0 コンピュータソフトウェア

*†**†* *†*

*†**†*

1

**Component-Based Framework of Lightweight Ruby for Efficient Embedded Software Development**

Takuro Yamamoto Hiroshi Oyama Takuya Azumi

The complexity and scale of embedded software have increased. To improve the productivity, the mruby on TOPPERS embedded component system (TECS) framework, which employs a scripting language (i.e., lightweight Ruby) and supports component-based development, has been proposed. In the current mruby on TECS framework, mruby programs must be compiled and linked every time they are modified, because mruby bytecode are incorporated in the platform. Moreover, while the framework supports multiple vir­tual machines (VMs), developers must be familiar with the functions of real-time operating systems to effectively execute multiple mruby programs concurrently or in parallel. This paper proposed an extended mruby on TECS framework that improves development efficiency more than the current framework. We implemented a Bluetooth loader receives an mruby bytecode, and a RiteVM scheduler simplifies multitask­ing. Synchronization of initializing multiple tasks is also implemented using an Eventflag. Experimental results demonstrate the advantages of the proposed framework.

1 Introduction

Currently, embedded systems, e.g., Internet of Things applications, must demonstrate high qual­ity and high performance. This requirement has led to an increase in their complexity and scale; more­over, these systems need to have low production costs and short development cycles.

Complex and large-scale software systems can be developed efficiently by using component-based techniques [18], [17]. Component-Based Develop­ment (CBD) is a design technique that can be applied to reusable software development. Verifi­cation of component-based systems has been ex­tensively researched [20], [16]. Individual compo­nent diagrams enable the visualization of an en­tire system. In addition, component-based systems are flexible with regard to extensibility and speci-

軽M Ruby vol,, it t\*ntsc込 w,) 7 l•ウェアrygnJ /71-;-\*/l•ベ-A71/-/..,,7-,

it! 本 拓朗, ge 積 卓1, )CIN)CN )CN IQU礎 INTIA 4, Graduate School of Engineering Science, Osaka University.

)Cit! 博司, オ-,マ株式会社,OKUMA Corporation. J/ピュ-タソ7l•ウェア, Vol.0, No.0 (1983), pp.0–0. [V7l•ウェVeDZ] 0000 SEP 00 n 00 19\*1-4.

fication changes. The TOPPERS embedded com­ponent system (TECS) [14], AUTOSAR [1], and SaveCCM [23] are typical CBD tools for embedded systems.

In addition, scripting languages, such as Ruby, JavaScript, Perl, Python, and Lua, offer efficient approaches to software development. Currently, most software are programmed in C language. However, development in C language results in large code size, incurs high costs, and requires sig­nificant development time. In contrast, the use of scripting languages improves the efficiency of soft­ware engineering and can shorten the development period because it is relatively easy to reuse scripts.

For embedded systems, real-time properties, such as estimation of worst-case execution time, are very important. Although scripting languages are easy to use and read, their execution requires more time than that required by the codes written in C. Therefore, applying scripting languages to em­bedded systems is difficult.

To address the above limitation, “mruby on TECS,” a component-based framework for running script programs, has been proposed [12]. This framework integrates two technologies, i.e., mruby, which is a lightweight implementation of Ruby for embedded systems [27], [6], and TECS, which is

2 コ:/e°..m•-3zsi71-7,s7

a component-based framework for embedded sys­tems [14], [11].

Even though execution times of mruby on TECS are 100 times faster than those of mruby, it is not particularly efficient, at present, and imposes a heavy burden on developers. Moreover, mruby on TECS only supports a storage/ROM device for loading mruby programs. Consequently, if mruby programs are modified, a secure digital (SD) card must be inserted and removed repeatedly or ROM must be rewritten; moreover, developers need to restart real-time operating systems (RTOSs) on the target device. In addition, although mruby on TECS can support multiple virtual machines (multi-VMs), executing multiple tasks requires the developers to call the OS function.

This paper proposes an extended framework of mruby on TECS that comprises a Bluetooth loader for mruby bytecode and a RiteVM scheduler for fairly executing mruby programs. To improve de­velopment efficiency, in the proposed framework, developers need to implement the platform on a storage device only once at the beginning and can transfer mruby application programs from a host to a target device using the Bluetooth loader. Note that RiteVM is the Ruby VM specifically designed for embedded systems. The RiteVM scheduler manages the execution of multiple RiteVMs and al­lows developers to program multitasking more eas­ily than the current version of mruby on TECS.

**Contributions**: The proposed framework pro­vides the following contributions:

1. **Improved software development effi­ciency.** Developers do not need to rewrite a storage/ROM device and restart an RTOS. The Bluetooth loader supports continuous loading, which reduces Bluetooth set-up time (i.e., pairing).
2. **Execution of multiple mruby programs concurrently or in parallel.** Develop­ers can implement multiple tasks without RTOS knowledge because the RiteVM sched­uler switches tasks cyclically.
3. **Synchronized execution of multiple RiteVM tasks.** The proposed framework synchronizes multiple RiteVM tasks (i.e., mruby applica­tions).
4. **Benefits of CBD:** The paper focuses on the benefits of CBD and provides specific exam-

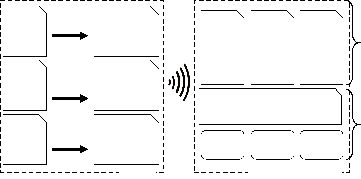
**2 Background**

Fig.1 shows the system model of the proposed framework. Note that the RiteVMs and the mruby library are assumed to be prepared in advance. Bytecodes are transferred from the host to the tar­get device via Bluetooth, and each RiteVM is allo­cated a bytecode. Bytecodes transferred from the host via Bluetooth can run in multitask.

**Fig. 1 System model of the proposed   
framework**

ples.

**Organization**: The reminder of this 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. Section 4 evaluates the proposed framework. Related work is discussed in Section 5. In Section 6, the utilization results are reported. Conclusions and suggestions for future work are presented in Section 7.



Load via Bluetooth Implement in advance

mruby   
Source   
File 1

(.rb)

mruby Compile r

mruby   
Bytecode 1   
(.mrb)

mruby Bytecode (.mrb)

mruby Bytecode (.mrb)

mruby Bytecode (.mrb)

mruby   
Source   
File 2

(.rb)

mruby   
Bytecode 2   
(.mrb)

1

2

3

mruby Compile r

mruby   
library   
(.mrb)

mruby   
Source   
File 3

(.rb)

mruby Compile r

mruby   
Bytecode 3   
(.mrb)

RiteVM

RiteVM

RiteVM

Target Device

Host

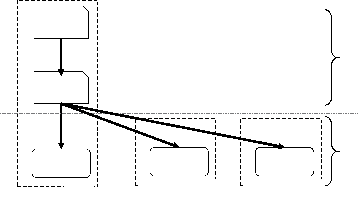
The proposed framework is an extension of mruby on the TECS framework [12], and utilizes two technologies: mruby and TECS. In this section, mruby (2. 1), TECS (2. 2), and mruby on the TECS framework (2.3) are described.

