Journal of Information Processing Vol.22 No.4 1–5 (July 2014)

[DOI: 10.2197/ipsjjip.22.1]

Regular Paper

Component-based Software Development Framework for Embedded IoT Devices（仮）

Takuro Yamamoto[1] Takuma Hara[2] Takuya Ishikawa[2] Hiroshi Oyama[3] Hiroaki Takada[2]
Takuya Azumi[1]

Received: March 4, 2014, Accepted: August 1, 2014

Abstract:

Keywords:

1. Introduction

The Internet of Things (IoT) is an essential next evolutionary step for the Internet [**?**] in which various items and platforms, for example, wearable devices and smart devices, will be connected via the Internet to further enrich people's lives. The IoT uses embedded systems such as data sensors and controlling actuators as elemental constituents, and they 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 [**?**], [**?**]. 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 [**?**], [**?**]. Individual component diagrams enable the visualization of an entire system. In addition, component-based systems are flexible with regard to extensibility and specification changes. The TOPPERS embedded component system (TECS) [**?**], AUTOSAR [**?**], and SaveCCM [**?**] 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 embedded 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 [1]. This framework integrates two technologies, i.e., mruby, which is a lightweight implementation of Ruby for embedded systems [**?**], [**?**], and TECS, which is a component-based framework for embedded systems [**?**], [**?**].

This paper proposes an extended framework of mruby on TECS that can be applied to embedded network software development for IoT devices. In the proposed framework, a component-based TCP/IP protocol stack, TINET+TECS [**?**], is comprised, and it is possible to utilize TINET function from mruby programs.

TINET is a compact TCP/IP protocol stack for embedded systems [**?**]. TINET comprises many complex source codes, i.e., it contains many files and defines many macros, which can be problematic for software developers seeking to maintain, extend, and analyze the software. TINET+TECS is a componentized TIENT with TECS to improve the configurability and scalability of TCP/IP software.

In addition, this paper proposes a component-based dynamic memory allocator, TLSF+TECS. TLSF is a dynamic memory allocator for real-time systems, which can always run with $O(1)$ and improve memory usage efficiency by deviding memory blocks in two stages. In the current version of TLSF, memory contention may occur when multiple threads run at the same time. TLSF+TECS is a componentized TLSF memory allocator, which can be thread-safe allocator because each component has its own heap area.

Contributions: The proposed framework provides the following contributions.

(1) :

(2) :

(3) :

(4) :

[1] Graduate School of Engineering Science, Osaka University
[2] Graduate School of Informatics, Nagoya University
[3] OKUMA Corporation

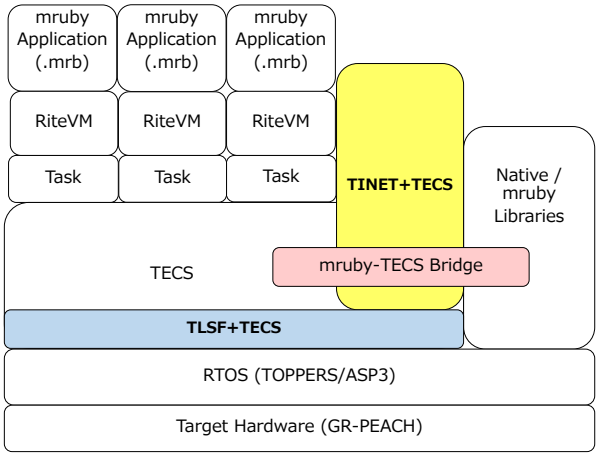

Fig. 1 System model of the proposed framework

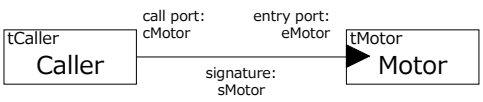

Fig. 2 Component Diagram

```
1  signature sMotor {
2      int32_t getCounts( void );
3      ER resetCounts( void );
4      ER setPower( [in]int power );
5      ER stop( [in] bool_t brake );
6      ER rotate( [in] int degrees,
7                 [in] uint32_t speed_abs,
8                  [in] bool_t blocking );
9      void initializePort( [in]int32_t type );
10 };
```

