Hardware RTOS Services for Full Hardware Implementation of RTOS-Based Systems

Hiro MINAMIGUCHI Masaki NAKAHARA Yugo ISHII
Yukino SHINOHARA Iori MUGURUMA [†,*] Nagisa ISHIURA
Kwansei Gakuin University
1 Gakuen Uegahara, Sanda, Hyogo, 669-1330 Japan

Abstract—This paper presents hardware implementation of RTOS services for full hardware implementation of RTOS-based systems, where all the task programs and all the RTOS functions are implemented as hardware. Hardware methods for processing services of mutexes, event flags, data queues, shared variable accesses, and task control are proposed. Wait and release operations necessary in synchronization and communication services are efficiently performed using a request arbitration module. Timeouts are also handled by hardware using distributed timers. A hardware module that contains two mutexes, two event flags, one data queue of 320B data, and shared variable of 1024B, as well as task scheduling and control functions, has been designed in Verilog HDL. It was synthesized to an FPGA circuit of 4,300 LUTs and 2,200 flip-flops (Xilinx Artix-7). All the services can be executed well in 150 ns, which is fast enough even for extreme applications.

I. Introduction

Recent advances in information and communication technologies are driving invention and deployment of new digital services for our society. Within this trend, embedded devices are getting more and more rich and sophisticated in their functionalities. In some areas such as unmanned aerial vehicle, autonomous cars, and service robots, high response performance is also required in addition to the functionalities.

A real-time operating system (RTOS) is an indispensable building block for such real-time systems, where tasks must be processed within the specified periods in response to input events. It helps designers to implement real-time systems by providing controllability and predictability on the execution time of concurrent tasks. However, as the heavier loads are posed on the tasks, it is getting the more difficult to ensure real-time performance.

As an approach to lessen burden on CPUs, there have been many efforts to implement some or all functions of RTOS as hardware [1, 2, 3, 4, 5]. On the other hand, there have also been some attempts to execute tasks by hardware [6, 7]. While only some parts of systems are implemented as hardware in these methods, [8] and [9] have proposed methods to implement a whole system as hardware, which are however only applicable to bare metal systems.

As a method to implement a whole RTOS-based system as hardware, Oosako [10] proposed a full hardware scheme using 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 tasks are controlled by a manager hardware that runs/stops the tasks based on the tasks statuses, which does not need ready queues. The preliminary implementation supports FreeRTOS[1] [11] as well as TOPPERS/ASP3[2]. However, the resulting circuits were too large even for small demo programs.

To address this issue, Ando and Muguruma has proposed a different architecture for full hardware implementation of RTOS-based systems [12]. Duplication of hardware for processing RTOS services (service hardware) is eliminated by moving the service hardware from the task modules to the manager. It also enables the use of a general commercial high-level synthesizer, instead of an in-house binary synthesizer. They have succeeded in drastically reducing the circuit size by this architecture.

This paper proposes hardware implementation of RTOS services for the architecture in [12]. Mutexes, event flags, data queues, task control, and shared memory access, are implemented as hardware modules, where wait and release for task synchronization are efficiently implemented utilizing a request arbitration hardware. Timeouts are also efficiently handled by hardware using distributed timers. An example manager module which can control 4 tasks and provide 5 services has been designed in Verilog HDL, which is synthesized into a circuit of reasonable size and of extremely high response performance.

II. full hardware implementation of RTOS-based systems

A. full hardware implementation

A real-time operating system (RTOS) controls the execution of concurrently running sequential computation entities, called tasks. It schedules execution of the tasks based on their priorities so that tasks associated with input events will be performed within specified periods.

Oosako proposed a full hardware scheme where both the tasks and the RTOS functions are implemented as hardware which is functionally equivalent to a CPU that runs the program [10]. Fig. 1 shows the concept. “task1” through “task*n*” are software tasks running under an RTOS. They are converted

[*] Currently with Honda Motor Co., Ltd., Japan.

