Journal of Information Processing Vol.22 No.4 1–10 (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 [1] 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 [2], [3]. 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 [4], [5]. 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) [6], AUTOSAR [7], and SaveCCM [8] 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 is 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 [9], [**?**]. This framework integrates two technologies, i.e., mruby, which is a lightweight implementation of Ruby for embedded systems [10], [11], and TECS, which is a component-based framework for embedded systems [6], [12].

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 [13]. 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 the case of only sending values obtained by sensors, it is better to custimize easily, such as removing unused functions. The improved configurability leads to satisfy strict memory constrains of embedded systems. 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) **Applicable to various devices:** The proposed framework does not depend on RTOSs or hardware, it can be utilized in various devices. mruby code is portable, so it is possible to run the same program on different devices.

(2) **Improve configurability:** Because TINET+TECS is a

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

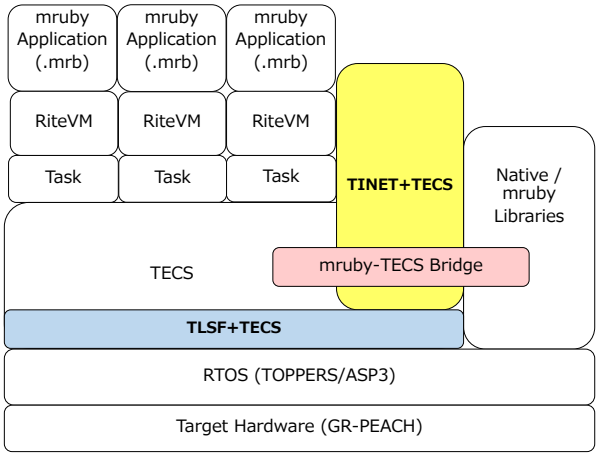

Fig. 1 System model of the proposed framework

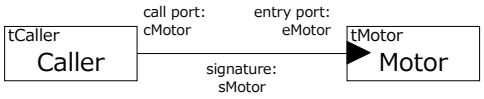

Fig. 2 Component Diagram

```
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

component-based system, its software can flexibly change as system's configuration by, for example, resizing network buffer, adding/removing TCP (or UDP) functionality, or supporting either IPv4 or IPv6. In addition, the use by TINET+TECS of individual component diagrams enables visualization of an entire system.

(3) **Thread-safe memory allocator:** TLSF+TECS runs thread safely without exclusive control even if multiple threads operate in parallel. Moreover, each thread can easily set its own heap area.

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., TECS, mruby, 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 and mruby. The proposed framework is an extension of mruby on TECS framework [9] [14], and utilizes two technologies: mruby and TECS. The system model of the proposed framework is shown in Fig.1. In the proposed framework, each mruby program runs on a RiteVM mapped to a componentized task of an RTOS. mruby programs can call the TINET functions required for network programming through the mruby-TECS bridge, and thus software to be embedded in IoT devices can be developed. Moreover, the proposed framework comprises TLSF+TECS. TLSF+TECS is utilized for memory management of mruby's RiteVMs and TINET+TECS, which helps to improve the efficiency of memory consumption.

The following subsection explains TECS, mruby, and mruby on TECS framework.

2.1 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.

2.1.1 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*.

2.1.2 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

```
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

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

Fig. 5 Build Description

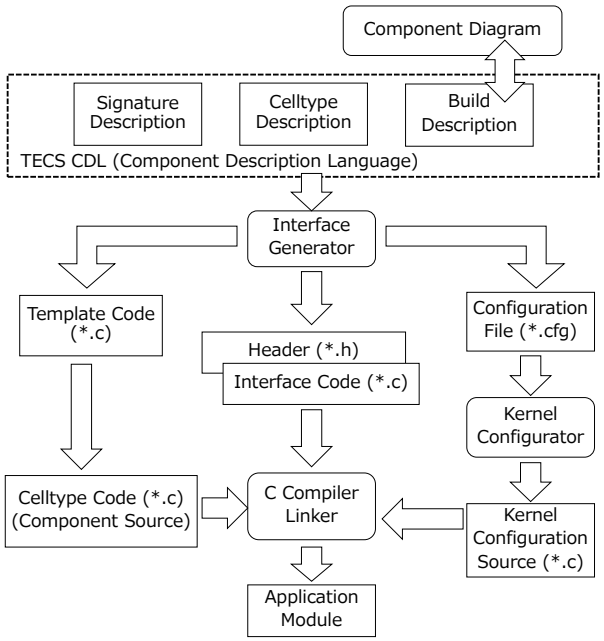

Fig. 6 Development flow using TECS

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

2.1.3 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 *signature*s, which are interfaces between *cell*s, and *celltype*s, which are types of *cell*s. 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 *cell*s with build description. An application module is generated by compiling and linking the header, the interface code, and the *celltype* code.

