A functionality expansion of the lightweight runtime environment mROS for the user defined message types

Hidetoshi Yugen[1,a] Hideki Takase[1,2,b] Kazuyoshi Takagi[1] Naofumi Takagi[1]

Abstract: The mROS is a lightweight execution environment that enables the node programs of the Robot Operating System (ROS) to be executed on embedded devices. In this research, we aim at removing the constraint on the message types that can be handled by mROS. We propose an approach that automatically generates the header files of message types for mROS. Moreover, we propose an operation flow for the mROS communication library. The proposed approach and flow enable the mROS environment to handle various message types, including primitive and user-defined types. Therefore, the versatility of the mROS can be improved, and the development of a cooperative system of general-purpose devices using ROS and embedded devices using mROS will become easier.

Keywords: ROS, embedded devices, real-time operating systems, communication

1. Introduction

In recent years, software frameworks that aid in the development of robot software have attracted significant attention. Such software improves the productivity of robot software development. Among this software, the Robot Operating System (ROS)[1] has been the focus of substantial research.

The ROS also enables nodes to communicate messages with one another. This communication is performed via topics, which are communication channels, and this feature enables easy and rapid implementation of inter-node communication functions. Moreover, the executable and configuration files of the nodes can be grouped together in packages for reuse or distribution. Robot applications can be developed effectively by installing existing packages.

The ROS communication feature is realized as Linux middleware. Therefore, high-ended devices with high power consumption are required to operate Linux and employ the ROS. Furthermore, it is difficult to enhance the real-time performance using Linux.

To address this problem, we are currently developing mROS[2], a lightweight execution environment for ROS nodes. With mROS, programs that behave as ROS nodes and communicate with native ROS nodes can be executed on embedded devices equipped with a real-time OS. Moreover, the programming interfaces of the mROS communication library are designed similarly to that of the ROS communication library. Therefore, existing ROS packages can easily be ported to mROS applications.

However, the message types that can be handled by mROS is limited to the string type only. Therefore, the ROS node applications that communicate with mROS node applications are required to be implemented under this limitation. This constraint also results in differences in the programming interfaces between mROS and ROS node applications. Owing to this problem, when porting ROS node applications to mROS applications, all the codes using the ROS communication library need to be replaced. Consequently, the development of systems in which general-purpose devices equipped with the ROS can cooperate with embedded devices equipped with the mROS is difficult and complicated.

In this research, we aim to improve the versatility of the mROS. We propose a method to support any message types in mROS, as well as a flow of mROS communication library, in order to remove the limitation.

In the proposed method, header files for message types are generated for use in mROS applications. A header file is prepared for each message type, and the data of the type, such as the structure and functions for the types, are packaged in the file. The header files offer lightweight implementation and are designed for execution on embedded devices.

In the proposed mROS operation flow, type-specific methods are called during the communication process. These type-specific methods are generated from the method mentioned above and are available from standardized programming interfaces. Therefore, the appropriate process for each message type is called and executed without changing mROS communication library code.

This research contributes to improving the versatility of mROS. As a result, various types of robot applications can be realized using the mROS. Moreover, the development of robot applications using mROS may become easier and simpler, particularly for developers of ROS applications.

The remainder of this paper is organized as follows. We introduce the ROS, mROS, and the existing research in Section 2. We consider the key goals of the research in Section 3. Thereafter, we propose the method for generating header files for each message type in Section 4, and the flow of the mROS communication

[1] Graduate School of Informatics, Kyoto University
[2] JST PRESTO
[a] emb@lab3.kuis.kyoto-u.ac.jp
[b] takase@i.kyoto-u.ac.jp

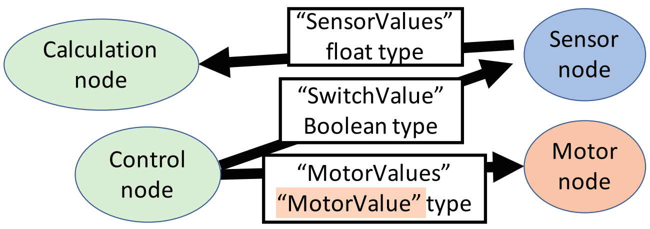

Calculation node
"SensorValues" float type
Sensor node
"SwitchValue" Boolean type
Control node
"MotorValues" "MotorValue" type
Motor node

Fig. 1 Example of system utilizing ROS

library in Section 5. In Section 6, we discuss the effectiveness of our proposal. We present conclusions and possible future work in Section 7.