Fig. 3 Signature Description

```
1  celltype tCaller {
2      call sMotor cMotor;
3  };
4  celltype tMotor {
5      entry sMotor eMotor;
6      attr {
7          int32_t port;
8      };
9      var {
10         int32_t currentSpeed = 0;
11     };
12 };
```

Fig. 4 Celltype Description

Organization: The remainder of this paper is organized as follows. Section 2 introduces the system model of the proposed framework and the basic technologies, i.e., mruby, TECS, and mruby on TECS. Section 3 describes the design and implementation of the proposed framework, including TINET+TECS and TLSF+TECS. Section 4 evaluates the proposed framework. Related work is discussed in Section 5. Conclusions and suggestions for future work are presented in Section 6.

2. System Model

This section describes the system model of the proposed framework, including basic technologies such as TECS. The system model of the proposed framework is shown in Fig.1.

The proposed framework is an extension of mruby on TECS framework [1] [2], and utilizes two technologies: mruby and TECS. In this section, mruby (2.1), TECS (2.2), and mruby on TECS framework (2.3) are described.

2.1 mruby

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.2 shows a component diagram. A *cell*, which is an instance of a component in TECS, consists of *entry* ports, *call* ports, attributes and internal variables. 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.3). 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.4). 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 connect *cell*s. Fig.5 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.5, *entry* port eMotor in *cell* Motor is connected to *call* port

```
1 cell tMotor Motor {
2       port = C_EXP("PORT_A");
3 };
4 cell tCaller Caller {
5       cMotor = Motor.eMotor;
6 };
```

Fig. 5 Build Description

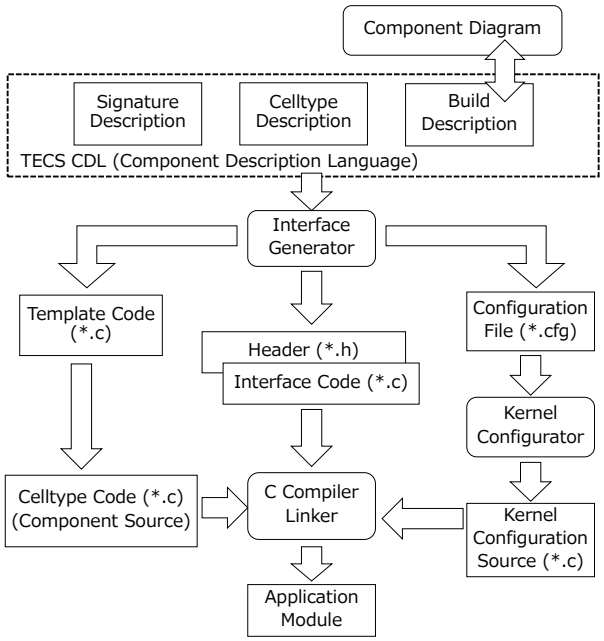

Fig. 6 Development flow using TECS

cMotor in *cell* Caller. *C_EXP* calls macros defined in C files.

2.2.3 Development Flow

Fig.6 shows the development flow using TECS. TECS generator generates the interface code (.H and .C) and the configure file of the RTOS (.cfg) from the CDL file.

Software developers using TECS can be divided into component designers and application developers. Component designers define signatures, which are interfaces between cells, and celltypes, which are types of cells. Using the template code generated from the CDL file in which these are defined, component designers implement the functions and behaviors of the component in C language. The source code implementing the function of the component is called a celltype code. Application developers develop applications by using component diagrams and predefined celltype to connect cells with build description. An application module is generated by compiling and linking the header, the interface code, and the celltype code.

2.3 mruby on TECS

3. Proposed Framework

3.1 TLSF+TECS

3.1.1 TLSF

TLSF (Two-Level Segregate Fit) memory allocator [3] [4] is a dynamic memory allocator suitable for the real-time system proposed by M. Masmano et al. TLSF memory allocator has the following features.

