electronics

MDPI

Article

Open Embedded Real-time Controllers for Industrial Distributed Control Systems

Raimarius Delgado, Jaeho Park and Byoung Wook Choi *

Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul 01811, Korea; raim223@seoultech.ac.kr (R.D.); jaeho@seoultech.ac.kr (J.P.)

* Correspondence: bwchoi@seoultech.ac.kr; Tel.: +82-02-970-6412

Received: 5 December 2018; Accepted: 14 February 2019; Published: 17 February 2019

Abstract: This paper presents design details adopting open embedded systems (OES) as real-time controllers in industrial distributed control systems. OES minimize development cost and enhance portability while addressing widely known shortcomings of their proprietary counterparts. These shortcomings include the black box method of distribution which hinders integration to more complex systems. However, OES are highly dependent on the compatibility of each software components and essential benchmarking is required to ensure that the system can satisfy hard real-time constraints. To address these issues and the notion that OES will find broader distributed control applications, we provide detailed procedures in realizing OES based on an open source real-time operating system on various low-cost open embedded platforms. Their performance was evaluated and compared in terms of periodicity and schedulability, task synchronization, and interrupt response time, which are crucial metrics to determine stability and reliability of real-time controllers. Practical implementations, including the modernization of a multi-axis industrial robot controller, are described clearly to serve as a comprehensive reference on the integration of OES in industrial distributed control systems.

Keywords: open embedded systems; real-time controller; real-time operating systems; Xenomai; industrial distributed control systems

1. Introduction

Distributed control systems (DCS) are widely found in numerous control applications requiring operation of various hardware devices and complex control algorithms. This is particularly relevant to industrial control systems that are required to control distributed devices behaving reactively with the working environment. The most essential part of a DCS is the main controller whose functionality is to calculate and transfer motion commands to the actuators and to collect feedback data from sensory devices. The main controller is responsible in ensuring minimal computational delay when processing significant amount of data from the distributed devices. To cope with this demand, it should achieve precise control period and demonstrate deterministic response times. In other words, the entire system should satisfy real-time constraints [1]. Considering the objective mentioned above and evaluating previous studies in literature, the general approach is to design the main controller based on a real-time operating system (RTOS). An RTOS usually have all the necessary features as good building blocks to build real-time systems. It allows priority-based scheduling of multiple tasks. Most RTOS are commercially distributed that runs on large and often expensive hardware platforms that support monolithic control applications [2]. Therefore, the software, together with the entire system, are usually distributed in a black box that prevents any changes and hinders integration to more complex systems [3,4].

These shortcomings are addressed by open source projects by enabling developers to freely add and modify the software. With more users simultaneously sharing their contributions, the evolution of the system is enhanced and debugging of problems can be easier. Due to its open source nature and easy accessibility, Linux has gained increasing popularity as the general-purpose operating system for embedded systems applied in scientific experiments that do not require strict temporal determinism [5]. However, its scheduling policy, which utilizes fairness [6] rather than priorities of tasks, prevents its integration to real-time control systems. Various real-time Linux approaches were developed in the recent past with the aim to improve the priority scheduling of the standard Linux to operate in real-time [7–10]. The two major approaches of real-time Linux are classified as the preemptible kernel, and the dual-kernel approaches. In the former, the source code of the standard Linux kernel is directly modified. The scheduler and the timers are enhanced to enable priority-based scheduling so that higher priority tasks can preempt lower priority ones. This means that the preemptible kernel is dependent on the kernel version and sometimes it might be a proprietary software distributed by commercial purpose. On the other hand, the dual-kernel approach employs a real-time kernel to function alongside the standard Linux through an abstraction layer called the adaptive domain environment for operating systems (ADEOS) [11]. There are two projects employing the dual-kernel approach, namely RTAI and Xenomai. Ceria et al. presented a comparison of the real-time performance between the RT_PREEMPT (preemptible kernel) and RTAI (dual-kernel) as the main controller of an EtherCAT network [12]. However, the RTAI project has become stagnant and has limited central processing unit (CPU) support (Intel x86, PowerPC, and older ARM processors). Xenomai, on the other hand, has a very active community supporting various embedded hardware platforms and leading to wide-range of control applications [10,13,14]. The performance of Xenomai compared with the RT_PREEMPT is presented by Brown et al. [15]. Comparing their performance, the RT_PREEMPT is significantly better in terms of limiting the maximum scheduling jitter. Whereas both dual-kernel approach showed better accuracy in meeting hard real-time deadlines with lower standard deviations.

We focus on designing real-time controllers based on Xenomai because of its availability to a large number of platforms and it provides user-friendly application programming interfaces (API). The performance was also proven as a worthy alternative to a commercial RTOS [5]. As OES are composed of both hardware and software components, we have selected popular low-cost embedded hardware platforms to implement the real-time environment. Open embedded hardware platforms are gaining popularity as main controllers in different control systems including sensor networks [16,17], image processors [18], and industrial robot controllers [19,20] due to their portability, low power requirements, and inexpensive costs in comparison to high-end computers. However, developing the real-time environment is more difficult owing to the limited availability of systematic documentations and technical support. These hardware platforms are available off-the-shelf and comes with Linux kernel sources provided by their manufacturers. Depending on the control application, compatibility with other software components is an open problem that is very time consuming and without a standard solution.

This paper aims to provide a comprehensive reference for readers on the design and integration of OES as industrial controllers of distributed control systems. To begin, the paper provides the complete procedures, considering software compatibility, in designing Xenomai-based real-time controllers on different embedded platforms namely, BeagleBone Black, Raspberry Pi 3, Zybo-7020, and i.MX6Q SABRELite. We also discuss their differences in terms of the control application. The performance of each OES was evaluated focusing on three metrics including: periodicity and responsiveness, performance of the task synchronization mechanisms, and the interrupt response times. The experiments were conducted to evaluate whether the system ensures deterministic behavior and responsiveness in various conditions, which is critical to determine stability and reliability of the entire system. Practical cases of that the authors have implemented previously are discussed to provide readers for implementation examples. This paper is organized as follows: Section 2 presents the design details of the real-time environment for different open embedded platforms. The performance

evaluation and experiment results are discussed in Section 3. Practical example cases are described in Section 4 and the last section summarizes the concluding remarks.

2. Designing Real-time Embedded Controllers

In this section, we discuss the different real-time approaches of Linux and describe their differences in terms of functionality and performance. As OESes are comprised with both hardware and software, we have selected four popular low-cost embedded hardware platforms (namely BeagleBone Black, Raspberry Pi 3, Zybo-7020, and i.MX6Q SABRELite) to implement the real-time environment based on Xenomai, a dual-kernel approach of real-time Linux. We also provide the detailed procedures considering software version compatibility to design real-time controllers utilizing the embedded platforms mentioned above.

2.1. Real-Time Embedded Linux Approaches

Linux is currently considered a soft RTOS owing to the rapid improvements of the kernel and the continuous advancements in the computer power of hardware platforms. However, it is still not suitable for hard real-time applications that require strict timing constraints and preemption of low priority tasks because of its scheduling policy of utilizing fairness over priorities [6]. Several real-time Linux approaches were introduced in the recent past which improves the response time and priority-based scheduling in order to meet hard real-time deadlines. These are largely classified into two major approaches: the preemptible kernel [8] and the dual-kernel approaches [7,9,10]. In the fully preemptible kernel approach, all parts of the standard Linux kernel with relationship to the scheduler and timers are directly modified to render lower priority tasks preemptible by higher priority ones. The most popular distribution of this approach is RT_PREEMPT [4], with its architecture shown in Figure 1a. Herein, the entire Linux kernel should be modified which is very time consuming. The performance is also variable depending on the version of the Linux kernel. Meaning a single change in the kernel or an update of the kernel version can greatly affect the performance of the system and would require the same tedious and time-consuming job. For this reason, fully preemptible kernel is often distributed by commercial vendors.

