Full Hardware Implementation of FreeRTOS-Based Real-Time Systems

Wakako Nakano†, Yukino Shinohara
Graduate School of Science and Technology
Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo, Japan

Nagisa Ishiura
School of Engineering
Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo, Japan

***Abstract*—As higher and higher functionalities are being implemented in embedded systems, it is becoming difficult to ensure their real-time performance. As one approach to enhancing response performance of RTOS-based systems, Oosako proposed a method for implementing both kernel objects and RTOS functionalities as hardware utilizing high-level synthesis, where TOPPERS/ASP3 was assumed as an RTOS. This paper extends this method to deal with systems based on FreeRTOS. In FreeRTOS, tasks can be generated either statically or dynamically, whose control data are managed in linked lists. We place restrictions that all the tasks are generated before scheduler starts so that we can keep the task control data in an array. Software timers are dealt with as tasks that have their own timers. We also present methods to implement dispatch disabling for mutual exclusion and a data queue for asynchronous data communication. We have implemented a hardware module from a reduced version of a demo program main_full.c and TimerDemo.c bundled with FreeRTOS, which took less than 300 ns and 700 ns for task control and data queue operations, respectively.**

***Index Terms*—real-time systems, RTOS, FreeRTOS, hardware accelerator, system synthesis, high-level synthesis**

I. Introduction

Recent advances in information and network technologies have made it possible to deploy various new services for our everyday life. Accordingly more and more functionalities are implemented in embedded devices. Especially in the area of unmanned aerial vehicle, autonomous cars, and service robots, high response performance is required as well as rich and sophisticated functionalities.

These real-time systems, where tasks associated with input events must be processed within specified periods, are implemented using a real-time operating system (RTOS). The RTOS helps designers to implement real-time systems by providing controllability and predictability on the execution time of concurrent tasks. However, it is getting more and more difficult to ensure real-time performance as the complexity of the systems grows.

One approach to this problem is hardware implementation. There have been many efforts to implement some or all functions of RTOS in hardware [1]–[6]. On the contrary, there have been some attempts to convert software tasks into hardware [7], [8], using high-level synthesis [9]. While these methods replace some parts of the systems by hardware, full-hardware implementation scheme have proposed [10], [11], which is however applicable only to bare metal systems.

For full-hardware implementation of RTOS-based systems, Oosako [12] proposed a method to synthesize both tasks and RTOS functions into hardware by high-level synthesis. Assuming that the tasks are created statically at compile time, every task is synthesized into an independent hardware component which can run in parallel with those of the other tasks. The scheduler is reduced to a simple controller without queues. Although this scheme drastically enhances response performance of systems, it has only been applied to systems using TOPPERS/ASP3 [13].

This paper extends this full-hardware scheme to support FreeRTOS [14]. All the tasks and handlers are resticted to be static so that the task control data can be managed in an array instead of linear lists. Methods to implement service calls, software timers, dispatch disabling, and data queues in this framework are presented.

A preliminary experiment on reduced version of a demo program bundled with FreeRTOS has demonstrated that the service calls for task control and for data queue operations took less than 300 ns and 700 ns, respectively.

II. Full Hardware Implementation of RTOS-Based Systems

A. Hardware Synthesis from TOPPERS-Based Systems

An RTOS (Real-Time Operating System) runs multiple sequential programs, called tasks, concurrently. It controls, or schedules, the execution of the tasks based on their priorities so that required work associated with input events to the system will be performed within specified periods. A handler is a special class of task that runs on an event, such as an external signal or timer expiration. We call tasks and handlers as kernel objects.

In [12], a method is proposed that synthesizes a given application program using system calls of TOPPERS/ASP3 into a hardware module which is functionally equivalent to a CPU that runs the program. In this method, each task or handler is synthesized into an independent hardware component by high-level synthesis [9].

