

Lightweight Ruby Framework for Improving Embedded Software Efficiency

Takuro Yamamoto
Graduate School of Engineering Science,
Osaka University

Hiroshi Oyama
OKUMA Corporation

Takuya Azumi
Graduate School of Engineering Science,
Osaka University

Abstract—Recently, the productivity of embedded systems applied to Cyber Physical 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 scripting language (i.e., a lightweight Ruby) and supports component-based development, has been proposed. This paper proposes an extended framework of mruby on TECS that improve development efficiency more than the current version. In the proposed framework, a Bluetooth loader which receives an mruby bytecode, and a RiteVM scheduler which simplifies multitasking are implemented.

Index Terms—scripting language; cyber physical systems; embedded software; component-based development

I. INTRODUCTION

Cyber Physical Systems (CPS) are the important keyword for enriching our lives. Embedded systems are applied to the CPS and expected to 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 [1]. Component-Based Development (CBD) is a design technique that can be applied to reusable software development. Individual component diagrams enable the visualization of an entire system. In addition, component-based systems are flexible with regard to extensibility and specification changes. TECS [2], AUTOSAR [3], and SaveCCM [4] 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 the large source code, 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 of their usability.

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 [5]. This framework integrates two technologies, i.e., mruby, which is a lightweight implementation of Ruby for embedded systems [6], and TECS, which is a component-based framework for embedded systems [2]. Execution times of mruby on TECS are 100 times faster than those of mruby, which are the same as those of C language. Moreover, Ruby is often utilized in network programming, therefore mruby on TECS can be applicable for the software involved network such as CPS applications.

However, it is not particularly efficient, at present, and imposes a heavy burden on developers. mruby on TECS only supports a storage/ROM device for loading mruby programs. Consequently, if mruby programs are modified, an SD card must be inserted and removed repeatedly or ROM must be rewritten; moreover, developers need to restart an RTOS on the target device. In addition, although mruby on TECS can support multi-VM, executing multiple tasks requires the developers to call the OS function.

This paper proposes an extended framework of mruby on TECS that comprises a Bluetooth loader for mruby bytecode and a RiteVM scheduler for fairly executing mruby programs. 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 via Bluetooth. The RiteVM scheduler manages the execution of multiple RiteVMs (i.e., mruby VM) 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.
- 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 (a Bluetooth loader, a scheduler, and Eventflags).

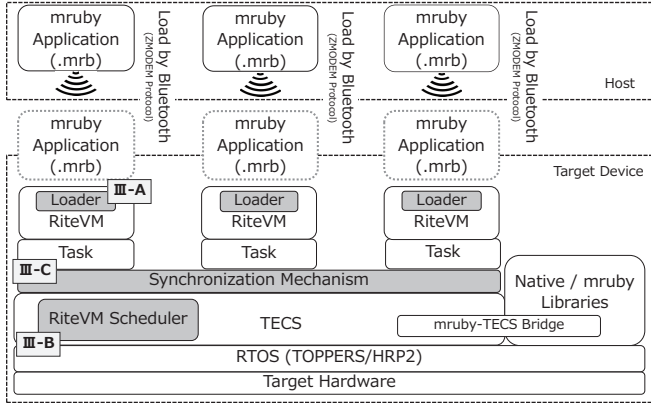


Fig. 1. System model of the proposed framework



Fig. 2. Component diagram of RiteVM with Bluetooth loader

Organization: The reminder of this paper is organized as follows. Section II introduces the system model, and Section III describes the design and implementation of the proposed framework. Section IV evaluates the proposed framework. Related work is discussed in Section V. Conclusions and suggestions for future work are presented in Section VI.

II. SYSTEM MODEL

The proposed framework is an extended framework of mruby on TECS¹, and utilizes mruby and TECS. mruby and TECS are described in Appendixes A and B.

Figure 1 shows the 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 (Section III-A). The mruby applications run simultaneously due to synchronized processing. The RiteVM scheduler switches RiteVM tasks, and multiple tasks can run concurrently (Section III-B). mruby applications are synchronized by the Eventflag (Section III-C). In this paper, TOPPERS/HRP2 [7] is the target RTOS and is based on μ ITRON [8] with memory protection.

III. DESIGN AND IMPLEMENTATION

A. Bluetooth Loader for mruby Bytecode

This section describes the proposed additional functionality, i.e., the Bluetooth loader.² 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

¹mruby on TECS is explained in detail in [5]

²The Bluetooth loader is intended to improve development efficiency; therefore, software developers should use it during the development phase. Note that the bytecode after the development phase should be compiled and linked on the storage/ROM device beforehand.

