The Journal of Systems and Software 151 (2019) 8–19

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Real-time control architecture based on Xenomai using ROS packages for a service robot

Raimarius Delgado[a], Bum-Jae You[b], Byoung Wook Choi[a,*]

[a] Department of Electrical and Information Engineering, Seoul National University of Science and Technology, 232 Gongneung-ro, Nowon-gu, Seoul 01811, South Korea

[b] Center of Human-centered Interaction for Coexistence, Hwarangno 14-gil 5, Seongbuk-gu, Seoul 02792, South Korea

ARTICLE INFO

Article history:
Received 9 May 2018
Revised 22 January 2019
Accepted 23 January 2019
Available online 24 January 2019

Keywords:
Real-time control architecture
Robot operating system
Xenomai
Cross-domain datagram protocol
Service mobile robots

ABSTRACT

This paper proposes a real-time (RT) control architecture based on Xenomai, an RT embedded Linux, to control a service robot along with non-real-time (NRT) robot operating system (ROS) packages. Most software, including device drivers and ROS, are developed to operate under the standard Linux kernel that does not provide RT guarantees. Standard Linux system calls in an RT context stimulates mode switching, resulting in non-deterministic responses and stability problems such as priority inversion and kernel panic. This paper overcomes such issues through a communication interface between RT and NRT tasks, termed cross-domain datagram protocol. The proposed architecture supports priority-based scheduling of multiple tasks while exposing an interface compatible with the original ROS packages. Moreover, it enables standard device driver operation inside RT tasks without developing RT device drivers that requires significant amount of development time. Feasibility is proven by implementation on a Raspberry Pi 3, a low-cost open embedded hardware platform, and conducted various experiments to analyze its performance and applied it to a service robot using ROS navigation packages. The results indicate that the proposed architecture can effectively provide an RT environment without stability issues when utilizing ROS packages and standard device drivers.

1. Introduction

The great demand for high-performance robots from the community results in a steady growth of complex hardware, software, and control architecture such that robots can efficiently interact with the environment. This includes the integration of numerous devices including sensors and actuators which require processing and acquisition of a significant amount of data to properly perform specific tasks. It is generally known that software complexity primarily depends on the hardware connected within a system (Vogel-Heuser et al., 2015).

Currently, software development methods are often based on custom software that are built from scratch or expensive proprietary architectures that are distributed in a black box, which hinders the integration to a more complex software (Omidvar et al., 2017; Paschali et al., 2017). Open-source projects overcome these problems by enabling developers to freely add and modify the source code, meaning better quality and lesser prone to bugs as more users can simultaneously contribute to the evolution of the system and debug any problems.

Further, open-source projects are developed to operate in many systems and should abstract the hardware and basic software to avoid the dependence on a particular vendor (Kilamo et al., 2012). Many studies are ongoing on a new methodology to enhance the design of robot control systems with the aim of an easier, faster, and more reliable development based on open-source software. This includes component-based frameworks such as the robot operating system (ROS) project (Zucker et al., 2015), yet another robot platform (YARP) (Cardellino et al., 2018), and open robot control software (OROCOS) (Ahmad and Babar, 2016).

Another crucial requirement to consider in designing robot applications is the precise control period (Li et al., 2016). In a system with complex software and various devices such as a robot control, the typical *super loop* concept (Fischmeister and Lam, 2010), where each function is executed in a fixed order and uses interrupt service routines (ISRs) for time-critical programs, is not applicable especially in a configuration where each component requires different cycle times. As the ISR becomes more difficult, it requires longer response times that block the completion of other functions inside the loop. Therefore, controlling the robot with strict real-time (RT) constraints is impossible.

* Corresponding author.
E-mail addresses: raim223@seoultech.ac.kr (R. Delgado), ybj@chic.re.kr (B.-J. You), bwchoi@seoultech.ac.kr (B.W. Choi).

https://doi.org/10.1016/j.jss.2019.01.052
0164-1212/

Consequently, it is highly likely for a mobile robot to collide with an obstacle because it must receive data from all the attached sensors first before it can perform an avoidance scheme. Real-time operating systems (RTOSes) are required to deal with this issue. An RTOS separates functions into self-contained tasks in a multitasking environment and implements the execution scheduling based on priorities, meaning lower priority tasks can be pre-empted by higher priority tasks. This ensures a deterministic behavior because interrupts from the hardware components and events from the algorithms are handled within a predefined time.

Model-based approaches including Matlab, Simulink, and LabView offers real-time toolboxes and modules for rapid development of real-time applications. Matlab emulates an RTOS for software simulations (Grepl, 2011) but is not viable in practical scenarios involving various digital devices. Simulink and LabView provide real-time modules with devices drivers for hardware-based simulation that requires a dedicated real-time computer (Henriksson et al., 2002; MathWorks 2019; Beck et al., 2006). All the software mentioned above are distributed commercially and are very expensive. A low-cost method which implements an open-source RTOS for hardware-in-the-loop simulations in power systems are presented in (Lu et al., 2007). Instead of using a dedicated real-time computer, they implemented on the same machine an open-source RTOS to handle real-time tasks in the Simulink for plant and control modeling. Their results show that the system was able to satisfy *soft* real-time requirements. Robot control, on the other hand, is an advanced system that includes the implementation of vast combinations of digital hardware and algorithms, satisfying hard real-time requirements.

An advanced approach involving domain specific languages and innovative model-driven software development of real-time software architectures for robots was presented by Gobillot et al. (2018). Their work includes a design methodology of software architectures, real-time analysis considering safety of software applications, and a code generation toolchain ensuring that the analyzed software eventually executed in the robot itself. Although very commendable in the software perspective, in this paper, we are focused on a real-time system based on fully open-source software for easier reproduction and low-cost development.

Linux, the most popular open-source operating system, is currently considered as a soft RTOS owing to the dramatic advances in computing power and the continuous efforts of developers. However, as the scheduling policy utilizes *fairness* instead of the *priority* of processes, it is still not suitable for hard RT applications that require the system to meet strict timing constraints and preemption of low priority tasks (Abbott, 2013). Several RT extensions of Linux were introduced in the recent past that improved the response times and priority scheduling of the Linux kernel for hard RT applications (Yang et al., 2016; Alho and Mattila, 2015; Zappulla et al., 2017).

RT embedded Linux is classified into two major approaches: the pre-emptible kernel and the dual-kernel approaches. In the pre-emptible kernel approach, all parts of the standard Linux kernel with relationship to the scheduler and timers are modified to render lower priority tasks pre-emptible by higher priority ones. De Oliveira and De Oliveira (2016) evaluated its performance. 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 (ADEOS) (Dantam et al., 2015). Ceria et al. presented a comparison of the real-time performance between the RT_PREEMPT (pre-emptible kernel) and RTAI (dual-kernel) as the main controller of an EtherCAT network (Cereia et al., 2011). As the RTAI project becoming stagnant and its limited support of CPU architectures, Brown and Martin (2018) provide an evaluation of Xenomai and its comparison to the pre-emptible kernel. Both studies concluded that the pre-emptible kernel approach of RT_PREEMPT is significantly better in terms of limiting the maximum scheduling jitter. On the other hand, the dual-kernel approach showed better accuracy in meeting the hard RT deadlines with lower standard deviations.

As we aim to develop a rigorous control architecture with the best hard RT performance as possible, the focus of this paper is to develop an environment based on the dual-kernel approach of Xenomai. Another issue for the RT_PREEMPT approach is that it requires modification of the kernel code every time the version is upgraded to maintain RT effectiveness (Gosewehr et al., 2016). Whereas, Xenomai runs independent with the standard Linux kernel if there is a compatible ADEOS patch. Xenomai is a real-time framework that cooperates with the standard Linux kernel to provide hard real-time support for user-space RT tasks (Choi et al., 2009). In an RT task, any system call from the standard Linux domain introduces an event, called mode switching. This occurs when the execution of the RT task unexpectedly switches to the standard Linux execution mode handled by the standard Linux scheduler. The most common reason for mode switching is invoking a system call during the access to non-real-time (NRT) resources such as standard Linux libraries and device driver routines. Mode switching causes priority inversion that allows NRT tasks to pre-empt higher priority RT tasks and unstable behavior in the entire system that leads to missed critical temporal deadlines, or worse, a system freeze.

Because of this constraint, it is impossible to achieve hard real-time performance, even with Xenomai, while using any software designed for the standard Linux kernel. These include device drivers and ROS for example. ROS is the most dominant robotic platform for easier integration of various control algorithms available as easily redistributable packages. However, it does not operate in real-time. Developing an RT ROS is an open problem with very few available solutions. The most common approach is running ROS on a host system and perform real-time control in a separate guest controller. Bouchier (2013) suggested implementing an RTOS on the guest hardware to execute RT tasks that are connected to a host system operating on the standard Linux to handle NRT ROS tasks. This is remarkably similar to the solution of Hasegawa et al. (2016) for the collaboration of robot technology middleware and TOPPERS embedded component system (RTM-TECS). These approaches cause a performance issues on the guest system, where the real-time tasks are bottlenecked by the communication protocol. Manufacturing cost is also an issue because these solutions require more than a single hardware platform.

Chitta et al. developed ros_control (Chitta et al., 2017), a robot-agnostic framework addressing the real-time issues of ROS control applications. Although it can support diverse types of robots through its hardware abstraction layer, multitasking is a concern because it uses the standard ROS communication between tasks that does not support real-time. Wei et al. (2016) proposed a method that explores the multicore architecture on a single hardware. Being dependent on Intel-based microprocessors for their inter-core communication mechanism, the flexibility and extendibility of this approach is questionable because many embedded platforms being used currently are based on the ARM architecture. Although this is an innovative way to solve the RT problem, the development of RT device drivers, is required to access hardware resources in the RT context.

