

Bluetooth Loader for mruby Bytecode in Multi-VM Environment

Takuro Yamamoto
Graduate School of
Engineering Science, Osaka
University
Osaka, JAPAN

Hiroshi Oyama
OKUMA Corporation
Aichi, JAPAN

Takuya Azumi
Graduate School of
Engineering Science, Osaka
University
Osaka, JAPAN

ABSTRACT

In recent years, the productivity of embedded systems has a problem due to their complexity and large-scale. For the purpose of improving the productivity for embedded software development, the mruby on TECS framework has been proposed that is applied mruby (Lightweight Ruby) and supports component-based development. In the current mruby on TECS, the mruby programs have to be compiled and linked every time the programs are modified because the mruby bytecodes are incorporated in the platform. Moreover, while the framework supports multi-VM, developers need to be familiar with the functions of real-time operating systems to effectively execute multiple mruby programs in concurrent or/and parallel. To improve the development efficiency, this paper proposes a Bluetooth loader for mruby bytecode as an extension of mruby on TECS. The loader executes two mruby bytecodes, mruby application bytecode and mruby library bytecode. mruby application bytecode is modified frequently and sent from a host to a target device by developers. mruby library bytecode modified infrequently is preserved beforehand with the platform in a storage/ROM device at the time of the first compilation. In addition, multiple mruby programs cooperatively run in the proposed framework. A RiteVM scheduler makes multitasking processing more easy-to-use than that of mruby on TECS. Synchronization of starting multiple tasks is also implemented with Eventflag. Experimental results demonstrate the advantages of the proposed framework.

Keywords

embedded software; script language; component-based development;

1. INTRODUCTION

In these days, embedded systems have been required the high-quality and the high-performance. Due to this trend, the complexity of embedded systems also increases and the

scale is larger. For example, IoT (Internet of Things) applications. The low production cost and the short developing period of time are also needed.

An approach of efficient software development is to use component-based techniques. CBD (Component-Based Development) is a design technique for constituting reusable components [16]. Complex and large scale software systems can be developed efficiently using component-based techniques. That is because software componentization provides high-reusability. Verification of component-based systems has been researched [18] [15]. It also visualizes the entire system due to component diagrams, and makes a system flexible in extensions and specification changes. There are TECS [13], AUTOSAR [1], and SaveCCM [21] as a typical component-based development for embedded systems.

In addition, another approach of efficient software development is to develop with script languages. Most of current software are programmed in C language. Development with C language makes the code size increase, and takes a high cost and more time to develop. Script languages make software engineering more efficient and shorten a development period because script languages have high-productivity from their usability. Ruby, JavaScript, Perl, Python, and Lua are well-known as representative script languages.

Although script languages are easy to use and read, their execution time are larger than C language's. For embedded systems, the real-time properties such as estimation of worst-case execution time are very important factors. Therefore, it is difficult to apply the script languages to embedded systems.

To improve the above problems, mruby on TECS has been proposed, which is a component-based framework for running script program [11]. It is integrated two technologies. One is mruby, which is a script language for embedded systems [25], [5]. The other is TECS (TOPPERS Embedded Component System), which is a component-based framework for embedded systems [13] [10]. mruby on TECS supports to effectively run mruby script language on embedded systems. mruby on TECS also makes execution time 100 times faster than that of mruby.

mruby on TECS has several problems at present. One of the problems is low software development efficiency. mruby on TECS only supports a storage/ROM device to load mruby programs. For example, an SD card should be inserted and pulled out repeatedly, or a ROM should be rewritten if mruby programs are modified. Developers should also restart an RTOS on a target device. Moreover, although

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EMSOFT '16 October 2–7, 2016, Pittsburgh, USA

© 2016 ACM. ISBN 123-4567-24-567/08/06...\$15.00

DOI: 10.475/123_4

mruby on TECS has supported the multi-VM, developers need to call OS's function in order to execute multiple tasks. These problems makes the burden heavy for developers.

This paper proposes an extended framework of mruby on TECS: a Bluetooth loader for mruby bytecode and a RiteVM scheduler for fairly executing mruby programs. To improve the development efficiency, the proposed framework enables developers to implement the platform in a storage device only once at the beginning and to transfer mruby application programs from a host to a target device using the Bluetooth Loader. A RiteVM is a virtual machine for mruby. The RiteVM scheduler manages the execution of multiple RiteVMs, and enables developers to program multitasking more easily than mruby on TECS.

Contribution: The proposed framework gives the contribution in the following points.

1. **To improve the software development efficiency:** Developers do not need to rewrite a storage/ROM device and also to restart an RTOS. The loader supports the continuous loading, which saves the Bluetooth set-up time. Therefore, the Bluetooth loader for mruby bytecode helps developers develop software.
2. **To effectively execute multiple mruby programs in concurrent or/and parallel:** Developers can implement multiple tasks without the knowledge of an RTOS because the RiteVM scheduler switches tasks cyclically.
3. **To synchronize the execution of multiple RiteVM tasks:** The proposed framework provides the synchronization of multiple RiteVM tasks (mruby applications).
4. **To focus on the benefits of component-based developments:** This paper shows the specific examples for the benefits of component-based development.

Organization: The paper is organized as follows. Section 2 introduces the basic technologies i.e. mruby, TECS and mruby on TECS. Section 3 describes the design and implementation of the proposed framework in detail. Section 4 evaluates the proposed framework, Section 5 discusses related work, and then Section 6 concludes this paper.

2. BACKGROUND

Figure 1 shows the system model of the proposed framework. The bytecodes are transferred from the host to the target device with Bluetooth. The RiteVMs and mruby library are assumed to be prepared in advance. Each bytecode is allocated to a RiteVM, respectively. Bytecodes transferred with Bluetooth from the host can run in multitask.

This section describes mruby on TECS on which the proposed framework is based. mruby on TECS is a component-based framework for running script programs. In mruby on TECS, two technologies are integrated: mruby and TECS. To explain the system, mruby and TECS are also respectively described in this section.