Fig. 1 illustrates the resulting hardware configuration. TSKi, CYCi, ALMi, INTi on the right-hand side are hardware

† Currently with Hitachi, Ltd., Tokyo, Japan.

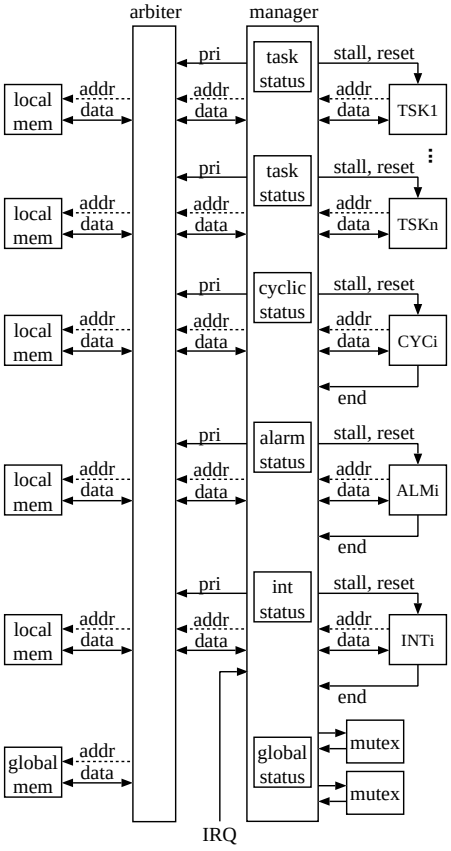


Fig. 1. Configuration of synthesized hardware [12]

components corresponding to tasks, cyclic handlers (executed periodically), alarm handlers (executed on timer expiration), and interrupt handlers (executed on external signals), respectively. The manager module provides the functions of the RTOS and the arbiter arbitrates the memory accesses to the same memory banks.

All the task modules are executed in parallel as soon as they become ready. The manager controls the task modules by the stall signals; a task module stops when the stall signal to the task is 1 and runs otherwise. The stall signal to a task module is generated from the state and priority of the task kept in the task status register and the system state in the global status register. The cyclic and alarm handler modules run in an autonomous way; they have their own timers and start execution when their timers expire. Atomic functions like lock/unlock are implemented by hardware.

The status registers are mapped in the memory space of the tasks/handlers, so the original code of the service calls in TOPPERS can be used with a little modification. Each task module is generated from source codes of the task and the called sevices by high-level synthesis.

Response time of the system will be drastically reduced by this scheme due to 1) no overhead regarding scheduling nor context switching, 2) parallel execution, and 3) hardware acceleration.

TABLE I
STATES OF TASK IN FREERTOS

State	Description
Running	The task is actually running
Ready	The task is able to execute but is not currently executing because a different task of equal or higher priority is already running
Blocked	The task is not able to execute for it is awaiting for either a temporal or external event
Suspended	The execution of the task was forcibly interrupted by another task

B. Differences between TOPPERS and FreeRTOS

From the viewpoint of our full-hardware scheme, there are two major differences between TOPPERS and FreeRTOS.

Creation of all the tasks and handlers are static in TOPPERS; the number of the tasks and handlers are fixed at compiler time. On the other hand in FreeRTOS, the tasks and handlers can be created dynamically. They are static (dynamic) if they are created before (after) the scheduler starts. For this reason, the TCBs (task control blocks) are maintained in linked lists, while an array is used in TOPPERS.

While TOPPERS defines cyclic and alarm handlers, FreeRTOS provides the same function in the form of software timers, which may be created dynamically. In the system call to create a software timer, xTimerCreate, a callback function along with a period is passed as an argument. The address of the callback function is stored in the control block of the software timer and the function is called when the timer expires.

III. FULL HARDWARE IMPLEMENTATION OF FREERTOS-BASED SYSTEMS

A. Overview

In this paper, we attempt to adapt Oosako's method to FreeRTOS to synthesize a hardware module which is functionally equivalent to a CPU that runs a given program written with service calls of FreeRTOS.

