Regular Paper

Component-Based mruby Platform for IoT Devices

Takuro Yamamoto[1,a]   Takuma Hara[2]   Takuya Ishikawa[2]   Hiroshi Oyama[3]   Hiroaki Takada[2]
Takuya Azumi[4]

Abstract: In embedded network software running on embedded systems within the Internet of Things (IoT), high levels of runtime efficiency and user productivity are required. As an approach to improve the productivity of software development, the mruby on TOPPERS embedded component system (TECS) framework has been proposed; note that mruby on TECS framework employs a scripting language (i.e., a lightweight Ruby implementation) and supports component-based development. In this paper, we propose an extended mruby on TECS framework for its application in developing software for IoT devices, including sensors and actuators. Our proposed framework enables mruby programs to utilize Tomakomai Internetworking (TINET), a TCP/IP protocol stack specifically designed for use in embedded systems. Further, the proposed framework incorporates two component-based functions, i.e., a componentized TINET stack called TINET+TECS and a componentized Two-Level Segregate Fit (TLSF) dynamic memory allocator called TLSF+TECS. Here, TINET+TECS improves configurability and scalability and offers software developers high levels of productivity through variable network buffer sizes and the ability to add or remove various TCP (or UDP) functions. TINET+TECS utilizes a dynamic TECS component connection method to satisfy the original TINET specifications. Further, TLSF+TECS is a thread-safe memory allocator that runs at high speeds and efficiently consumes memory. The experimental results of the comparison between TINET+TECS and the original TINET show that execution time and memory consumption overhead are both reduced; further, we conclude that configurability is improved. Finally, the TLSF+TECS function which obtains and reports statistical information regarding mruby's virtual machine (VM) memory usage, helps developers debug and verify their embedded IoT systems.

Keywords: Internet of Things, software development framework, component-based development, embedded systems

1. Introduction

As an essential next evolutionary step for the Internet, the Internet of Things (IoT) connects various items and platforms such as wearable devices and smart devices, via the Internet to further enrich people's lives [1]. The IoT uses embedded systems, such as data sensors and controlling actuators, as elemental constituents; therefore, these devices must demonstrate consistently high levels of quality and runtime performance. These requirements have consequently led to an increase in their complexity and scale; further, to be successful and widely adopted by users, these systems must have low production costs and short development cycles.

Complex and large-scale software systems can be developed efficiently using component-based techniques [2], [3]. Particularly, component-based development is a design technique that is applicable to the development of reusable software. Given its importance in terms of reliability, verification of component-based systems has been extensively researched [4], [5]. Individual component diagrams enable the visualization of an entire system. Further, component-based systems are flexible in terms of their extensibility and resilience to specification changes. Typical component-based development environments focused on embedded systems include the TOPPERS embedded component system (TECS) [6], AUTOSAR [7], and SaveCCM [8].

In addition to specific development environments, scripting languages, such as Ruby, JavaScript, Perl, Python, and Lua, offer efficient approaches to develop quality software. Currently, most embedded software is implemented using the C language, but development in C typically results in large code size, high costs, and significant development time. However, the use of scripting languages improves the efficiency of software engineering and can shorten the development period because it effectively supports and promotes the development of reusable scripts.

Embedded systems often have real-time requirements; therefore, real-time properties, such as estimating worst-case execution times, are very important. Although scripting languages are easy to use and read, their execution typically requires more time than that required for executing the code written in C. Therefore, applying scripting languages to embedded systems poses a major difficulty. To address this limitation, in Refs. [9] and [10], mruby on TECS has been proposed as a component-based framework for efficiently running script-based programs. This framework integrates two key technologies, i.e., mruby, which is a lightweight implementation of Ruby designed specifically for embedded systems [11], [12] and TECS, which is a component-based framework also designed specifically for embedded systems [6].

1   Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka 560–8531, Japan
2   Graduate School of Informatics, Nagoya University, Nagoya, Aichi 464–8603, Japan
3   OKUMA Corporation, Niwa-gun, Aichi 480–0193, Japan
4   Graduate School of Science and Engineering, Saitama University, Saitama 338–8570, Japan
a)   t-yamamoto@hopf.sys.es.osaka-u.ac.jp

In this paper, we propose an extended framework for mruby on TECS that can be applied to embedded network software development involving IoT devices. The proposed framework makes it possible to utilize Tomakomai Internetworking (TINET) functions from within the mruby programs. Note that TINET is a compact TCP/IP protocol stack for embedded systems [13] that comprises a number of complex source code files, 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. To overcome the above problem, TINET+TECS is a componentized TINET implementation that incorporates TECS to improve the configurability and scalability of TCP/IP software. For IoT applications that should only be sending values obtained by one or more than one sensors, it is ideal to easily customize the minimum configuration of the TCP/IP protocol stack, e.g., by removing unused functions. This improved configurability also leads to satisfying strict memory constraints of IoT devices.

In addition to TINET+TECS, the proposed framework incorporates a component-based dynamic memory allocator based on Two-Level Segregate Fit (TLSF) called TLSF+TECS. Note that TLSF is a dynamic memory allocator designed specifically for real-time systems that always run in constant time (i.e., $O(1)$) and improves memory usage efficiency by dividing memory blocks in two distinct stages. In the current version of TLSF, memory contention may occur when multiple threads run simultaneously. To address this problem, TLSF+TECS is a componentized TLSF memory allocator that is also entirely thread-safe because each component has its own heap area from which memory is actually allocated. In TLSF+TECS, developers can also obtain statistical information regarding memory usage, which is crucial in analyzing memory operations and locating bugs.

Contributions: The proposed framework provides the following three contributions.

Applicability to various devices. The proposed framework does not depend on real-time operating systems (RTOSs) or specific hardware; instead, it can be utilized by various devices, i.e., mruby code is portable. Therefore, it is possible to run the same program on different devices.

Improved configurability. Because TINET+TECS is a component-based system, its software can flexibly change in response to system configuration changes, such as resizing of network buffers, adding or removing TCP (or UDP) functions, and supporting both IPv4 and IPv6. Further, the use of individual component diagrams enables visualizations of the entire system.

Thread-safe memory allocation TLSF+TECS safely runs multiple threads without exclusive control even if the threads operate concurrently. To achieve this, each thread can easily set up its own heap area. Further, statistical information is available to help developers debug and verify the specific memory usage of the given system.

