mruby Bytecode Loader Using Bluetooth in Multi-VM Environment

Takuro Yamamoto

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

1. Introduction

In these days, embedded systems have been required the high-quality and the high-performance. Due to this trend, the complexity of embedded systems also increases and the scale is larger. For example, IoT (Internet of Things) applications. The low production cost and the short developing period of time are also needed.

An approach of efficient software development is to use component-based techniques. CBD (Component-Based Development) is a design technique for constituting reusable components [**?**]. Complex and large scale software systems can be developed efficiently using component-based techniques. That is because software componentization provides high-reusability and easy verification. It also makes a system flexible in extensions and specification changes. There are TECS [**?**], AUTOSAR [**?**], and SaveCCM [**?**] as a typical component-based development for embedded systems

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

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

mruby on TECS is a component-based framework for running script program [**?**]. It is integrated two technologies. One is mruby, which is a script language for embedded systems [**?**], [**?**]. The other is TECS (TOPPERS Embedded Component System), which is a component-based framework for embedded systems [**?**] [**?**]. mruby on TECS supports to effectively run mruby script language on embedded systems. mruby on TECS also makes execution time 100 times faster than that of mruby.

mruby on TECS has several problems at present. One of the problems is ~~that~~ low software development efficiency~~is low because~~ . mruby on TECS only supports ~~an SD card~~ a strage device in the platform for LEGO MINDSTORMS EV3 [**?**] to load mruby programs. ~~In the current development process, it is annoying that the~~ For example, an SD card should be inserted and pulled out repeatedly~~and the OS should be restarted.~~ , or a ROM should be rewritten if mruby programs are modified. Developers should also restart an OS on a target device. ~~Although~~ Moreover, although mruby on TECS has supported the multi-VM, developers need to call OS's function in order to execute multiple tasks. ~~Thus, that is~~ These problem are a big burden on developers.

This paper proposes an extended framework of mruby on TECS: an mruby bytecode loader using Bluetooth and a RiteVM scheduler for fairly executing mruby programs. To improve the development efficiency, the mruby bytecode loader using Bluetooth enables developers to ~~insert an SD card~~ write the platform in a strage device only once at the beginning and to transfer mruby application codes from a host to a target device. The RiteVM scheuler enables developers to run multitasking programs more easily than mruby on TECS.

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

1) To improve the software development efficiency: Developers do not need to ~~insert and pull out an~~

Figure 1. System Model

Figure 2. Mechanism of mruby/RiteVM

~~SD card repeatedly and~~ rewrite a storage device and also to restart an OS. The loader supports the countinuous loading, which saves the Bluetooth set-up time. Therefore, the mruby bytecode loader using Bluetooth helps developers develop software.

2) To effectively execute multiple mruby programs in concurrent or/and parallel:
   Developers can execute multiple tasks without the knowledge of an RTOS because the RiteVM sceduler switches tasks cyclically. In additon, synchronization of multipel tasks is also implemented.
3) To focus on the benefits of component-based developments:
   This paper shows the specific examples for the benefits of component-based development.

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

2. Background

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

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

2.1. mruby

mruby is the light-weight implementation of Ruby programming language complying to part of the ISO standard.

Ruby is a object-oriented script language [**?**]. As the main feature, Ruby is easy-to-use and easy-to-read due to its simple grammar. Ruby can run a program with fewer code lines than C language. Ruby improves the productivity of a software development owing to not only simple grammar also object-oriented functions such as classes and methods, exceptions, and garbage collection.

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

2.2. TECS

TECS (TOPPERS Embedded Component System) is a component system suitable for embedded systems. TECS helps decrease complexity and difficulty because the generated component diagram can visualize the structure of whole

Figure 3. Component Diagram

software. It can also help increase productivity and reduce development duplication because a common part of software is regarded as a component.

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

2.2.1. Component Model.

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

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

2.2.2. Component Description.

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

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

Figure 4. Signature Description

```
1  celltype tCaller{
2      call sMotor cMotor;
3  };
4  celltype tMotor{
5      entry sMotor eMotor;
6      attr{
7          int32_t attr = 100;
8      };
9      var{
10         int32_t var;
11     };
12 };
```

Figure 5. Celltype Description

Signature Description

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

Celltype Description

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

Build Description

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

```
1 cell tMotor Motor{
2 };
3 cell tCaller Caller{
4     cMotor = Motor.eMotor;
5 };
```

Figure 6. Build Description

2.3. mruby on TECS

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

2.3.1. System Model.

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

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

As the target RTOS, TOPPERS/HRP2 [**?**], [**?**] is used in this paper. TOPPERS/HRP2 is an RTOS based on μITRON [**?**] 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 [**?**] and TOPPERS/ASP [**?**].