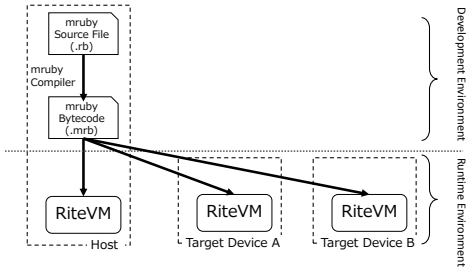

Fig. 7 mruby/RiteVM mechanism

2.2 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 [15] 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.

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.7. 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.

2.3 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.8. Each mruby program, which is a 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 provides native libraries for mruby and can call a native program (e.g., C legacy code) from an mruby program. The mruby-TECS bridge also provides TECS components for receiving the invocation from an mruby program.

In this paper, TOPPERS/ASP3 [16], [17] is the target RTOS and is based on μITRON [18] . However, mruby on TECS does not depend on the RTOS because TECS supports not only TOP-

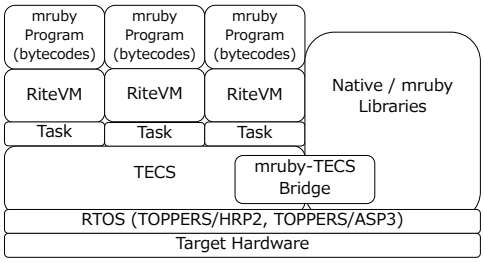

Fig. 8 System model of existing mruby on TECS

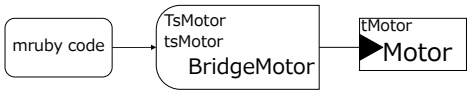

Fig. 9 mruby-TECS bridge

PERS/ASP3 but also the other RTOSs such as OSEK [19] and TOPPERS/HRP2 [20], [21].

2.3.2 mruby-TECS Bridge

There is a significant difference between the execution times of mruby and C language codes. According to [9], 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.9 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 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. Proposed Framework

The proposed framework is an extended mruby on TECS framework for network programming. It can use TINET+TECS functions from mruby programs. TINET+TECS is a component-based TCP/IP protocol stack comprised in the proposed framework, and it compensates for the original TINET's weak point, that it is hard to maintain, extend, and analyze the software due to many complex source codes and improves the configurability. In addition, TLSF+TECS, a component-based dynamic memory allocator, is used for memory management of mruby's RiteVMs and TCP/IP buffers in the proposed framework. Since each component holds its own heap area, TLSF+TECS allows concurrent operation without exclusive control while improving the efficiency of memory consumption which is a merit of TLSF.

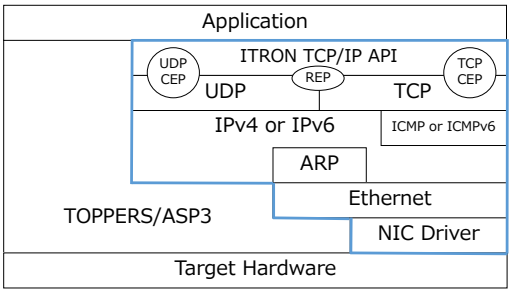

Fig. 10 TINET and TOPPERS/ASP3 hierarchy diagrams

3.1 TINET+TECS

3.1.1 TINET

TINET is a compact TCP/IP protocol stack for embedded systems based on the ITRON[*1] TCP/IP API Specification [22], developed by the TOPPERS Project [12]. TINET has been released as an open-source tool.

To satisfy restrictions for embedded systems in terms of, for example, memory capacity, size, and power consumption, TINET supports the following functions:

- minimum copy frequency,
- elimination of dynamic memory control,
- asynchronous interfacing,
- error detailing per API.

Overview: TINET runs as middleware on TOPPERS/ASP3 [23] [24], a real-time kernel based on μITRON [18]. As it is compatible with TOPPERS RTOS, TINET also supports other RTOSs such as TOPPERS/ASP and TOPPERS/JSP.

