情報処理学会論文誌 Vol.57 No.1 1–11 (Jan. 2016)

Bluetoothを用いたマルチVM対応 mrubyバイトコードローダ

山本 拓朗[1,a)] 大山 博司[1] 安積 卓也[1,†1,b)]

受付日 2015年3月4日, 採録日 2015年8月1日

概要：近年，組込みシステムは複雑化・大規模化しているため，組込みソフトウェアの生産性が問題になっている．
組込みソフトウェア開発の生産性の向上を目的として，mruby（軽量 Ruby）を適用させたコンポーネントベース開発が可能なフレームワークである mruby on TECS を提案してきた．
現状の mruby on TECS では，プラットフォームに mruby バイトコードを組み込んでいるため，mruby プログラムを修正する度にコンパイル・リンクし直す必要がある．さらに，マルチ VM を提供しているが，複数の mruby プログラムを効率よく並行動作させるには開発者がリアルタイム OS の機能を熟知している必要がある．
本研究では，mruby on TECS の拡張として，mruby アプリケーションのバイトコードを Bluetooth で転送することで開発効率を向上させる．さらに，複数の mruby プログラムを協調動作できるフレームワークを提案する．

キーワード：情報処理学会論文誌ジャーナル，LaTeX，スタイルファイル，べからず集

mruby Bytecode Loader Using Bluetooth in Multi-VM Environment

Takuro Yamamoto[1,a)] Hiroshi Oyama[1] Takuya Azumi[1,†1,b)]

Received: March 4, 2015, Accepted: August 1, 2015

Abstract: This document is a guide to prepare a draft for submitting to IPSJ Journal, and the final camera-ready manuscript of a paper to appear in IPSJ Journal, using LaTeX and special style files. Since this document itself is produced with the style files, it will help you to refer its source file which is distributed with the style files.

Keywords: IPSJ Journal, LaTeX, style files, “Dos and Don’ts” list

1. はじめに

近年，組込みシステムは高品質・高性能化に伴い，複雑化・大規模化している．例えば，IoT アプリケーションがある．その上，製品の低コスト化や短期間での開発も要求されている．

ソフトウェアを効率的に開発するアプローチのひとつに，コンポーネントベース開発がある．コンポーネントベース開発は，再利用可能なコンポーネントを構成するための設計手法である [?]．コンポーネント化されたソフトウェアは，再利用性が高く，検証が容易になるため，コンポーネントベース開発を適用することで，複雑かつ大規模なソフトウェアを効率的に開発することができる．システムの拡張や仕様変更にも柔軟に対応できる．組込みシステムのコンポーネント技術として，TECS [?] や AUTOSAR [?]，SaveCCM [?] がある．

効率的なソフトウェア開発のもう一つのアプローチとし

[1] 情報処理学会
IPSJ, Chiyoda, Tokyo 101–0062, Japan
[†1] 現在，情報処理大学
Presently with Johoshori University
[a)] joho.taro@ipsj.or.jp
[b)] gakkai.jiro@ipsj.or.jp

て，スクリプト言語がある．現在のソフトウェアのほとんどは C 言語で開発されているが，C 言語による開発はコストが高く，開発期間も長くなる．スクリプト言語は，その使いやすさから高い生産性をもっているため，効率的かつ短期間での開発ができる．有名なスクリプト言語として，Java script，Perl，Python，Lua，Ruby などがある．

しかし，スクリプト言語は可読性や使いやすさが高い反面，C 言語よりも実行時間が遅い．組込みシステムにとって，最悪実行時間や応答時間といったリアルタイム性は非常に重要であるため，スクリプト言語を組込みシステムに利用することは難しい．

mruby on TECS は，mruby(軽量 Ruby) と，組込みシステムに適したコンポーネントシステムである TECS (TOPPERS Embedded Component System) を組み合わせたフレームワークである [**?**]．mruby on TECS では，mruby プログラムから C 言語の関数を呼ぶ機能を提供しており，mruby に比べて，アプリケーションを約 100 倍速く実行できる．