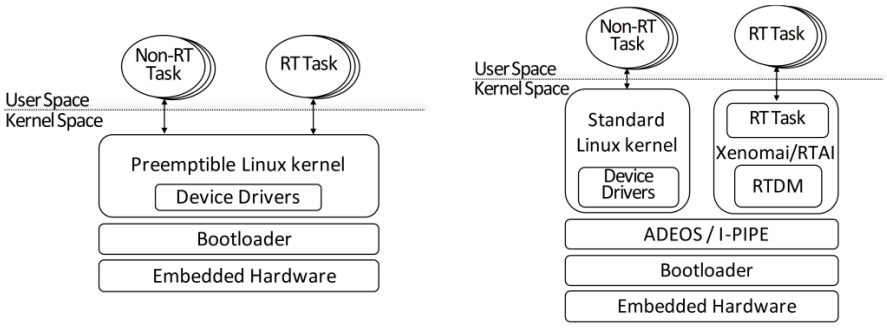

Figure 1. Classification of real-time embedded Linux systems: (**a**) fully preemptible kernel approach; (**b**) dual-kernel approach.

Conversely, the dual-kernel approach employs a real-time kernel to function alongside the standard Linux through a virtual layer called the adaptive domain environment for operating systems [11]. In comparison to the former approach, the real-time kernel can run independent with the standard Linux kernel as long as a compatible ADEOS patch is in existent. The architecture of the dual-kernel approach is shown in Figure 1b. In this approach, the standard Linux has the lowest priority and would only run if there are no tasks available for the real-time kernel. Two dual-kernel approach projects are currently available, namely Xenomai and RTAI [14,21,22]. Both Xenomai and RTAI are more accurate in meeting hard real-time deadlines with lower standard deviations and

offers API for the real-time driver model (RTDM) [23]. RTAI, however, has become inactive and only supports Intel, PowerPC, and older ARM processors. Therefore, we have selected Xenomai as the RTOS for designing OES-based real-time controllers presented in the next section.

2.2. Xenomai-Based Real-Time Environment

With the aim to become a helpful reference for developers with the intent of integrating OES in their own control systems, we provide the complete steps in designing Xenomai-based real-time controllers on four embedded platforms. Table 1 enumerates the real-time environment for each embedded platform considering software compatibility. Although manufacturers provide Linux kernel sources and the bootloader, compatibility with other software and patches is still an open problem. For example, a real-time environment based on Xenomai requires the ADEOS patch compatible with both the Linux kernel and Xenomai itself. The most common solution is look for the best compatible combination by trial and error in accordance to the control application deployed in the embedded platform. For instance, Raspberry Pi 3 is not applicable to EtherCAT application, but the others are possible.

Table 1. Real-time environment and the compatible version of each software components for the respective open embedded platform.

Software Component	BeagleBone Black	Raspberry Pi 3	Zybo-7020	i.MX6Q SABRELite
Toolchain	gcc-linaro-arm-linux-gnueabihf-4.7.3	gcc-linaro-arm-linux-gnueabihf-raspbian-4.8.3	gcc-linaro-arm-linux-gnueabihf-4.8.3	gcc-linaro-arm-linux-gnueabihf-4.8.3
Bootloader	U-Boot 2015.10	Broadcom Bootloader	U-Boot 2015.10	U-Boot 2015.10
ADEOS/I-PIPE	ipipe-3.8.13-arm-3	ipipe-4.1.18-arm-4	ipipe-3.14.17-arm-4	ipipe-3.14.17-arm-2
Xenomai	2.6.5 [24]	3.0.2 [24]	2.6.5	2.6.5
Linux Kernel	3.8.13 [25]	4.1.21 [26]	3.14.2 [27]	3.14.15 [28]
Root Filesystem	Minimal Ubuntu 14.04	Raspbian Jessie	Minimal Ubuntu 14.04	Minimal Ubuntu 14.04
IgH EtherCAT	1.5.2	-	1.5.2	1.5.2

Instead of directly building the Linux kernel, first, it should be patched with the compatible ADEOS version, available at the Xenomai i-pipe patch archives [29]. The patched kernel is configured disabling CPU features which causes unwanted voltage and frequency changes that greatly affects the real-time performance of Xenomai. These include CONFIG_CPU_FREQ, CONFIG_CPU_IDLE, and CONFIG_KGDB. Buffer overflow detection and protection is also disabled because it can trigger warnings when installing Xenomai. The kernel is compiled including device tree binaries (DTB). DTB is the newest data structure in Linux that contains the information of all the devices that are attached to the respective embedded hardware platform. DTBs are introduced from Linux kernel version 3.8. Therefore, the bootloader should be able to identify this new data structure to ensure successful booting process. In case of the Raspberry Pi 3, the bootloader is not open to the public. We decided to build the kernel on top of a complete working image available in the Raspberry Pi repository [30].

For the other platforms, we have chosen the same bootloader and version, U-Boot 2015.10 [31], which is the most stable compatible version with the other software components. The Raspberry Pi 3 is integrated to a DCS that does not require any additional software, thus the latest version of both the Linux kernel and Xenomai were implemented. Whereas the other platforms are designed with the purpose of being the main controller of an EtherCAT network therefore, compatibility with an open source EtherCAT master was also considered [32]. The practical applications of each hardware platform are thoroughly discussed in Section 4.

Xenomai is included with user space libraries and tools for easier application development without having to program in the kernel space. For all the embedded platforms, Xenomai was compiled enabling the following flags: -march = armv7-a and -mfpu = vfp3, which stands for the CPU and floating-point unit (FPU) architectures. To satisfy software dependencies and eliminate problems such as unusable binaries, missing libraries, and other build errors, we have selected toolchain versions compatible with all the software components and their respective hardware platform. For the

users to easily communicate with the kernel, we implemented the root filesystem, minimal Ubuntu 14.04 [33]. With the same reason as of the bootloader, the root filesystem image on the Raspberry Pi 3 is Raspbian Jessie.

A complete step-by-step guide for the compilation of i.MX6Q SABRELite is presented in our previous study [34]. The same method can be applied with the other embedded platforms only following the versions stated in this work. Building the environment is only the first step in designing real-time controllers based on OESes. Reliability of these platforms requires evaluation of their real-time performance; whether they can satisfy hard temporal constraints in various conditions. The next section offers intuitive experiment procedures to demonstrate the validity of OES as real-time controllers.

3. Performance Evaluation

This section describes the performance evaluation of the real-time controllers designed in the previous section. We focus on three performance metrics that are critical to determine reliability and stability of each OES in real-time applications for distributed control. First, the periodicity and responsiveness of each task was evaluated to ascertain whether the system shows deterministic behavior and can perform expected tasks while satisfying hard temporal deadlines. Next, various task synchronization mechanisms were defined and evaluated to give a guideline of the expected overheads produced when they are applied in user applications. Finally, experiments were conducted to measure the interrupt response time for each OES. This determines the behavior of the system when interacting with device drivers for digital input and output devices. The experiment procedures and conditions for each metric is discussed in detail in the following subsections.

3.1. Periodicity and Responsiveness

Schedulability of real-time tasks is highly dependable on the timing correctness of each task; whether all the tasks can execute within their respective deadline. The periodicity and responsiveness of the system is verified using the method called response-time analysis [35]. According to this method, schedulability of set of tasks can be analyzed according to worst case response time. The response time is defined as the duration in which a task starts its execution from a release point until it finishes its job. The behavior of the execution of a real-time task is illustrated in Figure 2.

