Chapter

Real-Time Robot Software Platform for Industrial Application

Sanghoon Ji, Donguk Yu, Hoseok Jung and Hong Seong Park

Abstract

In this study, we present the requirements of a real-time robot software (SW) platform that can be used for industrial robots and examine whether various kinds of existing middleware satisfy them. Moreover, we propose a real-time robot SW platform that extends RTMIA to various industrial applications, which is implemented on Xenomai real-time operating system and Linux. The proposed SW platform utilizes the timer-interrupt based approach to keep strict period and the shared memory for convenient usage, on which the shared variable is designed and used. We verify the proposed platform by showing that the robot task and the Programmable Logic Controller (PLC) program are performing with interlocking each other on the presented platform.

Keywords: software platform, real-time, middleware, industrial robot, PLC

1. Introduction

A robot software (SW) platform is an execution environment that provides various functions to easily execute various robot softwares. Therefore, a middleware that provides such functions can be considered as a platform.

Middleware is a type of software that exists between an application and an operating system, and it employs communication protocols on the hardware, thereby facilitating the development applications by users and reducing the cost and risk when developing them [1]. Several types of middleware have been applied in the field of robotics, such as Common Object Request Broker Architecture (CORBA) [2, 3], Real-Time CORBA (RT-CORBA) [2, 4], Data Distribution Service (DDS) [2, 5], OPC Unified Architecture (OPC-UA) [6], Robot Operating System (ROS) [7, 8], Open Platform for Robotic Services (OPRoS) [9–11], OpenRTM (open robotics technology middleware) [12], open robot control software (OROCOS) [13], XbotCore [14], and real-time middleware for industrial automation devices [15] (hereafter, referred to as RTMIA).

A common feature of these middleware is that they support message-based communication. Thus, a majority of middleware can be implemented in multiple nodes, such as processes, threads, or CPU boards that support middleware-unique communication protocol to facilitate data exchange between them. The difference among these middleware is whether the real-time constraint is satisfied or not, from the perspective of the application thread or process. Even in the real-time viewpoint, providing real-time from a communication point of view is different from providing real-time from a (application) process point of view. In general, the term

"real-time" in this study indicates the real-time used in the process viewpoint. Hence, middleware must be able to control user-created programs using processes or threads.

Even though middleware such as CORBA, RT-CORBA, OPRoS, openRTM, OROCOS, XbotCore, and RTMIA control threads, in other middleware such as OPC-UA, ROS, and DDS, the control of threads or processes is implemented by the operating system. Hence, they do not directly control threads or processes. Except CORBA, all of the above-mentioned thread controlling middleware satisfy the real time constraint. However, if they are analyzed in detail, RT-CORBA, OPRoS, openRTM, and OROCOS have conditions to satisfy the real time, in which the execution type is limited to thread type and threads can be controlled in the time range such as over 1 ms [8] or 10 ms [11]. In [14] EtherCAT-based robot real-time SW platform is proposed, which is a Xenomai-based robot control middleware that satisfies the hard real-time requirements and is designed to perform 1 kHz control loops in a multi-axis system consisting of 33 axes. RTMIA is middleware that can execute control loops in 100 μs and can simultaneously control both processes and threads in real-time.

OPC-UA, ROS, and DDS are a kind of communication middleware. That is, they do not satisfy the real time requirement since they are in charge of the data exchange part between multiple nodes. Note that ROS 2.0 utilizes DDS. OPC-UA is a middleware that performs data exchange among various robots and controls devices such as the FMS controller or the server on the upper level. OPC-UA is not a middleware that can be used for robot control like ROS. The use cases of ROS shows that most types of the user program are process types rather than thread types. In other words, the thread type provided by the real-time middleware is reluctant to be used because their basic usage forms may introduce inconvenience and unfamiliarity for robot software developers. ROS has big advantage of using the conventional programming method, which users are familiar with, and utilizes the ready-made existing robot program. However, ROS must use 'sleep()' function for periodic processing. 'sleep()' function does not always wake up at the correct time, resulting in time shift occurrence. Accumulation of such shifts cause loss of some periods, which may lead to states that cannot be controlled properly. In order to avoid such situation, some constraints must be kept to strictly control the number of applications and the execution environment.

The current trend of commercial PLC-related products [16, 17] shows that platforms such as CODESYS [16] and TwinCAT [17] control robot systems, including grippers, by embedding robot motion control SWs to PLC SWs. The feature of these platform is that they use the robot block language proposed by PLCopen [18], while their principal advantage is the ability to link robots to related automation devices efficiently using PLC program. However, the types of robots that can be controlled through these platforms are still limited. To solve this, the platform should simultaneously perform both PLC function blocks, including robot function blocks, and robot (task) programs defined directly by users or robot developers.

