Efficient Development Framework with mruby for Embedded Systems

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 has become problematic due to their increasing complexity and scale such by cyber physical systems. To improve the productivity, the mruby on TOPPERS embedded component system (TECS) framework, which employs scripting language (i.e., mruby) 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 a transferred mruby bytecode from a host, and a RiteVM scheduler which simplifies multitasking are implemented. Experimental results demonstrate the advantages of the proposed framework.**

***Index Terms*—embedded software; scripting language; component-based development**

I. Introduction

Currently, embedded systems such as Cyber Physical Systems 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 [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 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 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].

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. 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.

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

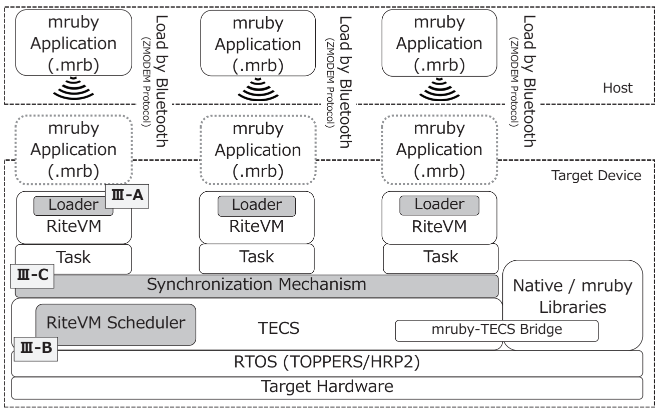


Fig. 1. System model of the proposed framework

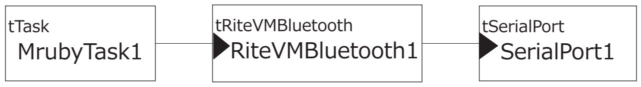
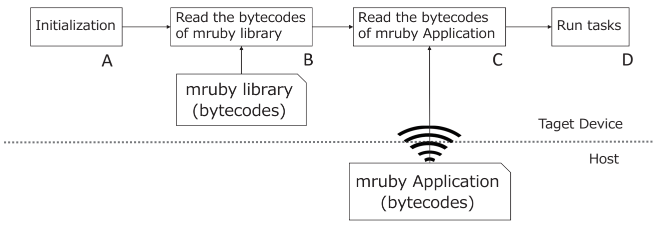


Fig. 2. Component diagram of RiteVM with Bluetooth loader


Initialization
Read the bytecodes of mruby library
Read the bytecodes of mruby Application
Run tasks
A
B
C
D
mruby library (bytecodes)
Taget Device
Host
mruby Application (bytecodes)
Fig. 3. Process flow of RiteVM with Bluetooth loader

```
celltype tRiteVMBluetooth1 {
    entry sTaskBody eMrubyBody;
    [optional] call sEventflag cEventflag[];
    [optional] call sSemaphore cSemaphore;
    attr {
        FLGPTN setPattern;
        [omit]char_t *mrubyLib;
        char_t *irepLib = C_EXP("&$cell_global$_irep");
        uint32_t irepAppSize = C_EXP( BUFFER_SIZE );
    };
    var {
        mrb_state *mrb;
        mrbc_context *context;
        [size_is(irepAppSize)] char_t *irepApp;
    };
};
```

Fig. 4. Celltype description for RiteVM with Bluetooth loader

II. System Model

The proposed framework is an extended framework of mruby on TECS, and utilizes mruby and TECS. These basic technologies are described in appendix of this paper.

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 II-A). The mruby applications run simultaneously due to synchronized processing. The RiteVM scheduler switches RiteVM tasks, and multiple tasks can run concurrently (Section II-B). mruby applications are synchronized by the Eventflag (Section II-C). In this paper, TOPPERS/HRP2 [7] is the target RTOS and is based on μITRON [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.[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.

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 *cell*s. The MrubyTask1 *cell* is a componentized task of the RTOS (TOPPERS/HRP2 [7]). The RiteVM-Bluetooth1 *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 for mruby bytecode, such as RiteVMBluetooth1. In this framework, ZMODEM [9] is used as the binary transfer protocol. The main tRiteVMBluetooth code is shown in Figure 5.

