

We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800

Open access books available

122,000

International authors and editors

135M

Downloads

Our authors are among the

154

Countries delivered to

TOP 1%

most cited scientists

12.2%

Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



Real-Time Scheduling Method for Middleware of Industrial Automation Devices

Hong Seong Park

Abstract

In this study, a real-time scheduling algorithm, which supports periodic and sporadic executions with event handling, is proposed for the middleware of industrial automation devices or controllers, such as industrial robots and programmable logic controllers. When sensors and embedded controllers are included in control loops having different control periods, they should transmit their data periodically to the controllers and actuators; otherwise, fatal failure of the system including the devices could occur. The proposed scheduling algorithm manages modules, namely, the thread type (or .so type) and process type (or .exe type), for periodic execution, sporadic execution, and non-real-time execution. The program structures for the thread-type and process-type modules that can make the proposed algorithm manage the modules efficiently are suggested; then, they are applied in periodic and sporadic executions. For sporadic executions, the occurrences of events are first examined to invoke the execution modules corresponding to the events. The proposed scheduling algorithm is implemented using the Xenomai real-time operating system (OS) and Linux, and it is validated through several examples.

Keywords: real-time scheduler, middleware, automation device, industrial robot, PLC, periodic execution, sporadic execution, thread type, process type

1. Introduction

Currently, there are many studies about Industry 4.0 [1–3], where numerous industrial automation devices such as industrial robots, programmable logic controllers (PLCs), and industrial Internet of things (IIOT) are used. Manufacturing processes in smart factories have achieved increased flexibility through robots, PLCs, and smart devices, which enable the production of various types of products. Because the robots and PLCs used in these factories are working with humans, the safety of human workers is critical and should be guaranteed. Note that PLCs generally control conveyors and the flow of production processes. In particular, sensors and embedded controllers should transmit their data periodically to the controllers and actuators according to the preset control periods if they are included in control loops having different periods. If some data transmissions fail, fatal failure of the system including the devices could occur. Hence, real-time characteristics, namely, periodic and sporadic, are extremely important for those devices. The periodic characteristic is required for operating the manufacturing system stably and safely, whereas the sporadic characteristic is needed to cope with safety issues.

In addition, mobile manipulators, which consist of mobile platforms and industrial robots, can make the production line more flexible. If this flexibility is expanded to some extent, the manufacturing process can become reconfigurable. However, it is required that the production line is also reconfigurable. A conventional production line is based on a long conveyor system controlled by a PLC, which can obstruct the reconfigurability of the production line. Moreover, the PLC is one of the most important automation devices, and its functional specifications including motion controls are standardized [4–5]. Hence, it is currently being implemented in the software (SW) of embedded controllers and used widely in industrial fields such as smart factories and industrial robots.

In general, industrial robots used in factories utilize PLCs because they must be able to move parts from one cell to another or assemble parts in a cell. Thus, if the production line is composed of two or more cells, which are implemented as moving units based on robots and PLCs, the production line can be reconfigurable. This means that the industrial robot systems used in the lines must perform various types of functions such as manipulation and moving of parts. Hence, the controller of industrial robot systems used in lines manages the motion control SW for manipulation and the PLC function for conveyors and grippers. Some current PLC products can simultaneously manage both conveyors and industrial robots but not all types of industrial robots [6, 7]. Industrial robots and PLCs can control both motion SW for manipulation and PLC functions. Furthermore, they should exchange data via communication among servers and various types of sensors because the program and measured data must be transmitted to other automation devices.

Information technology (IT) is at the center of these technologies. It is however difficult to integrate the rapidly developing IT with conventional robot systems and PLCs. Consequently, middleware technology has been studied for automation devices such as robots and PLCs [4–19]. The middleware used in automation devices manages the processes/threads related to manipulations, vision recognition, PLC functions, transmission of various types of data, safety, and security. This article focuses on the management of processes/threads, which is called real scheduling.

There are some middleware that can be used for automation devices [6–21]. Well-known examples are the CORBA [20], OPC-UA [21], the ones used in CoDeSys [6, 7] and TwinCAT [8], ROS [9, 10], OPRoS [11, 12], openRTM [13, 14], and OROCOS [15, 16]. Among these examples, the OPC-UA and ROS are a type of communication middleware. The ROS manages the execution periods of SW modules using the sleep function, but when SW modules are executed as a process type, it is difficult to keep the period of these modules accurate. Hence, most of the real-time operating systems (OSs) utilize the thread type to keep real-time characteristics. The CoDeSys and TwinCAT support a runtime system that executes control SW modules in real time, which is thought of as a type of middleware. Note that control SW modules used in the CoDeSys and TwinCAT are motion modules for manipulation and PLC functions. The ROS, OPRoS, openRTM, and OROCOS are types of middleware used in robot technology. The CORBA is the most famous middleware supporting communication and management of SW modules, but it is difficult to be implemented in automation devices due to the large size of SW.

The ROS is a popular open SW in the robot field and has been mainly implemented on Linux, but the OPRoS, OROCOS, and openRTM are performed on various types of OSs such as Windows, Linux, and real-time OS. The former executes SW modules as process types, whereas the latter executes SW modules as thread types. Note that general users can use the process type with ease but have difficulty in using the thread type because of its debugging issues and special format. However, the real-time characteristics of SW modules are kept more easily in the thread type. Hence, most of the real-time OSs provide only thread types of SW

modules for real time. Because the middleware is utilized in various types of OSs including real-time OSs, it should have a real-time scheduler, which can manage the SW modules in real-time whether they are of process type or thread type. Note that the OPRoS, OROCOS, and openRTM support only thread types of SW modules for real-time services.

Real-time services can be classified into periodic services and sporadic services. In particular, sporadic services of an SW module are processed as follows: the middleware checks the occurrence of an event and then executes the SW module related to that event. Hence, it is necessary for the middleware to support the event handling of sporadic services. Middleware systems applied to industrial robot controllers are the OPRoS and OROCOS, but they do not support sporadic services in real time.

