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 needs to be familiar with the functions of real-time operating systems to effectively execute multiple mruby programs in concurrent or/and parallel. In this paper, to improve the development efficiency, the mruby bytecode loader using Bluetooth is implemented as an extension of mruby on TECS. In addition, multiple mruby programs run cooperatively in the proposed framework. Experimental results demonstrates 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 [4]. 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.

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 [1]. It is integrated two technologies. One is mruby, which is a script language for embedded systems [9]. 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 these is that working efficiency is low because mruby on TECS only supports a SD card in the platform for LEGO MINDSTORMS EV3 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 cycle 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

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 repeatedly, and that improves the software development efficiency.
2) To effectively execute multiple mruby programs in concurrent or/and parallel:
   There is a probrem 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

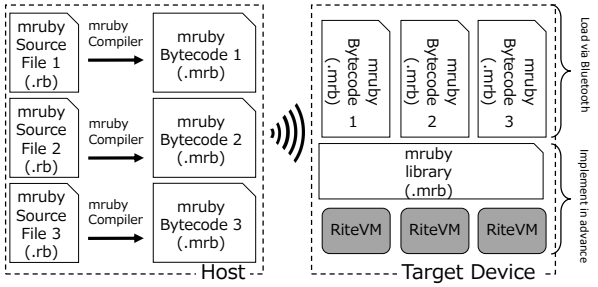

Figure 1. System Model

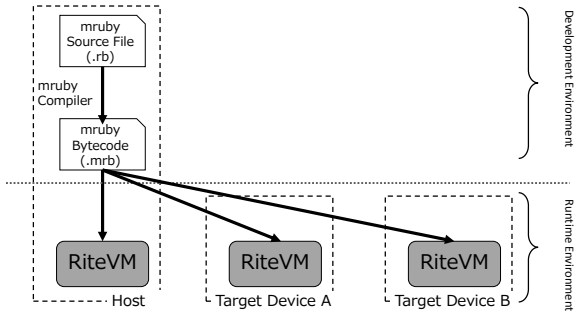

Figure 2. Mechanism of mruby/RiteVM

framework, it is easy for developers to use multitasking because multiple tasks are switched cyclically. Developers can easily program multitasking. Multitasked programing makes the range of software expand.

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. 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 a mruby code into a bytecode, which is an intermidiate code that can be interpreted by RiteVM. The bytecodes can run on a Rite VM, 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 overheads 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.

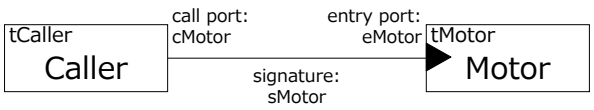

Figure 3. Component Diagram

```
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

The description of a component in TECS is divided into three parts, *signature*, *celltype*, and build description. The component description in TECS 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.

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  celltype tCaller{
2      call sMotor cMotor;
3  };
4
5  celltype tMotor{
6      entry sMotor eMotor;
7      attr{
8          int32_t attr = 100;
9      };
10     var{
11         int32_t var;
12     };
13 };
```

Figure 5. Celltype Description

```
1  cell tMotor Motor{
2  };
3
4  cell tCaller Caller{
5      cMotor = Motor.eMotor;
6  };
```

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 [7] 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 [8] and TOPPERS/ASP.

2.3.2. mruby-TECS Bridge.

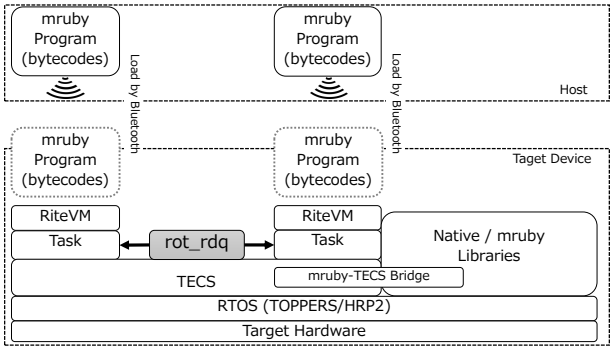

Figure 7. System Model

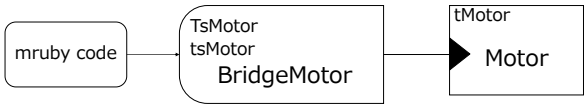

Figure 8. mruby-TECS Bridge

There is a great difference between the execution time of mruby and C language. According to [1], 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.

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 a mruby program, an mruby-TECS bridge calls the function defined in the TECS component such as a Motor *cell*.