2.1 mruby

mruby is the light-weight implementation of Ruby programming language complying to part of the ISO standard. Ruby is a object-oriented script language [8]. As the main feature, Ruby is easy-to-use and easy-to-read due to its simple grammar. Ruby can run a program with fewer code lines

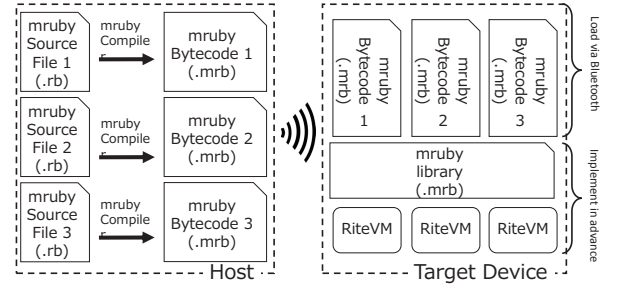


Figure 1: System Model of the proposed framework

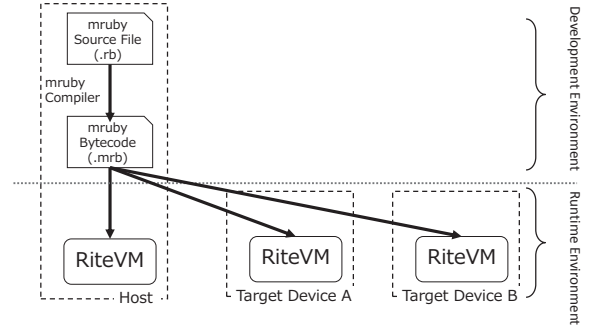


Figure 2: Mechanism of mruby/RiteVM

than C language. Ruby improves the productivity of a software development owing to not only simple grammar but also object-oriented functions such as classes and methods, exceptions, and garbage collection.

mruby is suitable for embedded systems because of execution with less amount of resources and takes over the usability and readability of Ruby. In addition, VM (Virtual Machine) mechanism is used in mruby, therefore mruby programs can run on any operating system as long as VM is implemented. A RiteVM is a VM in mruby, that runs mruby programs. The RiteVM mechanism is shown in Figure 2. The mruby compiler translates an mruby code into a bytecode, which is an intermediate code that can be interpreted by a RiteVM. The bytecodes can run on a RiteVM, and thus mruby programs can be executed on any target devices if only RiteVMs are implemented.

2.2 TECS

TECS is a component system suitable for embedded systems. TECS helps increase productivity and reduce development duplication because the reusability of software is improved. TECS also helps developers understand the system because the generated component diagram can visualize the structure of whole software.

The component deployment and composition in TECS are statically performed, which gives optimization. As a result, the overhead of connection of components and the memory consumption can be reduced. There are other features of TECS, implementation in C language, source level portability, fine-grained component, etc.

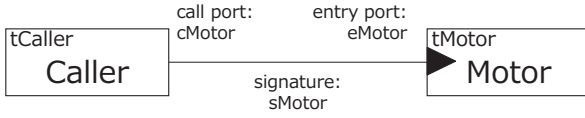


Figure 3: Component Diagram

```

1 signature sMotor {
2   int32_t getCounts( void );
3   ER resetCounts( void );
4   ER setPower( [in]int power );
5   ER stop( [in] bool_t brake );
6   ER rotate( [in] int degrees, [in] uint32_t speed_abs,
7             [in] bool_t blocking );
8   void initializePort( [in]int32_t type );
9 };

```

Figure 4: Signature Description

2.2.1 Component Model

Figure 3 shows a component diagram. A *cell* which is an instance of a component in TECS consists of *entry* ports, *call* ports, attributes and internal variables. A *entry* port is an interface to provide functions with other *cells*, and a *call* port is an interface to use functions of other *cells*. A *cell* has one or more *entry* ports and *call* ports. Functions of a *cell* are implemented in the C language.

A type of a *entry/call* port is defined by a *signature* which is a set of functions. A *signature* is the interface definition of a *cell*. The *call* port of a *cell* can be connected to the *entry* port of another *cell* with the same *signature*. A *celltype* is the definition of a *cell*. *Celltype* defines one or more *call/entry* ports, attributes and internal variables.

2.2.2 Component Description

The description of a component in TECS is divided into three parts, *signature*, *celltype*, and build description. TECS code is written in CDL (component description language) file. The component description is mentioned with an example shown in Figure 3 as follow.

Signature Description

The *signature* description defines an interface of a *cell*. A *signature* name is described following the keyword *signature*. It also has the prefix "s". In this way, a *signature* is defined such as sMotor shown in Figure 4. To make the definition of an interface clear, specifiers such as in and out are used in TECS. [in] and [out] represent input and output, respectively.

Celltype Description

The *celltype* description defines *entry* ports, *call* ports, attributes, and variable of a *celltype*. An example of a *celltype* description is shown in Figure 5. A *celltype* name following the keyword *celltype* with the prefix "t" and elements of a *celltype* is described. To define *entry* ports, a *signature* such as sMotor, and an *entry* port name such as eMotor follow the keyword *entry*. In the same way, *call* ports can be defined. Attributes and variables follow the keyword *attr* and *var* respectively.

```

1 celltype tCaller {
2   call sMotor cMotor;
3 };
4 celltype tMotor {
5   entry sMotor eMotor;
6   attr {
7     int32_t port;
8   };
9   var {
10    int32_t currentSpeed = 0;
11  };
12 };

```

Figure 5: Celltype Description

```

1 cell tMotor Motor {
2   port = C_EXP("PORT_A");
3 };
4 cell tCaller Caller {
5   cMotor = Motor.eMotor;
6 };

```

Figure 6: Build Description

Build Description

The build description is used to instantiate *cells* and connect *cells*. Figure 6 shows an example of a build description. A *celltype* name such as tMotor, and a *cell* name such as Motor follow the keyword *cell*. To compose *cells*, a *call* port, a *signature*, a *entry* port in order are described. In this example, a *entry* port eMotor in a *cell* Motor is connected to a *call* port cMotor in a *cell* Caller. The *C_EXP* is the keyword to use macros defined in C files.

2.3 mruby on TECS

mruby on TECS is a component-based framework for running script language. This framework uses two technologies, mruby and TECS.

2.3.1 System Model

The system model of mruby on TECS is shown in Figure 7. Each mruby program, which is bytecode, runs on its own RiteVM as a componentized task of an RTOS. TECS components support various embedded drivers such as motor and sensor drivers.

