Bluetooth Loader for mruby Bytecode in Multiple Virtual Machine Environment

ABSTRACT

Recently, the productivity of embedded systems has become problematic due to their increasing complexity and scale. To improve the productivity, the mruby on TOPPERS embedded component system (TECS) framework, which employs mruby (i.e., lightweight Ruby) and supports component-based development, has been proposed. In the current mruby on TECS framework, mruby programs must be compiled and linked every time they are modified, because mruby bytecode are incorporated in the platform. Moreover, while the framework supports multiple virtual machines (VMs), developers must be familiar with the functions of real-time operating systems to effectively execute multiple mruby programs concurrently or in parallel. To improve development efficiency, we propose a Bluetooth loader for mruby bytecode as an extension of mruby on TECS. The loader executes two mruby bytecodes, i.e., the mruby application bytecode and mruby library bytecode. The mruby application bytecode is modified frequently and sent from a host to a target device by the developers, while the mruby library bytecode is modified infrequently and preserved beforehand on a storage/ROM device during the first compilation. In addition, multiple mruby programs can run cooperatively in the proposed framework. Compared to mruby on TECS, the RiteVM scheduler simplifies multitasking. Synchronization of initializing multiple tasks is also implemented using an Eventflag. Experimental results demonstrate the advantages of the proposed framework.

Keywords

embedded software; scripting language; component-based development

1. INTRODUCTION

Currently, embedded systems, e.g., Internet of Things applications, must demonstrate high quality and high performance. This requirement has led to an increase in their complexity and scale; moreover, these systems need to have low production costs and short development cycles.

Complex and large-scale software systems can be developed efficiently by using component-based techniques [17], [16]. Component-Based Development (CBD) is a design technique that can be applied to reusable software development. Verification of component-based systems has been extensively researched [19], [15]. Individual component diagrams enable the visualization of an entire system. In addition, component-based systems are flexible with regard to extensibility and specification changes. The TOPPERS embedded component system (TECS) [13], AUTOSAR [1], and SaveCCM [22] are typical CBD tools for embedded systems.

In addition, scripting languages, such as Ruby, JavaScript, Perl, Python, and Lua, offer efficient approaches to software development. Currently, most software are programmed in C language. However, development in C language results in large code size, incurs high costs, and requires significant development time. In contrast, the use of scripting languages improves the efficiency of software engineering and can shorten the development period because it is relatively easy to reuse scripts.

For embedded systems, real-time properties, such as estimation of worst-case execution time, are very important. Although scripting languages are easy to use and read, their execution requires more time than that required by the codes written in C. Therefore, applying scripting languages to embedded systems is difficult.

To address the above limitation, "mruby on TECS," a component-based framework for running script programs, has been proposed [11]. This framework integrates two technologies, i.e., mruby, which is a lightweight implementation of Ruby for embedded systems [26], [5], and TECS, which is a component-based framework for embedded systems [13], [10].

Even though execution times of mruby on TECS are 100 times faster than those of mruby, it is not particularly efficient, at present, and imposes a heavy burden on developers. Moreover, mruby on TECS only supports a storage/ROM device for loading mruby programs. Consequently, if mruby programs are modified, a secure digital (SD) card must be inserted and removed repeatedly or ROM must be rewritten; moreover, developers need to restart real-time operating systems (RTOSs) on the target device. In addition, although mruby on TECS can support multiple virtual machines (multi-VMs), executing multiple tasks requires the developers to call the OS function.

