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 at the time of the first compailation. In addition, multiple mruby programs cooperatively run in the proposed framework. In the proposed framework, the cyclic handler switching tasks is implemented, and multitasking processing is more easy-to-use than that of mruby on TECS. 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 [1]. 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 [2], AUTOSAR [3], and SaveCCM [4] 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, 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 [5]. It is integrated two technologies. One is mruby, which is a script language for embedded systems [6], [7]. The other is TECS (TOPPERS Embedded Component System), which is a component-based framework for embedded systems [2]. 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.

This paper proposes two additional features of mruby on TECS, mruby bytecode loader using Bluetooth and user-friendly multitask. mruby on TECS has some problems at present. One of the problems is that working efficiency is low because mruby on TECS only supports an SD card in the platform for LEGO MINDSTORMS EV3 [8] to load mruby programs. In the current development process, it is annoying that the SD card should be insert and pulled out repeatedly. If a new feature, mruby bytecode loader Bluetooth, is used, the procedure is carried out only once at the beginning and a program code is transferred from a host to a target device by Bluetooth. It can improve the efficiency. Moreover, in the current system if developers attempt to run multiple tasks with the same priority, a task with first execution only runs and all other tasks would not run unless a OS's function must be called by developers. In addition developers must prepare as many VMs as the number of threads because one RiteVM supports only one thread. Thus that is a big burden on developers. Another new feature enables developers to run a multitask programs more easily. Specifically, a cyclic handler of switching tasks is implemented. This paper evaluates the proposed framework in term of data transfer rate, overhead of cyclic handler and execution time.

Contributions: This paper proposes mruby bytecode loader using Bluetooth and user-friendly multitasking. The proposed framework gives the contribution in the following points.

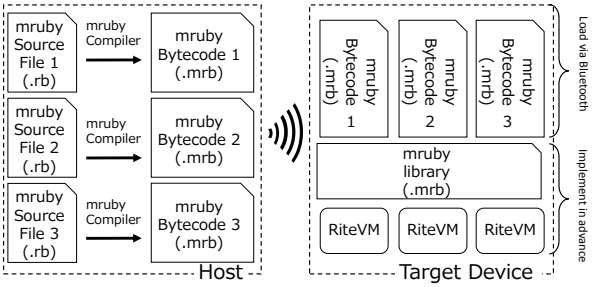

Figure 1. System Model

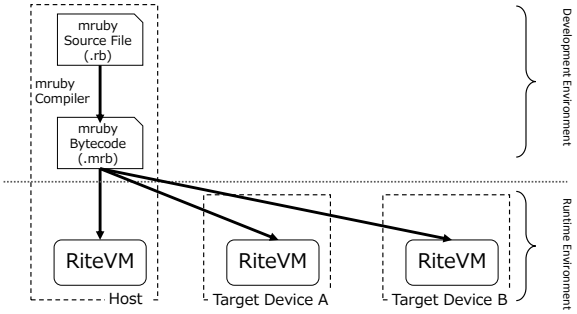

Figure 2. Mechanism of mruby/RiteVM

1) To improve the work efficiency in software development on mruby on TECS:
   mruby bytecode loader using Bluetooth helps developers develop software. Developers do not need to insert and pull out an SD card repeatedly and also to restart an OS. Thus, the software development efficiency is improved.
2) To effectively execute multiple mruby programs in concurrent or/and parallel:
   There is a problem in multitasking of the current mruby on TECS. It is that developers need to be well acquainted the RTOS if multiple mruby programs execute in multitasking. In the proposed framework, it is easy for developers to use multitasking because multiple tasks are switched cyclically. Multitasked programing makes the range of software expand.

Organization: The paper is organized as follows. Section 2 introduces the basic technologies i.e. mruby, TECS and mruby on TECS. Section 3 and Section 4 describe the design and implementation of the proposed framework in detail. Section 5 evaluates the proposed framework, Section 6 discuses related work, and then Section 7 concludes.

