A Distributed Computing Environment for Embedded Control Systems with Time-Triggered and Event-Triggered Processing

Yuichi Itami, Tasuku Ishigooka* and Takanori Yokoyama
Musashi Institute of Technology
1-28-1, Tamazutsumi, Setagaya-ku, Tokyo 158-8557 Japan
g0881306@sc.musashi-tech.ac.jp, tasuku.ishigoka.kc@hitachi.com, yokoyama@cs.musashi-tech.ac.jp

Abstract

The paper presents a distributed computing environment for embedded control systems with time-triggered and event-triggered distributed processing. We have already presented a time-triggered distributed object model and a time-triggered distributed computing environment for embedded control systems. However, there are many embedded control systems with time-triggered and event-triggered processing. In this paper, we present two kinds of event-triggered distributed object models, a pure event-triggered distributed object model and a data-triggered distributed object model, in addition to the time-triggered distributed object model. We also present a distributed object computing environment based on a time-division scheduling for the mixed architecture with time-triggered and event-triggered distributed processing. The time division scheduling divides an execution cycle into a time-triggered processing segment and a non-time-triggered processing segment. The time-triggered distributed processing is executed in the former segment, and the event-triggered distributed processing is executed in the latter segment. The distributed object computing environment consists of a real-time operating system with the time division scheduling and distributed computing middleware to support the three kinds of distributed object models. We provide a development environment that generates stubs and configuration data to build distributed control systems.

1. Introduction

Distributed embedded control systems are widely used in the domains of automotive control, factory automation, building control, and so on. Most embedded control systems are hard real-time systems. There are two approaches to the design of real-time systems, an event-triggered architecture and a time-triggered architecture[7]. The event-triggered architecture consists of program modules that respond external inputs or communication messages immediately. The time-triggered architecture consists of program modules that respond them periodically.

The time-triggered architecture is more suitable than the event-triggered architecture for hard real-time systems because of its predictable behavior[7]. However, there are many embedded control systems that contain not only periodically-activated program modules executed by time-triggered tasks but also eventually-activated program modules executed by event-triggered tasks. For example, an automotive engine control ECU (Electronic Control Unit) consists of time-triggered tasks such as calculation of physical values and event-triggered tasks such as injection and ignition outputs activated by inputs of a crank angle sensor. Those systems must be built based on the mixed architecture with time-triggered and event-triggered processing.

In an existing embedded control system, both periodic tasks and eventually activated tasks are managed by an operating system with fixed priority scheduling. A time-triggered task is a kind of periodic task, but it should be managed by static cyclic scheduling to reduce jitters. Furthermore, in a distributed control system, time-triggered tasks should be synchronized with the global time. Existing operating systems with fixed priority scheduling are not sufficient for the mixed architecture with time-triggered and event-triggered processing.

OSEK/VDX has presented the specifications of a time-triggered operating system called OSEKtime[17]. OSEKtime is based on static cyclic scheduling suitable for the time-triggered architecture. OSEK/VDX has also presented a layered operating system structure for the mixed architecture. The layered structure consists of OSEKtime and OSEK OS[16]. OSEK OS runs as an idle task of OSEKtime and manages event-triggered tasks with fixed priority scheduling. However, the layered operating system structure consumes more memory and causes more overhead than a monolithic operating system. Hattori et al.

*Presently with Hitachi, Ltd.

1533-2306/08 $25.00 © 2008 IEEE
DOI 10.1109/RTCSA.2008.38

have presented a monolithic operating system called TT-OS, which manages both time-triggered tasks and event-triggered tasks with priority-based scheduling[2]. The priority-based scheduling may consume more time than the static cyclic scheduling of OSEK-time. An efficient operating system for the mixed architecture is required.

A mixed architecture has been introduced to the communication protocol. FlexRay[11] is a hybrid type of protocol that is composed of static segments and dynamic segments. The static segment is used for time-triggered messages and the dynamic segment is used for event-triggered messages.

There are researches on scheduling for the mixed architecture with time-triggered and event-triggered processing. Lon et al. have done a comparison of fixed priority scheduling and static cyclic scheduling of tasks and communications for distributed automotive control applications[10]. Pop et al. have presented a holistic timing analysis and scheduling approaches for distributed heterogeneous time-triggered and event-triggered real-time systems[18].

There are, however, few distributed computing environments for the mixed architecture. CORBA[13] is widely used in distributed information systems. Real-Time CORBA[14], a real-time extension to CORBA, has been presented for real-time systems. CORBA also provides event service based on event channels[15]. CORBA event service utilizes dynamic routing and scheduling of event channels, so it is difficult to predict the delay times and to avoid the jitters. CORBA event service is not suitable for hard real-time systems.

We have already presented a time-triggered object model for embedded control systems in which the control logics are designed with block diagrams[20][21][22]. We have also already developed a time-triggered distributed object computing environment based on the model for automotive control systems[3]. An automotive control system consists of a number of ECUs (Electronic Control Units) connected with a real-time network. The environment provides replica-based location transparency. The time-triggered distributed object computing environment is suitable for hard real-time embedded control systems because network communications are not nested in the environment. The environment utilizes CAN[4] and the performance is sufficient for powertrain applications, not for x-by-wire applications. The environment cannot be applied to the mixed architecture because it does not support the event-triggered processing.

The goal of this paper is to develop a distributed object computing environment for embedded control systems based on the mixed architecture with time-triggered and event-triggered processing. The main target application domain is automotive control including x-by-wire applications.