Figure 7. System Model of existing mruby on TECS

2.3.2. mruby-TECS Bridge.

There is a great difference between the execution time of mruby and C language. According to [**?**], mruby programs are several hundreds times slower than C programs. The execution of mruby bytecode on a RiteVM is not as efficient as that of C language. Thus it is difficult to use mruby for all of code.

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

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

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

Figure 8. mruby-TECS Bridge

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

Figure 9. Detailed System Model of the proposed framework

Figure 10. Development Flow in mruby bytecode loader using Bluetooth

3. Design and Implementation

Figure 9 shows the detailed system model of the proposed framework. Each mruby application bytecode transferred from the host is received by the loader in the RiteVM. The RiteVM reads the own bytecode and executes it. The mruby applications run at the same time because of synchronization processing. The RiteVM scheuler switches tasks as multiple tasks can run in concurrent.

3.1. mruby Bytecode Loader Using Bluetooth

This section describes an additional functionality of mruby on TECS, mruby bytecode loader using Bluetooth. In the current system, ~~all binary data including~~ the platform including mruby bytecodes are saved in ~~an SD card~~a storage device. Developers must ~~insert and pull out the SD card~~ rewrite the storage device every time the application programs are modified. The OS ~~in~~ on the target device should be also restarted. It causes a low development efficiency to repeat that. An mruby bytecode loader using Bluetooth makes the developers' burden decrease. Developers should just once carry out the work to ~~insert the SD card~~ connect the storage device and start the OS.

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

The development flow in an mruby bytecode loader using Bluetooth is shown in Figure 10. The platform including RiteVMs and mruby library is compiled and copied in the ~~SD card~~ storage device at the first. In the host, the mruby application programs (.rb) are edited and compiled into the bytecodes (.mrb) by an mruby compiler. The generated bytecodes are transferred from the host to the target device with Bluetooth.

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

3.1.1. Component of RiteVM with mruby bytecode loader using Bluetooth.

The proposed framework provides a RiteVM with mruby bytecode loader using Bluetooth as TECS component. The component is an extension of the RiteVM component, which is described in [**?**]. The component plays a role in receiving bytecodes via Bluetooth, and also manages a RiteVM configuration such as automatically generates the bytecode in the build description. This generated bytecode is prepared beforehand in the ~~SD card~~ storage device such as mruby libraries, and different from a bytecode transferred with Bluetooth.

Figure 11 shows a component diagram of MrubyTask1 and MrubyBluetooth1 *cell*s. The MrubyTask1 *cell* is a componentized task of the RTOS (TOPPERS/HRP2). TOPPERS/HRP2 is described in [**?**], [**?**]. The MrubyBluetooth1 is a component of RiteVM with mruby bytecode loader using Bluetooth. A bytecode in the host is transferred and received

Figure 11. Component Diagram of mruby bytecode loader using Bluetooth

Figure 12. Control Flow of mruby bytecode loader using Bluetooth

at the top of the component. In this framework, ZMODEM [**?**] is used as a binary transfer protocol.

Figure 12 shows the process of executing mruby program in a component of RiteVM with mruby bytecode loader using Bluetooth such as MrubyBluetooth1. The concrete main code of tMrubyBluetooth is shown in Figure 14.

First, a pointer of *mrb_state* and *mrbc_context* are initialized. *mrb_state* is a set of states and global variables used in mruby.

Second, the RiteVM reads the bytecode of mruby libraries. mruby libraries are a set of Ruby classes such as motor class and sensor class. For example, motor class defines methods to rotate and stop a motor. The tMrubyBluetooth *cell* has two attributes as shown in Figure 13: *mrubyFile* and *irep*. The *mrubyFile* indicates the program files of mruby libraries. *[omit]* is only used for the TECS generator, thus the attribute, *mrubyFile*, does not consume memory. The *irep* is the pointer of the array stored the bytecode of mruby libraries. In short, the bytecode of mruby libraries is stored as an attribute of the component when compiling for the first time.