An mruby-TECS bridge plays a role to call a native program (e.g. C legacy code) from an mruby program. The mruby-TECS bridge provides native libraries for mruby. It also gives TECS components to receive the invocation from an mruby program. The mruby-TECS bridge is described in more detail below.

As the target RTOS, TOPPERS/HRP2 [27], [20] is used in this paper. TOPPERS/HRP2 is an RTOS based on μ ITRON [24] with memory protection. However, mruby on TECS does not depend on the RTOS because TECS supports not only TOPPERS/HRP2 but also the other RTOSs such as OSEK [23] and TOPPERS/ASP [12], [26].

2.3.2 mruby-TECS Bridge

There is a great difference between the execution time of mruby and C language. According to [11], mruby programs

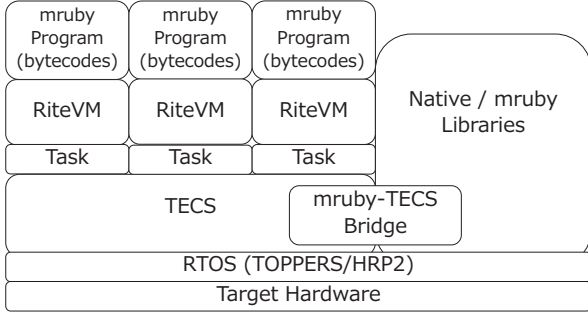


Figure 7: System Model of existing mruby on TECS



Figure 8: mruby-TECS Bridge

are several hundreds times slower than C programs. The execution of mruby bytecode on a RiteVM is not as efficient as that of C language. Thus it is difficult to use mruby for all codes.

The use of Ruby on embedded devices provides the benefit of productivity and maintainability due to the ease to use and read. On the other hand, it is necessary to implement parts of applications in C language in order to manipulate actuators and sensors, and also make a critical section of codes run quickly.

Figure 8 shows an example of use of an mruby-TECS bridge for controlling a motor. The left side of BridgeMotor belongs to the mruby program. The right side of BridgeMotor belongs to TECS component.

The mruby-TECS bridge generates two things. One is a *celltype* to receive invocation from the mruby program. The other is an mruby class that corresponds to the TECS component specified by the developers to invoke a C function from the mruby program.

A code of an mruby-TECS bridge is generated. The generation code supports registration of classes and methods for mruby. The methods in an mruby class are defined by generation codes for an mruby-TECS bridge, such as *setPower* and *stop*. Thus, when a method is called in an mruby program, an mruby-TECS bridge calls the function defined in the TECS component such as a *Motor cell*.

3. DESIGN AND IMPLEMENTATION

Figure 9 shows the detailed system model of the proposed framework. Each mruby application bytecode transferred from the host to the target devices received by the loader in the RiteVM. The RiteVM reads the transferred bytecode and executes it with libraries. The mruby applications run at the same time because of synchronization processing. The RiteVM scheduler switches RiteVM tasks as multiple tasks can run in concurrent. This section explains these proposed functionalities.

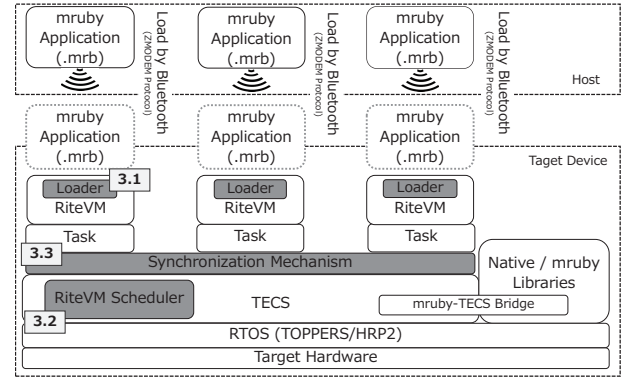


Figure 9: Detailed System Model of the proposed framework

3.1 Bluetooth Loader for mruby Bytecode

This section describes an additional functionality of mruby on TECS, Bluetooth loader for mruby bytecode.¹ In the current system, the platform including mruby bytecodes are saved in a storage/ROM device. Developers must rewrite the storage/ROM device every time the application programs are modified. The RTOS on the target device should be also restarted. It causes a low development efficiency to repeat that. A Bluetooth loader for mruby bytecode makes the developers' burden decrease. Developers should just once carry out the work to connect the storage/ROM device and start the RTOS.

mruby programs are consisted of mruby application and libraries. mruby application is the main code which software developers should program. mruby libraries are the codes defining the functions for application such as Ruby classes. The mruby bytecodes including mruby application and libraries can be sent and run. However, it is also wasteful in terms of the size and time to send bytecodes because libraries are not frequently modified. The proposed framework provides the design that only mruby applications are sent and mruby libraries are preserved with the platform in the storage/ROM device beforehand. Due to this design, RiteVMs can share the mruby libraries. In addition, a RiteVM can use the own library that other RiteVMs should not use.

In the proposed framework, the platform including RiteVMs and mruby library is compiled and copied in the storage/ROM device at the first. In the host, the mruby application programs (.rb) are edited and compiled into the bytecodes (.mrub) by an mruby compiler. The generated bytecodes are transferred from the host to the target device with Bluetooth.

Moreover, developers can save the time of Bluetooth pairing since the loader can continuously load the bytecode.

3.1.1 Component of RiteVM with Bluetooth loader for mruby bytecode

The proposed framework provides a RiteVM with Bluetooth loader for mruby bytecode as TECS component. The component is an extension of the RiteVM component, which is described in [11]. The component plays a role in receiving

¹Bluetooth loader is intended to improve the development efficiency, therefore software developers should use it for development phase. Complete software should be compiled and linked in the storage/ROM device beforehand.