Organization: The remainder of this paper is organized as follows. Section 2 introduces the system model and fundamental technologies, i.e., TECS, mruby, and mruby on TECS. Next, Section 3 describes the design and implementation of the proposed framework, including TINET+TECS and TLSF+TECS. In Section 4, we provide a detailed evaluation of the proposed frame-

work. Finally, related work is discussed in Section 5 and Section 6 concludes this paper.

2. System Model

This section describes the system model of the proposed framework, including fundamental technologies such as TECS and mruby. The proposed framework is an extension of mruby on TECS framework [9], [10], 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 software through the mruby-TECS bridge, and thus software to be embedded in IoT devices can be developed.

Requrements: The requirements of the proposed mruby platform for IoT devices are defined as follows.

R1: TCP/IP functions can be utilized from mruby programs and the protocol stack can be easily configured for productivity since the network function is essential to the IoT systems.

R2: Multiple mruby programs can run concurrently to improve the productivity of software development. A thread-safe memory allocator is required to prevent the multiple mruby tasks from conflicting their memory.

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.

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

2.1.1 Component Model

Figure 2 shows a component diagram. A *cell*, which is an instance of a TECS component, consists of *entry* ports, *call* ports, attributes and variables. An *entry* port is an interface that provides functions to other *cell*s, and a *call* port is an interface that
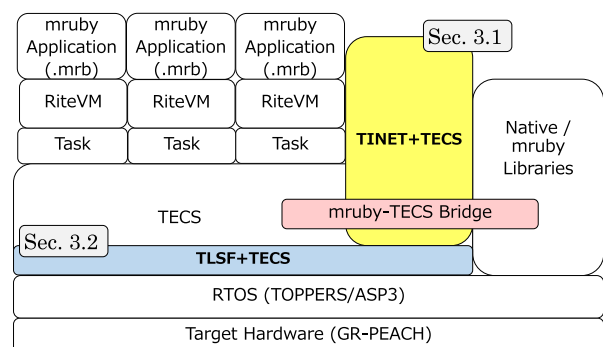

Fig. 1 System model of the proposed framework.
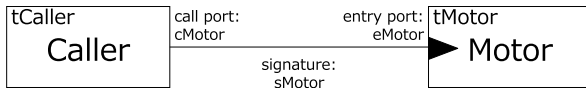
Fig. 2   Component diagram.

```
1  signature sMotor {
2      void initializePort( [in]int32_t type );
3      ER setPower( [in]int power );
4      ER stop( [in] bool_t brake );
5  };
```

Fig. 3   Signature description.

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

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.

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 *var*, respectively.

Build Description   The build description is used to instantiate and connect *cell*s. **Figure 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.
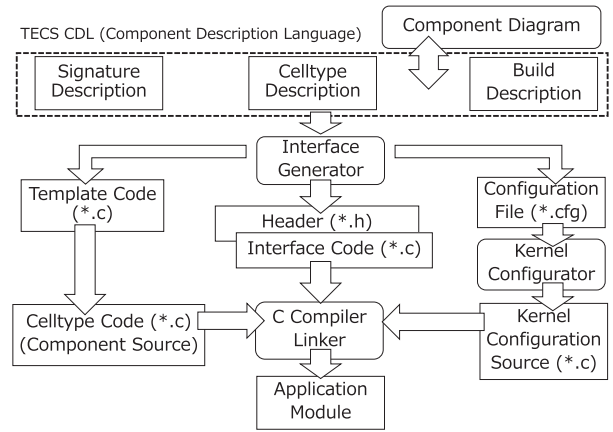

Fig. 6   Development flow using TECS.

2.1.3   Development Flow

Figure 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.

2.2   mruby

mruby is a light-weight implementation of the Ruby programming language complying to part of the ISO standard [12]. Ruby is an object-oriented scripting language [14] 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 the development of embedded software using a script language without slowing down the execution time.
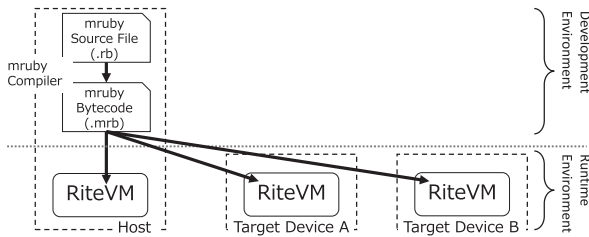
Fig. 7   mruby/RiteVM mechanism.

Fig. 8   mruby-TECS bridge.

2.3.1   System Model of mruby on TECS

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 [15], [16] is the target RTOS and is based on μITRON [17] . However, mruby on TECS does not depend on the RTOS because TECS supports not only TOPPERS/ASP3 but also the other RTOSs such as OSEK [18] and TOPPERS/HRP2 [19], [20].

2.3.2   mruby-TECS Bridge

There is a significant difference between the execution times of mruby and C language codes. According to Ref. [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.

Figure 8 illustrates an mruby-TECS bridge used to control a motor. The left side of BridgeMotor belongs to the mruby program. The right side of BridgeMotor belongs to the 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 the 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 to develop network software for IoT devices. De-
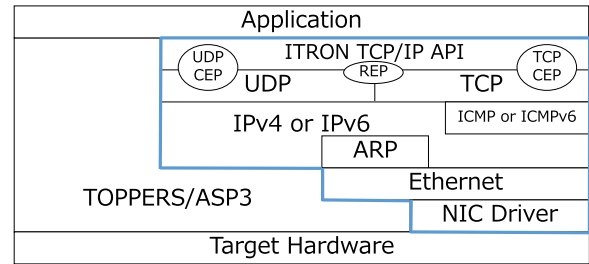

Fig. 9   TINET and TOPPERS/ASP3 hierarchy diagrams.

velopers can use TCP- and UDP-related functions from mruby programs. The proposed framework incorporates two functionalities: TINET+TECS and TLSF+TECS. 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. TLSF+TECS which is a component-based dynamic memory allocator is used for the memory management of RiteVMs and TCP/IP buffers in the proposed framework. Since each TLSF component maintains its own heap area, TLSF+TECS allows concurrent operation without exclusive control while improving the efficiency of memory consumption, which is the advantage of TLSF.

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 [21], developed by the TOPPERS Project [22]. 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 functions such as minimum copy frequency, elimination of dynamic memory control, asynchronous interfacing, error detailing per API.