Third, the RiteVM reads the bytecode of the mruby application transferred with Bluetooth. The mruby application bytecode is stored in an array of type uint8_t such as *irepApp*, which is a tMrubyBluetooth variable as shown in 13. The array is different from that of holding the mruby library bytecode. Two bytecodes are read separately in the RiteVM.

Finally, the mruby task runs. When the mruby application is modified, only the bytecode of the modified application should be transferred. mruby libraries need not be touched because libraries are not normally changed.

```
1  celltype tMrubyBluetooth{
2      entry sTaskBody eMrubyBody;
3      attr{
4          [omit]char_t *mrubyFile;
5          char_t *irep=C_EXP("&$cell_global$_irep");
6      };
7      var{
8          uint8_t *irepApp;
9      };
10 };
```

Figure 13. Celltype Description for RiteVM with mruby bytecode loader using Bluetooth

3.2. Multitask

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

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

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

As an approach for multitask processing, the proposed framework provides the RiteVM scheuler which is a fair

```
1  void
2  eMrubyBody_main(CELLIDX idx)
3  {
4    /* Declaration variables */
5    mrb_state *mrb;
6    mrbc_context *c;
7    /* New interpreter instance */
8    mrb = mrb_open();
9    // Omit: error check for mrb_state
10   /* New mruby context */
11   c = mrbc_context_new(mrb);
12   // Omit: initialization of mruby-TECS bridge
13   /* Receive the bytecode via Bluetooth */
14   bluetooth_loader( VAR_irepApp );
15   /* Load mruby library bytecode and run */
16   mrb_load_irep_cxt(mrb, ATTR_irep, c);
17   /* Load mruby application bytecode and run */
18   mrb_load_irep_cxt(mrb, VAR_irepApp, c);
19   if (mrb->exc) {
20     /* Failure to execute */
21     mrb_p(mrb, mrb_obj_value(mrb->exc));
22     exit(0);
23   }
24   /* Free mruby context */
25   mrbc_context_free(mrb, c);
26   /* Free interpreter instance */
27   mrb_close(mrb);
28 }
```

Figure 14. Main code for RiteVM with mruby bytecode loader using Bluetooth

scheduler and runs multiple tasks equally. The RiteVM scheuler is utilized only when application tasks have the same priority. mruby applications can run in concurrent without developers calling the OS's function. The application programs can also utilize the existing programs since the structures of the programs are not changed.

3.2.1. RiteVM Scheduler.

A RiteVM scheduler is a cyclic handler, and calls *rotateReadyQueue*, a service call of μITRON to switch tasks with the same priority. In other words, the RiteVM scheduler calls *rotateReadyQueue* cyclically. *rotateReadyQueue* is described as follows.

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

When *rotateReadyQueue* is called, tasks with the same priority are switched as shown in Figure 15. The argument of *rotateReadyQueue* is the priority.

The *rotateReadyQueue* can be performed if the number of tasks is more than two. For example, three tasks are in the order: task 1, 2, and 3. In this case, the order is rotated, task 2, 3 and, 1, when the *rotateReadyQueue* is called.

3.2.2. Component of RiteVM Scheduler.

Figure 16 shows a component diagram of the cyclic handler. The components of cyclic handler consist of two components: CyclicHandler and CyclicMain. CyclicHandler *cell* configures the cyclic handler based on μITRON. Cyclic handlers based on μITRON are described in detail [**?**]. The cyclic handler has five arguments: ID, attribute, cyclic time, cyclic phase and access pattern. CyclicHandler *cell* has these arguments as attributes of the *cell*. CyclicMain *cell* is a component to perform the processing body of a cyclic handler. *rotateReadyQueue* is implemented as the body. Figure 17 shows tCyclicMain *celltype*, which has a *call* port, an *entry* port and an attribute. The *call* port is connected with the *entry* port of the Kernel *cell* (*tkernel.eiKernel*) to call functions of the kernel. The attribute is used as an arguments of *rotateReadyQueue*.