Xenomai provides an application programming interface called the RT driver model, or RTDM (Kiszka, 2005). However, this requires a long development time that could require days or months depending on the characteristics and required features of the devices that will be used in an RT context. With the aim to design RT applications for robot control that does not require development of RT device drivers and possesses the advantages of Xenomai such as multitasking, priority-based scheduling, and

deterministic response, as well as the rapid development tools offered by ROS packages, we propose an RT control architecture by adapting a communication mechanism between RT and NRT tasks called cross-domain datagram protocol (XDDP).

The XDDP is an inter-process communication (IPC) mechanism based on the RTDM and RT tasks can communicate with NRT tasks without experiencing the stability issues prompted by mode switching. Anistratov et al. (2015) implemented the mechanism to execute NRT program codes and libraries within RT tasks for the digital control of a pulse power supply system for the Tokamak T-15. The application of XDDP for robot applications is presented by Muratore et al. (2017), where they developed XBotCore, a software platform for EtherCAT-based robot platforms. In comparison to our approach, XBotCore is a homogenous architecture where there is no direct communication between RT and NRT tasks, which could cause data transmission. In such, schedulability of the RT tasks is also questionable because the performance highly depends on the component called the Plugin Handler. It is also specifically designed to control EtherCAT-based robots, whereas our architecture is a general solution integrable to a wide range of real-time systems. Moreover, performance evaluation of the mechanism in RT applications when using NRT libraries is also not provided.

We herein comprehensively describe the XDDP and developed user-friendly APIs that emulate Xenomai native functions for the creation, evaluation, and configuration of RT tasks to easily perform communication with NRT tasks. Using these functions, RT tasks can exchange data with NRT tasks that perform functions allowable only in the standard Linux domain instead of directly accessing any resources from the standard Linux kernel, thus avoiding a mode switch. Hence, standard Linux device drivers can be used in the RT context without exerting additional effort to develop their RT counterparts. Similarly, ROS packages can also be used in RT tasks that ensure priority-based scheduling and hard RT support. The proposed architecture guarantees stability and the deterministic response of RT tasks while implementing ROS packages and standard Linux device drivers using the existing open-source technologies that allow the reuse and easy integration to more complicated RT applications.

In order to demonstrate the validity of the proposed RT control architecture, we have implemented it on an open hardware platform, i.e., Raspberry Pi 3 (RPi3). As the control architecture is based on easily accessible open source software and mechanisms, implementation on non-embedded systems or high-end computers are also viable. However, in practice, main controllers of the robot are often based on embedded hardware because of its portability, low power requirements, and relatively inexpensive costs in comparison to high-end computers while remaining competitive in terms of performance. Open embedded hardware is gaining popularity as the primary controllers in robot control application such as sensor networks (Ferdoush and Li, 2014; Chianese et al., 2015), image processing (Honegger et al., 2013), and navigation (Kaliński and Mazur, 2016; Zhang et al., 2012). However, developing an RT environment for embedded platforms is more difficult owing to the limited availability of systematic documentations and technical support. Although manufacturers provide Linux kernel sources, compatibility with other software and patches is an open problem (Li et al., 2010).

With the aim to serve as a helpful guideline for software developers, especially those with limited practical experience, this paper, for the first time, provides detailed instructions on developing an RT environment for the RPi3 based on Xenomai, from the compatible versions of toolchains and standard Linux kernel, to the necessary patches required for successfully operating Xenomai. For the easier extendibility of devices without hardcoding any changes on the device tree of the Linux kernel, we have implemented the latest stable version of ROS in Raspbian, a file system distribution of RPi3, with a small tradeoff—manually building ROS sources instead of directly installing from the Ubuntu repository, which slightly increases development time. The RPi3 serves as the primary controller for the mobile base of a telepresence robot called the M4K (Lee et al., 2016).

We have developed standard Linux device drivers that can interact with the attached sensors and actuators of the mobile robot. Experiments and performance evaluation were conducted to prove the feasibility of the proposed architecture. Standard Linux device drivers were controlled inside RT tasks and navigation of M4K in an environment with static obstacles was performed using ROS navigation packages. The results show that the proposed architecture can efficiently provide RT support for standard Linux device drivers and ROS packages without experiencing stability issues due to mode switching.

2. Real-time control architecture based on Xenomai

In this section, we discuss the composition of the proposed RT control architecture using the RT extension of Xenomai. We provide a detailed explanation of the overall structure of the system including the motivation and the current problems in the state-of-the-art. Furthermore, the components that were used to build the proposed architecture, such as the Xenomai POSIX skin and XDDP, are described in detail. The improvements for an easier integration and reuse are also highlighted in the following subsections.

2.1. System structure

The advances in hardware computing power and the dramatic improvement in the standard Linux kernel has gathered interests for them to be considered in different robotic applications. However, owing to the scheduler policy of utilizing *fairness* rather than *priority* (Grepl, 2011), the RT performance of the standard Linux kernel is still not sufficient in handling hard RT tasks such as feedback control, which is commonly applied in trajectory tracking and the safe control of end effectors with high velocity and force. This requirement is satisfied by RT embedded Linux approaches: pre-emptible-kernel approach of RT-Preempt (De Oliveira and De Oliveira, 2016) or the dual-kernel approach either by RTAI (Erwinski et al., 2013) or Xenomai (Gosewehr et al., 2016; Choi et al., 2009). We herein focus more on the dual-kernel approach because of its superior performance as reported in Cereia et al. (2011) and Brown and Martin (2018). We preferred Xenomai over RTAI because of its wide range of supported CPU architectures and its active community. RTAI supports only PowerPC and Intel CPUs and has not been updated since 2012.

Xenomai runs alongside the standard Linux kernel through the ADEOS, a nanokernel hardware abstraction layer that enables multiple entities called domains to exist in the same hardware (Dantam et al., 2015). In this configuration, Xenomai has the higher priority and is handled first, causing the standard Linux kernel to be configured with the lowest priority. Thus, any standard Linux task will execute, if and only if, there are no pending Xenomai tasks. One characteristic of Xenomai is that it can access resources and switch between the two kernels at any time.

However, hard RT operation is only possible when all the tasks are scheduled by the Xenomai scheduler. Meaning, RT tasks should only use native Xenomai services and RTDM device drivers. The following sections use the terms “primary mode” and “secondary mode”. Both terms are specific to Xenomai and are defined as the following: If a Xenomai real-time task is running without any Linux system calls or APIs within the code, it is said to be running in “primary” mode. Linux systems calls within Xenomai tasks cause migration of the task from the Xenomai scheduler to the standard Linux scheduler, thus making it run in “secondary” mode and lose

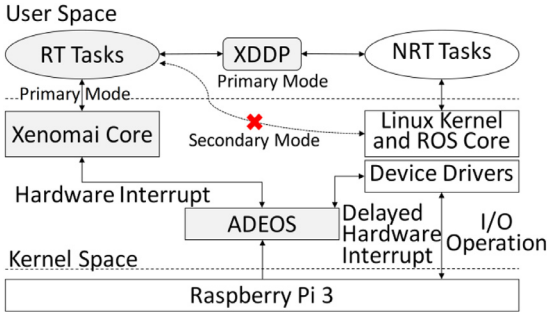


Fig. 1. Real-time control architecture based on Xenomai.

hard real-time capabilities. Any switch from primary to secondary mode is called "mode switching".

In a mode switch, RT tasks are vulnerable to priority inversion: higher priority tasks being interrupted or pre-empted by lower priority ones. This is a chaotic scenario which must be avoided in RT scheduling that can cause the entire system to become non-deterministic. In the worst case, a "system freeze" or a kernel panic can occur, especially when using standard Linux device drivers.

As most software including device drivers and ROS are originally designed for the standard Linux, a communication interface to successfully pass data which can avoid mode switching is required. Shared memory and message passing are the two types of methods to satisfy this requirement. In shared memory, a task can publish data in a piece of memory and the readers can decipher the data within that region. Concurrency issues can cause data loss owing to the lack of synchronization between the parallel tasks. RT synchronization mechanisms, such as a semaphore, are required to avoid this anomaly. However, it is not allowed for RT and NRT tasks to use the same mechanism.

In our proposed architecture, we utilized a pipe mechanism called the XDDP as the medium between RT and NRT tasks. The XDDP is available in the POSIX skin of Xenomai, which will be explained in the next subsection. In comparison to shared memory, the XDDP is a message-passing mechanism based on the RTDM and can solve the issues regarding data concurrency and mode switching. This solution is essential because software is typically developed only for the standard Linux such as device drivers and robot frameworks like ROS.

Instead of directly accessing standard Linux resources, RT tasks are connected to NRT tasks that perform the required functions from either the device driver or ROS. The proposed architecture is an intuitive approach to integrate ROS packages inside RT tasks that can ensure priority-based scheduling and the deterministic response of the system. Further, the development of RT drivers based on the RTDM is also eliminated, thus saving significant amount of development cost and time.

The proposed real-time control architecture is shown in Fig. 1. Xenomai is implemented alongside the standard Linux kernel that possesses device drivers and the ROS master core. Xenomai is assigned the highest priority by the ADEOS; thus, it receives hardware interrupts without being delayed. In contrast, NRT tasks can only execute after Xenomai tasks are finished. When the RT task is composed only of Xenomai APIs or is connected to the NRT task through the XDDP, it can consistently execute in the primary mode and guarantees hard RT operation. However, mode switching occurs when an RT task enters the secondary mode (dotted line), which is not recommendable, by directly accessing the standard Linux resources in this case, ROS core, and device drivers.

