2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing

A Distributed Real-Time Operating System with Distributed Shared Memory for Embedded Control Systems

Takahiro Chiba*, Myungryun Yoo and Takanori Yokoyama
Tokyo City University
1-28-1, Tamazutsumi, Setagaya-ku, Tokyo 158-8557 Japan
**Presently with Systems Engineering Consultants Co., LTD.*
Email: chiba@sec.co.jp, {yoo, yokoyama}@cs.tcu.ac.jp

***Abstract*—The paper presents a distributed real-time operating system (DRTOS) that provides a distributed shared memory (DSM) service for distributed control systems. Model-based design has become popular in embedded control software design and the source code of software modules can be generated from a controller model. The generated software modules exchange their input and output values through shared variables. We develop a DRTOS with a real-time DSM service to provide a location-transparent environment, in which distributed software modules can exchange input and output values through the DSM. The DRTOS is an extension to OSEK OS. We use a real-time network called FlexRay, which is based on a TDMA (Time Division Multiple Access) protocol. The consistency of the DSM is maintained according to the order of data transfer through FlexRay, not using inter-node synchronization. The worst case response time of the DSM is predictable if the FlexRay communication is well configured.**

***Keywords*-operating systems; real-time systems; embedded systems; distributed shared memory; distributed control systems;**

I. Introduction

An application program of an embedded control system is designed as a set of software modules. For example, an automotive engine control application program consists of a number of software modules for fuel injection, ignition, emission control and diagnosis. The software modules are executed by tasks on a real-time operating system (RTOS). For example, OSEK OS [1], a de facto standard operating system presented by OSEK/VDX, is widely used in automotive control systems.

Model-based design has become popular in embedded control software design, especially in the domain of automotive control design. In model-based design, a controller model is designed and verified using a CAD/CAE tool such as MATLAB/Simulink [2]. The source code of software modules can be generated from the controller model by a code generator such as Real-Time Workshop/Embedded Coder [2]. The generated software modules exchange their input and output values through global variables.

Distributed embedded control systems are used in the domains of automotive control, factory automation, building control, and so on. Time predictability is one of the most important issues for design of distributed automotive control systems [3]. Real-time and location-transparent distributed computing environments are required.

Message-based communication environments are used in distributed embedded control systems. For example, OSEK COM [4], a de facto standard communication environment presented by OSEK/VDX, is widely used in automotive control systems. Messages of OSEK COM are represented as message objects. An application program sends a message by calling *SendMessage()* and receives a message by calling *ReceiveMessage()*. If we build a distributed control system with software modules developed by model-based design on a message-based communication environment, we have to rewrite the generated source code to exchange input and output values by messages, not global variables.

Distributed shared memory (DSM) provides location-parent shared variables, so distributed software modules developed by model-based design can exchange their input and output values through shared variables on DSM. However, existing DSM systems are not suitable for embedded control systems. Most DSM systems are based on page-based DSM [5][6]. The response time of page-based DSM is difficult to predict in a distributed computing environment. It is also difficult to implement a page-based based DSM mechanism in a small RTOS with no virtual memory on a microcontroller without MMU (Memory Management Unit), which is widely used in embedded control systems.

The goal of the research is to develop a distributed real-time operating system (DRTOS) with DSM for embedded distributed control systems. To achieve the goal, we present a real-time DSM service suitable for distributed embedded control systems with software modules generated from Simulink models.

We have already developed a DRTOS with location-transparent system calls for task management and event control as an extension to OSEK OS [7]. An application task activates or synchronizes with remote tasks using the same APIs as local tasks. The DRTOS manages distributed tasks based on the global time, which is supported by the