[1] `https://www.freertos.org/`
[2] `https://www.toppers.jp/`

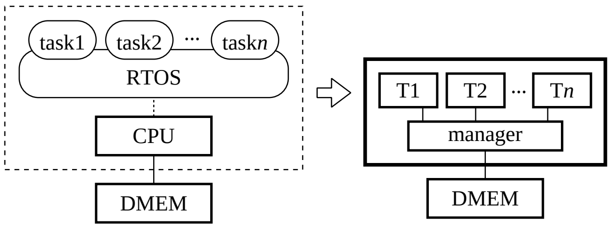


Fig. 1. full hardware implementation of RTOS-based system [12]

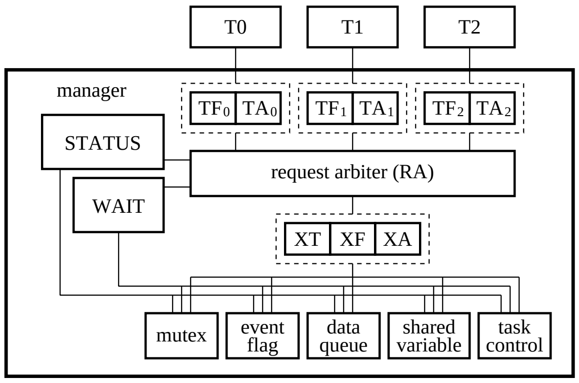


Fig. 2. Architecture of manager [12]

into hardware modules (task modules) "T1" through "Tn" by high-level synthesis [13]. "Manager" is a hardware module that replaces the RTOS. It controls execution of the task modules and provides services such as mutexes and data queues.

All the task modules are executed in parallel as soon as they become ready. The manager simply controls run/pause of each task by a signal generated from the task state, which eliminates the need for the ready queues and the task scheduler.

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

Limitation to this scheme is that all the tasks must be statically created (at compile time). Moreover, the number of tasks is limited to as much as 16. Although tasks are executed in parallel, RTOS services are processed sequentially so as to avoid interference among them.

B. *Architecture of Manager*

The hardware configuration of the manager proposed in [12] is shown in Fig. 2. "T0", "T1", "T2" are task modules. "mutex", "event flag", through "task control" in the manager are service modules that provides RTOS services. The services are processed one at a time to avoid conflicts among them. "request arbiter" (RA) is a module to serialize execution of services when multiple requests are issued from tasks at the same time. "STATUS" is a collection of registers that keeps the state, the priority, the timer, and so on of each task. "WAIT" is a two dimensional array of flags to keep track of which task is waiting for which service.

Ti issues a request for a service by writing the ID of the service and necessary arguments to registers TFi and TAi, respectively. If multiple tasks issued the requests at the same time, the RA chooses one of them based on the priorities of the tasks. The RA writes the ID of the selected task to register XT

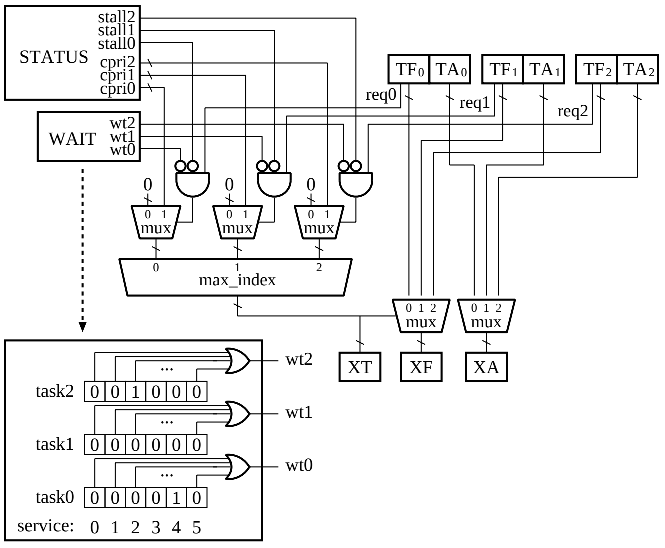


Fig. 3. Hardware configuration of request arbiter (RA)

and copies the contents of TFi and TAi to registers XF and XA, respectively. Then the service module in charge processes the request and writes the result to XA, which is forwarded back to the task through TAi.