現状の mruby on TECS では，いくつかの問題がある．mruby on TECS は，mruby プログラムをロードするための方法として，記憶装置しか対応していないため，作業効率が悪い．(LEGO MINDSTORMS EV3 のプラットフォーム [**?**]) mruby プログラムを修正する度に，SD カードの抜き差しや ROM への書き込みを行う必要がある．さらに，mruby on TECS はマルチ VM に対応してるが，複数の mruby アプリケーションを並行実行させる場合，開発者がタスクを待ち状態へ遷移させる OS の機能を呼び出さなければならない．

提案フレームワークは，mruby on TECS を拡張して，Bluetooth を用いた mruby バイトコードローダと，実用的なマルチタスク処理の実装を行った．提案フレームワークでは，プラットフォーム部分をコンパイル・リンクし，ターゲットデバイス上で起動する．ホスト PC 側では，mruby アプリケーション (.rb) をバイトコード (.mrb) にコンパイルし，Bluetooth を通してターゲットデバイスにバイトコードを転送する．ターゲットデバイス側では，RiteVM に実装されたローダが転送されたバイトコードを受信し，すでにプラットフォームに含まれている mruby ライブラリと合わせてアプリケーションを実行する．これによって，繰り返し SD カードの抜き差しや ROM への書き込みをする手間や OS を再起動する時間が省けるため，作業効率を上げることができる．さらに，各 VM の処理を平等に実行する RiteVM スケジューラを提供することで，マルチタスク処理を効率的に利用できる．

2. 既存フレームワーク

図 1 に提案フレームワークのシステムモデルを示す．RiteVM と mruby ライブラリを含むプラットフォーム部分は，はじめにターゲットデバイス上で起動されている．mruby アプリケーションのバイトコードは，ホストからターゲットデバイスへと転送される．それぞれバイトコードは，RiteVM に割り当てられ，並行して起動される．

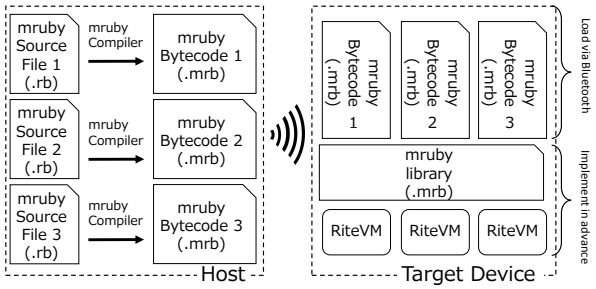


図 1 System Model

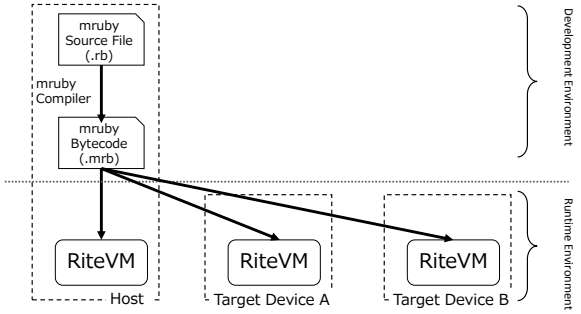


図 2 Mechanism of mruby/RiteVM

この章では，提案フレームワークのベースになっている mruby on TECS について述べる．mruby on TECS で利用されている mruby と TECS についても述べる．

2.1 mruby

mruby は，ISO 規格の一部に準拠した Ruby の軽量実装である．Ruby [**?**] は，オブジェクト指向型のスクリプト言語である．特徴としてシンプルな文法なため読みやすくかつ使いやすい上に，C 言語よりも少ないコード量で記述できる．他にも，クラスやメソッドといったオブジェクト指向関数，例外，ガベージコレクションなどの機能により，生産性を向上させることができる．

mruby は，Ruby の可読性や使いやすさはそのままに，リソースが少なく起動が速いため，組込みシステムに適している．mruby には，RiteVM と呼ばれる VM が実装されているため，どの OS 上でもアプリケーションを起動させることができる．図 2 に RiteVM のメカニズムを示す．mruby コンパイラは，mruby プログラム (.rb) を，中間言語であるバイトコード (.mrb) にコンパイルする．RiteVM が実装されいれば，どのターゲットデバイス上でも同じ mruby プログラムを実行できる．