**2. 1 mruby**

mruby is a light-weight implementation of the Ruby programming language complying to part of the ISO standard. Ruby is an object-oriented scripting language [9] with classes and methods, exceptions, and garbage collection functions. It is easy to use and read due to its simple gram­mar and Ruby requires fewer lines of code than C. Ruby improves the productivity of software de­velopment due to its simple grammar and object-oriented functions.

Vol. 0 No. 0 1983 3

**Fig. 2 mruby/RiteVM mechanism**



mruby Compiler

mruby Source File

(.rb)

RiteVM

mruby   
Bytecode   
(.mrb)

Host

Target Device A

RiteVM

Target Device B

RiteVM

Development Environment Runtime Environment

**Fig. 3 Component Diagram**

entry port: eMotor

call port: cMotor

tMotor

Motor

signature: sMotor

tCaller

Caller



mruby, which retains the usability and readabil­ity of Ruby, requires fewer resources, and thus, is suitable for embedded systems. In addition, mruby includes a VM mechanism, and thus, mruby pro­grams can run on any operating system as long as a VM is implemented. The mruby/RiteVM mechanism is shown in Fig.2. The mruby compiler translates an mruby code into a bytecode, which can be interpreted by a RiteVM; thus, mruby pro­grams can be executed on any target device with a RiteVM.

**2.2 TECS**

TECS is a component system suitable for embed­ded systems. TECS can increase productivity and reduce development costs due to improved reusabil­ity of software components. TECS also provides component diagrams, which help developers visu­alize the overall structure of a system.

In TECS, component deployment and composi­tion are performed statically. Consequently, con­necting components does not incur significant over­head and memory requirements can be reduced. TECS can be implemented in C, and demonstrates various feature such as source level portability and fine-grained components.

**2. 2. 1 Component Model**

Fig.3 shows a component diagram. A *cell*, which is an instance of a component in TECS, consists of

 signature sMotor 

 int32 t getCounts( void );

 ER resetCounts( void );

 ER setPower( [in]int power );

 ER stop( [in] bool t brake );

 ER rotate( [in] int degrees,

 [in] uint32 t speed abs,

 [in] bool t blocking );

 void initializePort( [in]int32 t type );

;

**Fig. 4 Signature Description**

*entry* ports, *call* ports, attributes and internal vari­ables. An *entry* port is an interface that provides functions to other *cell*s, and a *call* port is an inter­face that enables the use of other *cell*’s functions. A *cell* has one or more *entry* ports and *call* ports. *Cell* functions are implemented in C.

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

**2. 2.2 Component Description**

In TECS, components are described by *signature*, *celltype*, and build written in component descrip­tion language (CDL). These components are de­scribed as follows.

**Signature Description**

The *signature* defines a *cell* interface. The *sig­nature* name follows the keyword *signature* and takes the prefix “s” e.g., sMotor (Fig.4). In TECS, to clarify the function of an interface, specifiers such as [in] and [out] are used, which represent input and output, respectively. **Celltype Description**

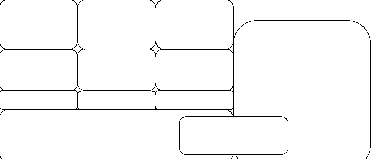
The *celltype* defines *entry* ports, *call* ports, at­tributes, and variables. A *celltype* name with the prefix “t” follows the keyword *celltype*, e.g., tCaller (Fig.5). To define *entry* ports, a *signa­ture*, e.g., sMotor, and an *entry* port name, e.g., eMotor, follow the keyword *entry*. *Call* ports are defined similarly. Attributes and variables follow the keywords *attr* and *var*, respectively.

**Build Description**

The build description is used to instantiate and

4 コンピュータソフトウェア

1 celltype tCaller {



mruby   
Program   
(bytecodes)

RiteVM

Task

mruby   
Program   
(bytecodes)

RiteVM

TECS

Task

mruby   
Program   
(bytecodes)

RiteVM

Task

mruby-TECS Bridge

Native / mruby Libraries

2 call sMotor cMotor;

3 };

4 celltype tMotor {

5 entry sMotor eMotor;

6 attr {

7 int32 t port;   
8};

9 var {

10int32 t currentSpeed = 0;

11 };

12 };

**Fig. 5 Celltype Description**

1 cell tMotor Motor {

2 port = C EXP(*”PORT A”*);

3 };

4 cell tCaller Caller {

5 cMotor = Motor.eMotor;

6 };

**Fig. 6 Build Description**

connect *cell*s. Fig.6 shows an example of a build description. A *celltype* name and *cell* name, e.g., tMotor and Motor, respectively, follow the keyword *cell*. To compose *cell*s, a *call* port, *cell*’s name, and an *entry* port are described in that order. In Fig.6, *entry* port eMotor in *cell* Motor is connected to *call* port cMotor in *cell* Caller. *C EXP* calls macros de­fined in C files.

**2.3 mruby on TECS**

mruby on TECS is a component-based frame­work for running an mruby script language on em­bedded systems. This framework integrates two technologies, mruby and TECS, and enables to develop embedded software using a script language without slowing down the execution time.

**2.3.1 System Model of mruby on TECS**

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

An mruby-TECS bridge provides native libraries for mruby and can call a native program (e.g., C legacy code) from an mruby program. The mruby-

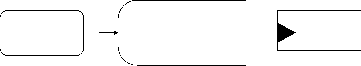
RTOS (TOPPERS/HRP2)

Target Hardware

**Fig. 7 System model of existing mruby on**

**TECS**

**Fig. 8 mruby-TECS bridge**



**tMotor**

**Motor**

**mruby code**

**TsMotor   
tsMotor**

**BridgeMotor**

TECS bridge also provides TECS components for receiving the invocation from an mruby program.

In this paper, TOPPERS/HRP2 [29], [22] is the target RTOS and is based on µITRON [26] 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 [25] and TOPPERS/ASP [13], [28].

**2. 3. 2 mruby-TECS Bridge**

There is a significant difference between the exe­cution times of mruby and C language codes. Ac­cording to [12], mruby programs are several hun­dred times slower than C programs and the execu­tion of an mruby bytecode on a RiteVM is not as efficient as that of C code. Thus, it is difficult to use mruby exclusively.