Real-time property

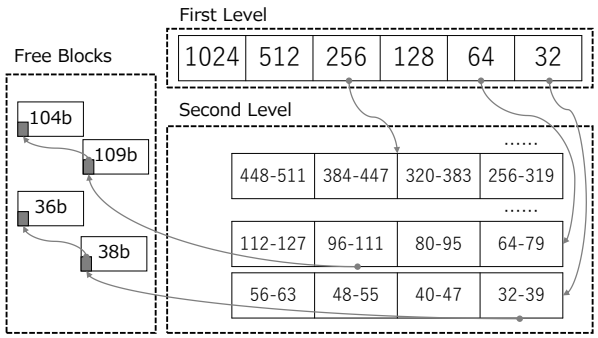

Fig. 7 TLSF Algorithm

```
1 signature sMalloc {
2       int initializeMemoryPool(void);
3       void *calloc( [in]size_t nelem,
4                     [in]size_t elem_size );
5       void *malloc( [in]size_t size );
6       void *realloc( [in]const void *ptr,
7                      [in]size_t new_size );
8       void free( [in]const void *ptr );
9 };
```

Fig. 8 Signature description of memory management

The worst execution time required for allocating and deallocating memory does not depend on the data size. TLSF always works with $O(1)$, and it is possible to estimate the response time.

Fast speed

In addition to being able to always estimate the worst execution time, TLSF is executed at high speed.

Efficient memory consumption

Memory efficiency is improved by suppressing memory fragmentation. Various tests have obtained an average fragmentation of less than 15% and a maximum fragmentation of less than 25%.

3.1.2 TLSF Algorithms

TLSF algorithm classifies memory blocks into two stages and searches for a memory block that is optimal for the requested memory size. The overview of TLSF algorithm is shown in Fig.7. Consider a case where a request, *malloc*(100), is called to allocate a memory. In the first step, it is classified by the most significant bit of the requested memory size. In this case, since 100 is represented by binary number as 1100100, it is in the range from 64 to 128 from the most significant bit. In the second step, it is further classified. In this case, 64 to 128 are divided into 4, and 100 is in the block of 96 to 111. Free block [*1] in this range is used.

A simple fixed-size memory block allocator results in waste of up to 50%, but TLSF classifies it finely in two steps, so it is a memory efficient algorithm. In addition, TLSF searches at the same speed and at high speed, $O(1)$.

3.1.3 Component Design of TLSF

This section describes the component design of TLSF memory allocator. In this research, we are using TECS to componentize TLSF. The version of TLSF used is 2.4.6[*2].

Fig.8 is a signature description for memory management used by the allocator. It defines the memory pool initialization function

[*1] Free block is an available memory block.
[*2] http://www.gii.upv.es/tlsf/main/repo

```
1  celltype tTLSFMalloc {
2      [inline]
3          entry sMalloc eMalloc;
4      attr {
5          /* memory pool size in bytes */
6          size_t memoryPoolSize;
7      };
8      var {
9          [size_is( memoryPoolSize / 8 )]
10             uint64_t *pool;
11     };
12 };
```

Fig. 9 Celltype description of TLSF memory allocator component

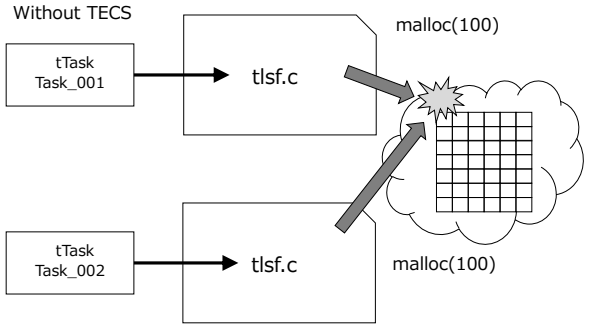

Fig. 10 TLSF before componentization

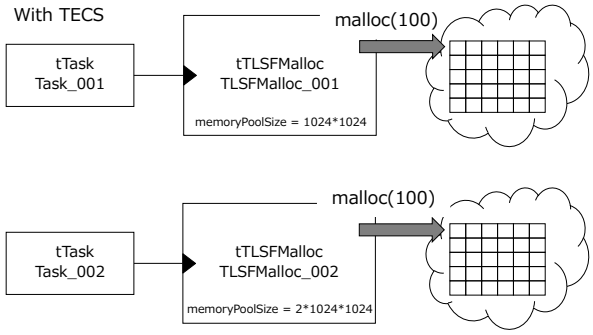

Fig. 11 TLSF after componentization

initializeMemoryPool, memory allocation function *calloc*, *malloc*, *realloc*, and memory release function *free* as a signatures.