To achieve the goal, we present distributed object models for the mixed architecture. We present two kinds of event-triggered distributed object models, a pure event-triggered distributed object model and a data-triggered distributed object model, in addition to the time-triggered distributed object model we have already presented,

We also present a distributed computing environment with time-division scheduling to run time-triggered tasks and event-triggered tasks concurrently. The environment consists of a RTOS (Real-Time Operating System) and distributed object computing middleware. We use FlexRay to support both time-triggered messages and event-triggered messages. We provide a development environment to develop distributed control systems based on the distributed object models efficiently.

The RTOS is an extension to OSEK OS. We have developed an extended scheduler and an extended dispatcher to support time-triggered tasks, in addition to non-time-triggered tasks of OSEK OS. The characteristics of the time-triggered task of our RTOS are similar to the time-triggered task of OSEKtime. The middleware is an extension to the time-triggered distributed object computing middleware[3]. The extended middleware provides distributed event service for event-triggered distributed processing.

The rest of the paper is organized as follows. The distributed object models and the time-division scheduling are presented in Section 2. Section 3 describes the structure of the environment, the RTOS, the middleware and the development environment. Section 4 describes the details of the distributed processing of the environment. Section 5 describes the implementation and the evaluation of the environment and Section 6 compares the environment with related work. In Section 7, we conclude the paper and mention future work.

2. Distributed Computing Model

2.1. Time-Triggered Distributed Object Model

Our main target applications are automotive control systems in which the control logics are designed with block diagrams. Block-diagram-based CAD/CAE tools such as MATLAB/Simulink[12] are widely used for automotive control design. Figure 1 shows an example of a block diagram. The block diagram consists of the block calculating *EngineTorque* and the block calculating *ThrottleOpening*. The calculation of each block is periodically executed in the control period (sampling period).

According to the development method we have presented[20][21], we convert a block diagram to a data flow diagram, then we identify objects referring to the data

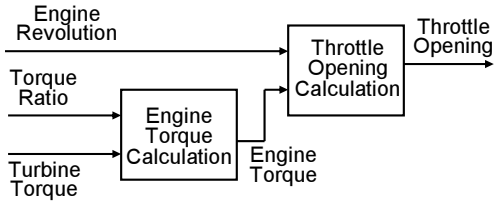

Figure 1. Example Block Diagram

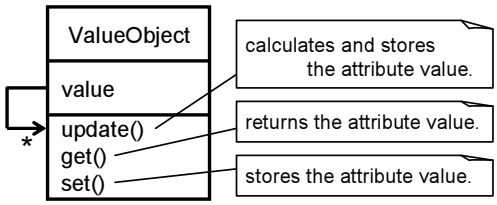

Figure 2. Base Class for Embedded Control Software

flow diagram. A data store in the data flow diagram is a candidate for a time-triggered object.

The time-triggered object model consists of objects that autonomously execute those operations periodically. Figure 2 shows the base class of time-triggered objects for embedded control software. The base class named *ValueObject* has an attribute named *value* and a method named *update* that calculates and stores the value of attribute *value*. Methods *get* and *set* are access methods to read and store the value of attribute *value*. The *update* method autonomously executes the calculation. The method has no arguments. If an object needs an another object's attribute, the former object calls the *get* method of the latter object.

Concrete classes of time-triggered objects for embedded control applications are subclasses of *ValueObject*. As shown later, the class is applicable not only to the time-triggered distributed processing but also to the event-triggered processing.

Figure 3 shows a combination of time-triggered objects for the control logic represented by Figure 1. For example, the *update* method of *ThrottleOpening* gets the value of *EngineRevolution* and the value of *EngineTorque* to calculate its own value (throttle opening). The *update* methods are invoked periodically in the control period.

In the time-triggered distributed object computing environment, replica objects are utilized for location transparency[3]. If an object in an ECU refers to another object in another ECU, a replica of the latter object is allocated in the former ECU. The former object refers to the replica object, not to the original object. The middleware of the time-triggered distributed object computing environment maintains the state of the replica to be consistent with the state of the original object.

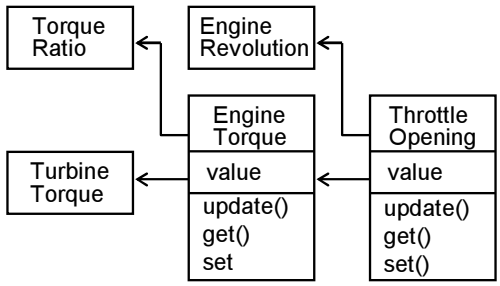

Figure 3. Example Class Diagram

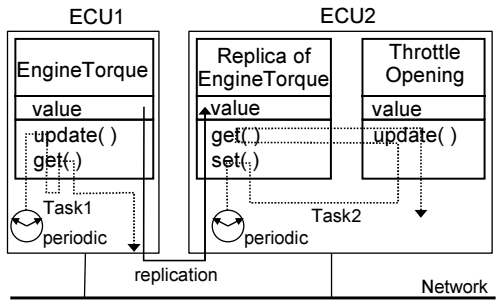

Figure 4. Example Time-Triggered Distributed Objects

Figure 4 shows an example of time-triggered distributed objects. In this example, two objects, *EngineTorque* and *ThrottleOpening*, are distributed to two ECUs. *EngineTorque* is located in *ECU1* and *ThrottleOpening* is located in *ECU2*. *Replica of EngineTorque* is located in *ECU2*, because *ThrottleOpening* in *ECU2* refers to *EngineTorque*.

