Regular Paper

mROS: A Lightweight Runtime Environment of ROS 1 nodes for Embedded Devices

Hideki Takase[1,2,a)] Tomoya Mori[1] Kazuyoshi Takagi[1,†1] Naofumi Takagi[1]

Received: May 22, 2019, Accepted: November 7, 2019

Abstract: The Robot Operating System (ROS) has attracted attention as a design platform for robot software development. One of the problems of ROS is that it is necessary to employ high-performance and power-hunger devices since ROS requires a Linux environment for operation. This paper proposes a novel solution called mROS, which is a lightweight runtime environment of ROS nodes, to execute robot software components on mid-range embedded devices. mROS consists of a real-time operating system (RTOS) and a TCP/IP protocol stack to provide a tiny ROS communication library. It provides connectivity from the edge node to the host and other nodes through the native ROS protocol. Additionally, we design mROS APIs that are compatible with ROS 1. Therefore, native ROS nodes can be ported from Linux-based systems to RTOS-based systems as mROS nodes. Experimental results confirmed that mROS meets the performance requirement for practical applications. Moreover, we showed the size of the library constituting mROS is small for target embedded devices. We further conducted a case study to validate the portability of mROS from ROS nodes. Our work is expected to contribute to the power saving and real-time performance enhancement of mobile robot systems.

Keywords: robot operating systems, real-time operating systems, distributed systems, TCP/IP protocol

1. Introduction

In the sophisticated information society era, demand for mobile robot systems has been increasing to support social life in various situations, such as care support, disaster relief. Unlike the industrial robots that have been developed so far, these robots must operate using energy from an internal battery, rather than an external power supply. The robot system should provide advanced and multi-functional services with limited power supply. In order to improve the quality of service provided by such mobile robots, it is necessary to achieve multiple functions with limited power consumption.

Recently, the Robot Operating System (ROS) [19] has attracted attention as a design platform to accelerate the productivity of robot software. ROS aims to realize the component-based development of robot systems. A software component is expressed as a node, and a robot system is realized by combining a plurality of nodes. ROS also serves as middleware, providing a communication layer between nodes. The communication is based on a publish/subscribe messaging model that identifies data to be transmitted and received via topics. One of the major reasons to employ ROS is that about 3,000 open source packages are available. Developers can use them to achieve the desired robot systems.

Since the implementation of ROS that is widely used at present assumes operation in a Linux/Ubuntu environment [*1], it is necessary to adopt a high-performance and power-hunger device. Therefore, it is difficult to achieve power saving in a robot system using this ROS. Furthermore, it is also difficult to enhance the real-time performance of robot systems. Here, real-time performance means that respective tasks can be completed within a certain time. Although some existing methods [1], [2], [3], [4], [7], [8], [12], [13], [15], [16], [17], [20], [21], [22] offer the adoption of embedded systems for ROS, it is not possible to use them to port open source ROS packages directly on the embedded devices.

In this paper, we propose a lightweight runtime environment of ROS nodes called mROS [*2], which is designed to be operated on an embedded device having a mid-range micro-processor. Because the power consumption of such embedded devices is low, their usage can reduce power consumption in the robot systems.

mROS provides a communication library to the host device, which is operated on native ROS/Linux. In order to allow the program executed on the embedded device to behave as an ROS node, we support two communication protocols that are compliant with the ROS to transfer data with the host system. We design APIs of the mROS communication library to be the same as those used in the native ROS program. mROS consists of a real-time operating systems (RTOS) and an embedded TCP/IP protocol stack. Currently, we are implementing mROS for GR-PEACH [5] using TOPPERS/ASP kernel [18] and lwIP protocol stack [10].

[1] Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
[2] PRESTO Program, Japan Science and Technology Agency, Kawaguchi, Saitama 332–0012, Japan
[†1] Presently with Graduate School of Engineering, Mie University
[a)] takase@i.kyoto-u.ac.jp

[*1] In this paper, we discuss ROS 1 unless otherwise noted.
[*2] The name of mROS is responsive to mruby, that is the implementation of Ruby for embedded systems.

The contributions of our work are summarized as follows;

- Providing such a lightweight runtime environment can makes it easier to employ embedded devices in robot systems that utilize ROS.
- Given that it is possible to design applications with a native ROS manners, the learning cost required for the transition of design development can be reduced. This also leads to improve the design productivity when utilizing existing packages for adopting mROS.
- We can utilize them for designing a system that guarantees real-time performance, because it is also possible to use the API provided by the mbed library and the TOPPERS/ASP kernel. real-time performance can be ensured easily. In addition, Our work might contribute to facilitate the use of low-power embedded devices that could not previously be incorporated into ROS systems.

The reminder of this paper is organized as follows. Section 2 introduces ROS as the preliminary of this work. Section 3 describes our proposal in detail. Section 4 shows the evaluation results of mROS. Section 5 demonstrates the case study with mROS for the distributed robot system. Section 6 introduces related works. Finally, Section 7 concludes this paper and identifies directions of future works.

2. Robot Operating System (ROS)

ROS [19] is a development platform for robot systems started by Willow Garage and Stanford University in 2009. Currently, Open Robotics holds the lead in development for the open source community. ROS has been developed for enhancing the productivity and reusability of robot systems. Thus far, about 3,000 open source packages that comply with ROS are published and available.

In ROS, a program is expressed as a node, and a robot system is realized by combining nodes to realize the component-based development process. The communication layer of ROS is provided as middleware of Linux/Ubuntu. Therefore, it is possible to separate an application node that executes a complicated algorithm from a device driver node that controls hardware. Robot hardware often varies depending on the purpose of use and developer preference. Even when using different hardware, it is possible to communicate via the ROS communication layer by providing a node that serves a corresponding device driver. In addition, by dividing and realizing functions by nodes, debugging and testing of the program for each node becomes easy.

2.1 Communication Model

There are two models for communication between nodes provided on ROS. One is the publish/subscribe messaging model, which is mainly addressed in this work. It classifies data types according to topic and transfers data between the publisher node and the subscriber node. The other is the server/client messaging model, which makes requests for data and responds to such requests.

Figure 1 shows the publish/subscribe messaging model with ROS nodes and topics. Note that words publish and subscribe are often abbreviated as pub and sub, respectively. Each subscriber node obtains data from publisher nodes. It is possible to retrieve data by designating the topic name for performing point-to-multipoint communication. By matching the name of topic to be published by nodes corresponding to I/O devices of different hardware, nodes can be used without modification to different robot hardware. In addition, the component set that is pair of node and hardware can easily be changed to the other if the name of topic is changed. When executing the robot software, the master node called `roscore` manages the namespace of each node for the communication of nodes in the robot systems.