Figure 15. rotateReadyQueue

Figure 18 shows a build description that corresponds to the component diagram shown in Figure 16. In the part of CyclicHandler *cell*, configurations of a cyclic handler is described such as attribute, cyclicTime and cyclicPhase. In this case, the cyclic handler is executed when it is generated because the attribute is *TA_STA* that represents the cyclic handler is in an operational state after the creation. The cyclic handler is executed every one msec. In another part, the priority of CyclicMain *cell* is described. RITEVM_PRIORITY defines the priority of mruby tasks. In the main of CyclicMain, *rotateReadyQueue* is implemented and the priority is passed as the argument.

3.2.3. Synchronization of Multiple RiteVM Tasks.

In the proposed framework, RiteVMs read mruby bytecodes, and then execute the applications. Eventflag, one of synchronous processing, is applied to synchronize the starting of multiple mruby applications. Each task sets the flag pattern such as 0x01 (01) and 0x02 (10), and then waits the flag pattern, 0x3 (11), with AND. This process can also

Figure 16. Component Diagram of Cyclic Handler

```
celltype tCyclicHandler {
    [inline] entry sCyclic eCyclic;
    call siHandlerBody ciBody;
    attr {
        [omit] ATR attribute = C_EXP("TA_NULL");
        [omit] RELTIM cyclicTime;
        [omit] RELTIM cyclicPhase = 0;
    };
};
celltype tCyclicMain{
    require tKernel.eiKernel;
    entry siHandlerBody eiBody;
    attr {
        PRI priority;
    };
};
```

Figure 17. Celltype Description of Cyclic Handler

apply to the case of the more tasks. For example, in the case of the four RiteVM tasks, each task sets the flag pattern such as 0x01 (0001), 0x02 (0010), 0x04 (0100) and 0x08 (1000), and then waits 0x0f (1111) with AND as shown in Figure 19 (A).

3.3. Benefits of Component-Based Development

This section describes the design using the benefits of component-based development.

In the framework, RiteVMs, the RiteVM scheuler, and Eventflag are implemented as components. Therefore, developers can easily add or remove these components and also reuse them. For example, if developers do not need the RiteVM scheuler, the .cdl files shown as Figures 17 and 18 should be commented out such as *//import(<tRiteVMScheduler.cdl>);*. This advantage of

```
cell tCyclicHandler CyclicHandler{
    ciBody = CyclicMain.eiBody;
    attribute = C_EXP("TA_STA");
    cyclicTime = 1;
    cyclicPhase = 1;
};
cell tCyclicMain CyclicMain{
    priority =
        C_EXP("RITEVM_PRIORITY");
};
```

Figure 18. Build Description of Cyclic Handler

Figure 19. The design for Eventflag using TECS

CBD saves developers the labor of rewriting a kernel configuration file.

In additon, the code size can desrease by developing with component-based. In the proposed framework, the advantage is applied in the Eventflag component. The set pattern and wait pattern are defined as attributes of the component as shown in Figure 19 (B). This design such as *cEventflag_set(ATTR_setPattern)* enables the program without "if" statements and reuses the identical .c file. Developers do not need to modify the .c file because the .cdl files are prepared in accordance with the number of RiteVMs In additon, the components of Eventflag are built with *[optional]* in TECS. *[optional]* means that the codes are run only when the call port is connected. The .c file does not be rewritten even if developers do not use Eventflag.

4. Experimental Evaluation

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

- Size and time for transferred mruby bytecodes

TABLE 1. COMPARISON OF THE SIZE AND LOAD PROCESS TIME BETWEEN AN MRUBY APPLICATION INCLUDING MRUBY LIBRARIES AND NOT

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

- Execution time with singletasking, co-routine, and multitasking
- Overhead for cyclic time
- Code size using the benefit of CBD

These evaluations are performed in order to indicate that an mruby bytecode loader improves the software development efficiency, and that the proposed multitask processing effectively executes compared with singletasking or co-routine, and also the overhead of the cyclic period. This paper demonstrates the proposed system on a LEGO MINDSTORMS EV3 (300MHz ARM9-based Sitara AM1808 system-on-a-chip) compiled with gcc 4.9.3 -O2 and mruby version 1.2.0.

The size, load process time, and compilation time for mruby application and mruby application including library is shown in Table 1. The overhead of load processing is 50.933 msec, which it takes to load a zero bytes bytecode. Similarly, the overhead of compilation is 46.9 msec, which it takes to compile a zero bytes program. The bytecode of mruby application is smaler and faster than that of including mruby libraries in all terms. In the proposed framework, developers send only mruby application and prepare mruby library in advance. Therefore, the design can send the bytecode faster. In addition, the design can save the time to ~~insert an SD card~~ connect a storage device to restart an OS. These advantages lead improvement of the software development efficiency.