Location transparency is provided with the replica. *ThrottleOpening* calls the *get* method of *Replica of EngineTorque* to get the value of *EngineTorque*. The state of the replica object is maintained to be consistent with the state of the original object by copying the attribute value of the original object to the replica.

Figure 5 shows an example time chart of processing for the time-triggered distributed object model. *Task1* of *ECU1* and *Task2* of *ECU2* are activated synchronously. In Figure 5, *Task1(n)* means *n*th job (cycle) of *Task1* and *Task2(n)* means *n*th job (cycle) of *Task2*. *Task2(n)* must be activated after *Task1(n)* completed because *Task2(n)* refers to the value calculated by *Task1(n)*. *Task2(n)* and *Task1(n+1)* are activated synchronously. Time-triggered tasks are statically scheduled[6].

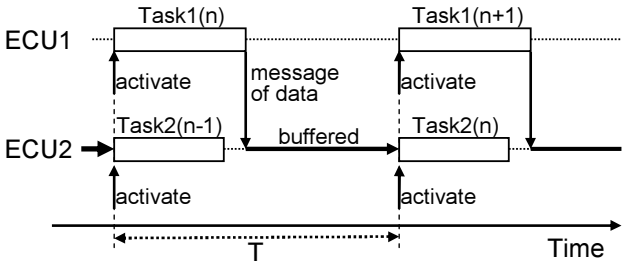

Figure 5. Time-Triggered Processing

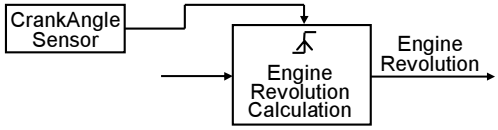

Figure 6. Example Block Diagram with Triggered Subsystem

2.2. Event-Triggered Distributed Object Model

In automotive control design, not only time-triggered processing but also event-triggered processing are designed with block-diagram-based CAD/CAE tools such as Simulink. There are no control flows in a pure block diagram, but Simulink provides notations for event-triggered processing such as a triggered subsystem. Figure 6 shows an example of a block diagram with a triggered subsystem. The block calculating *EngineRevolution* is triggered by the rising edge of *CrankAngleSensor*. The distributed object model for embedded control systems should be extended to support event-triggered processing.

We present two kinds of event-triggered distributed object models, a pure event-triggered distributed object model and a data-triggered distributed object model. The former is triggered by just events and the latter is triggered by events with data. For example, an object with a triggered subsystem of Simulink can be represented as an pure event-triggered distributed object model. On the other hand, an object activated by events of receiving data can be represented as an data-triggered distributed object model.

Figure 7 shows an example time chart of processing for the pure event-triggered distributed object model. *Task2* of *ECU2* is activated by message receive events.

Figure 8 shows an example of pure event-triggered distributed objects corresponding to the example shown in Figure 6. In this example, there are two objects, *CrankAngleSensor* and *EngineRevolution*. *CrankAngleSensor* is located in *ECU1* and *EngineRevolution* is located in *ECU2*.

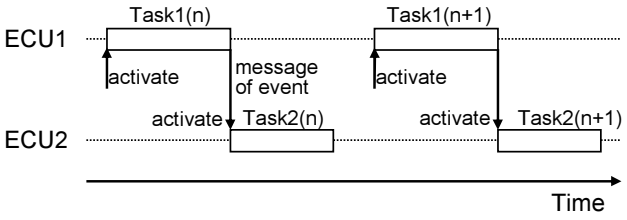

Figure 7. Event-Triggered Processing

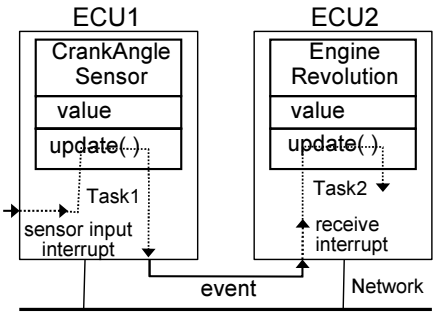

Figure 8. Example Event-Triggered Distributed Objects

The *update* method of *EngineRevolution* is executed when *ECU2* receives an event message sent by *CrankAngleSensor* in *ECU1*.

Figure 9 shows an example time chart of processing for the data-triggered distributed object model. *Task2(n)* of *ECU2* waits for the message sent by *Task1(n)* of *ECU1*. When the message is received by *ECU2*, *Task2(n)* resumes. In the case of *Task2(n+2)*, *ECU2* has received the message from *Task1(n+2)* of *ECU1*, so *Task2(n+2)* can get the data as soon as it requires the data.