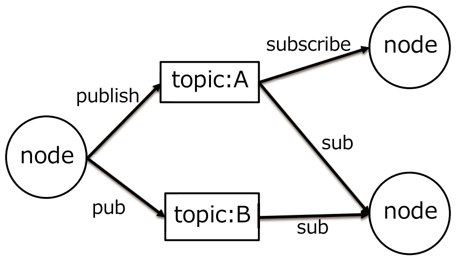


Fig. 1 Publish/Subscribe messaging model in ROS.

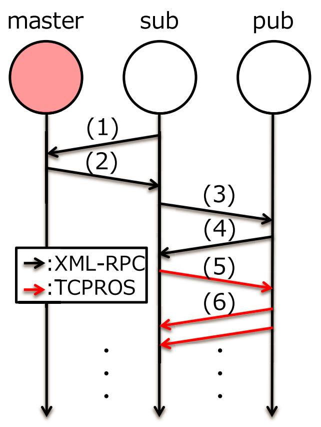


Fig. 2 Publish/Subscribe communication flow in ROS.

2.2 Communication Flow

There are two protocols to realize ROS communication. `XML-RPC` encodes communication data in the XML format and transports the encoded data via HTTP as a procedure call [14]. ROS uses its protocol for procedure calls between the ROS master and nodes. `TCPROS` is an ROS-specific protocol for ROS data communication through TCP/IP communication. In the `TCPROS` protocol, a unique field is defined corresponding to the setting of the node in order to identify the data content. The data corresponding to each field is given a bit string of 4 B length as one message. When a publish/subscribe connection is established, data transfer can be performed continuously after the connection header of a node is transmitted.

Figure 2 shows the communication flow of the publish/subscribe messaging model for ROS communication.

(1) The subscribe node registers its node name and topic name.
(2) The ROS Master tells XML-RPC port number of publisher node.
(3) The subscribe node sends subscribe request.
(4) The publish node tells TCPROS port number.

Table 1 Measured intra-device communication between ROS nodes [us].

Method	Average	Worst
TCP	4661.59	9785.13
TCP + shared_ptr	2090.04	4421.55
Nodelet	1006.20	6088.76
Nodelet + shared_ptr	50.35	507.54

(5) TCPROS port is connected by sending TCPROS header.
(6) The publish node sends TCPROS header.
(7) Then, the publish node sends data to the topic.

2.3 Intra-Device Communication Method

In ROS, the inter-node communication of ROS is performed using the TCP/IP protocol. Therefore, the communication overhead increases when data communication is performed on the same device via the TCP socket. To address this issue, a function called Nodelet [9] is provided. It uses shared memory for intra-device data communication between nodes. Therefore, it is possible to realize efficient communication only by memory copying without performing serialization/deserialization process.

We measured the performance of intra-device communication for TCP socket and Nodelet by using the package published in Ref. [21]. As the measurement environment, we used NEC's LAVIE Hybryd ZERO which has Intel Core-i7 2.4 GHz and 16 GB memory, Ubuntu 14.04 LTS for the host OS and indigo for the ROS distribution. In ROS, smart pointer (`boost::shared_ptr`) can be used to provide data sharing functions within the node. **Table 1** shows the average communication time and the worst case communication time in microseconds when 1,000 times of publish/subscribe communication was performed for 3 MB of data. Compared with communication using ordinary TCP socket, Nodelet can achieve about 4 times faster node-to-node communication. Also, when combined with a method of passing data by reference, it is possible to achieve 800 times faster. Nodelet is therefore highly effective to speed up node communication in the device.

2.4 Provided APIs

ROS provides APIs to describe the application as a ROS client library. Developers can construct ROS nodes by defining behavior of nodes using provided APIs. APIs for C++ are as follows.

`init(node_name)` It is the initialization function, that takes the name of the node and the command line argument as arguments.

`NodeHandle` It creates a handler to the node. The node is initialized when the first handler is created and the resources used by the node are released when the last handler is destroyed.

`NodeHandle::advertise<topic_type>(topic_name)` It creates the publisher node and registers it to the ROS master. It takes the topic name to be published, the data type of the topic and the buffer size as arguments. It returns `ros::Publisher` object as the return value.

`Publisher::publish(message_data)` The node that has been registered by `advertise()` publishes topic data using its method of the `ros::Publisher` object. It takes the message data as an argument.

`NodeHandle::subscribe(topic_name, Call_back)` It create the subscriber node and registers it in the ROS master. It takes the topic name to subscribe, buffer size and callback function as arguments. In the argument of the registered callback function, it is necessary to take a pointer indicating the destination where the topic is held.

3. mROS

This section describes details of mROS which enables operating ROS nodes on embedded devices in a lightweight runtime environment. The first subsection describes requirements and the development goals of our work. The second subsection describes the software structure of mROS. Then, we propose the communication method provided by the mROS communication library. Finally, we introduce the programming model supported by mROS.

3.1 Requirements and Goals

In this research, we target a distributed robot system that consists of the host device, on which a native ROS is operated in Linux, and edge devices on embedded devices. The ROS master is operated on the host device. ROS nodes are executed on both the host and edge devices. They communicate with each other for data transfer via topics. We design mROS for the purpose of power saving and ensuring real-time performance. In addition, by providing an application to be executed by mROS with the ROS programming model, we aim to enable the execution of the ROS open source package as it is.

We target mid-range class embedded devices for achieving low-power computing. This means that RTOS and TCP/IP protocol stack can be operated on these devices; however, Linux cannot be used since they do not have high-performance control units such as the MMU (Memory Management Unit).

A program executed on the embedded device behaves as an ROS node. To execute programs of ROS nodes onto embedded devices, it is necessary to provide a communication library for embedded devices that supports ROS communication. For native ROS in a general-purpose device, data communication between nodes is performed via a communication layer using the TCP/IP protocol stack provided by Linux. Therefore, in embedded devices equipped with mROS, it is necessary to employ a TCP/IP protocol stack that can operate without Linux for communicating with ROS. In addition, it is necessary to manage the communication process as well as the program resources on the embedded device. We realize these functions by employing RTOS. The functions of RTOS makes it possible to ensure the real-time performance for ROS nodes executed on mROS.