As soon as a task becomes *Ready*, the manager changes its state to *Running* in the next cycle, so that all the ready tasks run in parallel. In the hardware scheme of [12], all the tasks, including the ones in *Dormant* or *Suspended* states, are allowed to run, though only service requests from *Running* tasks are processed; requests from *Dormant* or *Suspended* tasks are blocked until they become *Running*. Since the access to the shared global variables is also dealt with as one of the RTOS services in this scheme, *Dormant* or *Suspended* tasks may update their local states but could not affect the other tasks nor external devices.

III. Hardware RTOS services

A. *Overview*

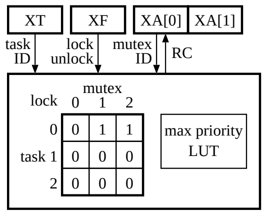
This paper proposes hardware implementation of RTOS services dedicated to the architecture of [12]. The technical features are as follows:

- Mutexes, event flags, data queues, task control, and shared memory access, are implemented as hardware that works in cooperation with the request arbiter module.
- Wait and release operations which are essential for synchronization and communication services are implemented with a combination of the WAIT register and the request arbiter, without using lists of waiting tasks.
- Timeouts are efficiently handled by hardware using distributed timers instead of a centralized timer.

B. *Request arbiter (RA)*

The request arbiter (RA) module receives requests from tasks and forwards them to the service modules. If there are multiple requests at the same time, they are arbitrated according to the priories of the tasks.

Fig. 3 shows the hardware configuration of the RA.



(a) Configuration

priority ceiling

mutex	2	1	0
ceiling	3	5	1

look-up table

lock[t][*]	max priority
0 0 0	–
0 0 1	1
0 1 0	5
0 1 1	5
1 0 0	3
1 0 1	3
1 1 0	5
1 1 1	5

(b) Table to lock-up maximum priority ceiling

Fig. 4. Mutex module

Signal stall*i* generated from the STATUS register means that task*i* must pause because the state of task*i* is not in the *Running* state. Signal cpri*i* is the current priority of task*i* where the larger value means the higher priority. WAIT is a two dimensional array of flags where WAIT[*i*][*s*] is set iff task*i* is waiting for some service *s*. Signal wt*i* means that task*i* is waiting for some service. "Max_index" unit outputs the task ID of the highest current priority. The inputs to the max_index unit are blocked if the tasks are not in the *Running* state or the tasks are waiting for services. The resulting task ID is written into register XT, and TF*i* and TA*i* of the selected task are copied into XF and XA.

The arbitration completes in a single clock cycle, though the tree to priority comparators in the max_index unit will form the critical path of the combinational logic.

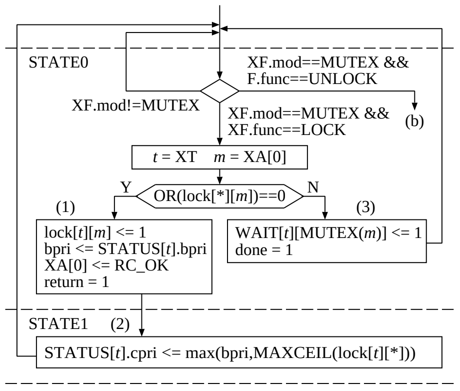
C. Mutex

A mutex provides the atomic lock/unlock operations with a protocol to prevent priority inversion. The priority ceiling protocol is assumed in this paper; a locker-priority is defined to each mutex, and when a task locks mutexes, the current priority is raised to the maximum value among the locker-priorities of the mutexes that the task locks and the task's base priority. When multiple tasks are waiting for a mutex and the locker task unlocks the mutex, the waiting task of the highest priority acquires the lock.

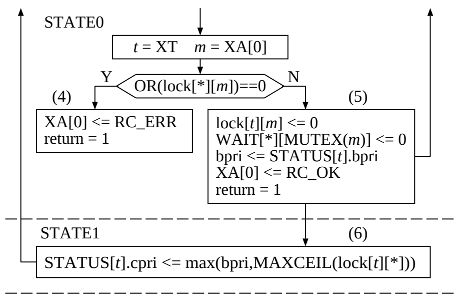
In our scheme, a single mutex module manages all the mutexes defined in a given system. Fig. 4 (a) illustrates its configuration. It contains a matrix of flags, where lock[*t*][*m*] is set iff task *t* is locking mutex *m*, and a look-up table to find the maximum priority ceiling. The RTL behavior of the mutex module for the lock operation is shown in Fig. 5 (a).