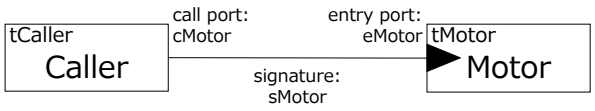


図 3 Component Diagram

2.2 TECS

TECS (TOPPERS Embedded Component System) は，組込みシステム向けのコンポーネントシステムである．TECS によるコンポーネントベース開発は，システム全体の構造を可視化するため，システムの複雑さや難しさを減らすことができる．さらに，ソフトウェアの共通部分はコンポーネントとして扱われるため，開発の重複を減らし，生産性を向上させる．

TECS でのコンポーネントの生成と結合は静的に行われるため，最適化され，実行時間や消費メモリのオーバヘッドは少ない．他の特徴として，C 言語での実装，ソースレベルでの移植性，コンポーネントの粒度が小さいことなどがある．

2.2.1 Component Model

図 3 にコンポーネント図の例を示す．*cell* which is an instance of a component in TECS consists of *entry* ports, *call* ports, attributes and internal variables.

A *entry* port is an interface to provide functions with other *cell*s, and a *call* port is an interface to use functions of other *cell*s. A *cell* has one or more *entry* ports and *call* ports. Functions of a *cell* are implemented in the C language.

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

2.2.2 Component Description

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

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.

```
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  };
```

図 4 Signature Description

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

図 5 Celltype Description

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

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

図 6 Build Description

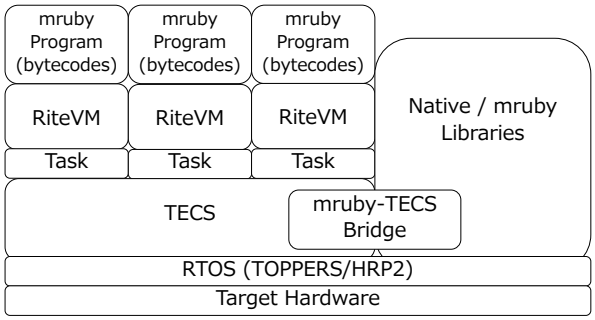


図 7 System Model of existing mruby on TECS

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 [**?**], [**?**] is used in this paper. TOPPERS/HRP2 is an RTOS based on μITRON [**?**] with memory protection. However, mruby on TECS does not depend on the RTOS because TECS supports not only TOPPERS/HRP2 but also the other RTOSs such as OSEK [**?**] and TOPPERS/ASP [**?**].

2.3.2 mruby-TECS Bridge

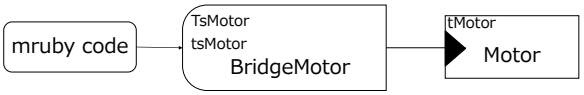


図 8 mruby-TECS Bridge

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

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

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

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

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. Design and Implementation

図 9 に提案フレームワークの詳細なシステムモデルを示す．ホストから転送された mruby アプリケーションのバイトコードは，それぞれの RiteVM に実装されたローダで受信され，同期処理により同時に実行される．RiteVM スケジューラがタスクを周期的に切り替えるため，mruby アプリケーションは並行動作することができる． Figure 9 shows the detailed system model of the proposed framework. Each mruby application bytecode transferred from the host is received by the loader in the RiteVM. The RiteVM reads the own bytecode and executes it. The mruby applications run at the same time because of syn-

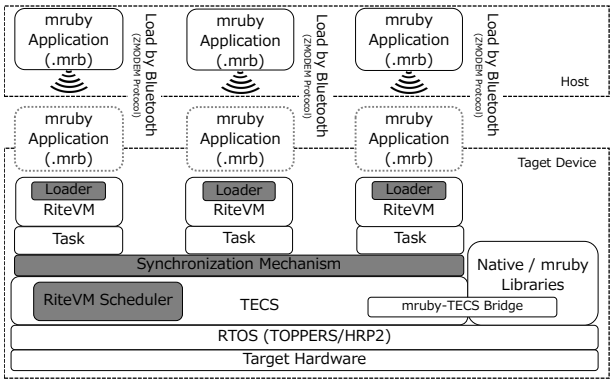


図 9 Detailed System Model of the proposed framework

chronization processing. The RiteVM scheuler switches tasks as multiple tasks can run in concurrent.