Using Ruby on embedded devices improves pro­ductivity and maintainability because it is easy to use and read. However, some C language codes are required to manipulate actuators and sensors and ensure that critical sections of the code run quickly.

Fig.8 illustrates an mruby-TECS bridge used to control a motor. The left side of BridgeMotor be­longs to the mruby program. The right side of BridgeMotor belongs to TECS component.

The mruby-TECS bridge generates a *celltype*, which is called from the mruby code, and an mruby class, which corresponds to a developer-specified

Vol. 0 No. 0 1983 5

mruby   
Application   
(.mrb)

**/RDG E\ %OXHWRRWK**

**=02'(0 3URWRFRO**

mruby   
Application   
(.mrb)

**/RDG E\ %OXHWRRWK**

**=02'(0 3URWRFRO**

mruby   
Application   
(.mrb)

**/RDG E\ %OXHWRRWK**

**=02'(0 3URWRFRO**

Host

**W7DVN**

**0UXE\7DVN**



**W5LWH90%OXHWRRWK**

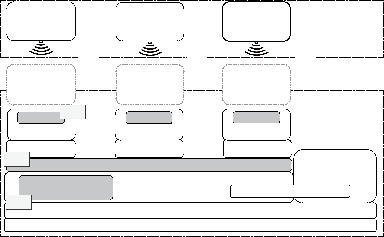
**5LWH90%OXHWRRWK**



**W6HULDO3RUW**

**6HULDO3RUW**

**Fig. 9 Detailed system model of the proposed framework**



3.3

3.2

mruby   
Application   
(.mrb)

RiteVM

RiteVM Scheduler

Loader

Task

3.1

Synchronization Mechanism

mruby   
Application   
(.mrb)

RiteVM

Loader

RTOS (TOPPERS/HRP2)

Task

TECS

Target Hardware

mruby   
Application   
(.mrb)

RiteVM

Loader

Task

mruby-TECS Bridge

Native / mruby Libraries

Target Device

TECS component to invoke a C function from the mruby program.

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

**3 Design and Implementation**

Fig.9 shows the detailed system model of the pro­posed framework. Each mruby application byte-code transferred from the host to the target de­vice is received by the loader in the RiteVM. The RiteVM reads the transferred bytecode and exe­cutes it with libraries. The mruby applications run simultaneously due to synchronized processing. The RiteVM scheduler switches RiteVM tasks be­cause multiple tasks can run concurrently. The fol­lowing subsection explains these functionalities.

**3.1 Bluetooth Loader for mruby Byte-code**

This section describes the proposed additional functionality of mruby on TECS, i.e., the Blue-tooth loader, for mruby bytecode.† In the current system, the platform including mruby bytecodes is saved on a storage/ROM device. Developers must

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

**Fig. 10 Component diagram of Bluetooth loader for mruby bytecode**

rewrite the storage/ROM device every time the programs are modified. In addition, the RTOS on the target device needs to be restarted. The repeti­tion hinders development efficiency. The Bluetooth loader for mruby bytecode decreases developer bur­den because developers only have to connect the storage/ROM device and start the RTOS once.

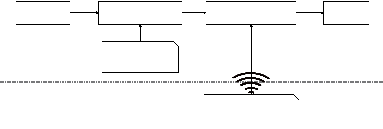
mruby programs consist of an mruby applica­tion and mruby libraries. An mruby application is the main program code, mruby libraries define the functions for the application, such as Ruby classes. The mruby bytecodes including both an mruby ap­plication and mruby libraries can be transferred and executed on the target device. However, this is also wasteful in terms of bytecode size and the time required to transfer the bytecodes, because the libraries are not modified frequently. With the proposed framework, only mruby applications are transferred, and the mruby libraries are preserved on the storage/ROM device beforehand. As a re­sult, RiteVMs can share mruby libraries. In addi­tion, a RiteVM can use its own library, which other RiteVMs should not use.

In the proposed framework, a platform that in­cludes RiteVMs and an mruby library is first com­piled and copied to the storage/ROM device. On the host, the mruby application programs (.rb files) are edited and compiled to bytecodes (.mrb files) by an mruby compiler. The generated bytecodes are transferred from the host to the target device via Bluetooth. This saves time since Bluetooth pair­ing can be avoided because the loader can load the bytecode continuously.

**3.1.1 RiteVM Component with Blue-tooth Loader for mruby Bytecode** The proposed framework provides a RiteVM with a Bluetooth loader for mruby bytecode as a TECS component. This component is an extension of the RiteVM component described in [12]. It receives bytecodes via Bluetooth and manages the RiteVM configuration, such as generating mruby library

6 *;/e°..n.---3zsi71-1,s7*

**Fig. 11 Process flow of Bluetooth loader for   
mruby bytecode**



Run tasks

Initialization

Read the bytecodes of mruby library

Read the bytecodes of mruby Application

Host

mruby Application (bytecodes)

A B

mruby library (bytecodes)

C D

Taget Device

1 */*\* *tRiteVMBluetooth.cdl* \**/*

2 celltype tRiteVMBluetooth1 {

3 entry sTaskBody eMrubyBody;

4 [optional] call sEventflag cEventflagü;

5 [optional] call sSemaphore cSemaphore;

6 attr {

7 [omit]char t \*mrubyLib;

8 char t \*irepLib =

9 C EXP(*”&*$*cell global*$ *irep”*);

10 uint32 t irepAppSize =

11 C EXP( BUFFER SIZE );

12 FLGPTN setptn;

13 };

14 var {

15 mrb state \*mrb;

16 mrbc context \*context;

17 [size is(irepAppSize)] uint8 t \*irepApp;

18 };

19 };

**Fig. 12 Celltype description for RiteVM with Bluetooth loader for mruby bytecode**

bytecodes automatically. This generated bytecode is prepared beforehand on the storage/ROM de­vice and differs from the bytecode transferred with Bluetooth.

Fig.10 shows a component diagram of Mruby-Task1 and RiteVMBluetooth1 *cell*s. The Mruby-Task1 *cell* is a componentized task of the RTOS (TOPPERS/HRP2 [29], [22]). The RiteVMBlue-tooth1 *cell* is the RiteVM component with the Blue-tooth loader for mruby bytecode. Bytecode on the host is transferred and received. In this framework, ZMODEM [19] is used as the binary transfer pro­tocol.

Fig.11 shows the process flow for executing an mruby program on the RiteVM component with the Bluetooth loader for mruby bytecode, such as RiteVMBluetooth1. The main tRiteVMBluetooth code is shown in Fig.13.

