiting queue and the state transits to RDYENQUEUE state. In RDYENQUEUE state, a command to RTOS Hardware Core is issued and the TCB of a designated task is registered to a ready queue with RTOS Hardware Core.

Registration of a TCB is finished in one clock cycle in the queue structure of RTOS Hardware Core. Detail of the registration mechanism is described in the literature [9].

After that, the state of RTOS hardware transits to the next state, HIGHEST. In HIGHEST state, a task ID of the highest-priority task queued in the ready queue is acquired.

Finally, the state of RTOS hardware transits to END state. When a system call returns without error, E_OK is passed to a processor as an output from RTOS hardware. With using this mechanism, software detects the normal return from RTOS hardware.

3.2 Software Structure

Table 4 shows interfaces between hardware and software. The software reads from or writes to the addresses in the table. "R" in the column "R/W" indicates that a value read from the corresponding address is a return value from the hardware. On the other hand, the addresses for "W" are written to so that the system call number and other parameter values are delivered to the hardware.

Table 4 Addresses for RTOS systemcalls.

Address	R/W	Operation
0xffff0008	R	Read RTOS return code
0xffff0100	W	Issue RTOS systemcall
0xffff0104	W	Set RTOS systemcall 1st parameter
0xffff0108	W	Set RTOS systemcall 2nd parameter
0xffff010c	W	Set RTOS systemcall 3rd parameter
0xffff0110	W	Set RTOS systemcall 4th parameter
0xffff0114	W	Set RTOS systemcall 5th parameter
0xffff0120	R	Read RTOS return parameter

Before the software issues a system call, it writes the parameter values to the same number of addresses (starting at 0xffff0104) as the number defined for the system call. After all the parameters are set, the software invokes the system call.

A system call is issued by writing the system call number to the corresponding address (0xffff0100). This makes the system call start by changing the state of the hardware. Then, the software reads from the address for a return code (0xffff0008) so that it checks completion of the processing and receives a task number of the highest priority task and a return value from the system call. That is, the most significant bit of the read value indicates the completion of the RTOS hardware, and the lower bytes contain a highest-priority task number and a return code. This is a busy-waiting procedure where, after the software writes the system call number to the address for "Issue RTOS systemcall" (0xffff0100), it repeatedly reads from the address for "Read RTOS return code" (0xffff0008) until it finds the most significant bit of 1. Then, it gets the lower bytes as a return code, and proceeds to the following processing.

Some system calls return not only a return code but the other results through call by reference. For example, wai_flg() returns a flag pattern through an address which a parameter specifies. In this case, the result is obtained by reading from the address dedicated to call by reference (0xffff0120).

Figure 4 is a flow in the software-side processing (wrapper function) for act_tsk(). Other system call functions follow a similar flow.

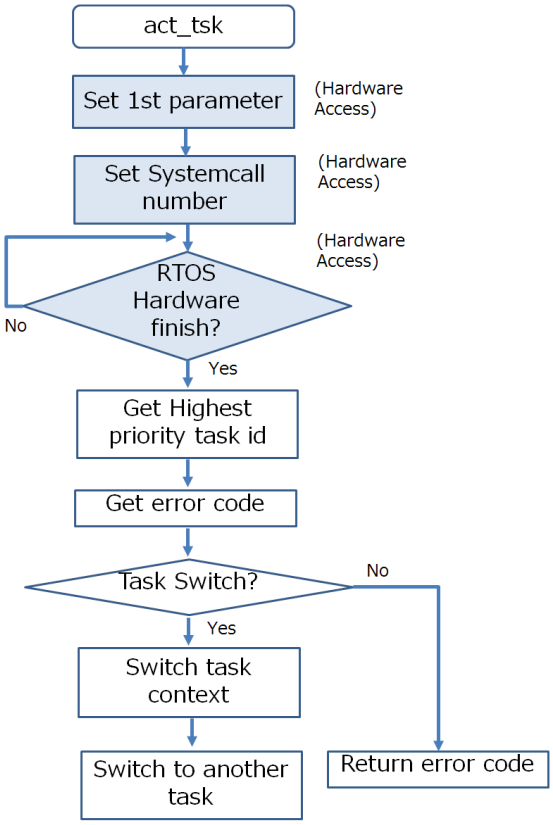

Fig. 4 Systemcall Software flow.