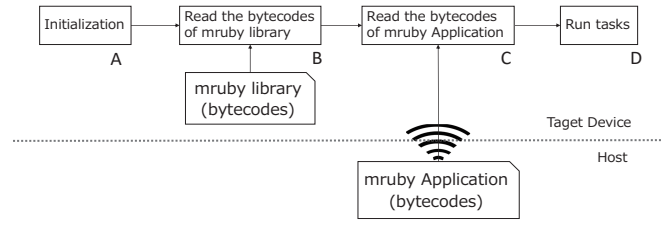


Fig. 3. Process flow of RiteVM with Bluetooth loader

decreases developer burden because developers only have to connect the storage/ROM device and start the RTOS once. An existing Bluetooth-based loader such as TOPPERS/EV3RT [9] has supported a load of object files. The proposed loader newly supports a load for bytecodes of scripting languages.

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.

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.

1) *RiteVM Component with Bluetooth Loader:* The proposed framework provides a RiteVM with a Bluetooth loader as a TECS component. This component is an extended RiteVM component described in [5]. 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 2 shows a component diagram of MrubyTask1 and RiteVMBluetooth1 cells. The MrubyTask1 cell is a componentized task of the RTOS (TOPPERS/HRP2 [7]). The RiteVMBluetooth1 cell is the RiteVM component with the Bluetooth loader. Figure 3 shows the process flow for executing an mruby program on the RiteVM component with the Bluetooth loader, such as RiteVMBluetooth1. In this framework, ZMODEM [10] is used as the binary transfer protocol. The main process of tRiteVMBluetooth is shown in Figure 4.

First, the Bluetooth loader receives the mruby application bytecode from the host (Figure 3(A); line 4 in Figure 4). The bytecode is stored in a component variable, such as `VAR_irepApp`, as shown in Figure 5. This process is exclusively carried out by the semaphore to prevent the other loading from interrupting.

Second, `mrub_state` and `mrbc_context` pointers are initialized (Figure 3(B); lines 7, 9 in Figure 4). `VAR_mrb` and

```

1 void eMrubyBody_main( CELLIDX idx ) {
2   /* Omit: start of exclusive process by semaphore */
3   /* Receive the bytecode via Bluetooth */
4   bluetooth_loader( VAR_irepApp );
5   /* Omit: end of exclusive process by semaphore */
6   /* New interpreter instance */
7   VAR_mrb = mrb_open();
8   /* New mruby context */
9   VAR_context = mrbc_context_new( VAR_mrb );
10  /* Omit: synchronization of initializing mruby application */
11  /* Load mruby library bytecode */
12  mrb_load_irep_cxt( VAR_mrb, ATTR_irepLib,
13                    VAR_context );
14  /* Load mruby application bytecode and run */
15  mrb_load_irep_cxt( VAR_mrb, VAR_irepApp,
16                    VAR_context );
17  /* Omit: synchronization of terminating mruby application */
18  /* Free mruby context */
19  mrbc_context_free( VAR_mrb, VAR_context );
20  /* Free interpreter instance */
21  mrb_close( VAR_mrb );
22 }

```

Fig. 4. Main code for RiteVM with Bluetooth loader

```

1 celltype tRiteVMBluetooth1 {
2   /* An entry port for tTask */
3   entry sTaskBody eMrubyBody;
4   /* A call port for Eventflag components */
5   [optional] call sEventflag cEventflag[];
6   /* A call port for a Semaphore component */
7   [optional] call sSemaphore cSemaphore;
8   attr {
9     [omit]char_t *mrubyLib;
10    char_t *irepLib = C_EXP("&$cell_global$_irep");
11  };
12  var {
13    mrb_state *mrb;
14    mrbc_context *context;
15    [size_is(irepAppSize)] char_t *irepApp;
16  };
17 };

```

Fig. 5. Celltype description for RiteVM with Bluetooth loader

VAR_context show the variables of the *cell*, and which are a set of states and global variables used in mruby. Note that the synchronization of multiple tasks is performed in this processing phase (Section III-C).

Third, the RiteVM reads the bytecode of mruby libraries (Figure 3(C); line 12 in Figure 4). 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 5. *ATTR* indicates 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 (i.e., *irepLib*) during the first compilation.