Overview: TINET runs as middleware on TOPPERS/ ASP3 [15], [16], a real-time kernel based on μITRON [17]. As it is compatible with TOPPERS RTOS, TINET also supports other RTOSs such as TOPPERS/ASP and TOPPERS/JSP.

Figure 9 shows the hierarchy diagram of TINET and TOPPERS/ASP3. Users send 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 at each protocol layer is minimized. In standard computing systems, the TCP/IP protocol

---

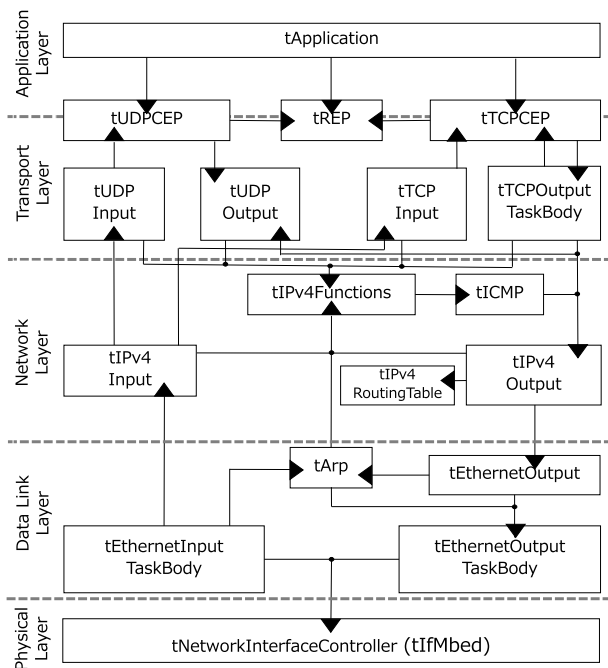[*1]   ITRON is an RTOS developed by the TRON project.

Fig. 10   Component diagram of a protocol stack.

Fig. 11   Component diagram of tMemoryAllocator.

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 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.

Components of a protocol stack

The components of the TINET+TECS protocol stack are shown in **Fig. 10**. 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/output functions are defined as respective components. By using such small grain components, software visibility is improved. The components of each protocol are described as follows.

Application layer: An application in TINET+TECS is implemented as a component such as tApplication.   Software with TINET uses ITRON TCP/IP API [21] 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.   For example, a server program supporting multiple clients can be developed by preparing multiple tTCPCEP components. tTCPInput and tTCPOutput are components for performing, respectively, receiving and sending processing in the transport layer.
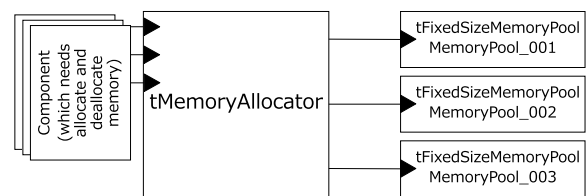
Network layer: The tIPv4Input and tIPv4Output components perform, respectively, the receiving and sending processing in the network layer.   The tIPv4Functions component performs functions such as checksum, the tICMP component is used for the Internet Control Message Protocol (ICMP), and the tIPv4RoutingTable 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 applications.   The communication object component corresponds to a CEP or REP of the original TINET. Application developers can utilize TINET+TECS functionalities by generating and combining as many components as necessary.

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 the 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. 11**.   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.

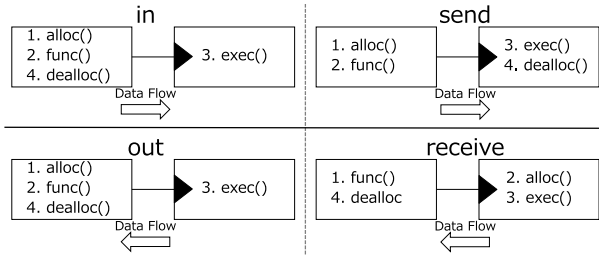**Send/receive specifiers:**   TECS supports *send*/*receive* inter-

Fig. 12   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. 13   Signature description of the nic driver (An example of send/receive).

face specifiers [23]. 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 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. The difference between *in* and *send* is whether the data memory is deallocated in the caller or callee, as shown in **Fig. 12**. 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. The difference between *out* and *receive* lies in whether the data memory is allocated in the caller or callee, as shown in Fig. 12. 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. 13**, 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*.

3.1.3   Dynamic Connection in TECS

TECS supports a dynamic connection, a method for switching the binding of components at runtime (**Fig. 14**) as a new functionality. In Fig. 14, the solid line represents binding and the dotted line represents non-binding. Note that all components are statically generated in TECS, which can optimize the overhead of componentization because components are statically configured. Dynamically generating components causes a good deal of mem-
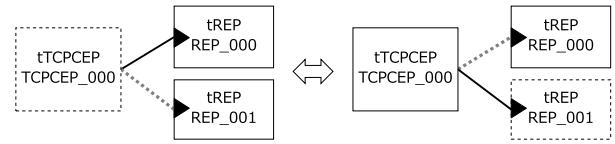

Fig. 14   Dynamic connection.

Fig. 15   Dynamic connection between CEP and REP.

```
1  signature sRepSelector {
2      void getRep([out]Descriptor(sREP4) *desc,
3              [in]int_t i);
4  };
5  celltype tRepSelector {
6      entry sRepSelector eRepSelector;
7      [ref_desc]
8        call sREP4 cREP[NUM_REP];
9  };
10 celltype tTCPCEP {
11     call sRepSelector cRepSelector;
12     [dynamic]
13       call sREP4 cREP;
14     /* Omit: other call/entry ports */
15     /* Omit: attributes and variables */
16 };
```

Fig. 16   Signature and celltype description for the dynamic connection.

ory consumption, which is a serious problem for embedded systems with strict memory constraints. The proposed framework can take advantage of the componentization in TINET while satisfying the memory constraint because components are statically generated and dynamically connected in TECS.

TINET+TECS utilizes the dynamic connection to switch between CEP and REP components, as shown in **Fig. 15**. In a server application, CEP is associated with REP in the state of waiting for a connection request from clients [*2]. For example, when processing with the HTTP protocol, CEP passively opens with an REP of port number 80.

To utilize dynamic connectivity, a selector should be defined. A selector connects all components that can be dynamically connected under a common descriptor that serves as an identifier to access each component [24]. The cREP ports form a call port array connecting to connecting to all tREP cells (Line 8 in **Fig. 16**). *[ref_desc]* is used to identify call ports referring to descriptors. In the case of Fig. 15, the tRepSelector cell connects all tREP cells.