We are planning to utilize embedded devices in the distributed robot system, which consists of a host device with Linux and embedded devices with mROS running ROS nodes as edge terminals. In distributed robot systems, applications to be executed as edge computing devices are assumed to target processing based on inputs of sensors or cameras, and controlling actuators. Assuming that the data to be published is an image file from the camera in the QVGA format, we set the goal of publishing approximately 512 KB of data at 100 ms intervals. In contrast, assuming that the data to be subscribed is mainly the control instruction data set, the size of subscription data is set to be a maximum of 1 KB. To guarantee the subscription of important data such as control commands at high speed, we set the goal of subscribing less than 1 KB in 1 ms. In addition, because it is desirable that the environment of mROS be as lightweight as possible, we aim to realize mROS in an environment with a memory of 10 MB at most.

3.2 Software Structure

Figure 3 shows software structure of mROS. We employ RTOS and TCP/IP protocol stack in the mROS environment for providing the mROS communication library.

As RTOS is responsible for scheduling communication processes and managing program resources, we adopt TOPPERS/ASP kernel [18], which is published by TOPPERS project in Japan and complied to the μITRON specification. TOPPERS/ASP kernel provides several resources such as tasks, semaphores and data queue. The reason why we employ TOPPERS kernel is that it is suitable for embedded systems, because TOPPERS/ASP kernel statically generates these resources. In addition, one of the main advantages for TOPPERS kernel compared to other kernels is its higher quality of source code and the expandability of applied area. TOPPERS project has published various kernels, such as a multi-core version, an automotive version, and an enhanced version for memory protection.

As the TCP/IP protocol stack, we adopt lwIP [10], which is included in the Arm mbed library. Because lwIP is a lightweight stack that is designed for embedded devices and is widely used as open source, we think that it is suitable for mROS. The mbed library includes unified device drivers of various devices and peripherals for Arm processors. In addition, because various libraries are published as open source, program implementation can be expected to be easy.

As shown in Fig. 3, our mROS communication library is located above RTOS and TCP/IP protocol stack layers. This means that our library does not depend on the hardware platform. In other words, mROS can be ported on embedded devices where TOPPERS kernel and lwIP are operated.

3.3 Communication Library

Our mROS library offers the communication function to ROS nodes on the host devices. mROS enables programs on the edge device to behave as an ROS node by using the functionality of the mROS communication library.

mROS supports the publish/subscribe messaging model of ROS. To realize the communication flow, as shown in Fig. 2, we implement following functions in the mROS communication library.

- data publication
- data subscription
- procedure call between the ROS master and other nodes
- acceptance of procedure call from nodes to the master

mROS receives and responds as the slave in XML-RPC communication, and it creates data in the XML format for transportation via HTTP as the master. When a connection between nodes is established, the TCPROS function in mROS generates a connection header in the TCPROS format and transports it to the publish

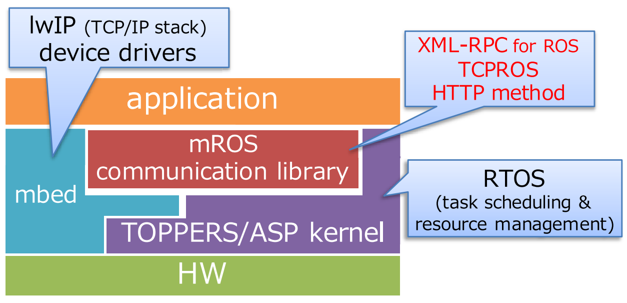


Fig. 3 Software structure of mROS.

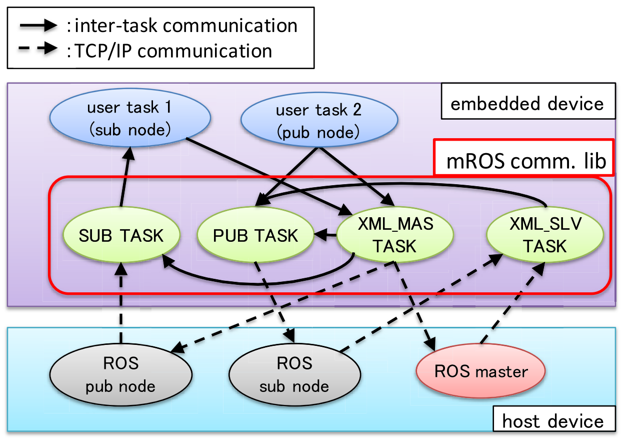


Fig. 4 Task structure in communication library.

node. Subsequently, topic data are encoded and transported to subscribe nodes, and the data received from the publish node is decoded. In ROS, data communication is performed using either TCP or UDP, but mROS currently supports only TCP.

3.3.1 Task Structure

In TOPPERS/ASP kernel, application resources are managed as task units. Therefore, the functions provided in the mROS communication library are constructed by multiple tasks. The mROS communication library consists of four tasks: `SUB`, `PUB`, `XML_MAS` and `XML_SLV TASK`s. `SUB TASK` performs the data subscription from publish nodes, such as a node on the host device. `PUB TASK` publishes data to the subscription tasks on the host device. `XML_MAS` and `XML_SLV TASK`s attain the behavior that is compliant with `XML-RPC` protocol. These tasks are automatically provided as system tasks according to the user task description by including the `mros.cfg` file, which is the TOPPERS configuration file of the mROS communication library. **Figure 4** shows the structure and the data flow among system tasks, user tasks, and external ROS nodes. In a user task, designers can describe a program using native ROS APIs. It is possible to execute multiple nodes on the same device by implementing multiple user tasks. So, `user task 1` and `2` are designed as TOPPERS tasks, but are also executed as ROS nodes, such as `sub` and `pub nodes` in Fig. 4. In the mROS communication library, ROS-compliant APIs are realized by calling each task. ROS APIs can be executed on the embedded device without any modification.

In the mROS communication library, the data queue and shared memory specified by μITRON are utilized for communication between tasks. The size of a data queue in mROS is set to 32 bits, and we call it a message ID. The message ID is composed of an

8-bit node ID and 24-bit data length. The node ID is a key value for referring to the node information. The data length indicates the data size stored in the shared memory. Because the message ID has a data length of 24 bits, the maximum data size for commuication is 16 MB. Considering the requirement of the edge terminal in the system configuration, we assume that the maximum data size to be published is approximately 512 KB (image data in the QVGA format) and to be subscribed is 1 KB (e.g., control command).

The node information is expressed as a node structure. The information of the nodes about each topic is stored as a node list in vector form. The node list is managed as global variables that can be referenced from all system tasks. Information on each node includes the node type (subscriber or publisher), node name, topic name to be handled, node ID, TCP socket, node URI, IP address of the correspondent node, port number, field information required for TCPROS, and a pointer to the callback function. In mROS, a node object is generated in each case when a node becomes a publisher or a subscriber for a plurality of topics. Data communication between tasks is performed by storing the data body encoded in XML in the shared memory, notifying other tasks of the message ID in the data queue, and extraction of the data from the shared memory by the receiving task. At this time, the shared memory area is statically allocated according to the inter-task communication.

