Electronic Preprint for Journal of Information Processing Vol.22 No.4

Regular Paper

TECS Components Providing Functionalities of OSEK Specifications for ITRON OS

Atsushi Ohno[1] Takuya Azumi[2,a)] Nobuhiko Nishio[1,b)]

Received: November 13, 2013, Accepted: May 17, 2014

Abstract: The number of electronic control units (ECUs) has increased to manage complicated vehicle systems. Many kinds of operating systems that run on ECUs exist: ITRON OS, OSEK OS, and so forth. Currently, developers implement the system control software according to the ITRON and OSEK specifications independently. For example, even though OSes provide similar functionalities, OSEK specifications have several differences from ITRON specifications such as scheduling policies (Non-preemptive scheduling), alarms, hook routines, and several system calls. Thus, when using legacy software following OSEK specifications on the ITRON OS, developers have to port the software to ITRON OS. This paper presents a component-based framework to fill the gap between OSEK and ITRON specifications by using TECS (TOPPERS Embedded Component System). The work required to port legacy OSEK applications built with TECS components to ITRON applications built for TECS is reduced by using our method. TECS is a high-level abstraction component system for enhancing the reusability of software. Examples for the characteristics of the framework are: (1) Non-preemptive scheduling tasks are implemented by changing the priority of the task to the highest priority; (2) The system works as the OSEK alarm based on a counter value, which is incremented at an arbitrary time interval; (3) OSEK hook routines are also available with a particular timing. Experimental results demonstrate that the overhead of the corresponding system calls compared to the original OSEK system calls is reduced to within 13.58 μsec.

Keywords: automobile, ITRON, OSEK, real-time OSes, component based development

1. Introduction

Recently, a number of automobiles systems have changed from mechanical to electronic control systems. For example, engine control and injection pumps were used to pump fuel into the cylinders, while electronically-controlled fuel injection is now used to make this process efficiently. In addition, hybrid cars and battery-powered cars are becoming prevalent, which has increased the rate of computerization of automobiles. As a result, automobile systems have become more complicated and larger in scale, as evidenced by the code used in such automobiles; for example, for Volt, manufactured by General Motors, the lines of code increased from 2.4 million to 6 million between 2005 and 2009 [1]; as of late, it has further increased to approximately 10 million [2].

To manage the complicated automobile systems, various operating systems [3] such as ITRON [*1] OS [4], [5], OSEK [*2] [6], and AUTOSAR OS [*3] OS [7] are used. Among them, the ITRON specifications, which include automotive control profiles, are now the de-facto standard for embedded systems in Japan [8]. On the other hand, OSEK OS and AUTOSAR OS [*4] are the architecture of choice for distributed control units in the automotive industry that enable more efficient development using a standardized software interface.

Currently, complicated automobiles are composed of a number of ECUs, each of which may use a different OS. In other words, multiple types of OSes are used (e.g., ITRON OS and OSEK OS) in a single automobile. Then, developers have to implement the system control software following the ITRON specifications and OSEK specifications independently because these OSes provide similar functionalities, but also have several differences, such as scheduling policy, counters, alarms, resources, events, hook routines, and system calls [9]. In the worst case, the software requires substantial changes, forcing developers to redesign it almost from the scratch. Generally, it is difficult and time-consuming to port software on a different OS because the software needs substantial changes, e.g., operable system calls, design of tasks in the application and implementation methods – this requires great design cost.

To enhance the reusability of software, component-based development is getting attention in the area of embedded software development. Various component systems for embedded systems have been proposed such as Koala [10], THINK [11], SaveCCM [12], and TECS [*5] [13], [14]. Use of such component systems is one of the ways to develop embedded system effi-

[1] Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[2] Osaka University, Toyonaka, Osaka 560–8531, Japan
a) takuya@sys.es.osaka-u.ac.jp
b) nishio@cs.ritsumei.ac.jp

[*1] Industrial TRON
[*2] Offense Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug/Vehicle DistributedeXecutive
[*3] AUTomotive Open System ARchitecture
[*4] AUTOSAR OS is based on the OSEK specifications with backward compatibility.
[*5] TOPPERS Embedded Component System

© 2014 Information Processing Society of Japan

ciently by dividing the software into sub-systems. This can increase software reusability and clarify architectures.

This paper presents a component-based framework which provides the functionalities of the OSEK specifications using TECS on ITRON OS. We aim to reduce work by reusing legacy OSEK software on the ITRON OS as the first step. This framework supports unsupported OSEK functionalities which do not have corresponding functionalities in ITRON OS. The unsupported OSEK functionalities are provided by preparing components and/or functions with ITRON functionalities and system calls on ITRON OS. In addition, TECS also exploits the aforementioned advantages of component systems such reusability. Therefore, the proposed framework can reduce the porting cost of applications that run on the different OSes.

The remainder of this paper is organized as follows. Section 2 briefly reviews related work, and TECS is explained in Section 3. Section 4 defines problems of our work. Requirements, design, and overview of the proposed framework are presented in Sections 5–7, respectively. Examples are presented in Sections 8. Then, the proposed framework is evaluated in Section 9, and finally Section 10 summarizes this paper.

2. Related Work

Component-based development has started to be widely used in embedded software development as follows; THINK [11], SaveCCM [12], [15], and SmartC [16].

THINK [11] is a software framework for implementing operating system kernels from components. The THINK software framework is constructed with a small set of concepts; components, interfaces, bindings, names, and domains.

SaveCCM [12], [15] is a component model used for embedded control applications in vehicular systems. In an effort to facilitate analysis, SaveCCM has limited flexibility in particular analysis of dependability and real-time.

SmartC [16] is a language for automotive electronics applications, such as engine control systems. SmartC has module level, task level, subtask level, and component level hierarchical models and implements the SmartOSEK [17] operating system, which is based on the OSEK specifications model.

However, these component technologies target only one OS and do not support the reuse of components from another OS.

Recently, the number of electronic control units in automotive control systems has been increasing because advanced electronic control functions requirements are becoming more complex. Several approaches have been proposed for the integration of ECUs into a single high-performance ECU and the unification of applications thereon in order to reduce the number of ECUs. DUOS [18] is a new real-time operating system framework for this purpose. The framework has a hierarchical scheduler and API layers for ITRON and OSEK specifications. This OS has APIs based on ITRON and OSEK specifications, although the hardware requires high performance and the kernel size is larger than the OS.

eCos [19] and Xenomai [20] are providing APIs of other OSes. In the framework, wrappers are used to provide APIs of other OSes. However, the common existing OSes for automobiles cannot be used for the OSes.

3. TOPPERS Embedded Component System

In this section, the specifications of TECS are described.