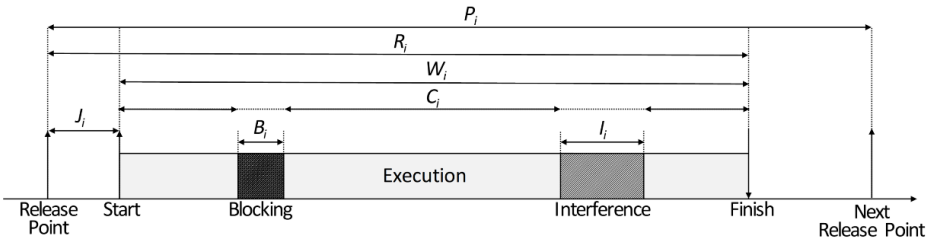

Figure 2. Timeline describing the behavior of a real-time task with a given response time and period.

In the figure, for task τ_i, timing characteristics are defined with their respective priorities, activation period (P), which is usually equal to the relative deadline where execution time is defined as computation time (C). The release jitter (J) which is the delay of execution at the beginning of the task due to context switching. The busy period (W) of the task which is the sum of computation time (C), the blocking time (B), and interference time (I). A blocking happens when a low priority task owns a resource needed by a high priority task. Whereas, an interference occurs when lower priority tasks are preempted by tasks with higher priorities [8].

$$W_i = C_i + B_i + \underbrace{\sum_{j \in hp(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil \cdot C_j}_{I_i} \tag{1}$$

In this equation, I_i denotes the sum of the computation time of all the elements included in the set of tasks with higher priority than task τ_i, $hp(i)$. If $\exists\tau\in hp(i)$, Equation (1) is iterated x number of times until $W_i^{(x+1)} = W_i^x$. The test should be stopped once the current iteration yields a value of beyond the deadline else, it would be impossible to terminate. This is only applicable to determine the busy period of the current task, τ_i, and should be repeated when needed for the other tasks. After the busy period is calculated, the overall response time of the system is determined by the following equation [8]:

$$R_i = W_i + J_i \tag{2}$$

Periodic tasks are schedulable if and only if all the scheduled tasks can complete their execution of the given computation time within the respective period/deadline. To demonstrate a practical example of the response-time test, we performed a simplified experiment consisting of two real-time tasks. The goal of the experiment is to verify whether the real-time controllers designed in the previous section can show deterministic behavior accordingly with the response time test. The experiment conditions are made as simple as possible for easier understanding and lesser calculations. Two tasks were generated with the given priority, period, and computation time. The first task, τ_1, has a priority of 99, computation time of 0.5 ms, and period of 1 ms. The lower priority task, τ_2, was generated with the priority of 80, computation time of 1.5 ms, and period of 5 ms. Note that according to the Xenomai documentation, the highest priority level is 99 and the lowest is 1. The tasks are scheduled to start approximately at the same time. To ensure that both tasks can fulfill the configured computation time, we have used the function *rt_timer_spin*(SPINTIME) which is available in the Xenomai user space library. This function burns the CPU in the specified SPINTIME, given as an argument in nanoseconds. The expected behavior of the tasks is shown in Figure 3. In this figure, τ_1 and τ_2 are represented by the blue and red lines, respectively. The interference is represented by the box with blue diagonal lines.

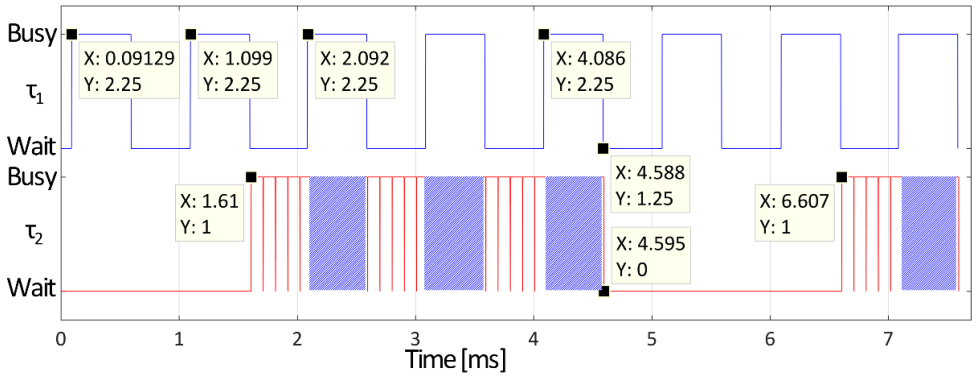

Figure 3. Execution timeline of the real-time tasks for the experiment with two tasks. The blue solid line represents the higher priority task, τ_1, and the red solid line represents τ_2.

It is important to notice that τ_2 is preempted by the higher priority task, thus there are points of interference during its busy period. For visibility purposes, the *rt_timer_spin*() function is encapsulated in a loop with the SPINTIME of 0.1 ms. The loop will terminate when the accumulated SPINTIME is equal to the configured computation time of 1.5 ms. This means that τ_2 completes its job when fifteen blocks of 0.1 ms is executed. The interference time is represented by the boxes with blue diagonal lines. Additionally, it is conspicuous that both tasks run periodically as shown in the values of the data cursor located at the release points. According to Equations (1) and (2), the expected response time for τ_1 is 0.5 ms because it has the highest priority and no block or interference occurs during its busy period. On the other hand, τ_2 will complete its execution every 3 ms. The calculation for the response time is presented in the Appendix A. The calculated response times are also verified in accordance to the schedulability analysis tool, Model Analysis Suite for Real-time Applications (MAST) [36]. Herein, the default toolset was selected to calculate the worst-case behavior of the system and whether it

can always meet the hard real-time deadline. We have enabled *slack* calculations, or the percentage by which the execution time can be increased while maintaining schedulability. The slack is a vital information in determining whether how close the system is becoming schedulable or non-schedulable. The results of the MAST analysis are shown in Table 2. The worst-case response time for each task is equal to the results of the calculation using Equations (1) and (2) and both tasks are schedulable with 24.61% system slack.

Table 2. Calculation of the worst-case response time using MAST [36].

Task	Priority	Worst-Case Response (ms)	Slack (%)
τ_1	99	0.500000	39.45%
τ_2	80	3.000000	65.63%

The experiments were performed on each real-time embedded controller for 10 minutes to verify whether they can show schedulability in comparison to the values above. During the experiment, the OESes are kept isolated to avoid any unwanted interruptions that could affect the performance of the entire system. For this reason, all the measured values are stored in a buffer for offline processing and analysis.

The results of the timing analysis are shown in Table 3 with the statistical average (avg), maximum (max), minimum (min), and standard deviation (σ) values of each timing metric. Analyzing these results, we could see that the measured average response time for all OESes were approximately equal to the expected response time of 0.5 and 3 ms for τ_1 and τ_2, respectively. Moreover, we can conclude that they were able to meet their respective deadlines producing promising results of the average period for both tasks. The σ shows that the Raspberry Pi 3 has the best performance with the lowest deviation to the statistical average. Although not reported in this paper, this difference can be accountable to the improved architecture of Xenomai 3.

Table 3. Statistical analysis of periodicity and response times of each embedded hardware platform.