Figure 10: Component Diagram of Bluetooth loader for mruby bytecode

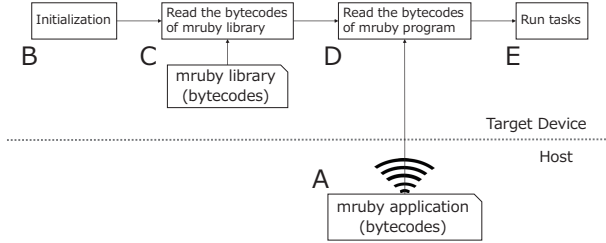


Figure 11: Process Flow of Bluetooth loader for mruby bytecode

ing bytecodes via Bluetooth, and also manages a RiteVM configuration such as automatically generating the mruby library bytecode. This generated bytecode is prepared beforehand in the storage/ROM device, and different from a bytecode transferred with Bluetooth.

Figure 10 shows a component diagram of MrubyTask1 and RiteVMBluetooth1 *cells*. The MrubyTask1 *cell* is a componentized task of the RTOS (TOPPERS/HRP2). TOPPERS/HRP2 is described in [27], [20]. The RiteVMBluetooth1 is the component of RiteVM with Bluetooth loader for mruby bytecode. A bytecode in the host is transferred and received at the top of the component. In this framework, ZMODEM [17] is used as a binary transfer protocol.

Figure 11 shows the process flow of executing mruby program in the component of RiteVM with Bluetooth loader for mruby bytecode such as RiteVMBluetooth1. The concrete main code of tRiteVMBluetooth is shown in Figure 13.

First, the Bluetooth loader receives the mruby application bytecode transferred from the host (Figure 11:A, Lines 6-7 in Figure 13). The bytecode is stored in the component variable such as `VAR_irepApp` shown in Figure 12. This process is exclusively carried out by the semaphore.

Second, pointers of `mrub_state` and `mrbc_context`, and mruby-TECS bridges are initialized (Figure 11:B, Lines 9-14 in Figure 13). `VAR_mrb` and `VAR_context` shows the variable of the *cell*. `mrub_state` is a set of states and global variables used in mruby. Synchronization of multiple tasks is performed in this processing phase. The RiteVM that ended up here waits for the other RiteVM to finish loading and initialization.

Third, the RiteVM reads the bytecode of mruby libraries (Figure 11:C, Lines 17-19 in Figure 13). mruby libraries are a set of Ruby classes such as motor class and sensor class. For example, motor class defines methods to rotate and stop a motor. The tRiteVMBluetooth *cell* has attributes as shown in Figure 12. *ATTR* shows the attribute, which is a fixed value and not rewritten unlike *VAR*. The `mrubyLib` indicates the program files of mruby libraries. `mrubyLib` is the attribute because mruby libraries are not modified in the proposed development process. *[omit]* is only used for the TECS generator, thus the attribute, `mrubyLib`, does not con-

```

1  /* tRiteVMBluetooth.cdl */
2  celltype tRiteVMBluetooth1 {
3      entry sTaskBody eMrubyBody;
4      [optional] call sEventflag cEventflag[];
5      [optional] call sSemaphore cSemaphore;
6      attr {
7          [omit]char_t *mrubyLib;
8          char_t *irepLib =
9              C_EXP("&$cell_global$_irep");
10         uint32_t irepAppSize =
11             C_EXP( BUFFER_SIZE );
12         FLGPTN setptn;
13     };
14     var {
15         mrub_state *mrub;
16         mrbc_context *context;
17         [size_is(irepAppSize)] uint8_t *irepApp;
18     };
19 };

```

Figure 12: Celltype Description for RiteVM with Bluetooth loader for mruby bytecode

sume memory. The `irepLib` is the pointer of the array stored the bytecode of mruby libraries. In short, the bytecode of mruby libraries is stored as an attribute of the component when compiling for the first time.

Fourth, the RiteVM reads the bytecode of the mruby application transferred with Bluetooth (Figure 11:D, Lines 20-22 in Figure 13). The mruby application bytecode is stored in an array of `irepApp`. The array is different from that of holding the mruby library bytecode. Two bytecodes are read separately in the RiteVM.

Finally, the mruby task runs (Figure 11:E, Lines 20-22 in Figure 13). When the mruby application is modified, only the bytecode of the modified application should be transferred. mruby libraries need not be touched because libraries are not normally changed.

The process shown in Figure 13:23-28 is carried out when an exception occurs. When all mruby applications finish, `mrub_state` and `mrbc_context` are freed (Figure 13:31-34). The proposed framework supports continuous loading, thus this process loops. After freeing variables, RiteVM will become the state to wait for the next mruby application bytecode.

3.2 RiteVM Scheduler

This section describes implementation of RiteVM scheduler in the proposed framework. mruby on TECS has supported multitasking. However, multitask processing in mruby on TECS requires the knowledge of the RTOS (TOPPERS/HRP2) for developers.

One of approaches for multitasking is co-routine. Co-routine is a cooperative thread, and scheduled by developers with the functions such as `resume` and `yield`. (Ruby co-routine is defined in class Fiber [2]) Co-routine is a non-preemptive multitasking, and does not receive the OS's support because developers have to switch tasks manually. Co-routine can not take advantage of multi core processing.

Besides, as another method, `delay()`, a service call of μ ITRON, can be used for multitasking. This service call delays the execution of the own task for the time of the argument. `delay()` is needed when scheduling fixed-priority tasks. However, the

```

1  /* tRiteVMBluetooth.c */
2  void
3  eMrubyBody_main( CELLIDX idx )
4  {
5      /* Omit: start of exclusive process by semaphore */
6      /* Receive the bytecode via Bluetooth */
7      bluetooth_loader( VAR_irepApp );
8      /* Omit: end of exclusive process by semaphore */
9      /* New interpreter instance */
10     VAR_mrb = mrb_open();
11     /* Omit: error check for mrb_state */
12     /* New mruby context */
13     VAR_context = mrbc_context_new( VAR_mrb );
14     /* Omit: initialization of mruby-TECS bridge */
15     /* Omit: synchronization of
16        starting mruby application */
17     /* Load mruby library bytecode */
18     mrb_load_irep_cxt( VAR_mrb,
19                       ATTR_irepLib, VAR_context );
20     /* Load mruby application bytecode and run */
21     mrb_load_irep_cxt( VAR_mrb,
22                       VAR_irepApp, VAR_context );
23     if ( mrb->exc ) {
24         /* Failure to execute */
25         mrb_p( VAR_mrb,
26               mrb_obj_value( VAR_mrb->exc ) );
27         exit( 0 );
28     }
29     /* Omit: synchronization of
30        finishing mruby application */
31     /* Free mruby context */
32     mrbc_context_free( VAR_mrb, VAR_context );
33     /* Free interpreter instance */
34     mrb_close( VAR_mrb );
35 }

```

Figure 13: Main code for RiteVM with Bluetooth loader for mruby bytecode

programming applied to *delay()* is difficult to use in the case of fair scheduling.