First, the Bluetooth loader receives the mruby application bytecode from the host (Fig.11(A);

1 */*\* *tRiteVMBluetooth.c* \**/*

2 void eMrubyBody main( CELLIDX idx )

3 {

4 */*\* *Omit: start of exclusive process by*

*semaphore* \**/*

5 */*\* *Receive the bytecode via Bluetooth* \**/*

6 bluetooth loader( VAR irepApp );

7 */*\* *Omit: end of exclusive process by semaphore* \**/*

8 */*\* *New interpreter instance* \**/*

9 VAR mrb = mrb open();

10 */*\* *Omit: error check for mrb state* \**/*

11 */*\* *New mruby context* \**/*

12 VAR context = mrbc context new(VAR mrb);

13 */*\* *Omit: initialize mruby*−*TECS bridge* \**/*

14 */*\* *Omit: synchronization of*

15 *initializing mruby application* \**/*

16 */*\* *Load mruby library bytecode* \**/*

17 mrb load irep cxt( VAR mrb,

18 ATTR irepLib,

19 VAR context );

20 */*\* *Load mruby application bytecode and run* \**/*

21 mrb load irep cxt( VAR mrb,

22 VAR irepApp,

23 VAR context );

24 if ( mrb−>exc ) {

25 */*\* *Failure to execute* \**/*

26 mrb p( VAR mrb,

27 mrb obj value( VAR mrb−>exc ) );

28 exit( 0 );

29 }

30 */*\* *Omit: synchronization of*

31 *terminating mruby application* \**/*

32 */*\* *Free mruby context* \**/*

33 mrbc context free( VAR mrb, VAR context );

34 */*\* *Free interpreter instance* \**/*

35 mrb close( VAR mrb );

36 }

**Fig. 13 Main code for RiteVM with Bluetooth loader for mruby bytecode**

lines 5-6 in Fig.13). The bytecode is stored in a component variable, such as *VAR irepApp*, as shown in Fig.12. This process is exclusively carried out by the semaphore to prevent the other loading from interrupting.

Second, *mrb state* and *mrbc context* pointers, and mruby-TECS bridges are initialized (Fig.11(B); lines 8-13 in Fig.13). *VAR mrb* and *VAR context* show the variables of the *cell*. *mrb state* is a set of states and global variables used in mruby. The syn­chronization of multiple tasks is performed in this processing phase. The RiteVM that finishes execu­tion at this point waits for the another RiteVM to finish loading and initialization.

Third, the RiteVM reads the bytecode of mruby libraries (Fig.11(C); lines 16-19 in Fig.13). The

Vol. 0 No. 0 1983 7

|  |  |
| --- | --- |
| **SULRULW\**  **6<67(0B35,25,7< +,\*+** | **rotateReadyQueue(RITEVM\_PRIORTY)** |

**6\VWHP7DVN**

**6\VWHP7DVN**

**7DVN 7DVN**

**SULRULW\**

**5,7(90B35,25,7< /2:**

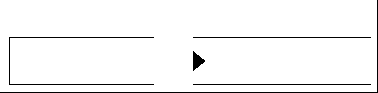
**7DVN**

**&\FOLF +DQGOHU ^**

**URWDWH5HDG\4XHXH5,7(90B35,25,7<**

**`**

**7DVN**



tRiteVMScheduler

RiteVMScheduler

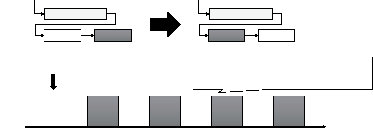
tCyclicHandler

CyclicHandler

tRiteVMSchedulerMain

RiteVMSchedulerMain

**Fig. 14 RiteVM scheduler design**



mruby libraries are a set of Ruby classes, such as the motor and sensor classes. For example, the mo­tor class defines methods to rotate and stop a mo­tor. The tRiteVMBluetooth *cell* has the attributes shown in Fig.12. *ATTR* indicates an attribute which is a fixed value that cannot be rewritten, un­like *VAR*. The *mrubyLib* indicates the program files of the mruby libraries, and is an attribute because mruby libraries are not modified in the proposed development process. Here, *[omit]* is only used for the TECS generator; thus, the attribute *mrubyLib* does not consume memory. *irepLib* is the pointer of the array in which the bytecode of mruby libraries is stored. To summarize, the bytecode of mruby libraries is stored as an attribute of the component during the first compilation.

Fourth, the RiteVM reads the bytecode of the mruby application transferred via Bluetooth (Fig.11(D); lines 20-23 in Fig.13). The mruby ap­plication bytecode is stored in an array of *irepApp*, which differs from the array that holds the mruby library bytecode. Note that two bytecodes are read separately in the RiteVM.

Finally, the mruby task runs (Fig.11(E); lines 20­23 in Fig.13). When an mruby application is mod­ified, only the bytecode of the modified application should be transferred; the mruby libraries do not need to be touched because they typically do not change.

The process shown in Fig.13 (lines 24-29) is car­ried out when an exception occurs. When all mruby applications are completed, mrb state and mrbc context are freed (lines of 32-35 in Fig.13). The proposed framework supports continuous load­ing; thus, this process loops. After the variables are freed, the RiteVM waits for the next mruby appli­cation bytecode.

**Fig. 15 Component diagram of RiteVM   
scheduler**

**3.2 RiteVM Scheduler**

This section describes the implementation of the RiteVM scheduler in the proposed framework. mruby on TECS supports multitasking; however, multitask processing in mruby on TECS requires the developers to have knowledge about the RTOS (TOPPERS/HRP2).

One approach for multitasking is a co-routine, which is a cooperative thread scheduled by devel­opers with functions such as *resume* and *yield* (the Ruby co-routine is defined in the Fiber class [3]). A co-routine is non-preemptive multitasking, which does not receive OS support because developers must switch tasks manually. A co-routine cannot take advantage of multicore processing.

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

For multitask processing, the proposed frame­work provides the RiteVM scheduler, which is a fair scheduler that runs multiple tasks equally. Note that the RiteVM scheduler is utilized only when application tasks have equal priority. mruby appli­cations can run concurrently without calling an OS function. The application programs can also utilize existing programs because their structures do not change.

The RiteVM scheduler is a fair scheduler that is implemented as a TECS component in the pro­posed framework. Therefore, when developers cre­ate software with priority-based scheduling, the RiteVM scheduler can be removed easily.

**3.2.1 RiteVM Scheduler Design**

A RiteVM scheduler is a periodic handler, and *rotateReadyQueue*, which is a service call of µITRON to switch tasks with equal prior-

8 コンピュータソフトウェア

ity, is implemented as the main process of the handler. In other words, the RiteVM scheduler calls *rotateReadyQueue* cyclically. The design of the RiteVM scheduler is shown in Fig.14. *ro-tateReadyQueue* is described as follows.