A CEP component has two call ports: the cRepSelector port, which connects to the eRepSelector port of tRepSelector cell, and the cREP4 port, which connects to either of the tREP cells (Lines 11–13 in Fig. 16). The cREP port is defined using *[dynamic]* to identify the call port used to dynamically switch the components. The call port with the *[dynamic]* specifier is not optimized and is allocated in RAM using a plug-in.

Figure 17 shows a sample code of the dynamic connection. The eAPI_accept function is the function wrapping *tcp_acp_cep* under TECS, which is set as the state waiting for a connection request. The dynamic connection in the function is performed as shown in Fig. 17. First, the descriptor of REP to be joined is

---

[*2]   tcp_acp_cep(ID cepid, ID repid, T_IPV4EP *p_dstaddr, TMO tmout).

```
1  eAPI_accept (.., ..) {
2      /* Get a descriptor of intended REP cell */
3      cRepSelector_getRep(&desc, repid);
4      /* Set the descriptor */
5      cREP_set_descriptor(desc);
6      /* Call the function of intended REP cell */
7      cREP_getEndpoint();
8  }
```

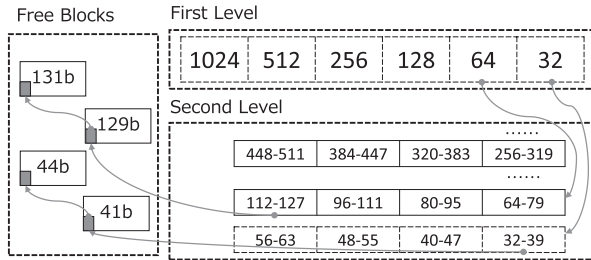**Fig. 17** Accept function (a dynamic connection example).

Fig. 18 TLSF algorithm.

obtained (Line 3 in Fig. 17). The first argument, *&desc*, is a variable used to store the descriptor information, whereas the second argument, *repid*, is the index of tREP cells. Next, the descriptor is set (Line 5 in Fig. 17), and the cREP port combines the tREP cell specified by the descriptor, enabling the tCEP cell to call the function of the tREP cell to be joined (Line 7 in Fig. 17).

3.2 TLSF+TECS

3.2.1 TLSF

The TLSF (Two-Level Segregate Fit) memory allocator [25], [26] is a dynamic memory allocator that is suitable for use in the real-time systems. As such, the TLSF memory allocator provides the following two features.

Real-time property. In TLSF, the worst-case execution time required for allocating and deallocating memory does not depend on the given data size. Instead, TLSF always runs in constant time (i.e., $O(1)$), and it is possible to estimate response time.

Efficient memory consumption. Memory efficiency is improved by suppressing memory fragmentation. Various experiments have achieved an average fragmentation of less than 15% and a maximum fragmentation of less than 25% [26].

3.2.2 TLSF Algorithm

As illustrated in **Fig. 18**, the TLSF algorithm classifies memory blocks into two stages and searches for a memory block that optimally lines up with the requested memory size. Particularly, consider the case in which a request to dynamically allocate 98 bytes is made via *malloc*(98). First, the request is classified based on the leftmost 1 bit in the requested memory size. In this example, since 98 is represented in binary as byte 01100010, it falls in the range of 64-128 based on the leftmost 1 bit. Second, the request is further classified as follows. The range from 64 to 128 is further divided into four equally sized groups, with 98 falling into the block ranging from 96 to 111. A free block [*3] in this range is then used to fulfill the memory allocation request.

Without using the above approach, a simple fixed-size memory block allocator results in wasted memory blocks of up to 50%. As illustrated above, TLSF classifies the memory allocation process

---

[*3] A free block is an available memory block.

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

Fig. 19 Signature description of memory management.

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

Fig. 20 Celltype description of TLSF memory allocator component.
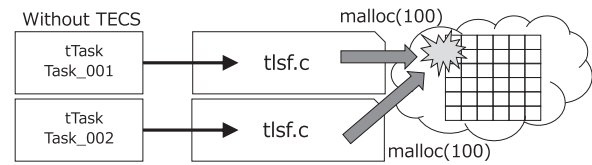

Fig. 21 TLSF before componentization.

finely in two steps. Therefore, it is a memory efficient algorithm. Fortunately, given its design, TLSF searches for memory blocks in constant time (i.e., $O(1)$).

3.2.3 Component Design of TLSF+TECS

In this subsection, we describe the component design of the TLSF memory allocator. Note that we use TECS to componentize TLSF. Further, the version of TLSF used is 2.4.6 [*4].

In **Fig. 19**, we summarize the signature used by the allocator for dynamic memory management. Particularly, this component defines the memory pool initialization function *initializeMemoryPool()*, memory allocation functions *calloc()*, *malloc()*, and *realloc()*, and the memory release function *free()*.

Next, **Fig. 20** shows the celltype description for the TLSF memory allocator component. Here, entry port *eMalloc* is connected to all components that perform memory management including *malloc*() and *free*(). Further, *[inline]* is a specifier to suggest to the TECS generator to implement as an inline function. Memory pool size is defined as an attribute, and a pointer to a memory pool is defined as a variable. Note that each component maintains its own heap area. Therefore, even when calling memory management functions simultaneously from different threads, it is possible to operate without any memory contention.

As shown in **Fig. 21**, since TLSF before componentization shares the heap area with multiple threads, if memory is allocated or released simultaneously via multiple threads, memory contention may occur in some cases, thus causing intermittent synchronization problems that can be extremely difficult to debug. In this study, TLSF is componentized using TECS, as shown in **Fig. 22**. It is possible to operate in thread safe without exclusive control because each component independently holds a heap area and manages memory within it.