2. Background

Figure 1 shows a 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 some 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 [9]. 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, so mruby programs can run on any operating system as long as VM is implemented. 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 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 the complexity and difficulty because the generated component diagram can visualize the structure of whole software. It can also help increase the 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*

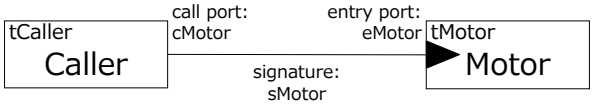

Figure 3. Component Diagram

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

Figure 4. Signature Description

has one or more *entry* ports and *call* ports. Functions of a *cell* are implemented in the C language.

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

2.2.2. Component Description.

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

Signature Description

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

Celltype Description

The *celltype* description defines *entry* ports, *call* ports, attributes, and 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.

```
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

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

Figure 6. Build Description

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.

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 [10], [11] is used in this paper. TOPPERS/HRP2 is an RTOS based on μITRON with memory protection. However, mruby on

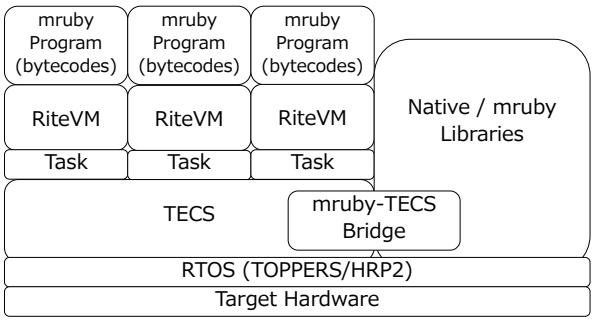

Figure 7. System Model of existing mruby on TECS

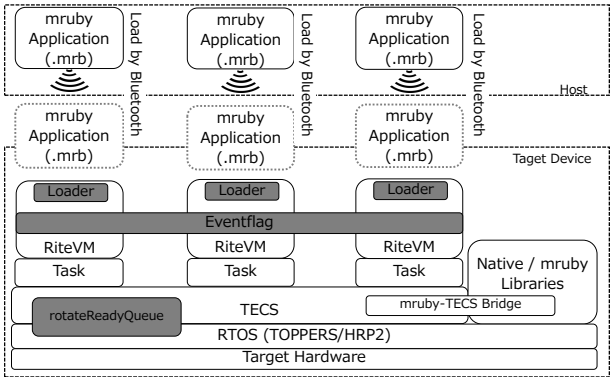

Figure 8. Specific System Model of the proposed framework

TECS does not depend on the RTOS because TECS supports not only TOPPERS/HRP2 but also the other RTOSs such as OSEK [12] and TOPPERS/ASP [13].

2.3.2. mruby-TECS Bridge.

There is a great difference between the execution time of mruby and C language. According to [5], 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 9 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.

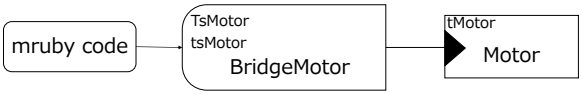

Figure 9. mruby-TECS Bridge

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*.

3. mruby Bytecode Loader Using Bluetooth

This section describes an additional feature of mruby on TECS, mruby bytecode loader using Bluetooth. In the current system, all binary data including bytecodes are in an SD card. Developers must insert and pull out the SD card every time the source programs are modified. It causes a low work efficiency that such a work must be repeated. mruby bytecode loader using Bluetooth makes the developers' burden decrease. The work that an SD card is in and out needs to be done just once in the beginning.

mruby programs are consisted of mruby application and libraries. mruby application is the main code of the application. mruby libraries are the code defining the functions for application such as Ruby class. The mruby bytecodes including mruby application and libraries can be sent and run. However it is wasteful in terms of the size and time to send because libraries are not frequently modified. In the proposed framework, mruby application only sends, and mruby libraries are preserved in the SD card beforehand.