Figure 10 shows an example of data-triggered distributed objects. *Replica of EngineTorque* is located in *ECU2* like the time-triggered distributed object model, and the state of the replica is maintained by the replication. When *ThrottleOpening* calls *get* method of *Replica of EngineTorque*, the execution of the *get* method waits for completing the

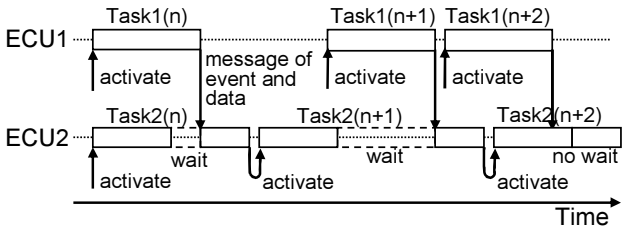

Figure 9. Data-Triggered Processing

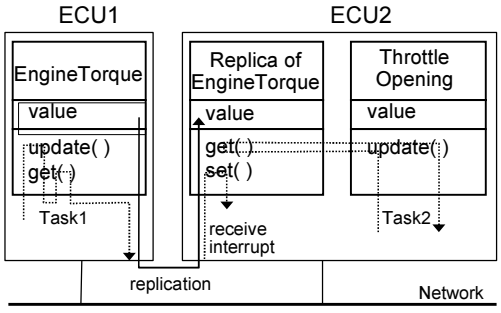

Figure 10. Example Data-Triggered Distributed Objects

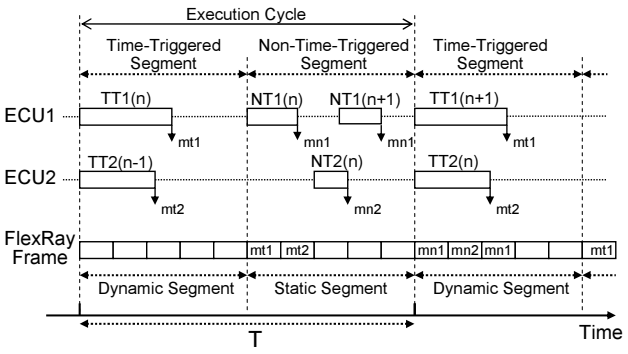

Figure 11. Time Division Scheduling

replication if *value* has not been updated. The execution of the *get* method does not need to wait if *value* has been updated.

2.3. Time Division scheduling

The jitter of a time-triggered task should be reduced because the performance of the control application depends on the jitter. So we give priority to reducing the jitter of time-triggered distributed processing over reducing the response time of event-triggered distributed processing.

We present time-division scheduling to execute both the time-triggered processing and the event-triggered processing and to reduce the jitter of the time-triggered processing. In this scheduling, an execution cycle consists of a time-triggered processing segment and a non-time-triggered segment. The execution cycle is repeated cyclically.

The time-division scheduling is applied to both the CPU scheduling and the network scheduling. The time division scheduling of the CPU is done by the RTOS shown in Section 3.2. The time division scheduling of the network is supported by FlexRay.

Figure 11 shows an example time chart of the time division scheduling. The period of the execution cycle is *T*. The timer of the RTOS of *ECU1* and the timer of the RTOS of *ECU2* are synchronized by FlexRay. In this example, the period of communication cycle of FlexRay is the same as the period of the execution cycle.

The execution cycle is divided into the time-triggered segment and the non-time-triggered segment. Tasks for the time-triggered processing are scheduled to execute in the time-triggered segment. Tasks for the event-triggered processing, the pure event-triggered processing and the data-triggered processing, are scheduled to execute in the non-time-triggered segment.

The communication cycle is divided into the static segment and the dynamic segment. Messages for the time-triggered distributed processing are transmitted in the static segment. Messages for the even-triggered distributed processing are transmitted in the dynamic segment. The network idle time is omitted in Figure 11.

In Figure 11, *TT1* and *TT2* are time-triggered tasks and *NT1* and *NT2* are non-time-triggered tasks. *TT2(n)* refers to the value calculated by *TT1(n)*, so *TT2(n)* is executed in the next execution cycle to the execution cycle in which *TT1(n)* is executed. *TT2(n)* and *TT1(n+1)* are activated simultaneously.

A number of periods are used in an embedded control system. The period of the execution cycle *T* is usually to be equal to the minimum period of the tasks, to put it more exactly, the greatest common divisor of the periods. In an automotive control system, periods $2^n T_{min} (n = 0, 1, 2, \cdots)$ are generally used, where T_{min} is the minimum period. So the period of the execution cycle is usually to be T_{min}. To reduce the response time of the event-triggered processing, $T_{min}/2^k (k = 0, 1, 2, \cdots)$ can be used for the period of the execution cycle *T*.

The time division scheduling can reduce the jitter of time-triggered processing because a time-triggered task is statically scheduled and not be preempted by other time-triggered tasks nor non-time-triggered tasks. The policy of the time division scheduling is the same as the scheduling of FlexRay. So the time division scheduling of tasks is suitable for distributed systems with FlexRay.

3. Software Structure

3.1. Overview

Figure 12 shows the structure of the distributed computing environment. The software consists of application program, distributed computing middleware, a RTOS, and a FlexRay network driver.

Application program consists of objects and stubs. Stubs are used to bridge objects and the middleware. Source code

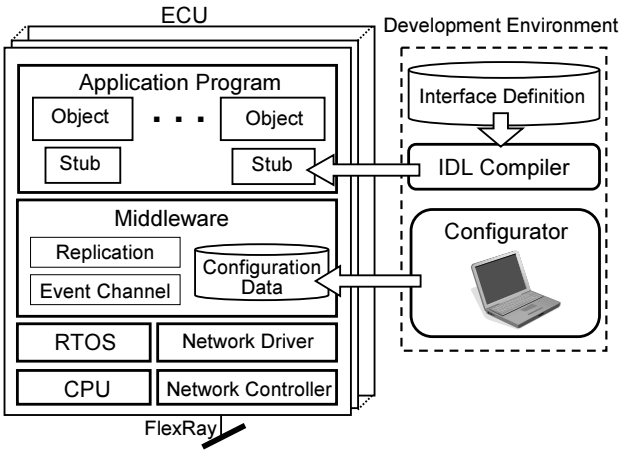

Figure 12. Distributed Computing Environment

files of stubs are automatically generated from interface definitions written in IDL (Interface Definition Language) by the IDL compiler.

