Automatic Generation of Management Module for Full Hardware Implementation of RTOS-Based Systems

Hiro Minamiguchi[†]
Grad. School of Sci. & Technol.
Kwansei Gakuin University
Sanda, Hyogo, Japan

Nagisa Ishiura
School of Eng.
Kwansei Gakuin University
Sanda, Hyogo, Japan

Hiroyuki Tomiyama
College of Sci. & Eng.
Ritsumeikan University
Kusatsu, Shiga, Japan

Hiroyuki Kanbara
Res. & Development Div.
ASTEM RI Kyoto
Kyoto, Japan

***Abstract*—This paper presents an automatic scheme for generating hardware that provides RTOS's functions for full-hardware implementation of RTOS-based systems. Though Muguruma and Ando et al. have proposed a method for implementing both tasks of a real-time system and RTOS functions as hardware, the management hardware that provides RTOS functions was designed manually in Verilog HDL. In our method, the manager hardware is automatically generated from a file that describes the configuration of the target system. Manager hardware for different RTOSs can be generated by describing RTOS dependent attributes in the configuration file. Generation of RTL description for unused function is omitted, which reduces the size of the resulting hardware. A prototype system based on the proposed method has been implemented in Perl5, which successfully generated manager hardware modules for 4 tasks for both TOPPERS/ASP3 and FreeRTOS. Manager hardware modules for 4, 8, and 16 tasks have also been generated, which suggests the circuit size and the critical path delay increases in proportion to the number of tasks and their logarithms, respectively.**

***Index Terms*—real-time systems, RTOS, full hardware implementation, hardware design automation**

I. Introduction

With the recent development of information and communication technology, various new services and devices are being developed every day, which in turn require higher functionality to embedded systems. In particular, control of automobile devices and unmanned aerial vehicles requires not only rich functionality but also high response performance. Such real-time systems are developed using a real-time operating system (RTOS). Although the RTOS provides functions to help designers ensure real-time responses, it is becoming increasingly difficult to achieve real-time performance as the systems become more and more sophisticated.

As a method to improve the response performance of systems using RTOS, hardware implementation of RTOS functionality was proposed. Cho [1], Kohout [2], and Vetromille [3] proposed hardware scheduler of RTOS, and Nakano [4] and Maruyama [5] proposed to implement most of the RTOS functions as hardware. However, tasks and handlers were implemented as software, overhead from context switching was inevitable.

As a new approach to address this issue, Oosako [6] has proposed a full hardware implementation scheme, where all the tasks/handlers as well as RTOS functions were realized as hardware. This dramatically improved the response performance of systems with a small number of tasks. Muguruma and Ando [7] has proposed an improved hardware architecture for full hardware implementation that consolidates RTOS service functions duplicated in tasks into management hardware so as to reduce the circuit size. They have also proposed a control method that allows task programs to be synthesized by a commercially available high-level synthesis tool. Minamiguchi et al. have proposed efficient hardware implementation of RTOS services [8].

However, the management hardware which provides the RTOS functions was designed manually in Verilog HDL. In addition, only TOPPERS/ASP3 [9] was assumed as the RTOS.

To address this issue, this paper proposes a method to automatically generate management hardware for the architecture in [7]. In the proposed scheme, the number of tasks, the RTOS services used by tasks, and the other system configuration information are described in a configuration file, from which RTL descriptions of the management hardware are automatically generated. By describing ROTS dependent specification in the configuration file, management hardware for RTOSs other than TOPPERS/ASP3 becomes possible.

We have implemented a system to autogenerate management hardware for both TOPPERS/ASP3 and FreeRTOS [10]. In a preliminary experiment of synthesizing hardware for systems with 4, 8, and 16 tasks, the circuit size and the critical path delay were roughly proportional to the number of tasks and their logarithms, respectively.