We keep the policy that a single task/handler is synthesized into an independent hardware module. Since hardware modules are inherently static objects, we place a restriction that all the kernel objects must be static. Namely, we deal with programs where all the tasks, handlers, and software timers are created before the scheduler is activated.

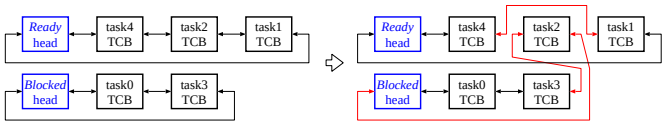
Based on this assumption, we manage the TCBs in an array instead of linked lists so that they are efficiently handled by hardware. We rewrite all the codes for the service calls that have depended on the linked lists, though there is no need to modify user programs.

Every software timer is implemented as an independent hardware module with its own timer whose body is synthesized from its callback function.

In addition, dispatch disabling for mutual exclusion and a data queue for asynchronous data communication are also implemented.

B. Management of TCBs

In FreeRTOS, each task is in one of the four states shown in TABLE I. The state of each task is kept in a TCB (task



(a) Linked lists of TCBs

(b) Array of TCBs

Fig. 2. Data structure for task management

TABLE II
STATE VARIABLES FOR TASK

Variable	Description
xState	* Current state
uxPriority	* Current priority
uxBasePriority	* Base priority
uxTimer	* Timer
ulNotifiedValue	* Value sent to notify
ucNotifyState	* State of notify
pcTaskName	task name
uxMutexesHeld	Flags of held mutexes
uxCriticalNesting	Nesting count of critical sections

* Kept in the status registers in the manager

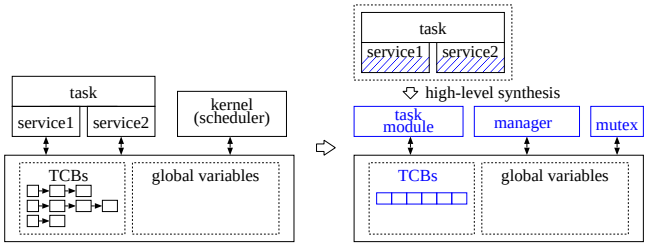


Fig. 3. Synthesis of service hardware.

```
1: void vTaskResume( TaskHandle_t xTaskToResume )
2: {
3:    TCB_t * const pxTCB = xTaskToResume;
4:    configASSERT( xTaskToResume );
5:    if ( pxTCB–>xState != eRunning && pxTCB != NULL )
6:    {
7:       _loc_service_call();
8:       {
9:          if( pxTCB –>xState == eSuspended )
10:         {
11:            pxTCB –>xState = eReady;
12:         }
13:      }
14:      _unl_service_call();
15:   }
16: }
```

Fig. 4. Implementation of vTaskResume

control block) along with the other information regarding the task.

FreeRTOS manages the TCBs in linked lists each of which keeps the TCBs of the tasks in the same state. As illustrated in Fig. 2 (a), when the state of task 2 changes from *Ready* to *Blocked*, the TCB of task 2 is relinked from the *Ready* list to the *Blocked* list.

We change the data structure so that all the TCBs are managed in a single array, for we have assumed all the tasks are static and the number of the tasks is fixed. The state of a task is simply stored as a member of its TCB; when the state of task 2 changes from *Ready* to *Blocked*, just the state in the TCB is updated, as shown in Fig. 2 (b).

All the members in the TCBs are basically stored in the task registers (labeled as "task status" in Fig. 1). However, to reduce the hardware size, the members which do not affect the stall signal may be stored in the local memories for the tasks. For example, out of 9 state variables listed in TABLE II, 6 variables marked by "*" should be held in the registers. The variables representing the system state are also held in registers in the manager (labeled as "global status" in Fig. 1).