As mentioned in the previous section, XDDP was implemented to avoid any case of mode switching when using standard Linux system calls inside RT tasks by the researchers in (Kiszka, 2005; Anistratov et al., 2015). However, the architecture of these studies did not offer the detailed instructions and performance evaluation of the mechanism in RT applications, which is addressed in this work.

2.2. Xenomai POSIX skin

Xenomai provides various user space libraries to emulate common RTOS APIs such as VxWorks and PSOS. It also offers its own API called the native skin. However, the XDDP is included in the POSIX skin that uses standard POSIX APIs, wrapped with Xenomai functions, inside RT tasks.

As it is not common to use POSIX functions to develop Xenomai applications, we have created simple APIs that emulate the native skin for better task and timer management. Further, instead of using the POSIX timer that uses signals or system calls, we have opted to use the high-resolution timer approach that uses a file descriptor through *timerfd* (Chianese et al., 2015).

- *pt_create_task_rt*() creates and starts an RT task with the desired priority, stack size, period, and task name. This function encapsulates the functions related to the thread and attribute handlers *pthread_t* and *pthread_attr*, and the thread creation function *pthread_create*(). This function is equivalent to the native skin function, *rt_task_spawn*().
- *pt_task_wait_period*() wait for the next periodic release point using a file descriptor or *timerfd*. In the native skin, this function is known as *rt_task_wait_period*().
- *pt_timer_read*() returns the current system time expressed in nanoseconds. This function uses *clock_gettime*() to measure the current time using the *timespec* structure. This function converts time stamp counters to nanoseconds. *rt_timer_read*() is the equivalent function in the Xenomai native skin.

2.3. Integration of cross-domain datagram protocol

Cross-domain datagram protocol or XDDP is a type of RT interprocess communication (RTIPC) mechanism offered by Xenomai. It is a message-passing interface based on the RTDM that exports a socket interface and allows a two-way channel communication to exchange datagrams between Xenomai RT tasks and standard Linux threads/processes using regular file operations for simplicity. It connects a socket to a pseudo device file in the standard Linux located in the root filesystem device (/dev) directory.

The XDDP can be operated in one of two modes: message oriented and byte streaming. In the message-oriented mode, a message buffer is pre-allocated to preserve a specific message boundary during a transaction. The message buffer should be filled before sending to the XDDP socket. This behavior can introduce blocking and huge delays in the execution of the RT task. This configuration is usually used when correctness of the data is more important than the speed of execution. The byte-streaming mode is a continuous stream that can yield the optimal output. As we aim to establish a continuous stream of asynchronous data from ROS and device drivers and improve reactiveness of RT tasks, all operations were conducted using the byte-streaming mode.

The XDDP structure is shown in Fig. 2. In this figure, the functions required to transfer data from RT tasks to NRT tasks and vice-versa are shown by the lines with an arrow head according to the corresponding direction. In the Xenomai side, an RT task should be bound to a socket through the function *bind*(). As this function is associated with the network interface of standard Linux, this should be called before the task enters the infinite loop. To bind a socket to an XDDP port, the parameters AF_RTIPC and IPCPROT_XDDP should be passed as arguments for the domain and protocol to be used, respectively. The socket acts as a communication proxy to send and receive data from the pseudo device in the

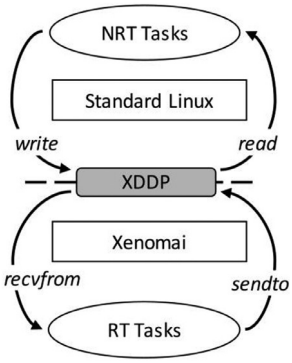


Fig. 2. Structure of cross-domain datagram protocol.

standard Linux domain. Meanwhile, an NRT task requires a standard character device file found in /dev/rtpX (X stands for the minor number) to be opened. As this is an ordinary device file, the function *open*() is used to create an XDDP instance in the NRT task.

XDDP ports are identified by the minor numbers of the device file, meaning that port 0 in the RT task is in connection with the device file /dev/rtp0. The number of useable XDDP ports depends on the kernel configuration namely, CONFIG_XENO_OPT_PIPE_NRDEV, which is 32 by default. From the Xenomai domain, data are sent to the XDDP port using *sendto*() for the NRT task to receive from the device file using *read*(). Conversely, data from the NRT task is sent *write*() and received by the RT task using the *recvfrom*() system call.

In summary, any task (ROS nodes or device driver handler) running in the NRT domain can access the XDDP ports using file descriptor operations. Conversely, in the RT domain, XDDP ports are accessed similar to typical socket operations. The data transfer operations using *write*()/*read*() or *sendto*()/*recvfrom*() involves a void buffer. Although there is no special data conversion required, users should be wary that the data type inserted on the buffer should be maintained for a seamless transfer of data between the RT and NRT domain.

For an easier implementation, we have also created an API that creates the descriptors for both RT and NRT tasks.

- *XENO_XDDP* is the structure that describes the XDDP port inside the RT task. This includes the XDDP port number to be used, send and receive buffers, and mode of operation.
- *LINUX_XDDP* is the structure that describes the XDDP implementation to be called in the NRT task. Within the structure, the minor number of the XDDP port, write and read buffers, and mode of operation.
- *rt_xddp_init*() is the function used to initialize an RT XDDP socket to the device file after initialization.
- *nrt_xddp_init*() is a similar function with the *rt_xddp_init*() but is initialized in the NRT domain.
- *rt_xddp_bind*() is the function used to bind an XDDP socket to the device file after initialization.
- *nrt_xddp_open*() is the function used to open a device file after initialization and is connected to an XDDP port.
- *rt_xddp_send*/*recv*() is the function that sends/receives data from the XDDP port, respectively.
- *nrt_xddp_write*/*read*() is the function that read/write to the device file connected to an XDDP port, respectively.

The sequence of major operations when implementing XDDP for the communication between an RT and an NRT task is as follows:

- An RT task is created using *pt_create_task_rt*() defining the desired priority, stack size, and deadline. A simple POSIX thread is created using *pthread_t*() that serves as the NRT task.
- Initialize XDDP structures for both the RT and NRT task using *XENO_XDDP* and *LINUX_XDDP*, respectively. Initialization can be performed either inside the respective task or as global resources.
- Inside the RT task, an XDDP socket is initialized for using the *rt_xddp_init*() function with the *XENO_XDDP* structure as an argument. On the other hand, the file descriptor in the Linux side is initiated using *nrt_xddp_init*() in the NRT task.
- The XDDP is bound to a socket using *rt_xddp_bind*(). This function, altogether with *rt_xddp_init*() should be performed before the real-time task loop. In the NRT task, the file descriptor is opened using *nrt_xddp_open*().
- *rt_xddp_send*/*recv*() is used inside the RT task to transfer and receive data from the XDDP socket while *nrt_xddp_write*/*read*() are used in the NRT task.

A simple example for the implementation of the XDDP communication is provided at https://github.com/SeoulTechEmbeddedLab/xddp.

3. Implementation of control architecture and device drivers

In order to demonstrate the validity of the proposed RT control architecture, we perform a navigation scheme for a telepresence robot called M4K. The robot is originally designed with an interactive 3D beam projector and cameras in addition to the basic telepresence robot to facilitate extensive human communication across distances (Lee et al., 2016). We have selected an embedded platform, RPi3, to show that the architecture is operational even in an embedded environment to reduce manufacturing costs and increase portability of the system.

Further, we conducted experiments to verify the feasibility of the proposed architecture to control the M4K using ROS packages running under RT tasks with priorities. The primary focus of the experiments is to prove the viability of the proposed architecture without mode switching and kernel panic while meeting the expected deadline. The performance measurements were concentrated only on the periodicity of RT tasks during data exchange with NRT task when using standard Linux device drivers and ROS packages to control M4K.

3.1. Real-time environment for Raspberry Pi 3 using ROS and Xenomai

Because the proposed architecture relies on the RT features offered by Xenomai, the development of an RT environment for the RPi3 is needed. Unlike Intel-processor-based desktop computers, the development of an RT environment for an embedded platform is more difficult owing to the limited availability of systematic documentations and technical support. This paper, for the first time, provides detailed instructions on the implementation of Xenomai and ROS on the RPi3. The RT computing environment that we have implemented on an RPi3, considering the compatibility of each software is shown in Fig. 3.

The latest available Xenomai version for RPi3 is the Xenomai 3.0.2 based on Linux kernel 4.1.21. To support both kernels in the same machine, the ipipe-core-4.1.18-arm-4 of ADEOS is ported on top of the Broadcom bootloader. Although the latest version of the ROS is available, we have selected the stable version of the ROS Kinetic Kame (Open Source Robotics Foundation, 2018). Ubuntu-MATE 16.04 is the recommended distribution and root filesystem for the ROS Kinetic. However, owing to the limitation in the modification of the device tree binary (DTB) to extend the SPI bus interface (motor and encoder support), disabling Bluetooth for full

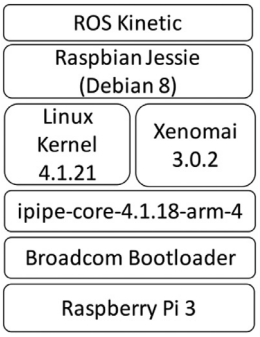


Fig. 3. Real-time environment of a Raspberry Pi 3.

RS232 support, and disabling the sound card (PWM for LED), we have selected the Raspbian Jessie (Debian 8) Linux distribution.

Xenomai is realized by sharing hardware resources with the standard Linux kernel through a hardware abstraction layer called ADEOS. CPU frequency scaling, programmable interrupt controllers, timer frequency, etc., are disabled during kernel configuration to avoid unwanted switching to the Linux domain.

