Dynamic Linking Method for an Embedded Component System

Katsuya Yamauchi[1,a)] Tomoaki Kawada[1] Hiroshi Oyama[2] Takuya Azumi[3] Hiroaki Takada[1]

Abstract: This paper presents a dynamic linking method using the TOPPERS Embedded Component System (TECS). Dynamic linking mechanisms (e.g., Shared Object and Dynamic Link Library) are responsible for loading and linking software modules at run-time. Such mechanisms are useful to allow a software module to be combined with other modules which are not known beforehand during the module's development, such as third-party applications and future versions of an application that is intended to be combined with the said module. Another benefit is the ability to update application modules individually, and to reduce the delivery cost of software updates. Those mechanisms have been put into practical use in general-purpose software systems. The use of a dynamic linking mechanism for embedded systems is also being considered; however, existing solutions are not always appropriate for embedded systems because of their tight memory constraints. Therefore, this paper proposes a linking method suitable for embedded systems that utilizes TECS, which has the advantage of ensuring interface consistency.

Keywords: Dynamic Linking, TECS, Loadable Module, Component-based Development, Embedded System

1. Introduction

Dynamic linking mechanisms (e.g., Shared Object and Dynamic Link Library) are responsible for loading and linking software modules at run-time [1], [2]. Such mechanisms are useful to allow a software module to be combined with other modules which are not known beforehand during the module's development. Another benefit is the ability to update application modules individually, and to reduce the delivery cost of software updates. Those mechanisms have been put into practical use in general-purpose software systems. On the other hand, in recent years, a technology for updating software wirelessly (over-the-air programming) has become widespread [3]. By using these technologies, partial software updates can be done efficiently. However, there is a problem that the memory amount required for linkage resolution becomes large. For this reason, existing solutions are not always appropriate for embedded systems that have tight memory constraints.

TOPPERS Embedded Component System (TECS) is a component system suitable for embedded systems [4]. Component system is a system that is divided into subsystems and made into parts to improve reusability. Dividing into subsystems facilitates parallel development and module changes. By using TECS, systems can enjoy the benefits of component-based development (e.g., reduction of initialization time and suppression of increased memory usage) without reducing execution efficiency [5], [6], [7]. Therefore, development suitable for a real-time system can be performed.

This paper proposes a dynamic linking method suitable for embedded systems utilizing TECS. By utilizing TECS, the benefits of component-based development can be applied to embedded systems. Therefore, even the embedded systems with tight memory constraints can use the dynamic linking mechanisms appropriately while suppressing the increase in memory usage.

In this study, we first applied TECS to a dynamic loading mode of TOPPERS/EV3RT which is a target environment of this study. Then, we designed a dynamic linking method utilizing TECS for TOPPERS/EV3RT and checked the operation if it works properly.

Our contributions in this paper are as follows:

(1) We designed to apply TECS to a dynamic loading mode of TOPPERS/EV3RT so that the functions by TECS can be used in this platform.
(2) We designed an API that utilizes TECS and implemented a dynamic linking method in embedded systems.
(3) We evaluated the application size before and after using TECS.

The remainder of this paper is organized as follows. After describing TECS and TOPPERS/EV3RT in Section 2, we will introduce the flow of applying TECS to TOPPERS/EV3RT in Section 3. Section 4 includes the design of the dynamic linking method utilizing TECS and comparison of application size. Finally, Section 5 concludes the paper and offers an outlook for future research.

2. Background

In this section, we describe TECS and TOPPERS/EV3RT, which our study targets.

2.1 TECS

TECS is a component system suitable for embedded systems. We explain the development process, component model, and

[1] Nagoya University, Nagoya, Japan
[2] OKUMA Corporation, Niwa-gun, Japan
[3] Saitama University, Saitama, Japan
[a)] katsuyama@ertl.jp

```
celltype tCellType1 {
    call sSignature cCall;        // Call Port
};
celltype tCellType2 {
    entry sSignature eEntry;      // Entry Port
    attr { int Attr; };           // Attribute
    var { int Var; };             // Variable
};

[domain(HRP, "user")]
region rSample {
    cell tCellType1 Cell1 { cCall = Cell2.eEntry; };      // Joined
    cell tCellType2 Cell2 { Attr = 1; };
};
```

Fig. 1 Description of celltypes and cells in CDL file

TECS plugin defined by the TECS specification [4], [8], [9].