The task status registers and the global status registers are mapped in the memory space of the tasks so that they can be accessed from tasks by reads/writes of the variables tied to them.

C. Task Control and Implementation of Service Calls

The basic scheme for task control and service calls is the same as that in [12]. Tasks are executed as soon as they become *Ready*. For this purpose, the manager forces the states of the tasks into *Running* as soon as they become *Ready*. Namely, when a task become *Ready*, it turns *Running* in the next cycle and starts execution.

Fig. 3 illustrates how service calls are synthesized into hardware. Basically, the body of the services that a task calls are converted into hardware together with the task by high-level synthesis. The body of the services are rewritten so that:

1) it conforms with the change of the data structure of TCBs, and
2) it can offload part of the scheduling or mutual exclusion related burden to dedicated hardware.

Note that there is no need to rewrite the user codes (of tasks and handlers).

For example, a service call vTaskResume, which changes the state of a specified task from *Suspended* to *Ready*, can be rewritten as shown in Fig. 4. Line 3 is to get an access to the specified task and line 4 is for an error check. Lines 7 and 14 are internal calls to acquire/release the hardware lock for serializing system calls (to avoid interference among service calls). Line 11 changes the task state, as was shown in Fig. 2. xSate of pxTCB is mapped to a member of the status register, so that the manager hardware takes care of the scheduling related work.

D. Disabling Dispatch

In this paper, we add a new feature of disabling dispatch, which is used as a measure for mutual exclusion among tasks.

In a common software RTOS implementation, dispatch is disabled by halting the scheduler so that the control will

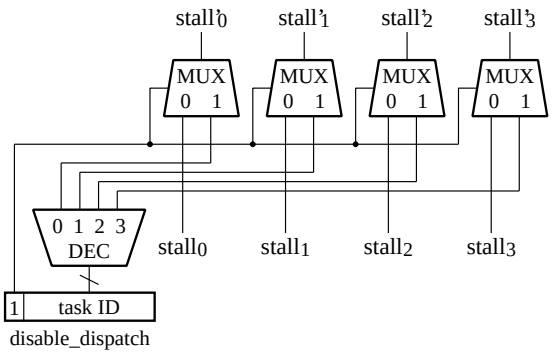


Fig. 5. Hardware design for disabling dispatch.

```
 1: xQueueReceive() {
 2:    if (Queue is not empty) {
 3:       get the head data d out of Queue;
 4:       if (there are tasks whose xQueueSend() is Blocked) {
 5:          make one of the Blocked tasks Ready;
 6:       }
 7:       pass d to the calling task;
 9:    }
10:    else {
11:       set the timer of the calling task;
12:       make the calling task Blocked;
13:       // wait (resumes as soon as the task becomes Ready)
14:       if (Queue is not empty) {
15:          get the head data d out of Queue;
16:          pass d to the calling task;
17:       }
18:       else { // timer has been expired
19:          return ERROR;
20:       }
21:    }
22: }
```

Fig. 6. Outline of xQueueReceive call

not be passed to the other tasks. In our design, however, all the possible tasks are running in parallel, so disabling the scheduler does not work. As an alternative way, we let the manager stall all the other tasks than the one called the service to disable dispatch. This guarantees mutual exclusion.

This scheme is implemented by extending a global flag variable, which indicates if dispatch is disabled, also to record the ID of the task which has requested to disable dispatch. If the flag is set, all the tasks whose ID is not equal to the recorded ID are stalled. Fig. 5 illustrates an example of hardware design. The global variable to control disabling dispatch is held in a register in the manager. The revised stall signal "stall'$_i$" is generated from the original stall signal "stall$_i$" by a simple hardware consisting of a decoder "DEC" and multiplexers "MUX."