Fig.10 shows the hierarchy diagram of TINET and TOPPERS/ASP3. Users transmit and receive data using a Communication End Point (CEP), an interface that functions like a socket. In the transmission process, headers are attached to the data body passed to the CEP at each protocol layer before the data are transmitted from the network device. In the reception process, the headers of the data bodies received by the network device are analyzed at each protocol layer, and the data are then passed to the CEP.

A TCP reception point called the REP stands by to receive connection requests from the partner side. The REP has an IP address (*myaddr*) and a port number (*myportno*) as attributes and performs functions such *bind()* and *listen()*.

In TINET, the amount of data copying between each protocol layers is minimized. In standard computing systems, the TCP/IP protocol stack has large overheads in terms of execution time and memory consumption because the data are copied at each protocol layer. To solve this problem, TINET does pass the pointer of the data buffer between each protocol layer instead of performing data copying.

3.1.2 Component Design of TINET+TECS

TINET+TECS, the proposed componentized TCP/IP protocol stack, comprises a number of some TECS components. This section describes the components of the TINET+TECS framework with the aid of component diagrams.

[*1] ITRON is a real-time operating system (RTOS) developed by the TRON project.

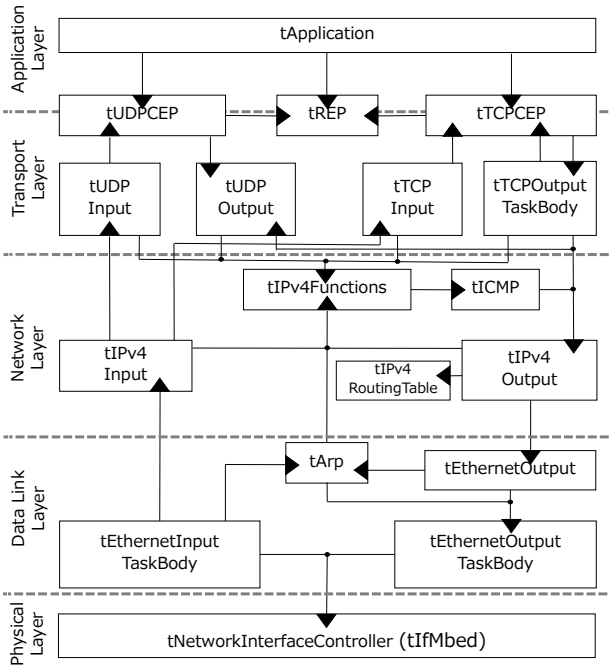

Fig. 11 Component diagram of a protocol stack

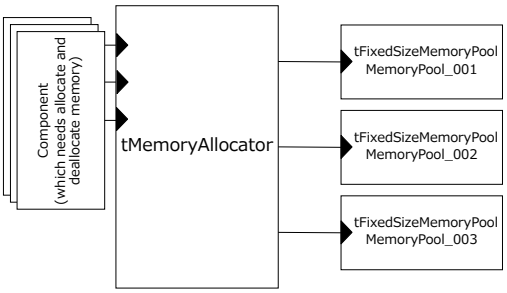

Fig. 12 Component diagram of tMemoryAllocator

Components of a protocol stack

The components of a TINET+TECS protocol stack are shown in Fig.11. Note that some small particle components, such as a kernel object, data queues, and semaphores, are omitted to simplify the component diagram. In TINET+TECS, the components are divided for each protocol, and functionalities such as input and output functions are defined as respective components. By using such small grain components, software visibility is improved. The components of each protocol are described in the following.

Application layer: An application in TINET+TECS is implemented as a component such as tApplication. Software with TINET uses ITRON TCP/IP API [22] such as *tcp_snd_dat* and *tcp_rcv_dat*. In TINET+TECS, the application component calls TECS functions such as *cTCPAPI_sendData* and *cTCPAPI_receiveData*. Moreover, in TINET+TECS supporting a TECS adapter, an existing application with TINET can run on the TINET+TECS framework without transporting, and therefore, software can be developed either using existing methods or as TECS components.

Transport layer: tTCPCEP (tUDPCEP) and tREP are, respectively, CEP and REP components TCP (UDP). For example, a server program supporting multiple clients can be developed by preparing multiple tTCPCEP components. tTCPInput (tUDPInput) and tTCPOutput (tUDPOutput) are components for performing, respectively, receiving and sending processing in the transport layer.