II. Full hardware implementation of RTOS-based system

A. Concept

Oosako [6] proposed a method to synthesize all RTOS functions as well as tasks and handlers into hardware. The concept is shown in Fig. 1. The upper figure shows conventional

[†]Currently with Mitsubishi Electric Corporation, Japan

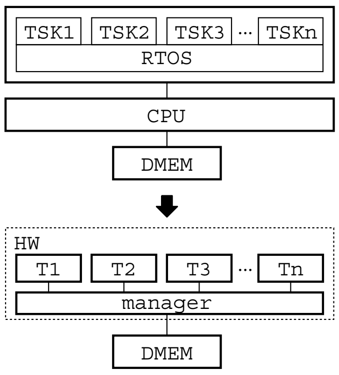

Fig. 1: Concept of full hardware implementation [6]

software implementation of a real-time system, where the tasks (TSKi) are run on a CPU under the control of the RTOS. The lower figure shows full hardware implementation of the above system. Each Ti is an independent hardware module and the RTOS functions are provided by the management hardware (manager).

In this hardware scheme, all tasks are executed in parallel whenever they are ready to run. Tasks are controlled by the manager module, which outputs execution/stop signals based on the states of the tasks. This substantially reduces the complexity of the scheduler in RTOS. Each task module is executed independently, which eliminates the overheads of CPU waiting and context switching. Accelerated execution by hardware further improves the response performance.

B. Hardware architecture

This paper assumes the architecture of [7]. The hardware configuration is shown in Fig. 2. The *STATUS* register in the manager module keeps the status information of each task (its state, current priority, base priority, timer, etc.), and the *WAIT* register holds the information about waiting services of each task. The *mutex*, *event flag*, etc. in the lower part are service modules that provide RTOS service functions. *shared_variable* is a module to read/write shared variables, and *control_task* is a module to provide services such as task activation, sleep, priority change, and so on. To avoid interference among services, only one service is executed at a time. *Request Arbiter* (RA) is a circuit that mediates service requests from multiple tasks based on the priority of each task.

All task modules are run in parallel whenever they are ready. The manager delivers the control signals to instruct each task to run/stop based on the status of each task in the STATUS register.

A task (Ti) requests a service by writing the ID of the service and the necessary arguments into registers TFi and TAi, respectively, and it waits for the completion of the service. RA finds the task with the highest priority by a tree of comparators, and writes the task ID, the service ID, and the arguments into registers XT, XF, and XA, respectively. The service module performs the requested operation and writes the return value into the XA register. The manager notifies

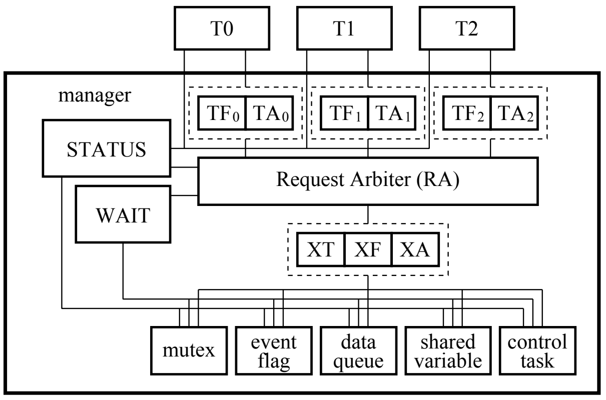

Fig. 2: Hardware architecture of literature [7]

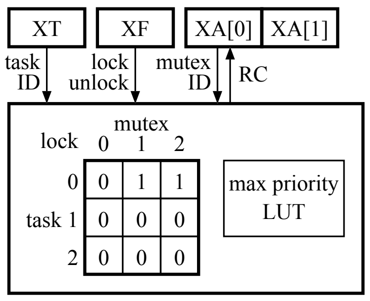

Fig. 3: Mutex module

the task of the completion of the service by copying the value of XA to TAi, then the task reads the return value from TAi and resumes its own processing.

Timeouts are handled by attaching a timer to each task. When a task requests a service with a timeout, the service module sets a timer for the task. The manager counts down the timer every clock, and when the timer hits 0, the manager forces the service module to cancel the service.