The RTOS has a scheduler based on the time division scheduling. The middleware has a replication mechanism and an event channel mechanism for distributed event service. The middleware executes processing for replication and distributed event service referring to the configuration data. The configuration data are information on ECUs, tasks, events, messages, transmission periods, and so on. The configuration data are generated by the configurator.

3.2. Real-Time Operating System

The RTOS is an extension to TOPPERS/OSEK kernel, an OSEK-compliant RTOS developed by TOPPERS project[19]. We extend the scheduler and the dispatcher of TOPPERS/OSEK kernel to support the time division scheduling.

The RTOS manages both time-triggered tasks and non-time-triggered tasks. A time-triggered task managed by the RTOS is similar to a task of OSEKtime. A non-time-triggered task is a task of OSEK OS. The RTOS schedules and dispatches time-triggered tasks to run in the time-triggered segment and non-time-triggered tasks to run in the non-time-triggered segments as shown in Figure 11.

Figure 13 shows task management by the RTOS. The timer of the RTOS is synchronized with the hardware timer of FlexRay. So each RTOS timer is synchronized with the global time. The scheduler is executed by an ISR (Interrupt Service Routine) of category 1 specified by OSEK OS specifications. The ISR of category 1 is not managed by the operating system.

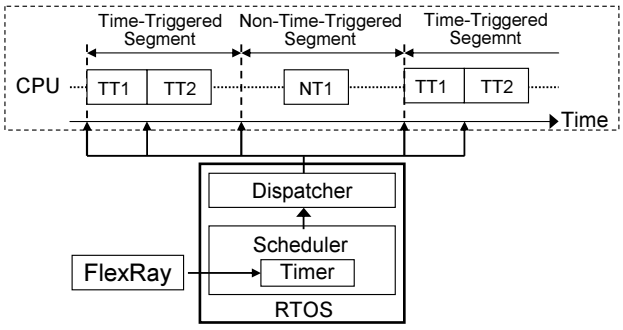

Figure 13. Real-Time Operating System

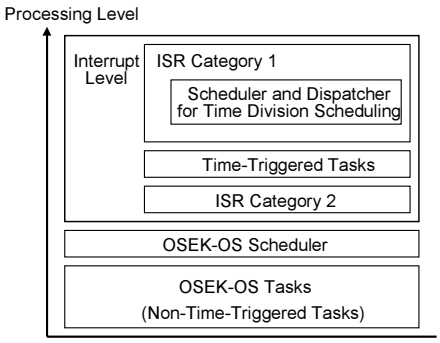

Figure 14. Processing Levels

The scheduler compares the time calendar of time-triggered tasks with the timer, set the flag of the task to be activated, and calls the dispatcher. The scheduler does not evaluate priorities nor queue time-triggered tasks, so the overhead of the scheduling of time-triggered tasks is less than the priority-based scheduling.

Figure 14 shows the processing levels of interrupts and tasks of the RTOS. The processing level of the time-triggered task is higher than the processing levels of ISR of Category 2, the OSEK OS scheduler and non-time-triggered tasks. The ISR of category 2 is managed by the operating system. A time-triggered task is executed by an ISR of Category 1. So the jitter of the time-triggered task can be reduced. We add a new operating system service for the ISR of Category 1 to activate a time-triggered task and the dispatcher for time-triggered tasks calls the operating system service.

3.3. Middleware

The middleware is an extension to the middleware of the time-triggered distributed computing environment. We add an event channel mechanism to provide the distributed event service. The distributed event service is an extension

Table 1. Distributed Event Service

Service	API	Arguments
set event	mw_SetEvent	event ID
wait event	mw_WaitEvent	event mask
clear event	mw_ClearEvent	event mask
activate task by event	mw_ActEvent	event ID

to the event service of OSEK OS. OSEK OS provides the local event service for inter-task synchronization. The event channel of the middleware utilizes OSEK OS event service for local event service.

Table 1 shows the API (Application Program Interface) of the distributed event service. API *mw_SetEvent*, *mw_WaitEvent* and *mw_ClearEvent* of the distributed event service are corresponding to OSEK OS event service API *SetEvent*, *WaitEvent* and *ClearEvent*. API *mw_ActEvent* of the distributed event service is a new service to activate a task by an event.

The event channel mechanism consists of an event router, an event communication module and event service API. The event router does routing of events. An event to a local task is routed to the task by using OSEK OS event service. An event to a task of another ECU is passed to the event communication module. The event communication module transmits the event to the destination task of another ECU via the network.

The distributed event service is utilized for the pure event-triggered distributed processing and the data-triggered distributed processing. The event communication module transmits not only events but also data related to the events. The event communication module also gets attribute values from objects and sets attribute values to objects. The detailed processing flows of the distributed event service are shown in the next section.