Network layer: The tIPv4Input and tIPv4Output components perform, respectively, the receiving and sending processing in the network layer. The tIPv4Functions component performs functions such as checksum, the tICMP component is used for the Internet Control Message Protocol (ICMP), and the tIPv4RoutingTable component operates a routing table.

Data link layer: tEthernetInputTaskBody and tEthernetOutputTaskBody (tEthernetOutput) are components for performing, respectively, receiving and sending processing in the data link layer. The tArp component is for implementing the Address Resolution Protocol (ARP).

Physical layer: The tNetworkInterfaceContoroller component implements a network device driver. Software can be run on other devices by replacing the component because only the component depends on the target device.

To utilize the protocol stack in the same manner in the original TINET, communication object components such as tTCPCEP, tUDPCEP, and tREP are defined as an interface between TINET+TECS and an application. The communication object component corresponds to a CEP or REP of the original TINET. Application developers can utilize the TINET+TECS functionalities by generating and combining as many components as necessary.

The protocol stack of TINET+TECS supports the coexistence of multiple protocols. Though its use of IPv6 and Point-to-Point Protocol (PPP) components, TINET+TECS can make IPv4 and IPv6 coexist and support PPP without modification of component implementation.

Memory allocator component

The original TINET eliminates dynamic memory control to meet the severe memory restrictions of embedded systems. A memory area for sending/receiving data in the protocol stack is allocated and released within a predetermined area. The memory allocator component allows for elimination of dynamic memory control in TINET+TECS by providing a requested memory area from the statically allocated memory area.

The memory allocator component connects to as many tFixedSizeMemoryPool as required, as shown in Fig.12. tFixedSizeMemoryPool is a componentized kernel object of TOPPERS/ASP3 for allocating and releasing memory areas of a requested size. tFixedSizeMemoryPool components of various sizes are prepared, and an appropriate memory area can be allocated according to the used data size. On the other hand, all components that need to allocate or deallocate memory, e.g., tTCPInput and tEthernetOutput, connect to the memory allocator component.

In addition, TINET+TECS utilizes the TECS *send/receive* specifier to minimize the memory copy frequency, which is a functionality supported by TINET.

Send/receive specifiers: TECS supports *send*/*receive* interface specifiers [25]. TINET+TECS uses *send* and *receive* specifiers instead of *in* and *out* to reduce the number of copies:

- *in* is a specifier for input arguments. A callee side uses

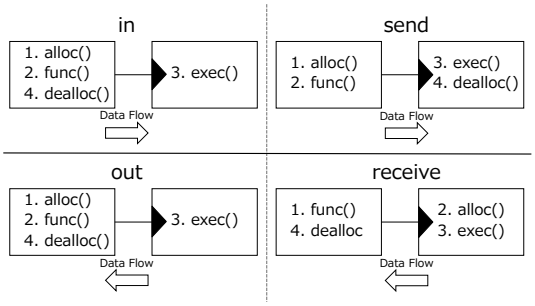

Fig. 13 Differences between in/out and send/receive

```
1  signature sNicDriver {
2     void start(
3        [send(sNetworkAlloc),size_is(size)]
4              int8_t *outputp, .., ..);
5     void read(
6        [receive(sNetworkAlloc),size_is(*size)]
7              int8_t **inputp, .., ..);
8        /* Omit: other functions */
9  };
```

Fig. 14 Signature description of the nic driver (An example of send/receive)

the memory of arguments with *in* when executing the callee function. When the processing returns to the caller side, the caller can reuse and deallocate the memory.

- *send* is another specifier for transferring data to a callee from a caller such as *in*. The difference between *in* and *send* is whether the data memory is deallocated in the caller or callee, as shown in Fig.13. In the case of the *in* specifier, both allocating and deallocating of the data memory are performed in the caller. By contrast, in the case of *send*, the caller allocates the data memory and the callee deallocates it.
- *out* is a specifier for output arguments through which a callee writes data in the memory allocated by a caller while the caller receives the data.
- *receive* is another specifier for a caller receiving data from a callee such as *out*. The difference between *out* and *receive* lies in whether the data memory is allocated in the caller or callee, as shown in Fig.13. In the case of *out*, the callee writes data in the memory allocated by a caller, whereas in the *receive* case, the callee allocates the data memory. Deallocating of the memory is performed in the caller in both cases.