3.3.2 Communication Method to External Nodes

mROS allows describing the user task by using ROS APIs. Functions corresponding to each ROS API are provided by the mROS communication library. Implementations of ROS-compliant APIs in the mROS communication library are as follows.

`advertise()` registers a node on mROS as a publisher to the ROS. **Figure 5** shows the execution flow of `advertise()` [*3].

(1) `user task` notifies `XML_MAS TASK` of the initialization request of the topic name for subscription to via the data queue. `XML_MAS TASK` assigns the node ID to `user task` and registers it in the node list of mROS.

(2) `XML_MAS TASK` generates the XML header and sends it to `ROS master`.

(3) `ROS master` replies with the result of registration through the XML-RPC communication.

(4) `XML_MAS TASK` notifies `PUB TASK` of the message ID for initialization via the data queue. `PUB TASK` performs the corresponding initialization process and generates the object of the TCP socket for data publication. When the initialization of the publisher task is completed, `XML_SLV TASK` starts the acceptance of the connection request from the external `sub node`.

(5) When a topic request from `sub node` occurs, `XML_SLV TASK`, which is periodically executed, returns the port number of `PUB TASK` as the response.

(6) `XML_SLV TASK` sends the arrival request to `PUB TASK`.

(7) `PUB TASK` accepts the TCPROS connection and exchanges TCPROS connection headers to establish the connection for the TCP socket of data subscription.

`publish()` is the member function of `Publisher` structure that is registered by `advertise()`. It takes the publication data as the argument. **Figure 6** shows the flow of `publish()`.

(1) `user task` writes the publication data in the shared memory of `publish()`.

(2) The message ID is generated from the node ID and length of the publication data and sent to `PUB TASK`.

(3) `PUB TASK` receives the message ID via the data queue and encodes it with the TCPROS protocol.

(4) `PUB TASK` searches for the corresponding TCP socket from the node list according to the node ID and publishes the data in the external `sub node`.

`subscribe()` takes the topic name to be subscribed and callback function as arguments. **Figure 7** shows the execution flow of `subscribe()`.

(1) `user task` notifies the initialization request to `XML_MAS TASK` in `subscribe()`.

(2) `XML_MAS TASK` generates the XML header and sends it to `ROS master` through the XML-RPC protocol.

(3) `ROS master` replies with the result of registration and sends the URI of the external `pub node`.

(4) `XML_MAS TASK` notifies `SUB TASK` of the initialization of `user task`.

(5) `SUB TASK` sends the request for subscribing the topic to `XML_MAS TASK` after the initialization has completed.

(6) `XML_MAS TASK` notifies `SUB TASK` about the port number of the external `pub node`, which is obtained from the subscription request.

(7) The TCPROS connection header is generated for the subscription request and exchanged between `SUB TASK` and `pub node` to establish the connection of socket.

`SUB TASK` is periodically operated to subscribe to the topic.

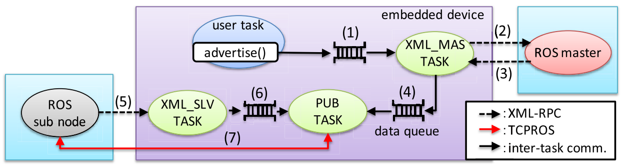


Fig. 5 Execution flow of `advertise()`.

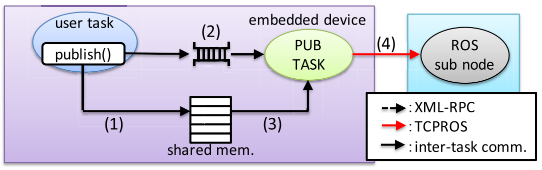


Fig. 6 Execution flow of `publish()`.

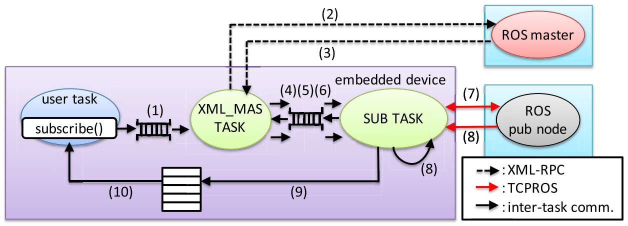


Fig. 7 Execution flow of `subscribe()`.

*3 Note that series of vertically long rectangles and one of horizontally long rectangles in Figs. 5, 6, 7, 8 mean the data queue and shared memory.

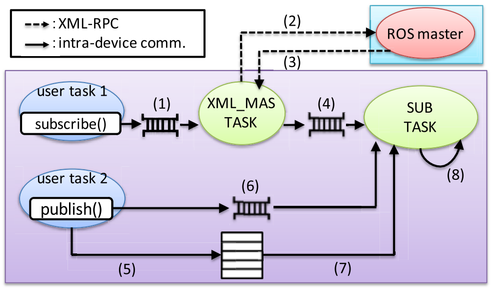


Fig. 8 Flow of intra-device communication.

The following are executed when topic data have arrived:

(8) The callback is activated in the context of `SUB TASK`.

(9) `SUB TASK` writes the return value of the callback function in the shared memory.

(10) `user task` obtains the return value of the callback function from the shared memory.

3.3.3 Communication Method of Intra-Device Nodes

In mid-range embedded devices targeted by mROS, the performance and memory resources of the microprocessor are considered sufficient to execute multiple nodes. For example, in addition to a node from which to detect data from the sensor, a node that publishes to the host device after processing its data can be implemented on the same embedded device. In such a case, rather than publishing the sensor value as it is in the host device, the amount of communication may be reduced by data processing. Therefore, we think that it is worthwhile to provide an intra-device communication method for improving communication within a device in mROS. In fact, we suggest that expediting the node communication in the device is highly effective.

We further design an intra-device communication method for ROS nodes. We employ the shared memory for data communication between nodes. Whether data publication is performed in the subscriber node in the device or not is designed to be judged in the context of the user task.

Figure 8 shows the execution flow of the intra-device communication method. In this figure, user task 1 is executed as the subscribe node, and user task 2 is the publish node.

(1) When `user task 1` executes `subscribe()`, `XML_MAS TASK` is notified of the initialization request of its task.

(2) `XML_MAS TASK` generates the XML header and sends it to `ROS master`.

(3) `ROS master` replies with the result of registration via XML-RPC communication, and sends the URI of the publish node. In `XML_MAS TASK`, the IP address and port number of the publish node are expressed from the received URI.