Next, **Fig. 23** shows the build description of the TLSF memory

---

[*4] http://www.gii.upv.es/tlsf/main/repo
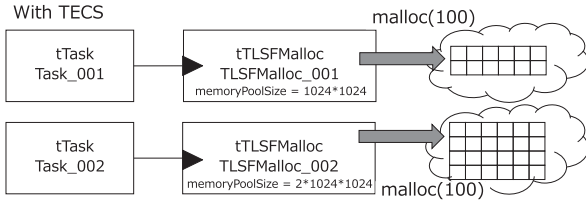
With TECS

malloc(100)

Fig. 22   TLSF after componentization.

```
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. 23   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. 24   Example of TLSF memory allocator component.

allocator component illustrated in Fig. 22 [*5]. Here, two sets of task components and TLSF components are combined. Further, each memory pool size can be configured as a variable (lines 5 and 11 in Fig. 23). **Figure 24** presents the code that actually calls functions of the TLSF memory allocator component. The use part shows a function in which the RiteVM allocates memory within the mruby on TECS framework [9], [10], which we introduced in Section 2.3 above. Line 8 calls the *free()* function of the TLSF memory allocator component. *cMalloc_* represents the name of the call port on line 2 in Fig. 23. Similarly, lines 13 and 17 call the memory allocation function. Particularly, the heap area for the *TLSFMalloc_001* component is used if the code from 24 is executed in *Task_001*; conversely, if that code is executed in *Task_002*, the heap area of the *TLSFMalloc_002* component is used. Using this approach, in component-based development using TECS, it is possible to operate with the same code without modifying the underlying C code, although the resulting cells differ.

Multiple RiteVM instances

The proposed framework uses the TLSF memory allocator for memory management within RiteVMs; however, since it is difficult to hundle multiple memory pools in the existing TLSF, if

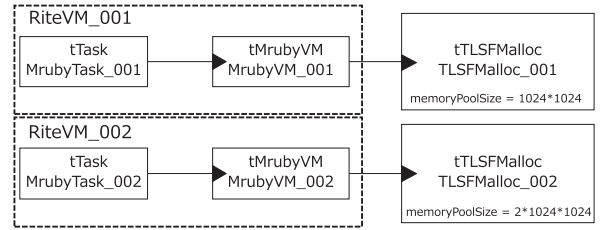*5   Other call/entry ports, attributes, and valuables are actually described, but it is omitted here for simplicity.

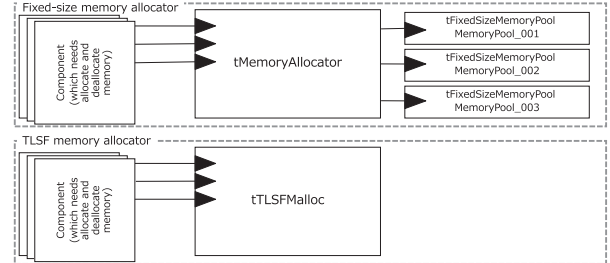**Fig. 25**   Component description of RiteVM and TLSF+TECS.

Fig. 26   Fixed-size and TLSF memory allocator components.

memory is allocated or deallocated from multiple threads, memory contention will likely occur. Here, as a RiteVM allocates and deallocates memory at high frequencies, memory contention quickly occurs when multiple RiteVMs are instantiated. Note that the TLSF components are connected to the RiteVMs to ensure that each component has its own heap area within each RiteVM, as illustrated in **Fig. 25**. Since each TLSF component maintains its own memory pool, multiple RiteVMs can be executed without the possibility of any memory contention. In Fig. 25, we observe that the first RiteVM has a heap area of 1 MB (i.e., $1,024 \times 1,024$ bytes) and the second RiteVM has a heap area of 2 MB (i.e., $2 \times 1,024 \times 1,024$ bytes). As illustrated in Fig. 25, it is easy to configure different-sized heap areas for each RiteVM. Further, each RiteVM performs incremental garbage collection (GC), and a RiteVM that has started GC does not disturb the execution of other RiteVMs in its GC execution.

Memory management for sending and receiving data

In the TCP/IP protocol stack, memory allocation and subsequent deallocation are repeated in each layer, including the TCP, IP, Ethernet, and other layers. Therefore, the role of the memory allocator is critical. The TINET+TECS framework combines all components that manage memory within the allocator. The TLSF memory allocator can execute at the same speed as that of the fixed-size memory allocator that TOPPERS/ASP3 supports as standard; further, the TLSF memory allocator can improve memory efficiency. As illustrated in **Fig. 26**, the fixed-size memory allocator prepares memory pools of different sizes and selects a memory pool whenever it is necessary. Conversely, TLSF+TECS efficiently manages memory without the need to select a memory pool. Finally, TLSF+TECS can be easily extended to TINET+TECS since TINET+TECS is a component-based system.

3.3   Use Case

In the proposed framework, applications can call TINET functions, such as specific TCP- and UDP-related functions, from mruby programs because the mruby-TECS bridge automatically

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

Fig. 27   An example of mruby application.

generates the code to link mruby and C. **Figure 27** shows an example of mruby program that transmits the value acquired from the sensor to another device or a server. In general, mruby makes it easier to develop applications than using C with the existing TINET.