Task	τ_1, High Priority (99, 1 ms Deadline)			τ_2, Low Priority (80, 5 ms Deadline)		
Metric (ms)	**Period (P)**	**Response (R)**	**Jitter (J)**	**Period (P)**	**Response (R)**	**Jitter (J)**
BeagleBone Black						
avg.	1.000000	0.502764	0.004424	5.000000	2.987725	0.001938
max.	1.029958	0.510792	0.029958	5.028000	3.001875	0.028000
min.	0.972792	0.502125	0.000000	4.976250	2.963000	0.000000
st.d (σ)	0.006238	0.000667	0.004398	0.003521	0.004687	0.002940
Raspberry Pi 3						
avg.	1.000000	0.500780	0.001121	5.000000	2.994471	0.000617
max.	1.007084	0.503437	0.008906	5.008959	2.998542	0.008959
min.	0.991094	0.500520	0.000000	4.993646	2.989844	0.000000
st.d (σ)	0.001429	0.000164	0.001124	0.000951	0.000666	0.000723
Zybo-7020						
avg.	1.000000	0.502655	0.000878	5.000000	2.998630	0.001120
max.	1.010889	0.509745	0.010909	5.015211	3.003186	0.015323
min.	0.989091	0.502341	0.000000	4.984677	2.984868	0.000001
st.d (σ)	0.001588	0.000332	0.001127	0.002187	0.002377	0.001878
i.MX6Q SABRELite						
avg.	1.000000	0.502450	0.001423	5.000000	2.996607	0.003339
max.	1.010803	0.508783	0.012045	5.015409	3.002187	0.015409
min.	0.987955	0.501965	0.000000	4.987518	2.984358	0.000000
st.d (σ)	0.002163	0.000913	0.001629	0.004488	0.003698	0.002999

As the objective of this paper is to prove the viability of OES in real-time applications, the difference in performance with respect to the Xenomai version is neglected as long as the OES were able to fulfill the requirements of a hard real-time control system. These results indicate that the designed real-time controllers based on OES is feasible for hard real-time applications.

3.2. Task Synchronization Mechanisms

Aside from periodicity and responsiveness of real-time tasks, correctness of the data being processed is another concern to ensure deterministic behavior of a real-time control system. Notably in a DCS, various devices are required to exchange data in a multitasking environment. These tasks are expected to execute in parallel and often need to access the same resources. However, asynchronization and concurrency issues between them can cause either data overflow (when the publisher is running faster than the reader), or data loss (when the publisher is slower). RTOSes offer inter-task communication (ITC) mechanisms to prevent such anomaly. ITCs are characterized into two main types: Shared memory protection and message-passing mechanisms. In a shared memory, different tasks can publish or read data stored in a region of memory. Mechanisms such as semaphores and mutexes prevent simultaneous access of that region, thus only one task can access the shared data and avoid the asynchronization issues mentioned above. In case of the message-passing, one tasks acts as the sender responsible for transmitting a specific data to the reader.

The reader continuously waits for the message from the sender and will not execute until it receives the entire message completely. In this paper, ITC mechanisms are evaluated to serve as a guideline and make developers aware of the amount of overhead when applying these mechanisms to user space applications. This is very helpful in particular to real-time applications in an embedded environment, where optimal size of the user code is required to save memory space and to efficiently predict the total task execution time.

3.2.1. Semaphore and Mutex

Semaphores are very useful in the synchronization of multiple tasks when communicating with shared data structures. As all tasks in the same process exist in the same address space, sharing data structures between tasks are vulnerable to data corruption. A semaphore gives exclusive access to the shared resources unto the task that possesses it. Other tasks requesting for the semaphore are suspended until the current owner releases it. In Xenomai, semaphores are counting semaphores that can allow N number of tasks to access the shared resources simultaneously. On the other hand, mutexes (MUTual Exclusion) are binary semaphores that can only have two values: unlocked or locked. In the locked state, the task in possession can access the shared resources and the other tasks should wait. Whereas in the unlocked state, the critical section is free for the other tasks to access and to acquire the mutex. Another feature of the mutex in Xenomai is that it enforces a priority inheritance protocol in order to solve priority inversions, a problem in scheduling real-time tasks where lower priority tasks preempts higher priority tasks. To measure the overhead produced by these mechanisms, the experiment condition in the previous section were reformulated including semaphores and mutexes as shown in the pseudo code in Figure 4.

Basically, the response time with either semaphore or mutex ($R_{Sem\,|\,Mtx}$) is equal to the response time in without any of the ITC mechanisms and the overhead. Therefore, the time duration for acquiring and releasing the semaphore or mutex is calculated using the following equation:

$$R_{Sem|Mtx} = R_0 + T_{Sem|Mtx} \tag{3}$$

where, R_0 denotes the response time measured in the previous section, $T_{Sem\,|\,Mtx}$ denotes the overhead for the mutex or the semaphore.

```
set task period(1 ms) and make periodic();
create semaphore/mutex;
set SPINTIME to 0.5 ms;
read end_time;
while (1)
    wait release time;
    read start_time;
    acquire semaphore/mutex;
    rt_timer_spin(SPINTIME);
    release semaphore/mutex;
    read execution_time;
    period = start_time - end_time;
    R_Sem|Mtx = execution_time - start_time;
    end_time = start_time;
endwhile;
```

(**a**)

```
set task period(5 ms) and make periodic();
set EXECTIME to 1.5 ms;
set SPINTIME to 0.1 ms;
read end_time;
while (1)
    wait release time;
    while(runtime < EXECTIME)
        read start_time;
        acquire semaphore/mutex;
        rt_timer_spin(SPINTIME);
        runtime += SPINTIME;
        release semaphore/mutex;
        read execution_time;
    endwhile;
    period = start_time - end_time;
    R_Sem|Mtx = execution_time - start_time;
    end_time = start_time;
endwhile;
```

(**b**)

Figure 4. Pseudo code for measuring the response time of real-time tasks with mutex and semaphore: (**a**) high priority task, τ_1; (**b**) low priority task, τ_2. Note that the same pseudo code can be applied in the periodicity and response time test (previous section) but omitting the mutex/semaphore operations.

Considering the same conditions, the experiments were conducted for each OES. Although the Xenomai semaphore is a counting semaphore, we decided to configure it behaving similar with a mutex that only has two values. This is to make a more legit comparison of both mechanisms. Moreover, we focus on acquiring the results from the higher priority task with straightforward implementation in order to neglect external factors that can contribute to the measurement. The semaphore/mutex operations in the low priority task is within a loop, which can produce unwanted computational delays. The results are summarized in Table 4 showing the statistical average of the response times and the time duration of each ITC mechanism. Herein, we can clearly see that the mutex has larger overhead than the semaphore, which is consistent for all the embedded platforms. We assume that this is due to the mutex having more features such as blocking interrupts and the priority inheritance scheme.

Table 4. Statistical average of the shared memory ITC mechanisms, semaphore and mutex.

Mechanism	**Semaphore**		**Mutex**	
avg. (µs)	R_{Sem}	T_{Sem}	R_{Mtx}	T_{Mtx}
BeagleBone Black	531.52	28.756	532.090	29.326
Raspberry Pi 3	527.443	26.663	531.018	30.238
Zybo-7020	533.255	30.6	534.376	31.721
i.MX6Q SABRELite	537.04	34.59	542.864	40.414

3.2.2. Message Queue

The message queue is very useful in sending data between real-time tasks. The message is sent from either interrupt service routines or tasks to another task. Centralization of a specific function, such as error handling, is the common application of message queues. If a task is waiting for a message and the queue is empty, then the task will be suspended until a message is posted in the queue. Meaning, the waiting task does not consume any CPU time while waiting for a message thus, other tasks can run continuously. The goal of the experiments is to measure the total time duration for the receiver task to be activated, which is shown in the pseudo code in Figure 5.

```
set task period(1 ms) and make periodic();
create message queue();
read end_time;
while (1)
        wait release time;
        read start_time;
        msg = "dummy";
        read T_Msgq_start;
        send message_queue(msg);
        period = start_time - end_time;
        end_time = start_time;
endwhile;
```

(**a**)

```
read end_time;
while (1)
        recieve message_queue(msg);
        read T_Msgq_end
        period = T_Msgq_end - end_time;
        T_Msgq = T_Msgq_end - T_Msgq_start;
        end_time = T_Msgq_end;
endwhile;
```

(**b**)

Figure 5. Pseudo code for measuring the elapse time of a message queue between two tasks: (**a**) high priority task (sender); (**b**) low priority task (receiver).