E. Implementation of Data Queue

A data queue provides asynchronous data communication among tasks. It keeps data in a fixed length array of bytes where xQueueSend call puts a byte at the tail and xQueueReceive call gets a byte from the head. If a task attempts xQueueSend on a full data queue, the task is blocked

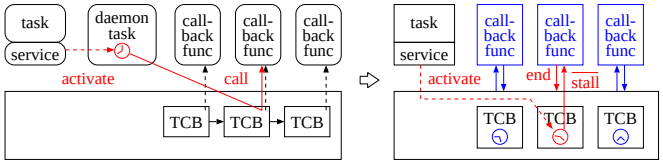


Fig. 7. Implementation of software timer

until another task dequeues data. Similarly, tasks attempting xQueueReceive on an empty data queue is blocked. Blocked tasks are kept in a linked list. Time to wait may be specified at xQueueSend/xQueueReceive calls; if send/receive does not complete in the specified time, an error code is returned.

We implement the data queue with a combination of hardware and software. The linked list of blocked tasks is replaced by an array of bits where the i-th element is set iff the task i is blocked. Instead of the centralized timer, we provide a timer to every task. The timer is kept in the status register of the task and the manager decrements all the non-zero timer at every tick and changes the state of the tasks to *Ready* whose timer become zero.

The outline of the body of the service call xQueueReceive is shown in Fig. 6. If the queue is not empty (line 2), then get the head data out of the Queue and pass it back to the calling task. If there are tasks whose xQueueSend() calls are *Blocked* (namely the Queue was full), wake up one of the tasks of the highest priority by making its state to *Ready*. If the Queue is empty (line 10), let the task wait until another task calls xQueueSend() by setting the timer of the task and making the task *Blocked*. The execution resumes at line 14 as soon as the task becomes *Ready*, when the timer expires or some task has called xQueueSend(). In either case, if Queue is not empty then dequeue the Queue. Otherwise, notify the calling task that it has timed out.

F. Implementation of Software Timers

Since we assume the software timers are created before the scheduler starts, the number of software timers is fixed. We provide for each software timer a hardware module, whose body is synthesized from the callback function of the software timer.

An overview of our software timer implementation is illustrated in Fig. 7. In the original scheme in FreeRTOS (left-hand side), the daemon task manages the execution of all the software timers. When a task calls a service call to activate a software timer, a command is sent to the daemon task which sets the timer period in the control block of the software timer. As soon as the daemon task notices the timer expilation, it calls the callback function whose address is stored in the control block. In our scheme, all the timers are autonomous. Tasks activate a software timer by directly writing its control block via the service call. Each software timer is provided with a timer in its control block, whose decrements and expiration is dealt with by the manager hardware. At timer expiration, the

TABLE III
IMPLEMENTED HARDWARE MODULES

module	function
top	top module to connect all modules
manager	manager
arbiter	memory arbiter
lock0	support hardware for lock
lock1	support hardware for lock
LIM_INC	task to increment shared variable
CNT_INC	task to increment shared variable
C_CTRL	task to control LIM_INC and CNT_INC
Tmr Tst	task to test timers
AR_TMR1	auto-reload timer
AR_TMR2	auto-reload timer
OS_TMR1	one-shot timer
ISR_OS_TMR1	one-shot timer activated from ISR
ISR_OS	ISR that activates timer
SUSP_SEND	task to test disable dispatch and xQueueSend
SUSP_RECV	task to test disable dispatch and xQueueReceive

TABLE IV
RESULT OF SYNTHESIS

(a) Execution cycles and latency

service call	#cycle	latency [ns]
xTaskResume	20	299.12
vTaskSuspend	11	164.52
vTaskDelay	9	134.60
vTaskSuspendAll	15	224.34
xTaskResumeAll	16	239.30
vTaskPrioritySet	20	299.12
xTimerStart	14	209.38
xTimerStop	15	224.34
xTimerReset	15	224.34
xQueueSend	43	643.11
xQueueReceive	46	687.98
interrupt	1	14.96

critical path delay = 14.956 [ns]