(4) `XML_MAS TASK` notifies `SUB TASK` of the initialization of the subscribe node. `SUB TASK` judges whether the publish node is on the same device by comparing its IP address with the notified IP address. If the publish node is on the same device, `SUB TASK` appends the node information to the node list of mROS. Note that the request for the topic subscription is not performed.

(5) When `user task 2` publishes data by `publish()`, `PUB TASK` writes corresponding data in the shared memory.

(6) In the process of `publish()`, `PUB TASK` judges whether its address is in the device from the node list. If the publication address is in the device, mROS generates the message ID and sends the data queue of `SUB TASK`.

(7) When `SUB TASK` receives the arrival of the topic, `SUB TASK` reads the data from the shared memory.

(8) The callback function is activated in the `SUB TASK`.

If there is an external subscribe node on the same topic, mROS also generates the message ID for the publish task and notifies the publication to the external device of the publication. These processes are executed in the back end through the API. Therefore, the same function can be used for the ROS node in the description of a user task. Consequently, the porting of nodes between devices equipped with the mROS environment is easy.

```
#include "../mros-lib/ros.h"
#include "user.h"
void user_function(int argc, char** argv){
  ros::init(argc, argv, "mros_node");
  ros::NodeHandle n;
  ros::Publisher chatter_pub =
              n.advertise ("mros_msg", 1);
  string msg;
  while (1) {
    chatter_pub.publish(msg.c_str());
  }
}
```

Fig. 9 Example of `user.cpp`.

```
INCLUDE("../mros-lib/mros.cfg ");
#include "../mros-lib/mros.h"
#include "user.h"
CRE_TSK (USER_TASK, {TA_NULL, 0, user_function ,
    MROS_USR_TASK_PRI, MROS_PUB_STACK_SIZE, NULL });
```

Fig. 10 Example of `user.cfg`.

3.4 Programming Model

In mROS, an application that communicates with the ROS system can be designed by using the API provided by the mROS communication library. APIs of mROS are defined with the same name as ROS APIs which were introduced in Section 2.4. Therefore, developers can describe the mROS application with a knowledge of ROS programming. We think that it is possible to port existing open source ROS packages onto embedded devices which employs mROS.

Figure 9 shows the example of ROS node description in mROS. By including `mros-lib/ros.h`, developers can use APIs provided by mROS communication library. This example expresses the publish node and the publication of `msg` string value. Also, a configuration file is required to generate tasks to execute ROS nodes. **Figure 10** shows the description of the configuration file required as the specification of the TOPPERS kernel. We provide a template like Fig. 10 for mROS user application. Therefore, the single task configuration file can be realized only by rewriting the function name to be executed as a task. Arguments of `CRE_TSK()`, that indicates the creation of task, should be set as default, but it is also possible for users to specify them.

Meanwhile, it is also possible to realize multitasking of applications by using TOPPERS API. This makes it possible to utilize the characteristics of the conventional embedded system, so that real-time performance is guaranteed. Applications of various devices can also be described by utilizing arm mbed library.

4. Evaluation

4.1 Environmental Setup

In this work, we implemented mROS onto the Renesas GR-PEACH board [5], which is adopted to TOPPERS/ASP kernel and Arm mbed library. To setup the evaluation environment of distributed robot systems, we used GR-PEACH equipped with mROS as the edge device and NEC's LAVIE HybrydZERO with ROS as the host device. GR-PEACH has an RZ/A1H 400 MHz microprocessor and 10 MB internal RAM. LAVIE HybrydZERO has an Intel Core-i7 2.4 GHz processor and 16 GB memory. We used Ubuntu 14.04 LTS as the host OS and indigo as the ROS distribution. We assumed that the host and the edge are connected by a wired LAN cable via a router in the same local network

4.2 Communication Performance

For the communication time, we evaluate the execution time of data publication and data subscription from the ROS node on mROS to the node on the host system. We used `get_utm()`, which is a microsecond precision measurement API provided by TOPPERS kernel.

4.2.1 Execution Time of `publish()`

We measured the execution time of `publish()` on mROS, which includes the execution of the mROS library and lwIP protocol, and the communication of the packet through the network. We varied the data size from 2 B to 512 KB in a power of 2. **Figure 11** shows the average execution time for 200 executions. We found that publication of less than 16 KB of data can be performed in less than 1 ms. In addition, there is no significant difference between the worst execution time and the average execution time. Small data such as data derived from sensors can be published at high speed. We also found that the execution time to publish 512 KB of data is less than 100 ms. Therefore, the publication of image data from a camera, which is expected to be used in edge devices equipped with mROS, can be realized at 10 fps. According to Ref. [11], a characteristic in ROS communication is that the communication time becomes large for large data because packet division occurs at 16 KB. We observed similar characteristics in mROS.

4.2.2 Execution Time of `subscribe()`

In this experiment, we measured the execution time of `subscribe()` from the loading of data into the socket buffer of lwIP to the start of the callback function of the subscribe node. We varied the data size from 1 B to 256 KB in powers of 2. **Figure 12** shows the average execution time for 200 executions. We found that subscription of less than 2 KB of data and the corresponding callback function call can be performed in less than 0.1 ms. However, the execution time sharply increases as the data size becomes larger than 4 KB because large data cannot be received in one reception loop.

4.2.3 Evaluation of Intra-Device Communication

We also evaluate the performance of the intra-device communication method. **Figure 13** shows the execution time of `publish()` to the same device with the intra-device communication method and compares it with the corresponding execution time for publishing to the host device. We found that the execution time of data publication to the same device is much smaller regardless of the data size. The execution time for publishing 4 KB data is less than 100 us. Therefore, our intra-device communication method can contribute to the reduction of the execution time of node communication. In addition, we found a linear correlation between the execution time and data size. This implies that the execution time can be estimated and real-time performance can be guaranteed easily.

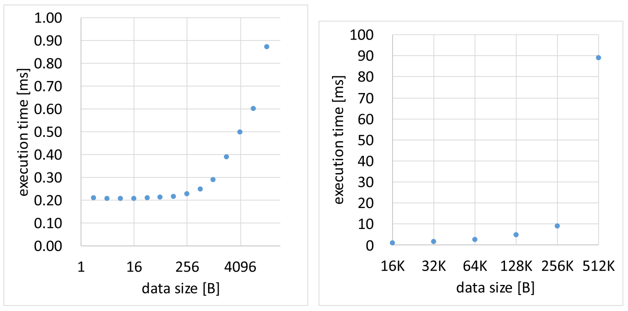


Fig. 11 Execution time of `publish()` on mROS.