4. Distributed Processing

4.1. Time-Triggered Distributed Processing

Figure 15 shows the processing flow of the time-triggered distributed processing corresponding to Figure 4. The middleware maintains the state of *Replica of EngineTorque* in *ECU2* to be consistent with the state of *EngineTorque* in *ECU1*. Two kinds of stubs, the original stub and the replica stub, are used for replication of the time-triggered processing.

The middleware executes the replication of the time-triggered distributed processing as follows. The replication module of the middleware in *ECU1* calls the original stub (*pack()*) to get the attribute value of *EngineTorque* and

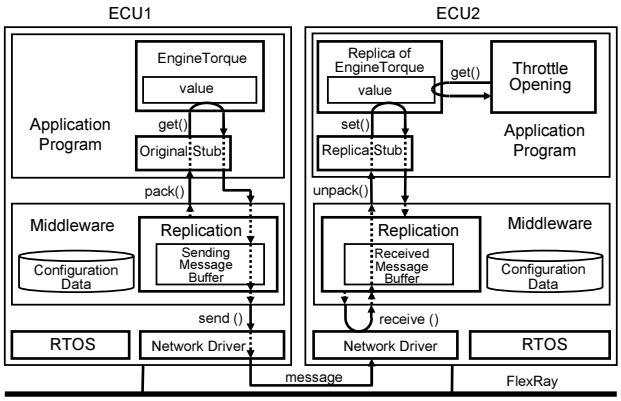

Figure 15. Time-Triggered Distributed Processing

stores the value in the sending message buffer. The replication module packs the replication data of objects into a message packet. After packing, the replication module calls the network driver (*send()*) to transmit the message.

The replication module of the middleware in *ECU2* calls the network driver (*receive()*) to get the received message and stores the message in the received message buffer. Then, the replication module calls the replica stub (*unpack()*) to unpack the message data and to set the unpacked attribute value to *Replica of EngineTorque*.

4.2. Event-Triggered Distributed Processing

Figure 16 shows the processing flow of the event-triggered distributed processing corresponding to Figure 8. *EngineRevolution* in *ECU2* is activated by the event from *CrankAngleSensor* in *ECU1*.

The middleware executes the event-triggered distributed processing as follows. *CrankAngleSensor* in *ECU1* is activated by the interrupt of the crank angle sensor synchronously to the engine revolution. *CrankAngleSensor* calls the event channel API of the middleware (*mw_ActEvent()*). The API module calls the event router. The event router determines the destination referring to the configuration data and then calls the event communication module. The event communication module calls the network driver (*send()*) to transmit the event message to the *ECU2*.

In *ECU2*, the message receive interrupt activates the event communication module of the middleware. The event communication module calls the network driver (*receive()*) to get the received event message, determines the event ID, and calls the event router specifying the event ID. The event

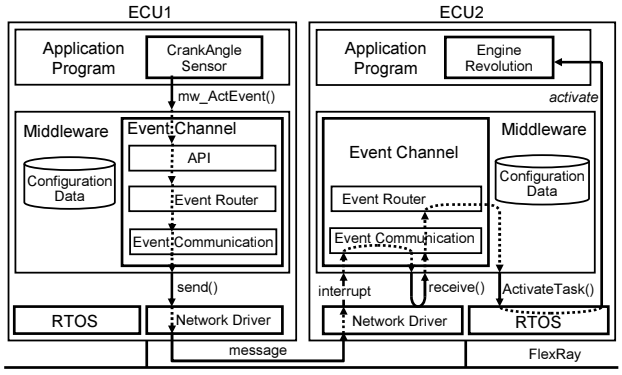

Figure 16. Event-Triggered Distributed Processing

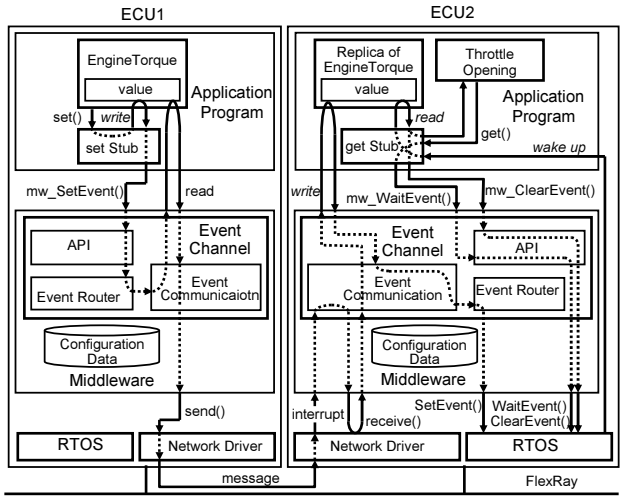

Figure 17. Data-Triggered Distributed Processing

router calls system service *ActivateTask()* of the RTOS to activate the task corresponding to the event ID. The activated task executes the method of *EngineRevolution*.

4.3. Data-Triggered Distributed Processing

Figure 17 shows the processing flow of the data-triggered distributed processing corresponding to Figure 10. When *ThrottleOpening* tries to get the attribute value of *Replica of EngineTorque*, it has to wait for the attribute value to be updated. If already updated, it does not have to wait. Two kinds of stubs, the set stub and the get stub, are used for the data-triggered distributed processing. The IDL compiler generates the stubs in which distributed event service calls are embedded.