3.1 TECS Component Model

A *cell* is a component in TECS, and has *entry port* and *call port* interfaces. The *entry port* is an interface to provide functionalities to other *cells*. The *call port* is an interface to use the functionalities of other *cells*. In this environment, a *cell* communicates with these interfaces. The *entry port* and the *call port* have service sets. A *signature* is the definition of the interfaces in a *cell*. The *celltype* define *cells*, like an object-oriented language Class. A *cell* is an entity of a *cell type*. **Figure 1** shows an example of the connection of *cells*. TECS component model is explained in detail in Ref. [21].

There are two *cells*; the left *cell* is an A *cell* and the right *cell* is a B *cell*. Here, tA and tB represent the *celltype* names. The triangle in the B *cell* depicts an *entry port* (eEntryPort). The connection of the *entry port* in the A *cell* describes a *call port* (cCallPort). A *call port* can be only connected to an *entry port*. Therefore, in case of joining several *entry ports*, a *call port* array can be used. An *entry port* also can be connected to several *call ports*. These ports are connected by the *signature* description to define a set of function heads.

3.2 Development Flow

Figure 2 shows the proposed development flow in TECS. TECS CDL [*6] is constructed with three descriptions; *signature* description, *celltype* description and *build* description.

The *signature* description is used to define a set of function

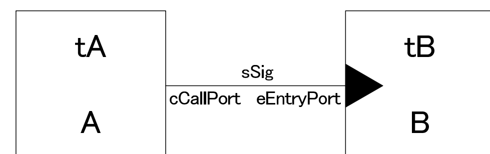

Fig. 1 Example of *cells*.

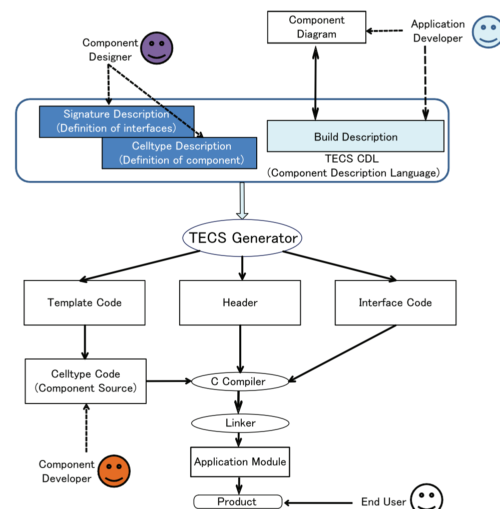

Fig. 2 Development flow in TECS.

[*6] Component Description Language

© 2014 Information Processing Society of Japan

heads. The *celltype* description is used to define the *entry ports*, *call ports*, *attributes*, and *variables* of a *celltype*. The *build* description is used to declare the *cells* and create the connections between *cells* in order to assemble an application. A *TECS generator* generates several interface C- (.h or .c) and template code for the component source from the *signature*, *celltype*, and *build* descriptions. In case of using the OSEK OS, OIL [*7] descriptions are also generated from the TECS CDL.

Developers in this framework are divided into three groups; a component designer, component developer and an application developer. The component designer defines the *signatures* and *celltypes*. The component developer implements components as well as write the implementation code (Component Source) of the *cells* by using template code which is generated from TECS CDL description with the *TECS generator*. Generally, a component is provided by the source code. After the *celltype code* is developed, the application developer implements the build description by connecting the *cells*. An appropriate application is composed of generated code (header and interface code) and the *celltype code*.

4. Problem Definition

There are many kinds of operating systems that run on ECUs; ITRON OS, OSEK OS, AUTOSAR OS, and so forth. Therefore, developers have to implement applications following these specifications individually. In such situations, developers need to port legacy applications on the other OS. For instance, even though OSes provide similar functionalities, some unsupported OSEK functionalities are exist on the ITRON OS such as scheduling policies, alarms, hook routines, and several system calls. A list of unsupported OSEK functionalities on the ITRON OS is:

Scheduling Policy
OSEK specifications has Non-preemptive Scheduling and Mixed-preemptive Scheduling. However, these scheduling policies are not supported on ITRON specifications.

Counter
The OSEK counter is a time count value which is incremented at arbitrary time intervals by the user. However, ITRON specifications do not have a similar time count value.

Alarm
The OSEK alarm works with an arbitrary time based on the time count value. Furthermore, the alarm will perform actions at arbitrary time intervals as specified by the user. An action is chosen from three functions; calling an alarm callback routine, activating a task, or setting an event. However, the OSEK alarm is different from ITRON specifications.

Resource
The OSEK resource changes its own task priority and/or the ISR [*8] priority to a *ceiling* priority. The *ceiling* priority is the highest priority in a group of tasks and ISR, which belonging to the resource, when the task or ISR receive the resource. However, the OSEK resource is not supported on ITRON specifications.

Event
The OSEK event of the SetEvent and GetEvent is called with the task ID, but *set_flg* and *ref_flg* on the ITRON specifications are called using the flag ID.

Application Mode
The OSEK application mode provides a function to change the set of tasks and alarms, which are automatically started after the OS is started. However, the OSEK application mode is not supported on ITRON specifications.

Hook
Four of the OSEK hook routines [*9] correspond to the component level; StartupHook, ShutdownHook, PreTaskHook, and PostTaskHook. However, the OSEK hook routines are not supported on ITRON specifications.

System Calls
Some of the OSEK system calls are not supported on the ITRON specifications. The corresponding OSEK System Calls on ITRON specifications is shown in **Table 1** [*10]. For example, SetEvent and GetEvent are different because the system calls need to specify the task ID on OSEK specifications. The circle symbol indicates that the system call of the ITRON OS has correspondence to the system call of OSEK OS. The cross symbol indicates that the system call has restrictions.

Hence, it is generally difficult to change applications running on a different OS because the applications needs to substantially change as follows; difference in the operable system calls, design of tasks in the applications and implementation methods. In the worst case, the applications need substantial changes and developers need to redesign the applications. As a consequence, the amount of work and the cost will be increased.

5. Requirement

Requirements to provide functionalities of the OSEK specifications on the ITRON OS are listed as below.

(1) Providing the corresponding OSEK functionalities on the ITRON OS
It is hard to port legacy OSEK applications on the ITRON OS because of the unsupported OSEK functionalities which do not have correspondence in ITRON specifications. Therefore a developer has to consider correspondence of the unsupported OSEK functionalities on the ITRON OS. As a consequence, the unsupported OSEK functionalities are needed to provide the corresponding OSEK functionalities on the ITRON OS.

(2) The existing ITRON OS is used without modification
Developers must verify the ITRON OS again if the existing ITRON OS is changed to provide the unsupported OSEK functionalities. In addition, developers have to change the existing ITRON OS each time and the work will be increased. Therefore, the OSEK functionalities are provided on the existing ITRON OS without modification.