3.1 mruby Bytecode Loader Using Bluetooth

この章では，Bluetooth を用いた mruby バイトコードローダについて述べる．現状の mruby on TECS では，mruby バイトコードをプラットフォームに組み込んでいるため，mruby プログラムを修正する度にプラットフォーム部分をもう一度コンパイル・リンクし，SD カードや ROM といったデバイスに再度書き込み，OS を再起動する必要がある．このような作業を繰り返すことで，作業効率は悪くなる．提案フレームワークでは，プラットフォーム部分をコンパイル・リンクし記憶デバイスに書き込むのは，はじめの一度だけで良いため，作業効率を向上させることができる．

mruby プログラムは，アプリケーション部分とライブラリ部分に分けられる．mruby アプリケーションは，開発者が主にプログラムするメインのコードであり，mruby ライブラリには，motor クラスや sensor クラスといった Ruby クラスのようにアプリケーションで利用される関数群が定義されている．アプリケーションとライブラリを含んだ mruby バイトコードを転送し，実行することもできるが，提案フレームワークでは，ライブラリ部分はプラットフォームに組込み，アプリケーション部分のみを転送する設計にした．このような設計にすることで，ライブラリは頻繁に変更されないため，毎回コンパイルする無駄を省くことができる．さらに，各 RiteVM でライブラリを共有することや，RiteVM 固有のライブラリを持つことも可能になる．

提案フレームワークでは，mruby バイトコードの連続ローディングにも対応している．

3.1.1 Component of RiteVM with mruby bytecode loader using Bluetooth

提案フレームワークでは，mruby バイトコードローダは TECS のコンポーネントとして提供されている．このコンポーネントは，RiteVM のコンポーネントの拡張として実装されている．(詳細については [**?**] を参照) このコンポーネントでは，転送されたバイトコードの受信に加え，RiteVM のコンフィグレーションが行われる．

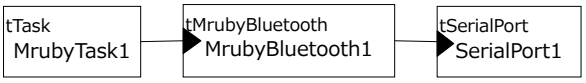


図 10 Component Diagram of mruby bytecode loader using Bluetooth

図 10 に，MrubyTask1 と MrubyBluetooth1 のコンポーネントの例を示す．MrubyTask1 は，コンポーネント化された RTOS (TOPPERS/HRP2 **????**) のタスクである．MrubyBluetooth1 は，mruby バイトコードを実装した RiteVM のコンポーネントであり，このコンポーネントのはじめで転送されてきたバイトコードを受信する．提案フレームワークでは，バイナリ転送プロトコルとして，ZMODE を適用した．

図 11 に，ローダを実装した RiteVM が mruby プログラムを実行するまでの制御フロー，図 13 には，RiteVM コンポーネントのソースコードを示す．

はじめに，*mrb_state* と *mrbc_context* のポインタを初期化する．*mrb_state* は，mruby で使われる状態と変数のセットである．

次に，mruby ライブラリのバイトコードを読み込む．図 12 に示すように，tMrubyBluetooth のセルは属性を持っている．*mrubyFile* は mruby ライブラリのファイルを示しており，*[omit]* は TECS ジェネレータによってのみ使えわれるため，この属性はメモリを消費することない．*irep* は，mruby ライブラリのバイトコードが格納されている配列へのポインタである．つまり，mruby ライブラリは始めのコンパイル時にコンポーネントの属性として，配列に保存されている．

次に，転送されてきた mruby アプリケーションのバイトコードを読み込む．mruby アプリケーションのバイトコードは，図 12 に示す，内部変数 uint8_t 型の配列 *irepApp* に格納されている．２つのバイトコードは，それぞれ別の配列に保存されており，別々に読み込まれる．