Here, assumed that two tasks with equal priority are in an infinite loop. In the current system, when one task is executed first, the other task would not be executed because the first task runs in the loop.

When *rotateReadyQueue* is called, tasks with equal priority are switched as shown in Fig.14. Note that the argument of *rotateReadyQueue* is the priority.

In addition, *rotateReadyQueue* can be performed if the number of tasks is more than two. For exam­ple, three tasks are in the order task1, task2, and task3. In this case, the order is rotated to task2, task3, and task1 when *rotateReadyQueue* is called.

**3. 2. 2 Component of RiteVM Scheduler**

Fig.15 shows a component diagram of the RiteVM scheduler. The RiteVM scheduler is a *composite cell* which consists of CyclicHandler and RiteVMSchedulerMain. The CyclicHandler *cell* configures the periodic handler based on µITRON. Cyclic handlers based on µITRON are described in the literature [26]. The CyclicHandler *cell* has the attributes of the *cell*. The RiteVMSchedulerMain *cell* processes the body of a periodic handler. Note that *rotateReadyQueue* is implemented as the body. Fig.16 shows tRiteVMScheduler *celltype*, which is a *composite cell* consisting of two *cell*s. The *call* port of RiteVMSchedulerMain is connected to the *entry* port of the Kernel *cell* (*tKernel.eiKernel*) to call kernel functions. The attribute is used as the *rotateReadyQueue* arguments.

Fig.17 shows the build description of the RiteVM scheduler. The RiteVMScheduler *cell* has at­tributes to configure the scheduler such as at­tribute, cyclicTime, cyclicPhase, and priority. In this case, the RiteVM scheduler is executed when it is generated, because the attribute is *TA STA*, which indicates that the periodic han­dler is in an operational state after creation. Note that the scheduler executes every 1 msec. RITEVM PRIORITY defines the priority of mruby tasks. In the function of RiteVMSchedulerMain, *rotateReadyQueue* is implemented and the priority is passed as the argument.

 */*\* *tRiteVMScheduler.cdl* \**/*

 celltype tCyclicHandler 

 [inline] entry sCyclic eCyclic;

 call siHandlerBody ciBody;

 attr 

 [omit] ATR attribute =

 C EXP(*”TA NULL*

*”*);

 [omit] RELTIM cyclicTime;

 [omit] RELTIM cyclicPhase;

 ;

;

 celltype tRiteVMSchedulerMain 

 require tKernel.eiKernel;

 entry siHandlerBody eiBody;

 attr 

 PRI priority;

 ;

;



 composite tRiteVMScheduler 

 attr 

 ATR attribute = C EXP(*”TA NULL”*);

 RELTIM cyclicTime = 1;

 RELTIM cyclicPhase = 1;

 PRI priority;

 ;

 cell tRiteVMSchedulerMain

RiteVMSchedulerMain 

 priority = composite.priority;

 ;

 cell tCyclicHandler CyclicHandler 

 ciBody = RiteVMSchedulerMain.   
eiBody;

 attribute = composite.attribute;

 cyclicTime = composite.cyclicTime;

 cyclicPhase = composite.cyclicPhase;

 ;

;

**Fig. 16 Celltype description of RiteVM   
scheduler**

 cell tRiteVMScheduler RiteVMScheduler 

 attribute = C EXP(*”TA STA”*);

 cyclicTime = 1;

 cyclicPhase = 1;

 priority =

 C EXP(*”RITEVM PRIORITY”*);

;

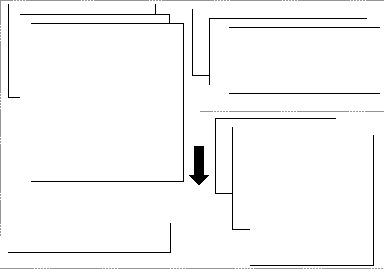
**Fig. 17 Build description of RiteVM scheduler**

**3. 3 Synchronization of Multiple RiteVM Tasks**

In the proposed framework, RiteVMs read mruby bytecodes and then execute applications. Eventflag is applied to synchronize the initiation of multiple mruby applications. Each task sets a flag pattern,