The middleware executes the data-triggered distributed processing as follows. *EngineTorque* in *ECU1* calls the set stub (*set()*) to update its own attribute value. After storing the updated value in the attribute, the stub calls the event channel API (*mw_SetEvent()*) of the middleware. The API module calls the event router. The event router determines the destination referring to the configuration data and calls the event communication module. The event communication module determines the attribute value to send referring to the configuration data, gets the attribute value, and calls the network driver (*send()*) to transmit the message to *ECU2*.

In *ECU2*, the message receive interrupt activates the event communication module of the middleware. The event communication module calls the network driver (*receive()*) to get the received message, determines the destination replica referring to the configuration data, writes the value in the attribute of the destination replica (*Replica of EngineTorque*). The event communication module also determines the event ID and then calls the event router specifying the event ID. The event router determines the task and the event mask corresponding to the event ID referring to the configuration data, and calls system service *SetEvent()* of RTOS to set the event.

ThrottleOpening, on the other hand, calls the get stub (*get()*) of *Replica of EngineTorque* to get the attribute value of *EngineTorque*. The stub calls the event channel API (*mw_WaitEvent()*) of the middleware. The API module calls system service *WaitEvent()* of RTOS to wait the event.

If the event has been already set, the task executing the get stub proceeds without waiting. If the event has not been set, the task waits for the event. When the event is set, the task is woken up. The get stub reads the updated attribute value, calls the event channel API (*mw_ClearEvent()*) of the middleware, and returns the attribute value. When *mw_ClearEvent()* is called, the API module calls system service *ClearEvent()* of RTOS.

5. Implementation and Experimental results

At first, we implemented the distributed event service with TOPPERS/OSEK kernel on a CAN evaluation board with a microcontroller called H8S/2638, on which we had developed the time-triggered distributed object computing environment. Then, we ported the distributed event service to a FlexRay evaluation board, which consists of CPU V850 (50MHz) and a FPGA FlexRay controller. We have developed the extended middleware on the FlexRay evaluation board and have done experiments to evaluate the performance of the middleware.

Table 2 shows the execution time of transmission and

Table 2. Execution Time of Communication of Middleware

Processing	Ave.	Max.	Min.	Dif.
Time-Triggered: Transmit	43.20	43.28	43.20	0.08
Time-Triggered: Receive	47.04	47.04	47.04	0.00
Pure Event-Triggered: Transmit	33.12	33.20	33.12	0.08
Pure Event-Triggered: Receive	6.00	6.08	6.00	0.08
Data-Triggered: Transmit	35.04	35.12	35.04	0.08
Data-Triggered: Receive	3.20	3.20	3.20	0.00

[μsec], resolution:0.02μsec

Table 3. Time-Triggered Task Activation Time of RTOS

OS	Condition	Time [μsec]
Extended OS	No Preemption	26.5
Extended OS	With Preemption	37.7
OSEK OS	No Preemption	63.1
OSEK OS	With Preemption	82.4

reception of the middleware, except for the execution time of RTOS, in the cases of time-triggered processing, event-triggered processing and data-triggered processing. The table shows the average values, the maximum values, the minimum values, and the differences between the maximum and the minimum values.

The differences of the middleware execution times cause the jitters. The jitters caused by the middleware should be adequately less than the jitters caused by application program. The difference time 0.08μsec means four machine cycles of the CPU. We think that the jitters caused by the middleware can be accepted for automotive control systems, because the time of four machine cycles is generally less than the difference of application program execution time.

We have developed the RTOS with the time division scheduling as an extension to TOPPERS/OSEK kernel on a evaluation board with CPU M16C/26(20MHz). We are porting the RTOS to the FlexRay evaluation board.

Table 3 shows the average time to activate a time-triggered task of the RTOS on the evaluation board with M16C/26. *Extended OS* means the RTOS we have developed. *No Preemption* means the case in which there are no running tasks and preemption is not occurred. *With Preemption* means the case in which there is a running task and preemption is occurred.

Table 3 also shows the average time to activate a task of OSEK OS (TOPPERS/OSEK kernel) on the same evaluation board for comparison. A periodic alarm is used to activate a periodic task in the case of OSEK OS. Table 3 shows the performance of the time-triggered task activation of our RTOS is better than the performance of the periodic task activation of OSEK OS. This is because the scheduler of our RTOS does not evaluate the priorities of time-triggered tasks as shown in Section 3.2. The time to activate a non-time-triggered task in our RTOS is the same as in OSEK OS.

6. Comparison with Related Work

Several real-time object models with periodical and eventual invocation have been presented. Callison has presented a TSO (Time-Sensitive Object) model[1]. Kim has presented a TMO (Time-Triggered Message-triggered Object)[5]. The inter-object communications of the models are based on the client-server model. The client-server-based distributed object models are, however, not suitable for control applications designed with block diagrams. Our distributed object models, time-triggered, pure event-triggered and data-triggered models, are suitable for the block-diagram-based control application design.

OSEK/VDX has presented a layered operating system structure that consists of OSEKtime[17] for time-triggered processing and OSEK OS[16] for non-time-triggered processing. The layered structure, however, consumes more memory. Our RTOS, an extension to OSEK OS, can decrease the memory consumption.

Hattori et al. has presented an operating system called TT-OS, which manages both time-triggered tasks and event-triggered tasks with priority-based scheduling[2]. We estimate the time for TT-OS to activate a time-triggered task is almost the same as the time for OSEK OS to activate a periodic task shown in Table 3. The time-triggered task activation of our RTOS is more efficient than TT-OS because our RTOS activates a time-triggered task without priority evaluation.