This paper proposes an extended framework of mruby on

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

DOI: 10.475/123_4

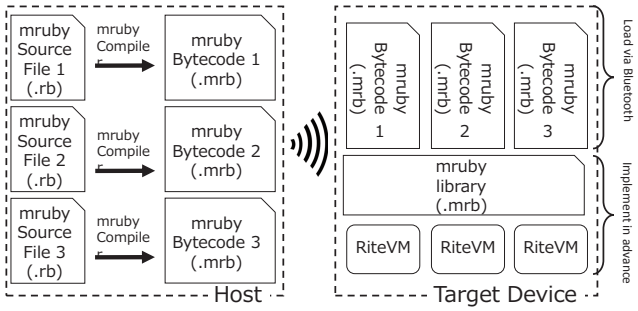

Figure 1: System model of the proposed framework

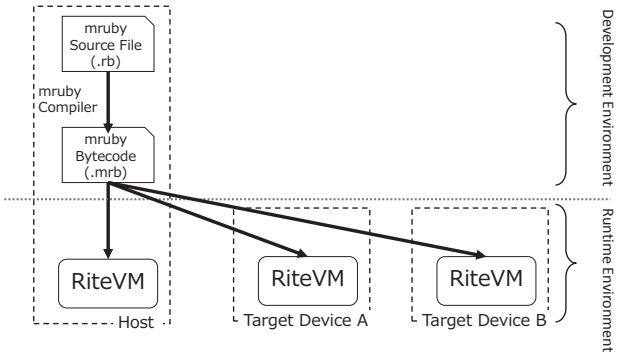

Figure 2: mruby/RiteVM mechanism

TECS that comprises a Bluetooth loader for mruby bytecode and a RiteVM scheduler for fairly executing mruby programs. To improve development efficiency, in the proposed framework, developers need to implement the platform on a storage device only once at the beginning and can transfer mruby application programs from a host to a target device using the Bluetooth loader. Note that RiteVM is the Ruby VM specifically designed for embedded systems. The RiteVM scheduler manages the execution of multiple RiteVMs and allows developers to program multitasking more easily than the current version of mruby on TECS.

Contributions: The proposed framework provides the following contributions:

1. **Improved software development efficiency.** Developers do not need to rewrite a storage/ROM device and restart an RTOS. The Bluetooth loader supports continuous loading, which reduces Bluetooth set-up time (i.e., pairing).
2. **Execution of multiple mruby programs concurrently or in parallel.** Developers can implement multiple tasks without RTOS knowledge because the RiteVM scheduler switches tasks cyclically.
3. **Synchronized execution of multiple RiteVM tasks.** The proposed framework synchronizes multiple RiteVM tasks (i.e., mruby applications).
4. **Benefits of CBD:** The paper focuses on the benefits of CBD and provides specific examples.

Organization: The reminder of this 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. Section 4 evaluates the proposed framework. Related work is discussed in Section 5. Conclusions and suggestions for future work are presented in Section 6.

2. BACKGROUND

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

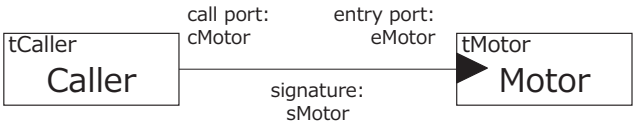

Figure 3: Component Diagram

2.1 mruby

mruby is a light-weight implementation of the Ruby programming language complying to part of the ISO standard. Ruby is an object-oriented scripting language [8] with classes and methods, exceptions, and garbage collection functions. It is easy to use and read due to its simple grammar and Ruby requires fewer lines of code than C. Ruby improves the productivity of software development due to its simple grammar and object-oriented functions.

mruby, which retains the usability and readability of Ruby, requires fewer resources, and thus, is suitable for embedded systems. In addition, mruby includes a VM mechanism, and thus, mruby programs can run on any operating system as long as a VM is implemented. The mruby/RiteVM mechanism is shown in Figure 2. The mruby compiler translates an mruby code into a bytecode, which can be interpreted by a RiteVM; thus, mruby programs can be executed on any target device with a RiteVM.

2.2 TECS

TECS is a component system suitable for embedded systems. TECS can increase productivity and reduce development costs due to improved reusability of software components. TECS also provides component diagrams, which help developers visualize the overall structure of a system.

In TECS, component deployment and composition are performed statically. Consequently, connecting components does not incur significant overhead and memory requirements can be reduced. TECS can be implemented in C, and demonstrates various feature such as source level portability and fine-grained components.

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,

```
signature sMotor {
    int32_t getCounts( void );
    ER resetCounts( void );
    ER setPower( [in]int power );
    ER stop( [in] bool_t brake );
    ER rotate( [in] int degrees, [in] uint32_t speed_abs,
               [in] bool_t blocking );
    void initializePort( [in]int32_t type );
};
```

Figure 4: Signature Description

```
celltype tCaller {
    call sMotor cMotor;
};
celltype tMotor {
    entry sMotor eMotor;
    attr {
        int32_t port;
    };
    var {
        int32_t currentSpeed = 0;
    };
};
```

Figure 5: Celltype Description

```
cell tMotor Motor {
    port = C_EXP("PORT_A");
};
cell tCaller Caller {
    cMotor = Motor.eMotor;
};
```

Figure 6: Build Description

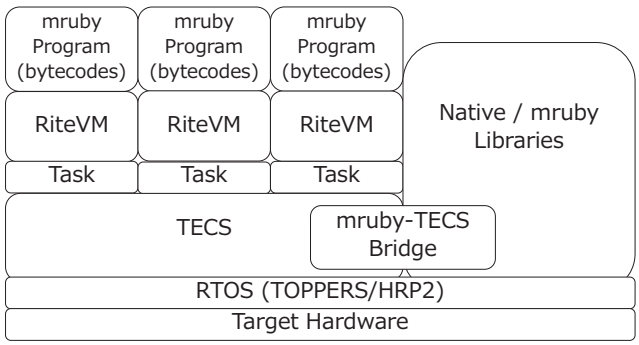

Figure 7: System model of existing mruby on TECS

call ports, attributes and internal variables. An *entry* port is an interface that provides functions to other *cell*s, and a *call* port is an interface that enables the use of other *cell*'s functions. A *cell* has one or more *entry* ports and *call* ports. *Cell* functions are implemented in C.