978-1-4799-3381-5/13 $31.00 © 2013 IEEE
DOI 10.1109/DASC.2013.71

IEEE computer society

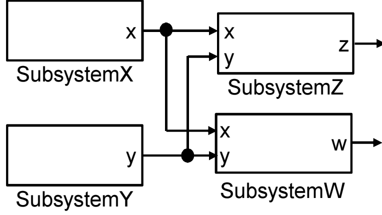

Figure 1. Example Structure Layer of Simulink Model

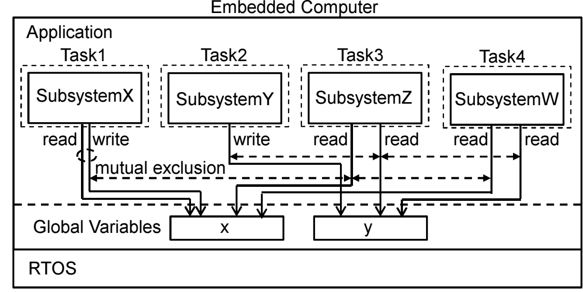

Figure 2. Example Control Software

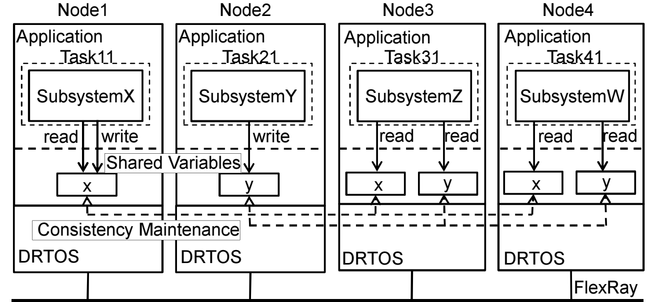

Figure 3. Example Distributed Control Software

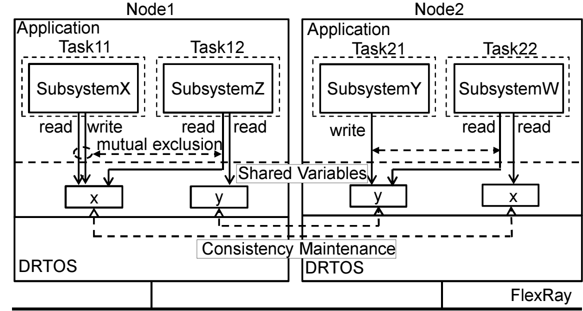

Figure 4. Another Example Distributed Control Software

clock synchronization of FlexRay [8]. FlexRay is a real-time network for automotive control systems based on a TDMA (Time Division Multiple Access) protocol. We add a real-time DSM mechanism to the DRTOS. The consistency of the DSM is maintained according to the order of data transfer through FlexRay. If the FlexRay communication is well configured, the worst case response time of the DSM is predictable.

The rest of the paper is organized as follows. Section II describes the DSM model for embedded control software. Section III describes the details of the DRTOS with DSM. Section IV describes implementation and experimental evaluation of the DSM. Section V concludes the paper.

II. Distributed Shared Memory for Embedded Control Systems

A. Distributed Control Software

A controller model built with MATLAB/Simulink is a layered model. According to the modeling guidelines presented by MAAB (Mathworks Automotive Advisory Board) [9], a controller model consists of the top layer, the trigger layer (optional), the structure layer and the dataflow layer. A structure layer model represents the structure of a controller model, which is a set of subsystem blocks. A data flow layer model represents detailed control logic (control algorithm). The source code a software module is generated from a subsystem block of a controller model.

Figure 1 illustrates a part of an example structure layer model, which consists of *SubsystemX*, *SubsystemY*, *SubsystemZ* and *SubsystemW*. The signal line from output *x* of *SubsystemX* is connected to input *x* of *SubsystemZ* and input *x* of *SubsistemW*. This means that *SubsystemX* outputs the value of *x* and *SubsustemZ* and *SubsystemW* input the value. The signal line from output *y* of *SubsystemY* is also similar. Data *x* and data *y* are represented as global variables in the source code generated from the model.

Figure 2 illustrates the structure of an example control software corresponding to the model shown by Figure 1. *SubsystemX*, *SubsytemY*, *SubsystemZ* and *SubsystemW* are respectively executed by *Task1*, *Task2*, *Task3* and *Task4*. *SubsystemX* reads and writes the value of global variable *x* and *SubsystemY* writes the value of global variable *y*. *SubsytemZ* and *SubsystemW* read the value of *x* and the value of *y*. Mutual exclusion is generally needed to access a global variable in a preemptive multi-task environment.

Software modules corresponding to subsystem blocks are distributed to a number of nodes in a distributed embedded control system. Figure 3 illustrates the structure of an example distributed control software with DSM. Software modules generated from the Simulink model shown by Figure 1 are distributed to four nodes. *SubsystemX*, *SubsytemY*, *SubsytemZ* and *SubsytemW* are respectively executed by *Task11* on *Node1*, *Task21* on *Node2*, *Task31* on *Node3* and *Task41* on *Node4*. The copies of shared variables *x* and *y* are located on the nodes. The consistency of the shared variables is maintained by the DSM service of the DRTOS.

Figure 4 illustrates the structure of another example distributed control software. The software modules are distributed to two nodes. *SubsystemX* and *SubsytemZ* are

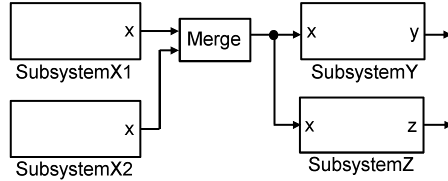

Figure 5. Example Structure Layer of Simulink Model with Merge Block

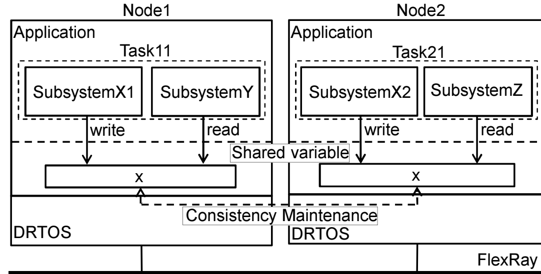

Figure 6. Example Distributed Control Software with Multiple Writers

respectively executed by *Task11* and *Task12* on *Node1*. *SubsytemY* and *SubsytemW* are respectively executed by *Task21* and *Task22* on *Node2*. Mutual exclusion is used between the tasks on the same node. The consistency of the shared variables between different nodes is maintained by the DSM service of the DRTOS.