As shown in Fig.14, sending and receiving arguments such as *outputp* and *inputp* are defined using, respectively, the *send/receive* specifier in the signature description. Developers hardly make mistakes of memory operation because these specifiers completely pass an ownership of memory. Common object request broker architecture (CORBA) does not consider memory sharing; CORBA has no functionalities such as *send*/*receive*.

3.2 TLSF+TECS

3.2.1 TLSF

TLSF (Two-Level Segregate Fit) memory allocator [26] [27] is a dynamic memory allocator suitable for the real-time system proposed by M. Masmano et al. TLSF memory allocator has the following features.

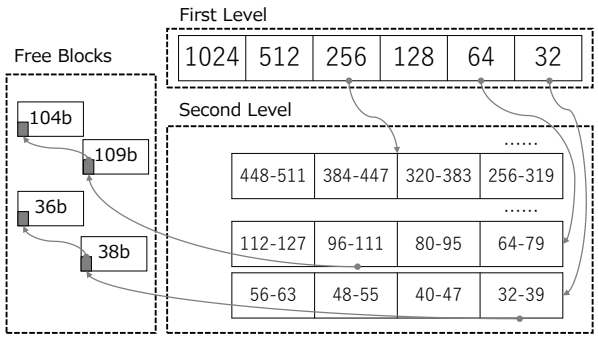

Fig. 15 TLSF Algorithm

```
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. 16 Signature description of memory management

Real-time property

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.

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%.

3.2.2 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.15. 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[*2] 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, $O(1)$.

3.2.3 Component Design of TLSF+TECS

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[*3].

Fig.16 is a signature description for memory management used by the allocator. It defines the memory pool initialization function *initializeMemoryPool*, memory allocation function *calloc*, *mal-*

[*2] Free block is an available memory block.
[*3] 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. 17 Celltype description of TLSF memory allocator component

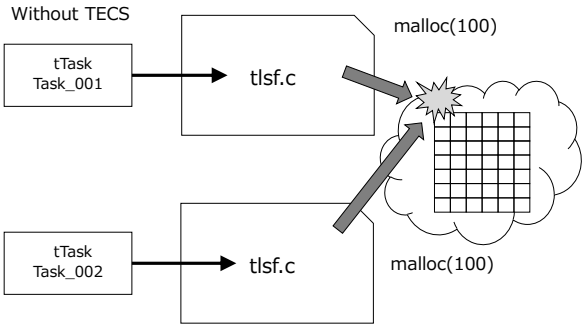

Fig. 18 TLSF before componentization

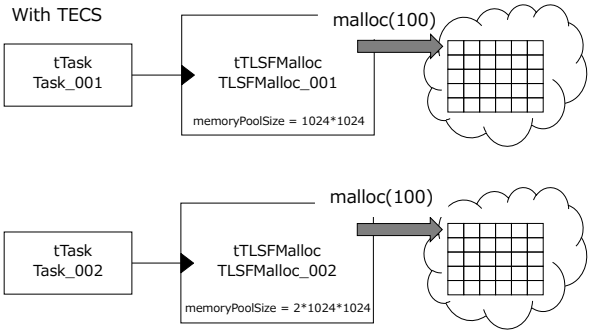

Fig. 19 TLSF after componentization

loc, *realloc*, and memory release function *free* as a signature.

The celltype description of the TLSF memory allocator component is shown in Fig.17. 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 a 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.18, 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.19. 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.20 shows the build description of the TLSF memory allocator component shown in Fig.19[*4]. Two sets of task components and TLSF components are combined. Each memory pool size can be configured as a variable (Line 5 and 11 in Fig.20). Fig.21 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 [9] [14] which is introduced in Section 2.3. Line 8 calls the *free* function of the TLSF memory allocator component. *cMalloc_* represents the name of the call port (Line 2 in Fig.20). Likewise, Line 13 and 17 call the function for memory allocation. The heap area of *TLSFMalloc_001* component is used if the code of 21 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. 20 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. 21 Example of TLSF memory allocator component