Development Process

The first step of the TECS-based development process is to define the component types and the interfaces between them. Component definitions are written in TECS CDL and then processed by the *TECS generator* to generate the interface code. This code is combined with the celltype code that defines the behavior of each component.

Component Model

Cells are instances of TECS components. Cells have zero or more *entry ports* and *call ports*. The entry port is an interface that provides its own functions, and the call port is an interface for using the functions of other cells. These ports can be *joined* and provide the functions of one cell to another. The types of ports are called *signatures*. Each signature can contain one or more function header definitions which allows access between ports. When an entry port and a call port are joined, their signatures must be the same. Each call port must be joined with one entry port, and when joining with multiple entry ports, should be defined as a call port array. On the other hand, the entry port can be joined with any number of call port including zero.

Cells can also contain zero or more *attributes* and *variables*. Attributes are constants associated with each cell, and they can be read by the application and plugins. Variables are similar to attributes, but their values can change dynamically.

A *celltype* is a concept for representing a type of component. The entry ports, call ports, attributes, and variables of cells are defined in the celltype, and each cell can have unique values for attributes and variables. The behavior of a cell is defined by the celltype code associated with its celltype.

A *region* defines the protected boundaries of the cell. Cells in the same region can be freely joined, and access between different regions is possible by specifying the region to which the joined cell belongs.

Fig. 1 and **Fig. 2** show an example of TECS CDL code and diagram that defines celltypes and cells.

TECS Plugin

TECS plugins are extensions written in Ruby that the TECS generator can use. When the TECS generator reads the CDL file, it calls and uses the corresponding plugin function. The domain plugin is one of the plugin types supported by the TECS generator as shown in Fig. 1. By defining *[domain(HRP, "user")]*, the plugin specifies *rSample* as the user domain.

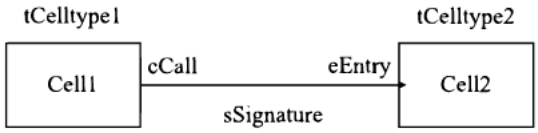

Fig. 2 Diagram corresponding to the description

2.2 TOPPERS/EV3RT

In this study, we used a robot development terminal named *Lego Mindstorms EV3* which was released by LEGO [10]. Functions such as screen display (LCD), motor, and sensor can be used with EV3. *TOPPERS/EV3RT* [11] is a software platform for this terminal based on *TOPPERS/HRP3* (a real-time operating system with memory protection) [12].

There are two types of executables for EV3RT, which are a dynamic loading mode and a standalone mode. The former is a mode that builds the *base system* and the *application* separately, and the latter is a mode that builds them as one binary file. In this study, we targeted the dynamic loading mode. The base system has functions that are common to all applications (e.g., kernel and device drivers for Mindstorms EV3). In the application, the tasks to run on EV3 are defined in C or Ruby language. Each application is built as a separate executable file and is called *Loadable User Module (LUM)*. When the LUM is executed, it loads the dynamic library, and the base system loads/links it. These modules can be added and removed dynamically at run-time in this mode.

TECS is not used in the current dynamic loading mode, and all application tasks must be prepared manually. If TECS can be used, the TECS generator processes the CDL file and generates the interface code, which reduces the amount of description and makes it easier to change the task. In the next section, this paper proposes the design of applying TECS to LUM to facilitate application changes.

3. Apply TECS to LUM

In this section, we describe the flow of applying TECS to the LUM. To use TECS, all developers have to do is call a set of TECS generator and TECS plugins from the makefile. In this study, the base system of EV3RT is not changed, and TECS is applied only to the LUM.