The total time duration of from the sender task posting a message to the queue, until the receiver task receives the message and gets activated, denoted as T_{Msgq}, is calculated using the following equation:

$$T_{Msgq} + T_{ctx} = T_{Msgq\_end} - T_{Msgq\_start} \tag{4}$$

In here, T_{ctx} denotes the context switching time, or the time it takes for the CPU to save the context (state) of the current task, restore and execute the context of the next scheduled task. In this paper we assume that the context switching time is very small that it is neglectable. For a practical implementation of message queues, we consider the pseudo code in Figure 5. The high priority task is set periodically for 1 ms. Note that, the receiver task depends on the periodicity of the sender task. Meaning, the period of the receiver should be equal to the period of the task posting the message. The sender task posts a dummy message to the queue and the receiver task waits for the message before doing its execution. The results are summarized in Table 5 with the statistical average (avg), maximum (max), minimum (min), and standard deviation (σ) values of T_{Msgq} and the periodicity of the two tasks.

Table 5. Statistical results of the message queue mechanism and the periodicity of two tasks.

Mechanism	**Message Queue**		
Metric	**P,τ_1 (ms)**	**P,τ_2 (ms)**	**T_{Msgq} (µs)**
	BeagleBone Black		
avg.	1.000000	1.000000	18.748
max.	1.036625	1.059042	41.459
min.	0.976375	0.961084	14.375
st.d (σ)	0.002621	0.004464	3.805
	Raspberry Pi 3		
avg.	1.000000	1.000000	15.341
max.	1.005989	1.015000	24.531
min.	0.994532	0.986979	14.740
st.d (σ)	0.000522	0.000609	0.224
	Zybo-7020		
avg.	1.000000	1.000000	14.627
max.	1.012644	1.029186	31.596
min.	0.985410	0.968832	11.454
st.d (σ)	0.000939	0.002162	0.789
	i.MX6Q SABRELite		
avg.	1.000000	1.000000	14.376
max.	1.009421	1.013699	24.722
min.	0.991685	0.985530	9.898
st.d (σ)	0.001676	0.002172	0.498

As expected, the periodicity (P) of the receiver task (τ_2) highly depends on that of the sender task (τ_1). This is evident in all the measured data from each OES. The trend of the periodicity for both tasks is also consistent with the results from Section 3.1 with the Raspberry Pi 3 showing the best performance and BeagleBone Black has the relatively worst results. Moreover, most of the OES produced similar statistical average of T_{Msgq}, BeagleBone Black has the worst average with 18.748 µs. The same trend is visible in the standard deviation where the BeagleBone Black produced very high value of 3.805 µs in comparison to the other OES that show deviations of less than 1 µs. We account this to the single core architecture of the BeagleBone Black whereas, all other OES systems are attached with multiple CPU cores.

3.3. Interrupt Response Time

As DCS are composed of different devices to interact with the environment, it is very important to measure the interrupt response time of the main controller. The interrupt response time is defined as the elapse time between an interrupt signal and the corresponding interrupt service routine. In a Xenomai environment, device drivers should be created in order to interact with the connected devices. However, most device drivers are only available to the standard Linux. Although it is possible to use these device drivers inside Xenomai tasks, it is not highly recommended because the event called mode switching could occur. Mode switching causes Xenomai tasks to be scheduled in the standard Linux scheduler, thus losing its real-time capabilities. To this end, Xenomai offers a RTDM to develop device drivers without suffering from the issues of mode switching. Using RTDM, we can expect that the interrupt response time would be lower than that of the standard Linux because of priority-based scheduling of Xenomai.

Comparative experiment measuring the interrupt response time was conducted by creating RTDM device drivers and standard Linux device driver that handle two general-purpose input and output (GPIO) ports. To gather accurate results, we used a function generator to generate square-wave signals that will be connected to the input port of the OES. An oscilloscope was used for data acquisition and determine the skew between the reference signal and the device driver output. The first port of the GPIO is configured as the input, connected to a square-wave function generator. The other port is configured as the digital output. The interrupt service routine is kept as simple as possible by acquiring the value of the input port and sending it directly to the output port. The input port is probed by the oscilloscope which becomes the reference signal. Another probe is placed on the output port. The time difference (skew) between the ports is the interrupt response time. The same procedures were implemented using the RTDM and standard Linux device drivers. The experiments were conducted for 10 minutes and the statistical measurements were acquired from the oscilloscope. The actual results for all the OES are shown in Figure 6. In the figure, the standard Linux device driver has shown interrupt response time that is four times at most than that of the RTDM. The average interrupt response time for RTDM ranges from 5.22 µs to 8.01 µs, with the Zybo-7020 showing the fastest response. The same experiments were repeated a few times producing results with the same trend. These promising results will serve as a good measure for developers willing to integrate various devices to Xenomai-based real-time controllers. Especially, with devices that requires fast response times, RTDM device drivers can minimize the interrupt response time, guaranteeing priority-based scheduling.

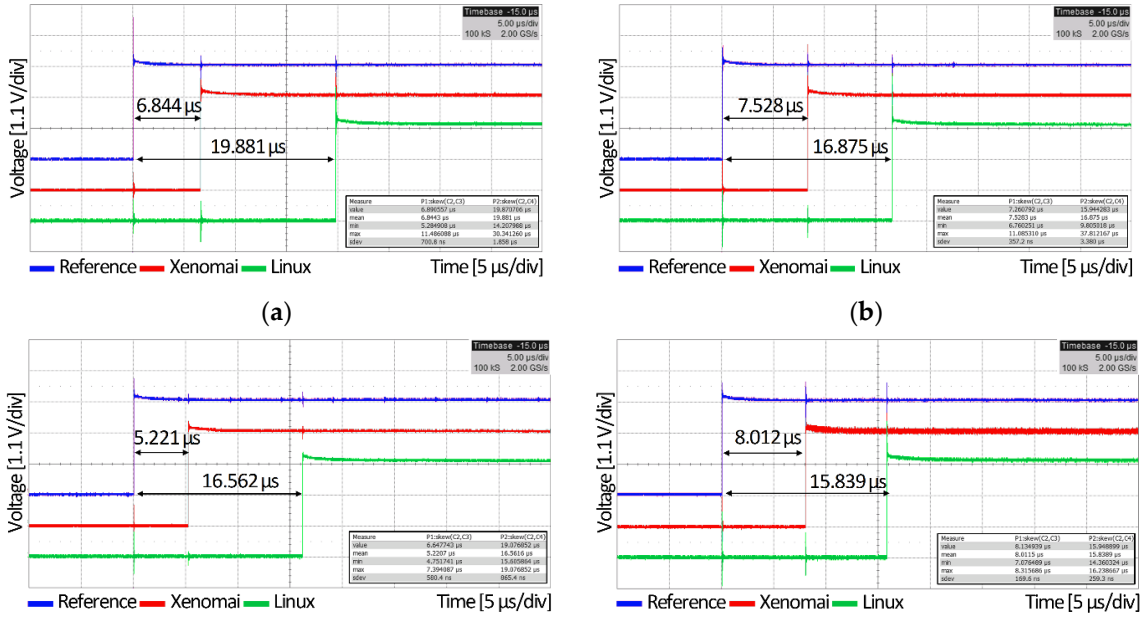

Figure 6. Comparison of interrupt response times of Xenomai and the standard Linux on different OES: (**a**) BeagleBone Black; (**b**) Raspberry Pi 3 (**c**) Zybo-7020; (**d**) i.MX6Q SABRELite.

Let us again consider the two-task application described in Figure 4, the output GPIO is toggled to visualize the jobs. Figure 7 shows the actual behavior of the tasks observed using an oscilloscope for all the designed real-time controllers. The high priority task (τ_1) runs periodically with an average of 1 ms for all the embedded controllers. The lower priority task (τ_2) at the bottom part of the plot also maintains periodicity running with an average of 5 ms. This shows that the minimal interrupt response time produced by the RTDM driver does not affect the periodicity of the real-time tasks. As expected, Raspberry Pi 3 shows the best performance with standard deviation of 1.938 µs and 443.6 ns for τ_1 and τ_2, respectively.