In the initial state (STATE0) it watches XF until the ID of this module, MUTEX, is written to its module field (mod). Then if LOCK function is specified in XF's function field (func), it gets task ID *t* and mutex ID *m* from XT and XA[0] and tests if mutex *m* is locked by any other task.

If it is not locked, the module lets task *t* lock mutex *m* as described in (1) and (2). It sets lock[*t*][*m*] flag and updates the current priority of task *t*. This takes two cycles; it reads



(a) lock



(b) unlock

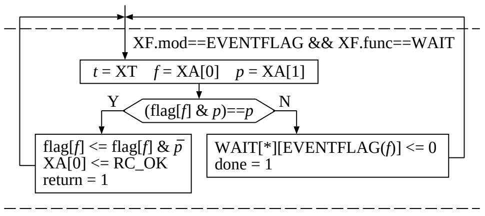
Fig. 5. RTL behavior of mutex module

the base priority (bpri) of *t* from the STATUS register and then updates the current priority (cpri). The maximum ceiling value among the mutexes locked by task *t* (MAXCEIL) is found in the look-up table. An example of the table is shown in Fig. 4 (b), where the ceiling value of mutexes 2, 1, and 0 are 3, 5, and 1, respectively. If task *t* has locked mutexes 2 and 0, then lock[*t*][*] is 101 and the maximum ceiling is found to be 3. After writing a return code (RC_OK) to XA[0], it raises the completion signal (return) to return control to the manager.

Though it takes two cycles to read and write the current priority, the write to the STATUS register can be done in parallel with the copy of XA to TA*t* by the manager. For this purpose, the mutex module sets the return code and notifies the completion in the first state.

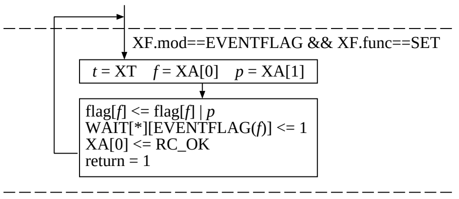
When the mutex is already locked, it proceeds to (3) in Fig. 5 (a). While typical software implementation involves manipulation of a list of waiting tasks, which should be kept sorted by the tasks' priorities, our hardware implementation does not use the list. It just sets the [*t*][MUTEX(*m*)] bit of the WAIT register, where MUTEX(*m*) is the serial service number of mutex *m*. This keeps the request from task *t* waiting at the input of the RA. A different completion signal (done) is used to pass control back to the manager not to return the result code to the task.

The behavior for the unlock operation is shown in Fig. 5 (b). If the mutex is not locked, an error code is returned (4). Otherwise, the lock is released in two cycles ((5) and (6)). After clearing the lock bit, it clears all the WAIT[*][MUTEX(*m*)] bits. This temporarily releases the requests for mutex *m* from waiting. Then, the request of the highest task priority will be selected by the RA and forwarded to the mutex module. The update of the current priority of the task is done in the same



XF.mod==EVENTFLAG && XF.func==WAIT
t = XT f = XA[0] p = XA[1]
(flag[f] & p)==p
Y
N
flag[f] <= flag[f] & p̄
XA[0] <= RC_OK
return = 1
WAIT[*][EVENTFLAG(f)] <= 0
done = 1

(a) wait



XF.mod==EVENTFLAG && XF.func==SET
t = XT f = XA[0] p = XA[1]
flag[f] <= flag[f] | p
WAIT[*][EVENTFLAG(f)] <= 1
XA[0] <= RC_OK
return = 1

(b) set

Fig. 6. RTL behavior of eventflag module

way as in the lock operation.

Although it takes 3 more cycles before the waiting task of the highest priority resumes execution, the release is done by the RA and thus the task that issued the unlock request does not have to wait for it.

D. Event flag

An event flag provides a tool for task synchronization. Its basic operations include set and wait. By the set operation, a task can set a bit pattern to the event flag, and by the wait operation, a task can wait for a desired bit pattern to be set in the event flag.