最後に，RiteVM は mruby アプリケーションであるタスクを実行する．mruby アプリケーションのプログラムが修正された場合は，そのバイトコードのみを再転送する．

3.2 Multitask

この章では，提案フレームワークでのマルチタスク処理の設計にちて述べる．mruby on TECS は，マルチタスクに対応しているが，開発者が RTOS の知識を熟知している必要がある．

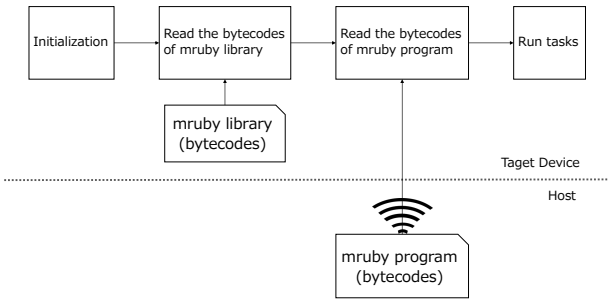


図 11 Control Flow of mruby bytecode loader using Bluetooth

```
1  celltype tMrubyBluetooth{
2      entry sTaskBody eMrubyBody;
3      attr{
4        [omit]char_t *mrubyFile;
5        char_t *irep = C_EXP("&$cell_global$_irep");
6        uint32_t irepAppSize = C_EXP( BUFFER_SIZE
             );
7      };
8      var{
9          [size_is(irepAppSize)] uint8_t *irepApp;
10     };
11 };
```

図 12 Celltype Description for RiteVM with mruby bytecode loader using Bluetooth

マルチタスク処理のアプローチのひとつにコルーチンがある．コルーチンは，協調的スレッドであり，開発者が *resume* や *yield* といった関数を呼び出すことで並列動作を行うことができる．(Ruby のコルーチンは，Fiber クラスに定義されている．[**?**]) コルーチンは，ノンプリエンプティブな処理で，開発者自身がタスクの切り替えを行う必要があるので，OS のサポートやマルチコアの恩恵を受けることができない．

その他のアプローチとして，μ ITRON のサービスコールである *delay()* 使った手法がある．このサービスコールは，引数として与えられた時間だけ，そのタスクの起動を遅らせる．*delay()* は，固定優先度スケジューリングの場合に用いられるが，フェアスケジューリングの場合には使用することが難しい．

提案フレームワークでは，フェアスケジューラである RiteVM スケジューラを実装し，複数のタスクを平等に並行動作させる．このスケジューラはアプリケーションタスクが同じ優先度を持つ場合に利用でき，開発者が OS の関数を呼び出すことなく，マルチタスク処理が可能になる．その上，既存のアプリケーションプログラムの構造を変えること使うことができる．

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

図 13 Main code for RiteVM with mruby bytecode loader using Bluetooth

3.2.1 RiteVM Scheduler

RiteVM スケジューラは，周期ハンドラであり，μ ITRON のサービスコールである *rotateReadyQueue* を呼び出す．*rotateReadyQueue* は，同優先度のタスクの実行を切り替える関数である．RiteVM スケジューラの設計を図 14 に示す．

無限ループを持った同優先度の 2 つのタスクの場合を考える．現状のシステムでは，始めにタスクが起動すると，そのタスクが無限ループに入るため，もう片方のタスクが起動することができない．

rotateReadyQueue が呼ばれると，図 14 に示すように，タスクの起動が切り替わる．*rotateReadyQueue* の引数は，優先度である．

rotateReadyQueue は，2 つ以上のタスクの場合にでも利用できる．例えば，3 つのタスク，task 1, 2, 3 が順に起動する場合，*rotateReadyQueue* が呼ばれると，task 2, 3, 1 の順にタスクの起動は切り替わる．