For a simple application, typically only a few functions are used, with numerous unused functions. As an example, the application code shown in Fig. 27 only uses a function to send data via TCP. The proposed framework incorporates TINET+TECS and can easily customize the TCP/IP protocol stack by removing functions, such as UDP functions, functions that support only IPv4, or TCP receiving functions. As such, the proposed framework can be applied to embedded systems with strict memory constraints by removing many unused functions.

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   Improved Configurability

As shown in **Table 2**, 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 TINET+TECS

To demonstrate the low overhead of TINET+TECS, we compared its execution time and memory consumption with that of TINET. The results with TCP are shown in **Fig. 28**. 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. 10. 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, tTCPIn-

Table 1   Evaluation board environment.

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

Table 2   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)
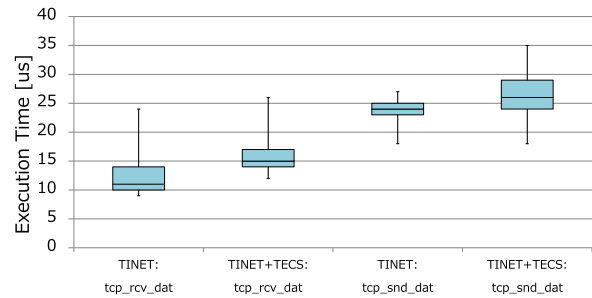

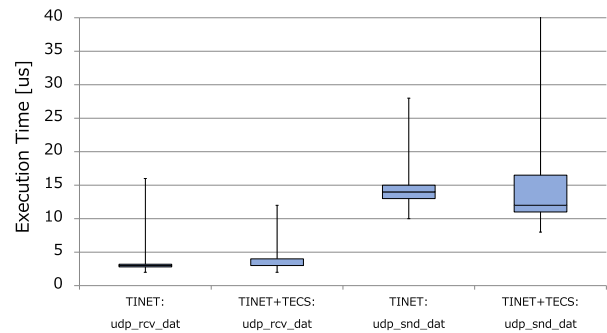**Fig. 28**   Execution times of TINET and TINET+TECS with TCP.

Fig. 29   Execution times of TINET and TINET+TECS with UDP.

Table 3   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 (TCP)	169.48 KB	5.37 KB	149.01 KB	323.96 KB
TINET+TECS (UDP)	169.26 KB	5.37 KB	149.04 KB	323.77 KB
TINET+TECS (TCP+UDP)	170.73 KB	5.37 KB	149.13 KB	325.23 KB

Including the application and kernel objects

put, tTCPCEP, and tApplication, as shown in Fig. 10. The execution time of TINET+TECS is close to that of TINET, with an overhead of about 3 us. Conversely, we evaluated the execution times with UDP as shown in **Fig. 29**. TINET+TECS can run at the same speed as TINET; therefore, the overheads of TINET+TECS are low. 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 3**. 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 compo-
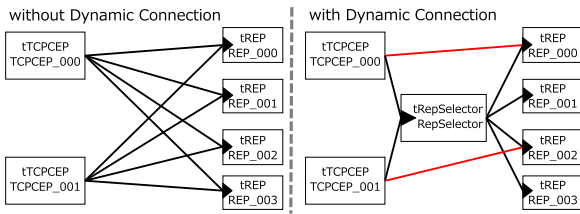
Fig. 30   Component diagrams for without/with dynamic connection.

Table 4   Memory consumption in two cases
(with/without dynamic connection).

	CEP:1 REP:1	CEP:1 REP:5	CEP:2 REP:5	CEP:5 REP:10
without	324.98 KB	325.34 KB	326.39 KB	331.68 KB
with	325.23 KB	325.32 KB	327.24 KB	330.48 KB

Table 5   CDL code lines of without/with dynamic connection.

	without	with	Diff
CEP:1 REP:1	344 lines	347 lines	−3 lines
CEP:1 REP:5	369 lines	367 lines	2 lines
CEP:2 REP:5	387 lines	382 lines	5 lines
CEP:5 REP:10	485 lines	445 lines	40 lines

nents increase memory consumption.

4.3 Dynamic Connection

Memory consumption without and with TECS dynamic connection was then evaluated. As shown in the left of **Fig. 30**, each CEP component should be statically connected to all REP components if the dynamic connection is not used. As the number of REPs increases, additional call ports of CEP are required, in turn increasing the consumption of memory. The dynamic connection reduces memory consumption because only one CEP-to-REP call port is required per CEP, as illustrated with red lines in the right paneof Fig. 30. Even if the number of REPs increases, additional call ports can be joined through the selector, instead of the CEPs.

Memory consumption of without and with dynamic connection is shown in **Table 4**. The dynamic connection case consumes the more RAM memory because, as mentioned in Section 3.1.3, call ports with *[dynamic]* are not optimized and allocated in RAM areas. However, the overall memory consumption is lower under the proposed framework.

The code lines in CDL of without and with the dynamic connection is shown in **Table 5** to demonstrate improved configurability. As the number of CEPs and REPs increases, the amount of CDL code lines to be added increases. In the left of Fig. 30, each CEP connects all REPs as shown in the upper of **Fig. 31**. In the right of Fig. 30, a CEP dynamically connects an REP, and only the selector connects all REPs as shown in the lower of Fig. 31. It is effective for software that uses many ports because the difference spreads as the number of CEPs and REPs increases.

4.4 Memory Usage of RiteVMs by TLSF+TECS

The proposed dynamic memory allocator, TLSF+TECS, executes multiple tasks without exclusive control concurrently because each TLSF component holds its own heap area. The TLSF+TECS framework provides functionality to acquire statistical information describing dynamic memory usage. Therefore, it is possible to analyze the operational status of the TLSF and