*7 OSEK Implementation Language
*8 Interrupt Service Routine
*9 Hook routines execute user-defined actions at the specific time such as before execution of a task or after execution of a task.
*10 ClearEven and WaitEvent are categorized in the supported system call because these system calls do not need to specify Task ID.

© 2014 Information Processing Society of Japan

Table 1 Corresponding OSEK system calls in the ITRON specifications.

Service	Task	ISR category 1	ISR category 2	ErrorHook	PreTaskHook	PostTaskHook	StartupHook	ShutdownHook	alarmcallback
ActivateTask	○		○						
TerminateTask	○								
ChainTask	×								
Schedule	×								
GetTaskID	○		○	×	×	×			
GetTaskState	○		×	×	×	×			
DisableAllInterrupts	×	×	×						
EnableAllInterrupts	×	×	×						
SuspendAllInterrupts	×	×	×	×	×	×			×
ResumeAllInterrupts	×	×	×	×	×	×			×
SuspendOSInterrupts	×	×	×						
ResumeOSInterrupts	×	×	×						
GetResource	×		×						
ReleaseResource	×		×						
SetEvent	×		×						
ClearEvent	○								
GetEvent	×		×	×	×	×			
WaitEvent	○								
GetAlarmBase	×		×	×	×	×			
GetAlarm	×		×	×	×	×			
SetRelAlarm	×		×						
SetAbsAlarm	×		×						
CancelAlarm	×		×						
GetActiveApplicationMode	×		×	×	×	×	×	×	

○ indicates that the system call has one-to-one correspondence to the system call.
× indicates that the achieved system call has restrictions.

(3) The applications do not need substantial changes.
Developers need to redesign the applications if the existing applications need to substantially change in order to provide the unsupported OSEK functionalities. Therefore, we need to provide the unsupported OSEK functionalities on the ITRON OS as mush as possible to reduce the work required.

6. Policy for Design about Component

The purpose of our research is to reduce the amount of work by providing OSEK functionalities on the targeted ITRON OS. Hence, OSEK functionalities are provided as components without changes to the ITRON OS. For instance, "ChainTask" of the OSEK system call is replaced with the prepared function in the *signature* description. In the function, "act_tsk" of the ITRON system call is called and then "ext_tsk" of the ITRON system call is called to provide the same functionalities as "ChainTask."

The OSEK functional specifications are provided by adding new components to the existing ITRON OS with TECS components in **Fig. 3**. The application is running on TECS components for OSEK functionalities. TECS components use the ITRON OS via the TECS components for the ITRON OS and SIL. SIL is the System Interface Layer for enhancing driver portability such as *SIL_UNL_INT* [*11] and *SIL_LOC_INT* [*12].

We classified the implemented new components into two groups: supported functionalities and unsupported functionalities. The supported functionalities are the OSEK functionalities which have the corresponding functionalities on the ITRON OS. The unsupported functionalities are not the OSEK functionalities which do not have corresponding functionalities. The unsupported OSEK functionalities are provided by preparing components or functions with functionalities and system calls on the ITRON OS. For instance, non-preemptive scheduling tasks are implemented by changing the priority of the task to the highest priority.

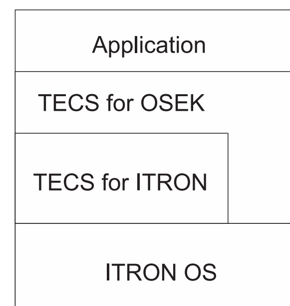

Fig. 3 Structure of TECS components for OSEK functionalities.

The components for providing the OSEK functionalities which the ITRON OS does not have are implemented by the component designer as the *signature* description and the *celltype* description. The application developer implements an appropriate application by connecting the new *cell* in the *build* description based on the component diagram. Finally, the component developer implement components the source by using ITRON functionalities.

In the development of the OSEK OS with TECS, the OIL description is generated from TECS CDL. Therefore, developer does not need to take into consideration the OIL description. In

*11 *SIL_UNL_INT* resets a flag to enable all interrupts.
*12 *SIL_LOC_INT* sets a flag to disables all interrupts except for non-maskable interrupts.

© 2014 Information Processing Society of Japan

case of ITRON specifications, static APIs of ITRON is generated instead of the OIL description.

7. Overview of Proposed Framework

We apply TOPPERS/ATK1 [22] (Automotive Kernel1) functionalities to TOPPERS/ASP [23] (hereinafter referred to as the ASP kernel) with TECS [24]. TOPPERS/ATK1 is a TOPPERS OS kernel profile and compliant with OSEK specifications, and TOPPERS/ASP is a TOPPERS OS kernel profile that is compliant with the ITRON OS. TECS is a high-level abstraction component system, as mentioned above.

7.1 Supported Functionalities

The supported functionalities are the OSEK functionalities which have the corresponding functionalities on the ITRON OS. The correspondences between OSEK and ITRON functionalities are provided below.

7.1.1 Conformance Classes

The conformance classes on OSEK specifications are BCC1, BCC2, ECC1, or ECC2. The conformance class of ATK1 is ECC2 and ASP corresponds to ECC2 according to OSEK specifications. Therefore, the corresponding conformance classes between ASP and ATK1's conformance class are the same.

7.1.2 Task States

Correspondence task states are shown in **Table 2**. Each task state on the ITRON specifications corresponds to the task state on the OSEK specifications. The ITRON task state of BLOCKED is classified into three sub-states; WAITING, SUSPENDED, and WAITING-SUSPENDED. In the case of the OSEK functionalities on the ITRON OS, the difference is not a problem. Therefore, the OSEK state of waiting corresponds to the ITRON task state BLOCKED.

7.1.3 System Calls

The corresponding system calls are shown in **Table 3**. The system calls have a one-to-one correspondence between OSEK system calls and ITRON system calls. Thus, these ITRON system calls are used in place of the OSEK system calls.

7.2 Unsupported Functionalities

The OSEK functionalities which are not supported by the ITRON OS provided by preparing components and/or functions with functionalities and system calls on the ITRON OS. A list of the unsupported OSEK functionalities on the ITRON OS is summarized below.

7.2.1 Non-preemptive Scheduling

The OSEK task has three scheduling policies; full preemptive scheduling, non-preemptive scheduling, and mixed preemptive scheduling. The ITRON scheduling policy corresponds to full preemptive scheduling. In addition, if preemptive and non-preemptive tasks are mixed on the same system, the context is referred to as mixed preemptive scheduling. Therefore, we accommodated non-preemptive scheduling by changing the priority of the task. Before a new task is executed, a system call is invoked to change the priority of the task that has the highest priority.