In this study, we presented the requirements of the real-time robot SW platform for industrial robots and examined whether various types of existing middleware satisfy them. Additionally, we extend the RTMIA [15] and propose a real-time robot SW platform that can be used for various industrial applications and satisfies the presented requirements. We implemented our proposed platform on the Xenomai real-time operating system and Linux, and verify it via a few practical examples, wherein the interlocking programs of robot task and PLC are simultaneously performed.

The remainder of the study is organized as follows. In Section 2, we present the requirements of the real-time robot SW platform for industrial applications and

discuss whether various kinds of existing middleware satisfy them. Section 3 presents the implementation of the real-time robot SW platform, and Section 4 describes the practical results obtained by mutually interlocked work of programs of robot task and PLC on the proposed platform. Finally, Section 5 concludes the study.

2. Operation requirements and analysis of real-time robot SW platform

For a real-time robotic SW platform for various industrial applications, we must know the characteristics (period, etc.) of the data used for processing periods and events. Referring to the data presented in [19], the data used in the manufacturing robot can be roughly described as in **Table 1**. It may differ slightly from other classifications because it is categorized based on type and period (or delay time) of control/sensor data.

In fact, the data generated at each level in **Table 1** is used as basis for the target data. For example, upon recognizing an obstacle at level 4, its position data is used to generate a level 3 position profile that provides relevant position data of level 2, based on which, at level 1, a position control loop moves the motor to the desired position by controlling the speed or torque of the motor. That is, data of each level is interlocked with each other in terms of control.

A real-time robot SW platform should satisfy the following requirements.

R1: Support data exchange
R2: Real time support (strict period execution and sporadic performance support)
R3: Supports thread and process types for user defined programs
R4: Easy configuration of applications (robot control SW, PLC SW, vision inspection SW, non-real-time SW, etc.)
R5: Support multiple periods.
R6: Threads or processes running in the same period are classified by priority.
R7: Check and handle the event through the event handler.

R1 serves as a communication middleware by providing a method for exchanging data between configured applications. R2 supports real time operation and requires a minimal amount of jitter to perform periodic operations. It must also be able to handle real-time events. For instance, conditions for use in various control loop programs are strict periodic execution conditions, including the program for directly controlling the motor without using a separate controller, and sporadic to handle emergency event such as an emergency stop. In many cases, such execution condition is required. In particular, the 100 μs period can be used to control the motor current. R3 uses threads for real-time support, but also allows the process to

Level	Data type	Period (delay time)
4	Obstacle recognition, object recognition, device status	Seconds
3	Position profiles	Hundreds of ms
2	Position data	Tens of ms
1	Motor torque and speed control data	100 μs

Table 1.
Types of control/sensor data used in manufacturing robots and their associated periods (delays).

be used for soft real-time execution. Process support, as mentioned earlier, is about using legacy programs. R4 does not control the developed application software modules in a hard-coded program, but rather configures them by freely setting the period, priority, application name, etc. through a configuration file.

R5 and R6 are about the execution period. As shown in **Table 1**, various execution periods are required. For example, in the case of motor control, if the order of sensor reading (RS), control value generation (GC), and motor value writing (WM) are different, the value sampled in the previous period is used. In other words, when RS, GC, and WM are executed, the control value can be read within one period and sent to the motor. However, when operating in the order of GC, RS, and WM, the control value calculated using the value read in the previous period affects the motor. That is, the previous value without using the current motor related data may affect the motor control. Therefore, to avoid such effects, it is necessary to set the execution priority of SW module within the same period so that they can operate in the order of RS, GC, and WM.

R7 relates to how to handle events. Depending on the event behavior utilized by the operating system, there exist interrupt-based and program-based event handling methods. Interrupt-based events can be used if the HW interrupt can be controlled directly, however, modern operating systems prevent such handling for system stability. Therefore, it is common to introduce an event handler to first check whether an event has occurred, and further handle it. Even those manufacturing robots can generate and handle relevant events such as when there is an emergency stop switch, or the maximum value of sensor input or output exceeds a certain limit. Meanwhile, the processing time for these events is also important. Emergency stop switches should be used as quickly as possible because they are related to safety.

The comparison of the results of whether the existing middleware satisfy each of the presented requirements, is presented in **Table 2**.

Requirement number		ROS	RT-CORBA	OPRoS	openRTM	OROCOS	XbotCore	CODESYS	RTMIA
R1		O	O	O	O	O	O	O	O
R2	General	△[1]	O[2]	O	O	O	O	O	O
	period of 100 μs	X	△ (25 ms)	△ (10 ms)	△ (10 ms)	△ (10 ms)	△ (1 ms)	n.a	O
	R3	△p[3]	O	△t[4]	△t[4]	△t[4]	△t[4]	n.a	O
	R4	△[5]	O	O	O	O	△	O	O
	R5	△[1]	O	O	O	O	O	O	O
	R6	X	X	X	X	X	X	n.a	X
	R7	X	△	X	X	X	X	n.a	X

n.a, not available.
O, support; X, not support; △, support under some constraints.
[1]Leverage OS features rather than middleware.
[2]Realtime only supports threads.
[3]p means only process type.
[4]t means only thread type.
[5]All source code is public, but all applications are compiled/linked.