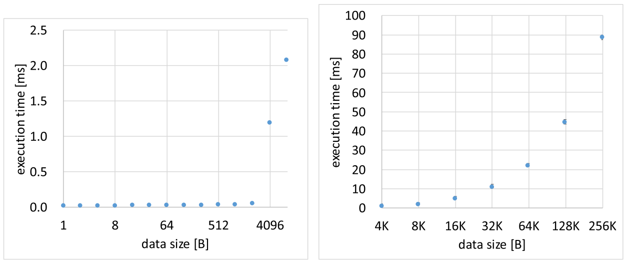


Fig. 12 Execution time of `subscribe()` on mROS.

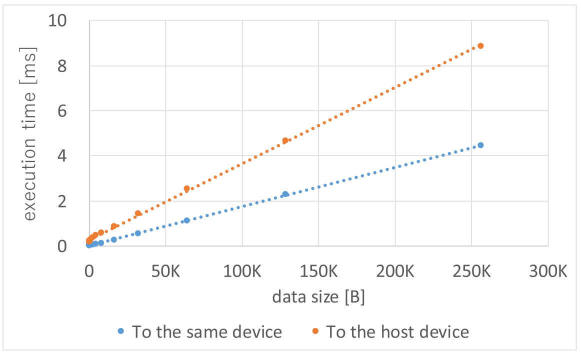


Fig. 13 Execution time of `publish()` with intra-device communication method.

Table 2 Size of runtime environment [B].

Name	text	data	bss	dec
kernel.a	99,676	0	16,408	116,084
libmbed.a	264,477	52,940	45,711	363,128
libmros.a	57,950	28	2,097,310	2,155,288
Total	422,103	52,968	2,159,429	2,634,500

4.3 Size of Runtime Environment

Table 2 lists the result for the mROS size. We used `size` command on `kernel.a`, which is a library file of TOPPERS/ASP kernel; `libmbed.a`, which is an Arm mbed library; and `libmros.a`, which is an mROS communication library.

The size of the mROS environment was approximately 2.6 MB, which means that we can implement a sufficiently lightweight runtime environment. The reason why the bss section of `libmros.a` became very large is that mROS statically assigns the shared memory capacity for inter-task communication. The current implementation of the shared memory is set to 2 MB. Note that it is possible to make the size smaller by adjusting the size of the shared memory according to the design choice.

4.4 Discussion about Power Consumption

This subsection discusses the effect on power consumption for mROS. In general, one of the most popular choice in the development of robot system is surely Raspberry Pi. We think this is no more an embedded systems since it has a high-performance computing resources and native versions of Linux and ROS can be operated on it. Also, it is employed as not only edge devices but also the central computer on distributed robot systems. According to a well-known article published on the web [*4], the amounts of current of the entire board at high load are said to be 730 mA for Raspberry Pi 3B and 980 mA for 3B+. On the other hand, the amount of current of GR-PEACH is said to be 165 mA [*5]. In addition, typical embedded processors, such as the RZ/A1H [6] which is mounted on GR-PEACH, have some low power modes in order to save power consumption. Of course we know that these numerical comparison and discussion are nonsense, we would like to suggest that mROS on mid-range embedded devices can contribute to power savings of distributed robot systems.

5. Case Study

This section demonstrates the case study of mROS to discuss the usefulness of our work.

5.1 Target System

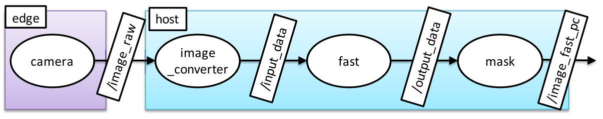
We developed a feature point detection system of an image file obtained from a CMOS camera by using an existing ROS package. We constructed its distributed system by using GR-PEACH with mROS as the edge and LAVIE HybridZERO as the host device which is proposed in Section 3.3.3.

Figure 14 shows the structure of the target system, which consists of 4 ROS nodes. `camera` node is the device driver for the camera to obtain the image file. We used the GR-PEACH AUDIO CAMERA shield for the CMOS camera. The device driver of the camera was implemented using the mbed library. `camera` node publishes the data of obtained image to `/image_raw` topic. `image_converter` node subscribes to the image and compresses it by using the OpenCV library. `fast` node detects feature points, following which `mask` node performs mask processing to create grayscale image data. Finally, the processing result is published to `/image_fast_pc`.

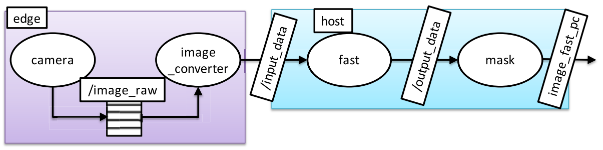
5.2 Implementation Result

Figure 14 (a) shows the case where only `camera` node is executed on the edge device. In order to discuss the portability that is contributed by mROS, we attempted to port the code of `image_converter` node to the edge device, as shown in Fig. 14 (b). The original ROS package includes the `cv_bridge`

[*4] http://www.pidramble.com/wiki/benchmarks/power-consumption
[*5] https://os.mbed.com/forum/team-886-GR-PEACH_producer_meeting-community/topic/5432/



(a) System that one node is executed on edge device.



(b) System that two node is executed on edge device.

Fig. 14 Organization of the distributed system.

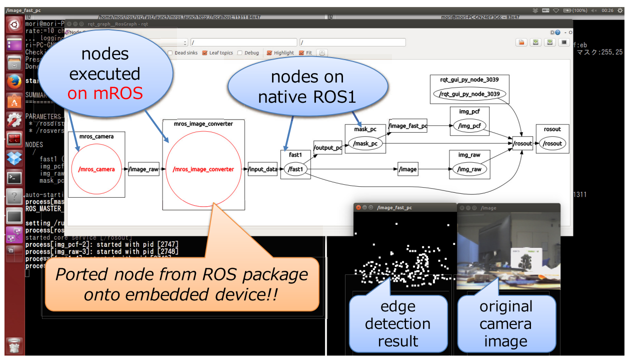


Fig. 15 Snapshot of developed systems.

library, which mutually converts the image object by the OpenCV library to the image object handled by the ROS. However, since `cv_bridge` library uses the function of dynamically generating threads, it could not be ported to the mROS environment. Therefore, we implemented the function to convert image objects for image compression by the OpenCV library. The communication between `camera` and `image_converter` nodes is performed on shared memory with our intra-device communication method.