We call a task that performs just read operations to a shared variable a reader task, call a task that performs just write operations to a shared variable a writer task, and call a task that performs both read and write operations to a shared variable a reader-writer task. For example, *Task11* is a reader-writer task of *x*, *Task21* is a writer task of *y*, and *Task31* and *Task41* are reader tasks of *x* and *y* in Figure 3.

There are three kinds of DSM models: single reader/single writer (SRSW) model, multiple reader/single writer (MRSW) model and multiple reader/multiple writer (MRMW) model [6]. A signal line of a Simulink model is connected to just one output of a subsystem block and connected to one or more inputs of other subsystem blocks. So the MRSW model usually fits the distributed software modules generated from Simulink models.

The MRMW model of DSM may be needed when a single shared variable is used for a special block that connects a number of signal lines. For example, Figure 5 illustrates a Simulink model with a *Merge* block, which merges the output signal lines of *SubsystemX1* and *SubsystemX2*. The merged signal line is connected to the inputs of *SubsystemY* and *SubsystemZ*. If *SubsystemX1* and *SubsystemX2* just perform write operations to *x*, *SubsystemX1*, *SubsystemX2*, *SubsystemY* and *SubsystemZ* can be connected with a shared variable corresponding to *x*.

Figure 6 illustrates the structure of an example distributed control software for the Simulink model shown by Figure 5. Shared variable *x* is used to store both the output value of *SubsystemX1* and the output value of *Subsystemx2*. The MRMW model of DSM is needed in this case because both *Task11* and *Task21* perform write operations to *x*. So we support not only the MRSW model but also the MRMW model. However, note that a single shared variable can not be used if either *SubsystemX1* or *SubsystemX2* performs both read and write operations to *x* because the calculation of *SubsystemX1* and the calculation of *SubsystemX2* must be executed independently.

B. Distributed Shared Memory Model

We present a DSM mechanism based on a shared-variable DSM architecture [5], not a page-based DSM architecture, because only certain variables are shared in distributed control software developed with MATLAB/Simulink.

The design policies of the DSM are shown below.

- MMU should not be used because most microcontrollers used in embedded control systems have no MMU.
- Inter-node synchronization should not be used because it may cause a performance problem (Intra-node synchronization (inter-task synchronization) is acceptable).
- No new API of DRTOS for DSM is required because new API may violate the compatibility (an extension of the semantics of an API is acceptable).
- Consistency sufficient for the control software generated from Simulink models should be provided.

The consistency of the DSM is maintained according to the order of data transfer through FlexRay. FlexRay communication is periodically performed with a communication cycle. The DRTOS receives the transferred data by cyclic polling, not by interrupt, for the predictability of the response time [7].

Some consistency models for DSM have been presented [5][6][10]. Sequential consistency [11] is the strongest consistency other than strict consistency. Sequential consistency is not needed for the control software generated from Simulink models, but the same sequential order of write operations to the same shared variable is required. We call this partially-sequential consistency.

To realize the partially-sequential consistency model not using inter-node synchronization, we present a method that the next access operation (write or read operation) after a write operation to the same shared variable is inhibited until the data transfer for the write operation is completed. If a write operation to a shared variable is performed by a task, the task cannot access the shared variable until its value is transferred to other nodes.

Figure 7 illustrates an example DSM access sequence in the case of Figure 3. The operations performed by a task on each node are shown horizontally, with time increasing to the right. Symbol *R(x)a* means that a task reads value

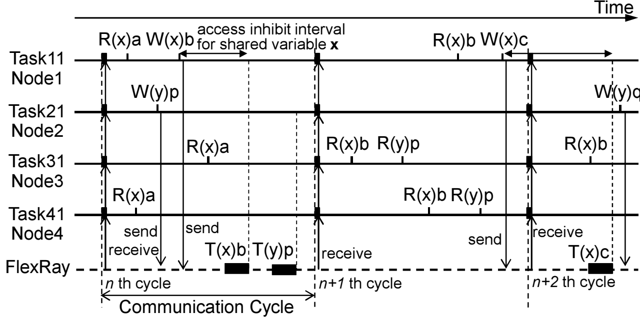

Figure 7. Example DSM Access Sequence

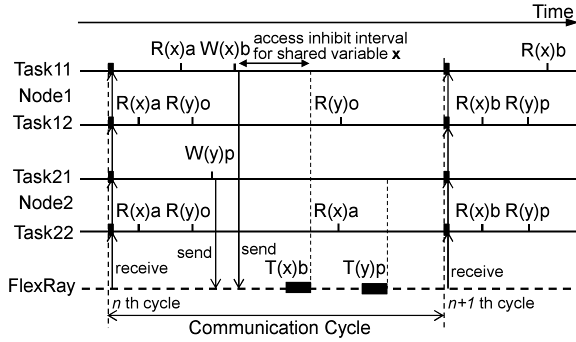

Figure 8. Another Example DSM Access Sequence

a from shared variable *x*. Symbol *W(x)b* means that a task writes value *a* into shared variable *x*.