7.2.2 Counter

A component that is activated by the cyclic handler is used to count time at arbitrary time intervals. Therefore, the period in the cyclic handler is an arbitrary time span set by the user. Accordingly, the time count in the component is incremented at arbitrary time intervals set by the user to provide the function of the OSEK counter on the ITRON OS.

7.2.3 Alarm

The alarm component on the ITRON OS is implemented as follows. An alarm component consists of a counter component to work on the time value in the counter component. An alarm component is called after the time value is incremented in a counter. Moreover, the counter component passes the counter value as an argument.

In the alarm component, the first alarm expiration time and the period are defined by the user. According to the user-defined time interval, the next alarm expiration time is compared to the time value of the counter. If the times are the same, the alarm will execute two processes. In the first process, the alarm expiration time is changed to the next alarm expiration time. The next alarm expiration time is the previous alarm expiration time plus the period. In the second process, the user-defined action is executed.

As mentioned above, the action is the alarm callback routine, an activating task, or setting of an event. Each action is implemented as a component, and one of the components of the action is connected to the alarm component. Therefore, by changing the component connected to the action, the user can select the process of the action.

7.2.4 Resource

We designed a component that will change the original priority of the task to a ceiling priority which is automatically chosen by the *TECS generator*. If *getResource* is called, the priority of the task is changed to the ceiling priority. The highest priority among the tasks which use the resource must be automatically chosen as the ceiling priority by the *TECS generator*.

The original task's priority is saved in the resource component, in case the resource is nested before the task's priority is changed. When *releaseResource* is called, the priority of the task will be the saved priority. In addition, if the resource is a linked resource, the ceiling priority will be compared to the ceiling priority of the linked resource. Therefore, the priority of the task is changed to the higher priority.

In case of internal resources, the task uses the same mechan-

Table 2 Corresponding task states.

OSEK Specifications	ITRON Specifications
running	RUNNING
ready	READY
waiting	BLOCKED
suspended	DORMANT

Table 3 Corresponding system calls.

OSEK Specifications	ITRON Specifications
ActivateTask	act_tsk / iact_tsk
TerminateTask	ext_tsk
GetTaskID	get_tid / iget_tid
GetTaskState	ref_tsk
ClearEvent	clr_flg
WaitEvent	wai_flg
ShutdownOS	ext_ker

© 2014 Information Processing Society of Japan

ics as the non-preemptive task, as mentioned in Section 7.2.1, and receives and releases the resource accordingly. In addition, the OSEK resources have RES_SCHEDULER. If a task receives RES_SCHEDULER, the task protects itself against preemptions by the other tasks. The state of receiving RES_SCHEDULER corresponds to transitioning the system state to the dispatching disable state.

The OSEK resource functionality between tasks is provided with the proposed framework, although the proposed framework for the OSEK resource has a restriction between tasks and ISRs, or ISRs and ISRs. The proposed framework for the OSEK resources such avoid the restrictions, which is a resource between tasks and ISRs, or between ISRs and ISRs, as discussed in Section 7.2.9.

7.2.5 Event

We designed a component that will call the Event control system calls through the task component. In this case, an event control task *celltype* includes two *cells*; a task *cell* and an event control body *cell*. The task *cell* works as usual.

On the other hand, the event control body *cell* calls *SetEvent* or *GetEvent* through an event control task to specify the event with a task ID.

7.2.6 Application Mode

Instead of the OSEK application mode, a task for the application mode on the ITRON OS is prepared for execution after the initialization process has finished. The components of the task for the application mode are constructed in two parts. In the first component, the component corresponding to an application mode is called. Then, a set of tasks and alarms is activated in the second component. By preparing several patterns of the second components, there are choices regarding the application modes on the ITRON OS.

7.2.7 Hook

A component that provides the StartupHook function is added to the last part of the initialization process. In a similar manner, a component that provides the ShutdownHook function is added to the first part of the termination process. A component that is executed after the OS initialization has finished provides the StartupHook function. A method as mentioned for the ApplicationMode in Section 7.2.6 is used as the StartupHook function.

In the case of PreTaskHook, components that provide user-defined routines before a component of a task are added. The *through* keyword [25] is then used to insert the component. In the component, the user-defined routines is executed and then the component of the task is executed.

PostTaskHook is implemented by preparing a component. The component is called after a task is executed. In the component for PostTaskHook, the user-defined routines of PostTaskHook is executed and then terminates the task.

7.2.8 System Calls

Some of the OSEK system calls are not supported on the ITRON OS. For example, SetEvent and GetEvent are different because the system calls need to specify the task ID for the OSEK specifications. Therefore, the unsupported OSEK system calls which are shown in Table 1 are supported by preparing functions with system calls on the ITRON OS. The functions are defined in the *signature* description.

ChainTask

The specified task is activated by calling *act_tsk* and then *ext_tsk* to terminate the caller of the task.

Schedule

A task is rescheduled by making a system call, which changes the priority of the task. The priority of the task is changed to the user defined original priority, when *Schedule* is called. After the other tasks are executed and control returns to the task that made the system call *Schedule*, the system call changes the priority of the task to the highest priority. Then, the remainder of the task that made the system call *Schedule* is executed.

DisableAllInterrupts

In this system call, the caller calls *SIL_LOC_INT* and disables all interrupts if the task does not disable all interrupts by *SuspendAllInterrupts* and OS interrupts by *SuspendOSInterrupts*. In addition, this system call does not allow nesting.

EnableAllInterrupts

The caller calls *SIL_UNL_INT* to enable all interrupts if *DisableAllInterrupts* is called and all interrupts are disabled.

SuspendAllInterrupts

In this system call, the caller calls *SIL_LOC_INT* and disables all interrupts if the task does not disable all interrupts by *DisableAllInterrupts* and OS interrupts with *SuspendOSInterrupts*. Moreover, this system call allows nesting. Therefore, a variable with which to count the number of nests is prepared.

ResumeAllInterrupts

In the first step, the caller checks the number of nests if *SuspendAllInterrupts* is called and all interrupts are disabled. In addition, the caller calls *SIL_UNL_INT* to enable all interrupts if there is no nesting.

SuspendOSInterrupts

In this system call, the caller calls the system call that transmits the CPU locked state [*13]. In addition, this system call disables OS interrupts if the task has not disabled all interrupts by *DisableAllInterrupts* and *SuspendAllInterrupts*. Moreover, this system call allows nesting. Therefore, a variable with which to count the number of nests is required.

ResumeOSInterrupts

In the first step, the caller checks the number of nests, if *SuspendOSInterrupts* is called and OS interrupts are disabled. Moreover, the caller calls the system call to transmit the CPU unlocked state [*14] to enabled OS interrupts if there is no nesting.