Service modules (for mutex, eventflag, dataqueue, etc.) handle multiple instances used in the system in a single module [8]. An example configuration of the mutex module is shown in Fig. 3, where the system runs three tasks (ID=0, 1, 2) which uses three mutexes (ID=0, 1, 2) of the priority ceiling protocol. An array of flags, *lock*, keeps track of which mutex is locked by which task. *Max priority LUT* is a table that calculates the maximum of the ceiling priorities of the mutexes that have been locked by a task.

When a task requests *lock* of a free mutex, the mutex module sets the flag and finds the maximum of the ceiling priorities in the table, updates the current priority of the task with the value, and returns a return code indicating normal completion. If the mutex that a task tries to lock has already been acquired by another task, the task that made this request is placed in the waiting state.

When a task releases an acquired mutex, the mutex module resets the flag, updates the current priority of the task, and wakes up a task that has been waiting for the mutex, if any.

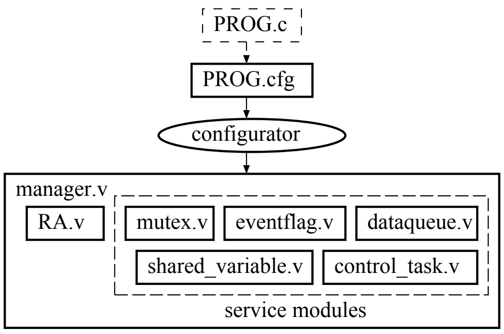

Fig. 4: Flow of hardware generation

III. Automatic generation of management module

A. Overview

This paper presents automatic generation of the manager module in the architecture of [7].

The number of tasks in the system and information on RTOS service modules used by the tasks are described in the configuration file by designers, from which an RTL description of a manager module that provides the required number of ports and service functions is generated. By describing RTOS dependent attributes in the configuration file, such as the min/max values of task priorities, error codes, detailed specification of services, etc., multiple RTOSs are supported. In addition, the size of the resulting circuit is reduced by generating only the necessary part of RTOS service functions.

The flow of hardware generation is shown in Fig. 4. PROG.c is the source code of the real-time system to be designed, and PROG.cfg is the configuration file describing the system configuration. From the information in the configuration file, the configurator generates the management hardware (manager.v), the submodules such as the request arbiter (RA.v) and the services modules (mutex.v, ..., shared_variable.v). Currently, the configuration file is assumed to be written manually, but we plan to generate them from source code in the future.

B. Generation of hardware from system configuration

The configurator in Fig. 4 generates a manager module with the RA modules and service modules as well as the necessary number of ports, STATUS registers, and WAIT modules, according to the number of tasks and RTOS services used by the tasks in the designed system. The configuration file is designed to contain the following information:

- The number of tasks
- The services that the tasks use and their attributes

An example configuration file description is shown in Fig. 5. Line 03 specifies the number of tasks. Lines 05–68 list the service modules used in the system. For each service module, attributes of the service and information for its instances are described. Lines 06–24 are specification of the mutex module. `protocol` in line 07 indicates the mutexes follow the priority ceiling protocol. Lines 09–15 are information of the first of the two instances, which specify ID, service calls invoked on

```
01 system => {
02
03  tasks => 3,
04
05  service_modules => [
06   mutexes => {
07    protocol => 'ceiling',
08    instances => [
09     {
10      id => 0,
11      services => [tloc_mtx, loc_mtx, unl_mtx],
12      tasks => [0, 2],
13      ceiling => 2
14      processing_order => 'priority',
15     },
16     {
17      id => 1,
18      services => [loc_mtx, unl_mtx],
19      tasks => [0, 1, 2],
20      ceiling => 1
21      processing_order => 'arrival',
22     }
23    ]
24   }
25   eventflags => {
26    instances => [
27     {
28      id => 0,
29      services => [set_flg, wai_flg, clr_flg],
30      and_or => 'and',
31      clear => 0
32      processing_order => 'arrival',
33     },
34     {
35      id => 1,
36      services => [set_flg, wai_flg, clr_flg,
37                   twai_flg],
38      and_or => 'and',
39      clear => 1
40      processing_order => 'priority',
41     }
42    ]
43   },
44   dataqueues => {
45    instances => [
46     {
47      id => 0,
48      services => [snd_dtq, rcv_dtq],
49      data => 10
50      processing_order => 'priority',
51     },
52     {
53      id => 1,
54      services => [snd_dtq, rcv_dtq],
55      data => 20
56      processing_order => 'priority',
57     }
58    ],
59   },
60   shared_variables => {
61    services => [read, write],
62    words => 32
63   },
64   control_task => {
65    services => [wup_tsk, sus_tsk, get_pri,
66                 chg_pri]
67   }
68  ]
69   ...
70 }
```