Some distributed object computing environments based on Real-Time CORBA have been presented for embedded systems. Lankes et al. have presented a Real-Time CORBA with a time-triggered Ethernet[8] and a CAN-based CORBA[9] for embedded systems. The inter-object communication in their environment is based on the client-server model with RPC. If RPC-based inter-object communications are nested, it is difficult to predict the delay times and to avoid the jitters. RPC-based distributed object computing environments are not suitable for hard real-time systems. On the other hand, the inter-object communication of our distributed object computing environment is one-way communication and not nested. So the environment is suitable for hard real-time systems.

Our distributed object computing environment supports time-triggered and event-triggered distributed processing.

The event channel of the environment is based on the static routing, which is more predictable than CORBA event service[15] with dynamic routing. So the environment is suitable for distributed embedded control systems.

7. Conclusions

We have presented distributed object models for the mixed architecture with time-triggered and event-triggered processing. We have developed a distributed object computing environment for the time-triggered and event-triggered distributed object models. The environment consists of a RTOS with time-division scheduling and distributed computing middleware with replication and distributed event service. We have also developed a development environment, an IDL compiler and a configurator. We are going to develop a scheduling method for the mixed architecture based on the time division scheduling.

Acknowledgments

This work is partially supported by KAKENHI (20500037). We would like to thank the developers of TOPPERS/OSEK kernel for the open source code of the kernel, which is a base operating system of our RTOS.

References

[1] Callison, H. R., A Time-Sensitive Object-Model for Real-Time Systems, *ACM Transaction on Software Engineering and Methodology*, Vol. 4, No. 3, pp. 287–317, 1995.

[2] Hattori, H., Ohnisi S., Morikawa A., Nakamura K. and Takada, H., Open Source FlexRay Communication: Time Triggered OS and FlexRay Communication Middleware, *Proceedings of IP-Based SoC Design Conference (IP/SOC 2006)*, pp. 227–233, 2006.

[3] Ishigooka, T. and Yokoyama, T., A Time-Triggered Distributed Object Computing Environment for Embedded Control Systems, *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 191–198, 2007.

[4] Kiencke, U., Controller Area Network - from Concept to Reality, *Proceedings of 1st International CAN Conference*, pp. 0-11–0-20, 1994.

[5] Kim, K. H., Object Structures for Real-Time Systems and Simulators, *IEEE Computer*, vol.30, no.8 pp.62–70, 1997.

[6] Kopetz, H., Damm.A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C. and Zainlinger, R., Distributed Fault-Tolerant Real-Time Systems: The Mars Approach, *IEEE Micro*, Vol. 9, No.1, pp.25–40, 1989.

[7] Kopetz, H., Should Responsive Systems be Event-Triggered or Time-Triggered?, *IEICE Transaction on Information & Systems*, Vol. E76-D, No. 11, pp. 1325–1332, 1993.

[8] Lankes, S., Jabs, A. and Reke, M., A Time-Triggered Ethernet Protocol for Real-Time CORBA, *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pp.215–222, 2002.

[9] Lankes, S., Jabs, A. and Bemmerl, T., Integration of a CAN-based Connection-oriented Communication Model into Real-Time CORBA, *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, p.123a, 2003.

[10] Lonn, H. and Axelsson, J., A Comparison of Fixed-Priority and Static Cyclic Scheduling for Distributed Automotive Control Applications, *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp.142–149, 1999.

[11] Makowitz, R. and Temple, C., FlexRay - A Communication Network for Automotive Control Systems, *Proceedings of 2006 IEEE International Workshop on Factory Communication Systems*, pp. 207–212, 2006.

[12] Moscinski, J., *Advanced Control with MATLAB and Simulink*, Ellis Horwood, Ltd., 1995.

[13] OMG Technical Document formal/02-06-01, *The Common Object Request Broker: Architecture and Specification, Version 3.0*, 2002.

[14] OMG Technical Document formal/02-08-02, *Real-Time CORBA Specification, Version 1.1*, 2002.

[15] OMG Technical Document formal/04-10-02, *Event Service Specification, Version 1.2*, 2004.

[16] OSEK/VDX, *Operating System, Version 2.2.3*, 2005.

[17] OSEK/VDX, *Time-Triggered Operating System, Version 1.0*, 2001.

[18] Pop, T., Eles, P. and Peng, Z., Schedulability Analysis for Distributed Heterogeneous Time/Event Triggered Real-Time, *Proceedings of 15th Euromicro Conference on Real-Time Systems*, pp.257–266, 2003.

[19] TOPPERS Project, http://www.toppers.jp/en/index.html

[20] Yokoyama, T., Naya, H., Narisawa, F., Kuragaki, S., Nagaura, W., Imai, T. and Suzuki, S., A Development Method of Time-Triggered Object-Oriented Software for Embedded Control Systems, *Systems and Computers in Japan*, Vol. 34, No. 2, pp. 338–349, 2003.

[21] Yokoyama, T., An Aspect-Oriented Development Method for Embedded Control Systems with Time-Triggered and Event-Triggered Processing, *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Application Symposium*, pp. 302–311, 2005.

[22] Yoshimura, K., Miyazaki, T., Yokoyama, T., Irie, T. and Fujimoto, S., A Development Method for Object-Oriented Automotive Control Software Embedded with Automatically Generated Program from Controller Models, *2004 SAE (Society of Automotive Engineers) World Congress*, 2004-01-0709, 2004.