Vol. 0 No. 0 1983 9

**Fig. 18 Design of Eventflag using TECS (only differences are shown)**



**W5LWH90F**

**F(YHQWIODJBVHW $775BVHW3DWWHUQ**

**F(YHQWIODJBZDLW ZDLW3DWWHUQ $1' IOJSWQ**

**W5LWH90BF**

**HOVH`**

**LI 90W5LWH90BF**

**^**

**LIF(YHQWIODJBVHW**

**90W5LWH90BF ^ [**

**LIF(YHQWIODJBVHW HQWIODJBZDLW $775B,' [I**

**[**

**$1^**

**`**

**`**

**LI $775B,'**

**HOVH**

**F(YH` YQWIDJBZDLW**

**HOVHF(YHQWIODJBVHW**

**LI$775B,' [ ^**

**`**

**HOVH` HOVHF(YHQWIODJBVHW**

**`**

**`**

**HOVH LI $775B,' ^ F(YHQWIODJBVHW [ F(YHQWIODJBZDLW [I $1' IOJSWQ**

**`**

**YHQWIODJBZDLWF(YHQWIODJBVHW[I[$1**

**0F(YHQWIODJBZDLW**

**^ [I $1' IOJSWQ**

**[I [ $1' ^ YHQWIODJBZDLWF(YHQWIODJBVHW[I[$1' F(YHQWIODJBZDLW [I $1' IOJSWQ**

**YHQWIODJBZDLWF(YHQWIODJBVHW[I[$1' F(YHQWIODJBZDLW [I $1' IOJSWQ**

**W5LWH90BFGO**

**FHOO 0UXE\90W5LWH90BFGO90^**

**,'FHOO W0UXE\90W5LWH90BFGO90^ FH**

**,'FHOO W5LWH90 90 ^ 'FHOO W5LWH90 90 ^**

**0U**

**` 90^,' ,'**

**FHOO**

**' W0UX` `**

**,'FHOO W5LWH90 90 ^ FHOO W5LWH90 90 ^**

**,' ,'**

**` `**

**` FHOO**

**`**

**`**

**W5LWH90BFGO**

**FRQVW )/\*371W5LWH90BFGOZDLW3DWWHU**

**FHOO FRQVW0UXE\90 )/\*37190^W5LWH90BFGOZDLW3DWHU**

**FHOOVHW3DWWHUQFRQVWW0UXE\90 )/\*37190^**

**[ ZDLW3DWWHUQ [I**

**` FHOOHW3DWHUQW5LWH90 90[^**

**90^**

**VHW3DWWHUQ [**

**FHOO 0U**

**`**

**`**

**VHW3DWHFHOO W0U**

**`**

**`**

**FHOO W0U`**

**FHOOHW3DWHUQW5LWH90 90[**

**`**

**[^**

**FHOOHW3DWHUQW5LWH90 90**

**^**

**VHW3DWWHUQ [**

**`**

**FHOO W5LWH90 90 ^ VHW3DWWHUQ [ `**

**VHW3DWWHUQ [**



such as 0x01 (01) and 0x02 (10), and then waits for the flag pattern 0x03 (11) with AND. This process can also be applied to more tasks. For example, for four RiteVM tasks, each task sets a flag pattern, such as 0x01 (0001), 0x02 (0010), 0x04 (0100), and 0x08 (1000), and then waits 0x0f (1111) with AND, as shown in Fig.18(A).

In addition, the termination of mruby applica­tions is synchronized to accept continuous load­ing. This termination synchronization prevents a RiteVM whose application finishes immediately from waiting for the next loading. Thus, all mruby applications finish at the same time, and all RiteVMs wait to receive the next mruby appli­cation bytecodes.

**3. 4 Utilization of Component-Based De­velopment**

In the proposed framework, RiteVMs, the RiteVM scheduler, and Eventflags are implemented as components. Therefore, developers can add, re­move, or reuse these components easily. For exam­ple, if the RiteVM scheduler is not necessary for the software, developers should comment out only the CDL file, e.g., *//import(”tRiteVMScheduler.cdl”);*. CBD eliminates the need for developers to rewrite kernel configuration files.

In addition, the code size can be reduced by using CBD. In the proposed framework, this advantage is applied in the Eventflag component. The set pat­tern and wait pattern are defined as attributes of the component as shown in Fig.18 (B). This design, e.g., *cEventflag set(ATTR setPattern)*, enables the

**Table 1 Development procedure of existing   
and proposed frameworks**

Development procedure



Exisiting framework Proposed framework

(1) + (2) + (3)

(4)

1. Rewrite the storage device
2. Insert the storage device into the target
3. Restart the RTOS on the target (about 4 s)
4. Transfer an mruby bytecode from host to target

program without “if” statements and reuses an identical C file. Developers do not need to mod­ify the C file because the CDL files are prepared according to the number of RiteVMs. In addition, the Eventflag components are built with *[optional]* in TECS. Here, *[optional]* means that the code is run only when the call port is connected. Thus, the C file does not need to be rewritten even if the Eventflag is not used.

**4 Experimental Evaluation**

This section discusses experimental results. To analyze the advantages of the proposed framework, we evaluated the following.

* Execution time of the platform
* Size and time of transferred mruby bytecodes by the Bluetooth loader
* Execution time with singletasking, co-routine, and proposed multitasking
* Overhead for periodic time
* Synchronization of multiple RiteVM tasks
* Code size with CBD

These evaluations were performed to demon­strate that a Bluetooth loader can improve the effi­ciency of software development, that the proposed multitask processing executes effectively compared to singletasking or co-routine, and that the initi­ation of mruby applications are synchronized. In addition, we focused on benefits of CBD. We im­plemented the proposed system on a LEGO MIND-STORMS EV3 [24] (300MHz ARM9-based Sitara AM1808 system-on-a-chip) compiled with gcc 4.9.3 -O2 and mruby version 1.2.0.

**4. 1 Improving Software Development Ef­ficiency by Bluetooth Loader**

The Bluetooth loader for mruby bytecode can re­duce the development time. With the proposed

10 ;/e°..m--3zsi71,=7

**Table 2 Comparison of size and load process   
time between an mruby application with and   
without mruby libraries**

**640**

**620**

**600**

**580**

**560**

**540**

Execution Time [msec]

**520**

|  |  |  |  |
| --- | --- | --- | --- |
|  | App&Lib | App | App&Lib/App |
| Bytecode Size | 14,044 bytes | 199 bytes | ×70.6 |
| Loading Time | 305.081 msec | 7.774 msec | ×39.2 |
| Compile Time | 8.7 msec | 0.3 msec | ×29.0 |

framework, developers do not need to rewrite a storage/ROM device because only the bytecode should be transferred. In the proposed frame­work, the development procedure is shortened as shown Table 1. For example, with the existing sys­tem, when mruby programs are modified, develop­ers must remove an SD card from the target de­vice, connect the host PC, compile/link the plat­form, and reinsert the SD card in the target de­vice in an experimental environment such as LEGO MINDSTORMS EV3. In addition, the proposed framework eliminates the need to restart the target RTOS. Note that the time of restarting the RTOS, including initializing device drivers, is 4 s in case of the EV3 platform [24]

In the proposed framework, to further improve software development efficiency, developers trans­fer only the mruby application bytecode; mruby libraries are incorporated in the platform. The size, load process time, and compilation time for an mruby application with and without mruby li­brary are shown in Table 2. The overhead of load processing to load a zero byte bytecode is 50.933 msec. Similarly, compilation overhead to compile a zero byte program is 46.9 msec. The mruby appli­cation bytecode is smaller and faster than including the mruby libraries for all terms. The difference increases as the number of RiteVMs increases be­cause 50 msec of overhead is incurred per RiteVM. These advantages improve the efficiency of software development.

**4.2 RiteVM Scheduler**

A comparison of the application execution time with singletasking, co-routine, and multitasking is shown in Fig.19. A program with 100,000 loops was used as an mruby application for the evalua­tion of execution time. Here, the singletask pro­gram looped 100,000 times and the multitask and co-routine programs looped 50,000 times for each

**660**

**6LQJOH7DVN 0XOWL7DVN 6FKHGXOHU 0XOWL7DVN 'HOD\ &RURXWLQH**





**Fig. 19 Comparison of application execution time with singletask, co-routine, and multitask**

task.

**4.2.1 Execution Time with Singletask-ing, Co-routine, and Proposed Multitasking**

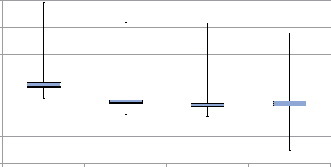
Fig.19 shows that the proposed design is supe­rior to multitask using delay method and co-routine in terms of execution time. The *delay* and co-routine methods run on a RiteVM, which causes the slowdown compared to the proposed multitask­ing. The scheduler runs as the periodic handler of RTOS. Moreover, developers can utilize the sched­uler practically because the RiteVM scheduler over­head is approximately 5%. The scheduler inter­rupts and switches tasks, which causes this over-head.Although the overhead is larger as the num­ber of RiteVMs increases, it also spreads the dif­ference between multitask and co-routine. There­fore, the scheduler is the more efficient approach for multitasking than the existing multitasking and co-routine. Here, the periodic time of the scheduler is 1 msec. It takes approximately 3 µsec on average to switch tasks once.