As with the mutex module, a single event flag module manages all the event flags in the system. The module has an array of registers (flag) where the bit pattern set for event flag f is kept in flag[f].

The RTL behavior for the wait operation is shown in Fig. 6 (a). It gets the ID f of the event flag and bit pattern p from XA. If all the bits in p are set in flag[f], it clears the bits and return to the task. Otherwise, it forces task t to wait by setting the flag in the WAIT register.

The behavior for the set operation is shown in Fig. 6 (b). It first updates flag[f] by ORing it with p. Then it clears all the bits regarding f in the WAIT register. This releases the requests waiting for f at the RA, if any, which are resent to the event flag module in the order of the task priorities.

The both operations take only a single cycle.

E. Data queue

A data queue is a mechanism for passing fixed length data between tasks. Its basic operations include send and receive, where send appends a datum at the end of the queue and receive takes the head data out of the queue. The maximum number of data is predefined for each queue. Thus, a send operation to a full queue makes the sender task wait until some other task receives from this queue. When multiple send operations are waiting, they are processed in the order of the task priorities. Receive operations on an empty queue are processed in the same manner.

Fig. 7 shows the configuration of the data queue module, which is in charge of all the data queues defined in the system. Each data queue q is managed with an array of data (data[q][*]), read and write pointers (rp[q] and wp[q]), and a register to record the number of data in the queue (used[q]).

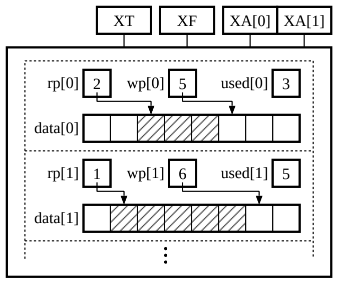


Fig. 7. Data queue module

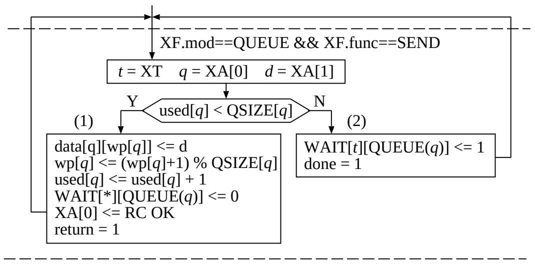


Fig. 8. RTL behavior of data queue module (send)

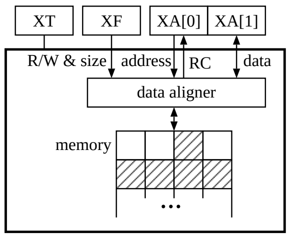


Fig. 9. Shared variable read/write module

The RTL behavior for the send operation is shown in Fig. 8. After receiving a task ID (t), a target queue ID (q), and a datum (d), it tests if there is a space for d in the queue. If yes, (1) it enqueues d and notifies success with the return code. At this point, it resets all the wait flags regarding q in the WAIT register, to release all the waiting send requests for queue q (if any). If the queue is full, (2) it sets the wait flag WAIT[t][QUEUE(q)] to let task t wait.

The receive operation is similar to the send operation. The both operations complete in a single cycle.

F. Shared variable access

An access to the shared (global) variable is dealt with as one of the services of the RTOS in the architecture of [12]. The read/write requests are processed in the order of task priorities.

As shown in Fig. 9, the major components of the shared variable module are a memory array and a data aligner. The module receives an operation code from XF, which specifies read or write and the number of bytes (1, 2, or 4) of the datum and an address from XA[0]. In the case of the read operation, the return code and the datum are written back to XA[0] and XA[1], respectively. In the case of the write operation, a datum in XA[1] is written into the memory and a code is returned to XA[0].