Fig. 5: Example of system specification (PROG.cfg)

this instance, tasks that use this instance, the ceiling of the priority, and the order in which the service is processed [11].

Based on the information on the number of tasks and service modules to be used, the ports and RA circuits are generated. The STATUS register and the registers in the WAIT module are generated in the similar way.

As for the service modules, the size and number of their registers and the functions to be implemented are determined from the number of instances, the number of tasks that uses them, and the set of service calls used in the system.

C. Supporting multiple RTOSs

Support of multiple RTOSs is realized by describing RTOS dependent information in the configuration file.

```
01 system => {
02
03  tasks => 3,
04
05  service_modules => [
06   mutexes => {
07    protocol => ’ceiling’,
08    instances => [
09     {
10      id => 0,
11      services => [tloc_mtx, loc_mtx, unl_mtx],
12      tasks => [0, 2],
13      ceiling => 2
14      processing_order => ’priority’,
15     },
16     {
17      id => 1,
18      services => [loc_mtx, unl_mtx],
19      tasks => [0, 1, 2],
20      ceiling => 1
21      processing_order => ’arrival’,
22     }
23    ]
24   }
25  ],
26
27   ...
28
29  priority => {MAX_PRI => 1, MIN_PRI => 16},
30
31  error_code => [
32   { mutex_unlock_ok => ’E_OK’ },
33   { mutex_other_locked_unlock_error
34                                      => ’E_OACV’ },
35   { mutex_unlocked_unlock_error => ’E_OACV’ },
37   { mutex_timeout => ’E_TMOUT’ },
37   { mutex_lock_ok => ’E_OK’ },
38   { mutex_locked_lock_error => ’E_OBJ’ },
39    ...
40  ]
41 }
```

(a) TOPPERS/ASP3

```
01 system => {
02
03  tasks => 3,
04
05  service_modules => [
06   mutexes => {
07    protocol => ’inheritance’,
08    instances => {
09     {
10      id => 0,
11      services => [tloc_mtx, loc_mtx, unl_mtx],
12      tasks=>[0, 2]
13      processing_order => ’priority’,
14     },
15     {
16      id => 1,
17      services => [loc_mtx, unl_mtx],
18      tasks=>[0, 1, 2]
19      processing_order => ’priority’,
20     }
21    ]
22   }
23    ...
24  ],
25
26  priority => {MAX_PRI => 31, MIN_PRI => 0},
27
28  error_code => [
29   { mutex_unlock_ok => ’pdTRUE’ },
30   { mutex_other_locked_unlock_error
31                                      => ’pdFALSE’ },
32   { mutex_unlocked_unlock_error => ’pdFALSE’ },
33   { mutex_timeout => ’pdFALSE’ },
34   { mutex_lock_ok => ’pdTRUE’ },
35   { mutex_locked_lock_error => ’pdFALSE’ },
36    ...
37  ]
38 }
```

(b) FreeRTOS

Fig. 6: Example of RTOS dependent specification

(1) Task’s priorities

While we may assume that the priorities of the tasks are expressed by integers, the maximum and the minimum values as well as whether the priority is in ascending order or in descending order depend on RTOSs. For example, in TOPPERS/ASP3 the range of the priority is 1 to 16 where 1 means the highest priority, while in FreeRTOS the range is 0 to 31 where 31 means the highest priority.