Real-time schedulers of middleware are generally designed and developed based on the execution lifecycle (or state machine), which consists of some states such as IDLE, EXECUTING, DESTROYED, and ERROR and is described in the next section. Note that the execution lifecycle is applied only to thread-type SW modules but not to process-type SW modules. In other words, the scheduler controls state transitions to enter into the target state and then manage the real-time threads in a safe manner. However, it is difficult for process-type SW modules to be managed by middleware as seen in the ROS. Hence, the OPRoS, openRTM, and OROCOS manage only the thread-type modules. If the real-time scheduler processes the modules as independent threads, the overhead time including context switching can be critical in the case where the shorter period (e.g., 100 μ s) is used. The overhead time needs to be reduced. In addition, it is necessary to consider the execution of process-type SW modules so that users can utilize the middleware easily.

Industrial automation devices used in Industry 4.0 should have flexibility, which can be provided by middleware with real-time schedulers and reliable communication. Real-time schedulers play important roles in supporting reliable and real-time communication. Real-time schedulers for industrial automation devices such as industrial robots [22] should have the following properties of P1–P6:

- (P1) support both process and thread types as execution models of SW modules.
- (P2) support periodic services and sporadic services as real-time services.
- (P3) support non-real-time service if necessary.
- (P4) keep jitters within the minimum bound.
- (P5) support the user-defined priority for each SW module.
- (P6) support the configuration of the SW modules that users can set up.

Examples of processes and threads can be motor control modules, multiple robot control modules, kinematic modules, path planning modules, PLC modules, human-robot interaction modules, and object recognition modules. Motor control modules are executed according to different periods, and PLC modules can be performed cyclically or periodically.

This study proposes a real-time scheduler that satisfies the properties listed above. To reduce the overhead time among threads, the proposed scheduler calls directly the related methods (or functions) of modules in the thread type, where the modules are loaded in .so type in Linux. In addition, it checks the event occurrences to process the corresponding SW modules as sporadic services and then invokes the corresponding SW modules if the related event condition is satisfied.

For this purpose, the middleware provides an event handling function. This study implements the proposed scheduler using Xenomai [23]. Some examples are given to validate the proposed scheduler and show that the worst-case jitters in thread/process types of modules are kept within the minimum bound and that the middleware is performed on Xenomai and Linux.

In Section 2, the requirements of real-time schedulers for middleware of industrial automation devices are proposed. The real-time scheduling algorithm and the program structures for periodic and sporadic executions are suggested, where those executions based on the state machine are related to the thread-type modules. In Section 3, some examples are presented to validate the proposed scheduler. Finally, the conclusions drawn are given in Section 4.

2. Real-time scheduler for middleware of industrial automation devices

2.1 Requirements

In industrial automation devices, such as industrial robots and PLCs for process control, most of the SW modules should be executed periodically. Obviously, PLCs used in discrete I/O controls such as control of conveyors are executed cyclically. Moreover, embedded controllers used in automation devices can execute both manipulation control of industrial robots and control of digital I/Os. Because SW modules used in those automation devices may have different execution periods, it is necessary to set the execution periods smoothly according to the target applications. For example, let us consider SW modules A, B, and C in application 1, which are executed at periods of 10, 30, and 20 ms, respectively. The same modules can be executed at periods of 15, 60, and 30 ms in application 2. The period of a module can vary depending on the application even though the same module is used; thus, it is necessary to set the period smoothly. In general, two terms, namely, basic period and macro period, are utilized in periodic applications. The former is computed using the greatest common divisor of the periods of the modules in the given application, whereas the latter is computed using their least common multiple.

The proposed real-time scheduler is designed and implemented to satisfy the following requirements, which are derived from properties P1–P6 mentioned in Section 1:

- Should support periodic services, sporadic services, and non-real-time services.
- Periodic/sporadic services are divided into thread and process types, and the corresponding information should be provided.
- Should support the process types of legacy SW modules, which can be performed in periodic, sporadic, and non-real-time modes.
- Should be triggered by an event so that sporadic services are performed.
- The event condition is enrolled so that the event handling function can be processed. If the enrolled event condition is satisfied, the corresponding sporadic service is invoked, regardless of the type (whether process type or thread type).
- The event handling function is executed periodically to check the event conditions.

- Periodic services have the highest priority and sporadic services the next priority. Periodic modules and sporadic modules can have different priorities as independent modules, regardless of the type.
- Should use a timer-based operation mode to keep jitters of thread and processes within the minimum bound.
- Should utilize a configuration file written in XML so that users can provide information related to the SW module to the middleware.

Particularly note that events considered in this study are not hardware-driven types because the middleware is executed over the OS. The scheduler in the next subsection is proposed based on these requirements.

2.2 Algorithm for real-time scheduler

The information for users to provide to the middleware is listed below:

- Module types (thread or process)
- Service/operation mode (periodic, sporadic, or non-real-time)
- Module name (file name)
- Period (for periodic service) or deadline (for sporadic service)
- Priority (lower priority or higher)
- Property (input parameters needed to execute the file)

The middleware reads the XML file containing this information and processes it accordingly.

An example of a file in which the above information is stored is shown in **Figure 1**, and its file name is `module.xml` written in XML. Note that the time unit in the file is nanosecond (ns).

Figure 2 shows a brief algorithm of the proposed real-time scheduler. In `main()` function, the algorithm reads the configuration file named `module.xml` and builds two tables, i.e., periodic scheduling table and sporadic scheduling table, according to the computed basic period and priorities of modules. After that, the method “`scheduler()`” and the basic period are set to link to the timer interrupt and then are periodically executed according to the basic period. The method `scheduler()` manages the periodic and sporadic threads and processes. The periodic and sporadic processes are managed via signals of `scheduler()`. Non-real-time modules are executed after the thread of `scheduler()` has linked to the timer interrupt routine. That is, the execution of the non-real-time modules is independent of the proposed real-time scheduler. The scheduler does not manage the non-real-time modules to reduce the computation time.