As an approach for multitask processing, the proposed framework provides the RiteVM scheduler which is a fair scheduler and runs multiple tasks equally. The RiteVM scheduler is utilized only when application tasks have the same priority. mruby applications can run in concurrent without developers calling the OS's function. The application programs can also utilize the existing programs since the structures of the programs are not changed.

The RiteVM scheduler is a fair scheduler and implemented as a TECS component in the proposed framework. Therefore, when developers create software in priority-based scheduling, the RiteVM scheduler can be easily removed.

3.2.1 Design of RiteVM Scheduler

A RiteVM scheduler is a periodic handler and *rotateReadyQueue*, a service call of μ ITRON to switch tasks with the same priority, is implemented as the main process of the handler. In other words, the RiteVM scheduler calls *rotateReadyQueue* cyclically. The design of the RiteVM scheduler is shown in Figure 14. *rotateReadyQueue* is described as follows.

The case is assumed that there are two tasks with the same priority, and both tasks are in an infinite loop. In the current system, when one task is executed first, another task would not be executed. That is because the task with first execution runs in the loop.

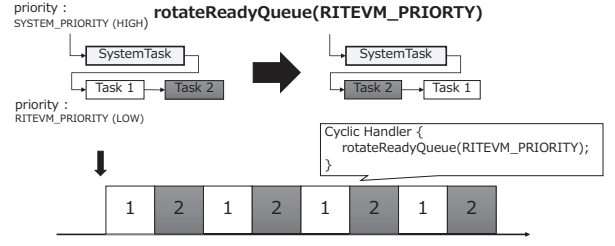


Figure 14: The design of RiteVM scheduler

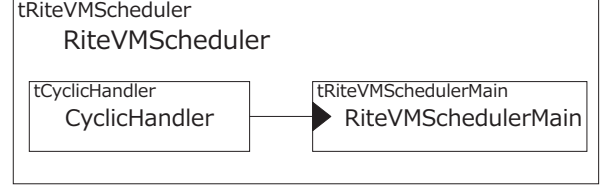


Figure 15: Component Diagram of RiteVM Scheduler

When *rotateReadyQueue* is called, tasks with the same priority are switched as shown in Figure 14. The argument of *rotateReadyQueue* is the priority.

rotateReadyQueue can be performed if the number of tasks is more than two. For example, three tasks are in the order: task1, task2, and, task3. In this case, the order is rotated, task2, task3, and, task1, when the *rotateReadyQueue* is called.

3.2.2 Component of RiteVM Scheduler

Figure 15 shows the component diagram of RiteVM scheduler. The component of RiteVM scheduler consist of *CyclicHandler* and *RiteVMSchedulerMain*. *CyclicHandler cell* configures the periodic handler based on μ ITRON. [24] describes cyclic handlers based on μ ITRON in detail. *CyclicHandler cell* has attributes of the *cell*. *RiteVMSchedulerMain cell* is a component to perform the processing body of a periodic handler. *rotateReadyQueue* is implemented as the body. Figure 16 shows *tRiteVMScheduler celltype*, which is the *composite cell* consisting of two *cells*. The *call* port of *RiteVMSchedulerMain* is connected with the *entry* port of the *Kernel cell* (*tKernel.eiKernel*) to call functions of the kernel. The attribute is used as an arguments of *rotateReadyQueue*.

Figure 17 shows the build description of RiteVM scheduler. *RiteVMScheduler cell* has attributes to configure the scheduler such as attribute, cyclicTime, cyclicPhase, and priority. In this case, RiteVM scheduler is executed when it is generated because the attribute is *TA_STA* that represents the periodic handler is in an operational state after the creation. The scheduler executes every one msec. *RITEVM_PRIORITY* defines the priority of mruby tasks. In the main of *CyclicMain*, *rotateReadyQueue* is implemented and the priority is passed as the argument.

3.3 Synchronization of Multiple RiteVM Tasks

In the proposed framework, RiteVMs read mruby bytecodes, and then execute the applications. Eventflag, one of synchronous processing, is applied to synchronize the starting of multiple mruby applications. Each task sets the flag pattern such as 0x01 (01) and 0x02 (10), and then waits the

```

1  /* tRiteVMScheduler.cdl */
2  celltype tCyclicHandler {
3      [inline] entry sCyclic eCyclic;
4      call siHandlerBody ciBody;
5      attr {
6          [omit] ATR attribute = C_EXP("TA_NULL");
7          [omit] RELTIM cyclicTime;
8          [omit] RELTIM cyclicPhase;
9      };
10 };
11 celltype tRiteVMSchedulerMain {
12     require tKernel.eiKernel;
13     entry siHandlerBody eiBody;
14     attr {
15         PRI priority;
16     };
17 };
18
19 composite tRiteVMScheduler {
20     attr {
21         ATR attribute = C_EXP("TA_NULL");
22         RELTIM cyclicTime = 1;
23         RELTIM cyclicPhase = 1;
24         PRI priority;
25     };
26     cell tRiteVMSchedulerMain RiteVMSchedulerMain {
27         priority = composite.priority;
28     };
29     cell tCyclicHandler CyclicHandler {
30         ciBody = RiteVMSchedulerMain.eiBody;
31         attribute = composite.attribute;
32         cyclicTime = composite.cyclicTime;
33         cyclicPhase = composite.cyclicPhase;
34     };
35 };

```

Figure 16: Celltype Description of RiteVM Scheduler

flag pattern, 0x3 (11), with AND. This process can also apply to the case of the more tasks. For example, in the case of the four RiteVM tasks, each task sets the flag pattern such as 0x01 (0001), 0x02 (0010), 0x04 (0100) and 0x08 (1000), and then waits 0x0f (1111) with AND as shown in Figure 18 (A).