The value of a shared variable written by a task is transferred to other nodes through FlexRay. Symbol *T(x)b* in Figure 7 means that value *b* of variable *x* is transferred to other nodes. The transferred value is received and written into the copy of the shared variable in each node at the beginning of the communication cycle. In Figure 7, value *b* of variable *x* sent by *Task11* on *Node1* and value *p* of variable *y* sent by *Task21* on *Node2* during the *nth cycle* are received by *Task31* on *Node2* and *Task41* on *Node3* at the beginning of the *n+1th cycle*.

Partially-sequential consistency is realized by the access inhibit interval. In the example of Figure 7, after *Task11* on *Node1* performs *R(x)a* and *W(x)b*, *Task11* cannot access *x* until *T(x)b* is completed. The access inhibit interval is needed just for reader-writer tasks, not for reader tasks or writer tasks.

Figure 8 illustrates an example DSM access sequence in the case of Figure 4. *W(x)b* by *Task11* on *Node1* and *W(y)p* by *Task21* on *Node2* are performed independently. *Task12* on *Node1* observes that *W(x)b* is performed before *W(y)p* and *Task22* on *Node2* observes that *W(y)p* is performed before *W(x)b*. Access operations to different shared variables may be observed in different order.

Figure 9 illustrates an example DSM access sequence in the case of Figure 6. *Task11* on *Node1* and *Task21* on *Node2*

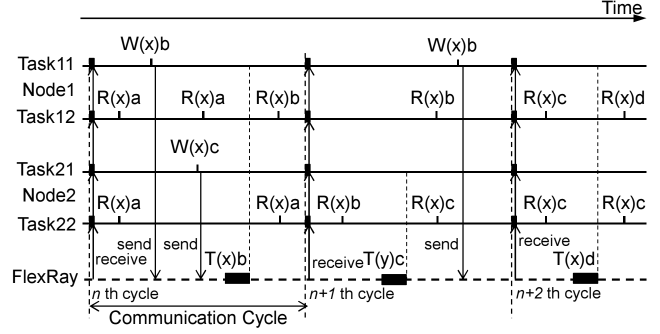

Figure 9. Example MRMW DSM Access Sequence

are writer tasks, not reader-writer tasks, so no access inhibit interval is needed. *Task12* on *Node1* and *Task22* on *Node2* observe that the write operations to *x* by *Task11* on *Node1* and by *Task21* on *Node2* are performed in the same order, for example, *W(x)b* is observed before *W(x)c*. The same sequential order of write operations is observed by *Task12* and *Task22*.

We assume the FlexRay communication cycle period is sufficiently shorter than the periods of periodic application tasks. The assumption is proper because the typical FlexRay communication cycle is 1msec and the typical period of automotive application tasks is 10msec or longer. So the access inhibit intervals are also sufficiently shorter than the interval time between write operations performed by periodic application tasks. If multiple data transfers for the same shared variable are performed during one communication cycle (this rarely occurs because of the above assumption), just the value of the last transfer is received. This is not a problem because it looks like the write operations are performed consecutively.

C. API of Distributed Shared Memory

OSEK OS provides resource access system calls for mutual exclusion: *GetResource()* and *ReleaseResource()*. When a task accesses a global variable that are shared with another task, the task calls *GetResource()* before the access and calls *ReleaseResource()* after the access.

We extend the semantics of the resource management system calls and use them to access shared variables on the DSM. A set of distributed shared variables is dealt with as a distributed shared resource. When a task accesses a distributed shared variable, the task calls *GetResource()* before the access and calls *ReleaseResource()* after the access. However, inter-node mutual exclusion is not supported as described in Section II-B.