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

2.2.2 Component Description

In TECS, components are described by *signature*, *celltype*, and build written in component description language (CDL). These components are described as follows.

Signature Description
The *signature* defines a *cell* interface. The *signature* name follows the keyword *signature* and takes the prefix "s" e.g., sMotor (Figure 4). In TECS, to clarify the function of an interface, specifiers such as [in] and [out] are used, which represent input and output, respectively.

Celltype Description
The *celltype* defines *entry* ports, *call* ports, attributes, and variables. A *celltype* name with the prefix "t" follows the keyword *celltype*, e.g., tCaller (Figure 5). To define *entry* ports, a *signature*, e.g., sMotor, and an *entry* port name, e.g., eMotor, follow the keyword *entry*. *Call* ports are defined similarly. Attributes and variables follow the keywords *attr* and *var*, respectively.

Build Description
The build description is used to instantiate and connect *cell*s. Figure 6 shows an example of a build description. A *celltype* name and *cell* name, e.g., tMotor and Motor, respectively, follow the keyword *cell*. To compose *cell*s, a *call* port, *cell*'s name, and an *entry* port are described in that order. In Figure 6, *entry* port eMotor in *cell* Motor is connected to *call* port cMotor in *cell* Caller. *C_EXP* calls macros defined in C files.

2.3 mruby on TECS

2.3.1 System Model

The present mruby on TECS system model is shown in Figure 7. Each mruby program, which is a 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 provides native libraries for mruby and can call a native program (e.g., C legacy code) from an mruby program. The mruby-TECS bridge also provides TECS components for receiving the invocation from an mruby program.

In this paper, TOPPERS/HRP2 [28], [21] is the target RTOS and is based on μITRON [25] 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 [24] and TOPPERS/ASP [12], [27].

2.3.2 mruby-TECS Bridge

There is a significant difference between the execution times of mruby and C language codes. According to [11], mruby programs are several hundred times slower than C programs and the execution of an mruby bytecode on a RiteVM is not as efficient as that of C code. Thus, it is

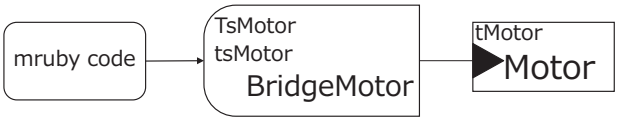

Figure 8: mruby-TECS bridge

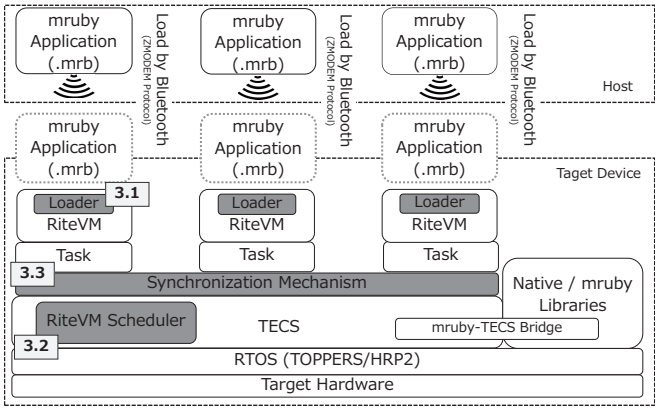

Figure 9: Detailed system model of the proposed framework

difficult to use mruby exclusively.

Using Ruby on embedded devices improves productivity and maintainability because it is easy to use and read. However, some C language codes are required to manipulate actuators and sensors and ensure that critical sections of the code run quickly.

Figure 8 illustrates an mruby-TECS bridge used to control 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 a *celltype*, which is called from the mruby code, and an mruby class, which corresponds to a developer-specified TECS component to invoke a C function from the mruby program.

The generated mruby-TECS bridge supports registration of classes and methods for mruby. 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, the 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 device is received by the loader in the RiteVM. The RiteVM reads the transferred bytecode and executes it with libraries. The mruby applications run simultaneously due to synchronized processing. The RiteVM scheduler switches RiteVM tasks because multiple tasks can run concurrently. The following subsection explains these functionalities.