The comparison of the application execution time with singletasking, co-routine, and multitasking is shown in Figure 20. The 100,000 times loop program is used as mruby application for evaluation of execution time. In detail, the singletask program loops 100,000 times, and the multitask and co-routine programs loop 50,000 times in each task. In Figure 20, the cyclic time of the cyclic handler for multitasking is one msec. Multitasking's execution time is about the same as singletasking's while co-routine takes more time than them to execute an mruby application. This result shows the proposed design is superior to co-routine in terms of execution time. Switching tasks' overhead is also evaluated, that is about three μsec on average. The number of switching tasks is from 500 to 600 because singletasking process in Figure 20 takes about 540 msec. Therefore, the overhead of multitasking is 0.5 % or less, and multitasking can be used without the overhead.

Figure 21 shows the execution time of multitasking with the cyclic handler. A lower limit of the cyclic time is one msec due to the specification of TOPPERS/HRP2, the used RTOS. More than 10 msec do not be evaluated in this paper because it is thought the larger cyclic time influences applications. Each execution time of cyclic period

Figure 20. Comparison of the application execution time with singletask, co-routine, and multitask

Figure 21. Comparison of the overhead for each cyclic period of calling rotateReadyQueue

is the same as the others. That is because the overhead of switching tasks (about 3 μsec) is small in comparison with the execution time. The smaller cyclic time is better in multitasking due to concurrent and/or parallel processing.

CodeSize

5. Related Work

The open-source run-time systems for scripting languages have been proposed such as follow: python-on-a-

TABLE 2. COMPARION OF THE PROPOSED AND PREVIOUS WORK

	Bluetooth Loader	Call C Function	Legacy Code of Embedded System	VM Managenment	VM Scheduler	Synchronization of Application	Co-routine
python-on-a-chip [?]							✓
Owl system [?]		✓	Partially				✓
eLua [?]		✓	Partially				✓
mruby [?]		✓					✓
mruby on TECS [?]		✓	✓	✓	△		✓
Proposed framework	✓	✓	✓	✓	✓	✓	✓

chip [?], the Owl system [?], eLua [?], mruby [?], [?], and mruby on TECS [?].

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

The Owl system is an embedded Python run-time system. The Owl 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. The interpreter of the Owl system is the same as that of python-on-a-chip.

eLua offers the full implementation of Lua programming language to the embedded systems. Lua is one of the most popular script languages for embedded systems [?], [?]. Lua supports co-routine, referred to collaborative multitasking. A co-routine in Lua is used as an independently executed thread. A co-routine can just suspend and resume multiple routines. Thus, a Lua co-routine is not like multitasks in multitask systems.

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

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

Table 2 shows a comparison between the proposed framework and previous work.

6. Conclusion

This paper has presented an extended framework of mruby on TECS: mruby bytecode loader using Bluetooth and RiteVM scheuler. The loader provides developers with the software development efficiency without ~~disconnecting an SD card~~ rewriting a storage device and restarting an OS. The proposed framework can be applied to many kinds of embedded systems because the loader can use not only Bluetooth also wired serial connection. The RiteVM scheuler makes multitasking more easily than the current mruby on TECS. In the evaluation, experimental results of the loader and the RiteVM scheduler show their advantages. The loader can improve the software development efficiency on mruby on TECS. The RiteVM sceduler has the effectiveness in terms of execution time and ease of use compared with singletasking and co-routine because of the low overhead.

The proposed framework is developed in component-base by TECS. The facilities such as RiteVMs, the RiteVM scheuler, and Eventflag are implemented as components. Therefore, developers can easily add or remove the functionalities as necessary, and also reuse them. Developers can choose fair scheduling or fixed-priority scheduling since the RiteVM scheuler can be easily removed. Component-based development can increase productivity and decrease complexity.

In the future, the mruby bytecode loader using Bluetooth will be supported to handle multiple bytecodes and run the application in multi-VM. Moreover, the .cdl files for RiteVM and mruby-TECS bridge are generated automatically using a plugin, and developers can send a bytecode with ZMODEM protocol on the command line.