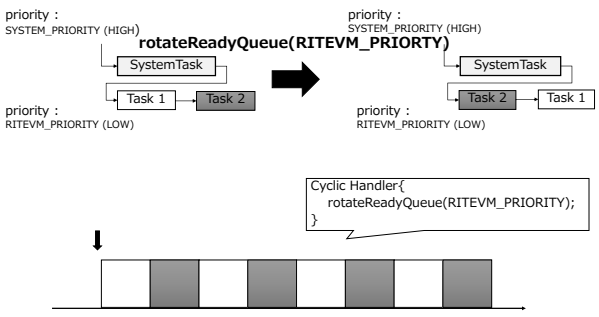


図 14 The design of RiteVM sceduler

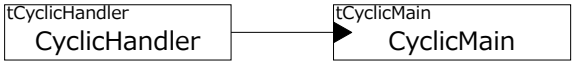


図 15 Component Diagram of Cyclic Handler

3.2.2 Component of RiteVM Scheduler

図 15 に RiteVM スケジューラのコンポーネント図を示す．RiteVM スケジューラのコンポーネントは，CyclicHandler と CyclicMain から構成される．CyclicHandler セルは，μ ITRON の周期ハンドラの設定を行う．μ ITRON の周期ハンドラは，[?] に詳しく書かれている．

周期ハンドラは，5 つの引数を持っている．(ID, attribute, cyclic time, cyclic phase, access pattern) CyclicHandler セルは，3 つの引数をセルの属性として持っている．CyclicMain セルは，周期ハンドラの処理を行うコンポーネントであり，*rotateReadyQueue* が実装されている．図 16 に，セルタイプ tCyclicMain のセルタイプ記述を示す．*call* ポートは，カーネルの機能を使用するためにカーネルセルの *entry* ポート (*tkernel.eiKernel*) に接続されている．属性は，*rotateReadyQueue* の引数として使われる．

図 17 に，図 15 に示されるコンポーネントの組み上げ記述を示す．

CyclicHandler セル部分では，cyclicTime や cyclicPhase といった属性で与えられる周期ハンドラの設定を行う．attribute は *TA_STA* であるので，OS が起動すると周期ハンドラが実行状態となる．周期時間は，1msec である．CyclicMain セル部分には，属性として priority が記述されている．RITEVM_PRIORITY は mruby タスクの優先度として定義されている．この属性は，*rotateReadyQueue* の引数として与えられる．

3.2.3 Synchronization of Multiple RiteVM Tasks

複数の mruby アプリケーションの起動を同期させるために，提案フレームワークでは，同期処理の手法としてイベントフラグを適用した．タスクはそれぞれ 0x01(01), 0x02(10) のパターンをセットし，待ちパターン 0x3(11) で

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

図 16 Celltype Description of Cyclic Handler

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

図 17 Build Description of Cyclic Handler

AND 待ちする．この機構は，2 つ以上のタスクの場合でも適用できる．例えば，タスクが 4 つの場合，タスクはそれぞれ，0x01(0001), 0x02(0010), 0x04 (0100), 0x08 (1000) のパターンをセットし，待ちパターン 0x0f(1111) で AND 待ちする．(図 18 (A))

さらに，提案フレームワークでは，バイトコードの連続ローディングに対応するため，アプリケーションの終了も同期を行う．これによって，すぐに終了するようなアプリケーションの RiteVM が，次のロードの待機状態に入るのを防ぐことができる．すべてのアプリケーションは同時に終了し，すべての RiteVM は同時に次のロードの待機状態へと入る．