The middleware reads the configuration file such as `module.xml` in **Figure 1** and computes the basic period of $100\ \mu\text{s}$ and the macro period of $600\ \mu\text{s}$. Using these two periods, the middleware generates the periodic scheduling table shown in **Figure 3** and the sporadic scheduling table shown in **Figure 4**. Note that `pRun()` denotes the pointer of the function `run()`, which is described in Section 2.3.

```

<?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>build/controller1.so</filename>
    <moduletype>thread</moduletype>
    <operationtype>periodic</operationtype>
    <period>600000</period>
    <priority>3</priority>
    <property>
      <value name="counter">5</value>
    </property>
  </module>
  <module>
    <filename>./control2.so</filename>
    <moduletype>thread</moduletype>
    <operationtype>periodic</operationtype>
    <period>100000</period>
    <priority>2</priority>
  </module>
  <module>
    <filename>./control3.so</filename>
    <moduletype>thread</moduletype>
    <operationtype>periodic</operationtype>
    <period>100000</period>
    <priority>1</priority>
  </module>
  <module>
    <filename>./control4.exe</filename>
    <moduletype>process</moduletype>
    <operationtype>periodic</operationtype>
    <period>300000</period>
    <priority>4</priority>
  </module>
  <module> <!-- sporadic module with deadline of 100 us -->
    <filename>./emergency.so</filename>
    <moduletype>thread</moduletype>
    <operationtype>sporadic</operationtype>
    <deadline>1000000</deadline>
    <priority>1</priority>
  </module>
  <module>
    <filename>./vision.exe</filename>
    <moduletype>process</moduletype>
    <operationtype>sporadic</operationtype>
    <deadline>100000000</deadline>
    <priority>2</priority>
  </module>
  <module> <!-- non-real-time module -->
    <filename>./monitoring.exe</filename>
    <moduletype>process</moduletype>
    <operationtype>non-real</operationtype>
  </module>
</root>

```

Figure 1.
Module.xml file.

```
main()
{
    read configuration file
    calculate the basic period and the macro period
    load the thread-type SW modules and the process-type SW modules
    set the periodic scheduling table and sporadic scheduling table.
    Invoke start() methods of the periodic modules and sporadic modules
    set basic period and the thread, scheduler(), to timer interrupt
    execute the non-real-time modules defined in the configuration file
    set this process to the lowest priority
    wait(); // wait for all processes and threads to stop
}
// execute the scheduler in thread mode and manage the periodic and
// sporadic modules.

scheduler()
{
    get the entry from one row of the periodic scheduling table
    while(entry != NULL) {
        // the corresponding method is called if thread type
        if the entry is .so type,
        call the function pRun() of the entry
        else // process type
        send signal to the process to continue its execution
        get the next entry from the row
    }
    increase the row by one to move to the next row of the periodic scheduling table
    get the entry from the sporadic table
    while(entry != NULL)
    {
        if (the entry is .so type) {
            if (condition() of the entry is satisfied)
                call the function pRun() of the entry
        }
        else
            send signal to the sporadic process to continue its execution
    }
} // while
}
```

Figure 2.
Brief algorithm for the proposed real-time scheduler.

In **Figure 3**, the index is the execution order. That is, four periodic modules are executed in the first period (index 0) starting at 0 μ s. In the first period, control3.so is first executed, and control4.exe is executed last according to the priority in **Figure 1**. Control1.so is executed once every 600 μ s, and control4.exe is executed once every 300 μ s. Obviously, the real-time scheduler distinguishes threads and processes and executes them properly. The sporadic scheduling table is shown in **Figure 4**. The sporadic modules are listed in order of priority in the table. For execution of the SW modules, the function pointers and the process IDs are stored according to the type (thread or process) in the scheduling table.

index	name of module (its symbol) - ordered by priority
0	control3.so(c3), control2.so(c2), controller1.so(c1), control4.exe(c4)
1	control3.so, control2.so,
2	control3.so, control2.so,
3	control3.so, control2.so, control4.exe
4	control3.so, control2.so,
5	control3.so, control2.so,

Figure 3.
Example of periodic scheduling table based on Figure 1.

index	name of module (priority order)
0	emergency.so, vision.exe

Figure 4.
Example of sporadic scheduling table based on Figure 1.

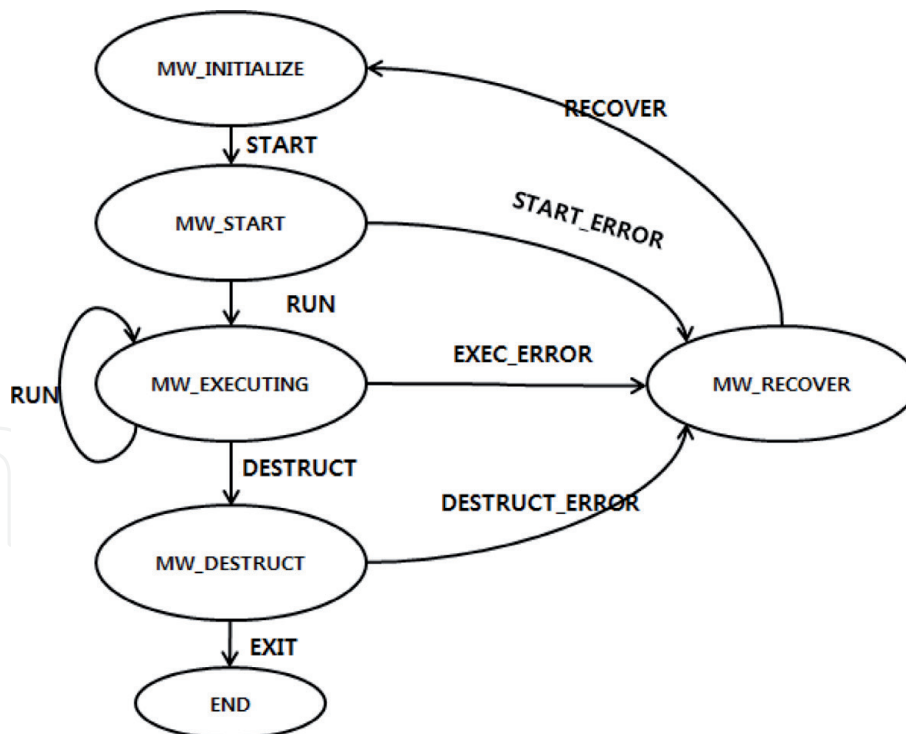


Figure 5.
Execution lifecycle of thread-type SW module.