Fourth, the RiteVM reads the bytecode of the mruby application transferred via Bluetooth (Figure 3(D); line 14 in Figure 4). The mruby application bytecode is stored in an array of *irepApp*. Here, *[size_is()]* is the specifier for an array. Note

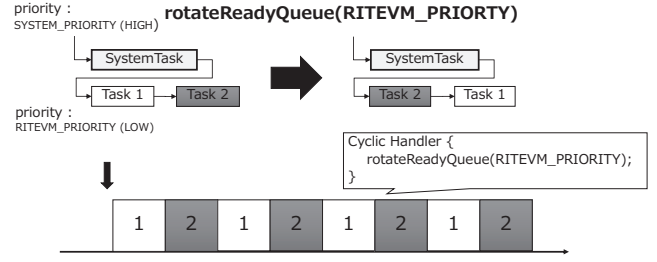


Fig. 6. RiteVM scheduler design

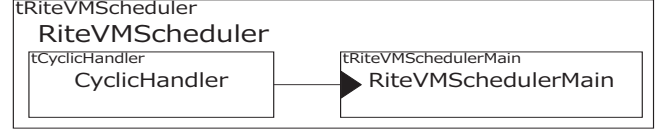


Fig. 7. Component diagram of RiteVM scheduler

that two bytecodes, *irepLib* and *irepApp*, are read separately in the RiteVM.

Finally, the RiteVM runs (Figure 3(E); line 14 in Figure 4). When an mruby application is modified, only the bytecode of the application should be transferred; the mruby libraries do not need to be touched because they typically do not change. The proposed framework supports continuous loading; thus, this process loops. After the variables are freed (lines 17, 19 in Figure 4), the RiteVM waits for the next mruby application bytecode. In addition, the proposed framework supports the reset component which force applications stop and free, thus developers can transfer new bytecodes and run tasks.

B. 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 [11]). A co-routine is non-preemptive multitasking, which does not receive OS support because developers must switch tasks manually.

As another method, *delay()*, which is a service call of μ ITRON, 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 RiteVM scheduler is a fair scheduler that is implemented as a TECS component in the proposed framework. Therefore, when developers create software with fixed-priority based scheduling, the RiteVM scheduler can be removed easily.

```

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

```

Fig. 8. Build description of RiteVM scheduler

1) *RiteVM Scheduler Design*: 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.

The design of the RiteVM scheduler is shown in Figure 6. A RiteVM scheduler is a periodic handler, and *rotateReadyQueue*, which is a service call of μ ITRON to switch tasks with equal priority, is implemented as the main process of the handler. *rotateReadyQueue* switches tasks with equal 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, 2, and 3. In this case, the order is rotated to Task2, 3, and 1 when the function is called.

2) *Component of RiteVM Scheduler*: Figure 7 shows a component diagram of the RiteVM scheduler. The RiteVM scheduler is a *composite cell* which consists of *CyclicHandler* and *RiteVMSchedulerMain*. The *CyclicHandler cell* configures the periodic handler based on μ ITRON. Cyclic handlers based on μ ITRON are described in the literature [8]. 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 8 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.

C. 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 0x03 (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 9(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.

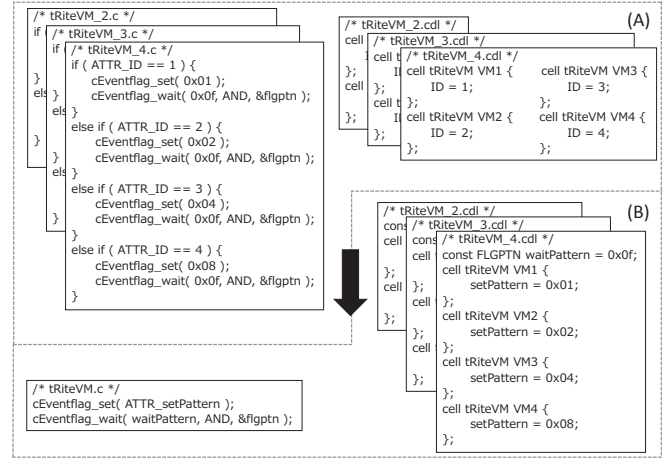


Fig. 9. Design of Eventflag using TECS (only differences are shown)

D. Utilization of Component-Based Development

In the proposed framework, RiteVMs, the 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 9 (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. Thus, the C file does not need to be rewritten even if the Eventflag is not used.

IV. EXPERIMENTAL EVALUATION

We evaluated to demonstrate that a Bluetooth loader can improve the efficiency of software development, that the proposed multitask processing executes effectively compared to existing multitask processing 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 [9] (300MHz ARM9-based Sitara AM1808 system-on-a-chip) compiled with gcc 4.9.3 -O2 and mruby version 1.2.0.