Figure 10 shows a fragment of example source code of application program. The name of the shared variable is *shared_x* and the identifier of the resource for *shared_x* is *Res_shared_x*. We have to insert system calls *GetResource()* and *ReleaseResource()* into the source code generated from

```
. . . . . . . .
/* get the resource for the shared variable */
GetResource(Res_shared_x);
/* update the shared variable */
shared_x = a * shared_x + b;
/* release the resource for the shared variable */
ReleaseResource(Res_shared_x);
. . . . . . . .
```

Figure 10. An Example Source Code

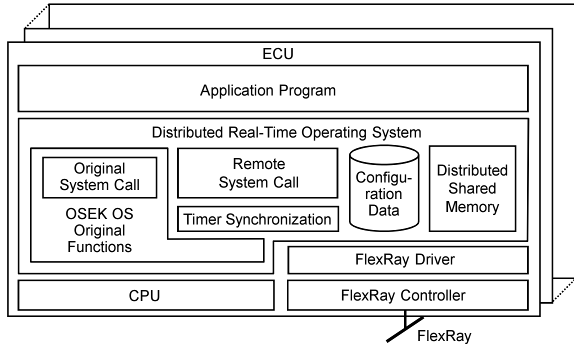

Figure 11. Structure of Distributed Real-Time Operating System

Table I
OSEK OS SYSTEM CALLS FOR TASK MANAGEMENT AND EVENT CONTROL

Category	API	Remote Call
Task Management	ActivateTask(*Task*)	Yes
	TerminateTask()	No
	ChainTask(*Task*)	Yes
	Schedule()	No
	GetTaskID(*TaskRef*)	No
	GetTaskStatus(*Task, StateRef*)	Yes
Event Control	SetEvent(*Task, Event*)	Yes
	ClearEvent(*Event*)	No
	GetEvent(*Task, EventRef*)	Yes
	WaitEvent(*Event*)	No

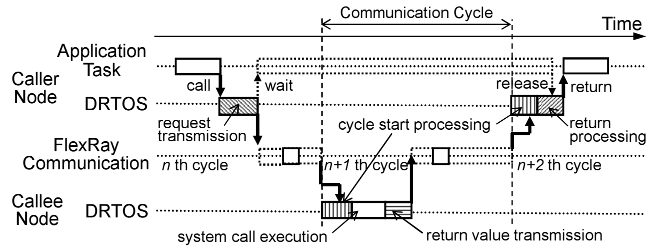

Figure 12. Time Chart of Remote System Call

Simulink models to utilize DSM. However, mutual exclusion must be also considered even when the software modules are executed in a preemptive multi-task environment on a single processor system.

A set of shared variables, not just a shared variable, can be handled as a distributed shared resource. For example, a set of shared variables x and y in Figure 3 and Figure 4 can be handled as the same resource.

The configuration of an application on OSEK OS is described in OIL (OSEK Implementation Language) [12]. For example, tasks and events are statically declared in an OIL file. The configuration data of OSEK OS are generated by the system generator (SG) referring to the OIL file. We extend OIL to declare distributed shared variables and distributed resources.

III. DISTRIBUTED REAL-TIME OPERATING SYSTEM WITH DISTRIBUTED SHARED MEMORY

A. Overview

We have already developed a DRTOS with location-transparent system calls [7] as an extension to TOPPERS/OSEK kernel, an OSEK-compliant operating system developed by TOPPERS project [13]. We extend the DRTOS to support DSM based on the model presented in Section II-B.

Figure 11 illustrates the structure of the DRTOS, which consists of the OSEK OS original functions, a timer synchronization module, a remote system call module, a distributed shared memory module and configuration data.

The timer synchronization module manages the global time. FlexRay provides the network time that is synchronized between the FlexRay controllers. The timer of the DRTOS on each node is periodically synchronized with the clock of the network time. The value of the timer of the DRTOS is used as the global time.

Table I shows the OSEK OS system calls for task management and event control. The column *Remote Call* of Table I shows whether the system call is extended to be a remote system call or not, i.e., *ActivateTask()*, *ChainTask()*, *GetTaskStatus()*, *SetEvent()* and *GetEvent()* are extended. Parameter *Task* means the task ID. In the extended system calls, a task ID is a unique ID in a distributed system, not only unique in a node. If one of these system calls is issued specifying a remote task, the remote system call module executes the processing for the remote system call.

Figure 12 shows a time chart of a remote system call. When an application task issues a remote system call, the remote system call module determines the node on which the target task resides, generates a request message, calls the FlexRay driver to write the request message in the message RAM of the FlexRay controller, and shifts the caller task to the *waiting* state. The communication of the request message is executed by FlexRay controllers.

Received request messages are stored in the message RAM of the FlexRay controller of the callee node. The cycle start processing is executed by an ISR (Interrupt Service Routine), which is activated at the beginning of each FlexRay communication cycle. The cycle start processing executes global time maintenance, calls the FlexRay driver

to read messages received in the previous communication cycle, interprets the request message, and calls the original system calls of OSEK OS. Then the ISR generates a return message and calls the FlexRay driver to write the return message in the message RAM.

The ISR of the caller node executes the cycle start processing, stores the return value and output parameters in a buffer, and releases the caller task from the *waiting* state. The caller task reads the return value and output parameters from the buffer and resumes executing the application program.