2. Background

2.1 ROS

The ROS[1] is a series of libraries and tools that support the development of robotic systems. The ROS provides the middleware that provides communication between nodes.

A system realized by using the ROS consists of nodes, which are functional units in the system, and the master node, which manages the entire system. The master manages the information of nodes within the system and passes it to them so that the nodes can communicate with one another.

The communication feature provided by ROS is based on the publish-subscribe messaging model. In this model, topics are prepared according to the message purposes and types. By means of the topics, communications are performed between publishers, which send messages, and subscribers, which receive messages. Therefore, all of the nodes communicate with one another via topics.

Figure. 1 presents an example of robotic software realized using the ROS. In the system, nodes communicate with one another by means of topics according to the communication purposes. In the Figure. 1, the ovals represent nodes, while the rectangles represent topics. In the rectangles, the double-quoted strings are the names of the topics, and the strings below these represent the message types employed in the topic. Note that the "MotorValue" and "Image" types in the figure are user-defined message types, and the variables within the types are listed below the names.

Users can define message types by combining primitive types. Existing types, including user-defined types, are also available as the variable types. For example, the "MotorValues" type in Figure .1 can be used as a variable type in other message types. Hereinafter, we refer to a message type with such a structure as a nested type. Arrays can also be included in a type.

A message type can be defined by the following procedure: Firstly, create the type definition file (`.msg`) and write the type definition therein. Next, generate the appropriate program code for the type using the ROS build system. For example, when C++ is employed in the implementation, header files that define the the message type class are produced.

2.2 mROS

The mROS[2][3] is a lightweight runtime environment for ROS nodes that has been under development since 2018. The purpose of the mROS development is to execute programs that operate as ROS nodes. It is aimed at embedded devices with mid-range microprocessors that are equipped with TCP/IP protocol stacks and Real-Time Operating Systems (RTOS). The programming interface of the mROS communication library is similar to that of the ROS, thereby providing improved compatibility between mROS and ROS applications.

The mROS communication feature is provided by several tasks, including user tasks. When sending messages, the operating flow proceeds as follows:

(1) The message to be sent is written into shared memory by the user task, and the message information is queued in the data queue of the publishing task.
(2) When the information is queued, the publishing task wakes up.
(3) The publishing task serializes the message in the shared memory, using the information in the queue, and sends it.

When receiving messages, the operating flow proceeds as follows:

(1) Once the connection to the publisher has been established, the subscribing task awaits the messages.
(2) When messages are published regarding the topic, the subscribing task receives these as a bit string, and copies their body into the shared memory.
(3) The bit strings in the shared memory are deserialized to the message objects.
(4) The function is called back, providing the object as an argument.

2.3 Related works

In this section, we provide an overview of existing studies that have proposed distributed systems with the ROS[4][5] or have utilized the ROS on embedded systems[7][8].

In [4], the ROS was employed to design a system for an autonomous wheelchair. By utilizing the ROS, the proposed system is more flexible than traditional robot systems. Furthermore, the system uses Arduino as a motor control unit, and motor control is performed by sending messages to the ROS node on the Arduino, resulting in a lower system cost.

In [5], a system utilizing a cloud for mobile robots was realized by means of the ROS. In the proposed system, the mobile robot offloads the calculation for vSLAM to the cloud, which is achieved by the ROS inter-node communication feature.

In both [4] and [5], the proposed systems includes edge devices, which collect information, send and receive messages, and control themselves according to the messages received. The edge devices in both of these works are laptops. Embedded devices equipped with the ROS can operate as the edge devices and offer alternatives to the laptops. As a result, the edge devices can be smaller and their power consumption can be reduced.

The ROS package rosserial[6] has been used to execute ROS nodes on embedded devices. By using rosserial, the programs operating as ROS nodes can be run on embedded devices. However, only serial communication is supported; therefore, the transmission range and speed are limited compared to those of network communication.

In [7], an ROS node that serves as a bridge between ROS nodes and LUNA, a real-time software framework, was proposed. The bridge node communicates directly with LUNA applications, and sends and receives messages via ROS topics. In this manner, the communication between ROS nodes and LUNA applications can be realized. However, a bridge node is required between the two frameworks. Furthermore, systems utilizing this bridge node require a higher learning cost, because two different frameworks are used.