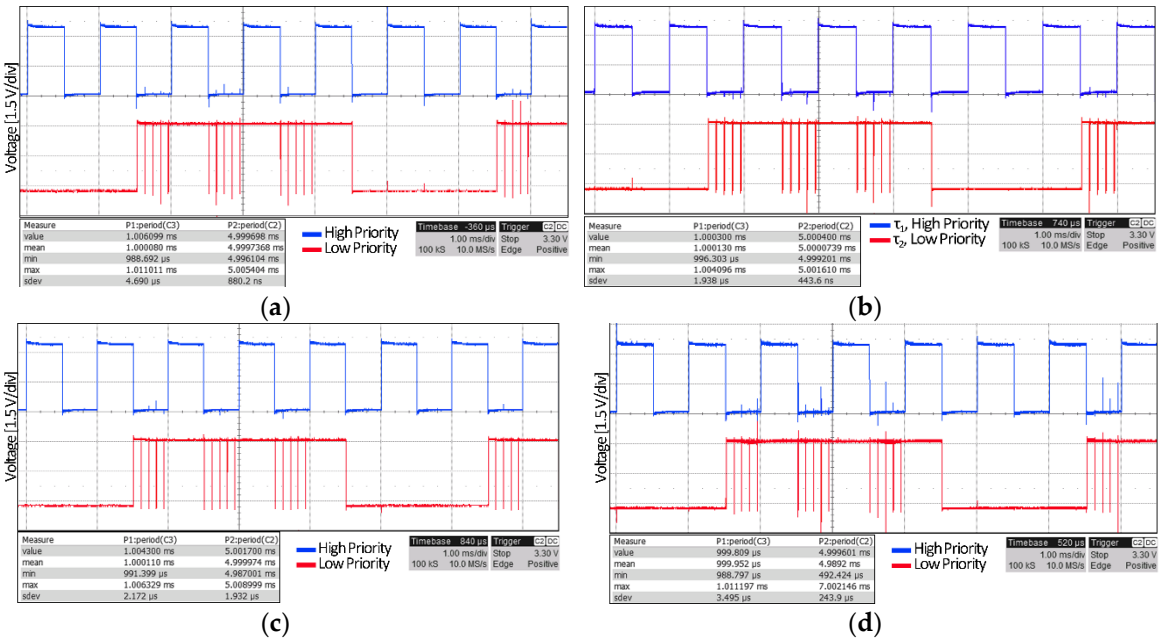

Figure 7. Periodicity of the real-time tasks with a real-time device driver: (**a**) BeagleBone Black; (**b**) Raspberry Pi 3 (**c**) Zybo-7020; (**d**) i.MX6Q SABRELite.

4. Example Cases Using OES as Real-time Controllers

Given the advantages of OES, various practical applications have been proposed in the literature and research projects with most focusing on the distributed control of numerous devices interfaced directly to the real-time controllers. In this section, three example cases are presented to demonstrate the validity of OES as main controllers of industrial robots and an omnidirectional mobile robot [37] on an EtherCAT network, and the integration of Robot Operating System (ROS) to Xenomai real-time tasks to control a telepresence robot [38,39].

4.1. Real-Time Controller for Joint Space Motion of an Omnidirectional Mobile Robot

Mobile robot control requires a DCS composed of a main controller, servo drives, actuators, and sensors to interact with the environment. The main controller is responsible for the calculation of the motion commands and collection of the feedback from the other components. It should be small enough to enable the robot in navigating easily within the environment. There are several attempts of embedded controllers for mobile robots [40–42]. However, these studies were unable to accurately track a desired path. This is due to the failure of meeting real-time requirements. We have developed an EtherCAT-based omnidirectional mobile robot [37] to ensure real-time responses of all the connected devices. To address the size and power requirements of the main controller, we have selected the embedded platform i.MX6Q SABRELite. Figure 8a shows the control architecture with an EtherCAT master based on the i.MX6Q SABRELite. IgH EtherLAB 1.5.2 was implemented on top of the real-time environment discussed in Section 2. Figure 8b shows the actual image of the mobile robot without the cover to make the internals visible as possible.

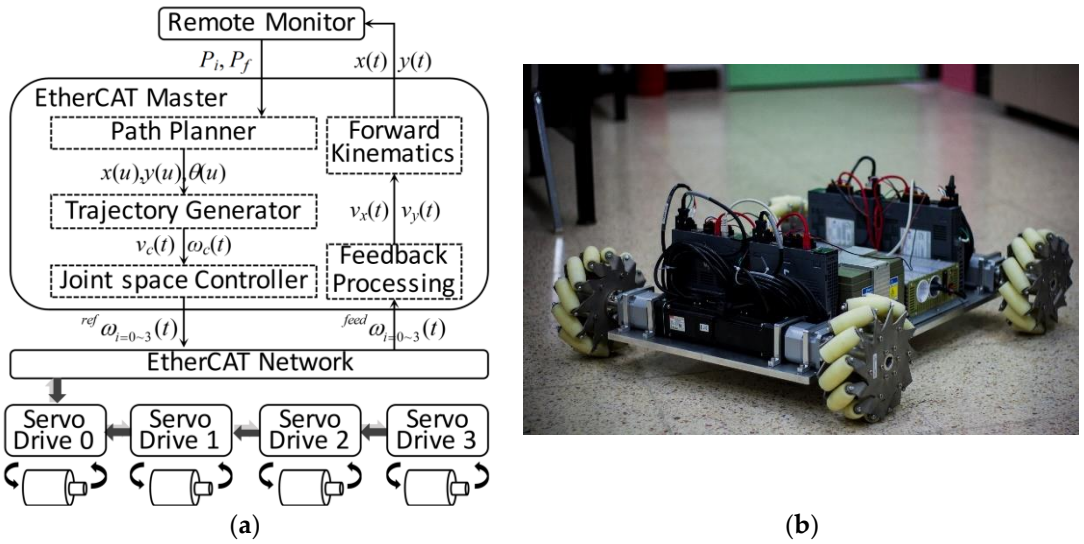

Figure 8. EtherCAT-based omnidirectional mobile robot: (**a**) Control architecture; (**b**) Actual image of the mobile robot.

To demonstrate the validity of the system, a path planning method in Reference [10], was implemented to generate a path from (0 m, 0 m) to (3 m, 1.5 m). The convolution-based trajectory generator produces central velocity commands that can track the planned path as shown in the black solid line in Figure 9a. This is decomposed into the joint space velocities through the joint space controller. The joint space velocities v_i (I = 0,1,2,3) are the actual velocities sent to the actuators. The feedback from the encoders are acquired and are analyzed to show the actual position of robot as shown in Figure 9b. The results show that the mobile robot was able to accurately track the desired path with minimal error at the end point.

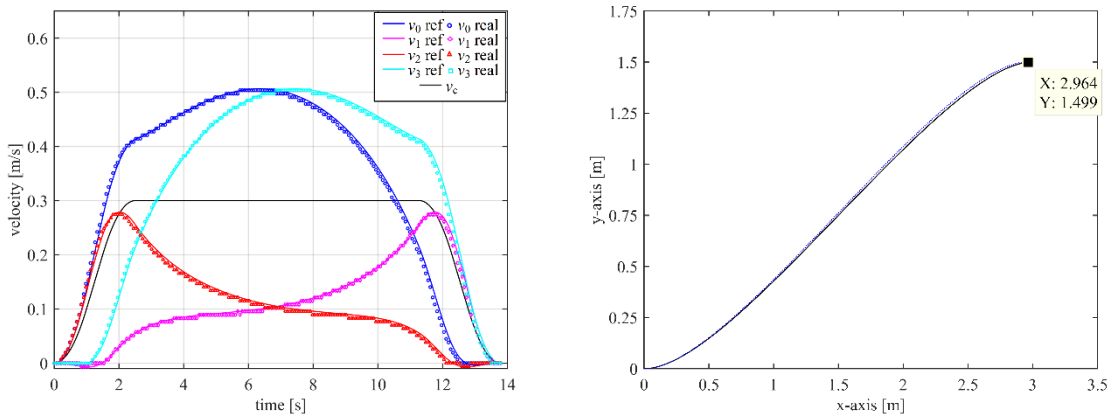

Figure 9. Navigation of the omnidirectional mobile robot: (**a**) Velocity profiles; (**b**) Trajectory.