Though the memory array is logically inside the module,

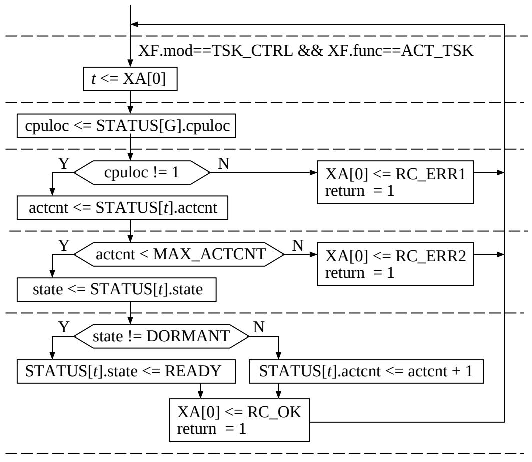


Fig. 10. RTL behavior for task activation

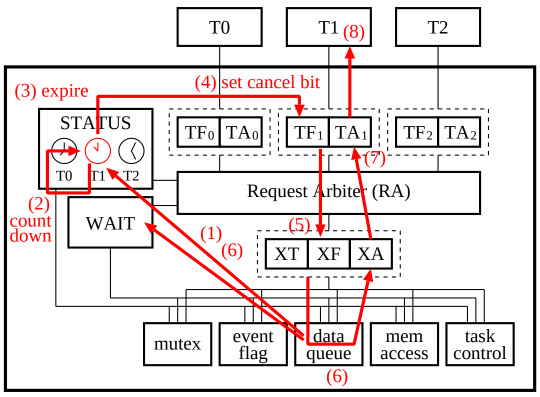


Fig. 11. Mechanism of timeout

it may be physically implemented with flip-flops, on-chip RAMs, or off-chip RAMs.

G. *Task control*

A task control module processes all the service calls to change the states or the priorities of the tasks. The services are processed mainly by reading/writing the STATUS register.

An example of the RTL behavior for task activation is shown in Fig. 10, where the state of the task specified in register XA[0] is changed from *Dormant* to *Ready*. Though the essential operation is just the update of STATUS[t].state in the final state, this operation takes 5 cycles including error handling.

H. *Timeouts*

An optional timeout period may be specified for mutex lock, event flag wait, and data queue send/receive requests. This paper proposes an efficient way of handling timeouts by hardware.

The mechanism of timeout is shown in Fig. 11. Each task is provided with a timer which is kept in the STATUS register.

When a service module lets a request wait with a timeout period, it sets the period into the timer of the task at the same time it sets the flag in the WAIT register (1).

The manager decrements active timers at every clock (2). When a timer expires (3), the manager lets the service module deal with cancellation. For this purpose, the manager sets the cancel bit in register TF (4). If this bit is set, the RA forwards

TABLE I
SYNTHESIS RESULT

module	#LUT	#FF	delay
mutex (2)	1070	45	
eventflag (2)	170	64	
data queue ($1 \times 1B \times 8$)	117	10	
shared variable (32B×32)	112	0	
task control	2,417	151	
manager	697	1,913	
total	4,583	2,183	7.349 [ns]

Logic synthesizer: Xilinx Vivado (2016.4)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

the request with the highest priority (5) [3].

Recognizing the cancel bit, the service module processes cancellation (6), which includes reset of the WAIT flag as well as returning of an error code. The manager forwards the error code to the task (7) and then the task knows the request has been expired.

For service modules, the overhead of timeout handling is just to set timers, which can be done in parallel with the normal processing in many cases, and to do cancel processing. The rest of the work is done by the manager hardware. In the worst case, where another request which takes T cycles is forwarded to the service module at the same cycle as the cancel bit is set, the waiting task resumes execution $6 + T + C$ cycles after the timer expiration, where C is the number of cycles for the cancel processing.

IV. EXPERIMENTAL RESULT

Based on the proposed architecture, a manager module along with service modules has been designed in Verilog HDL. The design is synthesized using Xilinx Vivado (2016.4) targeting Xilinx Artix-7 (xc7a100tcsg324-3) FPGA.

The size in terms of the LUT and flip-flop counts and the critical path delay of the resulting circuit are shown in Table I. The experimental setup is as follows:

- The manager has 4 ports for tasks.
- The mutex module has 2 instances.
- The event flag module has 2 instances.
- The data queue module has an instance of 32B×10 data.
- The shared variable module has $32B \times 32$ word memory.
- The task control module implements the following 12 services of TOPPERS/ASP3

  act_tsk, can_act, ter_tsk, chg_pri, get_pri, wup_tsk, can_wup, rel_wai, sus_tsk, rsm_tsk, loc_cpu, unl_cpu
- At this point, timeout has not been implemented.