The thread modules are executed according to the execution lifecycle shown in **Figure 5**. After loading the thread-type SW modules, the module is initialized by the method initialize(), which is illustrated in **Figure 8**, and the module enters into the MW_INITIALIZE state. If the method start() is invoked, the module enters into the MW_START state. After all the thread-type modules enter into the MW_START states, execution of the real-time scheduling algorithm, scheduler() is started by

invoking the method `run()`, which is also shown in **Figure 8**. The module is periodically called or receives signal at the `MW_EXECUTING` state. After completion of the module execution, the module invokes the method `destroy()` and then enters into the `MW_DESTRUCT` state, and consequently, the execution of the module is ended. In the `MW_RECOVER` state, some error handling is possible using recover-related methods. Note that the scheduler directly calls or invokes the method `run()` of modules to reduce the OS overhead time such as context switching time.

Process-type modules are also executed, and they immediately wait for the period-starting signal. If a module receives signals from the scheduler, the module is re-executed from the waiting position.

The operation of the scheduler in **Figure 2** is shown in **Figure 6**. In this figure, it can be observed that the non-real-time modules can be executed only when a sufficient idle time remains in one period. The scheduler calls the `run()` method of the modules to reduce the overhead of controlling the threads or processes, where the `run()` method is shown in **Figures 7** and **8**. After the executions of periodic modules are finished in a period, the scheduler checks the event conditions of sporadic modules. If the condition is true, the scheduler calls the corresponding sporadic method for thread types and sends the signal to the sporadic module for process

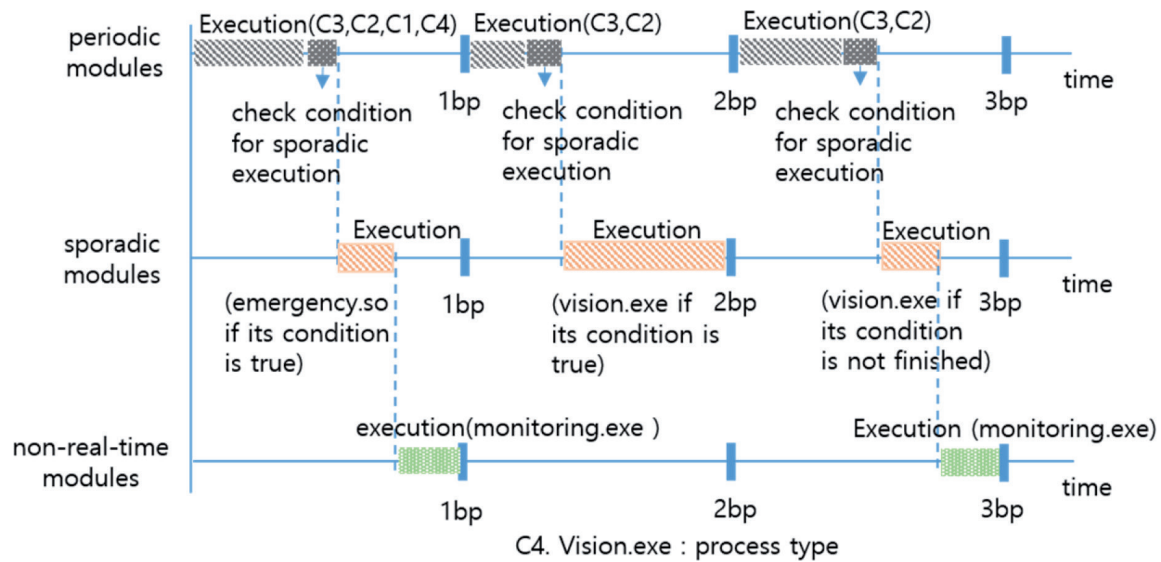


Figure 6. Operation of real-time scheduler for periodic, sporadic, and non-real-time services based on **Figures 3** and **4**.

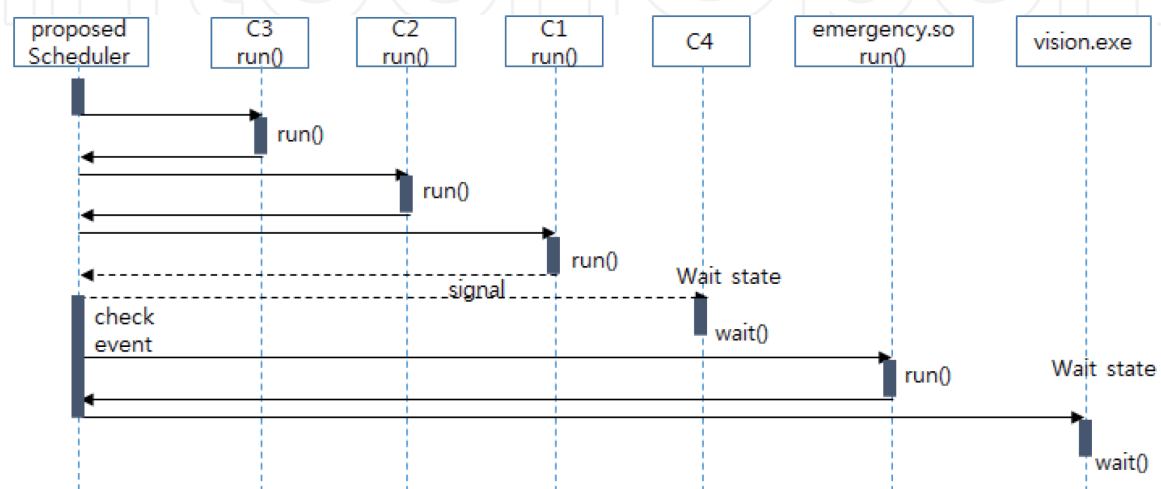


Figure 7. Example of control scheme of real-time scheduler for periodic and sporadic modules.

```

extern "C"{
    void initialize(irp::Property const& property){
        ... //user program for initialize() method
    }
    void start(void){
        ... //user program for start() method
    }
    void run(void){
        ... //user program for run() method
    }
    void error(int type){
        ... //user program for error() method
    }
    void destruct(void){
        ... //user program for destruct() method
    }
    void recover(void) {
        ... //user program for recover() method
    }
}

```

Figure 8.
Program structure of thread-type periodic module.

type. Note that the modules are executed over the multicore CPU and modules with thread types and process types are executed on different cores. Hence, two types of modules can be executed at the same time.