4.2. Integration of ROS in Real-Time Control Systems

Robot operating system (ROS) is the most dominant open source robotic platform that offers various robot control algorithms that available as easily redistributable packages. The main drawback of ROS is that it does not operate in real-time. As most software, including device drivers and ROS, are originally designed for the standard Linux, we have implemented a message-passing communication mechanism to successfully pass data which can avoid mode switching (refer to Section 3.3). In comparison to the message queue in the previous section, the cross-domain datagram protocol (XDDP) can transfer data from the Xenomai domain and the Linux domain without the risk of the real-time tasks being scheduled by the standard Linux scheduler. To this end, a control application for a telepresence robot, named M4K [43], was designed aiming to easily realize navigation scheme using ROS navigation packages [38,39]. The mobile base includes actuators and sensors. Standard Linux device drivers are developed for each of the devices and are executed in the multi-tasking environment of Xenomai. On a Raspberry Pi 3, the ROS nodes and Linux tasks are connected to the Xenomai using XDDP as shown in Figure 10.

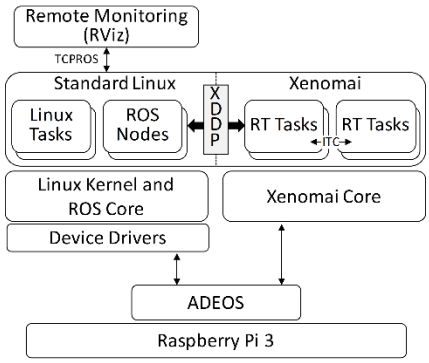

Figure 10. Integration of ROS to Xenomai real-time tasks.

The navigation package includes various path planners that can be selected by the users depending on the kinematics of the mobile robot in hand. In case of the M4K, we have selected the default *base_local_planner* because of its simplicity and compatibility to two-wheeled differential drive mobile robots. The mobile base of the robot is equipped with different sensory and actuator devices such as ultrasonic distance sensor (Sonar), inertial measurement unit (IMU), laser rangefinder (LRF), and light-emitting diode (LED) strips. The actuators are consisted of two DC motors with absolute encoders. Each device has their own standard Linux device drivers connected to five Xenomai real-time tasks using XDDP. From the worst-case execution time presented in Reference [39], schedulability

of the real-time tasks for the navigation of the robot were analyzed using MAST and the results are shown in Table 6. The worst-case response time of the lowest priority task is 2.273 ms, which is less than the respective hard real-time deadline. Meaning, all the tasks including high priority tasks can execute within their respective hard temporal deadline all the time. The MAST analysis shows that the all the tasks are schedulable with 914.06% of system slack. Although we have presented a simple application, more sophisticated real-time systems need worst-case response-time analysis to determine schedulability of the entire system. MAST is a very valuable tool to meet these requirements.

Table 6. Schedulability analysis for the navigation of M4K using MAST [36].

Task	Period (ms)	Execution (ms)	Priority	Worst-Case Response (ms)	Slack (%)
Actuator	10.000	0.029	99	0.029	31,082.4%
IMU	10.000	0.432	95	0.461	2086.3%
LED	20.000	0.590	90	1.051	3055.5%
Sonar	30.000	0.460	85	1.511	5864.5%
LRF	200.000	0.762	80	2.273	23,694.1%

5. Conclusions

In this paper, we present the design details to minimize the development cost and for easier distribution of industrial DCSes adopting OES as real-time controllers. Depending on the control application, developers should carefully select the suitable hardware platform considering the available interfaces and hardware layout. Additionally, compatibility of the software components with the hardware platform and other software is another issue that highly depends on the designed application. We have provided implementation procedures in detail considering software compatibility for real-time environment based on Xenomai for various open hardware platforms. To verify the real-time performance of each OES, experiments were conducted focusing on the periodicity and response time of real-time tasks which were also evaluated using MAST [36] for schedulability analysis. Timing characteristics of the real-time mechanisms, which ensure accuracy of data manipulation in a multitasking environment were also measured. The interrupt response time using RTDM of Xenomai was measured to show the latency improvements in comparison to the standard Linux device drivers. Although the measurements for each metric showed small difference between the OES-based real-time controllers, the acquired results are very important to serve as references in predicting the behavior of OES when applied in more complex and advanced distributed control systems. Additionally, we have given suitable example practical cases for each embedded platform provided practical applications that were discussed in detail in various control applications. Given these advantages, it is in the view of the authors that OES-based real-time controllers are worthy of consideration as alternatives to their proprietary counterparts.

Author Contributions: R.D. surveyed the background of this research, developed the environment for each embedded hardware, formulated the experiment procedures, and analyzed the results to show the benefits and performance comparison of the real-time controllers. J.P. performed the experiments and assisted in the performance evaluation. B.W.C. supervised and supported this study.

Funding: This research received no external funding.

Acknowledgments: This work was supported by the Human Resources Development of the Korea Institute of Energy Technology Evaluation and Planning (KETEP) grant funded by the Ministry of Trade, Industry & Energy of the Korea government (No. 20174030201840).

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

The higher priority task, τ_1, would not experience any blocking or interference, thus the busy period is equal to 0.5 ms. On the other hand, because of the interference from the higher priority task, the busy period of τ_2 is calculated as follows:

$$W_2^0 = 1.5 + \left\lceil \frac{1.5}{1} \right\rceil \cdot 0.5 = 1.5 + 1 = 2.5$$
$$W_2^1 = 1.5 + \left\lceil \frac{2.5}{1} \right\rceil \cdot 0.5 = 1.5 + 1.5 = 3.0$$
$$W_2^2 = 1.5 + \left\lceil \frac{3.0}{1} \right\rceil \cdot 0.5 = 1.5 + 1.5 = 3.0$$

Therefore, the expected response time of τ_2 is approximately 3 ms.