Figure 10: Component diagram of Bluetooth loader for mruby bytecode

3.1 Bluetooth Loader for mruby Bytecode

This section describes the proposed additional functionality of mruby on TECS, i.e., the Bluetooth loader, for mruby bytecode.[1] In the current system, the platform including mruby bytecodes is saved on a storage/ROM device. Developers must rewrite the storage/ROM device every time the programs are modified. In addition, the RTOS on the target device needs to be restarted. The repetition hinders development efficiency. The Bluetooth loader for mruby bytecode decreases developer burden because developers only have to connect the storage/ROM device and start the RTOS once.

mruby programs consist of an mruby application and mruby libraries. An mruby application is the main program code, mruby libraries define the functions for the application, such as Ruby classes. The mruby bytecodes including both an mruby application and mruby libraries can be transferred and executed on the target device. However, this is also wasteful in terms of bytecode size and the time required to transfer the bytecodes, because the libraries are not modified frequently. With the proposed framework, only mruby applications are transferred, and the mruby libraries are preserved on the storage/ROM device beforehand. As a result, RiteVMs can share mruby libraries. In addition, a RiteVM can use its own library, which other RiteVMs should not use.

In the proposed framework, a platform that includes RiteVMs and an mruby library is first compiled and copied to the storage/ROM device. On the host, the mruby application programs (.rb files) are edited and compiled to bytecodes (.mrb files) by an mruby compiler. The generated bytecodes are transferred from the host to the target device via Bluetooth. This saves time since Bluetooth pairing can be avoided because the loader can load the bytecode continuously.

3.1.1 RiteVM Component with Bluetooth Loader for mruby Bytecode

The proposed framework provides a RiteVM with a Bluetooth loader for mruby bytecode as a TECS component. This component is an extension of the RiteVM component described in [11]. It receives bytecodes via Bluetooth and manages the RiteVM configuration, such as generating mruby library bytecodes automatically. This generated bytecode is prepared beforehand on the storage/ROM device and differs from the bytecode transferred with Bluetooth.

Figure 10 shows a component diagram of MrubyTask1 and RiteVMBluetooth1 *cell*s. The MrubyTask1 *cell* is a componentized task of the RTOS (TOPPERS/HRP2 [28], [21]). The RiteVMBluetooth1 *cell* is the RiteVM component with the Bluetooth loader for mruby bytecode. Bytecode on the

[1]The Bluetooth loader is intended to improve development efficiency; therefore, software developers should use it during the development phase. Note that the complete software should be compiled and linked on the storage/ROM device beforehand.

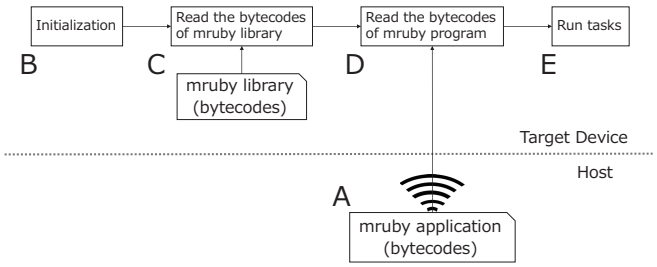

Figure 11: Process flow of Bluetooth loader for mruby bytecode

```
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         mrb_state *mrb;
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

host is transferred and received. In this framework, ZMODEM [18] is used as the binary transfer protocol.

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

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

Second, *mrb_state* and *mrbc_context* pointers, and mruby-TECS bridges are initialized (Figure 11(B); lines 9-14 in Figure 13). *VAR_mrb* and *VAR_context* show the variables of the *cell*. *mrb_state* is a set of states and global variables used in mruby. The synchronization of multiple tasks is performed in this processing phase. The RiteVM that finishes execution at this point waits for the another RiteVM to finish loading and initialization.

Third, the RiteVM reads the bytecode of mruby libraries (Figure 11(C); lines 17-19 in Figure 13). The mruby libraries are a set of Ruby classes, such as the motor and sensor classes. For example, the motor class defines methods to rotate and stop a motor. The tRiteVMBluetooth *cell* has the attributes shown in Figure 12. *ATTR* indicates

```
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                 initializing 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                 terminating 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

an attribute which is a fixed value that cannot be rewritten, unlike *VAR*. The *mrubyLib* indicates the program files of the mruby libraries, and is an attribute because mruby libraries are not modified in the proposed development process. Here, *[omit]* is only used for the TECS generator; thus, the attribute *mrubyLib* does not consume memory. *irepLib* is the pointer of the array in which the bytecode of mruby libraries is stored. To summarize, the bytecode of mruby libraries is stored as an attribute of the component during the first compilation.