In [8], a communication interface between the ROS and CODESYS, one of the main products of the PLC software, was proposed. In the proposed system, an ROS node serves as an interface, and it sends and receives messages as well as accesses to the shared memory of CODESYS. This system also requires a bridge node between the two different systems. Moreover, in the [8], the support of user-defined message types and arrays was not realized, although it was stated that it is possible.

3. Objectives

In this section, we consider the objectives to be realized in this research.

The processes that should be porformed by mROS communication library to communicate with ROS node applications can be classified into two opeeration types: the operations that are common among all message types, and operations specific to each message type. The operations specific to each message type constitute the processes that should be changed according to the structure of the messages.

We take this fact into account when considering the detailed specification. Firstly, we separate these operations, so that operations of one type will not affect others. Type-specific operations are defined for each type. A file is prepared for each message, and the operations are packaged therein. Thereafter, a common programming interface is designed to call the type-specific processes. The operations called are selected appropriately by specifying the proper type. Therefore, the type-common operations can call all of the type-specific operations through a single interface. Furthermore, by simply changing the specifying types, the operations called can be altered without changing the type-common processes.

Furthermore, we design the operation flow of the mROS communication library. The operations that should be performed by the mROS communication library to communicate with the ROS nodes can be classified into the following three operations: the registration of nodes for publishing or subscribing, message sending, and message receiving. We consider the requirements and design an operation flow for each process.

When registering nodes, the node sends the topic information to the master node, which includes the node name, topic name, and message type transmitted through the topic. To achieve this, the type-specified functions that return this information should be prepared.

When sending messages, the messages are serialized into bit strings and then stored in the shared memory. To realize this behavior, functions that process messages appropriately should be provided and called according to the message structures.

However, messages are received as bit strings, and then converted into the appropriate message structures. To enable this behavior, type-specific functions for deserializing the messages are required.

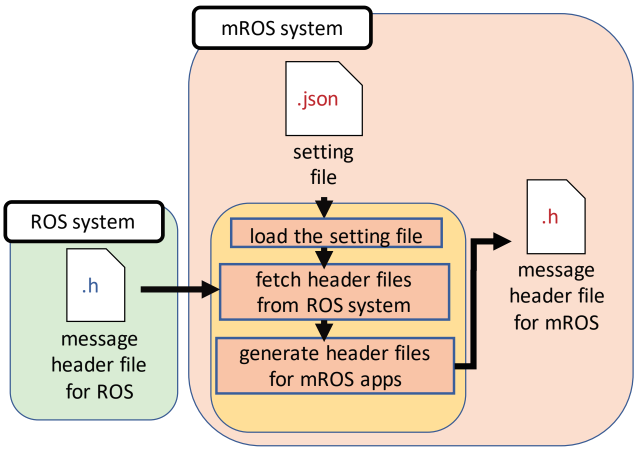

mROS system
.json
setting file
ROS system
.h
message header file for ROS
load the setting file
fetch header files from ROS system
generate header files for mROS apps
.h
message header file for mROS

Fig. 2 Generation flow of message header file

4. Proposed Method for Generating Message Header Fiiles

In this section, we propose a method for generating type-specified operations as header files to fulfill the requirements described in the previous section. The proposed flow is illustrated in Figure. 2. The ROS system is represented by the green area on the left side of the figure, while the mROS system is indicated by the orange area on the right side. In the orange area, the procedures that should be performed by developers are represented by the yellow rounded rectangles, while those that are conducted automatically are denoted by the red rectangles.

The proposed method is implemented as follows: Firstly, prepare the configuration file and specify the location of the ROS workspace therein. Next, from the specified location, identify the header files for the desired message types and extract the type information from the file. The information includes the type name, definition, and aggregate hash. Finally, process the header files for use in mROS applications. The values extracted in the previous procedure are copied into the new file without changes. Furthermore, the message structure is interpreted from the extracted definition. Serializing and deserializing functions are generated, corresponding to the structure. The implementations of the processed header files are light, so that they can be executed on devices with fewer resources, such as embedded devices.