We confirmed that the same function as the original ROS package could be realized on the distributed robot system. **Figure 15** shows a snapshot of the developed system, which includes the image obtained by extracting feature points and a graph visualization of the ROS component obtained with `rqt_graph`. Nodes indicated by the red circle in `rqt_graph` are those running on mROS. Therefore, we were able to confirm that mROS has the portability for utilizing existing ROS packages onto the embedded device. In the systems in Fig. 14 (b), the binary size of the application in the edge device was 3,416 KB, even including the OpenCV library. Most of the source code of `image_converter` in the existing package could be diverted to the mROS environment. In addition, we confirmed the reduction in the communication overhead by processing data on the edge device.

5.3 Discussion

In this case study, `camera` node in the system processes 320×240 size BGRA8 format and publish it. Therefore, in the case of the system of Fig. 14 (a) in which only the camera node is executed on the edge device, it is necessary to publish about 300 KB of data to the host device. From the results in Section 4.2.1, the execution time for data publication was about 20 ms. In the system of Fig. 14 (b) in which `image_converter` node is executed in addition to `camera` node, it is possible to reduce the data to be published by compressing the image data. In this package, compression is performed on image data of 160×120 size BGRA8 format. Therefore, data size to the host device will be about 75 KB of data and data publication time can be about 600 us. Also, communication related to `/image_raw` topic can be performed at high speed by proposed intra-device communication method. From the results in Section 4.2.3, it takes about 6 ms to transfer 300 KB of data.

We found that it is possible to reduce the communication overhead by processing data on the edge device equipped with mROS. Moreover, it is possible to further reduce the communication overhead by porting another nodes which were not implemented in this work onto mROS.

Although we used the OpenCV library that was ported to the RZ/A1H processor in order to implement the `image_converter` node on the edge device, most of source code of `image_converter` in the existing package could be diverted in mROS environment. Therefore, we can utilize existing ROS package resources onto the embedded device by using mROS.

6. Related Works

This section describes related works that proposed a method to utilize embedded devices in ROS based systems.

Ref. [3] introduces brief summaries of existing communication interfaces between ROS host system and external embedded devices. Rosserial [4] that is introduced as one of them is a serial communication interface between embedded device and ROS host node. It supports Arduino, ChibiOS, embedded Linux and other environments by providing ROS client libraries. However, rosserial only offers the low-speed serial communication between host and embedded devices. rosc [8] and μROSnode [15] provide ROS client libraries for embedded systems. Both of them make it possible to connect and exchange messages with ROS nodes by describing and building the system in C language. rosc covers bare-metal systems with small memory resources. Ref. [12] uses μROSnode as the interface to the host system to realize the robot system prototyping. However, these software have not been updated since 2013 to 2014 and have not been maintained. In Ref. [7], a communication layer between the ROS system and non-ROS program is established. rosbridge provides a higher level abstraction layer for the ROS system, making it possible to access using the web socket. In the non-ROS program, the communication message is described in JSON format. rosbridge converts its message into data for the ROS system. The most problematic point in these methods is that the programs executed on the embedded device are related on each environment, and ROS node cannot be executed directly. Therefore, there is no chance to port open source ROS packages onto embedded devices.

Ref. [1] proposes a method that achieves complicated processing required for robot system and reduction of power consumption at same time. Processing for executing complicated algorithms is executed with advanced devices, and loop control is performed in robot devices that require high energy efficiency. For the embedded platform with the LUNA [2] environment, which is a hard real-time framework, they provide the communication bridge interface between different platforms to the ROS system. In Ref. [20], a hybrid real-time ROS architecture is proposed for the purpose of guaranteeing real-time performance in robot systems. The proposed architecture is realized by using RGMP [22] which is a software framework as the virtual communication channel between CPUs. Nuttx as a RTOS and Linux as a general-purpose OS are employed to build an execution environment of ROS nodes. To support communication between ROS nodes on different CPUs by using RGMP, guaranteeing real-time performance can be achieved to ROS nodes executed on RTOS. Ref. [13] proposes PX4 which is a middleware for embedded systems that provides a programming environment enabling communication to the ROS communication layer for programs on embedded devices. However, the communication method between the target embedded device and the device executing the ROS node is only serial communication. In Ref. [17], authors point out the problems that it is difficult to guarantee real-time performance for the intra-system network provided by ROS when the network becomes huge. They propose a method of dividing a system configured using ROS into subsystems by using Smart Resource [16]. Each of these studies provides their own RTOS or runtime environment, and also provides an original communication bridge between them and ROS. This policy is similar to mROS. However, mROS has the advantage of improving the portability of ROS nodes through API compatibility.

Currently, ROS version 2 (ROS 2) has been developed as a next-generation version of ROS. It is designed to be executable in various OS environments including RTOS. However, there is no compatibility between ROS and ROS 2 since the communication protocol of ROS 2 is different from ROS. Therefore, existing resources such as open source packages for ROS cannot be ported to ROS 2 based systems. In Ref. [11], the characteristics of ROS and ROS 2 are evaluated. Authors measured the communication performance of ROS and other ROS compatible middleware. In the publish/subscribe communication model of ROS, authors pointed out that there is a possibility of losing data immediately after the start of communication, and the occurrence of packet division due to data size.

We assumed that embedded devices with mROS are used as edge devices of a distributed robot system. In such a case, it is expected that the embedded device can be easily used without much effort such as modifying the source code. In addition, by utilizing the functions of RTOS, it becomes easy to design a real-time system.

7. Conclusion

This paper proposed a lightweight runtime environment for ROS nodes. We designed mROS, which provides the communication function to a host device and enables ROS nodes to be executed on an embedded device. In order to be operated on an embedded device having a mid-range micro-processor, we employed TOPPERS/ASP kernel as RTOS and lwIP as TCP/IP protocol stack to design the internal structure of mROS. Currently, we have published mROS as open source on GitHub [*6]. Exper-

[*6] https://github.com/tlk-emb/mROS

imental results confirmed that mROS meets the performance requirement for practical applications.

We evaluated the execution time of the API provided by mROS communication library. The result shows that the target performance of mROS can be achieved for the application. In addition, as a result of evaluation of communication time between nodes in the device, we showed that the proposed intra-device communication method becomes faster than the inter-node communication between devices. Moreover, we evaluated the size of the library that composes mROS and showed that it matched to the target embedded devices. In addition, we showed the case study of utilization of mROS in development of distributed robot system. We found that it is possible to utilize the existing ROS package in the mROS environment.

In the future, we will develop a server/client communication model to mROS. It is also necessary to perform a quantitative evaluation of mROS while considering related methods to utilize embedded devices in ROS-based systems. Additionally, we are planning to establish correspondence of the mROS communication library with the ROS 2, which is expected to become the mainstream.