Table 2.
Comparing existing middleware to platform requirements.

3. Implementation and motion analysis of real-time robot SW platform

In general, multiple periods are controlled by using the greatest common divisor (GCD) and the least common multiple (LCM) of the periods. If the period, which is the greatest common divisor, is less than the minimum period provided by the middleware, the multiple periods must be adjusted so that the GCD period is equal to or greater than the minimum period. The smaller the provided period, the better it is because it can be used for various applications. Let the GCD period be the basic period and the LCM period be the macro period [15]. **Table 3** illustrates a periodic scheduling table using GCD/LCM based on **Figure 1**.

The middleware must be implemented to execute the periodic threads and processes, the sporadic threads and processes, as well as the non-real-time processes, as shown in **Figure 1**. Periodic threads and processes should be executed within given periods, while sporadic threads and processes should be executed within the deadline once the corresponding events occur. An example of this behavior is shown in **Figure 2**. Of course, non-real-time modules are executed only when execution time remains in a period. As shown in **Figure 2**, periodic execution has the highest priority, followed by sporadic execution. After the execution of periodic modules, sporadic execution checks whether the event occurs and if true, the event handler invokes the corresponding event processing function to handle it. These operations demonstrate that requirements R2, R3, R5, R6, and R7 are satisfied. Especially, we use time-based interrupts to keep the period strict and calculate the n-th jitter, as shown in Eq. (1) [15]:

$$J_n = P_n - T_n = T_0 + n * period - T_n \quad (1)$$

where
T_0: the first execution time of the target module (reference time)
P_n: start time of the n-th period, indicated by $T_0 + n * period$
T_n: start time of the n-th module
$period$: basic period

From Eq. (1) and **Figure 2**, we can observe that it is important to keep the period (n = 0,1,2, ...) correctly. There are two approaches to implement the periodic execution: use timer interrupt and use the sleep function. The second method is the most frequently used one and its usage can be described as follows: after executing the SW modules, the operating system sleeps during the remaining time until the next period. However, this method has a disadvantage that the remaining time period does not keep up correctly, leading to increasing jitter. Therefore, in this study, we use a method of implementing period based on timer interrupts. That is, the minimum period is the minimum interval of the timer interrupt generated by the operating system. This allows middleware to be designed to meet requirements

No. of basic period	Execution time (ms)	Execution module (in priority)	Macro period
0	0	cntr3 > cntr2 > cntr1 > cntr4	Repeat basic periods every 0.5 ms
1	0.1	cntr3 > cntr2	
2	0.2	cntr3 > cntr2	
3	0.3	cntr3 > cntr2	
4	0.4	cntr3 > cntr2	