First, the Bluetooth loader receives the mruby application

[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.

```
 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,
          VAR_context );
13   /* Load mruby application bytecode and run */
14   mrb_load_irep_cxt( VAR_mrb, VAR_irepApp,
          VAR_context );
15   /* Omit: synchronization of terminating mruby application */
16   /* Free mruby context */
17   mrbc_context_free( VAR_mrb, VAR_context );
18   /* Free interpreter instance */
19   mrb_close( VAR_mrb );
20 }
```

Fig. 5. Main code for RiteVM with Bluetooth loader

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

Second, *mrb_state* and *mrbc_context* pointers (Figure 3(B); lines 7, 9 in Figure 5). *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 3(C); line 12 in Figure 5). 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 4. *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 during the first compilation.

Fourth, the RiteVM reads the bytecode of the mruby application transferred via Bluetooth (Figure 3(D); line 14 in Figure 5). The mruby application bytecode is stored in an array of *irepApp*. Note that two bytecodes, irepLib and irepApp, are read separately in the RiteVM.

Finally, the mruby task runs (Figure 3(E); line 14 in Figure 5). 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 proposed framework supports continuous

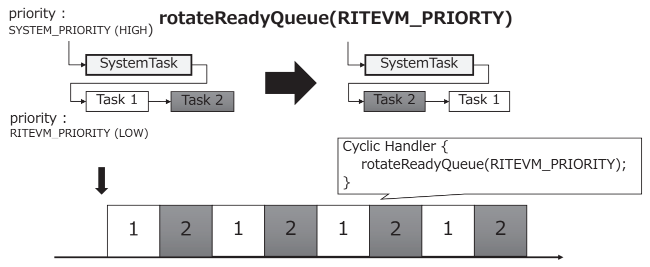


Fig. 6. RiteVM scheduler design

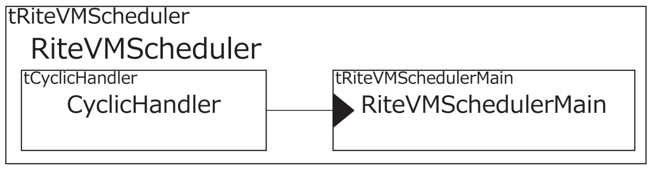


Fig. 7. Component diagram of RiteVM scheduler

loading; thus, this process loops. After the variables are freed (lines 17, 19 in Figure 5), the RiteVM waits for the next mruby application bytecode.

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 [10]). 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 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 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) 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 μITRON to switch tasks with equal priority, is implemented as the main process

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

Fig. 8. Build description of RiteVM scheduler

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 μITRON. Cyclic handlers based on μITRON are described in the literature [8]. The CyclicHandler *cell* has the attributes of the *cell*. The RiteVM-SchedulerMain *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.

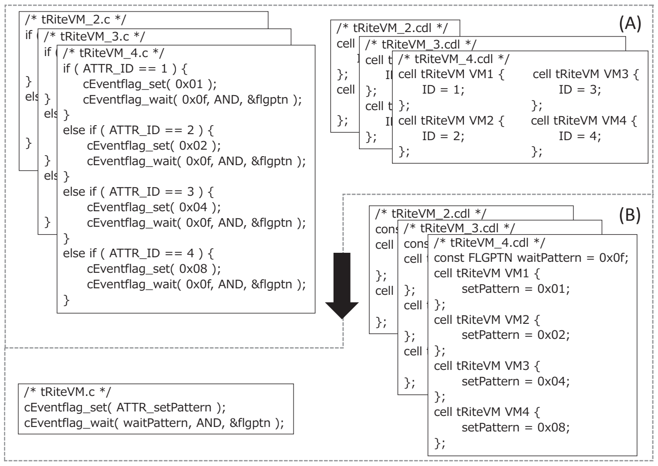
C. Synchronization of Multiple RiteVM Tasks

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.

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,



```
/* tRiteVM_4.c */
if ( ATTR_ID == 1 ) {
    cEventflag_set( 0x01 );
    cEventflag_wait( 0x0f, AND, &flgptn );
}
else if ( ATTR_ID == 2 ) {
    cEventflag_set( 0x02 );
    cEventflag_wait( 0x0f, AND, &flgptn );
}
else if ( ATTR_ID == 3 ) {
    cEventflag_set( 0x04 );
    cEventflag_wait( 0x0f, AND, &flgptn );
}
else if ( ATTR_ID == 4 ) {
    cEventflag_set( 0x08 );
    cEventflag_wait( 0x0f, AND, &flgptn );
}
```

```
/* tRiteVM_4.cdl */
cell tRiteVM VM1 {        cell tRiteVM VM3 {
    ID = 1;                   ID = 3;
};                        };
cell tRiteVM VM2 {        cell tRiteVM VM4 {
    ID = 2;                   ID = 4;
};                        };
```

```
/* tRiteVM.c */
cEventflag_set( ATTR_setPattern );
cEventflag_wait( waitPattern, AND, &flgptn );
```

```
/* tRiteVM_4.cdl */
const FLGPTN waitPattern = 0x0f;
cell tRiteVM VM1 {
    setPattern = 0x01;
};
cell tRiteVM VM2 {
    setPattern = 0x02;
};
cell tRiteVM VM3 {
    setPattern = 0x04;
};
cell tRiteVM VM4 {
    setPattern = 0x08;
};
```

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

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 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 [11] (300MHz ARM9-based Sitara AM1808 system-on-a-chip) compiled with gcc 4.9.3 -O2 and mruby version 1.2.0.