The distributed shared memory module manages the copies of shared variables and maintains the consistency. The details of the DSM mechanism are described in Section III-B.

The communication cycle of FlexRay is divided into the static segment for periodic messages and the dynamic segment for event-triggered messages. The messages of remote system calls and DSM are transmitted in the dynamic segment because system calls and DSM accesses are eventually performed.

The configuration data consists of the original OSEK configuration data, task location data for remote system calls, and DSM configuration data.

B. Distributed Shared Memory Mechanism

This section describes the details of the DSM mechanism of the DRTOS. The copies of shared variables are allocated in the data section of application program on each node. The DRTOS on each node has shared data buffers and received data buffers.

We add DSM functionalities to *GetResource()* and *ReleaseResource()* as described in Section II-C. We call the former *DSM access preprocessing* and the latter *DSM access postprocessing*.

Figure 13 shows a time chart of the DSM processing on the writer node. When the application task calls *GetResource()* before accessing to a shared variable, the DRTOS executes the processing of the original *GetResource()* of OSEK. Then, the DRTOS determines if the resource is a DSM resource or not. If the resource is a DSM resource, the DRTOS executes the DSM access preprocessing, which copies the value of the shared data buffer to the shared variable.

When the application task calls *ReleaseResource()* after accessing to the shared variable, the DRTOS determines if the resource is a DSM resource or not. If the resource is a DSM resource, the DRTOS executes the DSM access postprocessing. The DSM access postprocessing compares the value of the shared variable and the value of the shared data buffer, and calls the FlexRay driver to send the former value if the values are different. Then the DRTOS executes the processing of the original *ReleaseResource()* of OSEK OS.

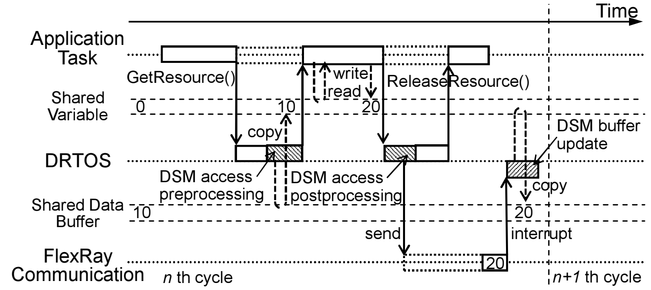

Figure 13. Time Chart of Writer Node Processing

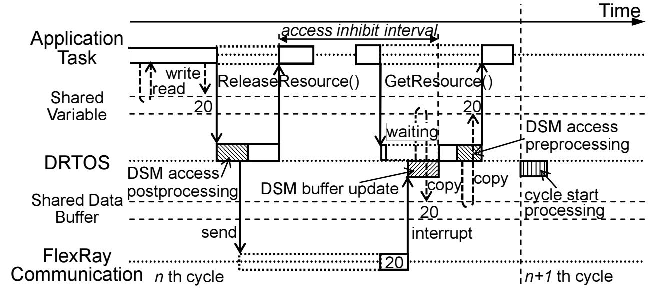

Figure 14. Access Inhibit for Partially-Sequential Consistency

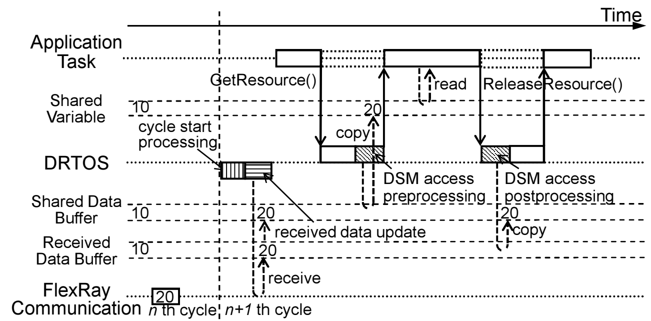

Figure 15. Time Chart of Reader Node Processing in Case 1

When the FlexRay controller completes the data transfer, an interrupt occurs. The interrupt executes DSM buffer update, which copies the value of the shared variable to the shared data buffer.

Figure 14 shows a time chart with an access inhibit interval. When the application task (reader-writer task) calls *GetResource()* until the FlexRay controller completes the data transfer, the DRTOS shifts the state of the task *waiting*. When the FlexRay controller completes the data transfer, the interrupt checks if there is a task with state *waiting* or not. If such a task exists, the interrupt shifts the state of the task *ready*. This is implemented using the event mechanism of OSEK OS.