Table 3.
Periodic scheduling table for ***Figure 1****.*

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- filename: file to be loaded and executed by middleware -->
<!-- moduletype: process(exe type), thread(so or dll type) -->
<!-- operationtype: periodic, sporadic, non-real -->
<!-- period: nano sec -->
<!-- priority: the lowest value is the highest priority -->
<!-- property: input parameters needed to execute the module -->
<root>
  <module> <!-- periodic module with period of 100 us -->
    <filename>./cntr1.so</filename>
    <moduletype>thread</moduletype>
    <operationtype>periodic</operationtype>
    <period>500000</period>
    <priority>3</priority>
    <property>
      <value name="intialize_value">5</value>
    </property>
  </module>
  <module>
    <filename>./cntr2.so</filename>
    <moduletype>thread</moduletype>
    <operationtype>periodic</operationtype>
    <period>100000</period>
    <priority>2</priority>
  </module>
  <module>
    <filename>./cntr3.so</filename>
    <moduletype>thread</moduletype>
    <operationtype>periodic</operationtype>
    <period>100000</period>
    <priority>1</priority>
  </module>
  <module>
    <filename>./cntr4.exe</filename>
    <moduletype>process</moduletype>
    <operationtype>periodic</operationtype>
    <period>500000</period>
    <priority>4</priority>
  </module>
  <module> <!-- sporadic module with deadline of 100 us -->
    <filename>./event1.so</filename>
    <moduletype>thread</moduletype>
    <operationtype>sporadic</operationtype>
    <deadline>1000000</deadline>
    <priority>1</priority>
  </module>
  <module>
    <filename>./visionEvt.exe</filename>
    <moduletype>process</moduletype>
    <operationtype>sporadic</operationtype>
    <deadline>100000000</deadline>
    <priority>2</priority>
  </module>
  <module> <!-- non-real-time module -->
    <filename>./nonReaTime.exe</filename>
    <moduletype>process</moduletype>
    <operationtype>non-real</operationtype>
  </module>
</root>
```

Figure 1.
Example of module.xml.

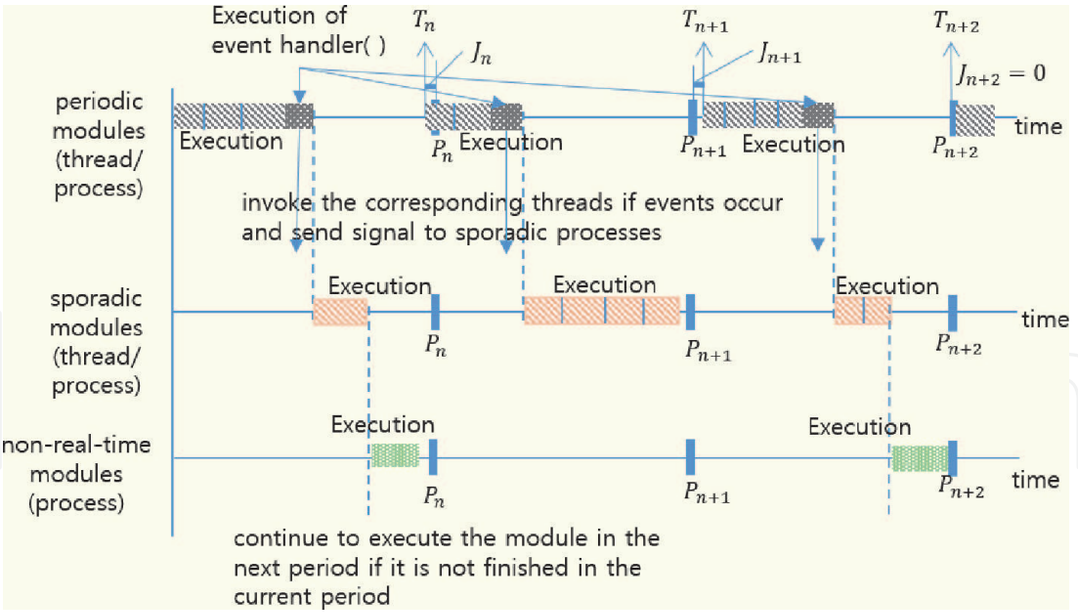

Figure 2.
Example of execution timing diagram of SW modules.

R2 and R5. In addition, the platform should be designed using the POSIX (Portable Operating System Interface) standard.

The real-time robot SW platform should be able to exchange data or remotely perform necessary functions among periodic SW module, sporadic SW module and non-real-time SW module. This is because only cooperation between SW modules can achieve the desired result. For this purpose, the proposed middleware implements a method of obtaining the desired results by performing exchange of data and remote functions using shared memory. The advantage of using shared memory is that it is easy to maintain data consistency and to use the data. In other words, when SW modules including input module write input data to shared memory, other SW modules read them and calculate the control values at the same time. And if necessary, the module writes the result to shared memory for other modules to use the data. In addition, it provides a shared function in the shared memory method so that other SW modules can use the shared functions of the modules. Of course, when using shared memory, the problem of mutual exclusion between threads is solved.

A file named .glb is used to define shared variables that are implemented using shared memory. For example, if a file called knuRobot.glb is created as shown in **Figure 3**, a conversion function creates the header file and a .cpp function to access shared variables. The difference from existing shared variable is that, just like global variables, the shared variable that resides in shared memory can be used in all the software modules. This method makes the program very simple. Example programs that use shared variables in **Figure 3** are shown in **Figures 4** and **5**, in which,

```
// knuRobot.glb
namespace knu.Robot;
uint32_t tick;
struct {
   double x; double y; double z;
} joints[10]
```

Figure 3.
Example code of a file defined shared variables.

```
// output.cpp
#include <indurop/indurop.h> // header file for Middleware
#include <indurop/generated.h> // generated by .glb

using namespace knu::Robot;
void onRun() {
   INDUROP_TYPEOF(joints)::value_type v = joints[0];
   irp::printf("joints[0] = {%lf, %lf, %lf}₩n", v.x, v.y, v.z);
}
INDUROP_MODULE(run = &::onRun)
```

Figure 4.
Example program 1 that reads and prints shared variable defined in ***Figure 3****.*