Fourth, the RiteVM reads the bytecode of the mruby application transferred via Bluetooth (Figure 11(D); lines 20-22 in Figure 13). The mruby application bytecode is stored in an array of *irepApp*, which differs from the array that holds the mruby library bytecode. Note that two bytecodes are read separately in the RiteVM.

Finally, the mruby task runs (Figure 11(E); lines 20-22 in Figure 13). When an mruby application is modified, only the bytecode of the modified application should be transferred; the mruby libraries do not need to be touched because they typically do not change.

The process shown in Figure 13 (lines 23-28) is carried out when an exception occurs. When all mruby applications are completed, mrb_state and mrbc_context are freed (lines of 31-34 in Figure 13). The proposed framework supports continuous loading; thus, this process loops. After the variables are freed, the RiteVM waits for the next mruby application

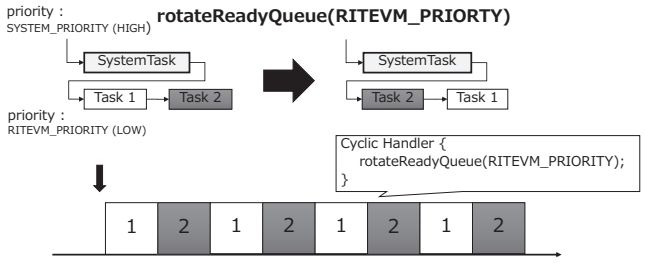

Figure 14: RiteVM scheduler design

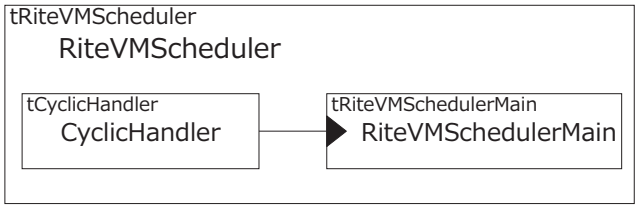

Figure 15: Component diagram of RiteVM scheduler

bytecode.

3.2 RiteVM Scheduler

This section describes the implementation of the RiteVM scheduler in the proposed framework. mruby on TECS supports multitasking; however, multitask processing in mruby on TECS requires the developers to have knowledge about the RTOS (TOPPERS/HRP2).

One approach for multitasking is a co-routine, which is a cooperative thread scheduled by developers with functions such as *resume* and *yield* (the Ruby co-routine is defined in the Fiber class [2]). A co-routine is non-preemptive multitasking, which does not receive OS support because developers must switch tasks manually. A co-routine cannot take advantage of multicore processing.

As another method, *delay()*, which is a service call of μITRON, can be used for multitasking. This service call delays the execution of its own task for the time of the argument. *delay()* is needed when scheduling fixed-priority tasks. However, the programming applied to *delay()* is difficult to use with fair scheduling.

For multitask processing, the proposed framework provides the RiteVM scheduler, which is a fair scheduler that runs multiple tasks equally. Note that the RiteVM scheduler is utilized only when application tasks have equal priority. mruby applications can run concurrently without calling an OS function. The application programs can also utilize existing programs because their structures do not change.

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

3.2.1 RiteVM Scheduler Design

A RiteVM scheduler is a periodic handler, and *rotateReadyQueue*, which is a service call of μITRON to switch tasks with equal 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.

Here, assumed that two tasks with equal priority are in an infinite loop. In the current system, when one task is executed first, the other task would not be executed because the first task runs in the loop.

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

In addition, *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 to task2, task3, and task1 when *rotateReadyQueue* is called.

```
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

3.2.2 Component of RiteVM Scheduler

Figure 15 shows a component diagram of the RiteVM scheduler. The RiteVM scheduler consists of CyclicHandler and RiteVMSchedulerMain. The CyclicHandler *cell* configures the periodic handler based on μITRON. Cyclic handlers based on μITRON are described in the literature [25]. The CyclicHandler *cell* has the attributes of the *cell*. The RiteVMSchedulerMain *cell* processes the body of a periodic handler. Note that *rotateReadyQueue* is implemented as the body. Figure 16 shows tRiteVMScheduler *celltype*, which is a *composite cell* consisting of two *cell*s. The *call* port of