Because the bootloader of the Raspberry Pi boards is not available as an open source, we decided to build the working environment based on a complete image available from the Raspberry repository (Raspberry Foundation, 2018). This includes the bootloader, a standard Linux kernel, and Raspbian Jessie as the root file system. Although Raspbian Jessie is available in both desktop and minimal versions, we have chosen the latter because the former uses GUI libraries that introduces high jitters during context switching. These were found during a stress test performed for an entire day; the results are not within the scope of this paper and are thus omitted.

To satisfy software dependencies and eliminate problems such as unusable binaries, missing libraries, and build errors, a suitable toolchain was used that supports the RPi3 environment. We used a toolchain called gcc-linaro-arm-linux-gnueabihf-raspbian-4.8.3, which is also available in (Raspberry Foundation, 2018). The ipipe-core-4.1.18-arm-4 is the suitable version of ADEOS for both kernels available with the Xenomai 3.0.2 package (Xenomai Project, 2018). This patch does not originally contain the information for BCM2710, which is the SoC used by RPi3. An additional patch is required to add the necessary contents that are available in (Blaess, 2018). Aside from the features that were disabled as mentioned above, RTIPC drivers that are located under the Xenomai kernel configuration were enabled such that the XDDP can be used. In comparison to the ROS installation under UbuntuMATE 16.04 that directly fetches compiled packages from the Ubuntu repository, the Raspbian implementation requires the sources to be downloaded directly from the ROS repository, called the build farm (ROS Build Farm, 2018), and requires them to be compiled separately, slightly increasing the development time. During installation, a lack of swap memory would lead to a kernel panic, thus extending the swap space from the default 100 MB to the recommended 2 GB.

3.2. Interfacing M4K mobile robot to ROS and Xenomai

The experiments are focused on the control of the mobile base of M4K that requires handling of different sensors such as an ultrasonic distance sensor (sonar), LED strip, laser range finder (LRF), and inertial measurement unit (IMU). These sensors are used to enable the M4K to navigate freely within an environment and avoid collisions with obstacles. The actuators of the mobile robot are also connected directly to the RPi3, which consist of two DC motors equipped with absolute encoders. As RPi3 only has one PWM port (used in LED), DAC modules were used and are connected to the SPI interface to control the motors. GPIO pins are configured and extended to act as SPI chip selections to accommodate the absolute encoders.

To maximize the performance of RPi3 while building a map during navigation (Aagela et al., 2017), the ROS package for navigation called *move_base* is deployed in a desktop (denoted as PC) computer while the RT tasks to handle the sensors and actuators are deployed locally in the RPi3, as shown in the deployment diagram in Fig. 4. In this configuration, the local planner deployed in the PC sends velocity commands every 50 ms to the RPi3 using the standard ROS communication protocol based on TCP/IP. In the RPi3, two NRT ROS nodes (blue boxes) are deployed running alongside Xenomai RT tasks (gray boxes).

The M4K node is the interface that is exposed to the ROS core enabling the usage of original ROS functions, connected using XDDP, in an RT task with a control period of 10 ms. The RT task handles the control algorithm that converts velocity commands to DAC and sends the generated values to the M4K through another XDDP port, connected to the device driver. Another node (URG Node) that produces global and local maps is connected directly to the navigation stack to generate the global and local maps. RT tasks that handle the sensors are connected to NRT tasks (white boxes), which execute the functions of the corresponding standard Linux device driver (green boxes). The actual navigation of M4K using the standard Linux device drivers is realized in a multitasking environment with priority-based scheduling provided by the proposed architecture.

The purpose of this experiment is to show the feasibility of the proposed architecture; thus, the performance measurements are acquired in a minimal configuration by operating each device driver independently. To validate the feasibility of the suggested architecture, we have developed standard Linux device drivers for each sensor and actuators.

Two GPIO pins were used for the ultrasonic sensor that serves as the trigger and the echo, with their respective ISR. According to the datasheet of the sensor that we used, US-100, the trigger should remain for a minimum of 10 µs in a high signal to initiate one cycle of reading. This requirement was realized in the trigger ISR using a call to *udelay*(). An echo pin was configured to capture both the rising and falling edges to calculate the total receive time of the pulse. In developing this device driver, we have implemented the *sysfs* structure (Negus, 2015) to easily generate device files for a limitless number of devices, depending on the number of available GPIO pins.

We used an LED strip (model NS-LED-02) from a local manufacturer. The LED is controlled through a sequence of 24-bit command comprising the RGB and brightness values using the hardware PWM of RPi3. As the PWM port of the RPi3 is used by the audio driver in default settings, we have blacklisted the ALSA module before the implementation of the device driver. We designed the driver with a clock generator to match the required timing sequence for each color value of the LED pixel.

The I2C interface was utilized to acquire data from myAHRS+, an IMU manufactured by WithRobot (myAHRS+ – WITHROBOT 2018). Because I2C is a platform device driver automatically enabled by the Linux kernel, we developed an application interface to enable the communication between the IMU and the primary board, check the current state of operation, calibrate the sensor, and interpret the data (raw, Euler, and Quaternion) from the IMU.

Hokuyo URG-04LX-UG01 is an immensely popular model of an LRF, which is used by many researchers. It is equipped with a microcontroller to implement an abstract control model. Thus, an existing Linux driver termed *cdc_acm* is already available and does not require any alteration. Further, we used the available API from the Hokuyo repository to interpret the data from the LRF.

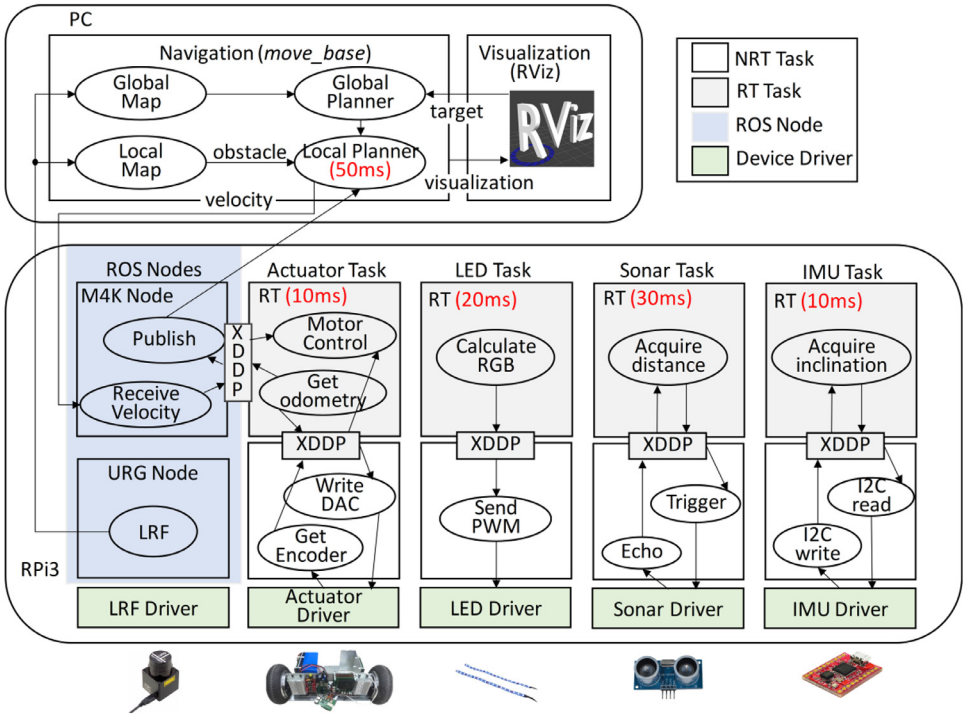


Fig. 4. Deployment diagram of the real-time control architecture to control M4K using ROS packages.

Finally, the drivers for the actuators are implemented using the SPI interface and APIs available in the standard Linux kernel. A 10-bit DAC is connected to each of the motors to emulate a PWM pulse using SPI write messages. Data from the 12-bit absolute encoders were read from the SPI datagrams and were converted into the local position of the M4K, on which the control algorithm to drive the robot is implemented.

We tested the developed device drivers in the standard Linux domain first, before applying them to Xenomai RT tasks to debug any problems that may occur that could affect the experimental results. Using the standard Linux drivers inside an RT task introduces mode switching. An RT task should always avoid any mode switch, especially inside the task loop. Mode switching is partly realized by a gatekeeper process that is operating in the standard Linux domain with a response time of approximately 50 ms per mode switch. During a mode switch, Xenomai suspends the RT task operating in the Xenomai scheduler and sends a virtual interrupt to the standard Linux through the ADEOS. The Xenomai operates in an idle mode, which awakens the standard Linux scheduler. When the standard Linux kernel receives the virtual interrupt and interrupt request handler calls, the RT task is queued into the standard Linux schedule, causing the RT task to lose its real-time capabilities.

The detection of unwanted mode switches is realized in various methods (Detection of Mode Switching – Xenomai, 2018). Xenomai was designed to excite the signal SIGXCPU in an event of a mode switch; thus, catching this signal by registering a signal hook inside a user program and by supplying a signal handler that can display the back trace of the stack can detect the source of a mode switch. Another method is to use the Xenomai utility called *slackspot*.