The celltype description of the TLSF memory allocator component is shown in Fig.9. An entry port, *eMalloc*, is connected to all components that perform memory management such as *malloc* and *free*. Here, *[inline]* is a specifier for Implementation as inline functions. A memory pool size is defined as an attribute, and a pointer to a memory pool is defined as an variable. Each component holds its own heap area, so even when calling functions for memory management at the same time in different threads, it is possible to operate without memory contention.

As shown in Fig.10, since TLSF before componentization shares the heap area with multiple threads, if memory is allocated or released simultaneously from multiple threads, memory competition may occur in some cases. In the research, TLSF is componentized using TECS as shown in Fig.11. It is possible to operate in thread safe without exclusive control, because each component independently holds a heap area and manages memory within it.

Fig.12 shows the build description of the TLSF memory allocator component shown in Fig.11[*3]. Two sets of task components and TLSF components are combined. Each memory pool size can be configured as an variable (Lines 5 and 11 in Fig.12). Fig.13 is the part of the code actually calling the function of the TLSF memory allocator component. The use part shows a function that the mruby VM allocates memory in the mruby on TECS framework [1] [2] which is introduced in Section 2.3. Lines 8 calls the *free* function of the TLSF memory allocator component. *cMalloc_* represents the name of the calle port (Lines 2 in Fig.12). Likewise, Lines 13 and 17 call the function for memory allocation. The heap area of *TLSFMalloc_001* component is used if the code of 13 is executed in *Task_001*, and if that is executed in *Task_002*, the heap area of *TLSFMalloc_002* component is used, respectively. In this way, in the component-based development using TECS, it is possible to operate with the same code without modifying the C code, although the cells are different.

```
1  cell tTask Task_001 {
2      cMalloc = TLSFMalloc_001.eMalloc;
3  };
4  cell tTLSFMalloc TLSFMalloc_001 {
5      memoryPoolSize = 1024*1024; /* 1MB */
6  };
7  cell tTask Task_002 {
8      cMalloc = TLSFMalloc_002.eMalloc;
9  };
10 cell tTLSFMalloc TLSFMalloc_002 {
11     memoryPoolSize = 2*1024*1024; /* 2MB */
12 };
```

Fig. 12 Build description of TLSF memory allocator component

```
1  void*
2  mrb_TECS_allocf(mrb_state *mrb, void *p,
3                  size_t size, void *ud)
4  {
5      CELLCB *p_cellcb = (CELLCB *)ud;
6      if (size == 0) {
7          //tlsf_free(p);
8          cMalloc_free(p);
9          return NULL;
10     }
11     else if (p) {
12         //return tlsf_realloc(p, size);
13         return cMalloc_realloc(p, size);
14     }
15     else {
16         //return tlsf_malloc(size);
17         return cMalloc_malloc(size);
18     }
19 }
```

Fig. 13 Example of TLSF memory allocator component

4. Evaluation

5. Related Work

6. Conclusion

Acknowledgments

References

[1] Azumi, T., Nagahara, Y., Oyama, H. and Nishio, N.: mruby on TECS: Component-Based Framework for Running Script Program, *Proceedings of the 18th IEEE International Symposium on Real-Time Comput-*

[*3] Other call/entry ports, attributes, and valuables are actually described, but it is omitted here for the simplicity.

ing (ISORC), pp. 252–259 (2015).
[2] Yamamoto, T., Oyama, H. and Azumi, T.: Lightweight Ruby Framework for Improving Embedded Software Efficiency, *Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pp. 71–76 (2016).
[3] Masmano, M., Ripoll, I., Crespo, A. and Real, J.: TLSF: a New Dynamic Memory Allocator for Real-Time Systems, *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 79–88 (2004).
[4] : TLSF, `http://www.gii.upv.es/tlsf/`.