Periodic and sporadic operations can be divided into thread type and process type as shown in **Figure 6**. The scheduler manages the SW modules according to their types. **Figure 7** shows an example of a control scheme of a real-time scheduler for periodic and sporadic modules, where the thread-type modules are directly called by the scheduler and the process-type modules are executed by the signal from the scheduler. As shown in **Figure 7**, the periodic modules are executed first. The scheduler executes a module by calling the run() method of the corresponding module in the .so file, where the module has a thread type. To execute a process-type module, the scheduler sends a signal to the process of the module. Note that process-type modules are executed independently of the scheduler as a type of process and the scheduler is also a type of process.

The program structures of thread-type and process-type modules are described in the next subsection.

2.3 Program structures for thread-type and process-type modules

Because the operation method of a thread-type periodic module is different from that of a process type, the program structure of the thread-type periodic module should be different from that of the process type, which are shown in **Figures 8** and **9** respectively. The method initialize() is executed immediately after the module is loaded in the memory. The methods start(), run(), destruct(), error(), and recover() are called

```
int main(int argc, char argv[]){
    initPeriodExe();
    ... //user program for initialize
    while(1){
        waitPeriod();
        ... //user periodic body
    }
    return 0;
}
```

Figure 9.
Program structure of process-type periodic module.

```
extern "C"{
    int condition(){
        check condition;
        if(condition)
            return TRUE; // condition is met.
        else
            return FALSE; //otherwise
    }

    void initialize(void){
        ... //user event initialize
    }
    void start(void){
        ... //user program for start() method
    }
    void run(void){
        ... //user event body
    }
    // similar methods to the .so-type periodic module
    // destruct(), error(), recover() are added
}
```

Figure 10.
Program structure of .so-type sporadic module.

when events such as START, RUN, DESTRUCT, XXX_ERROR, and RECOVER occur, respectively, which are shown in **Figure 5**.

Users should insert proper codes into parts named user program. In general, a legacy program has a structure of a process type, which is simpler than that of a thread type. To use a legacy program on the proposed middleware, two functions

```

int condition() {
    waitSporadic(); // wait for signal
    check condition;
    if(condition)
        return TRUE; // condition is met.
    else
        return FALSE; //otherwise
}
int main(int argc, char argv[]){
    initSporadicExe();
    ... //user program for initialize
    while(1){
        if (!condition()) // if not satisfied
            continue; // enter into wating state
        ... //user sporadic body
    }
    retrun 0;
}

```

Figure 11.
Program structure of process-type sporadic module.

initPeriodExe() and waitPeriod() should be added in it. initPeriodExe() is a function for enrollment of the corresponding process to the scheduler, and waitPeriod() is a type of function to wait for a signal from the scheduler. Note that the scheduler sends a signal to a process when the corresponding process wants to be executed. Upon receiving the periodic signal, the module transitions from the waiting state to the executing state and then executes the main body, which is the part marked user periodic body in **Figure 9**. The module enters into the waiting state by the waitPeriod() function.

After the execution of periodic modules, the scheduler checks whether any events for sporadic modules have occurred. The modules corresponding to such events are listed in order of priority using EDF (earliest deadline first) method [6–9]. As shown in **Figures 2** and **6**, the scheduler checks periodically by calling or invoking the condition() function in **Figures 10** and **11**. If condition() returns a value of TRUE in **Figure 10**, the scheduler executes the corresponding .so-type module. The process-type sporadic module is designed so that it receives sporadic signals from the scheduler, checks its condition, and executes the user execution body if condition() is satisfied.

3. Evaluation

Experiments were performed using the test cases in **Table 1** on a PC with the following specifications to validate the proposed scheduling algorithm:

- Ubuntu 14.04 LTS (64 bit), kernel: Linux 4.1.18
- Xenomai 3.0.3, ipipe-core-4.1.18
- CPU: Intel(R) Core(TM) i7-7700 CPU @ 3.60 GHz, four cores

The purpose of the experiments was to determine how the periodic modules are affected by the execution of all types of modules. Hence, the worst-case jitter and related jitter statistics are measured and analyzed. Let J_n , T_n , and P_n denote the n th jitter, the starting time of the n -th execution of the target, and the starting time of the n th period, respectively. The jitter considered in this article is calculated as follows:

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

where T_0 denotes the reference time of the target module and period, period denotes a basic period, and $P_n = T_0 + n * \text{period}$.

The modules were executed using a basic period of 100 μs . In **Table 1**, the type of measured module indicates whether the jitter was measured in a .so (thread) or .exe (process) module. The test results are also presented in **Table 1** and **Figures 12–19**. Note that the jitter is computed using Eq. (1) and it is measured in a special module of periodic modules and the type of the measured module is given in **Table 1**.

Table 1 and **Figures 12–19** show that the ranges of mean values and variances of the process-type periodic module in the test cases are -5.5519 μs to -5.991 μs and 11.598–12.438 μs , respectively. The ranges of mean values and variances of the thread-type periodic module in the test cases are -0.027 μs to -0.105 μs and 0.716–3.772 μs , respectively. The worst-case jitter is 12.438 μs , which is measured in the process-type periodic module, and the jitter rate is 12.438% with respect to the basic period of 100 μs . The worst-case jitter measured in the thread-type periodic

No.	Test cases					Test results			
	Number of periodic modules		Number of sporadic modules		Number of non-real-time modules	Type of measured module	Jitter mean (μs)	Jitter variance (μs)	Worst-case jitter (μs)
	.so	.exe	.so	.exe					
1	1	0	0	0	0	Thread	-0.027	0.000839	3.772
2	15	0	0	0	0	Thread	0.127	0.002037	2.933
3	5	3	0	0	0	Thread	-0.042	0.001263	2.793
4	5	3	0	0	0	Process	-5.723	19.480341	11.598
5	5	3	11	0	0	Thread	0.033	0.001174	0.716
6	5	3	11	0	0	Process	-5.580	19.624376	11.614
7	5	3	5	6	0	Thread	-0.105	0.001633	1.405
8	5	3	5	6	0	Process	-5.991	21.136330	12.438
9	5	3	5	5	8	Thread	0.007	0.001424	1.104
10	5	3	5	5	8	Process	-5.519	20.617964	11.758

Table 1.
 Test cases for evaluation of scheduling algorithm.