In our scheme, the maximum and minimum priority values (MAX_PRI and MIN_PRI, respectively) are specified in the configuration file. If MAX_PRI is smaller than MIN_PRI, the priority is interpreted as in ascending order, and vice versa. From the information, RA comparison tree and the registers and wires with the necessary number of bits to handle the priorities are generated.

Fig. 6 (a) is an example description for a system using TOPPERS/ASP3, where the priority information is given in line 29. In the case of FreeRTOS, the priority is specified as in line 26 of Fig. 6 (b).

(2) Attributes of services

Even for the same service, detailed specifications may differ from RTOS to RTOS. For example, in the case of mutex, TOPPERS/ASP3 assumes the priority ceiling protocol, while FreeRTOS uses the priority inheritance protocol. By specifying detailed specification in the configuration file, hardware description that matches the specification is generated.

For example, in the example description for TOPPERS/ASP3 in Fig. 6 (a), the protocol is described as ”ceiling” in line 07 and the ceiling priorities for the two instances are specified in lines 13 and 20. In the example of Fig. 6 (b), protocol is specified as ”inheritance” in line 07. From this information, the RTL description of the mutex module with the specified protocol is generated.

(3) Error codes

The error codes for the services are also different among RTOSs. This kind of differences are also handled via the configuration file.

For example, in Fig. 6 (a), lines 31–40 describes the error codes for the mutex module in TOPPERS/ASP3. `mutex_unlock_ok` in line 32 is the case for successful completion of mutex unlock, which is represented as `E_OK` in TOPPERS/ASP3. In the case of FreeRTOS, the module returns `pdTRUE` for the same case, which is specified in line 29 in Fig. 6 (b).

D. *Pruning of redundant hardware*

In our scheme, unnecessary function in the service module may be omitted so that smaller hardware can be generated. The following situations are assumed:

- Some service calls are not used by any of the tasks.
- Service calls with timeout are not used for some services.

For example, in lines 65 and 66 of Fig. 5, we can see the tasks only call `wup_tsk`, `sus_tsk`, `get_pri`, and `chg_pri` for the control_task module, so RTL description for the other calls can be deleted.

Similarly in lines 48 and 54 in Fig. 5, we see that only `snd_dtq` and `rcv_dtq` are called for the two dataqueue instances. Since the service call for receiving data with timeout is `trcv_dtq`, hardware for timeout processing can be omitted for the dataqueue module.

TABLE I: Synthesis result of management module

(a) TOPPERS/ASP3 and FreeRTOS

	#LUT	#FF	delay [ns]
TOPPERS/ASP3	3,994	2,261	8.59
FreeRTOS	4,212	2,258	8.43

(b) 4, 8, and 16 tasks

#task	#LUT	#FF	delay [ns]
4	5,105	2,399	8.76
8	9,809	4,198	10.95
16	19,104	7,832	13.91

IV. Implementation and Experiment

Based on the proposed method, a configurator for generating Verilog HDL design of management hardware has been implemented in Perl5.

In the first experiment, manager modules following the specifications of TOPPERS/ASP3 and FreeRTOS were generated. The specification of the manager module was as follows:

- 4 tasks.
- A data queue module with 2 instances, each holds 10 data of 4B.
- All the services are processed in the arrival order [11].
- A shared variable module accommodates 32 words of 4B.
- The control_task module supports `wup_tsk`, `sus_tsk`, `get_pri`, `chg_pri` for TOPPERS/ASP3, and `xTaskResume`, `vTaskSuspend`, `vTaskPriorityGet`, `vTaskPrioritySet` for FreeRTOS.