3.1 Change the Kernel of LUM

Before applying TECS to the LUM, we focused on how to generate *static APIs*. Static API is an interface for defining kernel object generation information and initial state. The HRP version of the application uses TECS plugin to generate it. **Fig. 3** shows an example of a static API for creating tasks. *TA_ACT*, *TASK_PRIORITY*, and *STACK_SIZE* represent the task's initial state, priority, and stack size, respectively. The developers set the values for these parameters, and the TECS generator automatically creates a static API.

LUM uses a static API for LUM instead of a static API for TOPPERS/HRP. If anything, a static API for LUM is similar to a static API for *TOPPERS/ASP* (a real-time operating system that

```
CRE_TSK ( TASK1 , { TA_ACT, 0, task,
  TASK_PRIORITY, STACK_SIZE, NULL } );
```

Fig. 3 Static API

```
factory {
  write( " tecsgen.cfg " ,
    " CRE_TSK(%s, { %s, $cbp$, tTask_start_task,
                %s, %s, NULL}); " ,
        id, taskAttribute, priority, stackSize);
};
```

Fig. 4 Factory description of static API in ASP

is the basis of HRP). There are not as many APIs for LUM as for HRP, and the processing is not complicated. In other words, LUM does not have enough static API to define even using a TECS plugin like HRP. Therefore, we changed the kernel of LUM from HRP to ASP. In the ASP version, the static API is generated by factory description as **Fig. 4**.

Due to the kernel change, some of the description used in the ASP kernel had to be modified for use in LUM (e.g., change specifier and removal of unsupported functions). After all the changes are complete, call the new kernel and TECS generator from the makefile to apply TECS to LUM.

3.2 Adding a Task by TECS

In the previous section, we made kernel changes and applied TECS to LUM. This makes it possible to add tasks with TECS without changing the entire application. We added a task in the application by TECS to check if TECS was applied correctly. The component definition of the added task is as **Fig. 5**.

tSampleTaskMain is the celltype of *SampleTaskMain*, and this task has an entry port (*eTaskBody*). *rSample* is a definition of the region and is specified as user domain by the plugin. *tTask* is an existing celltype used to generate a task. It has attributes that define the task status, priority, and stack size, and also has an entry port and a call port (*cBody*). By joining *cBody* and *eTaskBody*, *SampleTaskMain* becomes the entry point of the application's main task. The TECS generator processes this TECS CDL code and generates a template file to define the task. The task is completed when the developer describes the operation in this template file.

By inspecting the log output of the message with this task, we confirmed that the task added by TECS worked as intended. Thus, we confirmed that TECS was successfully applied to LUM.

4. Dynamic Linking Method Utilizing TECS

In this section, we describe the design of the dynamic linking method utilizing TECS.

4.1 Adding API Cells by TECS

We applied TECS to the LUM in the previous section. The next thing to do is to add cells that define the API of Mindstorms EV3. By adding these cells, we extended the behavior of API to work within the TECS framework. **Fig. 6** shows a system configuration of a dynamic loading mode with added cells.

User Application, *Application Programming Interface*, *Device Drivers*, and *TOPPERS/HRP3 Kernel* are the existing system configurations used in the dynamic loading mode. The first two are defined in the *user domain*, and the other two are defined in the *kernel domain*, which makes up LUM and the base system, respectively.

```
celltype tSampleTaskMain {
  entry sTaskBody eTaskBody;
};

[domain(HRP, "user")]
region rSample {
  cell tSampleTaskMain SampleTaskMain {};
  cell tTask SampleTask {
    taskAttribute = C_EXP("TA_ACT");
    priority = 42;
    stackSize = 1024;
    cBody = SampleTaskMain.eTaskBody;
  };
};
```

Fig. 5 The component definition of the added task

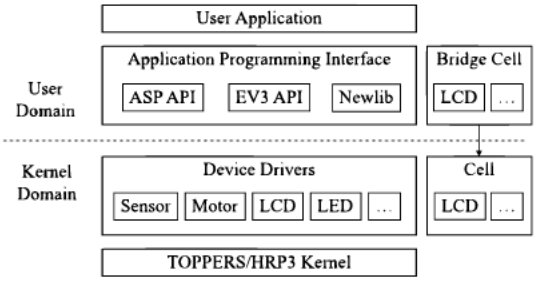

Fig. 6 A system configuration of dynamic loading mode with added cells

```
[domain(HRP, "user")]
region rUserDomain {
  cell nMruby::tsLCD BridgeLCD {
    cTECS = rKernelDomain::LCD.eLCD;
  };
};

[domain(HRP, "kernel")]
region rKernelDomain {
  cell tLCD LCD {
    cButton = Button.eButton;
  };
};
```