```
// modify.cpp
#include <indurop/indurop.h> // header file for Middleware
#include <indurop/generated.h> // generated by .glb
 using namespace knu::Robot;
 void onRun() {
    tick = tick + 10;
    INDUROP_TYPEOF(joints)::value_type v = joints[0];
    v.x += tick; v.y += tick; v.z += tick;
    joints[0] = v;
 }
 INDUROP_MODULE(run = &::onRun)
```

Figure 5.
Example program 2 to update shared variables defined in ***Figure 3****.*

"#include <indurop/generated.h>" is a header file that should be added when shared variables are used, which is generated from .glb file during preprocessing. Therefore, simply by adding "indurop/generated.h", the variables 'tick' and 'joints [10]' defined in the knuRobot.glb file can be accessed. The INDUROP_MODULE statement in **Figures 4** and **5** represents information about the module's name, description, author, license, and callback function. The middleware periodically executes the module, where the function to be called is specified as "run = &:: onRun." Furthermore, 'run' is the tag name of the periodically called callback function and "&::onRun" defines the address of the function "void onRun()" presented in each figure. Using INDUROP_TYPEOF(argument) in **Figure 5**, the same data type as the variable defined in .glb file can be created and used.

The event handler in **Figure 5** is invoked using the basic period in **Figure 2**. In other words, it is invoked after executing periodic modules and before processing the sporadic service. Event handlers can be implemented using sporadic threads or sporadic processes. The algorithm of the event handler is shown in **Figure 6**.

Figures 7 and **8** show examples of sporadic thread and process-related programs that allow event handling, respectively.

Figure 6 depicts an algorithm that describes the event handler's operation. It can execute the thread program of **Figure 7** and the process program of **Figure 8**. The event handler first calls the thread's 'condition()' function and if the execution result is true, calls the 'run()' function to handle the event. Then it sends signal to the processes to handle the event occurred. In case of thread, event handling function can be specified separately, but, in this study, 'run()' function is used for simplicity, whose example is shown in **Figure 7**. **Figure 8** shows an example of a process that handles an event. In this example, it utilizes signal because the process cannot be called directly from an event handler. **Figures 7** and **8** also illustrate the use of shared variables.

```
eventHandler()
{
  for (all threads enrolled to event scheduler) {
    call the functions 'condition()' of sporadic threads,
    if (the result of condition() is true)
      invoke run() of the corresponding thread
  }
  for (all processes enrolled to event scheduler) {
    send signal to each process
  }
}
```

Figure 6.
Algorithm for event handler.

```
#include <indurop/generated.h> // generated by .glb
extern "C" int condition(){
    if(joint[0].x >= 180){        // event is occurred
        return 1;
    } else{
        return 0;
    }
}
extern "C" void run(void){
    // process the corresponding event
}
```

Figure 7.
Example code of a sporadic thread program handled by an event handler.

```
//sporadic-process1
#include <indurop/indurop.h>
#include <fstream>

int condition(event){
if(event <0 || event >= 180)
        return 1;  // occur event
else
        return 0;
}
int main(int argc, char* argv[]){
    double y_pos;
    channel::Handle handle;   // connection channel for event handler
    irp::initPeriodExe();    // enrollment of signal to event handler


    // connect channel to process for shared variables ;
    ...
   while(){
        irp::waitPeriod();  // wait signal from event handler
        read_ret = read(handle, "joint[1].y", sizeof(y_pos), &y_pos);
          // read value from shared variable named "joint[1].y" in Fig. 3
        if (condition(y_pos))
             processEvent(y_pos) ;
    }
    return 0;
}
void processEvent(double event)
{
  // process event
}
```

Figure 8.
Example code of a sporadic process program handled by an event handler.

4. Robot system integrating robot controller and PLC

The XML file in **Figure 9** is a configuration file consisting of programs of robot task, PLC, and an input/output data processing. As shown in **Figure 9**, the program consists of three periodic threads, where the period of all SW modules is 10 ms. main.so consists of a ladder program, which is converted to C/C++ program, and a C++ program that calls the program, which is shown in **Figure 12**. The robot task program in **Figure 13** is stored in Robot.so. The reason why we only used thread type is that, comparing to process type, it runs more precisely with real-time. The execution order is as follows: the input/output data processing module, the robot task module, and the PLC module. Of course, users can modify the execution order by changing priorities of modules. Once this configuration file is processed, the robot system will operate as shown in **Figure 10**, which can be configured and operated with the process.

Defined in **Figure 11**, shared variables are used in the integrated application system of **Figure 10**, in the PLC ladder program in **Figure 12** and also in the robot control program in **Figure 13**. The name space 'PLC_ROBOT' is used in **Figure 13**, which makes the program look more sophisticated. As shown in **Figures 11** and **13**,

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
<module>
 <filename>./build/main.so</filename>
 <moduletype>thread</moduletype>
 <operationtype>periodic</operationtype>
 <period>10000000</period>
 <priority>3</priority>
 <property>
   <value name="counter">5</value>
 </property>
</module>
<module>
 <filename>./build/Robot.so</filename>
 <moduletype>thread</moduletype>
 <operationtype>periodic</operationtype>
 <period>10000000</period>
 <priority>2</priority>
 <property>
   <value name="counter">5</value>
 </property>
</module>
<!--module>
<filename>/usr/local/InduRoP/lib/libiodispat.so</filename>
<moduletype>thread</moduletype>
 <operationtype>periodic</operationtype>
 <period>10000000</period>
 <priority>1</priority>
 <property>
   <value name="config">/home/rcmain1/cmd_Demo/ioconfig.xml</value>
 </property>
</module!-->
</root>
```

Figure 9.
Configuration xml file for integrated application system.