The serializing and deserializing functions are generated so that each variable defined in the message type is processed sequentially, according to the variable type. This is an effective approach because all message types are composed of variables of the primitive, `Hearder`, or the array type. Moreover, messages may exhibit a nested structure; that is, one message type may include other message types as its variable. However, the included type can also be broken down recursively into variables of primitive, `Header`, or array types. Therefore, it can be stated that the proposed method for generate serializing or deserializing functions is valid for any type.

We present the details of the function in the following section.

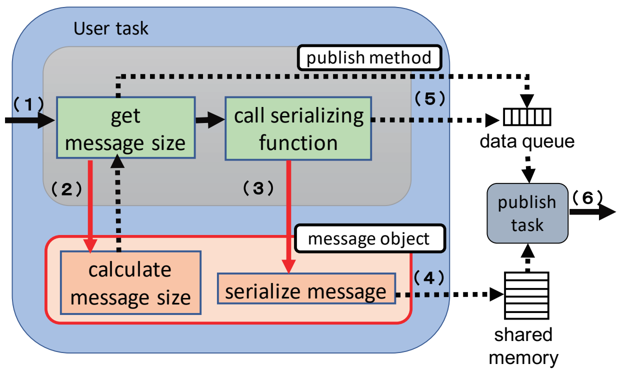


Fig. 3 Operation flow of publishing message

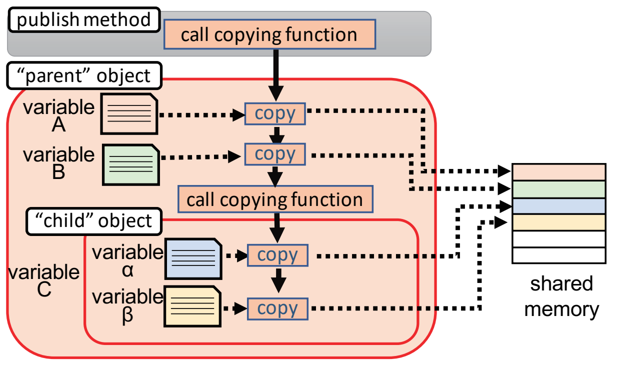


Fig. 4 Operation flow of processing of nested messages

5. Operation Flow of mROS Communication Library

5.1 Registration of nodes

The proposed flow for registering nodes proceeds as follows:

(1) When calling the registering function, the argument for specifying the message type is passed.

(2) The type-specific function is called using the argument, and the information required to register a node is obtained from the function.

(3) The information is sent to the master node using the XML-RPC protocol.

During registration of a node, the information of the node and the topic with witch it is concerned is sent to the master node. The information is defined in header files for the mROS, which are generated from the header files in the ROS system and can be called from the getter functions. All message types have their getter functions with the same interface, regardless of the types. In this manner, the coupling between the mROS communication library program and called getter functions is weakened. That is, with this specification, the mROS communication library is less dependent on the message types. As a result, the library codes become more flexible to the types they can handle.

5.2 Sending messages

The proposed message sending flow is illustrated in Figure. 3. The details of the flow described in the figure can be summarized as follows:

(1) The sending message is received as an object from a user application.

(2) A method is called on the object to calculate the message size.

(3) The serializing method is called on the object

(4) The serializing function copies the message data as a bit string into the shared memory.

(5) The message size and the pointer to the copied data in the shared memory are queued in the publishing task data queue.

(6) When the data are queued, the publishing task wakes up and sends the data in the shared memory to other nodes.

In the figure, the gray rounded rectangles represent processes that are common to all message types, while the orange rounded rectangles represent processes that are specific to a message type. Hereafter, we refer to the former as common processes and the

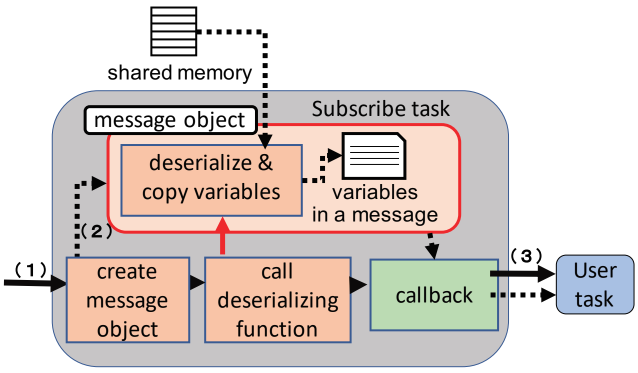


Fig. 5 Operation flow of message receiving

latter as type-specific processes. Note that the codes corresponding to the operations in the orange rectangle are generated by the proposed method, as described in the previous section. The red arrows in the figure represent the procedure calls. As indicated by the arrows, the common processes call the type-specific processes in the proposed flow. The type-specific processes are also responsible for copying the messages into the shared memory. Furthermore, the type-specific processes only return the calculated message sizes to the common processes. These values are integers, regardless of the handled message type. This specification results in weak coupling between the common and type-specific processes. Therefore, the common processes can deal with any message type by changing the type-specific processes that they call. This change can be achieved by changing the argument passed to the common process.