The development flow in mruby bytecode loader using Bluetooth is shown in Figure 10. The RiteVM and mruby library are compiled and copied in the SD card at the first. These are assumed not to be modified. The binary data transferred with Bluetooth is the bytecode of the main source. In the host, the source files (.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.

3.1. Component of mruby bytecode loader using Bluetooth

The proposed framework provides components of mruby bytecode loader using Bluetooth to use the function. A component of mruby bytecode loader using Bluetooth is an extension of the RiteVM component, which is described in [5]. The component plays a role in receiving bytecodes via Bluetooth, and also manages RiteVM configuration such

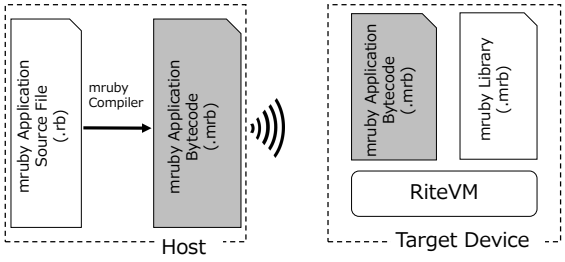

Figure 10. Development Flow in mruby bytecode loader using Bluetooth

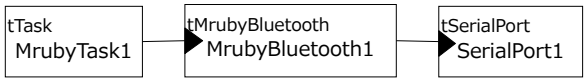

Figure 11. Component Diagram of mruby bytecode loader using Bluetooth

as automatically generates the bytecode in the build description. This generated bytecode is prepared beforehand in the SD card 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 [10], [11]. The MrubyBluetooth1 is a component of mruby bytecode loader using Bluetooth. A bytecode which is a compiled mruby program in the host is transferred and received at the top of the mruby bytecode loader using Bluetooth component. In this framework, ZMODEM [14] is used as a binary transfer protocol.

Figure 12 shows the process of executing mruby program in a component of mruby bytecode loader using Bluetooth, which is like MrubyBluetooth1. First, a pointer of *mrb_state* structure is initialized. *mrb_state* is a set of states and global variables used in mruby. Second, the bytecode of mruby libraries is read. 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 bytecode of the mruby program transferred with Bluetooth is read . The bytecode transmitted via Bluetooth is stored in an array of type uint8_t, which is the same as type char. The array is different from that of holding the mruby library bytecode. Two bytecodes are read separately in the

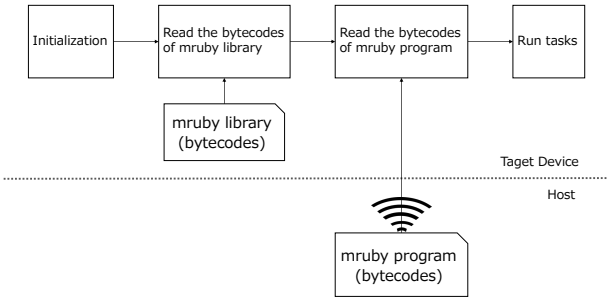

Figure 12. Control Flow of mruby bytecode loader using Bluetooth

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

Figure 13. Celltype Description for mruby bytecode loader using Bluetooth

mrb_state. Finally, an mruby task runs. When the mruby program is modified, only the bytecode of the modified program should be transferred. mruby libraries need not be touched because libraries are not normally changed. The concrete main code of tMrubyBluetooth is shown in Figure 14.

4. 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 [15]) Co-routine is a non-preemptive mulititasking, and does not receive the OS's support because developers have to switch tasks manually. Co-routine can not take advantege 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. In the approach of *delay* fubction, developers have to schedule as the same as co-routine.

In the proposed framework, as an approach for multitask processing, *rotateReadyQueue*, a service call of μITRON, is implemented. *rotateReadyQueue* rotates the precedence of the task with the priority of the argument. A cyclic