SignalCounter

This system call adds a defined value to the counter value.

GetAlarmBase

This system call returns the user-defined minimum cycle of the alarm, the user-defined cycle of the counter, and maximum value of the counter.

GetAlarm

This system call returns the remainder of the difference between the current counter value and the next alarm expiration time.

[*13] All interrupts except for non-kernel interrupts are disabled in the CPU locked state.

[*14] All interrupts are enabled in the CPU unlocked state.

© 2014 Information Processing Society of Japan

SetRelAlarm

This system call sets the next alarm expiration time and alarm cycle if the alarm control flag is false. The value of the next alarm expiration time is specified by the relative time. The alarm control flag is then changed to true.

SetAbsAlarm

This system call sets the next alarm expiration time and alarm cycle if the alarm control flag is false. The value of the next alarm expiration time is specified by the absolute time. The alarm control flag is then changed to true.

CancelAlarm

The alarm control flag is changed to false.

ErrorHook

An error code is passed to the function as an argument. This system call is used for centralized error handling.

GetActiveApplicationMode

This system call returns the application mode, which is set by the application mode mentioned in Section 7.2.6.

GetResource

The priority of the task is changed to a ceiling priority, which is automatically decided by the *TECS generator*. The original task's priority is saved in the resource component in case of the nesting of resources, before the task's priority is changed.

If the resource is a linked resource, the priority is compared to the priorities of the other linked resources. Therefore, the priority of the task is changed to a higher priority.

ReleaseResource

The priority of the task is changed to the saved priority.

GetRES_SCHEDULER

When *GetRES_SCHEDULER* is called, the caller of the task is transitioning the system state to the dispatching disable state instead of the received *RES_SCHEDULER*.

ReleaseRES_SCHEDULER

When *ReleaseRES_SCHEDULER* is called, the caller of the task transitions the system state to the dispatched enable state instead of releasing *RES_SCHEDULER*.

SetEvent

The *set_flg* is called via an event control task to specify the event with task ID.

GetEvent

The *ref_flg* is called via an event control task to specify the event with task ID. The bit pattern of the event flag is saved as flgptn in the argument of the event flag state packet pointer.

7.2.9 System Calls from ISR

System calls which are called from a task and an ISR are different on ITRON specifications. In the ITRON OS, corresponding system calls such as *GetTaskState* and *GetEvent* are not available from ISR with the proposed framework because *ref_tsk* and *ref_flg* are not available from ISR. A task having the highest priority is activated to call the system calls instead from the ISR. In the activated task, the system calls are executed instead.

The activated task must be executed immediately because the system calls should be called with higher priorities than the activated task. If other higher priority tasks are already activated, the activated task which made the system calls cannot be executed immediately. In this situation, *irot_rdg* is called until the activated task enters the running state, before the task exits from the ISR. In addition, the task may be interrupted by a lower-priority ISR than the activated ISR. Accordingly, the interrupt priority mask is changed to the priority of the ISR that the task is called from, after the task execution begins.

8. Example of Proposed Framework

In this section, examples of the proposed framework is described. OSEK functionalities of non-preemptive scheduling and application mode are described as the important unsupported OSEK functionalities on the ITRON OS.

8.1 Example of Non-preemptive Scheduling

In this section, an example of non-preemptive scheduling components that provide the unsupported OSEK functionality of non-preemptive scheduling is described.

8.1.1 Mechanism of Non-preemptive Scheduling

Non-preemptive scheduling is accommodated by changing the priority of the task that has the highest priority on the ITRON OS.

An example non-preemptive task flow is shown in **Fig. 4**. In this case, there are two tasks. Task 1 has a higher priority, and Task 2 has a lower priority. Moreover, both tasks are non-preemptive tasks.

First, Task 2 is activated. A system call to change the priority of the tasks is invoked when Task 2 is executed. The priority of the task is then changed to the highest priority. Second, Task 1 is activated ((1) in Fig. 4), although the priority of Task 1 is lower than the new priority of Task 2. Therefore, Task 2 is not preempted. When Task 2 is terminated ((2) in Fig. 4), the priority of the task is changed to the original priority. The OSEK non-preemptive scheduling corresponds to this mechanism.

8.1.2 Component of Non-preemptive Scheduling

We implemented a non-preemptive scheduling policy with *through components*. The construction of a component of a non-preemptive task is shown in **Fig. 5**. As usual, the task component

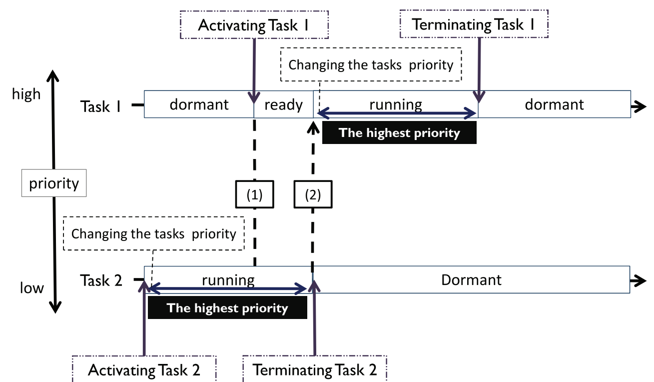

Fig. 4 Non-preemptive task flow.

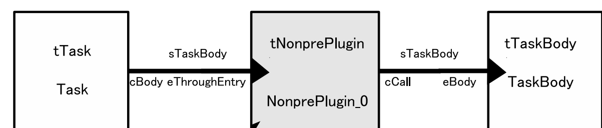

Fig. 5 Structure of non-preemptive task.

© 2014 Information Processing Society of Japan

is activated by the other components. Then, the main component provides the action of the task, when the new task is executed.

This mechanism uses the *through* keyword [25] to insert a component to change the priority of the task. In the component, a system call to change the priority of the task that has the highest priority is invoked. After that, the main action of the task is executed.

8.2 Example of Application Mode

In this section, an example of the application mode components that provide the unsupported OSEK functionality of the application mode is described.

8.2.1 Mechanism of Application Mode

In the OSEK, an application mode is a set of tasks and alarms which are activated automatically at the time of the executing application. The set of tasks and alarms for the application mode are selected by the developer in the configuration file. The OSEK functionality of an application mode is accommodated by a number of components on the ITRON OS.

Two types of components are prepared: (1) a component to choose the application mode and (2) a component that activates a specified set of tasks and alarms. The first component is a task that is executed after the initializing process is finished. In this component, an application mode that the user wants to use is chosen. Then, all of the tasks and alarms that belong to the application mode are activated in the second component. In addition, the name of the used application mode is registered.