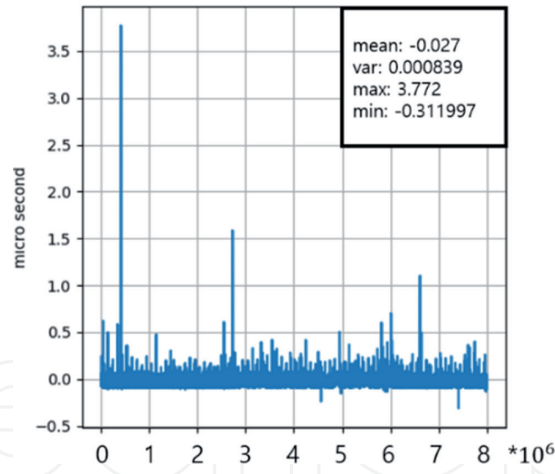


Figure 12.
Jitters in test case 1.

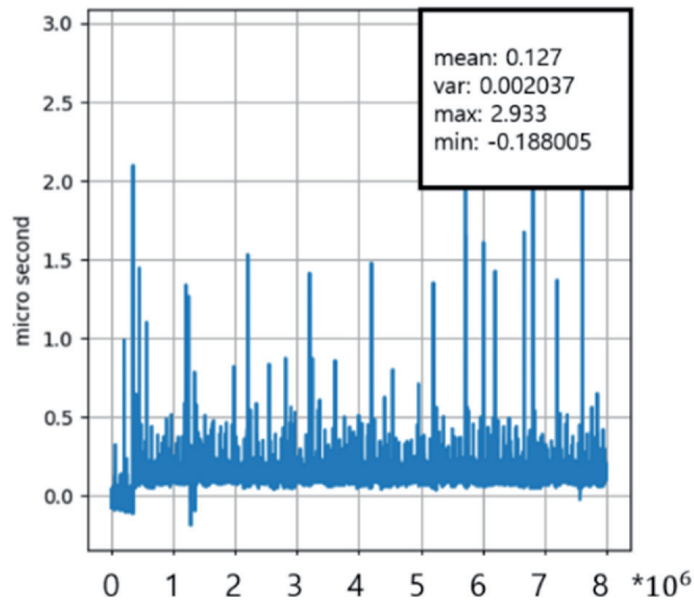


Figure 13.
Jitters in test case 2.

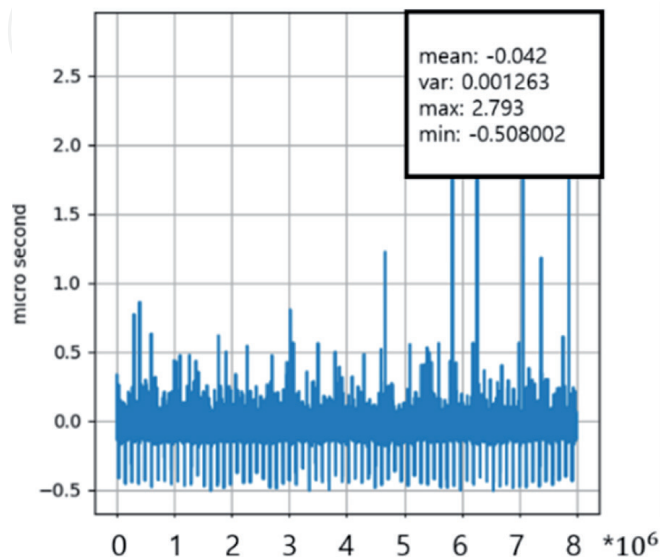


Figure 14.
Jitters in test case 3.

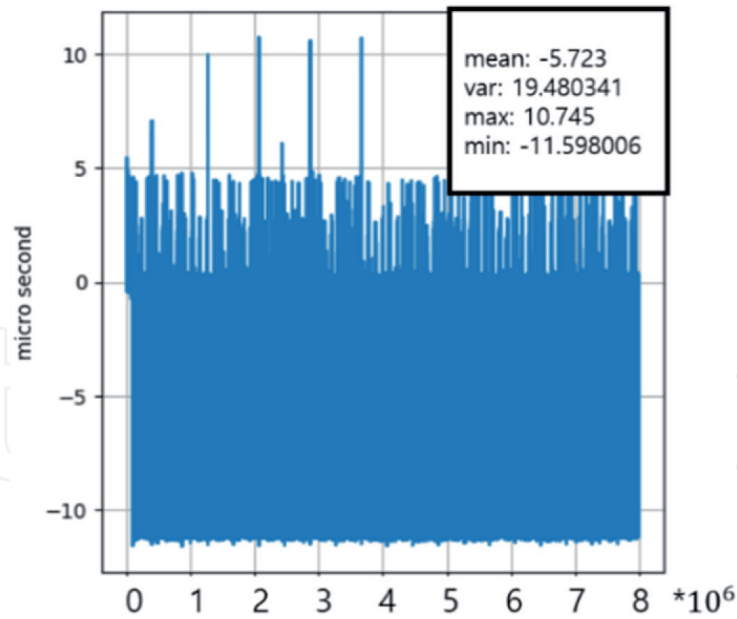


Figure 15.
Jitters in test case 4.