Fig. 7 The definition of extended service call

We added *Bridge Cells* for user domain and *Cells* for the kernel domain, and joined them. With such a cell configuration, the access is performed in the flow of an application, Bridge Cell, and Cell at the time of loading. However access beyond the domain is not allowed. It can be done by using the *extended service call* function. Extended service calls are generated by the TECS plugin by specifying the domain to join as shown in **Fig. 7**. *rKernelDomain* is specified when joining from BridgeLCD to LCD.

There are more than a dozen APIs for Mindstorms EV3, only the LCD and Button which are the most frequently used APIs were joined with the extended service call. For other APIs, both Bridge Cell and Cell are in the user domain.

```
[node, to_through(rKernelDomain, HRPSVCThroughPlugin, "")]
region rSample { ... };

[node, to_through(rKernelDomain, HRPSVCThroughPlugin, "")]
region rUserDomain { ... };

[domain(HRP, "kernel")]
region rKernelDomain { ... };
```

Fig. 8 The division method by node

```
namespace nLUM {
  celltype tTask {
    [inline] entry sTask eTask;
    [inline] entry siTask eiTask;
    call sTaskBody cBody;
  };
};
```

Fig. 9 Kernel division by namespace

4.2 Division of Cell Structure and Kernel Functions

We added Bridge Cells and Cells to define the APIs for Mindstorms EV3 in the previous section. Since all of these cells are defined in the same CDL file, TECS generator cannot determine which cells belong to the LUM. First of all, we describe how to divide these cells into the base system side and the LUM side.

By specifying the region, Bridge Cells belong to the user domain and Cells belong to the kernel domain. A region can optionally have an attribute called *node*. It is a unit consisting of one processor and one memory, and one node can be configured for each region. By specifying a node in the region, it can be regarded as a separate system even within the same CDL file. **Fig. 8** shows the division method by the node.

HRPSVCThroughPlugin is a plugin for extended service call [13]. By making such a description, *rSample*, *rUserDomain*, and *rKernelDomain* are devided into node units, and access to *rKernelDomain* from *rSample* and *rUserDomain* becomes possible. In the kernel domain, it is done implicitly without defining the node. These three regions are regarded as separate nodes, and can be distinguished by this definition.

Then, we describe the division of kernel functions. Since we changed the kernel of LUM in Section 3.1, the kernel used in the base system and LUM are different. These kernels are also called from the same CDL file, and some functions (e.g., tKernel and tTask) are commonly defined in both kernels. Therefore, TECS generator cannot determine which function to use in LUM.

Namespace is effective for kernel division. It is an identifier used to separate name scopes and prevents name collisions of celltypes and signatures. **Fig. 9** shows kernel division by namespace.

Fig. 9 is a part of the kernel on the LUM side. By enclosing the celltype in a namespace, *tTask* becomes a celltype with an identifier of *nLUM*. However, the same *tTask* definition exists in the kernel on the base system side, since it does not have an identifier, it is treated differently from the *tTask* as shown in Fig. 9. This allows functions with an *nLUM* identifier to be used by LUM, and functions without an *nLUM* identifier to be used by the base system.

```
celltype tSampleTaskMain {
  entry nLUM::sTaskBody eTaskBody;
  call sLCD cLCD;
  [optional] call nMruby::sInitializeBridge cInit;
};

region rSample {
  cell tSampleTaskMain SampleTaskMain {
    cLCD = rKernelDomain::LCD.eLCD;
    cInit = VM_TECSInitializer.eInitialize;
  };
};
```