This flow enables serializing of messages with a nested structure. In the following description, we refer to a message object with a nested structure as a parent object, and to another object included in the parent object as a child object. Figure. 4 illustrates the serializing flow of a parent object.

As indicated in the figure, in this operation, the parent object recursively calls the serializing method of the child object. The parent and child objects share the pointer, so that both can copy into the appropriate area of the shared memory. This is feasible because all of the message objects have deserializing methods with the same interface.

5.3 Receiving messages

The proposed message receiving procedure is illustrated in Figure. 5. The process details are as follows:

(1) A message object is generated, which is the same type as the receiving message.
(2) The deserializing method is called on the object. In the deserializing function, the received bit string is copied into the object as the message values.
(3) The callback is executed, providing the object as an argument.

The orange rounded rectangle in the figure expresses the type-specific procedure. In this flow, the deserializing procedure corresponds to the type-specific processes. The procedure is realized as a method in the message object, and is called by the common processes. The method interprets the bit string in the shared memory, and copies values from here to the variables in the message. Thereafter, the object is passed to the callback function. In the figure, this procedure is represented by the green rectangle with the description ”executing callback.” The dashed arrows correspond to the data flow. The dashed arrow pointing to the green rectangle indicates the data flow of the message objects passed to callbacks. The orange rounded rectangle in the figure represents the type-specific procedure. It should be noted that the deserializing process returns nothing to the common processes. As a result of this specification, the coupling between the common processes is weakened, and the receiving processes become more flexible to different message types. Moreover, as in the sending case, messages with nested structures can be processed using this flow.

6. Evaluation and Discussion

In this section, we evaluate and confirm the effectiveness of our proposal and discuss its usefulness.

6.1 Evaluation environment

We implemented our proposed method as a tool, as well as the proposed mROS operation flow in the communication library.

We executed mROS applications on GR-PEACH, which is a development board with a microcontroller that can be equipped with the TOPPERS/ASP kernel and mbed library. The board can also be connected to a network using an Ethernet cable.

We prepared the local network and connected a laptop wirelessly and GR-PEACH through a cable. The laptop was equipped with Ubuntu 16.04 LTS and ROS Kinetic, while GR-PEACH was equipped with the mROS. We executed the nodes and the master node on the laptop. During the test, the mROS applications communicated with these to send and receive messages.

6.2 Operation verification

We verified that the functionality expansion based on our proposal made it possible for the mROS applications to handle all primitive types as well as user-defined message types. To achieve this, we prepared ROS node applications and mROS applications, and performed message communications with these message types. During the evaluation, messages of all primitive types and a user-defined type were tested.

The test and results of the communication of the user-defined message type are presented below. We defined the“`PersonalData`” type and used it in the test. This type included the variables below:

```
ros::init(argc,argv,"mros_node");
ros::NodeHandle n;
ros::Publisher pub = n.advertise<
mros_test::PersonalData>("mros_str",1);
ros::Rate loop_rate(5);
mros_test::PersonalData msg;
msg.first_name = "Phil";
msg.last_name = "Woods";
msg.age = 83;
msg.score = 100000;
while(1){
  wait_ms(1000);
  pub.publish(msg);
  msg.score ++;
}
```

Fig. 6 Example code to publish messages from mROS

```
void Callback(mros_test::PersonalData::Ptr msg){
  syslog(LOG_NOTICE, "I heard a msg from ros host");
  string name = msg->first_name + " " + msg->last_name;
  syslog(LOG_NOTICE, "name:%s",name.c_str());
  syslog(LOG_NOTICE, "age:%u",msg->age);
  syslog(LOG_NOTICE, "score:%u",msg->score);
}
void usr_task2(){
#ifndef _USR_TASK_2_
#define _USR_TASK_2_
  ros::init(argc,argv,"mros_node2");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("test_msg",1,Callback);
  ros::spin();
#endif
}
```