```
1  /* without Dynamic Connection */
2  cell tTCPCEP TCPCEP_000 {
3      cREP[0] = REP_000.eREP;
4      ..
5      cREP[n] = REP_00n.eREP;
6  };
7  cell tTCPCEP TCPCEP_001 {
8      cREP[0] = REP_000.eREP;
9      ..
10     cREP[n] = REP_00n.eREP;
11 };
12 ..
13 cell tTCPCEP TCPCEP_00n {
14     cREP[0] = REP_000.eREP;
15     ..
16     cREP[n] = REP_00n.eREP;
17 };
```

```
1  /* with Dynamic Connection */
2  cell tRepSelector RepSelector {
3      cREP[0] = REP_000.eREP;
4      ..
5      cREP[n] = REP_00n.eREP;
6  };
7  cell tTCPCEP TCPCEP_000 {
8      cRepSelector = RepSelector.eRepSelector;
9  };
10 cell tTCPCEP TCPCEP_001 {
11     cRepSelector = RepSelector.eRepSelector;
12 };
13 ..
14 cell tTCPCEP TCPCEP_00n {
15     cRepSelector = RepSelector.eRepSelector;
16 };
```

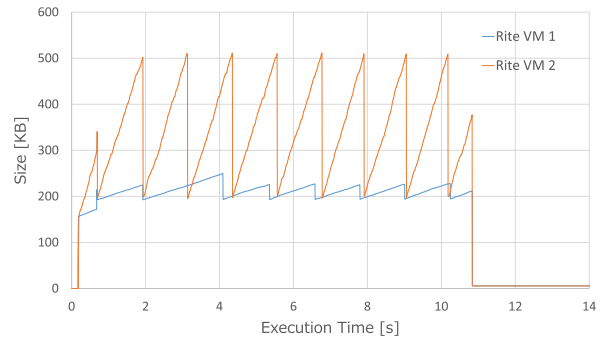**Fig. 31**   Two CDL codes (without/with dynamic connection).

Fig. 32   Memory usage of RiteVMs with TLSF+TECS.

GC on RiteVMs.

Figure 32 shows the memory usage of two RiteVMs, with presented data acquired from the statistical information available via the TLSF+TECS framework. From the figure, we observe that when the RiteVM are first activated, a large amount of memory is allocated for each RiteVM and initialization is performed within one second. Next, the applications runs for 10 s and the RiteVMs subsequently terminate. The reduction in memory usage that occurs at regular intervals is due to the GC function of the RiteVM. Here, an application that allocates tens of KB of memory runs on RiteVM 1 and an application that allocates hundreds of KB of memory runs on RiteVM 2. **Figure 33** shows the memory usage when two RiteVMs are executed in one heap area with exclusive control. In Fig. 4, two applications are finished at 10.837 s, and in Fig. 33, they are finished at 10.877 s, which demonstrates that the execution time of the applications has increased due to exclusive control. In addition to the overhead of execution time, a RiteVM is not affected by other RiteVMs since TLSF+TECS keeps heap area independently. GC on a RiteVM is performed periodically; however, there is a part that GC is not performed periodically in RiteVM 1 of Fig. 28. In Fig. 33, two RiteVMs share one heap
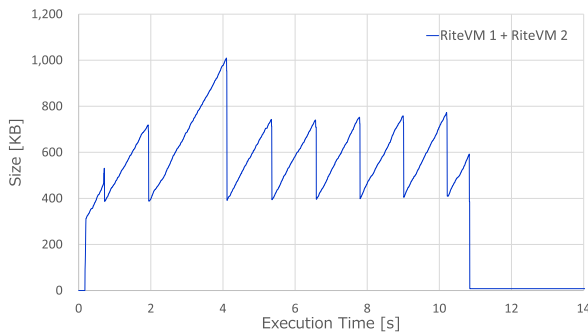
Fig. 33   Memory usage of RiteVMs with exclusive control.

area; therefore, RiteVM 2 is affected by RiteVM 1. On the other hand, in Fig. 32, RiteVM 2 is not affected by RiteVM 1.

As such, the TLSF+TECS framework provides users with the ability to visualize GC behavior and help verify its operation. Further, when the RiteVM terminates, the memory used by the RiteVM is not completely released, i.e., a few kilobytes remain. This remaining memory causes a memory leak when the number of RiteVMs increases or a RiteVM repeatedly activates and shuts down. Using the proposed environment here proves useful for detecting bugs related to memory, which in practice can be extremely difficult to detect.

5.   Related Work

To develop the software of IoT systems, several approaches have been proposed [27] such as Wireless Sensor Network (WSN) macroprogramming, Cloud-based platforms, and Model-Driven Development (MDD), General-purpose Programming Languages (GPLs).

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 [28], has been proposed. nesC/TinyOS is designed for WSN nodes with limited resources e.g., 8 KB of program memory, 512 bytes of RAM, but not supported TCP/IP implementation.

A cloud-based platform reduces development efforts by providing cloud-base APIs and high-level constructs (e.g., drag-and-drop) [29]. In addition, it offers easy deployment and evolution because the application logic is centrally located in a cloud platform. However, it is a platform-dependent design, and restricts developers in terms of functionality such as in-network aggregation or direct node-to-node communication locally. The cloud-based mruby framework, enzi Board [30], has been proposed. enzi can be developed and simulated on the Web, and the developer can download and run the program on the board.

To address the issues of development efforts and platform-dependent design, MDD has been proposed [31]. MDD provides the benefits of reusable, platform-independent, extensible design, however it needs a long development time to build MDD systems.

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. Several Open-source runtime sys-

tems for scripting languages have been proposed such as python-on-a-chip [32], the Owl system [33], eLua [34]. mruby programs on TECS can be executed approximately 100 times faster than standard mruby programs [9]. That is, mruby programs can be executed at the same speed as C, which is faster than other scripting languages.

The proposed framework can configure the TCP/IP protocol stack with minimum set compared to the other platforms. Therefore, the proposed framework can reduce memory consumption. In addition, the extended mruby on TECS supports a legacy code such as a motor driver. mruby has the same syntax as Ruby which has advantages for web application development as it uses in Rails framework [35].

6.   Conclusion

This paper presented an extended framework of mruby on TECS, including TINET+TECS and TLSF+TECS. In the proposed framework, mruby programs can call TINET+TECS functions through the mruby-TECS bridge. The development of software for IoT devices such as sensors and actuators will be more efficient due to the high productivity of mruby.

TINET+TECS is a componentized version of TINET, a compact TCP/IP protocol stack that uses TECS. It improves on TINET configurability while suppressing the overhead of componentization. Scalability is also improved because the component-based framework simplifies to add/remove and change protocols such as TCP/UDP, IPv4/IPv6, and Ethernet/PPP. This paper also presented the dynamic connection, a new TECS functionality, to enable dynamic processing while reducing memory consumption. TINET+TECS utilizes the dynamic connection to satisfy the TINET specification for supporting the static generation of CEPs and REPs. TLSF+TECS is a component-based dynamic allocator. Since the TLSF+TECS can hold its own heap area, memory contention will not occur even if memory is simultaneously allocated or released from multiple threads. TINET+TECS and TLSF+TECS can be applicable to various embedded systems because these frameworks are executed on TECS and not limited to mruby.

In addition, the RiteVMs, TINET+TECS, and TLSF+TECS are implemented as components; therefore, developers can add, remove, or reuse their functionalities easily as required. Note that our prototype system and the application programs used in the performance evaluation are all open-source and can be downloaded from our website [36]. In the future, we will support mruby libraries as mrbgems, which is an mruby distribution packaging system.

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, *Proc. 27th ACM 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, *Proc. 7th Asia-Pacific Software Engineering Conference* (*APSEC*), pp.372–379 (2000).