8.2.2 Components of Application Mode

An example of the components to provide the OSEK functionality of the application mode is shown in **Fig. 6**. When the components are used, tasks and alarms must not be activated automatically after the OS has started.

The task is executed immediately after the OS is started. The components of the application mode that an user defines are connected to the first component (component of the StartOSBody in Fig. 6). A call port array is used to connect the components. The number of components that are connected with the same *signature* varies with the call port array that is used.

In the first component, the component of the application mode that the user wants to use is called. In the second component, all of the tasks and alarms that are connected to this component are

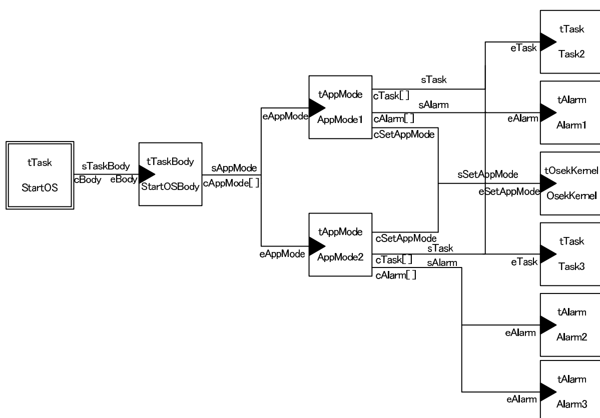

Fig. 6 Structure of application mode.

activated through system calls. In this case, two call port arrays are used to switch the number of components that are connected. One of the call port arrays is for task components that are activated by the system call *act_tsk*, and the other call port array is for alarms that are started by changing the alarm control flag to true. In addition, the call array port is defined with the keyword *optional* in order to allow the situation in which no component is connected. Finally, the name of the used application mode is registered to the component with the name *OSEK kernel*.

9. Evaluation

In this section, we describe the overhead of the applicable functionalities that provide the OSEK system call functionalities and the restrictions of implemented components.

9.1 Experimental Environment

The experimental environment is shown in **Table 4**. The experiment was tested using MINDSTORMS NXT, provided by LEGO. In addition, ASP is used as the ITRON OS, and ATK1 is used as the OSEK OS.

9.2 Overheads of System Calls

The OSEK system calls do not correspond one-to-one with the ITORN system calls. We implemented these functionalities and measured the overhead of the targeted OSEK system calls in the evaluation program. The execution time of OSEK system calls using the proposed framework on the ITRON OS and the OSEK OS by using TECS were measured on NXT. The resulting overheads of the applicable functionalities are shown in **Table 5**. As mentioned above, TECS is used as a component system.

Each functionality was tested one hundred times, and the average of the results was taken. The maximum overhead was 13.58 μsec, which indicates that the effect on an application running in 10 ms or 100 ms is less than 1%. Therefore, the defect rate is tolerable.

The overhead of *ChainTask* is due to act_tsk and ext_tsk being executed for the system call to provide the corresponding function. ITRON system calls to get the information refer more information than the OSEK system calls. Hence, the OSEK system calls that refer to the information have more overhead such as *GetTaskState* and *GetEvent*. The overhead of *GetResource* is caused by saving the current priority before the priority has changed to the *ceiling* priority. The system calls of the alarm have more overhead because the alarm functionalities are implemented by using several ITRON functions. In addition, about 5 μsec more overhead is caused, if the task is dispatched to the non-preemptive task.

9.3 Deadline Miss Ratios of Non-preemptive Scheduling

The performance on the ITRON OS is evaluated using non-

Table 4 Performance of MINDSTORMS NXT.

Hardware	Performance
Microprocessor	32-bit ARM7
Flash memory	256 Kbyte
RAM	64 Kbyte
Frequency	48 MHz

© 2014 Information Processing Society of Japan

Table 5 Overheads of applicable functionalities.

System call	ASP (μsec)	ATK1 (μsec)	Overhead (μsec)
ActivateTask	5.48	7.05	1.57
TerminateTask	7.74	6.82	0.92
ChainTask	14.75	9.54	5.21
Schedule	8.73	8.60	0.13
GetTaskID	2.81	2.29	0.52
GetTaskState	5.47	2.29	3.18
EnableAllInterrupts	1.12	0.55	0.57
DisableAllInterrupts	1.45	0.53	0.92
ResumeAllInterrupts	1.34	1.21	0.13
SuspendAllInterrupts	1.68	1.58	0.10
ResumeOSInterrupts	2.02	1.93	0.09
SuspendOSInterrupts	2.51	2.35	0.16
SignalCounter	2.85	4.59	−1.74
GetAlarmBase	7.05	2.47	4.58
GetAlarm	6.72	3.64	3.08
SetRelAlarm	6.91	5.25	1.66
SetAbsAlarm	7.20	3.72	3.48
CancelAlarm	1.22	3.37	−2.15
GetActiveApplicationMode	0.66	0.61	0.05
GetResource	16.96	3.38	13.58
ReleaseResource	8.24	7.43	0.81
SetEvent	10.39	7.09	3.30
ClearEvent	3.76	2.71	1.05
GetEvent	10.12	3.10	7.02
WaitEvent	11.83	7.13	4.70

preemptive task components. In this case, the proposed framework is used for the non-preemptive task in Section 8.1.1. The deadline miss ratio in a task set, which is defined as M_{rate} in Eq. (1), on the ITRON OS is measured as:

$$M_{rate} = \frac{1}{n}\sum_{i=1}^{n} miss(f_i). \quad (1)$$

The number of tasks in a task set is n, and $miss(f_i)$ is defined as in Eq. (2):

$$miss(f_i) = \begin{cases} 0 & (\textit{The task executed in time.}) \\ 1 & (\textit{The task missed its deadline.}) \end{cases} \quad (2)$$

In the evaluation, RMS [*15] is used for each scheduling policy. The CPU utilization of the task set is sampled every 5% between 5% and 100%. The period is each 100 ms, from 100 ms to 1,000 ms. Each sampling point runs three task sets, each of which consists of 20 tasks, and the averages of the deadline miss ratios are shown in **Fig. 7**.

As a result, the measured deadline miss ratio using the ITRON OS is identical to the measured deadline miss ratio using the OSEK OS between 0% to 85%. Tasks are also started to miss their deadline if the CPU utilization exceeds 85%. Therefore, this proposed framework using the ITRON OS could be used similar to the system using the OSEK OS in practical uses while the CPU utilization is less than 85%.