Fig. 7 Example code to subscribe from mROS

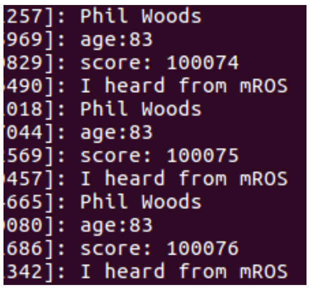
- “first_name” of string type
- “last_name” of string type
- “age” of unsigned 16bit integer type
- “score” of 32bit integer type

Firstly, we tested sending messages from an mROS application. The code of the mROS publisher application is presented in Figure. 6, while the output of the ROS subscriber application is presented in Figure. 8.

As Indicated in Figure. 8, the subscriber ROS application correctly received and output the data set in lines 7 to 10 of Figure. 6. Moreover, it can be observed from Figure. 8 it that the value of “score” was incremented per message, as a result of line 14 in Figure. 6. This indicates that the variable “score” was definitely handled as an integer.

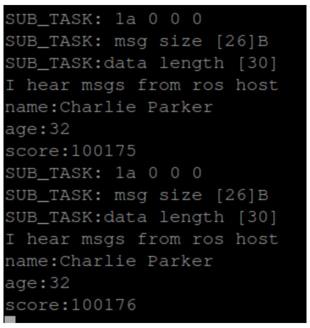
Thereafter, we examined receiving messages on an mROS application. The code of the mROS publisher application is presented in Figure. 7, while the output of the mROS subscriber application is shown in Figure. 9.

The ROS publisher node application exhibited almost the same behavior as that of the mROS. The differences were the values of `first_name` and `last_name`. In the ROS application, `first_name` was set to “Charlie” and `last_name` was set to “Parker.” As illustrated in Figure. 9, the values set on the ROS publisher application were included in the output. Regarding the



```
257]: Phil Woods
969]: age:83
829]: score: 100074
490]: I heard from mROS
018]: Phil Woods
044]: age:83
569]: score: 100075
457]: I heard from mROS
665]: Phil Woods
080]: age:83
686]: score: 100076
342]: I heard from mROS
```

Fig. 8 Result of message recieving on ROS node



```
SUB_TASK: 1a 0 0 0
SUB_TASK: msg size [26]B
SUB_TASK:data length [30]
I hear msgs from ros host
name:Charlie Parker
age:32
score:100175
SUB_TASK: 1a 0 0 0
SUB_TASK: msg size [26]B
SUB_TASK:data length [30]
I hear msgs from ros host
name:Charlie Parker
age:32
score:100176
```

Fig. 9 Result of message recieving on mROS

first_name and last_name variables, it can be observed from Figure. 8 that the two strings were connected as a result of line 3 in the code of Figure. 7. As a core score variable, it can be seen that the value was incremented per message, which reflects the implementation of the ROS publisher application These results confirmed that the mROS applications could receive, interpret, and handle correctly.

6.3 Discussion

In this section, we discuss the improvements this research contributes to the mROS.

Firstly, conversion between the string type and other message types is no longer required, which results in a reduction in the communication overhead and more efficient application executions.

Secondly, it is possible to select the appropriate type for each message, and therefore, the numbers of symbols required to communicate messages can be reduced. In most cases, more symbols are required to send numbers as a string than as numbers. Moreover, when sending variable-length messages such as strings, the message length of the 4-byte integer should appear on top of the messages, owing to the ROS specification. This research has solved this problem, and developers can now select the appropriate message types with less traffic.

This can be explained by using a message type with three variables of 32-bit float as an example. Messages with this type have a size of 12 bytes, including three float variables with a size of 4 bytes each. If this message is sent as the string type, the body size will be 25 bytes. This is because each float requires seven letters for the float type precision, as separators should be inserted between floats, and the body length expressed by a 4-byte integer must be included in the message. As explained below, by selecting appropriate message types, the symbols required for messages, particularly those with numeric types, can be reduced.

Thirdly, the message types used in mROS applications can be specified during their construction. Therefore, the size of the shared memory used by an mROS application can be optimized by fitting it to the message size. To confirm this, we implemented the memory optimizing function according to the message type. We built an mROS application using this function, and compared its size to the same application built without using the function.