In addition, the end of mruby applications is synchronized to accept the continuous loading. The ending synchronization prevents a RiteVM whose application immediately finishes from waiting the next loading. All of mruby applications finish at the same time, and all RiteVMs wait for receiving the next mruby application bytecodes.

3.4 Utilization of Component-Based Development

This section describes the design applying component-based development. In the proposed framework, RiteVMs, the RiteVM scheduler, and Eventflag are implemented as components. Therefore, developers can easily add or remove these components and also reuse them. For example, if RiteVM scheduler is not necessary for software, developers should only comment out the CDL file such as *//import(<tRiteVMScheduler.cdl>);*. This advantage of CBD saves developers the labor of rewriting a kernel configuration file.

In addition, the code size can decrease by developing with component-based. In the proposed framework, the advantage is applied in the Eventflag component. The set pattern and wait pattern are defined as attributes of the compo-

```

1  cell tRiteVMScheduler RiteVMScheduler {
2      attribute = C_EXP("TA_STA");
3      cyclicTime = 1;
4      cyclicPhase = 1;
5      priority =
6          C_EXP("RITEVM_PRIORITY");
7  };

```

Figure 17: Build Description of RiteVM Scheduler

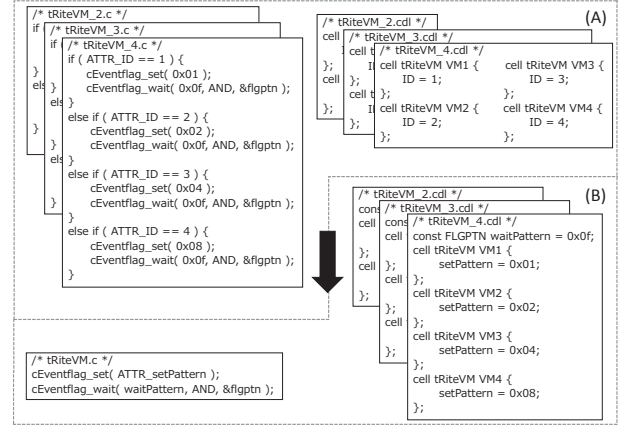


Figure 18: The design for Eventflag using TECS (only differences)

nent as shown in Figure 18 (B). This design such as *cEventflag_set(ATTR_setPattern)* enables the program without "if" statements and reuses the identical C file. Developers do not need to modify the C file because the CDL files are prepared in accordance with the number of RiteVMs. In addition, the components of Eventflag are built with *[optional]* in TECS. *[optional]* means that the codes are run only when the call port is connected. The C file does not be rewritten even if developers do not use Eventflag.

4. EXPERIMENTAL EVALUATION

This section mentions experimental results and their consideration. To analyze the advantages of the proposed framework, the evaluations are performed as follows.

- Size and time of transferred mruby bytecodes by Bluetooth loader
- Execution time with singletasking, co-routine, and proposed multitasking
- Overhead for periodic time
- Synchronization of multiple RiteVM tasks
- Code size utilizing component-based development

These evaluations are performed in order to indicate that a Bluetooth loader improves the software development efficiency, and that the proposed multitask processing effectively executes compared with singletasking or co-routine, and also the overhead of the cyclic period. This paper demonstrates the proposed system on a LEGO MINDSTORMS

Table 1: Comparison of the size and load process time between an mruby application including mruby libraries and not

	App&Lib	App	App&Lib/App
Bytecode Size	14,044 bytes	199 bytes	$\times 70.6$
Loading Time	305.081 msec	7.774 msec	$\times 39.2$
Compile Time	8.7 msec	0.3 msec	$\times 29.0$

EV3 [22] (300MHz ARM9-based Sitara AM1808 system-on-a-chip) compiled with gcc 4.9.3 -O2 and mruby version 1.2.0.

4.1 Improving the software development efficiency by Bluetooth loader

The Bluetooth loader for mruby bytecode can save development time. In the proposed framework, developers do not need to rewrite a storage/ROM device because the bytecode should be just transferred. For example, when mruby programs are modified, developers must pull out an SD card from the target device, connect the host PC, compile/link the platform, and insert the SD card in the target device, in the experimental environment such as LEGO MINDSTORMS EV3. In addition, it can save the time to restart a target RTOS.

In the framework, to further improve the software development efficiency, developers transfers only the mruby application bytecode and mruby libraries are incorporated in the platform. The size, load process time, and compilation time for mruby application and mruby application including library is shown in Table 1. The overhead of load processing is 50.933 msec, which it takes to load a zero bytes bytecode. Similarly, the overhead of compilation is 46.9 msec, which it takes to compile a zero bytes program. The mruby application bytecode is smaller and faster than that of including mruby libraries in all terms. The difference becomes larger as the number of RiteVMs increases because it takes the 50 msec overhead per one RiteVM. These advantages lead improvement of the software development efficiency.

4.2 RiteVM Scheduler

The comparison of the application execution time with singletasking, co-routine, and multitasking is shown in Figure 19. The 100,000 times loop program is used as mruby application for evaluation of execution time. In detail, the singletask program loops 100,000 times, and the multitask and co-routine programs loop 50,000 times in each task.

4.2.1 Execution time with singletasking, co-routine, and purposed multitasking

In Figure 19, the periodic time of the periodic handler for multitasking is one msec. This result shows the proposed design is superior to co-routine in terms of execution time. Moreover, developers can utilize the scheduler practically because the overhead of the RiteVM scheduler is about 5 %. The scheduler interrupts and switches tasks, which causes this overhead. It takes about three μ sec on average to switch tasks once.