3.2.4 Multiple RiteVM

The proposed framework uses TLSF memory allocator for memory management of RiteVMs. However, since the existing TLSF is not thread-safe, if memory is allocated or deallocated from multiple, memory contention will occur. As a RiteVM allocates and deallocates memory at high frequency, the memory contention immediately occurs in the case of multiple VMs.

The TLSF components are connected to the RiteVMs to hold its own heap area in each RiteVM as shown in Fig.22. Since each TLSF component holds a memory pool, multiple RiteVMs can be executed without memory contention. Fig.22 shows that the first RiteVM has a heap area of 1 MB (1024*1024) and the second

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

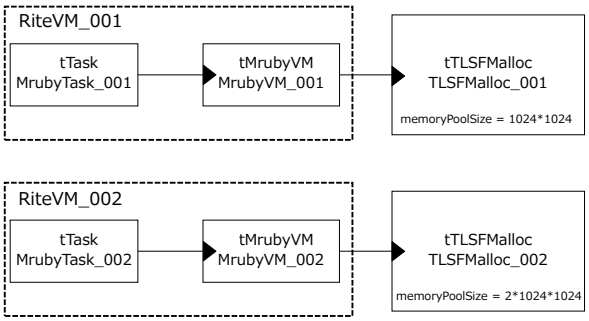

Fig. 22 Component description of RiteVM and TLSF+TECS

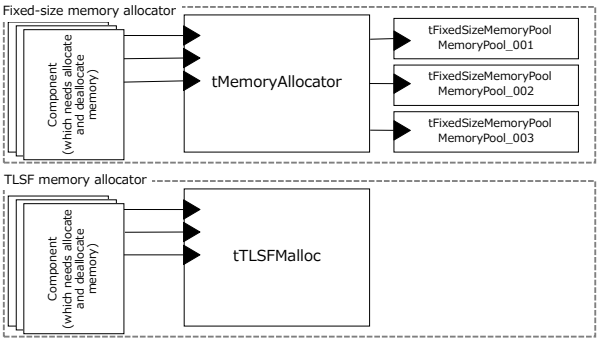

Fig. 23 Fixed-size and TLSF memory allocator components

VM has a heap area of 2 MB (2*1024*1024). It is easy to set different heap areas in each RiteVM. In addition, RiteVM performs incremental GC (Garbage Collection), and a VM that started GC does not disturb the execution of other VMs by running GC.

3.2.5 Memory management for data sending and receiving

In TCP/IP protocol stack, memory allocation and release are repeated in each layer such as TCP layer, IP layer, and Ethernet layer, so the role of allocator is important. TINET+TECS combines all components that manage memory with the allocator. The TLSF memory allocator can execute at the same speed as compared with the fixed-size memory allocator which TOPPERS/ASP3 supports as standard, and can improve the memory efficiency. In addition, as shown in Fig.23, the fixed-size memory allocator needs to prepare memory pools of different sizes, and select a memory pool as necessary. TLSF+TECS efficiently manages the memory without selecting a memory pool. Since TINET+TECS is component-based system, TLSF+TECS can be easily extended to TINET+TECS.

3.3 Use case

In the proposed framework, applications can call TINET functions such as TCP and UDP from mruby programs because mruby-TECS bridge automatically generates the code linking mruby and C. Fig.24 shows an mruby program to transmit the value acquired from a sensor. mruby makes it easier to develop applications than existing TINET, i.e., C language.

In the case of a simple application, a few functions are only used and there are some unused functions. For example, the application only uses a function to send by TCP protocol as shown in Fig.24. The proposed framework incorporates TINET+TECS, and can easily custimize such as removing UDP functions, supporting only IPv4, or removing TCP receiving function. It can be applied to embedded systems with strict memory constraints by removing unused functions.