Figure 5 shows a part of sig_sem() system call, which is a software-side wrapper function. In this code, a parameter is set at line 1 through the dedicated address defined as "RTOSPARAM1", and the hardware processing starts at line 2 by setting the system call number ("CODE_SIG_SEM") to the corresponding address defined as "RTOSSYSCALL". Lines 5 to 6 perform busy waiting for completion of the RTOS hardware, where the most significant bit of the return code is checked. Then, the task

number of the highest-priority task (“highesttask”) and a return code (“errorcode”) are extracted from the read value at lines 8 and 9, respectively. (Bits 23 to 16 contain the highest-priority task number and bits 7 to 0 are for a return code.) The system-call processing finishes by returning the return code, after task switching, if necessary (from line 11 to 26).

sig_sem (excerpt)

```
 1:  RTOSPARAM1 = semid;
 2:  RTOSSYSCALL = CODE_SIG_SEM;
 3:
 4:  /* When RTOS HW finish, bit:31 is on */
 5:  while (((rtosreturn = PRIHIGHEST)
 6:             & 0x80000000) == 0);
 7:
 8:  highesttask = ((rtosreturn >> 16) & 0xff);
 9:  errorcode = (rtosreturn & 0xff);
10:
11:  if (errorcode == 0) {
12:
13:    if (highesttask == 0) {
14:
15:      /* No task switch */
16:      return(E_OK);
17:    } else {
18:
19:      /* Task switch */
20:      RunTask(highesttask);
21:      return(E_OK);
22:    }
23:  } else {
24:
25:      /* Error case */
26:    return((ER)errorcode);
27:  }
```

Fig. 5 Example of system call wrapper (sig_sem).

4. Evaluation

The processor core and RTOS hardware described in Section 3 are implemented in an FPGA. We used an FPGA device of Xilinx Spartan-6 (XC6SLX16CSG324C) [14] and the evaluation board of Digilent NEXYS3 [15]. The processor core runs at 50MHz in the FPGA and executes MIPS instruction set [13]. As described in Section 3.2, the RTOS hardware is accessed and controlled by reading from/writing to the specified memory addresses.

We used TOPPERS Kernel Test Suites [16] for application programs, and confirmed the behavior of the software implementation and hardware implementation of the RTOS kernel.

4.1 Results

Table 5 shows the number of resources occupied by the processor core and RTOS hardware and minimum period/maximum clock frequency for the configuration with error checking including five tasks, four semaphores, and three eventflags. These numbers of resources are reported by PlanAhead, the Xilinx development tool, and the values shown in the column of “Usage (%) to All Resources” are the rate of used resources with respect to all resources in the FPGA (Xilinx Spartan-6 XC6SLX16CSG324C), which is used for this implementation. “Minimum period” is the minimum period of the clock signal of this implementation in the FPGA and “Maximum frequency” is the value of the corresponding clock frequency when the clock period is the minimum one.

Table 5 FPGA Resources (Full)

Resources	# of Used Resources	Usage (%) to All Resources
Register	2076	11%
LUT	4699	51%
Slice	1444	63%
Minimum period	19.972ns	
Maximum frequency	50.07MHz	

In software implementation, an RTOS kernel for μITRON is provided in the library form, and therefore only system calls which are actually used are linked to the application binary. Similarly, our RTOS hardware implementation makes it possible to select functions to be implemented in hardware on a function basis such as semaphore and eventflag.

Table 6 shows the number of resources occupied and minimum period/maximum clock frequency for the configuration with error checking including five tasks, four semaphores, and no eventflags.

Table 6 FPGA Resources (Semaphore w/ Error Check)

Resources	# of Used Resources	Usage (%) to All Resources
Register	1431	7%
LUT	3601	39%
Slice	1111	48%
Minimum period	18.583ns	
Maximum frequency	53.813MHz	

When static analysis of application program codes guarantees that some errors never occur, the hardware mechanisms for the corresponding code fragments of dynamic error checking can be eliminated[*1]. In Table 7, the resource usage and minimum period/maximum frequency for the configuration without error checking, with five tasks, four semaphore, and no eventflags.

Next, excluding semaphores, the results for the configuration including five tasks and three eventflags with error checking is shown in Table 8

[*1] The analysis for error-checking in the software implementation is described in detail in [8].