(b) Circuit size

module	#LUT	#FF
top	370	0
manager	10,106	3,736
arbiter	624	11
lock0	32	21
lock1	32	21
LIM_INC	10,928	902
CNT_INC	11,817	992
C_CTRL	13,478	1,020
Tmr Tst	11,385	1,031
AR_TMR1	12,635	939
AR_TMR2	12,319	895
OS_TMR1	12,080	934
ISR_OS_TMR1	12,829	905
ISR_OS	—	—
SUSP_SEND	10,347	986
SUSP_RECV	12,693	951
total	131,675	13,344

High-level synthesizer: ACAP (2016.10)
Logic synthesizer: Xilinx Vivado (2018.3)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

manager suppresses the stall signal to run the task module. The task module notifies completion by raising its end signal.

The detailed execution flow of software timers is as follows:

0) Initially all the software timers are in the *Blocked* state.
1) A task calls xTimerStart to activate a software timer T. xTimerStart accesses the control block of T to change the state of T from *Blocked* to *Running* and to set time to T's timer.
2) The manager decrements the timer of the T at every tick, and when it becomes 0 then make the stall signal 0 to allow execution of the hardware module for T.
3) The hardware module notifies the completion by setting the end signal to 1.
4) As soon as the manager detects end==1:
   - it changes the state of T to *Blocked*, if the T is a one-shot timer,
   - it re-sets the period to the timer and return to 2, if the T is an auto-load timer.

A small difference between TOPPERS and FreeRTOS is that the function associated with a timer is executed in the context of the handler in TOPPERS but of the daemon task in FreeRTOS. In the FreeRTOS-based design, we should change to the policy that the priorities of all the software timers are forced to be equal to that of the daemon task and that execution of the software timer modules are deferred when dispatch is disabled.

TABLE V
IMPLEMENTED SERVICE CALLS

task control	xTaskResume vTaskSuspend vTaskDelay vTaskSuspendAll xTaskResumeAll vTaskPriorityGet vTaskPrioritySet eTaskGetSet pcTaskGetName
mutual exclusion	vTaskENTER_CRITICAL vTaskEXIT_CRITICAL
timer control	xTimerIsTimerActive xTimerStart xTimerStartFromISR xTimerStop xTimerStopFromISR xTimerReset xTimerResetFromISR pcTimerGetName xTimerChangePeriod vTimerSetTimerID vTimerGetTimerID
queue	xQueueSend XQueueReceive
miscellaneous	configASSERT

IV. PRELIMINARY EXPERIMENT

Based on the proposed method, we have implemented a hardware module from main_full.c and TimerDemo.c, demonstration programs bundled with FreeRTOS. The programs are reduced so that they can test only switching of tasks, software times, and queues.

The resulting hardware modules are listed in TABLE III. The first 5 modules ("top" through "lock1") are generated by a script from a parameter file based on manual design in Verilog HDL. The other 11 modules for tasks, software timers, and a handler are generated by high-level synthesis from the source codes. Some manual code modifications have been made on the source codes; to adapt their interfaces to the high-level synthesizer or to identify functions for the tasks, the timers, and the handler, but the body of the functions are unmodified.

As a high-level synthesizer, we used ACAP [15]. ACAP can generate Verilog HDL of register transfer level circuits from executable binary codes of MIPS (R3000), which are generated by GCC. Since ACAP preserves memory accesses

of the original binary codes, there was no need to modify the source codes regarding global (common) variable accesses.

All the Verilog HDL codes are synthesized by the logic synthesizer of Xilinx Vivado (2018.3) targeting Xilinx FPGA Artix-7 (xc7a100tcsg324-3).