```
1  begin
2      io = AnalogIO.new(A0, INPUT)
3      cep = TCP.new()
4      cep.accept
5      loop do
6          val = io.read
7          cep.snd val.to_s + "\n"
8          RTOS.delay(1000)
9      end
10 rescue => e
11     puts "[ERROR]" + e
12 end
```

Fig. 24 Example of mruby application

Table 1 Evaluation Board Environment

Board	GR-PEACH
CPU	Cortex-A9 RZ/A1H 400MHz
Flash ROM	8 MB
RAM	10 MB
LAN Controller	LAN8710A

4. Evaluation

This section describes the experimental evaluation used to demonstrate the effectiveness of the proposed framework.

GR-PEACH was employed as the evaluation board. Detailed specifications of the board are shown in Table 1. We also employ TINET 1.5.4 and the compiler arm-none-eabi-gcc 5.2 To pretest the system, we connected the board to a host PC via a LAN cable and evaluated the data sending and receiving.

4.1 Performance of TINET+TECS

To demonstrate the low overhead of TINET+TECS, we compared its execution time and memory consumption with that of TINET, producing the results shown in Fig. 25.

The *tcp_snd_dat* and *tcp_rcv_dat* APIs were used in the evaluation to, respectively, send and receive TCP data. For *tcp_snd_dat*, we measured the executing time starting from the API call through the application until the return of the processing result. In TINET+TECS, this process is performed in the order tApplication, tTCPCEP, tTCPOutputTaskBody, tIPv4Output, tEthernetOutput, tArp, tEthernetOutputTaskBody, and tIfMbed, as shown in Fig. 11. For *tcp_rcv_dat*, we measured the execution time from the data receipt in the LAN driver until data acquisition in the application. In TINET+TECS, the process is performed in the order tIfMbed, tEthernetInputTaskBody, tIPv4Input, tTCPInput, tTCPCEP, and tApplication, as shown in Fig. 11. The execution time of TINET+TECS is close to that of TINET, with an overhead of about 3 us. As the use of the *send/receive* specifier enables accessing of the buffer address without data copying, the componentization overhead does not affect the execution time.

The memory consumptions of TINET and TINET+TECS are compared in Table 2. The memory consumption of TINET+TECS is about 1% higher than that of TINET, as the data and processes such as initialization of cells, descriptors, function tables, and skeleton functions needed to manage TECS components increase memory consumption.

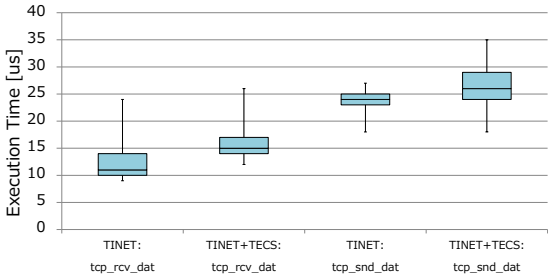

Fig. 25 Execution times of TINET and TINET+TECS

Table 2 Memory consumption of TINET and TINET+TECS

	text	data	bss	total
TINET	183.94 KB	5.37 KB	132.03 KB	322.34 KB
TINET+TECS	170.73 KB	5.37 KB	149.13 KB	325.23 KB

Including the application and kernel objects

Table 3 Modified code lines of CDL

	Size	Size (– Default)	CDL
Default	325.23 KB	0 KB	0 lines
I	305.40 KB	– 19.83 KB	18 lines
I + II	304.12 KB	– 21.10 KB	27 lines
I + II + III	303.45 KB	– 21.77 KB	32 lines

I: Remove TCP
II: Remove ICMP
III: Change network buffer (Remove memory pools)

As shown in Table 3, the code lines for modification were measured to demonstrate the improved configurability. This demonstrated the ability to change the composition of the protocol stack with a small workload, confirming that the proposed framework improves the configurability.

4.2 Performance of TLSF+TECS

5. Related Work

To develop the software of IoT systems, several approaches have been proposed [28] such as General-purpose Programming Languages (GPLs), Wireless Sensor Network (WSN) macroprogramming, Cloud-based platforms, and Model-Driven Development (MDD).

The development using GPLs such as C, JavaScript, Python, and Android allows the extremely efficient systems based on the complete control over individual devices. However, GPLs need more development effort, and it is difficult to reuse software due to platform-dependent design.

WSN macroprogramming provides abstractions to specify high-level collaborative behaviors, while hiding low-level details such as message passing and state maintenance. nesC, a programming language used to build applications for the TinyOS platform [29], has been proposed. nesC/TinyOS is designed for WSN nodes with limited resources e.g., 8 kilobytes of program memory, 512 bytes of RAM, but not supported TCP/IP implementation.

Cloud-based platform reduces development efforts by providing cloud-base APIs and high-level constructs (e.g., drag-and-drop). In addition, it offers the ease deployment and evolution because the application logic is centrally located in a cloud platform. However, it is platform-dependent design, and restricts developers in terms of functionality such as in-network aggregation or direct node-to-node communication locally.