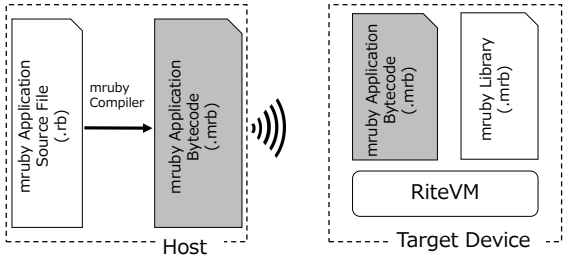

Figure 9. Development Flow in mruby bytecode loader using Bluetooth

3. mruby Bytecode Loader Using Bluetooth

This section describes an additional function of mruby on TECS, mruby bytecode loader using Bluetooth. In the current system, all binary data including bytecodes are in a SD card. Developers must insert and pull out the SD card every time the source program 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 a SD card is in and out needs to be done just once in the beginning.

The development flow in mruby bytecode loader using Bluetooth is shown in Figure 9. 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, which is identified by the .rb extension, are edited and compiled into the bytecodes by a 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 [1]. The component plays a role in receiving bytecodes via Bluetooth, and also manages RiteVM configuration such as automatically generates the bytecode in the build description. This bytecode in the build description means that prepared beforehand in the SD card such as mruby libraries, and different from a bytecode transferred with Bluetooth.

Figure 10 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 [7]. 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 [5] is used as a binary transfer protocol.

Figure 10. Component Diagram of mruby bytecode loader using Bluetooth

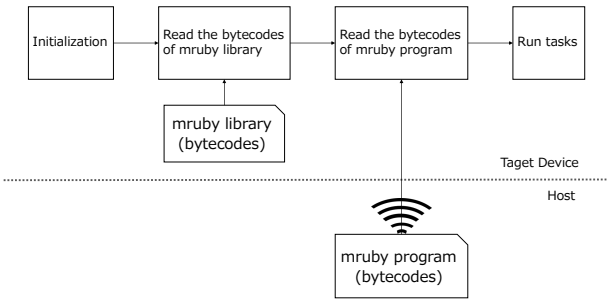

Figure 11. Control Flow of mruby bytecode loader using Bluetooth

Figure 11 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 12: mrubyFile and irep. Here, 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 mrb_state. Finally, a 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 13.

```
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  };
```

Figure 12. Celltype Description for mruby bytecode loader using Bluetooth

```
1  void
2  eMrubyBody_main(CELLIDX idx)
3  {
4    /* Receive the bytecode via Bluetooth */
5    serial_loader(SIO_PORT_BT);
6
7    /* Declaration variables */
8    CELLCB *p_cellcb;
9    mrb_state *mrb;
10   mrbc_context *c;
11
12   if (VALID_IDX(idx)) {
13     p_cellcb = GET_CELLCB(idx);
14   }
15
16   /* New interpreter instance */
17   mrb = mrb_open();
18   if (mrb == NULL) {
19     /* Invalid mrb_state */
20     return;
21   }
22   /* New mruby context */
23   c = mrbc_context_new(mrb);
24
25   if(is_cInit_joined()){
26     /* Initialization of mruby TECS Bridge */
27     cInit_initializeBridge( mrb );
28   }
29
30   /* Load mruby liblary bytecode and run */
31   mrb_load_irep_cxt(mrb, ATTR_irep, c);
32   /* Load mruby application bytecode and run */
33   mrb_load_irep_cxt(mrb, app_binary_buf, c);
34
35   if (mrb->exc) {
36     /* Failure to execute */
37     mrb_p(mrb, mrb_obj_value(mrb->exc));
38     exit(0);
39   }
40   /* Free mruby context */
41   mrbc_context_free(mrb, c);
42   /* Free interpreter instance */
43   mrb_close(mrb);
44 }
```

Figure 13. Main code for mruby bytecode loader using Bluetooth

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. In the proposed framework, multitask processing is user-friendly because developers need the less knowledge than before. A cyclic handler that switches tasks is implemented, and that enables developers to develop

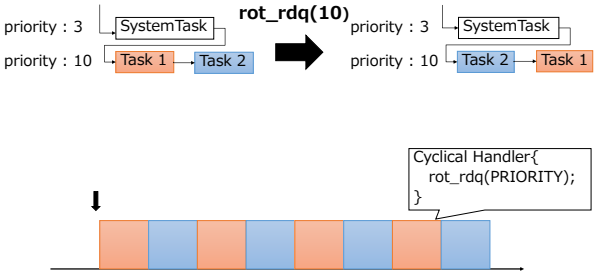

Figure 14. rot_rdq

Figure 15. Component Diagram of Cyclic Handler

```
1  celltype tCyclicHandler {
2      [inline] entry sCyclic eCyclic;
3      call siHandlerBody ciBody;
4      attr {
5          ID id = C_EXP( "CYCHDLRID_$id$" );
6          [omit] ATR attribute = C_EXP("TA_NULL");
7          [omit] RELTIM cyclicTime;
8          [omit] RELTIM cyclicPhase = 0;
9          [omit] ACPTN accessPattern[4] =
10             { C_EXP("OMIT"), C_EXP("OMIT"),
                   C_EXP("OMIT"), C_EXP("OMIT") };
11     };
12 };
13
14 celltype tCyclicMain{
15     require tKernel.eiKernel;
16     entry siHandlerBody eiBody;
17     attr {
18         PRI priority;
19     };
20 };
```

Figure 16. Celltype Description of Cyclic Handler

applications without concern for multitasking.