**4. 2.2 Periodic Time Overhead**

Fig.20 shows the execution time of multitasking with the periodic handler. The lower limit of the periodic time is 1 msec due to the specifications of TOPPERS/HRP2, i.e., the RTOS. The execution time decreases as the periodic time increases, be­cause the number of switched tasks decreases. Note that an execution time of 1 msec is approximately 1% greater than that of 8 msec. The RiteVM sched­uler with a short periodic time can execute multiple tasks effectively because the periodic time overhead is not large. Note that a smaller periodic time is better for multitasking due to concurrent or paral-

Vol. 0 No. 0 1983 11

600



590

Execution Time [msec]

580

570

560

550

540

1 msec 2 msec 4 msec 8 msec

Periodic Time

Fig. 20 Comparison of overhead for each   
cyclic period of RiteVM scheduler

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 3 2.5 2 1..5 1. 0.5  0 |  |  |  |  |  |
| Execution Time [msec] |  |  |  |  |  |
|  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |

VM2 VM3 VM4

Fig. 21 Synchronization of multiple RiteVM

tasks

lel processing.

**4.3 Synchronization of Multiple RiteVM Tasks**

To execute multiple mruby applications, a syn­chronization mechanism for RiteVM tasks is imple­mented in the proposed framework. We measured the time from the execution of the first RiteVM task to that of the last RiteVM task. It was confirmed that the time was within the theo­retical value. Theoretical value is

*periodic time ×* (*number of RiteVM tasks −* 1). Fig.21 shows the case of two to eight RiteVM tasks. Note that the periodic time is 1 ms. As shown in Fig. 21, the time is within the theoretical value, which indicates successful synchroniza­tion of multiple RiteVM tasks.

**4. 4 Benefits of Component-Based Devel­opment**

In the proposed framework, RiteVMs, the RiteVM scheduler, and Eventflags are implemented as TECS components. Developers can add or re-

Table 3 C and CDL file code for the number of RiteVMs

|  |  |  |  |
| --- | --- | --- | --- |
|  | (A) | (B) | Diff |
| C (Total) | 8a+134 | 130 | 8a+4 |
| C (Modification) | 10a−2 | 0 | 10a−2 |
| CDL | 18a+25 | 18a+25 | 0 |

a : the number of RiteVMs

move the functionalities easily by modifying the CDL file. Moreover, CBD decreases code size and improves productivity and maintainability.

To demonstrate the superiority of CBD, a com­parison of the number of lines of codes C and ACL codes between two situations is shown in Table 3. In Table 3, (A) and (B) represent the source files in the upper and lower parts of Fig.18, re­spectively. For C, (B)’s code lines do not increase even if the number of RiteVMs increases, while (A)’s code lines increase as the number of RiteVMs increases. Note that (B)’s C file can be utilized without modification regardless of the number of RiteVMs. Moreover, the number of code lines of two CDL files are equal. Skillful CBD yields advan­tages such as the decreased number of lines of codes and non-modified codes, which facilitates high pro­ductivity and maintainability.

**5 Related Work**

Open-source runtime systems for scripting lan­guages have been proposed previously such python-on-a-chip [8], the Owl system [15], eLua [4], ChaiS-cript [2], Squirrel [10], mruby [27], [6], and mruby on TECS [12].

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

**Owl system:** The Owl system is an embedded Python runtime system. It is a complete system for ARM Cortex-M3 microcontrollers. The Owl toolchain produces relocatable memory images that are directly runnable on the microcontroller from Python code objects. Note that the Owl system interpreter is the same as that of python-on-a-chip.

**eLua:** eLua (embedded Lua) offers a full imple­mentation of the Lua programming language for

12 コンピュータソフトウェア

Table 4 Comparison of proposed and previous methods

Call

C Function

Legacy Code of VM VM

Embedded System Management Scheduler

Synchronization of Applications

Bluetooth Loader

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| python-on-a-chip [8] |  |  |  |  |  |  |
| Owl system [15]  eLua [4] |  | V  V | Partially  Partially |  |  |  |
| ChaiScript [2] |  |  |  | - | - |  |
| Squirrel [10]  mruby [27] |  | V  V |  |  |  |  |
| mruby on TECS [12] |  | V | V | V |  |  |
| Proposed framework | V | V | V | V | V | V |

embedded systems. Lua is one of the most popular scripting languages for embedded systems [5], [21]. Lua supports a co-routine, which is referred to as cooperative multitasking. A co-routine in Lua is used as an independently executed thread. Note that a co-routine can only suspend and resume mul­tiple routines; thus, a Lua co-routine is not like multitasks in multitask systems.

**ChaiScript:** ChaiScript is a scripting language to be embedded in C++ applications. It is a dialect of ECMAScript (a.k.a. JavaScript). ChaiScript is header only, that is, it only needs to include the header files in order to be used in an application. ChaiScript will support co-routines, but not now.

**Squirrel:** Squirrel is an object-oriented pro­gramming language designed as a lightweight scripting language that satisfies the real-time re­quirements of applications. Squirrel was inspired by Lua. The Squirrel API is very similar to Lua and the table code is based on that of Lua; Squir­rel also supports co-routines.

**mruby:** mruby, a lightweight implementation of the Ruby language, has been proposed for em­bedded systems. mruby programs can run on a RiteVM, which is the VM for mruby and reads the mruby bytecode. Note that the RiteVM only sup­ports a single thread. In addition, mruby supports co-routines but does not support multitasking for RTOSs.

**mruby on TECS:** mruby on TECS is a component-based framework for running mruby programs. mruby programs on TECS can be ex­ecuted approximately 100 times faster than stan­dard mruby programs. In addition, software can be developed using CBD with mruby on TECS. Al­though multitasking has been supported in the cur­rent mruby on TECS, developers must be familiar

with the functions of the RTOS to use multitask­ing. Co-routines are also supported by mruby on TECS.

Table 4 compares the proposed framework to pre­vious methods. The proposed framework imple­ments all of the features shown in the table. In particular, the proposed framework supports the loader, the VM scheduler, and application synchro­nization. Note that the idea that a bytecode is transferred to the target device via Bluetooth can be applied to scripting languages other than mruby if the VM mechanism is used.