To address the issues of development efforts and platform-dependent design, MDD has been proposed [**?**]. MDD provides the benefits of re-usable, platform-independent, extensible design, however it needs long development time to build a MDD system.

6. Conclusion

Acknowledgments

References

[1] Perera, C., Zaslavsky, A., Christen, P. and Georgakopoulos, D.: Context Aware Computing for The Internet of Things: A Survey, *IEEE Communications Surveys Tutorials*, Vol. 16, No. 1, pp. 414–454 (2014).
[2] Crnkovic, I.: Component-based Software Engineering for Embedded Systems, *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pp. 712–713 (2005).
[3] Cai, X., Lyu, M. R., Wong, K.-F. and Ko, R.: Component-based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes, *Processings of 7th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 372–379 (2000).
[4] Gössler, G. and Aştefănoaei, L.: Blaming in Component-based Real-time Systems, *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14 (2014).
[5] Bonakdarpour, B. and Kulkarni, S. S.: Compositional Verification of Fault-tolerant Real-time Programs, *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pp. 29–38 (2009).
[6] Azumi, T., Yamamoto, M., Kominami, Y., Takagi, N., Oyama, H. and Takada, H.: A New Specification of Software Components for Embedded Systems, *Proceedings of the 10th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pp. 46–50 (2007).
[7] : AUTOSAR, `http://www.autosar.org/`.
[8] Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P. and Tivoli, M.: The SAVE Approach to Component-based Development of Vehicular Systems, *Journal of Systems and Software*, Vol. 80, No. 5, pp. 655–667 (2007).
[9] 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 Computing (ISORC)*, pp. 252–259 (2015).
[10] Tanaka, K. and Higashi, H.: mruby – Rapid IoT Software Development, *Proceedings of 17th International Conference on Computational Science and Its Applications (ICCSA)*, pp. 733–742 (2017).
[11] : mruby, `https://github.com/mruby/mruby`.
[12] : TOPPERS Project, `http://www.toppers.jp/en/index.html`.
[13] : TINET, `https://www.toppers.jp/en/tinet.html`.
[14] Yamamoto, T., Oyama, H. and Azumi, T.: Component-Based Framework of Lightweight Ruby for Efficient Embedded Software Development, *JSSST Journal on Computer Software*, Vol. 34, No. 4, pp. 3–16 (2017).
[15] : Ruby, `https://www.ruby-lang.org/en/`.
[16] Azumi, T., Ukai, T., Oyama, H. and Takada, H.: Wheeled Inverted Pendulum with Embedded Component System: A Case Study, *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pp. 151–155 (2010).
[17] : TOPPERS/ASP kernel, `https://www.toppers.jp/en/asp-kernel.html`.
[18] Takada, H. and Sakamura, K.: μITRON for Small-Scale Embedded Systems, *IEEE Micro*, Vol. 15, No. 6, pp. 46–54 (1995).
[19] Ohno, A., Azumi, T. and Nishio, N.: TECS Components Providing Functionalities of OSEK Specifications for ITRON OS, *Journal of Information Processing*, Vol. 22, No. 4, pp. 584–594 (2014).
[20] : TOPPERS/HRP2 kernel, `http://www.toppers.jp/en/hrp2-kernel.html`.
[21] Ishikawa, T., Azumi, T., Oyama, H. and Takada, H.: HR-TECS: Component Technology for Embedded Systems with Memory Protection, *Proceedings of the 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pp. 1–8 (2013).
[22] : ITRON TCP/IP API Specification (Ver. 2.00.00), `http://www.tron.org/wp-content/themes/dp-magjam/pdf/specifications/en_US/TEF024-S003-02.00.00_en.pdf`.

[23] Kawada, T., Azumi, T., Oyama, H. and Takada, H.: Componentizing an Operating System Feature Using a TECS Plugin, *Proceedings of the 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pp. 95–99 (2016).
[24] : TOPPERS/ASP3 kernel, `https://www.toppers.jp/asp3-kernel.html`.
[25] Azumi, T., Oyama, H. and Takada, H.: Memory Allocator for Efficient Task Communications by Using RPC Channels in an Embedded Component System, *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications (SEA)*, pp. 204–209 (2008).
[26] 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).
[27] : TLSF, `http://www.gii.upv.es/tlsf/`.
[28] Chauhan, S., Patel, P., Delicato, F. C. and Chaudhary, S.: A Development Framework for Programming Cyber-physical Systems, *Proceedings of the 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, pp. 47–53 (2016).
[29] Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E. and Culler, D.: The nesC Language: A Holistic Approach to Networked Embedded Systems, *Acm Sigplan Notices*, Vol. 38, No. 5, pp. 1–11 (2003).