This indicates the code locations that cause transition to the secondary mode by parsing the virtual file output to display the program *backtrace* with no external log file involved. The easiest method to detect any mode switch is by displaying the Xenomai process statistics file located at /proc/xenomai/sched/stat. This is the procedure that we have applied to detect the occurrence of

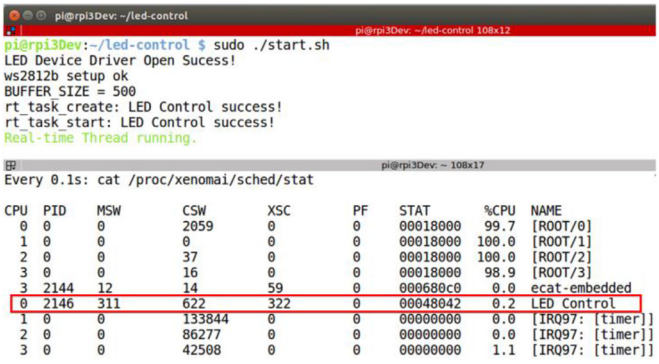


Fig. 5. Detection of mode switching for the "LED Control" RT task.

mode switches. Fig. 5 shows the statistics of the Xenomai RT task, LED_Control, highlighted by the red box. The task is created with the highest priority with a period of 20 ms. In this example, a Linux device driver function that sends color commands to an LED bar using PWM was directly called inside the task loop. As shown in the figure, 311 instances of mode switching occurred in 5 s of runtime.

This is indicated under the column labeled as MSW, which stands for mode switching. No conspicuous effects were visible during the mode switch in this scenario because only a lone RT task is running. In a multitasking case, the pre-emption of a high priority RT task by a lower priority task will result in data concurrency problems, which is chaotic in RT applications.

For example, in mobile robot control, a single data loss can lead to devastating accidents such as collisions with obstacles. Another problem that we have encountered owing to mode switching is a "system freeze" that occurs when directly accessing the sonar sensor device driver from an RT task, as shown in Fig. 6(a). In this figure, the device driver functions to trigger and receive distance information are called directly inside the loop of the RT task

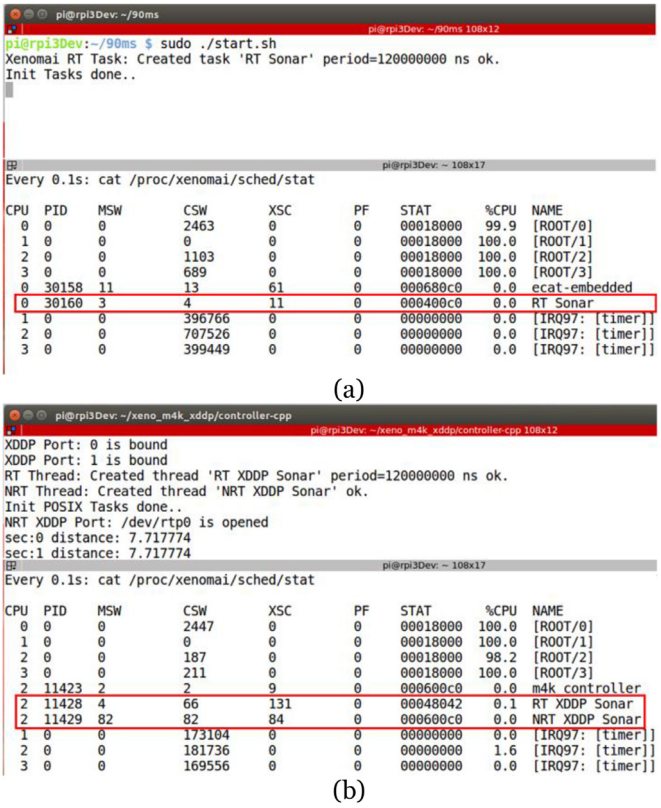


Fig. 6. Implementation of sonar device driver inside Xenomai RT tasks using (a) direct access, and (b) XDDP.

termed “RT Sonar”. After starting the process, the system abruptly freezes when the RT task accesses the device driver and did not respond to any CPU signals that we have assigned such as SIGKILL and SIGTERM using keyboard shortcuts.

Eliminating mode switching by the implementation of the proposed approach using XDDP solved this problem. This is illustrated in the proper operation of the sensor in Fig. 6(b). RT and NRT tasks were created to acquire data from the sonar sensor. The RT task sends a character to an NRT task that initiates the sensor trigger. The distance measurement is received by the NRT task and is sent back to the RT task using the XDDP port. In the figure, the RT task termed “RT XDDP Sonar”, receives the distance information from the NRT task termed “NRT XDDP Sonar” at a cyclic period of 120 ms and displays the feedback to the terminal every one second.

Four instances of mode switching occurred due to XDDP socket initialization and binding, which are executed before the RT task enters the infinite loop. The increased amount of context switches (CSW), which is 66 and 82, respectively for the RT and NRT tasks is a proof that a system freeze does not occur. From this example, we conclude that using the proposed real-time architecture can solve the system freeze that occurs when using standard Linux device drivers inside Xenomai tasks.

Performance of various RT mechanisms offered by Xenomai for synchronization of multiple RT tasks is reported in (Shin and Choi, 2017). The same method is performed analyzing the performance of XDDP in terms of the periodicity of the RT tasks during data exchange with NRT tasks. For each device driver, the RT tasks were assigned the highest priority level of 99 in accordance to the specifications of the Xenomai POSIX skin. Using the functions explained in Section 2, we created RT and NRT tasks for each device driver, represented by the green boxes in Fig. 4.

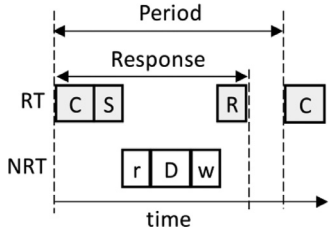


Fig. 7. Timing diagram of the real-time task accessing device drivers through XDDP.

NRT tasks execute the functions that were written in the standard Linux device drivers (e.g., send color to LED through PWM) while RT tasks generate the data required to be written (e.g., color commands). The RT and NRT tasks were connected through an XDDP port, where each sensor uses their own XDDP for data exchange. The RT tasks are configured to execute periodically in a cycle time of 30, 200, 20 ms for the sonar sensor, LRF, and LED, respectively. The actuator and IMU tasks were both set to a cycle time of 10 ms. The results were gathered and processed offline to calculate the statistical average, maximum, minimum, and standard deviation (St.D) values of each timing metric. Two probing points are configured for the timing analysis of the RT tasks. The first one is located at the start of the cyclic task. The second is placed at the end of the last executed command which is subtracted from the first probe to measure the overall time duration of executing all the commands; thus, it is called the response time.

Before the end of the cyclic task, the value of the first probe is stored into a different variable, which is subtracted in the next cycle to calculate the periodicity of the cyclic task. Fig. 7 shows the timing diagram of the RT task accessing a device driver through an XDDP port. For each cycle, the task executes a sequence of calculation, transmission of output data, and reception of input data from the device driver. The RT task calculates commands for the device driver represented by the C block. The commands are sent to the NRT task using *rt_xddp_send*() represented by S. The NRT task reads the XDDP port using *nrt_xddp_read*() (r). The commands are sent to the device driver and the corresponding feedback are acquired (D). Finally, the acquired data are sent back to the RT task using *nrt_xddp_write*() represented by w. From this flow of execution, we can define the response time as the sum of C, S, r, D, R, and w. These refer to the calculation time, the time taken to send the data from the RT task to NRT and vice versa, the time taken to receive data from the XDDP from either the RT or NRT task, and the time taken to execute device driver functions. Using this approach, no mode switching should occur.

Table 1 shows the results of the timing analysis for each device when the RT tasks access the respective device drivers through the XDDP ports. From these results, we found that the periodicity of each task is exceptional as no response time is greater than the expected period. Moreover, the St.D results are exceptionally low for all the tasks, meaning that the deadline is met most of the time which means that all the tasks are schedulable.

We performed the utilization bound (UB) test, which states that a set of tasks is schedulable if all the tasks were able to meet their respective deadlines and if the overall CPU utilization is less than the Liu and Leyland bound (Lundberg, 2002).

Accordingly, the deadline of each task is assumed to be equal to the expected period for each task. The tasks were scheduled with the task having the shortest deadline obtaining the highest priority. As the actuator and IMU tasks are running with the same period, we have configured the former to have higher priority than the latter. Thus, the priority order is the actuator, IMU, LED, sonar sensor, and LRF. We have considered the maximum response time as the worst-case in these results.

Table 1
Timing analysis results for each device driver.

Device	Range	Period (ms)	Response (µs)
Sonar	Ave	30.000	326.469
	Max	30.001	460.531
	Min	29.989	289.354
	St. d.	0.009	14.233
LRF	Ave	200.000	754.528
	Max	200.001	762.443
	Min	199.989	689.886
	St. d.	0.013	17.392
LED	Ave	20.000	586.469
	Max	20.003	590.521
	Min	19.951	575.469
	St. d.	0.047	24.358
IMU	Ave	10.000	316.627
	Max	10.001	432.194
	Min	9.891	303.187
	St. d.	0.115	16.785
Actuator	Ave	10.000	21.478
	Max	10.004	29.063
	Min	9.993	20.677
	St. d.	0.006	0.684

Table 2
Specifications of the M4K mobile robot.

Robot Part	Specification	Details
Body	External size	$45 \times 28 \times 20$ cm
	Weight	30 kg
Locomotive	Movement type	Differential drive
Section	Speed	200 cm/s
	Acceleration	37 cm/s^2
	Gear reduction	12:1
Wheels	Diameter	20.32 cm
	Width	5 cm

The expected CPU utilization for a task set that includes five RT tasks should not exceed 74.3%. If the CPU utilization is more than this bound, the set of tasks is considered non-schedulable and one or all the tasks can miss their respective deadlines. The results show that the CPU utilization for each task is 1.53%, 0.38%, 2.95%, 4.32%, and 0.29%, respectively. The total CPU utilization when operating the five tasks simultaneously in a multitasking environment is 9.47%. Thus, all the tasks executed inside the proposed RT architecture are schedulable when using standard Linux device drivers.