References

1. Colnaric, M.; Verber, D.; Halang, W.A. *Distributed Embedded Control Systems: Improving Dependability with Coherent Design*; Springer Publishing Company: London, UK, 2008; p. 250.
2. Brennan, R.W.; Fletcher, M.; Norrie, D.H. An agent-based approach to reconfiguration of real-time distributed control systems. *IEEE Trans. Robot. Autom.* **2002**, *18*, 444–451. [CrossRef]
3. Omidvar, M.N.; Yang, M.; Mei, Y.; Li, X.; Yao, X. Dg2: A faster and more accurate differential grouping for large-scale black-box optimization. *IEEE Trans. Evol. Comput.* **2017**, *21*, 929–942. [CrossRef]
4. Paschali, M.-E.; Ampatzoglou, A.; Bibi, S.; Chatzigeorgiou, A.; Stamelos, I. Reusability of open source software across domains: A case study. *J. Syst. Softw.* **2017**, *134*, 211–227. [CrossRef]
5. Barbalace, A.; Luchetta, A.; Manduchi, G.; Moro, M.; Soppelsa, A.; Taliercio, C. Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application. *IEEE Trans. Nucl. Sci.* **2008**, *55*, 435–439. [CrossRef]
6. Abbott, D. *Linux for Embedded and Real-Time Applications*, 4th ed.; Butterworth-Heinemann: Newton, MA, USA, 2003; p. 250.
7. Choi, B.W.; Shin, D.G.; Park, J.H.; Yi, S.Y.; Gerald, S. Real-time control architecture using xenomai for intelligent service robots in usn environments. *Intell. Serv. Robot.* **2009**, *2*, 139–151. [CrossRef]
8. Oliveira, D.B.; Oliveira, R.S. Timing analysis of the preempt rt linux kernel. *Softw. Pract. Exper.* **2016**, *46*, 789–819. [CrossRef]
9. Qian, K.Q.; Wang, J.; Gopaul, N.S.; Hu, B. Low cost multisensor kinematic positioning and navigation system with linux/rtai. *J. Sens. Actuator Netw.* **2012**, *1*, 166–182. [CrossRef]
10. Yang, G.J.; Delgado, R.; Choi, B.W. A practical joint-space trajectory generation method based on convolution in real-time control. *Int. J. Adv. Robot. Syst.* **2016**, *13*, 56. [CrossRef]
11. Dantam, N.T.; Lofaro, D.M.; Hereid, A.; Oh, P.Y.; Ames, A.D.; Stilman, M. The ach library: A new framework for real-time communication. *IEEE Robot. Autom. Mag.* **2015**, *22*, 76–85. [CrossRef]
12. Cereia, M.; Bertolotti, I.C.; Scanzio, S. Performance of a real-time ethercat master under linux. *IEEE Trans. Ind. Inform.* **2011**, *7*, 679–687. [CrossRef]
13. Jitendrasinh, T.B.; Deshpande, S. Implementation of can bus protocol on xenomai rtos on arm platform for industrial automation. In Proceedings of the 2016 International Conference on Computation of Power, Energy Information and Commuincation (ICCPEIC), Chennai, India, 20–21 April 2016; pp. 165–169.
14. Kim, I.; Kim, T. Guaranteeing isochronous control of networked motion control systems using phase offset adjustment. *Sensors* **2015**, *15*, 13945–13965. [CrossRef] [PubMed]
15. Brown, J.H.; Martin, B. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. In Proceedings of the 12th Real-Time Linux Workshop, Nairobi, Kenya, 25–27 October 2010; pp. 1–17.
16. Ferdoush, S.; Li, X. Wireless sensor network system design using Raspberry Pi and Arduino for environmental monitoring applications. *Procedia Comput. Sci.* **2014**, *34*, 103–110. [CrossRef]
17. Chianese, A.; Piccialli, F.; Riccio, G. Designing a smart multisensor framework based on beaglebone black board. In *Lecture Notes in Electrical Engineering*; Springer: Heidelberg, Germany, 2015; pp. 391–397.

18. Honegger, D.; Meier, L.; Tanskanen, P.; Pollefeys, M. An open source and open hardware embedded metric optical flow cmos camera for indoor and outdoor applications. In Proceedings of the 2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, 6–10 May 2013; pp. 1736–1741.
19. Kaliński, K.J.; Mazur, M. Optimal control at energy performance index of the mobile robots following dynamically created trajectories. *Mechatronics* **2016**, *37*, 79–88. [CrossRef]
20. Zhang, L.; Slaets, P.; Bruyninckx, H. An open embedded hardware and software architecture applied to industrial robot control. In Proceedings of the 2012 IEEE International Conference on Mechatronics and Automation, Chengdu, China, 5–8 August 2012; pp. 1822–1828.
21. Su, Y.; Qiu, Y.; Liu, P. The continuity and real-time performance of the cable tension determining for a suspend cable-driven parallel camera robot. *Adv. Robot.* **2015**, *29*, 743–752. [CrossRef]
22. Arm, J.; Bradac, Z.; Kaczmarczyk, V. Real-time capabilities of linux rtai. *IFAC-PapersOnLine* **2016**, *49*, 401–406. [CrossRef]
23. Kiszka, J. The real-time driver model and first applications. In Proceedings of the 7th Real-Time Linux Workshop, Lille, France, 2–4 November 2005; pp. 1–8.
24. Xenomai Archives. Available online: http://xenomai.org/downloads/xenomai/stable (accessed on 10 November 2018).
25. Beaglebone Black Linux Kernel. Available online: https://github.com/RobertCNelson/bb-kernel (accessed on 17 November 2018).
26. Raspberry pi 3 Linux Kernel. Available online: https://github.com/raspberrypi/linux (accessed on 22 November 2018).
27. Zybo-7020 Linux Kernel. Available online: https://github.com/Xilinx/linux-xlnx (accessed on 25 November 2018).
28. MX6Q SABRELite Linux Kernel. Available online: https://github.com/RobertCNelson/armv7-multiplatform (accessed on 25 November 2018).
29. Pipe Patch Archives. Available online: https://xenomai.org/downloads/ipipe/v3.x/arm/older (accessed on 10 November 2018).
30. Raspberry pi 3 Image Repository. Available online: http://downloads.raspberrypi.org/raspbian_lite/images/raspbian_lite-2017-07-05 (accessed on 22 November 2018).
31. U-Boot Bootloader. Available online: http://git.denx.de (accessed on 10 November 2018).
32. Pose, F. Igh Ethercat Master 1.5.2 Documentation. Available online: https://www.etherlab.org/download/ethercat/ethercat-1.5.2.pdf (accessed on 27 November 2018).
33. Minimal Ubuntu 14.04. Available online: https://rcn-ee.com/rootfs/eewiki/minfs (accessed on 11 November 2018).
34. Delgado, R.; Hong, C.H.; Shin, W.C.; Choi, B.W. Implementation and perfomance analysis of an Ethercat master on the latest real-time embedded Linux. *Int. J. Appl. Eng. Res.* **2015**, *10*, 44603–44609.
35. Joseph, M.; Pandya, P. Finding response times in a real-time system. *Comput. J.* **1986**, *29*, 390–395. [CrossRef]
36. Harbour, M.G.; Garcia, J.J.G.; Gutierrez, J.C.P.; Moyano, J.M.D. Mast: Modeling and analysis suite for real time applications. In Proceedings of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, 13–15 June 2001; pp. 125–134.
37. Delgado, R.; Shin, W.C.; Hong, C.H.; Choi, B.W. Development and control of an omnidirectional mobile robot on an ethercat network. *Int. J. Appl. Eng. Res.* **2016**, *11*, 10586–10592.
38. Delgado, R.; You, B.-J.; Choi, B.W. Real-time control architecture based on xenomai using ros packages for a service robot. *J. Syst. Softw.* **2019**, *151*, 8–19. [CrossRef]
39. Delgado, R.; You, B.-J.; Han, M.; Choi, B.W. Integration of ros and rt tasks using message pipe mechanism on xenomai for telepresence robot. *Electron. Lett.* **2019**, *55*, 127–128. [CrossRef]
40. Mamun, M.A.A.; Nasir, M.T.; Khayyat, A. Embedded system for motion control of an omnidirectional mobile robot. *IEEE Access* **2018**, *6*, 6722–6739. [CrossRef]
41. Arvin, F.; Espinosa, J.; Bird, B.; West, A.; Watson, S.; Lennox, B. Mona: An affordable open-source mobile robot for education and research. *J. Intell. Robot. Syst.* **2018**, 1–15. [CrossRef]

42. López-Rodríguez, F.M.; Cuesta, F. Andruino-a1: Low-cost educational mobile robot based on android and arduino. *J. Intell. Robot. Syst.* **2016**, *81*, 63–76. [CrossRef]
43. You, B.-J.; Kwon, J.R.; Nam, S.-H.; Lee, J.-J.; Lee, K.-K.; Yeom, K. Coexistent space: Toward seamless integration of real, virtual, and remote worlds for 4D+ interpersonal interaction and collaboration. In Proceedings of the SIGGRAPH Asia 2014 Autonomous Virtual Humans and Social Robot for Telepresence, Shenzhen, China, 3–6 December 2014; pp. 1–5.