The generated Verilog HDL descriptions were synthesized targeting an FPGA (Artix-7) using Xilinx's Vivado 2016.4. The circuit size and critical path delay of the generated manager module are shown in TABLE I (a). The number of look-up tables and the number of flip-flops are indicated by #LUT and #FF, respectively, for the circuit size. The circuit size and the critical path delay are considered reasonable. Difference of RTOSs showed no significant impact on the size and critical path delay of the synthesized circuits.

In the second experiment, we generated managers for 4, 8, and 16 tasks. It assumes TOPPERS/ASP3 and incorporates the following service modules:

- A mutex module with 2 instances.
- An event flag with 2 instances.
- A data queue module with 2 instances, each holds 10 data of 4B.
- A shared variable module which accommodates 32 words of 4B.
- A control_task module which supports the following 12 services:

  `act_tsk`, `can_act`, `ter_tsk`, `chg_pri`,
  `get_pri`, `wup_tsk`, `can_wup`, `rel_wai`,
  `sus_tsk`, `rsm_tsk`, `loc_cpu`, `unl_cpu`

The results of logic synthesis are shown in TABLE I (b). The circuit size is approximately 1.9 times larger when the number of tasks is doubled, and approximately 3.7 times larger when the number of tasks is quadrupled, indicating that the circuit size increases roughly in proportion to the number of tasks.

The critical path delay increased by approximately 2ns and 5ns when the number of tasks was doubled and quadrupled, respectively, and increased approximately in proportion to the logarithm of the number of tasks. We guess that the increase is caused by the number of stages in the comparison tree in RA.

The circuit size and critical path delay of the manager module for 16 tasks are a little too large for practical use. Since there is much room for reducing both the circuit size and critical path delay, we will continue improving or optimizing the design.

V. Conclusion

In this paper, a method to generate management hardware from configuration information of real-time systems has been proposed. The method supports multiple RTOSs and reduces the circuit size by generating only the necessary functions of the RTOS.

Since the automatic generation of the management hardware has revealed the increase of circuit size and critical path delay relative to the number of tasks, identification of bottlenecks and optimization of circuits are the next issues to be addressed. Other future work include automatic generation of the configuration file from a given program source code and application of this hardware scheme to design of practical real-time systems.

Acknowledgment

Authors would like to express their appreciation to Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their discussion and valuable comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin University. This work was partly supported by JSPS KAKENHI under Grant No. 19H04081, 20H00590, and 21K19776.

References

[1] Y.-C. Cho, S.-J. Yoo, K.-Y. Choi, N.-E. Zergainoh, and 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] P. Kohout, B. Ganesh, and B. Jacob: "Hardware support for real-time operating systems," in *Proc. International Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 45–51 (Oct. 2003).

[3] 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 Symposium on Rapid System Prototyping (RSP 2006)*, pp. 163–168 (June 2006).

[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. SASP 2010*, pp. 58–63 (June 2010).

[6] 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).

[7] Takuya Ando, Iori Muguruma, Yugo Ishii, Nagisa Ishiura, Hiroyuki Tomiyama, and Hiroyuki Kanbara: "Full Hardware Implementation of RTOS-Based Systems Using General High-Level Synthesizer," in *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2022)*, pp. 2–7 (Oct. 2022).
[8] Hiro Minamiguchi, Masaki Nakahara, Yugo Ishii, Yukino Shinohara, Iori Muguruma, and Nagisa Ishiura: "Hardware RTOS Services for Full Hardware Implementation of RTOS-Based Systems," in *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2022)*, pp. 14–19 (Oct. 2022).
[9] TOPPERS/ASP kernel, https://www.toppers.jp/ (accessed 2023-01-07).
[10] FreeRTOS kernel, https://www.freertos.org/ (accessed 2023-01-07).
[11] N. Ishiura, H. Kanbara, and H. Tomiyama: "Arrival Order Processing of Service Requests in Full Hardware Implementation of RTOS-Based Systems," in *Proc. International Technical Conference on Circuit/Systems, Computers and Communications (ITC-CSCC 2023)* (June 2023).