3.3 Benefits of Component-Based Development

この章では，コンポーネントベース開発の利点を使った設計について述べる．

提案フレームワークでは，RiteVM や RiteVM スケジュー

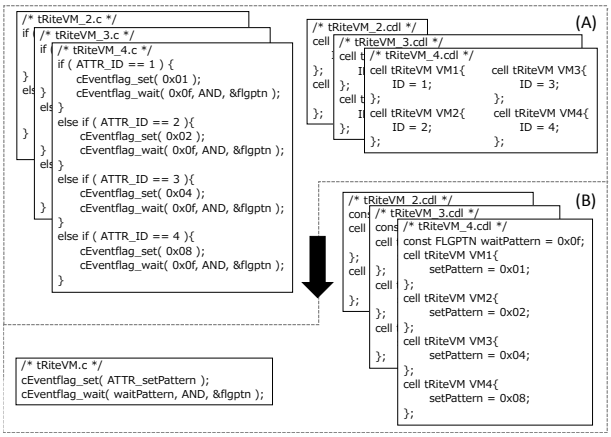


図 18 The design for Eventflag using TECS *only differences

ラ，イベントフラグはコンポーネントとして提供されているため，開発者はそれらのコンポーネントを容易に付け外し，再利用することができる．例えば，RiteVM スケジューラの機能を外したい場合，図 16，17 に示される cdl ファイルを *//import(<tRiteVMScheduler.cdl>);* のようにコメントアウトすることで容易に実現できる．開発者は，カーネルのコンフィグレーションファイルを修正する手間を省くことができる．

それに加えて，コンポーネントベース開発ではコード量を減らすことができる．提案フレームワークでは，イベントフラグにこの利点は見られる．図 18 に示されるように，イベントフラグのセットパターンと待ちパターンを属性として定義する．*cEventflag_set(ATTR_setPattern)* に見られるこの設計によって，if 文なしでプログラムを記述でき，RiteVM の数に関わらず同じ C ファイルを使用できる．さらに，TECS のオプションである *[optional]* によって，呼び口が結合した場合のみ処理を行うため，イベントフラグの結合を切った場合にでも同じ C ファイルを利用することができる．

4. Experimental Evaluation

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

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

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

表 1 Comparison of the size and load process time between an mruby application including mruby libraries and not

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

The size, load process time, and compilation time for mruby application and mruby application including library is shown in Table 1. The overhead of load processing is 50.933 msec, which it takes to load a zero bytes bytecode. Similarly, the overhead of compilation is 46.9 msec, which it takes to compile a zero bytes program. The mruby application bytecode is smaler and faster than that of including mruby libraries in all terms. The difference becomes larger as the number of RiteVMs increases because it takes the 50 msec overhead per one RiteVM. In addition, the design can save the time to connect a storage/ROM device to restart an OS. These advantages lead improvement of the software development efficiency.

The comparison of the application execution time with singletasking, co-routine, and multitasking is shown in Figure 19. 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 19, the cyclic time of the cyclic handler for multitasking is one msec. While multitask's execution time is slower than singletask's, co-routine takes more time than multitask to execute an mruby application. This result shows the proposed design is superior to co-routine in terms of execution time. The overhead of multitask is about 5 % $((multitask - singletask)/multitask)$. Switching tasks' overhead is also evaluated, that is about three μsec on average. The scheuler interrupts and switches tasks, which causes this overhead.

Figure 20 shows the execution time of multitasking with the cyclic handler. A lower limit of the cyclic time is one msec due to the specification of TOPPERS/HRP2, the used RTOS. More than 10 msec do not be evaluated in this paper because it is thought the larger cyclic time influences applications. Each execution time of cyclic period 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. This results also shows the overhead becomes smaller as the cyclic time is larger, because

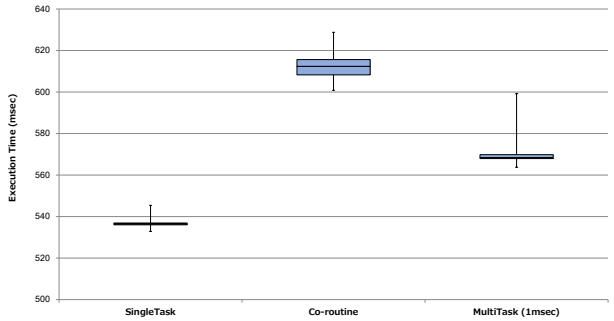


図 19 Comparison of the application execution time with singletask, co-routine, and multitask

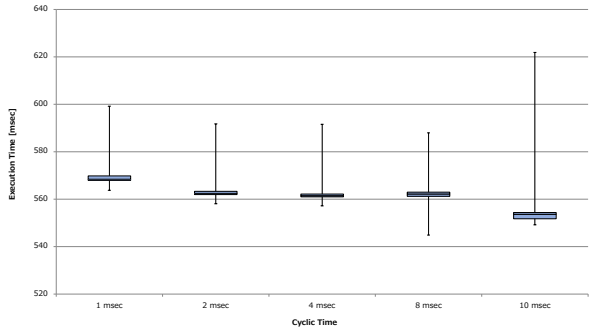


図 20 Comparison of the overhead for each cyclic period of calling rotateReadyQueue