Figure 15 shows a time chart of the DSM processing on the reader node. The cycle start processing of the DRTOS checks the data received during the previous communication cycle. If DSM data have been received, the DRTOS executes

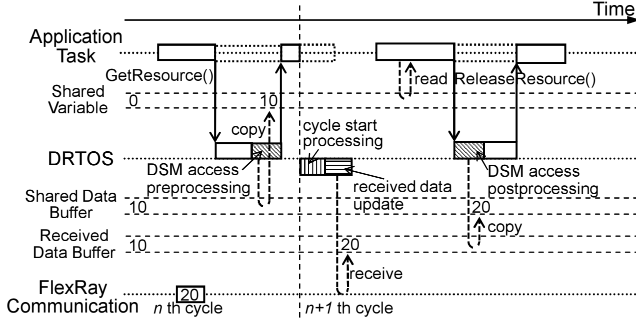

Figure 16. Time Chart of Reader Node Processing in Case 2

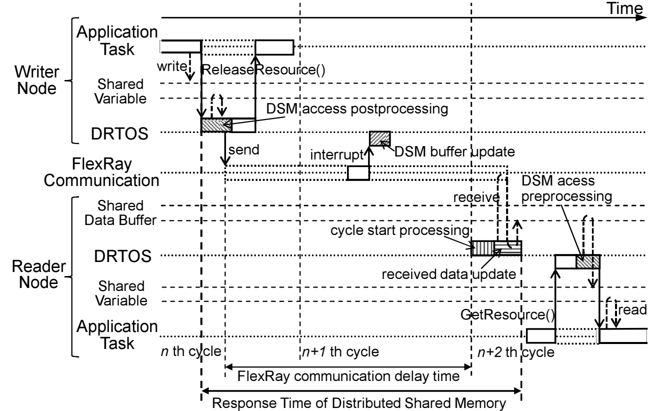

Figure 17. Response Time of Distributed Share Memory

received data update, which interprets the received data and checks if there is a task that holds the DSM resource.

Figure 15 shows the case that no task holds the DSM resource. In this case, the received data update processing writes the received value into both the received data buffer and the shared data buffer. When the application task calls *GetResource()* before accessing to a shared variable, the DRTOS determines if the resource is a DSM resource or not. If the resource is a DSM resource, the DRTOS executes DSM access preprocessing, which copies the value of the shared data buffer to the shared variable as described before. Then the application task reads the value from the shared variable.

Figure 16 shows a time chart in the case that there is a task that holds the DSM resource. In this case, the received data update processing writes the received value into just the received data buffer. When the application task calls *ReleaseResource()*, the DRTOS executes the DSM access postprocessing, which copies the value of the received data buffer to the shared data buffer.

C. Response Time of Distributed Shared Memory

FlexRay communication parameters are statically designed. Design and analysis methods for FlexRay communication have been presented [14][15]. FlexRay communication parameters must be designed considering frames for both the DRTOS and the application program. The frames for the DRTOS is determined by considering the maximum rate of DSM write operations and remote system calls performed in a communication cycle period. The maximum communication delay time is predictable if FlexRay communication is well configured, and so the response time of a DSM service is predictable.

In the example time chart shown by Figure 13, the data transfer after the write operation is performed in the same communication cycle as the DSM access postprocessing is executed. However, the data transfer may be postponed to the next communication cycle. Figure 17 shows a time chart of the case that the data transfer is postponed. This is the worst case because the data transfer is not postponed to the next to the next communication cycle if the FlexRay communication is well configured.

The response time of the DSM service consists of the DSM access postprocessing execution time, the FlexRay communication delay time, the cycle start processing execution time and the received data update execution time. In the worst case, the maximum total time of the DSM access postprocessing execution time and the FlexRay communication delay time is twice the communication cycle period. So the worst case response time is the total of twice the communication cycle period, the cycle start processing execution time and the received data update execution time. It is about twice the communication cycle period because the cycle start processing execution time and the received data update execution time are sufficiently less than the communication cycle period.

IV. Implementation and Experimental Evaluation

We have developed a prototype of the DRTOS with the DSM mechanism described in Section III-B. The prototype supports just the 32-bit data type for DSM. We manually define the DSM configuration data because the current version of SG does not support DSM. We are now extending OIL and SG to automatically generate the DSM configuration data.

We have done experiments to evaluate the performance of the prototype DRTOS. We use the evaluation boards called GT200N10, the CPU of which is V850E/PH03 with an on-chip E-Ray FlexRay controller. The clock rate of the CPU is 128MHz. The data transfer rate of FlexRay is 10MHz and the communication cycle period is 1msec.

We have run an evaluation program and measured the CPU execution times of DSM functions: the DSM access preprocessing execution time, the DSM access postprocessing execution time, the cycle start processing execution time, the received data update execution time, and the DSM buffer

Table II
EXECUTION TIME OF DSM MECHANISM

Processing		Execution Time [μsec]	
		Average	Worst
DSM Access Preprocessing		0.79	0.81
DSM Access Postprocessing	with Data Transfer	7.73	7.75
	without Data Transfer	0.90	0.90
Cycle Start Processing		32.07	32.09
Received Data Update		0.84	0.84
DSM Buffer Update		1.13	1.15

Table III
EXECUTION TIME OF RESOURCE ACCESS SYSTEM CALLS

System Call	Execution Time [μsec]			
	DRTOS		TOPPERS/OSEK Kernel	
	Average	Worst	Average	Worst
GetResource()	2.51	2.53	2.38	2.40
ReleaseResource()	2.79	2.81	2.66	2.68

update execution time. We have measured each execution time fifty times using a hardware counter, the clock rate of which is 32MHz. Table II shows their average values and the worst values. The values are in the case that the shared variable is a single 32-bit integer data.