**6 Actual Results**

The proposed framework on LEGO MIND-STORMS EV3 [7] has been certified for the official platform of ET Robocon 2016†. In ET Robocon 2016, the team with the proposed framework stood second in a regional competition.

In addition, the proposed framework on LEGO MINDSTORMS EV3 has been commercialized for education by Afrel Co., Ltd†, which is an official partner of LEGO Education.

**7 Conclusion**

This paper presented an extended framework of mruby on TECS. In the proposed framework, we implement a Bluetooth loader for mruby bytecode, a RiteVM scheduler, and a synchronization mecha­nism. The Bluetooth loader improves the software de­velopment efficiency because of omitting the pro­cess to rewrite a storage/ROM device and restart an RTOS on the target device. The loader can use both Bluetooth and a wired serial connection; thus, the proposed framework can be applied to vari-

f2 <http://www.etrobo.jp/2016/> (in Japanese)

f3 <http://www.afrel.co.jp/lineup/textset> (in Japanese)

Vol. 0 No. 0 1983 13

ous embedded systems. The RiteVM scheduler makes multitasking more user-friendly compared to the current mruby on TECS. Our experimental re­sults show the advantages of the proposed frame­work. The Bluetooth loader can improve the software development efficiency of mruby on TECS, and the RiteVM scheduler is effective in terms of execution time and ease of use compared to singletasking and co-routines. In addition, synchronization of multi­ple RiteVM tasks is implemented in the proposed framework.

The proposed framework is developed using CBD. In addition, the RiteVMs, RiteVM sched­uler, and Eventflags are implemented as compo­nents; therefore, developers can add, remove, or reuse their functionalities easily as required. More­over, developers can choose fair scheduling or fixed-priority scheduling because the RiteVM scheduler can be added and removed easily. For software de­veloped with fixed-priority-based scheduling, devel­opers only have to remove the RiteVM scheduler. Note that our prototype system and the applica­tion programs used in the performance evaluation are all open source and can be downloaded from our website [7].

In the future, CDL files for the RiteVM and mruby-TECS bridge will be generated automati­cally using a plugin, and developers will be able to transfer bytecodes using the ZMODEM protocol on the command line. Moreover, we will support mruby libraries as mrbgems, which is an mruby dis­tribution packaging system.

**8 Acknowledgement**

This work was supported by JSPS KAKENHI Grant Number 15H05305. We would like to thank Takuya Ishikawa, Hiroshi Mimaki, and Kazuaki Tanaka for supporting this research.

**References**

[ 1 ] AUTOSAR. [http://www.autosar.org/](http://www.autosar.org/.).

[ 2 ] ChaiScript. [http://chaiscript.com/](http://chaiscript.com/.).

[ 3 ] class Fiber. <http://docs.ruby-lang.org/en/2>. 3.0/Fiber.html.

[ 4 ] eLua. <http://www.eluaproject.net>.

[ 5 ] Lua. [http://www.lua.org/](http://www.lua.org/.).

[ 6 ] mruby. [https://github.com/mruby/mruby](https://github.com/mruby/mruby.).

[ 7 ] mruby-on-ev3rt+tecs. http://www.toppers.jp/ tecs.html#mruby\_ev3rt.

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

[ 9 ] Ruby. [https://www.ruby-lang.org/en/](https://www.ruby-lang.org/en/.).

1. Squirrel. [http://www.squirrel-lang.org/](http://www.squirrel-lang.org/.).
2. TOPPERS Project. http://www.toppers.jp/ en/index.html.
3. 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 Computing (ISORC)*, pages 252–259, 2015.
4. T. Azumi, H. Takada, T. Ukai, and H. Oyama. Wheeled Inverted Pendulum with Embedded Com­ponent System: A Case Study. In *Proceed­ings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 151–155, 2010.
5. 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)*, pages 46– 50, 2007.
6. 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)*, pages 297–308, 2012.
7. B. Bonakdarpour and S. S. Kulkarni. Compo­sitional verification of fault-tolerant real-time pro­grams. In *Proceedings of the Seventh ACM Interna­tional Conference on Embedded Software*, EMSOFT ’09, pages 29–38, 2009.
8. X. Cai, M. R. Lyu, K.-F. Wong, and R. Ko. Component-based software engineering: technolo­gies, development frameworks, and quality assur­ance schemes. In *Processings of Seventh Asia-Pacific Software Engineering Conference (APSEC 2000)*, pages 372–379, 2000.
9. I. Crnkovic. Component-based Software Engi­neering for Embedded Systems. In *Proceedings of the 27th International Conference on Software En­gineering*, pages 712–713, 2005.
10. C. Forsberg. The ZMODEM Inter Application File Transfer Protocol. [http://pauillac.inria. fr/~doligez/zmodem/zmodem.txt](http://pauillac.inria.fr/~doligez/zmodem/zmodem.txt,), 1988.
11. G. Gössler and L. A¸stef˘anoaei. Blaming in component-based real-time systems. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT ’14, 2014.
12. 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*, pages 2–1–2–26, 2007.

14 コンピュータソフトウェア

1. 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)*, pages 1–8, 2013.
2. M. kerholm, J. Carlson, J. Fredriksson, H. Hans-son, J. H˚akansson, A. Möller, P. Pettersson, and

M. Tivoli. The SAVE Approach to Component-based Development of Vehicular Systems. *Journal of Systems and Software*, 80(5):655–667, 2007.

1. Y. Li, T. Ishikawa, Y. Matsubara, and H. Takada. A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support. *OSPERT 2014*, pages 51–59, 2014.
2. A. Ohno, T. Azumi, and N. Nishio. TECS Com­

ponents Providing Functionalities of OSEK Speci­fications for ITRON OS. *Journal of Information Processing*, 22(4):584–594, 2014.

1. H. Takada and K. Sakamura. µITRON for Small-Scale Embedded Systems. *IEEE Micro*, 15(6):46– 54, 1995.
2. K. Tanaka, A. D. Nagumanthri, and Y. Mat­sumoto. mruby – Rapid Software Development for Embedded Systems. In *Proceedings of the 15th In­ternational Conference on Computational Science and Its Applications (ICCSA)*, pages 27–32, 2015.
3. TOPPERS. TOPPERS/ASP kernel. https: //[www.toppers.jp/en/asp-kernel.html](http://www.toppers.jp/en/asp-kernel.html.).
4. TOPPERS. TOPPERS/HRP2 kernel. http: //[www.toppers.jp/en/hrp2-kernel.html](http://www.toppers.jp/en/hrp2-kernel.html.).