Improving Development Efficiency by Bluetooth Loader. The Bluetooth loader 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

TABLE I
COMPARISON BETWEEN AN APPLICATION WITH AND WITHOUT LIBRARIES

	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	$\times 29.0$

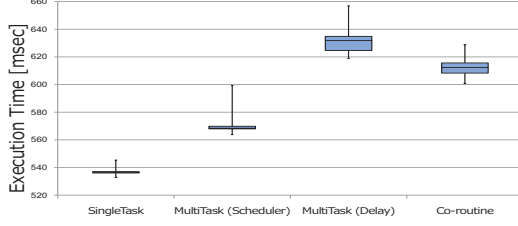


Fig. 10. Comparison of application execution time

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. To confirm the advantage of the proposed design, we evaluated the size, loading time, and compile time between an mruby application with and without mruby library are shown in Table I. Loading Time indicates how long it takes to process a transferred bytecode in the loader, and the overhead of loading a zero byte bytecode is 50.933 msec. Similarly, the 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.

RiteVM Scheduler. A comparison of the application execution time with singletask, proposed, existing multitask, and co-routine is shown in Figure 10. We applied a program with 100,000 loops as an mruby application for the evaluation of execution time so that the overhead comes out. Here, in the multitask and co-routine, we prepared two tasks and the programs looped 50,000 times for each task.

1) *Execution Time of Singletask, Proposed Multitask, Existing Multitask, and Co-routine:* Figure 10 shows that the proposed design is superior to multitask using delay method and co-routine in terms of execution time. The *delay* and *co-routine* methods run on a RiteVM, which causes the slowdown compared to the proposed multitasking. The scheduler runs as the periodic handler of RTOS. 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. Although the overhead is larger as the number of RiteVMs increases, it also spread the difference between multitask and co-routine. Therefore, the scheduler is the more efficient approach for multitasking than the existing multitasking and co-routine. Here, the periodic time of the scheduler is 1 msec.

2) *Periodic Time Overhead:* Figure 11 shows the execution time of multitasking with the periodic handler. The lower limit

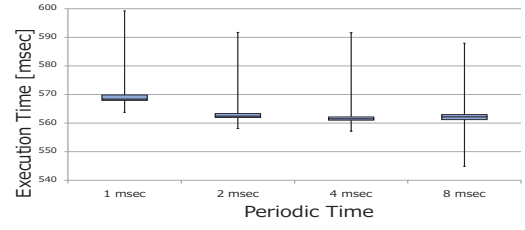


Fig. 11. Comparison of overhead for each cyclic period of RiteVM scheduler

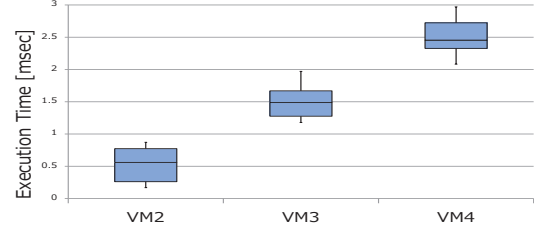


Fig. 12. Synchronization of multiple RiteVM tasks

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.

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 of the first RiteVM task to the execution of the last RiteVM task. It was confirmed that the time was within (periodic time) \times (number of RiteVM tasks - 1). Figure 12 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 12, the time is within 1 msec, 2 msec, and 3 msec respectively, which indicates successful synchronization of multiple RiteVM tasks.

Benefits of Component-Based Development. In the proposed framework, RiteVMs, the RiteVM scheduler, and Event-flags 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 C and CDL codes between two situations is shown in Table II. In Table II, (A) and (B) represent the source files in the upper and lower parts of Figure 9, 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. Moreover, (B)'s C file can be utilized without modification regardless of the number of RiteVMs, while 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.

TABLE II
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

V. RELATED WORK

Open-source runtime systems for scripting languages have been proposed such python-on-a-chip [12], Owl system [13], eLua [14], Squirrel [15], mruby [6], and mruby on TECS [5].

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 [16]. 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. 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 Ruby, has been proposed for embedded systems. mruby programs can run on a RiteVM which 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. Moreover, TECS supports CBD. Although multitasking has been supported in the current mruby on TECS, it is not user-friendly for developers.

VI. 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 more effective than the existing technique and co-routines for multitasking. Moreover, synchronization of multiple RiteVM tasks is implemented in the proposed framework.

The proposed framework is developed using CBD. The RiteVMs, the scheduler, and Eventflags are implemented as components; therefore, developers can add, remove, or reuse them easily as required. 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 is open-source and can be downloaded from our website [17].

In the future, CDL files for the RiteVM and mruby-TECS bridge will be generated automatically using a plugin. Moreover, we will support mruby libraries as mrbgems, which is an mruby distribution packaging system.