```
cell tRiteVMScheduler RiteVMScheduler {
    attribute = C_EXP("TA_STA");
    cyclicTime = 1;
    cyclicPhase = 1;
    priority =
        C_EXP("RITEVM_PRIORITY");
};
```

Figure 17: Build description of RiteVM scheduler

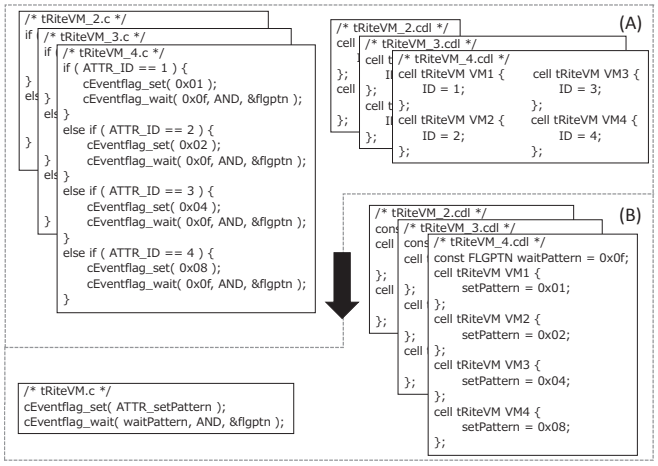

Figure 18: Design of Eventflag using TECS (only differences are shown)

RiteVMSchedulerMain is connected to the *entry* port of the Kernel *cell* (*tKernel.eiKernel*) to call kernel functions. The attribute is used as the *rotateReadyQueue* arguments.

Figure 17 shows the build description of the RiteVM scheduler. The RiteVMScheduler *cell* has attributes to configure the scheduler such as attribute, cyclicTime, cyclicPhase, and priority. In this case, the RiteVM scheduler is executed when it is generated, because the attribute is *TA_STA*, which indicates that the periodic handler is in an operational state after creation. Note that the scheduler executes every 1 msec. RITEVM_PRIORITY defines the priority of mruby tasks. In the function of RiteVMSchedulerMain, *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 applications. Eventflag is applied to synchronize the initiation of multiple mruby applications. Each task sets a flag pattern, such as 0x01 (01) and 0x02 (10), and then waits for the flag pattern 0x3 (11) with AND. This process can also be applied to more tasks. For example, for four RiteVM tasks, each task sets a 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 termination of mruby applications is synchronized to accept continuous loading. This termination synchronization prevents a RiteVM whose application finishes immediately from waiting for the next loading. Thus, all mruby applications finish at the same time, and all RiteVMs wait to receive the next mruby application bytecodes.

Table 1: Comparison of size and load process time between an mruby application with and without mruby libraries

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

3.4 Utilization of Component-Based Development

In the proposed framework, RiteVMs, the RiteVM scheduler, and Eventflags are implemented as components. Therefore, developers can add, remove, or reuse these components easily. For example, if the RiteVM scheduler is not necessary for the software, developers should comment out only the CDL file, e.g., *//import(<tRiteVMScheduler.cdl>);*. CBD eliminates the need for developers to rewrite kernel configuration files.

In addition, the code size can be reduced by using CBD. In the proposed framework, this advantage is applied in the Eventflag component. The set pattern and wait pattern are defined as attributes of the component as shown in Figure 18 (B). This design, e.g., *cEventflag_set(ATTR_setPattern)*, enables the program without "if" statements and reuses an identical C file. Developers do not need to modify the C file because the CDL files are prepared according to the number of RiteVMs. In addition, the Eventflag components are built with *[optional]* in TECS. Here, *[optional]* means that the code is run only when the call port is connected. Note that the C file does not need to be rewritten even if the Eventflag is not used.

4. EXPERIMENTAL EVALUATION

This section discusses experimental results. To analyze the advantages of the proposed framework, we evaluated the following.

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

These evaluations were performed to demonstrate that a Bluetooth loader can improve the efficiency of software development, that the proposed multitask processing executes effectively compared to singletasking or co-routine, and that the initiation of mruby applications are synchronized. In addition, we focused on benefits of CBD. We implemented the proposed system on a LEGO MINDSTORMS EV3 [23] (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 Software Development Efficiency by Bluetooth Loader

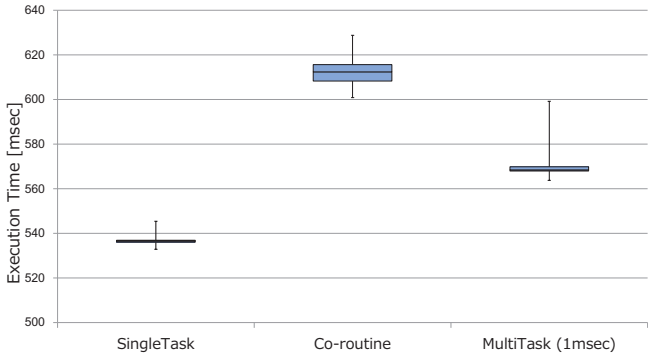

Figure 19: Comparison of application execution time with singletask, co-routine, and multitask

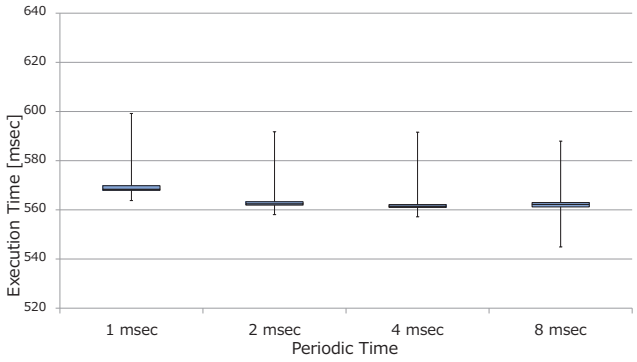

Figure 20: Comparison of overhead for each cyclic period of RiteVM scheduler

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