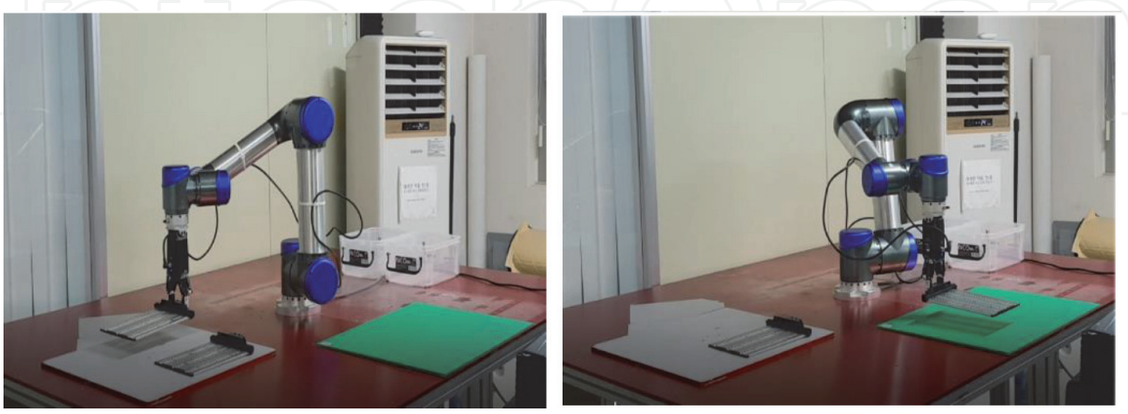

Figure 10.
Results from integrated application systems.

the name space 'PLC_ROBOT' indicates that shared variables are utilized. The PLC ladder program shown in **Figure 12** is converted to C/C++ function named 'POU_PROGRAM()' and made into a thread to run. Through this, various types of

```
namespace PLC_ROBOT;
short I0 // PLC input point 0
short I1 // PLC input point 1
short O0 // PLC output point 0

..... // Code omitted

#define MAX_JOINTS          6
struct {
unsigned short nAXESGROUP;
short MotionDone;
short Busy;
short eState;
short     bRuntimeService;
short     bJob_RobotMoving;          // Motion command interpreting
unsigned long     dCurJoint[6];      // servo current joint angle //// 6 --> MAX_JOINTS
unsigned long     dCmdJoint[6];      // servo command joint angle //// 6 --> MAX_JOINTS
unsigned long     dWorldPos[6];      // world position (command position)
unsigned long     dUserPos[6];       // User position (command position)
unsigned long     dBasePos[6];       // Base position (command position)
unsigned long     dJointSpeed[6]; // 6 --> MAX_JOINTS
unsigned long     dBaseSpeed[6];
unsigned long     dWorldSpeed[6];
unsigned char     nposeINFO[6]; // 6 --> MAX_JOINTS
} ROBOT_MODULE_STATE; // External Information
..
struct
{
int32_t           eOpCode;
unsigned short    t_pl;                        // position level
          unsigned long     t_speed;           // speed
          unsigned short    p_Target;
unsigned short    p_Mid;
} ROBOT_MODULE_SERVICE; // To ROBOT MODULE

..... // Code omitted

struct {
          uint16_t error;
          uint16_t status;
          int8_t mode;
          int32_t position;
} AXIS1_IN;
```

Figure 11.
Shared variables used in integrated application systems.