[4] Gössler, G. and Aştefănoaei, L.: Blaming in Component-based Real-time Systems, *Proc. 14th ACM International Conference on Embedded Software* (*EMSOFT*), pp.1–10 (2014).

[5] Bonakdarpour, B. and Kulkarni, S.S.: Compositional Verification of Fault-tolerant Real-time Programs, *Proc. 7th ACM International Conference on Embedded Software* (*EMSOFT*), 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, *Proc. 10th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing* (*ISORC*), pp.46–50 (2007).

[7] AUTOSAR: AUTOSAR (online), available from ⟨http://www.autosar.org/⟩ (accessed 2018-02-11).

[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, *Proc. 18th IEEE International Symposium on Real-Time Computing* (*ISORC*), pp.252–259 (2015).

[10] 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).

[11] Tanaka, K. and Higashi, H.: mruby – Rapid IoT Software Development, *Proc. 17th International Conference on Computational Science and Its Applications* (*ICCSA*), pp.733–742 (2017).

[12] mruby: mruby (online), available from ⟨http://mruby.org/⟩ (accessed 2018-02-11).

[13] TOPPERS Project: TINET (online), available from ⟨https://www.toppers.jp/en/tinet.html⟩ (accessed 2018-02-11).

[14] Ruby community: Ruby (online), available from ⟨https://www.ruby-lang.org/en/⟩ (accessed 2018-02-11).

[15] Kawada, T., Azumi, T., Oyama, H. and Takada, H.: Componentizing an Operating System Feature Using a TECS Plugin, *Proc. 4th IEEE International Conference on Cyber-Physical Systems, Networks, and Applications* (*CPSNA*), pp.95–99 (2016).

[16] TOPPERS Project: TOPPERS/ASP3 (online), available from ⟨https://www.toppers.jp/asp3-kernel.html⟩ (accessed 2018-02-11).

[17] Takada, H. and Sakamura, K.: μITRON for Small-Scale Embedded Systems, *IEEE Micro*, Vol.15, No.6, pp.46–54 (1995).

[18] 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).

[19] TOPPERS Project: TOPPERS/HRP2 (online), available from ⟨http://www.toppers.jp/en/hrp2-kernel.html⟩ (accessed 2018-02-11).

[20] Ishikawa, T., Azumi, T., Oyama, H. and Takada, H.: HR-TECS: Component Technology for Embedded Systems with Memory Protection, *Proc. 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing* (*ISORC*), pp.1–8 (2013).

[21] TRON Forum: ITRON TCP/IP API Specification (Ver. 2.00.00) (online), available from ⟨http://www.tron.org/wp-content/themes/dp-magjam/pdf/specifications/en_US/TEF024-S003-02.00.00_en.pdf⟩ (accessed 2018-02-11).

[22] TOPPERS Project: TOPPERS (online), available from ⟨http://www.toppers.jp/en/index.html⟩ (accessed 2018-02-11).

[23] Azumi, T., Oyama, H. and Takada, H.: Memory Allocator for Efficient Task Communications by Using RPC Channels in an Embedded Component System, *Proc. 9th IASTED International Conference on Software Engineering and Applications* (*SEA*), pp.204–209 (2008).

[24] Azumi, T., Takada, H. and Oyama, H.: Optimization of Component Connections for an Embedded Component System, *Proc. 7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing* (*EUC*), Vol.2, pp.182–188 (2009).

[25] Masmano, M., Ripoll, I., Crespo, A. and Real, J.: TLSF: A New Dynamic Memory Allocator for Real-Time Systems, *Proc. 16th Euromicro Conference on Real-Time Systems* (*ECRTS*), pp.79–88 (2004).

[26] TLSF: TLSF - Memory Allocator for Real-Time, Polytechnic University of Valencia (online), available from ⟨http://www.gii.upv.es/tlsf/⟩ (accessed 2018-02-11).

[27] Chauhan, S., Patel, P., Delicato, F.C. and Chaudhary, S.: A Development Framework for Programming Cyber-physical Systems, *Proc. 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems* (*SEsCPS*), pp.47–53 (2016).

[28] 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).

[29] Derhamy, H., Eliasson, J., Delsing, J. and Priller, P.: A survey of commercial frameworks for the Internet of Things, *Proc. 20th Conference on Emerging Technologies Factory Automation* (*ETFA*), pp.1–8 (2015).

[30] FUKUOKA CSK CORP.: enzi Board (online), available from ⟨http://enzi.cc/⟩ (accessed 2018-02-11).

[31] Kulkarni, V. and Reddy, S.: Separation of concerns in model-driven development, *IEEE Software*, Vol.20, No.5, pp.64–69 (2003).

[32] python-on-a-chip (online), available from ⟨http://code.google.com/archive/p/python-on-a-chip/⟩ (accessed 2018-02-11).

[33] Barr, T.W., Smith, R. and Rixner, S.: Design and Implementation of an Embedded Python Run-Time System, *Proc. USENIX Annual Technical Conference* (*USENIX ATC 12*), pp.297–308 (2012).

[34] eLua Project: eLua, eLua Project (online), available from ⟨http://www.eluaproject.net⟩ (accessed 2018-02-11).

[35] Ruby on Rails (online), available from ⟨http://rubyonrails.org/⟩ (accessed 2018-02-11).

[36] TOPPERS Project: TECS (online), available from ⟨http://www.toppers.jp/tecs.html⟩ (accessed 2018-02-11).

Takuro Yamamoto received his B.E. degree from Osaka University in 2016. His research interests include real-time and embedded systems.

Takuma Hara received his M.S. degree in Information Science from Nagoya University in 2013. His research interests include embedded component technology.

Takuya Ishikawa received his Ph.D. degree in Information Science from Nagoya University in 2013. He was a Researcher and then a Designated Assistant Professor at the Center for Embedded Computing Systems, Nagoya University from 2014 to 2016. His research interests include real-time scheduling theory, real-time operating systems, and embedded software platform.

Hiroshi Oyama received his Dr. Engineering degree from Gifu University in 2002. He works at Okuma Corporation as a Senior Engineer and also works at Nagoya University as a Visiting Professor. He is studying software components and programming language for embedded systems.

Hiroaki Takada is a professor at Institutes of Innovation for Future Society, Nagoya University. He is also a Professor and the Executive Director of the Center for Embedded Computing Systems (NCES), the Graduate School of Informatics, Nagoya University. He received his Ph.D. degree in Information Science from The University of Tokyo in 1996. He was a Research Associate at The University of Tokyo from 1989 to 1997, and was a Lecturer and then an Associate Professor at Toyohashi University of Technology from 1998 to 2003. His research interests include real-time operating systems, real-time scheduling theory, and embedded system design. He is a member of ACM, IEEE, IEICE, JSSST, and JSAE.

Takuya Azumi is an Associate Professor at the Graduate School of Science and Engineering, Saitama University. He received his Ph.D. degree from the Graduate School of Information Science, Nagoya University. From 2008 to 2010, he was under the research fellowship for young scientists for Japan Society for the Promotion of Science. From 2010 to 2014, he was an Assistant Professor at the College of Information Science and Engineering, Ritsumeikan University. From 2014 to 2018, he was an Assistant Professor at the Graduate School of Engineering Science, Osaka University. His research interests include real-time operating systems and component-based development. He is a member of IEEE, ACM, IPSJ, and IEICE.