In the proposed framework, to further improve software development efficiency, developers transfer only the mruby application bytecode; mruby libraries are incorporated in the platform. The size, load process time, and compilation time for an mruby application with and without mruby library are shown in Table 1. The overhead of load processing to load a zero byte bytecode is 50.933 msec. Similarly, compilation overhead to compile a zero byte program is 46.9 msec. The mruby application bytecode is smaller and faster than including the mruby libraries for all terms. The difference increases as the number of RiteVMs increases because 50 msec of overhead is incurred per RiteVM. These advantages improve the efficiency of software development.

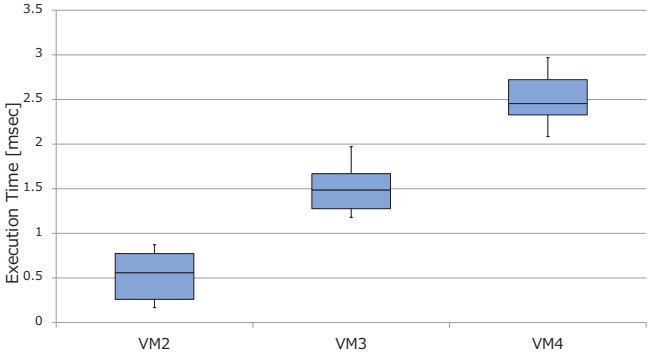

Figure 21: Synchronization of multiple RiteVM tasks

Table 2: C and CDL file code for the number of RiteVMs

	(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

4.2 RiteVM Scheduler

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

4.2.1 *Execution Time with Singletasking, Co-routine, and Proposed Multitasking*

In Figure 19, the periodic time of the periodic handler for multitasking is 1 msec. This shows that the proposed design is superior to co-routine in terms of execution time. Moreover, developers can utilize the scheduler practically because the RiteVM scheduler overhead is approximately 5%. The scheduler interrupts and switches tasks, which causes this overhead. It takes approximately 3 μsec on average to switch tasks once.

4.2.2 *Periodic Time Overhead*

Figure 20 shows the execution time of multitasking with the periodic handler. The lower limit of the periodic time is 1 msec due to the specifications of TOPPERS/HRP2, i.e., the RTOS. The execution time decreases as the periodic time increases, because the number of switched tasks decreases. Note that an execution time of 1 msec is approximately 1% greater than that of 8 msec. The RiteVM scheduler with a short periodic time can execute multiple tasks effectively because the periodic time overhead is not large. Note that a smaller periodic time is better for multitasking due to concurrent or parallel processing.

4.3 Synchronization of Multiple RiteVM Tasks

To execute multiple mruby applications, a synchronization mechanism for RiteVM tasks is implemented in the proposed framework. We measured the time from the execution

Table 3: Comparison of proposed and previous methods

	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 [26]		✓					✓
mruby on TECS [11]		✓	✓	✓			✓
Proposed framework	✓	✓	✓	✓	✓	✓	✓

of the first RiteVM task to the execution of the last RiteVM task. It was confirmed that the time was within (periodic time)×(number of RiteVM tasks − 1). Figure 21 shows the results for two, three, and four RiteVM tasks. The periodic time is 1 msec, and the number of RiteVMs is two, three, and four. As shown in Figure 21, the time is within 1 msec, 2 msec, and 3 msec respectively, which indicates successful synchronization of multiple RiteVM tasks.

4.4 Benefits of Component-Based Development

In the proposed framework, RiteVMs, the RiteVM scheduler, and Eventflags are implemented as TECS components. Developers can add or remove the functionalities easily by modifying the CDL file. Moreover, CBD decreases code size and improves productivity and maintainability.

To demonstrate the superiority of CBD, a comparison of the number of lines of codes between two C and CDL files is shown in Table 2. In Table 2, (A) and (B) represent the source files in the upper and lower parts of Figure 18, respectively. For 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 RiteVMs increases. Note that (B)'s C file can be utilized without modification regardless of the number of RiteVMs. Moreover, the number of code lines of two CDL files are equal. Skillful CBD yields advantages such as the decreased number of lines of codes and non-modified codes, which facilitates high productivity and maintainability.

5. RELATED WORK

Open-source runtime systems for scripting languages have been proposed previously such python-on-a-chip [7], the Owl system [14], eLua [3], Squirrel [9], mruby [26], [5], and mruby on TECS [11].

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

Owl system: The Owl system is an embedded Python runtime system. It is a complete system for ARM Cortex-M3 microcontrollers. The Owl toolchain produces relocatable memory images that are directly runnable on the microcontroller from Python code objects. Note that the Owl system interpreter is the same as that of python-on-a-chip.