A. 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 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

TABLE I
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

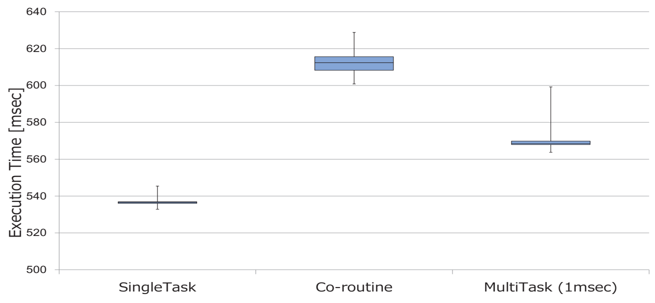


Fig. 10. Comparison of application execution time

platform. The size, load process time, and compilation time for an mruby application with and without mruby library are shown in Table I. 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.

B. RiteVM Scheduler

A comparison of the application execution time with singletasking, co-routine, and multitasking is shown in Figure 10. 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.

1) Execution Time with Singletasking, Co-routine, and Proposed Multitasking: In Figure 10, 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.

2) Periodic Time Overhead: Figure 11 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.

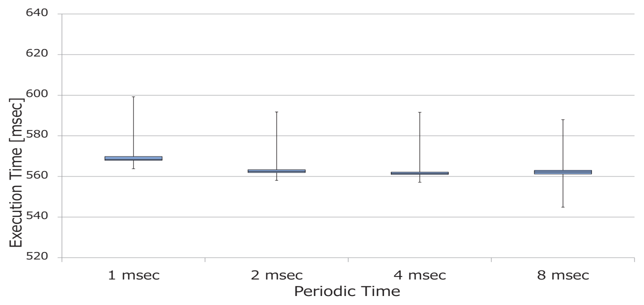


Fig. 11. Comparison of overhead for each cyclic period of RiteVM scheduler

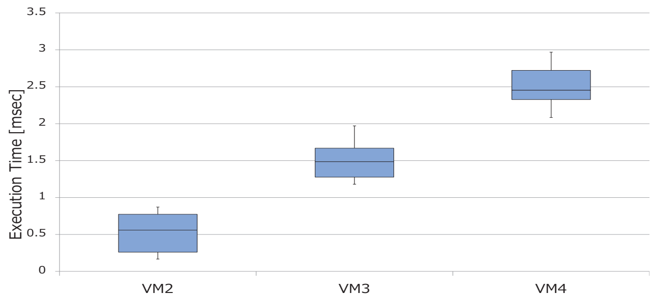


Fig. 12. Synchronization of multiple RiteVM tasks

C. 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)×(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.

D. 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 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. 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.

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 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. 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 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.

References

[1] I. Crnkovic, "Component-based Software Engineering for Embedded Systems," in *Proc. of the 27th International Conference on Software Engineering*, 2005, pp. 712–713.

[2] T. Azumi, M. Yamamoto, Y. Kominami, N. Takagi, H. Oyama, and H. Takada, "A New Specification of Software Components for Embedded Systems," in *Proc. of the 10th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2007, pp. 46–50.

[3] "AUTOSAR," http://www.autosar.org/.

[4] 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*, vol. 80, no. 5, pp. 655–667, 2007.

[5] T. Azumi, Y. Nagahara, H. Oyama, and N. Nishio, "mruby on TECS: Component-Based Framework for Running Script Program," in *Proc. of the 18th IEEE International Symposium on Real-Time Computing*, 2015, pp. 252–259.

[6] K. Tanaka, A. D. Nagumanthri, and Y. Matsumoto, "mruby – Rapid Software Development for Embedded Systems," in *Proc. of the 15th International Conference on Computational Science and Its Applications*, 2015, pp. 27–32.

[7] T. Ishikawa, T. Azumi, H. Oyama, and H. Takada, "HR-TECS: Component technology for embedded systems with memory protection," in *Proc. of the 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2013, pp. 1–8.