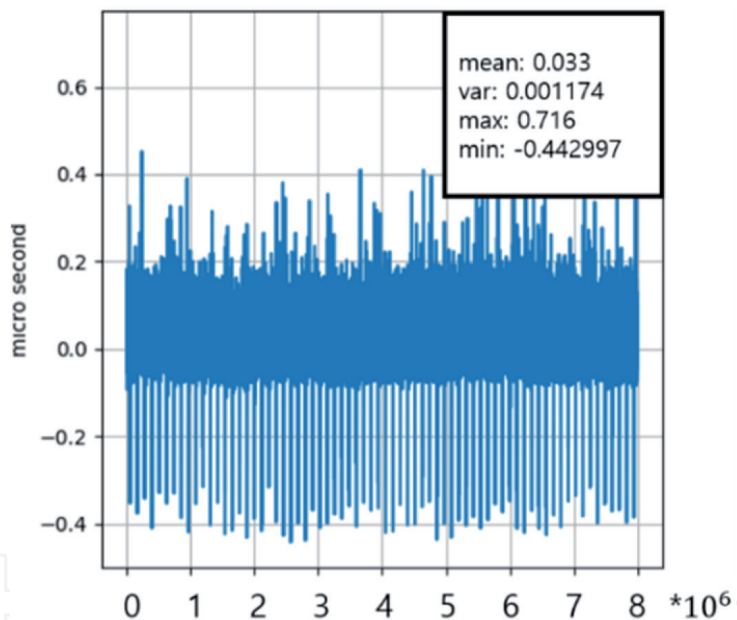


Figure 16.
Jitters in test case 5.

module is 2.933 μs , which is the result of test case 2, and the jitter rate is 2.933% with respect to 100 μs .

For the thread-type periodic module, as the load increases, the variation amount of jitters also increases; however, the worst-case jitter does not exceed 4 μs , and the change in the jitter variances is not significant. For the process-type periodic module, as the load increases, the variation amount of jitters increases significantly; however, the worst-case jitter does not exceed 12.5 μs , and the changes in the jitter variances and worst-case jitters are not large. The test results in **Table 1** and **Figures 11–18** indicate that the proposed scheduling algorithm can be efficiently used by industrial automation devices even for various types of applications.

It is evident from the results in **Table 1** and **Figures 12–19** that the basic period is maintained very satisfactorily in all test cases. This means that the proposed

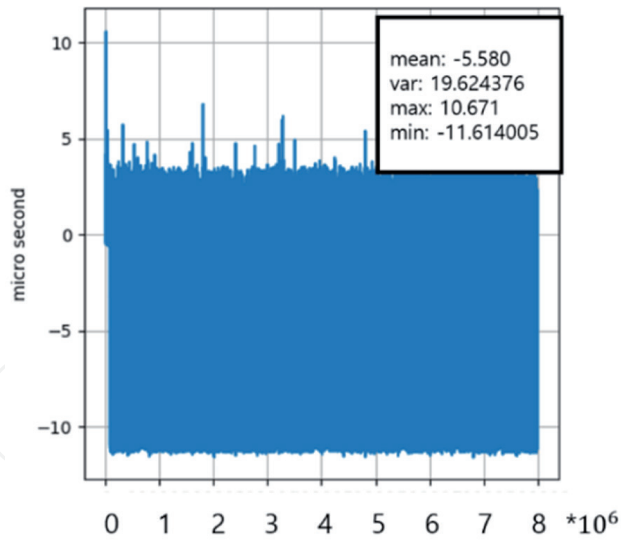


Figure 17.
Jitters in test case 6.

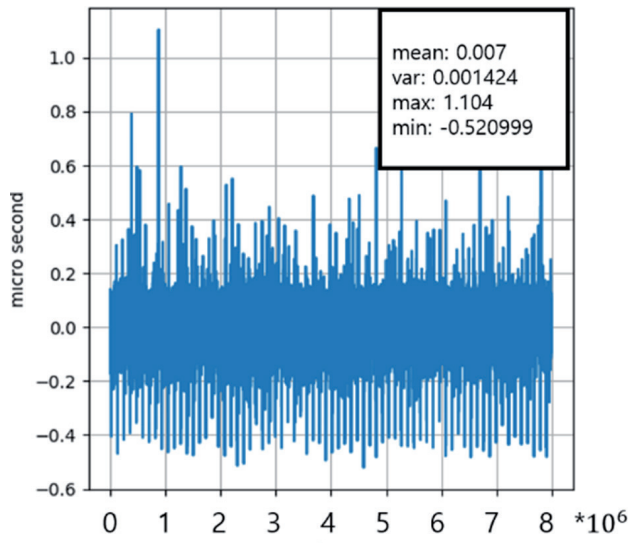


Figure 18.
Jitters in test case 9.

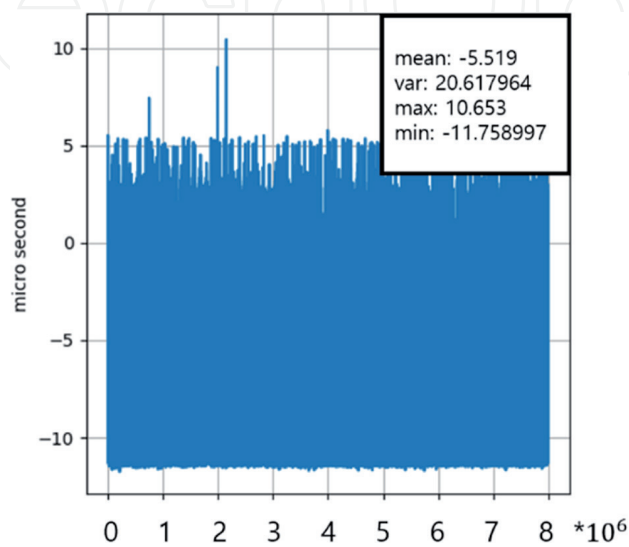


Figure 19.
Jitters in test case 10.

scheduler can work effectively in various situations. Moreover, it can be observed from the test results that the process-type periodic module has a greater effect on jitters than the thread-type and the sporadic modules. Hence, it is better to use the thread-type than the process-type for periodic modules. If the process-type modules as legacy modules are utilized, it is necessary to reduce their number.

4. Conclusion

This study proposed a real-time scheduling algorithm for middleware of industrial automation devices or controllers such as industrial robots and PLCs. The proposed algorithm strictly maintains the periods and supports both periodic and sporadic executions with event handling. It has managed modules, namely, the thread type (or .so type) and process type (or .exe type), for periodic execution, sporadic execution, and non-real-time execution. This study provided the program structures of the thread-type and process-type modules for periodic and sporadic services to manage them efficiently. For sporadic services, the scheduler checks for the occurrence of events using the condition() method in the sporadic modules before invoking the corresponding module.