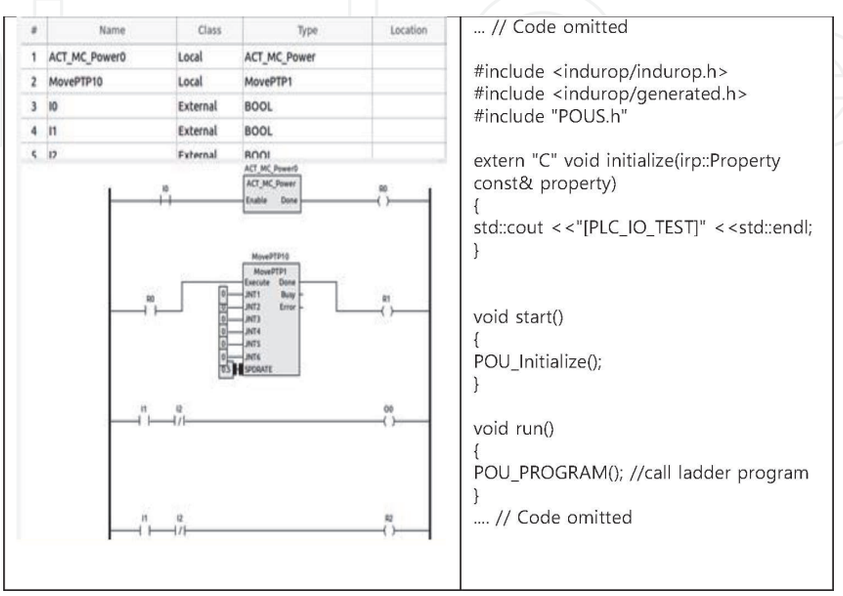

Figure 12.
PLC ladder program example using shared variable.

```
.. // Code omitted

#include <indurop/indurop.h>
#include <indurop/generated.h>

#include "platform.h"
#include "KinematicFunction.h"
#include "MotionControl.h"
#include "MotionPlan.h"
..
void start()
{
int i, j, k;
int nRetCode;

std::cout <<"[START]" <<std::endl;

PLC_ROBOT::ROBOT_MODULE_STATE.eState = plcRobot.eState = ROBOT_GROUP_DISABLED;
PLC_ROBOT::ROBOT_MODULE_SERVICE.eOpCode = SVC_AD_GROUP_ENABLE;
PLC_ROBOT::msgRobotServiceEditNum=1;
plcRobot.plcRobotInfo.nRobotAxis = 6;

for (k=0; k<6; k++)
PLC_ROBOT::ROBOT_MODULE_STATE.dCurJoint[k] = 0.0;
}

void run()
{
int i, j, k;
int nRetCode;
int32_t joint[6];

joint[0] = PLC_ROBOT::AXIS1_IN.position / 1320000 * 360;
joint[1] = PLC_ROBOT::AXIS2_IN.position / 1320000 * 360;
joint[2] = PLC_ROBOT::AXIS3_IN.position / 1320000 * 360;
joint[3] = PLC_ROBOT::AXIS4_IN.position / 1320000 * 360;
joint[4] = PLC_ROBOT::AXIS5_IN.position / 1320000 * 360;
joint[5] = PLC_ROBOT::AXIS6_IN.position / 1320000 * 360;

for (k=0; k<6; k++)
plcRobot.dCurJoint[k] =PLC_ROBOT::ROBOT_MODULE_STATE.dCurJoint[k] = (unsigned long)

if (PLC_ROBOT::msgRobotServiceEditNum!=0) {
msgRobotService[0].eOpCode=
(OPCODE_ROBOT_SERVICE)((int) PLC_ROBOT::ROBOT_MODULE_SERVICE.eOpCode);
msgRobotService[0].arg.eOpCode = OPCODE_MOVJ;
msgRobotService[0].arg.ARG_CMD.argMOVE.t_speed.dValue= 50;
//:ROBOT_MODULE_SERVICE.t_speed;
nRetCode = Handling_Robot_Service(&msgRobotService[0]);
if(PLC_ROBOT::ROBOT_MODULE_STATE.MotionDone==1)
PLC_ROBOT::msgRobotServiceEditNum = 0;

... // Code omitted
N=5 ; // N is Gear Ratio
PLC_ROBOT::AXIS1_OUT.position = N*1320000 * joint[0] / 360;
PLC_ROBOT::AXIS2_OUT.position = N*1320000 * joint[1] / 360;
PLC_ROBOT::AXIS3_OUT.position = N*1320000 * joint[2] / 360;
PLC_ROBOT::AXIS4_OUT.position = N*1320000 * joint[3] / 360;
PLC_ROBOT::AXIS5_OUT.position = N*1320000 * joint[4] / 360;
PLC_ROBOT::AXIS6_OUT.position = N*1320000 * joint[5] / 360;
}
```

Figure 13.
Robot joint control SW module.

PLC programs can be operated. Threads that link shared variables to I/O modules are defined in the libiodispat.so, which is illustrated in the third part of **Figure 11**, and the middleware executes the thread. If the thread libiodispat.so periodically reads sensing data from the input module and stores into shared variables, then control threads of robot of Robot.so and PLC of main.so access shared variables.

5. Conclusions

In this study, we presented seven requirements of real-time robot SW platform that can be used for industrial robot and examined whether existing middleware such as ROS, OPROS, openRTM, and RTMIA [15] satisfy these requirements. In particular, communication middleware such as ROS has a disadvantage of demanding the user to have more knowledge about the real-time operating system to use the industrial robot but advantage that its usage is simpler because it does not manage execution of processes and/or threads. On the other hand, OPRoS, openRTM, and OROCOS manage the execution of threads for periodic execution but does not control the execution of processes.

In this study, we proposed the real-time robot SW platform that satisfied the presented seven requirements R1–R7 by extending RTMIA and demonstrated its implementation on the Xenomai real-time operating system and Linux. The proposed SW platform utilized the timer-interrupt based approach to keep strict period and the shared memory for convenient usage.