4. Real-time control application for navigation of M4K

Using the proposed architecture, we have implemented a navigation scheme for the M4K using the ROS package, called the navigation stack. The purpose of this experiment is to validate the RT performance of the proposed architecture when using ROS packages that are integrable to more complex systems. The navigation stack consists of a global planner and local planner to navigate a mobile robot inside an environment. Among the path planning and trajectory generation algorithms within the ROS, we have selected the default packages: standard and *base_local_planner* for the global and local planners, respectively. The *base_local_planner* was tuned to accommodate the M4K using its kinematics, as shown in Table 2.

Although the specifications specify 2 m/s as the maximum velocity of the M4K, we have performed the experiments at 0.25 m/s to prevent any accidents that could occur because of slip. The local planner requires the minimum velocity that could actuate the mobile robot as a parameter. For the M4K, we have configured this value as 0 m/s, considering that the mobile robot is actuated in a cyclic velocity mode. The acceleration limits are configured according to the specification. Because the M4K is a two-wheeled differential drive mobile robot, the center velocity commands from the local planner is converted to the joint space velocities considering the gear ratio, radius of the wheels, and the length between each of the M4K wheels. Further, the robot footprint of the M4K is constructed using its external size and weight to design a virtual robot model that emulates the physical characteristics of the M4K implemented on a monitoring program using *RViz*, a robot visualization tool provided by the ROS.

The actual experiment was conducted in an environment surrounded by white panel boards, as shown in Fig. 8. On the left side of the figure, the M4K was settled in a starting point within the map. The robot was given a target position from ROS to traverse 3 m toward the triangular-shaped obstacle located at approximately 1.5 m from the starting point. On the right side of the figure, the monitoring and command tool based on *RViz* is shown. The red points represent the footprint of the mobile robot, and the black dots represent the obstacles detected by the LRF.

From the software development diagram shown in Fig. 4, the navigation stack produces center velocity commands every 50 ms. These are received by the M4K node connected to the RT task, M4K actuator, through an XDDP port. This task is responsible for the motor control algorithm that generates joint space velocities for the M4K. Another XDDP port is configured to communicate with the NRT task that handles the device driver in sending the calculated joint space velocities and acquiring the feedback from the encoders. The encoder values are calculated for the current pose of the M4K and these values are sent back to the ROS navigation stack through the first XDDP port. The RT actuator task is set periodically with a cycle time of 10 ms to meet the sampling time requirement of the motor controller. The calculated position of the M4K is sent to the navigation stack every 50 ms according to the limitations of the ROS package.

Fig. 9 shows the trajectory of the M4K using the proposed RT architecture for navigation. In the figure, the M4K was commanded to traverse a straight line 3 m forward along the *x*-axis while avoiding collision with the obstacle. Fig. 9(a) shows the actual trajectory acquired from the encoder values of the M4K represented in Cartesian space. The velocity commands generated by the ROS *base_local_planner* to exhibit such motions are shown in Fig. 9(b). In this figure, the black line denoted as the reference is the center velocity generated by the ROS navigation stack every 50 ms. The dotted blue line denoted by M4K represents the calculated feedback velocity from the M4K encoder. The result shows that although the M4K was able to avoid a collision with the obstacle, it was not able to arrive exactly at the desired target position.

This is due to the different parameters of the ROS package such as goal tolerance, simulation time, and trajectory scoring parameters. The improvement in the local planner is required to reach the final goal successfully; nevertheless, this is outside the scope of this paper. To demonstrate the improvements of the real-time performance using the proposed architecture, we have compared the periodicity of the actuator task with the classical method of using only the ROS software running in the standard Linux kernel to navigate the M4K.

In this condition, the motor control task and device driver functions are implemented inside the M4K node that directly sends joint-space velocity commands to the M4K. The M4K is assigned the same commands to traverse 3 m from the starting position while avoiding a static obstacle. The results of the comparison are shown in Fig. 10. In this figure, navigation using the proposed architecture is denoted as XDDP, while ROS represents the results of the implementation in the standard Linux.

The measurement is focused on the actuator task which has the highest priority. The task handles calculation of joint-space velocities from the velocity commands generated by ROS and communication with the device drivers of motors and encoders. The

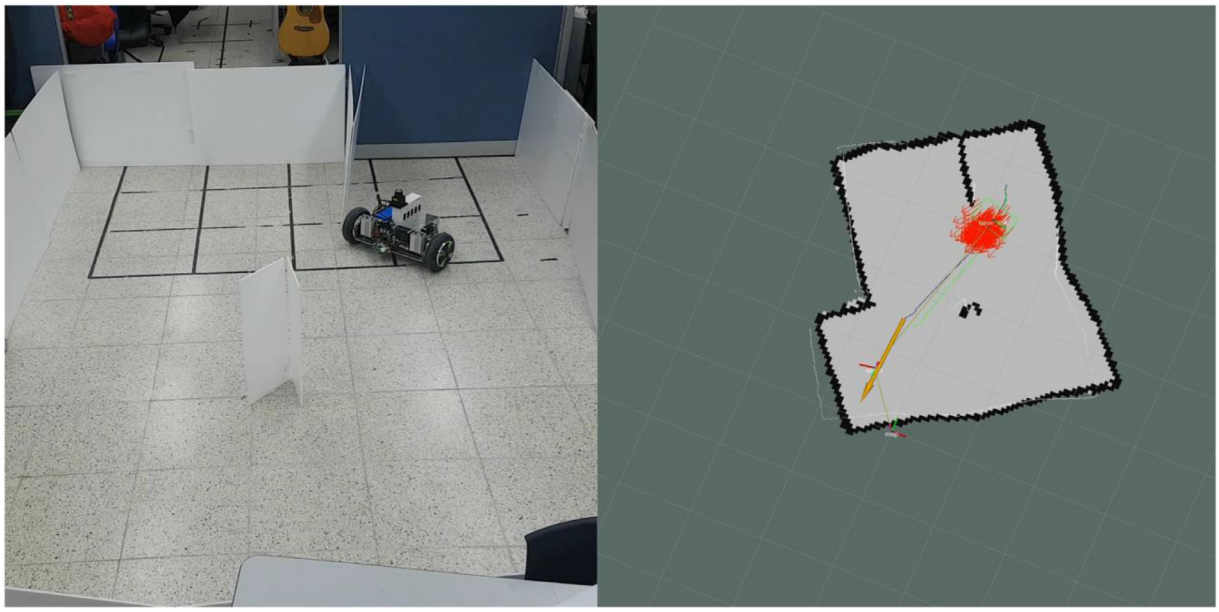


Fig. 8. Actual experiment environment.

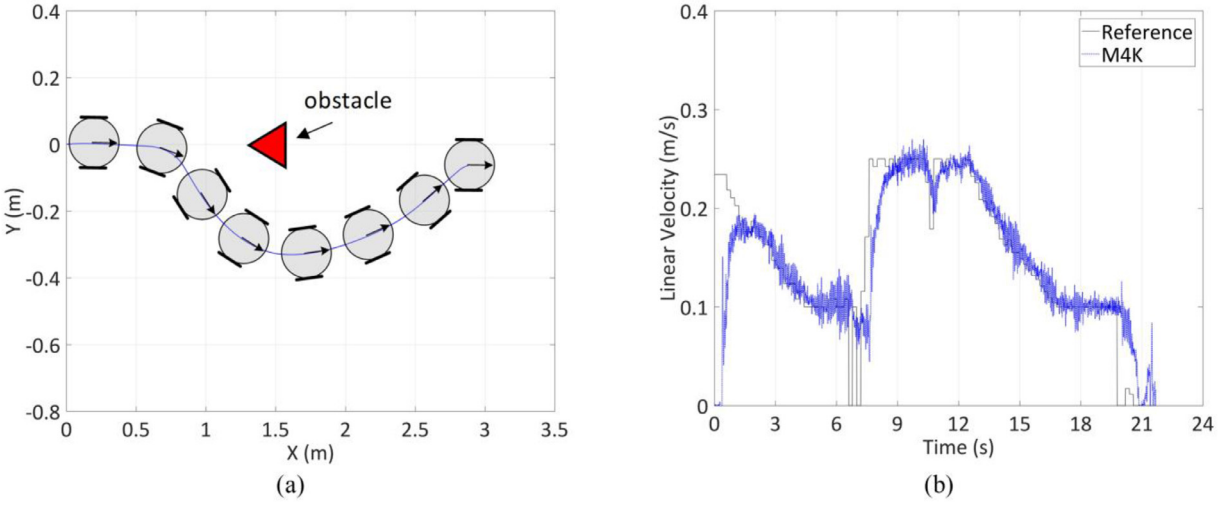


Fig. 9. Trajectory of M4K using the proposed real-time control architecture in (a) the Cartesian space and (b) center velocity space.

experiment is performed in normal condition without stress and when the system is under full stress using the tool Stress-ng (Ubuntu, 2018). The tool is used in assigning excessive load to both the CPU and the memory. ROS implemented in the standard Linux has the worst performance when under stress (ROS-STRESS) with 362 µs, 4782.816 µs, 0.007 µs, and 4420.113 µs for the average, maximum, minimum, and St.D, respectively. In normal condition (ROS), the results were 40.533 µs, 1814.054 µs, 0.007 µs, and 173.527 µs respectively for the average, maximum, minimum, and St.D. The calculated jitter shows that the actuator task could not meet the predefined deadline of 10 ms due to the lack of hard RT support, even in normal condition.