4.2.2 Overhead for periodic time

Figure 20 shows the execution time of multitasking with the periodic handler. A lower limit of the periodic time is one msec due to the specification of TOPPERS/HRP2, the

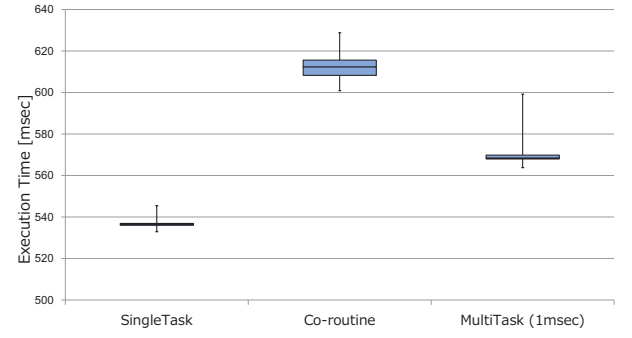


Figure 19: Comparison of the application execution time with singletask, co-routine, and multitask

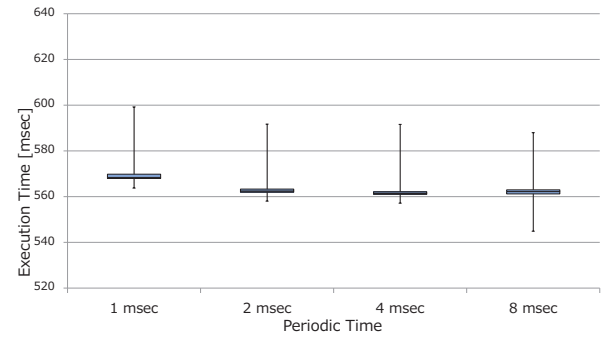


Figure 20: Comparison of the overhead for each cyclic period of RiteVM scheduler

used RTOS. More than eight msec do not be evaluated in this paper because it is thought the larger periodic time influences applications. The execution time decreases as the periodic time become larger, because the number of switching tasks decreases. The execution time of one msec is only about 1 % larger than that of eight msec. The RiteVM scheduler with small periodic time can effectively execute multiple tasks because the periodic time overhead is not large. The smaller periodic time is better in multitasking due to concurrent and/or parallel processing.

4.3 Synchronization of multiple RiteVM tasks

To execute multiple mruby applications, synchronization mechanism of RiteVM tasks is supported in the proposed framework. We measured the time from first RiteVM task executing to last RiteVM task executing, and confirm that the time is within (periodic time) \times (the number of RiteVM tasks - 1). Figure 21 shows the cases where the number of RiteVM tasks is two, three, and four. The periodic time is one msec. The time is within one msec, two msec, and three msec respectively as shown in Figure 21, which shows synchronization of multiple RiteVM tasks is performed.

4.4 Benefits of component-based development

In the proposed framework, RiteVMs, RiteVM scheduler, and Eventflags are implemented as TECS components. Developers can easily add or remove the functionalities by modifying the CDL file. Moreover, component-based development decreases code sizes and improve productivity and

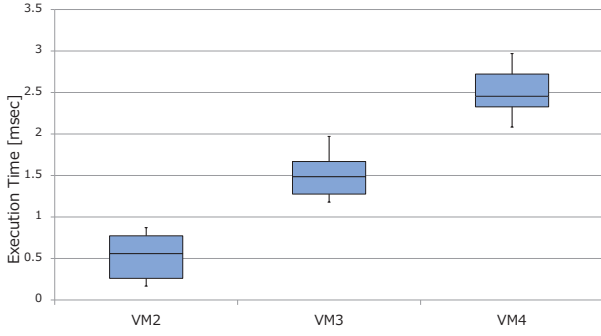


Figure 21: Synchronization of multiple RiteVM tasks

Table 2: Code Lines of C and CDL file for the number of RiteVM

	(A)	(B)	Diff
C (Total)	$8 \times \alpha + 134$	130	$8 \times \alpha + 4$
C (Modification)	$10 \times \alpha - 2$	0	$10 \times \alpha - 2$
CDL	$18 \times \alpha + 25$	$18 \times \alpha + 25$	0

α : the number of RiteVM

maintainability.

To indicate the superior of component-based development, the comparison of code lines between two C and CDL files is shown in Table 2. (A) and (B) mean the source files in the upper and lower of Figure 18, respectively. In terms of C, (B)'s code lines do not increase even if the number of RiteVMs increases, while (A)'s code lines increase as the number of RiteVM increases. (B)'s C file can be utilized without modification, regardless of the number of RiteVMs. Moreover, code lines of two CDL files are equal. The skillful component-based development brings this advantages such as the decrease of code lines and non-modified code, which leads high productivity and high maintainability.

5. RELATED WORK

The open-source run-time systems for scripting languages have been proposed such as follow: python-on-a-chip [7], the Owl system [14], eLua [3], Squirrel [9], mruby [25], [5], and mruby on TECS [11].

python-on-a-chip: python-on-a-chip (p14p) is a Python run-time system that uses a reduced Python VM called PyMite. The VM runs a significant subset of Python language with few resources on a micro-controller. p14p can run multiple stackless green threads.

Owl system: The Owl system is an embedded Python run-time system. The Owl is a complete system for ARM Cortex-M3 micro-controllers. The Owl toolchain produces relocatable memory images, that are directly runnable on the micro-controller, from Python code objects. The interpreter of the Owl system is the same as that of python-on-a-chip.

eLua: eLua offers the full implementation of Lua programming language to the embedded systems. Lua is one of the most popular script languages for embedded systems [4], [19]. Lua supports co-routine, referred to collaborative multitasking. A co-routine in Lua is used as an independently executed thread. A co-routine can just suspend and

resume multiple routines. Thus, a Lua co-routine is not like multitasks in multitask systems.

Squirrel: Squirrel is an object-oriented programming language, designed to be a lightweight scripting language that fits in real-time requirements of applications. Squirrel is inspired by especially Lua. The API is very similar to Lua and the table code is based on the Lua one. Squirrel also supports co-routine as well as Lua.

mruby: mruby, the lightweight implementation of the Ruby language, has been proposed for embedded systems. mruby programs can run on a RiteVM, which is the VM for mruby and reads the mruby bytecode. RiteVM supports only single thread. mruby has supported co-routine, but not supported multitasking for RTOSs.

mruby on TECS: mruby on TECS is a component-based framework for running mruby programs. The programs on mruby on TECS can execute about 100 times faster than the mruby programs. Software can be also developed with component base by mruby on TECS. Although multitasking has been supported in the current mruby on TECS, developers need to be familiar with functions of an RTOS to use multitasking. The co-routine is supported as same as mruby.

Table 3 shows a comparison of the proposed framework and previous work. Note that the proposed framework has demonstrated all features shown in Table 3. In particular, the proposed framework supports the loader, the VM scheduler, and synchronization of application.