TABLE IV (a) shows the response performance of the synthesized hardware for the service calls. "#cycle" and "latency" are the clock cycles and estimated time from the calls to completion of the services. The latency is the product of the cycle and the critical path delay, which is 14.96 ns. For example, xTaskResume takes 20 cycles (299.12 ns) to change the state of the target task from *Suspended* to *Ready*; it takes just 1 cycle to update the state but consumes 19 cycles for error checking and lock/unlock for service call serialization. All the services for task control were completed within 300 ns, which is by far the faster than software implementation. Even the time consuming services for data queue manipulation were processed within 700 ns.

TABLE IV (b) shows the circuit size of the hardware module in the numbers of flipflops (#FF) and LUTs (#LUT). The total LUT count of the hardware exceeded 130,000, which is unfortunately a little too large for practical use. This is just because this implementation is the very first version; there are many duplicated and redundant submodules, which should be removed by optimization or improvement of the architecture.

At this point, we have implemented 25 services (listed in TABLE V) of FreeRTOS out of 132.

V. Conclusion

This paper has proposed a method for full hardware implementation of FreeRTOS-based systems. By assuming all the tasks, handerls, and software timers are created statically, a given application program using system calls of FreeRTOS is synthesized into a hardware module which is functionally equivalent to a CPU that runs the program.

A preliminary experiment on reduced version of a demo program bundled with FreeRTOS demonstrated that the service calls for task control and for the data queue operations took less than 300 ns and 700 ns, respectively.

However, the size of the synthesized circuit is a little too large. We are now working on an improved architecture to remove duplicated or redundant hardware components. We are also looking for a way to adapt our scheme to more general purpose high-level synthesizer such as Vitis HLS. On the other side, we are also working on implementing efficient RTOS service modules such as mutexes, event groups, and message buffers, in a hardware/software hybrid manner.

Acknowledgment

Authors would like to express their appreciation to Dr. Hiroyuki Kanbara of ASTEM/RI, Prof. Hiroyuki Tomiyama of Ritsumeikan University, and Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their valuable advice and comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin University for their discussion on this work. This work was partly supported by JSPS KAKENHI under Grant No. 19H04081.

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 Conference (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. International Workshop on Rapid System Prototyping (RSP '06)*, pp. 163–168 (June 2006).

[3] P. Kohout, B. Ganesh, and B. Jacob: "Hardware support for real-time operating systems," in *Proc. International Conference on Hardware/-Software Codesign and System*, pp. 45–51 (Oct. 2003). DOI:http//doi.org/10.1145/944645.944656

[4] 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).

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

[6] https://www.renesas.com/en-us/products/factory-automation/multi-protocol-communication/r-in32m3-hardware-rtos.html (accessed 2021-09-05).

[7] S. Shibata, S. Honda, H. Tomiyama, and H. Takada: "Advanced system-builder: A tool set for multiprocessor design space exploration," in *Proc. International SoC Design Conference (ISOCC 2010)*, pp. 79–82 (Nov. 2010).

[8] Y. Ando, S. Honda, H. Takada, M. Edahiro: "System-level design method for control systems with hardware-implemented interrupt handler," *IPSJ Journal of Information Processing*, vol. 23, no. 5, pp. 532–541 (Sept. 2015).

[9] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[10] N. Ito, N. Ishiura, H. Tomiyama, and H. Kanbara: "High-level synthesis from programs with external interrupt handling," in *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2015)*, pp. 10–15 (Mar. 2015).

[11] N. Ito, Y. Oosako, N. Ishiura, and H. Tomiyama, and H. Kanbara: "Binary synthesis implementing external interrupt handler as independent module," in *Proc. International Symposium on Rapid System Prototyping (RSP 2017)*, pp. 92–98 (Oct. 2017).

[12] 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 2017)*, pp. 1–7 (Oct. 2018).

[13] https://www.toppers.jp/ (accessed 2021-09-05).

[14] https://www.freertos.org/ (accessed 2021-09-05).

[15] N. Ishiura, H. Kanbara, and H. Tomiyama: "ACAP: Binary synthesizer based on MIPS object codes," in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).