9.4 Comparison of Amount of Code

As a use case, porting an OSEK application to the ITRON OS is compared. For instance, 226 system calls are needed to be ported in the small samples [*16]. However, OSEK system calls, which do not have corresponding system calls in ITRON OS, are needed to be implemented by using ITRON functionalities. The

*15 Rate-Monotonic Scheduling

*16 The samples are downloaded from https://www.nces.is.nagoya-u.ac.jp/NEP/materials/.

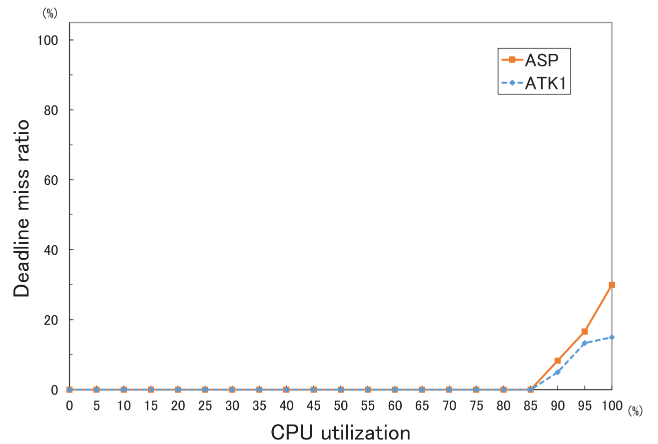

Fig. 7 Deadline miss ratio of non-preemptive scheduling.

176 lines of the C code and 289 + 8α [*17] lines of the CDL description are needed to use OSEK functionalities on the ITRON OS with TECS. As an example of the effectiveness of TECS, the developer does not need to rewrite kernel configuration files, such as static APIs for defining the kernel configuration and initial states of tasks and semaphores. For example, the OSEK kernel configuration (OIL description) for the inverted pendulum [*18] in the OSEK OS needs 75 lines of code. The OSEK kernel configuration is needed to be ported to the ITRON kernel configuration (static API) on the ITRON OS. The developer needs to rewrite 28 lines of the ITRON kernel configuration to use an OSEK application on the ITRON OS. Therefore, the amount of work for the developer to port an OSEK application to the ITRON OS with TECS is reduced by using the proposed framework.

9.5 User Restrictions

At this stage, the components still have user restrictions. The proposed OSEK system calls on the ITRON OS are shown in **Table 6**. The colored symbols indicate system calls are achieved. The circle symbol indicates that the system call correspondences to the system call. The triangle symbol indicates that the corresponding system call has limits. The cross symbol indicates that the system call has restrictions. The limits and restrictions are stated below.

9.5.1 System Calls while All Interrupts are Disabled

In OSEK specifications, *ResumeAllInterrupts* and *SuspendAllInterrupts* save and restore the recognition status of all interrupts that the hardware supports. These functionalities are implemented with *SIL_LOC_INT* and *SIL_UNL_INT*, although *SIL_LOC_INT* and *SIL_UNL_INT* restrict the number of system calls that can be used while all interrupts are disabled. APIs on the system interface layer, a system call to check the non-operating state of the kernel, and a system call to terminate the kernel can be used. Therefore, there are limitations on the system call that can be used while all interrupts are disabled.

9.5.2 System Calls from ISR1

The OSEK ISR1 does not use operating system service calls, except for system services to enable and disable interrupts. Non-kernel interrupts correspond to the OSEK ISR1 on the ITRON

*17 α is the number of supported application mode.

*18 http://www.hokutodenshi.co.jp/7/PUPPY.htm

© 2014 Information Processing Society of Japan

Table 6 Corresponding OSEK system calls on the ITRON OS.

Service	Task	ISR category 1	ISR category 2	ErrorHook	PreTaskHook	PostTaskHook	StartupHook	ShutdownHook	alarmcallback
ActivateTask	○		○						
TerminateTask	○								
ChainTask	●								
Schedule	●								
GetTaskID	○		○	●	●	●			
GetTaskState	○		●	●	●	●			
DisableAllInterrupts	●	×	●						
EnableAllInterrupts	●	×	●						
SuspendAllInterrupts	▲	×	▲	▲	▲	▲			▲
ResumeAllInterrupts	▲	×	▲	▲	▲	▲			▲
SuspendOSInterrupts	●	×	●						
ResumeOSInterrupts	●	×	●						
GetResource	●		●						
ReleaseResource	●		●						
SetEvent	●		●						
ClearEvent	○								
GetEvent	●		●	●	●	●			
WaitEvent	○								
GetAlarmBase	●		●	●	●	●			
GetAlarm	●		●	●	●	●			
SetRelAlarm	●		●						
SetAbsAlarm	●		●						
CancelAlarm	●		●						
GetActiveApplicationMode	●		●	●	●	●	●	●	

○ indicates that the system call has one-to-one correspondence to the system call.
● indicates that the system call is achieved.
▲ indicates that the achieved corresponding system call has limits.
× indicates that the achieved system call has restrictions.

OS. Although no system calls can be invoked from interrupt handlers started by non-kernel interrupts. Therefore, system calls from ISR1 cannot be used.

10. Conclusion

Several operating system specifications for automobiles such as ITRON and OSEK specifications are used in a number of ECUs. Developers need to implement applications following these specifications. For instance, it is hard to port legacy OSEK applications on the ITRON OS because of unsupported OSEK functionalities such as scheduling policy, counter, alarm, resources, events, hook routines, and system calls. As a result, legacy OSEK applications need substantial changes and developers to redesign the applications. As a consequence, the amount of work increases.

This paper proposed the framework to apply the unsupported OSEK functionalities on the ITRON OS with TECS components. The components help developers to port legacy OSEK applications on the ITRON OS. In addition, OSEK functionalities are provided as a component without changes to the ITRON OS. As a result, the work of the developers are reduced by using the proposed framework. As experimental results, the overhead of the corresponding system calls are compared to the original OSEK system calls and suppressed within 13.58 μsec. Applying the proposed framework to AUTOSAR OS is future work because AUTOSAR OS extends OSEK specifications.

Acknowledgments This research was partially supported by A-STEP (Adaptable and Seamless Technology Transfer Program through Target-driven R&D) 2011–2012 (AS231Z02289A) and JSPS KAKENHI Grant Number 40582036. Takayuki Ukai and Takuya Ishikawa's supporting comments were invaluable.

References

[1] Merritt, R.: IBM tells story behind Chevy Volt design, UBM Tech (online), available from ⟨http://eetimes.com/electronics-news/4215735/IBM-tells-story-behind-Chevy-Volt-design⟩ (accessed 2013-11-13).

[2] Aoyama, M.: Computing for the next-generation automobile, *Computer*, Vol.45, No.6, pp.32–37, IEEE (2012).

[3] Hellestrand, G.: The engineering of supersystems, *Computer*, Vol.38, No.1, pp.103–105, IEEE (2005).