The performance greatly improved when running ROS in the proposed architecture with XDDP. The results exhibit periodic behavior and minimal deviations from the expected cycle. When under stress (XDDP-STRES), the jitter of the actuator task is 6.115 µs, 83.958 µs, 0 µs, and 5.599 µs for the average, maximum, minimum, and St.D, respectively. In normal condition (XDDP), the calculated jitter is 0.281 µs, 5.834 µs, 0 µs, and 0.4 µs respectively for the average, maximum, minimum, and St.D. The difference between the two methods is clearly shown in the distribution plot with the proposed architecture showing superior performance as expected.

These results are highly comparable with the performance evaluation conducted for ROS2 running under RT_PREEMPT in a report presented at ROSCON 2015 (Kay and Tsouroukdissian, 2018). Their system is running with a faster cycle time of 1 ms in comparison to the 10 ms requirement of our experimental platform. The comparison is focused on the jitter, because real-time does not necessarily mean fast or short cycle times, but rather how accurate can the system satisfy temporal constraints. The proposed real-time architecture has shown better performance in a stress-free environment. Lower values of the jitter were measured in all statistical metrics. In a stressed configuration, the RT_PREEMPT implementation produced a lower average of 3.729 µs, but the maximum jitter is 258.064 µs. The real-time performance is questionable when implemented on an embedded environment. There is a huge possibility that worse results would be acquired because of the difference in computing power.

The proposed real-time architecture has shown promising results in a simple robotic control application such as the naviga-

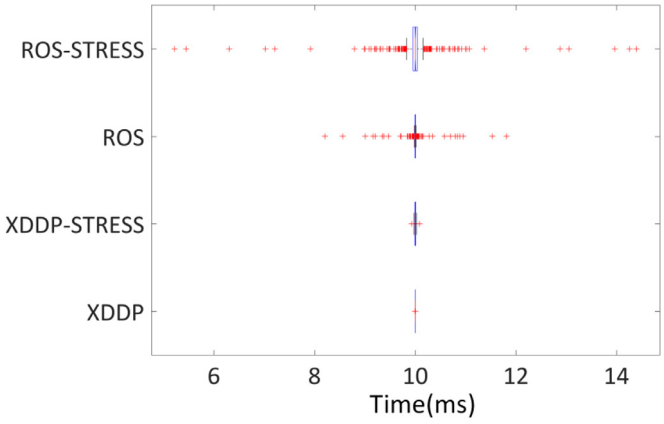


Fig. 10. Periodicity of the actuator task in various conditions during navigation.

tion of M4K. It also enhances robotic application design exploiting the rapid-development tools of ROS and its integration to a multitasking environment offered by Xenomai. This architecture can be the backbone for more complex real-time systems including humanoids (Muratore et al., 2017; Jung et al., 2018) and industrial distributed control systems (Lu et al., 2016).

5. Conclusion and discussion

This paper proposed an RT control architecture based on Xenomai to design RT robot applications using NRT ROS packages. An ROS is the most dominant robotic platform for the rapid development of robot applications; however, it is only designed to operate under the standard Linux kernel that does not provide RT guarantees. System calls from the standard Linux inside Xenomai tasks cause stability problems because of mode switching. Our proposed architecture adapts a communication interface of the XDDP to facilitate data exchange between RT and NRT tasks to solve this issue. Using this method, standard device drivers can be used inside RT tasks without having to develop RT device drivers that consumes significant amount of development time. Moreover, the proposed approach enables designing robot applications that withholds RTOS advantages such as multitasking, priority-based scheduling, and predictable response with the benefits of ROS tools and packages for rapid development.

We have comprehensively described the XDDP and developed simple APIs that emulate the Xenomai native functions to create RT tasks that can help users re-use or integrate the proposed architecture to more complicated systems. The feasibility of the approach is tested by the implementation on an embedded platform, RPi3. Experiments were conducted to measure the RT performance of the suggested architecture in terms of the periodicity when using standard Linux device drivers and ROS packages inside RT tasks.

Our experimental result shows that the proposed approach is essential in designing RT control applications to control a mobile robot, M4K, while using standard Linux device drivers and ROS packages. We have observed that mode switching can directly affect the system stability when using device drivers that can lead to a system freeze. It is noteworthy that the using the suggested architecture significantly improves the periodicity of the control task that handles the M4K actuation. These results are important especially in RT applications that require simultaneous executions of multiple tasks.

In particular, the proposed architecture is applicable to improve homogenous systems such as XBotCore (Muratore et al., 2017) and PODO (Jung et al., 2018), the software implemented to control WALKMAN and DRC-HUBO, respectively, in the DARPA robotic challenge. Instead of a single RT task governing the communication between hardware and other RT tasks, all the tasks are connected with their respective device driver to ensure priority based scheduling and deterministic response of the entire system.

Aiming for an RT control architecture using ROS packages for primary controllers based on low-cost embedded hardware, we have achieved the desired performance by deploying ROS packages that need extensive calculations in a PC. For future work, we will advance our research by considering the performance and the deployment of ROS packages inside the embedded hardware while conserving a high RT performance. Also, ROS2 is an innovative project to deal with the real-time issue of its predecessor. However, there is still limited source of documentation that verifies its validity for real-time control. For example, Maruyama et al. (2016) presented a performance evaluation only of the communication module. We will study the real-time performance of ROS2 with actual control workload and its implementation on other distributions of RTOSes such as Xenomai.

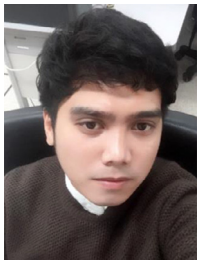
Acknowledgments

This work was supported by the Human Resources Development of the Korea Institute of Energy Technology Evaluation and Planning (KETEP) grant funded by the Korea government Ministry of Trade, Industry & Energy. (no. 20174030201840) and by the Global Frontier R&D Program on <Human-centered Interaction for Coexistence> funded by the National Research Foundation of Korea grant funded by the Korean Government (MSIP) (2010-0029759).

References

Aagela, H., Al-Nesf, M., Holmes, V., 2017. An Asus_xtion_probased indoor MAPPING using a Raspberry Pi with Turtlebot robot Turtlebot robot. In: 2017 23rd Int. Conf. Autom. Comput. IEEE, pp. 1–5.
Abbott, D., 2013. Linux for embedded and real-time applications. Newnes.
Ahmad, A., Babar, M.A., 2016. Software architectures for robotic systems: a systematic mapping study. J. Syst. Softw. 122, 16–39.
Alho, P., Mattila, J., 2015. Service-oriented approach to fault tolerance in CPSs. J. Syst. Softw. 105, 1–17.
Anistratov, P., Golobokov, Y., Pavlov, V., 2015. Hardware-software complex prototyping for the pulse power supply control system of Tokamak T-15. Procedia Comput. Sci. 546–555.
Beck, D., Brand, H., Karagiannis, C., Rauth, C., 2006. The first approach to object oriented programming for LabVIEW real-time targets. IEEE Trans. Nuclear Sci. 53, 930–935.
Blaess, C., 2018. Xenomai patch for Raspberry Pi 3. https://www.blaess.fr/christophe/files/article-2017-03-20/001-adapt-4.1.18-patch-to-rpi-4.1.21-kernel.patch (accessed April 10, 2018).
Bouchier, P., 2013. Embedded ROS. IEEE Robot. Autom. Mag. 20, 17–19.
Brown, J.H., Martin, B., 2018. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. https://pdfs.semanticscholar.org/9eb5/1dbe38fb23034e80b8664d8281996d2a5ef6.pdf (accessed September 17, 2018).
Cardellino, A., Ruzzenenti, A., Natale, L., 2018. Design and implementation of a YARP device driver interface: the depth-sensor case. Front. Robot. AI 5, 1–6.
Cereia, M., Bertolotti, I.C., Scanzio, S., 2011. Performance of a real-time EtherCAT master under Linux. IEEE Trans. Ind. Inf. 7, 679–687.
Chianese, A., Picciali, F., Riccio, G., 2015. Designing a Smart Multisensor Framework based on Beaglebone Black Board. In: Designing a Smart Multisensor Framework based on Beaglebone Black Board. Springer, Berlin, Heidelberg, pp. 391–397.
Chitta, S., Marder-Eppstein, E., Meeussen, W., Pradeep, V., Tsouroukdissian, A.R., Bohren, J., Coleman, D., Magyar, B., Raiola, G., Lüdtke, M., Fernandez Perdomo, E., 2017. ros_control: a generic and simple control framework for ROS. J. Open Source Softw. 1, 456.
Choi, B.W., Shin, D.G., Park, J.H., Yi, S.Y., Gerald, S., 2009. Real-time control architecture using Xenomai for intelligent service robots in USN environments. Intell. Serv. Robot. 2, 139–151.
Dantam, N.T., Lofaro, D.M., Hereid, A., Oh, P.Y., Ames, A.D., Stilman, M., 2015. The ach library: a new framework for real-time communication. IEEE Robot. Autom. Mag. 22, 76–85.
De Oliveira, D.B., De Oliveira, R.S., 2016. Timing analysis of the PREEMPT RT Linux kernel. Softw. Pract. Exp. 46, 789–819.
Detection of Mode Switching – Xenomai, 2018. https://xenomai.org//2014/06/finding-spurious-relaxes/ (accessed April 10, 2018).
Erwinski, K., Paprocki, M., Grzesiak, L.M., Karwowski, K., Wawrzak, A., 2013. Application of Ethernet Powerlink for communication in a Linux RTAI open CNC system. IEEE Trans. Ind. Electron. 60, 628–636.