Fig. 10 The component definition of MrubyVM

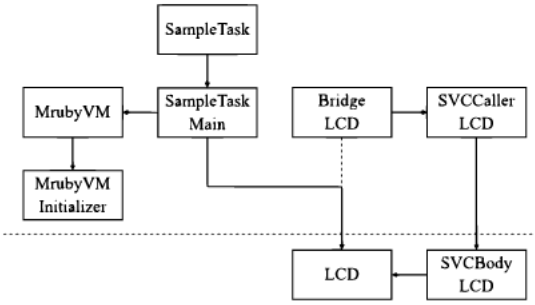

Fig. 11 The final cell composition of dynamic loading mode

Nodes and namaspaces allow the TECS generator to determine the information needed to build a LUM. As a result, the information required for each of the base system and the application could be divided.

4.3 Adding MrubyVM

By designing up to Section 4.2, the structure utilizing TECS was added to the existing dynamic loading mode. This change allowed to extend the dynamic loading mode to the system that is more suitable for embedded systems. In addition, this section introduces the design of the environment that uses *MrubyVM*.

MrubyVM is a set of cells for executing an application written in Ruby [14]. It can be used by calling the VM initialization cell from the main task cell as shown in **Fig. 10**. Since the initialization cell of MrubyVM is prepared, it can be used by defining a call port in the task and joining it with the entry port of MrubyVM cell.

Fig. 11 shows the final cell composition of our system. *SampleTask* and *SampleTaskMain* are the tasks added in Section 3.2, and are included in LUM. *Bridge LCD* and *LCD* are the API cells added in Section 4.1. *MrubyVM* and *MrubyVMInitializer* are the cells added in this section. *SVCCallerLCD* and *SVCBodyLCD* are the intermediate cells automatically generated by the extended service call plugin. *SampleTaskMain* is the entry point of the application's main task and calls the LCD function of EV3 and MrubyVM.

4.4 Evaluation of Application Size

With the designs so far, TECS could be applied to LUM and utilizing it. This section evaluates how the application size changed before and after applying TECS.

Table 1 shows the size breakdown of applications without and

Table 1 Size comparison of applications without and with TECS

	without TECS	with TECS
.text	32.41	32.90
.rodata	3.91	3.91
.dynsym	0.11	0.11
.rel	1.38	1.40
.dynstr	0.06	0.06
.hash	0.05	0.05
.got	0.44	0.44
.data	1.05	1.05
.bss	12.18	12.18
total	51.59	52.10

Round off to the second decimal place. [KB]

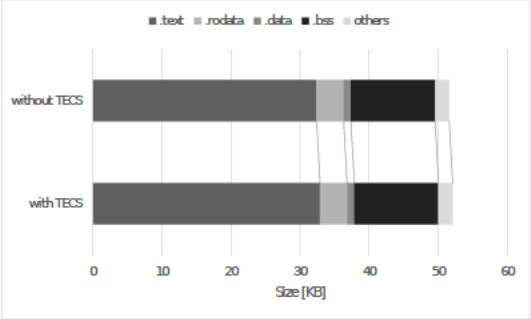

Fig. 12 The graph of size comparison

```
arm-none-eabi-strip app –strip-debug
```

Fig. 13 Strip command

with TECS. **Fig. 12** also shows the size comparison of the main elements in the breakdown. These applications have the same behavior of displaying a simple message on the screen of EV3. In addition, since the application without TECS does not have MrubyVM, the application with TECS in this evaluation has uncoupled MrubyVM.

The *.text* section stores the executable code. The reason for the increase in this section is due to the addition of new functions by applying TECS. The *.rodata* section stores the constants. The *.data* section and the *.bss* section stores the variables. Variables of the *.data* section have initial values, and variables of the *.bss* section do not have initial values. The size did not change in these sections because there were not many additional constants and variables defined when the task was created. Finally, Table 1 shows an increase in the size of the *.rel* section, which contains relocation information.

The compiled application contained debug information that had nothing to do with execution. Although it does not affect the application size, it can be removed with the *strip* command shown in **Fig. 13**. The values shown in Table 1 and Fig. 13 exclude debug information.