It seems that the data arrays in the data queue and shared variable modules were synthesized with LUT-RAMs rather than block RAMs. The task control module was large because it must handle many services with long state transitions for error handling. The critical path delay, which is considered to come from the comparator tree to find the highest priority task in the RA, was small enough in this setup.

[3]If the cancel occurs during some other service is being processed, the cancel is processed after the current service is finished.

TABLE II
RESPONSE PERFORMANCE

module	service	#cycle
mutex	lock	5
	unlock	5 (+3)
event flag	wait	5
	set	5 (+3)
data queue	send	5 (+3)
	receive	5 (+3)
shared memory	read	5
	write	5
task control	act_tsk	10
	wup_tsk	10
	ext_txk	10
	ras_ter	9
	ter_tsk	7
	slp_tsk	15

The response performance of the services are summarized in Table II. The number of cycles for locking a free mutex, for example, consists of 1) a cycle for a task to write the request to the TF and TA registers, 2) a cycle for the RA to forward the request to the XF and XA registers, 3) a cycle for a service module to lock the mutex and to write the return code back to the XA register, 4) a cycle to forward the code to the TA register, and 5) a cycle for the task to read the code to resume execution. Mutex unlock also takes 5 cycles for the task to receive the return code for the request regardless of whether other tasks are waiting for the mutex. It takes extra 3 cycles until the task that has newly acquired the mutex resume to its execution. The synchronization and communication services other than the mutex take the same cycles. The task control calls take longer cycles for error handling.

All the services can be executed well in 150 ns, which is fast enough even for extreme applications.

V. CONCLUSION

Hardware design of RTOS services necessary for full hardware implementation of RTOS-based systems has been proposed in this paper. All the functions of mutexes, event flags, data queues, shared variables, and task control services are implemented as hardware. Wait and release of service requests are efficiently handled by hardware. The resulting hardware is of reasonable size and runs extremely fast.

There is much room for improvement in the proposed architecture. The issues to be addressed are as follows:

- We have also implemented other service modules for synchronization and communication, such as task notification and message buffers. However, dynamic memory allocation has not been implemented yet.
- In our implementation, waiting requests are released only in the order of the task priorities. However, in some RTOS specification, waiting tasks must be processed in the order of their arrival. We are now working on release of waiting requests in the arrival order.
- We have only designed a manager module for up to 4 tasks, but more tasks should be handled. We are now working on automatically generating managers for an arbitrary number of tasks. After finding a bottleneck of our scheme, we will try to improve our architecture to handle the larger number of tasks.

Acknowledgments

Authors would like to express their appreciation to Dr. Hiroyuki Kanbara of ASTEM/RI, Prof. Hiroyuki Tomiyama of Ritsumeikan Univ., Mr. Takayuki Nakatani (formerly with Ritsumeikan Univ.), Mr. Shimpei Tamura (formerly with Kwansei Gakuin Univ.) for their discussion and valuable comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin Univ. 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. 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. RSP '06* pp. 163–168 (June 2006).

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

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

[5] P. Kohout, B. Ganesh, and B. Jacob: "Hardware support for real-time operating systems," in *Proc. CODES+ISSS '03*, pp. 45–51 (Oct. 2003).

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

[7] 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).

[8] N. Ito, N. Ishiura, H. Tomiyama, and H. Kanbara: "High-level synthesis from programs with external interrupt handling," in *Proc. SASIMI 2015*, R1-3, pp. 10–15 (Mar. 2015).

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

[10] Y. Oosako, N. Ishiura, H. Tomiyama, and H. Kanbara: "Synthesis of full hardware implementation of RTOS-based systems," in *Proc. RSP 2018*, pp. 1–7 (Oct. 2018).

[11] W. Nakano, Y. Shinohara, and N. Ishiura: "Full hardware implementation of FreeRTOS-based real-time systems," in *Proc. TENCON 2021*, pp. 435–440, (Dec. 2021).

[12] T. Ando, I. Muguruma, Y. Ishii, N. Ishiura, H. Tomiyama and H. Kanbara: "Full Hardware implementation of RTOS-based systems using general high-level synthesizer," in *Proc. SASIMI 2022* (Oct. 2022).

[13] 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).