```
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( ATTR_irepApp );
15   /* Load mruby library bytecode and run */
16   mrb_load_irep_cxt(mrb, ATTR_irepLib, c);
17   /* Load mruby application bytecode and run */
18   mrb_load_irep_cxt(mrb, ATTR_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 mruby bytecode loader using Bluetooth

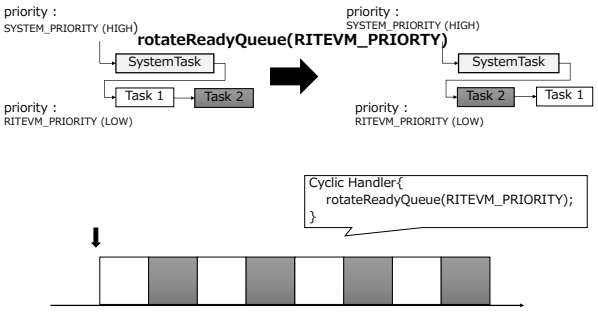

Figure 15. rotateReadyQueue

handler calls *rotateReadyQueue*, which enables developers to develop applications without concern for multitasking.

4.1. rotateReadyQueue

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.

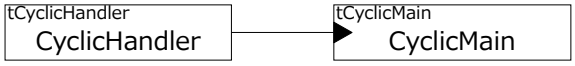

Figure 16. Component Diagram of Cyclic Handler

4.2. Component of Cyclic Handler

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 ITRON are described in detail [16]. 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 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. EV3_MRUBY_VM_PRIORITY defines the priority of mruby tasks. In the main of CyclicMain, *rotateReadyQueue* is implemented and the priority is passed as the argument. If developers do not need the cyclic handler, the .cdl files shown as Figures 17 and 18 should be commented out such as *//import(<tCyclicHandler.cdl>);*. This ease comes from component-based development with TECS.

4.3. 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 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.

In the proposed framework, the set pattern and wait pattern are defined as attributes of the component

```
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

```
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

as shown in Figure 19. 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 in accordance with the number of RiteVMs are prepared. 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. These features are also advantages of component-based development.

5. Experimental Evaluation

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

- the comparison of the size and transfer time between an mruby application including mruby library and not
- the comparison of the application execution time with singletasking, co-routine, and multitasking
- the comparison of the overhead for each cyclic period of calling *rotateReadyQueue*

These evaluations are performed in order to indicate that mruby bytecode loader improves the work 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

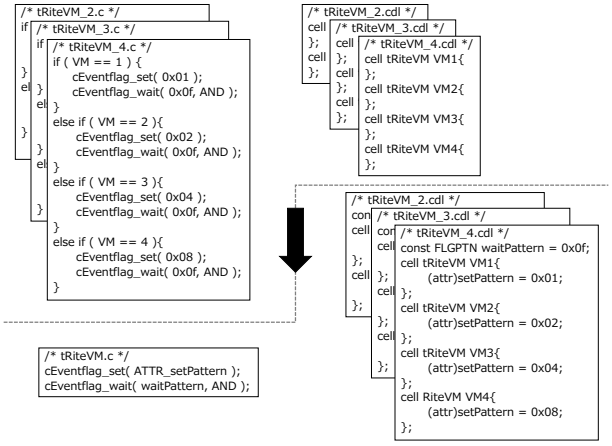

Figure 19. The design for Eventflag using TECS

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

	mruby Application & Library	mruby Application
Size	14,044 bytes	199 bytes
Transfer Time	356.015 msec	58.708 msec

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 and transfer time for mruby application and mruby application including library is shown in Table 1. The bytecode of mruby application including library is larger and slower than that of only mruby application in terms of size and transfer time. 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 to restart an OS. These advantages lead to improving the work 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's 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,