[8] H. Takada and K. Sakamura, "μITRON for Small-Scale Embedded Systems," *IEEE Micro*, vol. 15, no. 6, pp. 46–54, 1995.

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

[10] "class Fiber," http://docs.ruby-lang.org/en/2.3.0/Fiber.html.

[11] Y. Li, T. Ishikawa, Y. Matsubara, and H. Takada, "A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support," *OSPERT 2014*, pp. 51–59, 2014.

[12] "p14p," http://code.google.com/archive/p/python-on-a-chip/.

[13] T. W. Barr, R. Smith, and S. Rixner, "Design and Implementation of an Embedded Python Run-Time System," in *Proc. of the USENIX Annual Technical Conference*, 2012, pp. 297–308.

[14] "eLua," http://www.eluaproject.net.

[15] "Squirrel," http://www.squirrel-lang.org/.

[16] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The Evolution of Lua," in *Proc. of the Third ACM SIGPLAN Conference on History of Programming Languages*, 2007, pp. 2–1–2–26.

[17] "mruby-on-ev3rt+tecs," http://www.toppers.jp/tecs.html#mruby_ev3rt.

Appendix A
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 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 13. 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.

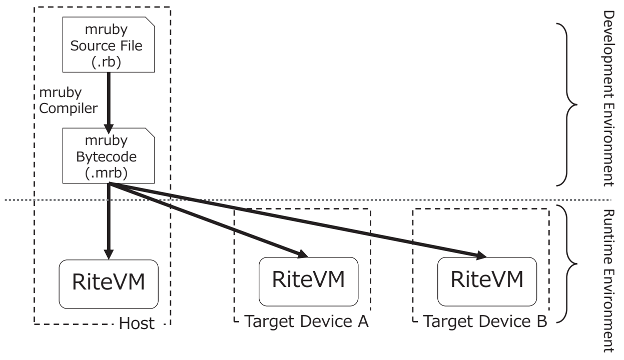


Fig. 13. mruby/RiteVM mechanism

Appendix B
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.

1) Component Model: Figure 14 shows a component diagram. A *cell*, which is an instance of a component in TECS, consists of *entry* ports, *call* ports, attributes and 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 variables of a *cell*.

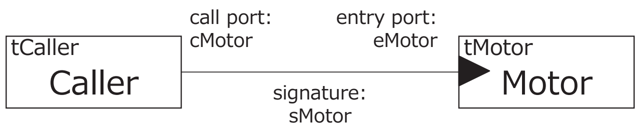


Fig. 14. Component Diagram

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 15). In TECS, to clarify the function of an interface, specifiers such as [in] and [out] are used, which represent input and output, respectively.

```
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 );
};
```

Fig. 15. Signature Description

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 16). 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.

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

Fig. 16. Celltype Description

Build Description
The build description is used to instantiate and connect *cell*s. Figure 17 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 17, *entry* port eMotor in *cell* Motor is connected to *call* port cMotor in *cell* Caller. *C_EXP* calls macros defined in C files.

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

Fig. 17. Build Description

Appendix C
mruby on TECS

A. System Model

The present mruby on TECS system model is shown in Figure 18. 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. mruby on TECS does not depend on the RTOS because TECS supports not only TOPPERS/HRP2 but also the other RTOSs.

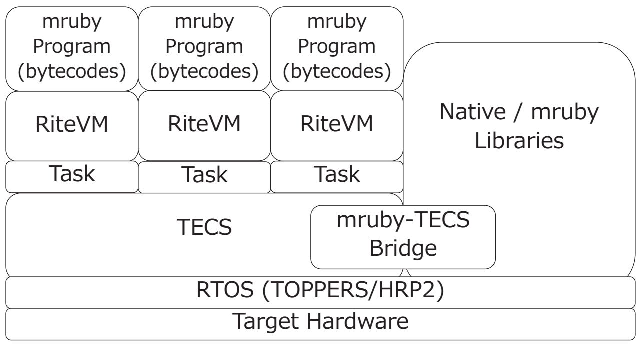


Fig. 18. System model of existing mruby on TECS

B. mruby-TECS Bridge

There is a significant difference between the execution times of mruby and C language codes. According to [5], 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 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 19 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.

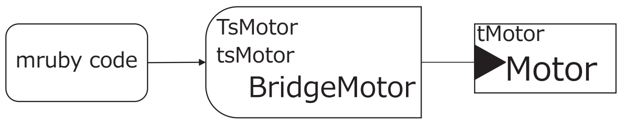


Fig. 19. mruby-TECS bridge

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*.