Table 7 FPGA Resources (Semaphore w/o Error Check)

Resources	# of Used Resources	Usage (%) to All Resources
Register	1405	7%
LUT	3538	38%
Slice	1085	47%
Minimum period	17.655ns	
Maximum frequency	56.641MHz	

Table 8 FPGA Resources (Eventflag w/ Error Check)

Resources	# of Used Resources	Usage (%) to All Resources
Register	1969	10%
LUT	4430	48%
Slice	1369	60%
Minimum period	19.915ns	
Maximum frequency	50.213MHz	

Then, excluding semaphores, the results for the configuration including five tasks and three eventflags without error checking is shown in Table 9

Table 9 FPGA Resources (Eventflag w/o Error Check)

Resources	# of Used Resources	Usage (%) to All Resources
Register	1967	10%
LUT	4330	47%
Slice	1369	60%
Minimum period	19.956ns	
Maximum frequency	50.11MHz	

For reference, Table 10 shows the resource usage and minimum period/maximum frequency for only a processor core (without the RTOS hardware).

Table 10 FPGA Resources (w/o RTOS Hardware)

Resources	# of Used Resources	Usage (%) to All Resources
Register	770	4%
LUT	1529	16%
Slice	497	21%
Minimum period	16.6ns	
Maximum frequency	60.241MHz	

Table 11 shows the sizes of binary codes of system calls and common routines. “Soft Only” means the software-implemented RTOS kernel. “With Hard” means the proposed implementation where the main processing for the RTOS kernel is preformed by the hardware. As shown in the code for sig_sem system call in Section 3.2, since the main processing is covered (or hidden) by the RTOS hardware, the size of the software-side system call (wrapper) is reduced. From the table, it is confirmed that the hardware implementation reduces the code sizes of all system calls and common functions.

Table 11 RTOS Kernel Software Size (bytes)

Systemcall	Soft Only	With Hard	Hard/Soft
act_tsk	416	256	61.5%
chg_pri	1008	272	27.0%
ter_tsk	1248	640	51.3 %
rel_wai	720	288	40.0%
sig_sem	528	304	57.6%
wai_sem	672	336	50.0%
pol_sem	304	256	84.2%
set_flg	704	368	52.3%
wai_flg	864	464	53.7%
pol_flg	432	336	77.8%
Soft Kernel	1280	0	0%
Common Routine	1184	1152	97.3%

Execution time for each system call is shown in Table 12. For measurement of execution times, the processor core is equipped with a hardware counter which increases by one every clock cycle. Execution time is obtained by reading the values of this counter at the entry and the exit points of a system call and subtracting between them. Considering the processor core’s running clock frequency of 50MHz, the obtained execution time in cycles is converted to that in microseconds by multiplying it by the clock cycle time (period) of 20 nanoseconds. Since execution of system calls can involve task switching, the table includes execution times in both cases with task switching and without it.

In Table 12, “×” in the column “Task switch” indicates that the system call is executed and completed without task switching. On the other hand, “○” corresponds to the situation where the issue of the system call leads to task switching. In this case, execution time is the time from the issue of the system call to the (re-)start of a task after task (context) switching. From the table, it is confirmed that use of the RTOS kernel hardware shortens execution times of system calls compared to the software implementation.

Table 12 Execution Time for System Calls

Syscall	Task switch	Soft		Hard	
		clock	time @50MHz (μsec)	clock	time @50MHz (μsec)
sig_sem	×	115	2.3	98	2.0
sig_sem	○	323	6.5	194	3.9
wai_sem	×	88	1.8	88	1.8
wai_sem	○	351	7.0	184	3.7
pol_sem	×	88	1.8	76	1.5
set_flg	×	122	2.4	107	2.1
set_flg	○	409	8.2	208	4.2
wai_flg	×	119	2.4	117	2.3
wai_flg	○	375	7.5	205	4.1
pol_flg	×	119	2.4	105	2.1
Average	w/o switch	108.5	2.2	98.5	2.0
	w/ switch	364.5	7.3	197.8	4.0

4.2 Discussion

According to Table 5, the usage of hardware resources is 2,076 registers, 4,699 LUTs, and 1,444 slices which occupy

11%, 51%, and 63%, respectively, of the total capacities of Xilinx Spartan-6 FPGA (XC6SLX16CSG324C), where the number of tasks is five, that of semaphores is four, and that of eventflags is three. On the other hand, Table 6 shows that, when the numbers of tasks, semaphores, and eventflags are five, four, and zero, the usage is reduced to 1,431 registers, 3,601 LUTs, and 1,111 slices, which are 7%, 39%, and 48% of the capacities. Therefore, elimination of eventflags reduces registers, LUTs, and slices by 31.1%, 23.4%, and 23.1%, respectively, and improves the maximum frequency by a factor of 1.07. The same trend can be found from comparison between Table 5 and 8, in terms of presence of eventflags. From these results, reduction in hardware resources and improvement of the maximum frequency are expected by selecting only required RTOS functions.