Ferdoush, S., Li, X., 2014. Wireless sensor network system design using raspberry Pi and Arduino for environmental monitoring applications. Procedia Comput. Sci. 34, 103–110.
Fischmeister, S., Lam, P., 2010. Time-aware instrumentation of embedded software. IEEE Trans. Ind. Inf. 6, 652–663.
Gobillot, N., Lesire, C., Doose, D., 2018. A design analysis methodology for component-based real-time architectures of autonomous systmes. J. Intell. Rob. Syst.
Gosewehr, F., Wermann, M., Colombo, A.W., 2016. From RTAI to RT-Preempt a quantative approach in replacing Linux based dual kernel real-time operating systems with Linux RT-Preempt in distributed real-time networks for educational ICT systems. In: IECON 2016 - 42nd Annu. Conf. IEEE Ind. Electron. Soc. IEEE, pp. 6596–6601.
Grepl, R., 2011. Real-time control prototyping in MATLAB/Simulink: review of tools for research and education in mechatronics. In: 2011 IEEE Int. Conf. Mechatronics. IEEE, pp. 881–886.
Hasegawa, R., Yawata, N., Ando, N., Nishio, N., Azumi, T., 2016. RTM-TECS: collaboration framework for robot technology middleware and embedded component system. In: 2016 IEEE 19th Int. Symp. Real-Time Distrib. Comput. IEEE, pp. 212–220.
Henriksson, D., Cervin, A., Årzén, K.-E., 2002. Truetime: simulation of control loops under shared computer resources. IFAC Proc. 35, 417–422.
Honegger, D., Meier, L., Tanskanen, P., Pollefeys, M., 2013. An open source and open hardware embedded metric optical flow CMOS camera for indoor and outdoor applications. In: Proc. IEEE Int. Conf. Robot. Autom. IEEE, pp. 1736–1741.
Jung, T., Lim, J., Bae, H., Lee, K.K., Joe, H.-M., Oh, J.-H., 2018. Development of the humanoid disaster response platform DRC-HUBO+. IEEE Trans. Robot. 34, 1–17.
Kaliński, K.J., Mazur, M., 2016. Optimal control at energy performance index of the mobile robots following dynamically created trajectories. Mechatronics. 37, 79–88.
Kay, J., Tsouroukdissian, A.R., 2018. Real-time control in ROS and ROS 2.0 n.d. https://roscon.ros.org/2015/presentations/RealtimeROS2.pdf (accessed September 18, 2018).
Kilamo, T., Hammouda, I., Mikkonen, T., Aaltonen, T., 2012. From proprietary to open source - growing an open source ecosystem. J. Syst. Softw. 85, 1467–1478.
Kiszka, J., 2005. The real-time driver model and first applications 7th Realt. Linux Work. Lille Fr. http://xenomai.org/documentation/xenomai-2.4/pdf/RTDM-and-Applications.pdf (accessed April 10, 2018).
Lee, H., Kim, Y.H., Lee, K.K., Yoon, D.K., You, B.J., 2016. Designing the Appearance of A Telepresence Robot, M4K: A Case Study. Springer, Cham, pp. 33–43. (Including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics).
Li, J., Pilkington, N.T., Xie, F., Liu, Q., 2010. Embedded architecture description language. J. Syst. Softw. 83, 235–252.
Li, Z., Deng, J., Lu, R., Xu, Y., Bai, J., Su, C.-Y., 2016. Trajectory-tracking control of mobile robot systems incorporating neural-dynamic optimized model predictive approach. IEEE Trans. Syst. Man Cybern. Syst. 46, 740–749.
Lu, B., Wu, X., Figueroa, H., Monti, A., 2007. A low-cost real-time hardware-in-the-loop testing approach of power electronics controls. IEEE Trans. Ind. Electron. 54, 919–931.
Lu, C., Saifullah, A., Li, B., Sha, M., Gonzalez, H., Gunatilaka, D., Wu, C., Nie, L., Chen, Y., 2016. Real-time wireless sensor-actuator networks for industrial cyber-physical systems. Proc. IEEE 104, 1013–1024.
Lundberg, L., 2002. Utilization based schedulability bounds for age constraint process sets in real-time systems. Real-Time Syst. 23, 273–295.
Maruyama, Y., Kato, S., Azumi, T., 2016. Exploring the performance of ROS2. In: Proc. 13th Int. Conf. Embed. Softw. - EMSOFT '16. ACM Press, New York, New York, USA, pp. 1–10.
MathWorks, 2019. Simulink real-time. https://ww2.mathworks.cn/en/products/simulink-real-time.html (accessed January 21, 2019).
Muratore, L., Laurenzi, A., Hoffman, E.M., Rocchi, A., Caldwell, D.G., Tsagarakis, N.G., 2017. XBotCore: a real-time cross-robot software platform. In: 2017 First IEEE Int. Conf. Robot. Comput. IEEE, pp. 77–80.
myAHRS+ – WITHROBOT, 2018. http://withrobot.com/sensor/myahrsplus/ (accessed April 10, 2018).
Negus, C., 2015. Linux Bible. John Wiley & Sons. doi:10.1002/978111920953.
Omidvar, M.N., Yang, M., Mei, Y., Li, X., Yao, X., 2017. DG2: a faster and more accurate differential grouping for large-scale black-box optimization. IEEE Trans. Evol. Comput. 21 1–1.
Open Source Robotics Foundation, 2018. ROS Kinetic Kame (n.d.). http://wiki.ros.org/kinetic (accessed September 18, 2018).
Paschali, M.-E., Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A., Stamelos, I., 2017. Reusability of open source software across domains: a case study. J. Syst. Softw. 134, 211–227.
Raspberry Foundation, 2018 Raspberry pi. https://github.com/raspberrypi/ (accessed April 10, 2018).
ROS Build Farm, 2018. http://build.ros.org/ (accessed April 17, 2018).
Shin, U.C., Choi, B.W., 2017. Performance evaluation of real-time mechanisms on open embedded hardware platforms. J. Inst. Control. Rob. Syst. 23, 60–66.
Ubuntu, K., 2018 Stress-ng. http://kernel.ubuntu.com/~cking/stress-ng/ (accessed September 18, 2018).
Vogel-Heuser, B., Fay, A., Schaefer, I., Tichy, M., 2015. Evolution of software in automated production systems: challenges and research directions. J. Syst. Softw. 110, 54–84.
Wei, H., Shao, Z.Z., Huang, Z., Chen, R., Guan, Y., Tan, J., Shao, Z.Z., 2016. RT-ROS: a real-time ROS architecture on multi-core processors. Futur. Gener. Comput. Syst. 56, 171–178.
Xenomai Project, 2018 Xenomai Repository. https://xenomai.org/downloads/xenomai/stable/ (accessed September 17, 2018).
Yang, G.J., Delgado, R., Choi, B.W., 2016. A practical joint-space trajectory generation method based on convolution in real-time control. Int. J. Adv. Robot. Syst. 13.
Zappulla, R., Virgili-Llop, J., Zagaris, C., Park, H., Romano, M., 2017. Dynamic air-bearing hardware-in-the-loop testbed to experimentally evaluate autonomous spacecraft proximity maneuvers. J. Spacecr. Rockets 54, 825–839.
Zhang, L., Slaets, P., Bruyninckx, H., 2012. An open embedded hardware and software architecture applied to industrial robot control. In: 2012 IEEE Int. Conf. Mechatronics Autom. ICMA 2012. IEEE, pp. 1822–1828.
Zucker, M., Joo, S., Grey, M.X., Rasmussen, C., Huang, E., Stilman, M., Bobick, A., 2015. A general-purpose system for teleoperation of the DRC-HUBO humanoid robot. J. Fields Rob. 32, 336–351.



Raimarius Delgado was born in Makati, Philippines in 1990. He received the B.S. and M.S. degrees in Electrical and Information Engineering from Seoul National University of Science and Technology, Seoul, Korea, in 2014 and 2016, respectively. He is currently working toward the Ph.D. degree at the same university under the supervision of Prof. Byoung Wook Choi. His research interests include real-time systems, industrial control and automation, and service robotics. He is a student member of IEEE.

Bum-Jae You received the B.S. degree in control and instrumentation engineering from Seoul National University, Seoul, South Korea, in 1985, and the M.S. and Ph.D. degrees in electrical and electronic engineering from the Korea Advanced Institute of Science and Technology, Seoul, South Korea, in 1987 and 1991, respectively. From 1991 to 1994, he was with Turbo-Tek Co., Ltd., South Korea, as the Head of the Robotics Division, and then he joined the Korea Institute of Science and Technology (KIST), Seoul, South Korea, in 1994. From 2004 to 2011, he was the Director of the Center for Cognitive Robotics Research. He is currently a principal research scientist in the center for intelligent robotics, KIST and the director of the Center of Human-Centered Interaction for Coexistence from 2010. His research interests include human-centered interaction, vision-based robotics, network-based humanoid robots, and embedded system applications.

Byoung Wook Choi received the M.S. and Ph.D. degrees in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST), Seoul, Korea in 1988 and 1992, respectively. He is currently a professor in the Department of Electrical and Information Engineering at Seoul National University of Science and Technology. Previously, he was a principal research engineer in LG from 1992-2000 and a professor in Sun Moon University from 2000-2005. He was the CEO of Embedded Web Co., Ltd. from 2001-2003. Also, he was a Senior Fellow in Nanyang Technological University, Singapore from 2007-2008. Prof. Choi has published textbooks on Embedded Linux. His current research interests include real-time systems design, embedded systems, and intelligent robot software.