表 2 Lines of .c and .cdl file for the number of RiteVM

	(A)	(B)	Diff
.c	134+8×α	130	4+8×α
.cdl	25+18×α	25+18×α	0

α : the number of RiteVM

the overhead depends on the number of switching tasks. The smaller cyclic time is better in multitasking due to concurrent and/or parallel processing.

To indicate the superior of component-based development, the comparison of code lines between two .c files is shown in Table 2. (A) and (B) mean the source files in the upper and lower of Figure 18, respectively. In terms of .c, (B)'s lines do not increase even if the number of RiteVMs increases, while (A)'s lines increase in proportion to the number. (B) is not modified, and can be utilized regardless of the number of RiteVMs. Moreover, lines of two .cdl files are equal. The skillfull component-based development brings this advantage such as the decrease of code lines, which leads high productivity and high maintainability.

5. Related Work

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

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

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

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

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

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

Table 3 shows a comparison between the proposed framework and previous work. The proposed framework supports the loader, the VM scheuler, and synchronization of application.

6. Conclusion

This paper has presented an extended framework of mruby on TECS: mruby bytecode loader using Bluetooth and RiteVM scheuler. The loader provides developers with the software development efficiency without rewriting a storage/ROM device and restarting an OS. The proposed framework can be applied to many kinds of embed-

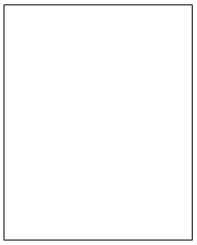
表 3 Comparion of the proposed and previous work

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

ded systems because the loader can use not only Bluetooth also wired serial connection. The RiteVM scheuler makes multitasking more easily than the current mruby on TECS. In the evaluation, experimental results of the loader and the RiteVM scheduler show their advantages. The loader can improve the software development efficiency on mruby on TECS. The RiteVM sceduler has the effectiveness in terms of execution time and ease of use compared with singletasking and co-routine.

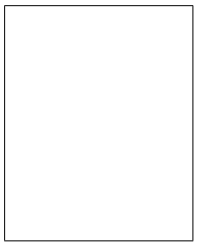
The proposed framework is developed in component-base by TECS. The facilities such as RiteVMs, the RiteVM scheuler, and Eventflag are implemented as components. Therefore, developers can easily add or remove the functionalities as necessary, and also reuse them. Developers can choose fair scheduling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

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



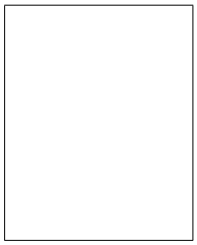
情報 太郎 （正会員）

1970 年生．1992 年情報処理大学理学部情報科学科卒業．1994 年同大学大学院修士課程修了．同年情報処理学会入社．オンライン出版の研究に従事．電子情報通信学会，IEEE，ACM 各会員．本会シニア会員．



処理 花子

1960 年生．1982 年情報処理大学理学部情報科学科卒業．1984 年同大学大学院修士課程修了．1987 年同博士課程修了．理学博士．1987 年情報処理大学助手．1992 年架空大学助教授．1997 年同大教授．オンライン出版の研究に従事．2010 年情報処理記念賞受賞．電子情報通信学会，IEEE，IEEE-CS，ACM 各会員．



学会 次郎 （名誉会員）

1950 年生．1974 年架空大学大学院修士課程修了．1987 年同博士課程修了．工学博士．1977 年架空大学助手．1992 年情報処理大学助教授．1987 年同大教授．2000 年から情報処理学会顧問．オンライン出版の研究に従事．2010 年情報処理記念賞受賞．情報処理学会理事．電子情報通信学会，IEEE，IEEE-CS，ACM 各会員．本会終身会員．