From comparison between Table 6 and 7 in terms of error checking, it is confirmed that removal of the error-checking hardware contributes to resource reduction and higher frequency.

In addition, it is shown in Table 11 that our hardware implementation of RTOS, where main processing for RTOS is performed by the hardware, makes the code sizes of system calls 27.0% to 97.3% of those in the software-only implementation. Notice that the common processing functions (1,280 bytes for Soft Kernel) among various RTOS system calls, e.g., queuing operations for RTOS objects, is completely eliminated in the hardware implementation.

As for execution times of RTOS system calls shown in Table 12, compared to the software implementation, the hardware implementation takes 9.2% and 45.7% shorter execution times in the cases with task switching and without switching, respectively.

5. Conclusion

In this paper, we presented the hardware implementation of an RTOS kernel based on μITRON 4.0, where functions of system calls including error checking are built in an FPGA hardware resources. In the system, the RTOS kernel functions are invoked and used via memory reference instructions of a processor core with the MIPS instruction set. The hardware-implemented RTOS functions are selectable not only on a function basis but on a finer-unit basis, e.g., error-checking code fragment, which enables the system to adapt to the application codes and reduce the usage of hardware resources.

In the evaluation, it is confirmed that the hardware implementation proposed in this paper can simplify the software processing and reduce the size of software as well as the execution times. In addition, the results show that the proposed strategy can further reduce the hardware amount according to the application program by providing only functions/mechanisms required by it.

In the future, we extend the adaptation technique to automatic generation of both RTOS hardware and processor functions.

References

[1] M. Imai, Y. Takeuchi, K. Sakanushi, N. Ishiura, Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP), IPSJ Transactions on System LSI Design Methodology Vol.3, pages 161–178, 2010.

[2] M. K. Jain, M. Balakrishnan, A. Kumar, ASIP Design Methodologies: Survey and Issues, Proc of 14th Intl. Conf. on VLSI Design, pp.76–81, 2001.

[3] A.B. Lange, K.H. Andersen, U.P. Schultz, A.S. Sorensen: HartOS – a Hardware Implemented RTOS for Hard Real-Time Applications, 11th IFAC, IEEE International Conference on Programmable Devices and Embedded Systems, Volume 45, Issue 7, pp. 207–213, 2012.

[4] N. Maruyama, T. Ishihara, H. Yasuura, An RTOS in Hardware for Energy Efficient Software-based TCP/IP Processing, IEEE 8th Symposium on Application Specific Processors (SASP), 2010.

[5] N. Maruyama, T. Ichiba, S. Honda, H. Takada, A Hardware RTOS for Multicore Systems,' IEICE Transactions on Information and Systems, D, Vol.J96-D, No.10, pp.2150–2162, 2013. (In Japanese)

[6] N. Maruyama, T. Ishikawa, S. Honda, H. Takada, K. Suzuki, ARM-based SoC with Loosely coupled type hardware RTOS for industrial network systems, the 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, 2014.

[7] T. Miyauchi, K. Tanaka, Building Automatic Optimizing Environment for Multicore processors, Embedded Systems Symposium, pp.99–104 , 2015. (In Japanese)

[8] T. Miyauchi, K. Tanaka, Fine-Grained Configuration of RTOS Adapted to Applications, Embedded Systems Symposium, pp.73–81 , 2016. (In Japanese)

[9] T. Miyauchi, K. Tanaka, Building a Framework for an Application-Adaptive Processor System on FPGA-based SoC, The 21st Workshop on Synthesis And System Integration of Mixed Information technologies, pp.359-364, 2018

[10] H. Mori, K. Sakamaki, H. Shigematsu, Hardware Implementation of a real-time operating system for embedded control systems, Tokyo Metropolitan Industrial Technology Bulletin of Study No.8, pp55–58, 2005. (In Japanese)

[11] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, M. Imai, VLSI Implementation and Evaluation of a Real-Time Operating System, IEICE Trans. Inf.&Syst. (Japanese Edition) Vol.J78-D1 No.8, pp.679–686, 1995 (In Japanese)

[12] μITRON4.0 Specification Ver.4.01.00, ITRON Committee, TRON ASSOCIATION.

[13] MIPS® Architecture For Programmers, Volume II-A: The MIPS32® Instruction Set.

[14] "Spartan6" [Online] Available `http://www.xilinx.com/products/silicondevices/fpga/spartan-6.html`

[15] "Digilent" [Online] Available `http://store.digilentinc.com/`

[16] TOPPERS Kernel Test Suites "`https://www.toppers.jp/testsuites.html`"