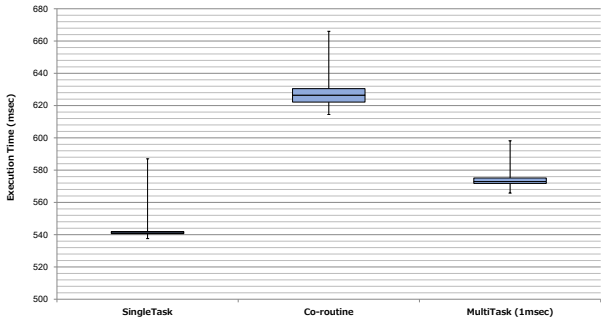

Figure 20. Comparison of the application execution time with singletask, co-routine, and multitask

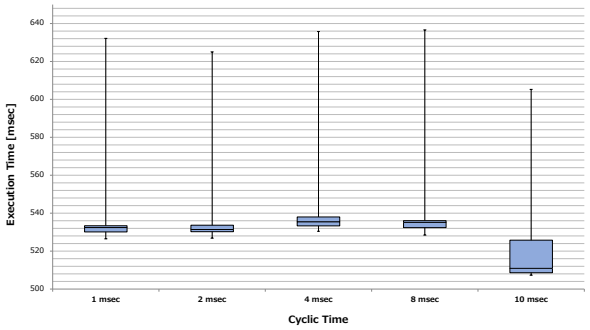

Figure 21. Comparison of the overhead for each cyclic period of calling rotateReadyQueue

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 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.

6. Related Work

The open-source run-time systems for scripting languages have been proposed such as follow: eLua [21], python-on-a-chip [19], the Owl system [20], mruby [6], [7], and mruby on TECS [5].

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 [17], [18]. 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.

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.

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.

7. Conclusion

This paper has presented the mruby bytecode loader using Bluetooth in multi-VM on mruby on TECS. The proposed framework provides developers the efficient development without disconnecting an SD card and restarting an OS. Furthermore, the proposed multitasking can be use more easily than the current mruby on TECS. In the evaluation, experimental results of the mruby bytecode loader and proposed multitasking show their advantages. mruby bytecode loader using Bluetooth can improve the work efficiency. The proposed multitasking has the effectiveness in terms of execution time 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 the loader, the cyclic handler, and the Eventflag are implemented as components. Developers can easily add or remove the facilities as necessary. Component-based development can increase the productivity and decrease the 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.

References

[1] I. Crnkovic, "Component-based Software Engineering for Embedded Systems," in *Proceedings 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 *Proceedings of the 10th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2007, pp. 46–50.

TABLE 2. COMPARION OF THE PROPOSED AND PREVIOUS WORK

	Bluetooth Loader	Call C function	Legacy code of embedded system	VM managenment	Synchronization of application	Co-routine
Lua [17], [18]		✓	Partially			✓
python-on-a-chip [19]						✓
Owl system [20]		✓	Partially			✓
mruby [6], [7]		✓				✓
mruby on TECS [5]		✓	✓	✓		✓
Proposed framework	✓	✓	✓	✓	✓	✓

[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 *Proceedings of the 18th IEEE International Symposium on Real-Time Distributed Computing (ISORC)*, 2015, pp. 252–259.

[6] K. Tanaka, Y. Matsumoto, and H. Arimori, "Embedded System Development by Lightweight Ruby," in *Proceedings of the International Conference on Computational Science and Its Applications (ICCSA)*, 2011, pp. 282–285.

[7] "mruby," https://github.com/mruby/mruby.

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

[9] "Ruby," https://www.ruby-lang.org/en/.

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

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

[12] A. Ohno, T. Azumi, and N. Nishio, "TECS Components Providing Functionalities of OSEK Specifications for ITRON OS," *Journal of Information Processing*, vol. 22, no. 4, pp. 584–594, 2014.

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

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

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

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

[17] "Lua," http://www.lua.org/.

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

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

[20] T. W. Barr, R. Smith, and S. Rixner, "Design and Implementation of an Embedded Python Run-Time System," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 297–308.

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