In the evaluation, we prepared the message type with the three 32-bit float variables used in the previous evaluation. As mentioned previously, the size of a message of this type is 12 bytes. We measured the size of libmros.a by using a size command of Linux. The libmros.a includes the mROS communication library, and is generated when building an mROS application.

The actual sizes obtained by means of the size command are presented in Table 1. Note that hereinafter the "new implementation" means the mROS application with a new implementation based on this research, while "old implementation" means the mROS application with the old implementation, as designed in [2] and [3].

Table 1 Comparation of the program size of mROS communication library

	text[Byte]	data[Byte]	bss[Byte]	dec[Byte]
old impl.	69,464	28	2,097,319	2,166,811
new impl.	69,460	28	1,048,739	1,118,281

As indicated in the table, the size of the bss section was reduced. In the old implementation, a memory size of 1 MB was allocated to the shared memory for inter-task communication. The large size of the shared memory is for messages with large sizes, such as QVGA-formatted image data. The contribution of this study makes it possible to shrink this memory area and fit it to the size of the message types. When the message type includes variables of a variable length, the memory area can be optimized by specifying the variable lengths.

As a result of this research, the message types of topics can be specified when the publish or subscribe method is called. Therefore, the programming interface of the mROS communication library can be designed to be more similar to that of the ROS communication library. Figures 6 and 7 illustrate certain important parts of the mROS code for communication operations. As indicated in the figures, the mROS communication library can be called by codes that are quite similar to those that call the ROS communication library. Consequently, the compatibility between ROS and mROS applications has been improved in this work.

7. Conclusion

In this research, we have improved the versatility of the mROS by removing the constraint on the message types that can be handled. To achieve this, we have proposed a method for generating header files for message types and a flow of the mROS communication library that can handle any message types. The proposed method generates header files for an mROS application based on header files that exist in the ROS. The proposed operation flow calls message-specific functions defined in the above header files.

We implemented our proposed method and operation flow on the mROS to evaluate them. It was confirmed that the mROS can send and receive messages of the following types: primitive types except for time and duration types, arrays of primitive types, and user-defined message types. Note that Time and Duration types are available on the mROS using our proposal, although we could not implement these owing to a lack of time.

Moreover, we discussed the usefulness of this extension.

This study contributes to easier development of robotic applications using the mROS. Furthermore, the versatility of the mROS is improved and it is therefore capable of developing various applications.

Future work will include the support of message types that are not yet supported, evaluation of the communication between nodes on the same device, and comparisons with other frameworks.

Acknowledgments The part of this work was supported by JST PRESTO Grant Number JPMJPR18M8 and JSPS KAKENHI Grant Numbers JP18K18024.

References

[1] Quigley, M., Conley, K., Gerkey, B., 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, pp. 5 (2009).

[2] Hideki, T., Tomoya, M., Kazuyoshi, T., and Naofumi, T.: Work-in-Progress: Design Concept of a Lightweight Runtime Environment for Robot Software Components Onto Embedded Devices, *2018 International Conference on Embedded Software (EMSOFT)*, pp. 1–3 (2018).

[3] Hideki, T., Tomoya, M., Kazuyoshi, T., and Naofumi, T.: mROS: A Lightweight Runtime Environment for Robot Software Components onto Embedded Devices *HEART 2019 Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, Article No. 7 (2019).

[4] Zhengang, L., Yong, X., and Lei, Z.: ROS-Based Indoor Autonomous Exploration and Navigation Wheelchair, *2017 10th International Symposium on Computational Intelligence and Design (ISCID)*, Vol. 2, pp. 132–135 (2017).

[5] Patrick, B., Mohan, M., Paul, R., John, J. P., Mo, J., and Lutcher, B.: Cloud-Based Realtime Robotic Visual SLAM, *2015 Annual IEEE Systems Conference (SysCon) Proceedings*, pp. 773–777 (2015).

[6] Ferguson, M.: rosserial, http://wiki.ros.org/rosserial.

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

[8] Tiago, P., Rafael, A., and Germano, V.: Bridging Automation and Robotics: an Interprocess Communication between IEC 61131–3 and ROS, *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pp. 1085–1091 (2018).

[9] TOPPERS Project : TOPPERS/ASP kernel, https://www.toppers.jp/asp-kernel.html.