We think each execution time is practically small for automotive control systems. The typical period of the automotive control application periodic tasks is 10msec or more. The typical event-triggered task of the automotive powertrain control system is a task synchronized with the crankshaft rotation, the period of which is 10msec when the speed of the crankshaft rotation is 6000rpm. The typical FlexRay communication cycle period is 1 msec. The dominant factor of the response time of the DSM is the FlexRay communication delay time and the worst case response time is about twice the communication cycle. The worst case response time is about one fifth of the application task period in the typical case, so we think the response time is practically small.

We have also measured the execution time of *GetResource()* and *ReleaseResource()* for a resource supported by original OSEK OS, not a distributed shared resource for DSM, to evaluate the overhead of the system calls. Table III shows their execution time in the case of the DRTOS and in the case of original TOPPERS/OSEK Kernel. The difference between them means the overhead caused by the DSM mechanism. We think the overheads are practically small because their values are less than 10% of the execution times of the system calls.

V. CONCLUSIONS

We have presented a DRTOS that provides a DSM service for distributed embedded control systems. The consistency of the DSM is maintained according to the order of data transfer through FlexRay, not using inter-node synchronization. The worst case response time of the DSM is predictable if the FlexRay communication is well configured. We have developed a prototype of the DRTOS and evaluated the performance of the DSM. According to the evaluation results, we think the performance is practically small for automotive control applications.

We are now developing the next version of the DRTOS with DSM that supports various data types of shared variables. We are also extending OIL and SG to automatically generate the DSM configuration data.

ACKNOWLEDGMENT

We would like to thank the developers of TOPPERS/OSEK Kernel. This work was supported in part by JSPS KAKENHI Grant Number 24500046.

REFERENCES

[1] OSEK/VDX, Operating System, Version 2.2.3, 2005.
[2] The MathWorks Inc., http://www.mathworks.com/.
[3] A. Sangiovanni-Vincentelli and M. Di Natale, “Embedded system design for automotive applications,” IEEE Computer, Vol.40, No.10, pp.42–51, 2007, doi:10.1109/MC.2007.344.
[4] OSEK/VDX, Communication, Version 3.0.3, 2004.
[5] A. S. Tanenbaum, Dstributed Operating Systems, Prentice Hall, New Jersey, 1995.
[6] J. Protic, M. Tomasevic and V. Milutinovic, “Distributed shared memory: concepts and systems,” IEEE Parallel & Distributed Technology: Systems & Applications, Vol.4, No.4, 1996, pp.63–71, doi:10.1109/88.494605.
[7] T. Chiba, Y. Itami, M. Yoo and T. Yokoyama, “A Distributed Real-Time Operating System with Location-Transparent System Calls for Task Management and Inter-task Synchronization,” Proc. IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2011, pp.1133–1138, doi:10.1109/TrustCom.2011.154.
[8] R. Makowitz and C. Temple, “FlexRay - a communication network for automotive control systems,” Proc. 2006 IEEE International Workshop on Factory Communication Systems, pp.207–212, 2006, doi:10.1109/WFCS.2006.1704153.
[9] MathWorks Automotive Advisory Board (MAAB), Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow, Version 3.0, 2012.
[10] S. V. Adve and K. Gharachorloo, “Shared memory consistency models: a tutorial,” IEEE Computer, Vol.29, No.12, 1996, pp.66–76, doi:10.1109/2.546611.
[11] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” IEEE Transactions on Computers, Vol.C-28, No.9, 1979, pp.690–691, doi:10.1109/TC.1979.1675439.
[12] OSEK VDX, OSEK/VDX System Generation OIL: OSEK Implementation Language Version 2.5, 2004.
[13] TOPPERS Project, http://www.toppers.jp/en/
[14] T. Pop, P. Pop, P. Eles, Z. Peng and A. Andrei, “Timing analysis of the FlexRay communication protocol,” Proc. 18th Euromicro Conference on Real-Time Systems, pp.203–216, 2006, doi:10.1109/ECRTS.2006.31.
[15] J. Ben, B. Yongming and L. Anhu, “A method for response time computation in FlexRay communication system,” Proc. IEEE International Conference on Intelligent Computing and Intelligent Systems, Vol.3, pp.47–51, 2009, doi:10.1109/ICICISYS.2009.5358231.