6. CONCLUSION

This paper has presented an extended framework of mruby on TECS: Bluetooth loader for mruby bytecode and RiteVM scheduler. The loader provides developers with the software development efficiency without rewriting a storage/ROM device and restarting an RTOS. The proposed framework can be applied to many kinds of embedded systems because the loader can use not only Bluetooth also wired serial connection. The RiteVM scheduler makes multitasking more easily than the current mruby on TECS. In the evaluation, experimental results of the loader and the RiteVM scheduler show their advantages. The loader can improve the software development efficiency on mruby on TECS. The RiteVM scheduler has the effectiveness in terms of execution time and ease of use compared with singletasking and co-routine. In addition, synchronization of multiple RiteVM tasks is implemented.

The proposed framework is developed in component-base by TECS. The facilities such as RiteVMs, the RiteVM scheduler, and Eventflag are implemented as components. Therefore, developers can easily add or remove the functionalities as necessary, and also reuse them. Developers can choose fair scheduling or fixed-priority scheduling since the RiteVM scheduler can be easily added and removed. When software in priority-based scheduling is developed, developers only have to remove the RiteVM scheduler. Component-based development can increase productivity. Our prototype system and application programs used in the performance evaluation are all open-source, and may be downloaded from our website [6].

In the future, the CDL files for RiteVM and mruby-TECS bridge are generated automatically using a plugin, and developers can send a bytecode with ZMODEM protocol on the command line. Moreover, we will support mruby libraries to mrbgems, which is a distribution packaging sys-

Table 3: Comparison of the proposed and previous work

	Bluetooth Loader	Call C Function	Legacy Code of Embedded System	VM Management	VM Scheduler	Synchronization of Applications	Co-routine
python-on-a-chip [7]							✓
Owl system [14]		✓	Partially				✓
eLua [3]		✓	Partially				✓
Squirrel [9]		✓					✓
mruby [25]		✓					✓
mruby on TECS [11]		✓	✓	✓			✓
Proposed framework	✓	✓	✓	✓	✓	✓	✓

tem for mruby.

7. ACKNOWLEDGEMENT

This work was supported by JSPS KAKENHI Grant Number 15H05305. We would like to thank Takuya Ishikawa, Hiroshi Mimaki, and Kazuaki Tanaka for supporting this research.

8. REFERENCES

- [1] AUTOSAR. <http://www.autosar.org/>.
- [2] class Fiber. <http://docs.ruby-lang.org/en/2.3.0/Fiber.html>.
- [3] eLua. <http://www.eluaproject.net>.
- [4] Lua. <http://www.lua.org/>.
- [5] mruby. <https://github.com/mruby/mruby>.
- [6] mruby-on-ev3rt+tecs. http://www.toppers.jp/tecs.html#mruby_ev3rt.
- [7] python-on-a-chip. <http://code.google.com/archive/p/python-on-a-chip/>.
- [8] Ruby. <https://www.ruby-lang.org/en/>.
- [9] Squirrel. <http://www.squirrel-lang.org/>.
- [10] TOPPERS Project. <http://www.toppers.jp/en/index.html>.
- [11] T. Azumi, Y. Nagahara, H. Oyama, and N. Nishio. mruby on TECS: Component-Based Framework for Running Script Program. In Proceedings of the 18th IEEE International Symposium on Real-Time Computing (ISORC), pages 252–259, 2015.
- [12] T. Azumi, H. Takada, T. Ukai, and H. Oyama. Wheeled Inverted Pendulum with Embedded Component System: A Case Study. In Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), pages 151–155, 2010.
- [13] T. Azumi, M. Yamamoto, Y. Kominami, N. Takagi, H. Oyama, and H. Takada. A New Specification of Software Components for Embedded Systems. In Proceedings of the 10th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), pages 46–50, 2007.
- [14] T. W. Barr, R. Smith, and S. Rixner. Design and Implementation of an Embedded Python Run-Time System. In Proceedings of the USENIX Annual Technical Conference (USENIX ATC 12), pages 297–308, 2012.
- [15] B. Bonakdarpour and S. S. Kulkarni. Compositional verification of fault-tolerant real-time programs. In Proceedings of the Seventh ACM International Conference on Embedded Software, EMSOFT '09, pages 29–38, 2009.
- [16] I. Crnkovic. Component-based Software Engineering for Embedded Systems. In Proceedings of the 27th International Conference on Software Engineering, pages 712–713, 2005.
- [17] C. Forsberg. The ZMODEM Inter Application File Transfer Protocol. <http://pauillac.inria.fr/~doligez/zmodem/zmodem.txt>, 1988.
- [18] G. Gössler and L. Aştefănoaei. Blaming in component-based real-time systems. In Proceedings of the 14th International Conference on Embedded Software, EMSOFT '14, 2014.
- [19] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The Evolution of Lua. In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, pages 2–1–2–26, 2007.
- [20] T. Ishikawa, T. Azumi, H. Oyama, and H. Takada. HR-TECS: Component technology for embedded systems with memory protection. In Proceedings of the 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), pages 1–8, 2013.
- [21] M. kerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli. The SAVE Approach to Component-based Development of Vehicular Systems. Journal of Systems and Software, 80(5):655–667, 2007.
- [22] Y. Li, T. Ishikawa, Y. Matsubara, and H. Takada. A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support. OSPERT 2014, pages 51–59, 2014.
- [23] A. Ohno, T. Azumi, and N. Nishio. TECS Components Providing Functionalities of OSEK Specifications for ITRON OS. Journal of Information Processing, 22(4):584–594, 2014.
- [24] H. Takada and K. Sakamura. μ ITRON for Small-Scale Embedded Systems. IEEE Micro, 15(6):46–54, 1995.
- [25] K. Tanaka, A. D. Nagumanthri, and Y. Matsumoto. mruby – Rapid Software Development for Embedded Systems. In Proceedings of the 15th International Conference on Computational Science and Its Applications (ICCSA), pages 27–32, 2015.
- [26] TOPPERS. TOPPERS/ASP kernel. <https://www.toppers.jp/en/asp-kernel.html>.

- [27] TOPPERS. TOPPERS/HRP2 kernel.
<http://www.toppers.jp/en/hrp2-kernel.html>.