From these results, applying TECS does not significantly affect the application size. This is due to the advantage of component-based development, which is the effect of suppressing the increase in memory usage. Therefore, even in embedded systems with tight memory constraints, a dynamic linking method that utilizes TECS is effective.

5. Conclusion

This paper proposed a dynamic linking method suitable for embedded systems that utilizes TECS. First, we applied TECS to a dynamic loading mode of EV3RT to make it easier to add tasks to applications. Furthermore, we constituted API cells that can be used in EV3, and expanded systems to work within the TECS framework. Finally, we compared the size of applications without and with TECS. These changes allowed to extend the existing dynamic linking mechanism that can be used well in embedded systems.

However, this study has not made a perfect design. The TECS generator and plugin did not support this design method, and some files could not be automatically generated. Therefore, it was necessary to prepare an ideal output file by hand-coding. Extending the TECS generator and plugins is one of the future works.

Another future work is to strengthen the security aspect. Currently, the extended service calls make it easy to access cells in different domains. Therefore, we are considering a method of assigning a unique ID to the entry port. The task has the ID of the entry port to be joined as an attribute. After authenticating the declared ID in the task and the ID of the entry port, the application starts the operation of the task. By designing as above, the application can be executed only when the ID is known, and we believe that the safety of the base system can be further improved.

References

[1] M. Zaslavskiy, E. Ryabikov, K. Krinkin: *Lightweight Linux dynamic libraries profiling technique for embedded systems*, in Proc. of the 9th Central & Eastern European Software Engineering Conference, (2013).

[2] A. Acharya, J. Saltz: *Dynamic Linking for Mobile Programs*, International Workshop on Mobile Object Systems, (1996).

[3] M. Villegas, C. Orellana, H. Astudillo: *A study of over-the-air (OTA) update systems for CPS and IoT operating systems*, in Proc. of the 13th European Conference on Software Architecture (ECSA 19), (2019).

[4] T. Azumi, M. Yamamoto, Y. Kominami, N. Takagi, H. Oyama, and H. Takada: *A new specification of software components for embedded systems*, in Proc. of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 07), pp. 46-50 (2007).

[5] X. Cai, M. R. Lyu, K.-F. Wong, R. Ko: *Component-based software engineering: Technologies, development frameworks, and quality assurance schemes*, in Proc. of the 7th Asia-Pacific Software Engineering Conference (APSEC 2000), pp. 372-379 (2000).

[6] I. Crnkovic: *Component-based Software Engineering for Embedded Systems*, in Proc. of the 27th International Conference on Software Engineering, pp. 712-713 (2005).

[7] B. Bonakdarpour, S. S. Kulkarni: *Compositional Verification of Fault-tolerant Real-time Programs*, in Proc. of the 7th ACM International Conference on Embedded Software (EMSOFT 09) pp. 29-38 (2009).

[8] TECS Specification, V1.0.2.32 ed., TOPPERS Project, Inc., (2011).

[9] T. Kawada, T. Azumi, H. Oyama, H. Takada: *Componentizing an Operating System Feature Using a TECS Plugin*, 2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA), (2016).

[10] LEGO Education (online), available from ⟨https://education.lego.com/en-us/⟩, (accessed 2020-11-2).

[11] Y. Li, Y. Matsubara, H. Takada: *EV3RT: A Real-time Software Platform for LEGO Mindstorms EV3*, Computer Software Volume 34 Issue 4, pp. 91-115 (2017).

[12] TOPPERS Project, Inc. (online), available from ⟨https://dev.toppers.jp/trac_user/ev3pf/wiki/WhatsEV3RT⟩, (accessed 2020-9-29).

[13] T. Ishikawa, T. Azumi, H. Oyama, H. Takada: *HR-TECS: Component Technology for Embedded Systems with Memory Protection*, in Proc. of the 16th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2013), (2013).

[14] T. Azumi, Y. Nagahara, H. Oyama, N. Nishio: *mruby on TECS: Component-based Framework for Running Script Program*, in Proc. of the 18th IEEE International Symposium on Real-Time Computing (ISORC 2015), (2015).