The proposed scheduling algorithm was implemented using the Xenomai real-time OS and Linux, and it was validated through some test cases. The worst-case jitters measured in the thread-type periodic module and the process-type periodic module were 2.933 and 12.438 μ s, respectively, where the jitter rates were 2.933 and 12.438% with respect to the basic period of 100 μ s. The basic period was maintained very satisfactorily without missing any periods in all the test cases. The test results showed that the proposed scheduler could work well in various situations. Furthermore, it is better to use the thread-type module than the process-type module when periodic modules are used. It was demonstrated that the proposed scheduling algorithm could be used for the middleware of industrial automation devices or controllers.

In future research, the proposed scheduling algorithm will be tested to handle periodic modules, sporadic events, and non-real-time modules in multicore systems, manage process-type periodic modules with smaller worst-case jitters, and support various types of OSs.

Acknowledgements

This work was partly supported by Korea Evaluation Institute of Industrial Technology (KEIT) grant funded by the Korea government (MOTIE) (No. 10067414, development of real-time-assisting SW platform for industrial robot).

IntechOpen


IntechOpen

Author details

Hong Seong Park
Department of Electrical and Electronic Engineering, Kangwon National
University, South Korea

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

IntechOpen

© 2019 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/3.0>), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. 

References

- [1] Rueßmann M, Lorenz M, Gerbert P, Waldner M, Justus J, Engel P, Harnisch M. Industry 4.0: The Future of Productivity and Growth in Manufacturing Industries. Boston Consulting Group; 2015. Available from: https://www.bcg.com/publications/2015/engineered_products_project_business_industry_4_future_productivity_growth_manufacturing_industries.aspx
- [2] Thoben K, Wiesner S, Wuest T. Industrie 4.0 and smart manufacturing—A review of research issues and application examples. *International Journal of Automation Technology*. 2017;**11**:4-16. DOI: 10.20965/ijat.2017.p0004
- [3] MacDougall W. Industrie 4.0 smart manufacturing for the future. Available from: <https://www.manufacturing-policy.eng.cam.ac.uk/documents-folder/policies/germany-industrie-4-0-smart-manufacturing-for-the-future-gtai/view>
- [4] IEC. IEC 61131-3 Programmable controllers—Part 3: Programming languages; 2013
- [5] PLCopen Technical Committee 2. Function Blocks for Motion Control: Part 3—User Guidelines. 2013. Available from: https://www.plcopen.org/system/files/downloads/plcopen_motion_control_part_3_version_2.0.pdf
- [6] CODESYS Group. WHY CODESYS? [Internet]. Available from: <https://www.codesys.com/the-system/why-codesys.html>
- [7] Korobiichuk I, Dobrzahnsky O, Kachniarz M. Remote control of nonlinear motion for mechatronic machine by means of CoDeSys compatible industrial controller. *Tehnički Vjesnik*. 2017;**24**:1661-1667. DOI: 10.17559/TV-20151110164217
- [8] Beckhoff. TWINCAT-PLC and motion control on the PC [Internet]. Available from: <https://www.beckhoff.com/twincat/>
- [9] OSRF Site [Online]. Available from: www.ros.org
- [10] Wei H, Shao Z, Huang Z, Chend R, Guanb Y, Tanc J, et al. RT-ROS: A real-time ROS architecture on multi-core processors. *Future Generation Computer Systems*. 2016;**56**:171-178. DOI: 10.1016/j.future.2015.05.008
- [11] Han S, Kim M, Park HS. Open software platform for robotic services. *IEEE Transactions on Automation Science and Engineering*. 2012;**9**: 467-481. DOI: 10.1109/TASE.2012.2193568
- [12] OPRoS Site [Online]. Available from: www.ropros.org
- [13] OpenRTM Site [Online]. Available from: www.openrtm.org
- [14] Hasegawa R, Yawata N, Ando N, Nishio N, Azumi T. Embedded component-based framework for robot technology middleware. *Journal of Information Processing*. 2017;**25**: 811-819. DOI: 10.2197/ipsjip.25.811
- [15] OROCOS site [Online]. Available from: www.orocos.org
- [16] Rastogi N, Dutta P, Krishna V, Gotewa KK. Implementation of an OROCOS based real-time equipment controller for remote maintenance of tokamaks. In: *Proceedings of the Advances in Robotics*; June 28-02 July 2017; New Delhi, India; DOI: 10.1145/3132446.3134900
- [17] Muratore L, Laurenzi A, Hoffman EM, Rocchi A, Caldwell DG, Tsagarakis NG. XBotCore: A real-time cross-robot software platform. In: *2017 First IEEE*

International Conference on Robotic Computing (IRC); 10-12 April 2017; Taichung, Taiwan; DOI: 10.1109/IRC.2017.45

[18] YARP site [Online]. Available from: www.yarp.it

[19] Paikan A, Pattacini U, Domenichelli D. A best-effort approach for run-time channel prioritization in real-time robotic application. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS); 28 Sept.-2 Oct. 2015; Hamburg, Germany. 2015. pp. 1799-1805. DOI: 10.1109/IROS.2015.7353611

[20] Levine DL, Schmidt DC, Flores-Gaitan S. CORBA measuring OS support for real-time CORBA ORBs. In: 1999 Proceedings Fourth International Workshop on Object-Oriented Real-Time Dependable Systems; 27-29 Jan. 1999; Santa Barbara, CA, USA. 1999. pp. 9-17. DOI: 10.1109/WORDS.1999.806555

[21] OPC Foundation. PLCopen and OPC Foundation: OPC UA Information Model for IEC 61131-3. 2010

[22] Yu D, Park HS. Real-time middleware with periodic service for industrial robot. In: 2017 14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI); 28 June-1 July 2017; Jeju, South Korea. 2017. pp. 879-881. DOI: 10.1109/URAI.2017.7992853

[23] Xenomai site [online]. Available from: xenomai.org