We applied our method to a practical robot system, wherein the programs of PLC and the robot were used simultaneously, and their corresponding operating results were also presented. In this implementation, PLC ladder program and robot control program were managed using period of 10 milliseconds. As the implemented application was simple, it was not shown that event handling and execution of periodic processes were working well. But they did work well. That is, it can be known that the proposed platform satisfied requirements R1–R7. As a result, the platform proposed in this study was verified.

The other advantage of this study is the method to access shared variable. It can be known that it is generally easy and convenient to read and write the shared variables using the conventional variable access method when the shared variables defined in other threads are accessed. Hence the proposed method for accessing of shared variables is designed for multiple processes or threads to have mutually exclusive access to shared variables using the conventional variable access method. And it is shown that the proposed method for accessing of shared variables is working well.

Future work may include investigating a robotic platform that optimally operates multiple threads in multicore systems. And the proposed platform will be implemented on the μC/OS.

Acknowledgements

This work was partially supported by KITECH research fund and Korea Evaluation Institute of Industrial Technology (KEIT) grant funded by the Ministry of Trade, Industry & Energy (KOREA) (No. 20004535 and No. 20005055).

DOI: http://dx.doi.org/10.5772/intechopen.91014

Author details

Sanghoon Ji[1], Donguk Yu[2], Hoseok Jung[1] and Hong Seong Park[3]*

1 Convergent Technology R&D Division, Robot R&D Group, Korea Institute of Industrial Technology, Gyeonggi-do, South Korea

2 Navcours, Daejeon, South Korea

3 Department of Electrical and Electronic Engineering, Kangwon National University, Gangwon-do, South Korea

*Address all correspondence to: hspark@kangwon.ac.kr

References

[1] Mahmoud QH. Middleware for Communication. Chichester: John Wiley and Sons, Ltd.; 2004. DOI: 10.1002/0470862084

[2] Pérez H, Gutiérrez JJ. Mint: A survey on standards for real-time distribution middleware. ACM Computing Surveys. 2014;**46**(4):49. DOI: 10.1145/2532636

[3] OMG. CORBA Core Specification. v3.3 [Internet]. 2012. Available from: https://www.omg.org/spec/CORBA/3.3/

[4] OMG. Real-time CORBA Specification. v1.2 [Internet]. 2005. Available from: https://www.omg.org/spec/RT/About-RT/

[5] OMG. Data Distribution Service for Real-Time Systems. v1.4 [Internet]. 2015. Available from: https://www.omg.org/spec/DDS/

[6] OPC Foundation. OPC Unified Architecture Specification Part 1: Overview and Concepts. 2017. Available from: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-1-overview-and-concepts

[7] Open Source Robotics Foundation. ROS (Robot Operating System) [Internet]. Available from: www.ros.org

[8] Kay J, Tsouroukdissian AR. "Real-time Performance in ROS 2," ROSCon2015 [Internet]. 2015. Available from: https://roscon.ros.org/2015/presentations/RealtimeROS2.pdf

[9] OPRoS (Open Platform for Robotic Services). [Internet]. Available from: https://github.com/opros-wiki/OPRoS_v1.1_Components/wiki/Open-Platform-for-Robotic-Services

[10] Han S, Kim M, PHS. Mint: Open software platform for robotic services. IEEE Transactions on Automation Science and Engineering. 2012;**9**(3): 467-481. DOI: 10.1007/978-3-319-11900-7

[11] Lee D, AH. Mint: Real-time characteristics analysis and improvement for OPRoS component scheduler on windows NT operating system. Journal of Institute of Control, Robotics and Systems (in Korean). 2011; **17**(1):38-46. DOI: 10.5302/J.ICROS.2011.17.1.38

[12] openRTM (open Robotics Technology Middleware). [Internet]. Available from: http://openrtm.org/

[13] OROCOS (Open Robot Control Software) project. [Internet]. Available from: http://www.orocos.org/

[14] Muratore L, Laurenzi A, Hoffman EM, Rocchi A, Caldwell DG, Tsagarakis NG. Mint: On the design and evaluation of XBotCore, a cross-robot real-time software framework. Journal of Software Engineering for Robotics. 2017;**8**(1):164-170. DOI: 10.6092/JOSER_2017_08_01_p164

[15] Park HS. Real-time scheduling method for middleware of industrial automation devices. In: Open Access Peer-Reviewed Chapter—Online First. 2019. DOI: 10.5772/intechopen.86769

[16] CODESYS. [Internet]. Available from: https://www.codesys.com

[17] TWINCAT. [Internet]. Available from: https://www.beckhoff.com/twincat/

[18] PLCopen. [Internet]. Available from: https://plcopen.org/

[19] Meier L, Honegger D, Pollefeys M. Mint: PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In: 2015 IEEE International Conference on Robotics and Automation (ICRA); Seattle, WA, USA; 26–30 May 2015. DOI: 10.1109/ICRA.2015.7140074