4.1. rot_rdq

The case is assumed that there are two tasks with the same priority, and both are in a 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 a loop.

The rot_rdq is a service call of μITRON. When rot_rdq is called, tasks with the same priority are switched as shown Figure 14. The argument of rot_rdq is the priority.

The rot_rdq 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 changed, task 2, 3 and 1, when the rot_rdq is called.

4.2. Component of Cyclic Handler

Figure 15 shows a component diagram of the cyclic handler. The components of cyclic handler consist of two components: CyclicHandler and CyclicMain. CyclicHandler *cell* configurates the cyclic handler based on ITRON. Cyclic handlers based on ITRON 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. Here, rot_rdq is implemented as the body. Figure 16 shows tCyclicMain *celltype*, which has a *call* port which is connected tkernel.eiKernel to call functions of the kernel, an *entry* port and an attribute, priority. The attribute, priority is used as an arguments of rot_rdq.

Figure 17 shows a build description that corresponds to the component diagram shown in Figure 15. 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. The cyclic handler is executed every 1 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, rot_rdq is implemented and the priority is passed as the argument.

5. Evaluation

To analyze the advantages and disadvantages of the proposed framework, the evaluations are performed as follows.

- the comparison of the size and transfer time between an mruby application including mruby libraries and not.
- the comparison of the overheads for each cyclic period of calling rot_rdq.
- the comparison of the application execution time with singletasking, co-routine and multitasking.

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 1.1.0.

6. Related work

The open-source run-time systems for scripting languages have been developed such as follow: mruby on TECS

TABLE 1. COMPARION OF THE PROPOSED AND PREVIOUS WORK

	mruby bytecode loader using Bluetooth	Preemptive-multitask (multi-VM)	Nonpreemptive-multitask (co-routine)	GIL
mruby on TECS		✓	✓	
mruby			✓	
Owl system		✓		✓
Lua		✓	✓	
Proposed framework	✓	✓	✓	

```
1  cell tCyclicHandler CyclicHandler{
2      ciBody = CyclicMain.eiBody;
3      attribute = C_EXP("TA_STA");
4      cyclicTime = 1;
5      cyclicPhase = 0;
6  };
7
8  cell tCyclicMain CyclicMain{
9      priority =
10         C_EXP("EV3_MRUBY_VM_PRIORITY");
11 };
```

Figure 17. Build Description of Cyclic Handler

[1], mruby [9], the Owl system [3], and Lua [6]

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 a RTOS to use multitasking. The co-routine is supported as same as mruby.

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.

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 []. python-on-a-chip supports multitask managed by GIL (Global Interpreter Lock). GIL locks a CPU in order to prevent the non-thread-safe code from sharing the other threads. GIL leads the limitation of the concurrency on multitasking.

Lua is one of the most popular script languages for embedded systems. Lua supports co-routines, 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.

Table 1 shows a comparison between the proposed framework and previous work.

7. Conclusion

References

[1] T. Azumi, Y. Nagahara, H. Oyama, and N. Nishio. mruby on TECS: Component-Based Framework for Running Script Program. In *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*, pages 252–259, April 2015.

[2] T. Azumi, M. Yamamoto, Y. Kominami, N. Takagi, H. Oyama, and H. Takada. A New Specification of Software Components for Embedded Systems. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*, pages 46–50, May 2007.

[3] Thomas W. Barr, Rebecca Smith, and Scott Rixner. Design and Implementation of an Embedded Python Run-Time System. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 297–308, Boston, MA, 2012. USENIX.

[4] Ivica Crnkovic. Component-based Software Engineering for Embedded Systems. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 712–713, New York, NY, USA, 2005. ACM.

[5] Chuck Forsberg. The ZMODEM Inter Application File Transfer Protocol, 1988.

[6] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The Evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 2–1–2–26, New York, NY, USA, 2007. ACM.

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

[8] A. Ohno, T. Azumi, and N. Nishio. TECS Components Providing Functionalities of OSEK Specification for ITRON OS. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1434–1441, June 2012.

[9] K. Tanaka, Y. Matsumoto, and H. Arimori. Embedded System Development by Lightweight Ruby. In *Computational Science and Its Applications (ICCSA), 2011 International Conference on*, pages 282–285, June 2011.