Acknowledgments The part of this work was supported by JST PRESTO Grant Number JPMJPR18M8 and JSPS KAKENHI Grant Numbers JP18K18024. We thank Prof. Takeshi Ohkawa for providing the native ROS version of the edge detection package as the case study of this work.

References

[1] Bezemer, M. and Broenink, J.: Connecting ROS to a real-time control framework for embedded computing, *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation* (*ETFA*), pp.1–6, IEEE (2015).

[2] Bezemer, M. and Wilterdink, R.: *LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework*, Concurrent System Engineering Series, No.WoTUG-33, pp.157–175, IOS Press (2011).

[3] Bouchier, P.: Embedded ROS [ROS Topics], *IEEE Robotics Automation Magazine*, Vol.20, No.2, pp.17–19 (online), DOI: 10.1109/MRA.2013.2255491 (2013).

[4] Bouchier, P. and Purvis, M.: rosserial (2018), available from ⟨http://wiki.ros.org/rosserial⟩.

[5] Renesas Electronics Corporation: Gadget Renesas GR-PEACH board (2018), available from ⟨http://gadget.renesas.com/en/product/peach.html⟩.

[6] Renesas Electronics Corporation: RZ/A1H (2018), available from ⟨https://www.renesas.com/en-us/products/microcontrollers-microprocessors/rz/rza/rza1h.html⟩.

[7] Crick, C., Jay, G., Osentoski, S., Pitzer, B. and Jenkins, O.C.: Rosbridge: Ros for non-ros users, *Robotics Research*, pp.493–504, Springer (2017).

[8] Ensslen, N.: Introduction rosc, *ROS Developers Conference* (2013).

[9] Foote, T. and Rusu, R.B.: nodelet (2018), available from ⟨http://wiki.ros.org/nodelet⟩.

[10] Goldschmidt, S. and Ziegelmeier, D.: lwIP - A Lightweight TCP/IP stack (2018), available from ⟨https://savannah.nongnu.org/projects/lwip/⟩.

[11] Maruyama, Y., Kato, S. and Azumi, T.: Exploring the Performance of ROS2, *2016 International Conference on Embedded Software* (*EMSOFT*), pp.1–10 (2016).

[12] Matteucci, M., Migliavacca, M. and Bonarini, A.: Practical applications of the R2P embedded framework for robot rapid development, *IEEE International Conference on Technologies for Practical Robot Applications* (*TePRA*), pp.1–6 (2015).

[13] Meier, L., Honegger, D. and Pollefeys, M.: PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms, *2015 IEEE International Conference on Robotics and Automation* (*ICRA*), pp.6235–6240 (2015).

[14] Merrick, P., Allen, S. and Lapp, J.: XML remote procedure call (XML-RPC) (2018).

[15] Migliavacca, M., Zoppi, A., Matteucci, M. and Bonarini, A.: μROSnode: Running ROS on microcontrollers, *ROS Developers Conference* (2013).

[16] Munera, E., Alcobendas, M.M., Poza-Lujan, J.-L., Yague, J.L.P., Simo-Ten, J. and Noguera, J.F.B.: Smart Resource Integration for Robot Navigation on a Control Kerned Middleware Based System, *International Journal of Imaging and Robotics*, Vol.15, No.4 (2015).

[17] Munera, E., Poza-Lujan, J.-L., Posadas-Yague, J.-L., Simo, J. and Noguera, J.F.B.: Distributed Real-time Control Architecture for ROS-based Modular Robots, *IFAC-PapersOnLine*, Vol.50, No.1, pp.11233–11238 (2017).

[18] TOPPERS Project: TOPPERS/ASP kernel (2018), available from ⟨https://www.toppers.jp/en/asp-kernel.html⟩.

[19] Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R. and Ng, A.Y.: ROS: An open-source Robot Operating System, *ICRA Workshop on Open Source Software*, No.3.2, p.5 (2009).

[20] Wei, H., Huang, Z., Yu, Q., Liu, M., Guan, Y. and Tan, J.: RGMP-ROS: A real-time ROS architecture of hybrid RTOS and GPOS on multi-core processor, *2014 IEEE International Conference on Robotics and Automation* (*ICRA*), pp.2482–2487 (2014).

[21] yoneken: measure message passing (2018), available from ⟨https://github.com/yoneken/measure_message_passing⟩.

[22] Yu, Q., Wei, H., Liu, M. and Wang, T.: A novel multi-OS architecture for robot application, *2011 IEEE International Conference on Robotics and Biomimetics*, pp.2301–2306 (online), DOI: 10.1109/ROBIO.2011.6181641 (2011).

Hideki Takase is an Associate Professor at Kyoto University, and also a PRESTO researcher at Japan Society and Technology Agency. He received his Ph.D. degree of Information Science from Nagoya University in 2012. From 2009 to 2012, he was a research fellow of the Japan Society for the Promotion of Science (DC1). He had been an Assistant Professor at Graduate School of Informatics, Kyoto University since 2012, and promoted to current title in 2018. He received the Incentive Award from Computer Science group of IPSJ in 2008, the Funai Research Incentive Award from the Funai Foundation for Information Technology in 2015. His research interests include runtime platform and system-level design methodology for embedded/real-time/IoT computing. He is the member of IEICE and RSJ.

Tomoya Mori received his M.E. degree of informatics from Kyoto University, Kyoto, Japan in 2018. His research interest is the software platform for robot software and embedded systems. He received the Incentive Award from Computer Science group of IPSJ in 2018.

Kazuyoshi Takagi received his B.E., M.E. and Dr. of Engineering degrees in information science from Kyoto University, Kyoto, Japan, in 1991, 1993 and 1999, respectively. From 1995 to 1999, he was a Research Associate at Nara Institute of Science and Technology. He had been an Assistant Professor since 1999 and promoted to an Associate Professor in 2006, at the Department of Information Engineering, Nagoya University, Nagoya, Japan. He moved to Department of Communications and Computer Engineering, Kyoto University in 2011. In 2019, he joined Department of Information Engineering, Mie University, as a Professor. His current interests include system LSI design and design algorithms.

Naofumi Takagi received his B.E., M.E., and Ph.D. degrees in information science from Kyoto University, Kyoto, Japan, in 1981, 1983, and 1988, respectively. He joined Kyoto University as an instructor in 1984 and was promoted to an associate professor in 1991. He moved to Nagoya University, Nagoya, Japan, in 1994, and promoted to a professor in 1998. He returned to Kyoto University in 2010. His current interests include computer arithmetic, hardware algorithms, and logic design. He received Japan IBM Science Award and Sakai Memorial Award of the Information Processing Society of Japan in 1995, and The Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology of Japan in 2005.