eLua: eLua (embedded Lua) offers a full implementation of the Lua programming language for embedded systems. Lua is one of the most popular scripting languages for embedded systems [4], [20]. Lua supports a co-routine, which is referred to as cooperative multitasking. A co-routine in Lua is used as an independently executed thread. Note that a co-routine can only 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 as a lightweight scripting language that satisfies the real-time requirements of applications. Squirrel was inspired by Lua. The Squirrel API is very similar to Lua and the table code is based on that of Lua; Squirrel also supports co-routines.

mruby: mruby, a 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. Note that the RiteVM only supports a single thread. In addition, mruby supports co-routines but does not support multitasking for RTOSs.

mruby on TECS: mruby on TECS is a component-based framework for running mruby programs. mruby programs on TECS can be executed approximately 100 times faster than standard mruby programs. In addition, software can be developed using CBD with mruby on TECS. Although multitasking has been supported in the current mruby on TECS, developers must be familiar with the functions of the RTOS to use multitasking. Co-routines are also supported by mruby on TECS.

Table 3 compares the proposed framework to previous methods. The proposed framework implements all of the features shown in the table. In particular, the proposed framework supports the loader, the VM scheduler, and application synchronization.

6. CONCLUSION

This paper presents an extended framework of mruby on TECS, i.e., the proposed framework implements a Bluetooth loader for mruby bytecode and a RiteVM scheduler. The Bluetooth loader improves software development efficiency by eliminating the need to rewrite a storage/ROM device and restart an RTOS. The proposed framework can be applied to various embedded systems because the loader can use both Bluetooth and a wired serial connection. The RiteVM scheduler simplifies multitasking compared to the current mruby on TECS. Our experimental results for the Bluetooth loader and RiteVM scheduler show their advantages. The Bluetooth loader can improve software development efficiency on mruby on TECS, and the RiteVM scheduler is effective in terms of execution time and ease of use compared to singletasking and co-routines. In addition, synchronization of multiple RiteVM tasks is implemented in the

proposed framework.

The proposed framework is developed using CBD. In addition, the RiteVMs, RiteVM scheduler, and Eventflags are implemented as components; therefore, developers can add, remove, or reuse their functionalities easily as required. Moreover, developers can choose fair scheduling or fixed-priority scheduling because the RiteVM scheduler can be added and removed easily. For software developed with priority-based scheduling, developers only have to remove the RiteVM scheduler. Note that our prototype system and the application programs used in the performance evaluation are all open-source and can be downloaded from our website [6].

In the future, CDL files for the RiteVM and mruby-TECS bridge will be generated automatically using a plugin, and developers will be able to transfer bytecodes using the ZMODEM protocol on the command line. Moreover, we will support mruby libraries as mrbgems, which is an mruby distribution packaging system.

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] X. Cai, M. R. Lyu, K.-F. Wong, and R. Ko. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In Processings of Seventh Asia-Pacific Software Engineering Conference (APSEC 2000), pages 372–379, 2000.

[17] I. Crnkovic. Component-based Software Engineering for Embedded Systems. In Proceedings of the 27th International Conference on Software Engineering, pages 712–713, 2005.

[18] C. Forsberg. The ZMODEM Inter Application File Transfer Protocol. http://pauillac.inria.fr/~doligez/zmodem/zmodem.txt, 1988.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] H. Takada and K. Sakamura. μITRON for Small-Scale Embedded Systems. IEEE Micro, 15(6):46–54, 1995.

[26] 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.

[27] TOPPERS. TOPPERS/ASP kernel. https://www.toppers.jp/en/asp-kernel.html.

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