[4] Takada, H. and Sakamura, K.: μITRON for small-scale embedded systems, *Micro*, Vol.15, No.6, pp.46–54, IEEE (1995).

[5] TRON Association: ITRON Project Archive, ITRON Project (online), available from ⟨http://www.ertl.jp/ITRON/home-e.html⟩ (accessed 2013-11-13).

[6] OSEK/VDX: OSEK/VDX Operating System Specification Version 2.2.3, OSEK/VDX (online), available from ⟨http://portal.osek-vdx.org/files/pdf/specs/os223.pdf⟩ (accessed 2013-11-13).

[7] AUTOSAR development partnership: AUTOSAR, AUTOSAR development partnership (online), available from ⟨http://www.autosar.org/index.php⟩ (accessed 2013-11-13).

[8] T-Engine Forum: The Past, Present, and Future of Smart "Kaden", T-Engine Forum (online), available from ⟨http://www.t-engine.org/wp-content/themes/wp.vicuna/pdf/d70924f49201d66e0b9508be0ab22fad1.pdf⟩ (accessed 2013-11-13).

[9] Ishitani, T., Yamazaki, F., Nagao, T., Yamada, M., Matsubara, Y., Honda, S. and Takada, H.: Communication middleware between different kind OSes for in-vehicle ECU integration, *IPSJ SIG Technical Reports, EMB, Embedded Systems*, Vol.2010, No.3, pp.1–11 (online), available from ⟨http://ci.nii.ac.jp/naid/110007993213/⟩ (2010).

[10] Van Ommering, R., Van Der Linden, F., Kramer, J. and Magee, J.: The Koala component model for consumer electronics software, *Computer*, Vol.33, No.3, pp.78–85, IEEE (2000).

[11] Fassino, J., Stefani, J., Lawall, J. and Muller, G.: Think: A software framework for component-based operating system kernels, *2002*

© 2014 Information Processing Society of Japan

USENIX Annual Technical Conference, pp.73–86 (2002).
[12] Hansson, H., Akerholm, M., Crnkovic, I. and Torngren, M.: SaveCCM – A component model for safety-critical real-time systems, *Proc. 30th Euromicro Conference, 2004*, pp.627–635, IEEE (2004).
[13] Azumi, T., Ukai, T., Oyama, H. and Takada, H.: Wheeled Inverted Pendulum with Embedded Component System: A Case Study, *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing* (*ISORC*), pp.151–155, IEEE (2010).
[14] Azumi, T., Yamamoto, M., Kominami, Y., Takagi, N., Oyama, H. and Takada, H.: A new specification of software components for embedded systems, *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, (*ISORC '07*), pp.46–50, IEEE (2007).
[15] Carlson, J., Hakansson, J. and Pettersson, P.: SaveCCMX: An analysable component model for real-time systems, *Electronic Notes in Theoretical Computer Science*, Vol.160, pp.127–140 (2006).
[16] Yang, G., Li, H. and Wu, Z.: SmartC: A Component-Based Hierarchical Modeling Language for Automotive Electronics, *2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pp.203–210, IEEE (2006).
[17] Zhao, M., Wu, Z., Yang, G., Wang, L. and Chen, W.: SmartOSEK: A real-time operating system for automotive electronics, *Embedded Software and Systems*, pp.437–442 (2005).
[18] Nagao, T., Yamada, M., Ishitani, T., Matsubara, Y., Yamazaki, Y., Honda, S. and Takada, H.: DUOS: A Real-Time OS Framework for Integrating Electronic Control Units in Automotive Control Systems, *IPSJ SIG Technical Reports, EMB, Embedded Systems*, Vol.2010, No.2, pp.1–9 (online), available from ⟨http://ci.nii.ac.jp/naid/110007993212/⟩ (accessed 2010-05-25).
[19] eCosCentric: eCos, eCosCentric (online), available from ⟨http://ecos.sourceware.org/⟩ (accessed 2014-02-24).
[20] Xenomai project: Xenomai, Xenomai project (online), available from ⟨http://www.xenomai.org/⟩ (accessed 2014-02-24).
[21] Azumi, T., Oyama, H. and Takada, H.: Optimization of component connections for an embedded component system, *International Conference on Computational Science and Engineering*, (*CSE '09*), Vol.2, pp.182–188, IEEE (2009).
[22] TOPPERS Project: TOPPERS/ATK1 (in Japanese), TOPPERS Project (online), available from ⟨http://www.toppers.jp/atk1.html⟩ (accessed 2013-11-13).
[23] TOPPERS Project: TOPPERS/ASP Kernel, TOPPERS Project (online), available from ⟨http://www.toppers.jp/tecs.html⟩ (accessed 2013-11-13).
[24] TOPPERS Project: TECS (in Japanese), TOPPERS Project (online), available from ⟨http://www.toppers.jp/tecs.html⟩ (accessed 2013-11-13).
[25] Azumi, T., Yamada, S., Oyama, H., Nakamoto, Y. and Takada, H.: A new security framework for embedded component systems, *Proc. 11th IASTED International Conference on Software Engineering and Applications*, pp.584–589, ACTA Press (2007).

Atsushi Ohno was born in 1989. He received his M.S. degree from the Graduate School of Information Science and Engineering, Ritsumeikan University in 2014. His research interest is real-time embedded systems and Model-Based Development.

Takuya Azumi is an Assistant Professor at the Graduate School of Engineering Science, Osaka University. He received his Ph.D. degree form the Graduate School of Information Science, Nagoya University. From 2008 to 2010, he was under the research fellowship for young scientists for Japan Society for the Promotion of Science. From 2010 to 2014, he was an Assistant Professor at the College of Information Science and Engineering, Ritsumeikan University. His research interests include real-time operating systems and component based development. He is a member of IEEE, IEICE, and JSSST.

Nobuhiko Nishio was born in 1962. Graduated Mathematical Engineering and Information Physics, School of Engineering, The University of Tokyo in 1986. M.S. from Graduate School of The University of Tokyo in 1988. After quitting Ph.D. course, worked at Keio University Shonan Fujisawa Campus from 1993 to 2003. Ph.D. in Media and Governance in 2000. Associate professor at College of Science and Engineering, Ritsumeikan University in 2003. Since 2005, Professor at College of Information Science and Engineering, Ritsumeikan University (Current profession). Visiting Scientist in Google Inc. from 2007 to 2008. Majored in autonomous, distributed and collaborative systems, ubiquitous computing and networking and sensing systems. Awarded as the Yamashita Memorial Research Prize from IPSJ. Regular member of IPSJ, ACM and IEEE Computer Society.

© 2014 Information Processing Society of Japan