Implementation of a network-based Real-Time Embedded Linux platform

Byoung-Wook Choi*, and Eun-Cheol Shin**, Ho-Gil Lee**

* Department of Electrical Engineering, Seoul National University of Technology, Seoul, Korea
(Tel : +82-2-970-6412; E-mail: bwchoi@snut.ac.kr)
**Korea Institute of Industrial Technology, Ansan, Korea
(Tel : +82-31-400-3991; E-mail: unchol@kitech.re.kr)
**Korea Institute of Industrial Technology, Ansan, Korea
(Tel : +82-31-400-3991; E-mail: leehg@kitech.re.kr)

Abstract: The SoC and digital technology development recently enabled the emergence of information devices and control devices because the SoC present many advantages such as lower power consumption, greater reliability, and lower cost. It is required to use an embedded operating system for building control systems. So far, the Real-Time operating system is widely used to implement a Real-Time system since it meets developer's requirements. However, Real-Time operating systems reveal a lack of standards, expensive development, and license costs. Embedded Linux is able to overcome these disadvantages. In this paper, the implementation of control system platform using Real-Time Embedded Linux is described. As a control system platform, we use XScale of a Soc and build Real-Time control platform using RTAI and Real-Time device driver. Finally, we address the feasibility study of the Real-Time Embedded Linux as a Real-Time operating system for mobile robots.

Keywords: Real-Time system, Real-Time device driver, RTAI, Embedded Systems

1. INTRODUCTION

Embedded systems such as information appliances and control devices are being small-sized and light-weighted while providing various high performances. These devices usually employ the SoC (System on chip) in which MPU (Micro Processing Unit), memory, DSP (Digital Signal Processor) and other devices are included. Because SoC is developed on a single semiconductor chip, it is low power consumption and very reliable because of using functionally certified IP (Intellectual Property). And one can reduce development cost. However, its development requires great amount of effort and becomes time consuming, because it is integrated with one chip embedding complex functions. [1] To cope with the disadvantage, Embedded Operating System is widely used. [2, 3, 4]

Real-Time Operating Systems have been largely used as an embedded operating system. Because it is an operating system that satisfies time limitation, it is being largely employed in the military-equipment filed, and recently in fields such as robots and automation systems. However, most RTOS (Real-Time Operating System) are commercially available so that the technology may be dependent on the RTOS vendors, and furthermore development cost is expensive. [5]

In this paper, the open source, Real-Time Embedded Linux, RTAI (Real-Time Application Interface) has been ported to Intel PXA255-based hardware to build a network-based control system platform. For RTAI applications, Boot Loader has been developed. And RTAI-patched Linux kernel should be ported to the SoC-based hardware by modifying hardware-dependent portions. And then, we have implemented the Real-Time device driver for peripheral devices and developed file system. Finally, in order to verify the feasibility of Real-Time Embedded Linux based control platform, we applied it to mobile robot. [6]

2. REAL-TIME OPERATING SYSTEM AND RTAI

RTOS is defined as the system guaranteeing response in accordance to the operation, in a given period of time. This does not necessarily mean the fast response system, but it represents the system providing the desired output using system functions with predictable response time. RTOS is a software to manage time to support multi-tasking programs, and provides inter task mechanism such as synchronization, semaphore and pipe, and scheduling mechanism to guarantee that important task run first. Also, recently, the integrated development environment is provided to reduce development time. Therefore, RTOS may be considered as a good building block to develop Real-Time systems.

RTOS, despite their advantages, is difficult to use and are very expensive, not to mention that sources are not open to the public in most cases, and needs to re-invest the development environment when changing the processor. And all services are provided by the single OS, and are limited to what is provided by eth vendor. Although Embedded Linux is being used to solve these problems, Embedded Linux does not support Real-Time systems. For these reasons, many researches have been taken to insure Real-Tim while also utilizing all advantages of the Embedded Linux. Especially RTLinux (Real-Time Linux) and RTAI has been widely used. [7]

RTAI is a hard Real-Time extension to the Linux kernel. The RTAI project is a Free Software project that was founded by the Department of Aerospace Engineering of Politecnico di Milano (DIAPM). It has evolved into a community project coordinated by Professor Paulo Mantegazza of DIAPM. RTAI is a sub-kernel that runs under Linux. It provides hare Real-Time response by running Linux as the idle task. Interrupts are intercepted by RTAI where they may be processed by a RTAI interrupt handler or passed up to Linux. RTAI supports various hardwares such as ARM, PowerPC, MIPS, and CRIS. Truthfully, RTAI is not a RTOS, but it rather an interface for Real-Time tasks. That is, an operating system is required to use RTAI. In this paper, Linux is used as the operating system to interface with RTAI. [8]

Figure 1 shows the basic structure of RTAI, and it can be known that dual kernels are being used. RTAI and Linux kernel are executed by HAL (Hardware Abstract Layer), and it may interface with the hardware. HAL, when executing RTAI and Linux kernel, primarily executes Real-Time tasks (RT_TASK) of RTAI, and then runs the Linux kernel in the lowest priority. That is, Real-Time tasks are first executed, and when these tasks are not in ready-to-run state, Linux kernel and processes of the Linux kernel are then executed. [9, 10]

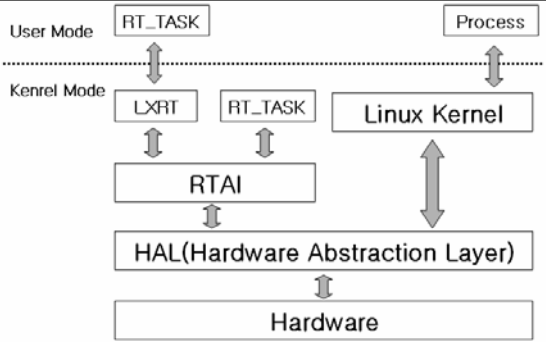

Fig. 1 RTAI Structure

3. REAL-TIME EMBEDDED LINUX PORTING & RTAI INSTALLATION

3.1 Development Environment & Boot Loader

In this paper, RTAI has been applied to an Intel PXA255 based system board. PXA255 is a SoC of ARM10 core architecture, reaching at a maximum operating frequency of 400MHz, and includes memory controllers such as DMA controller, and MMU (Memory Management Unit). As peripherals devices, it includes USB Slave, UART, I2C, and LCD interface. The system board also includes SDRAM (64MB), Flash memory (32MB), 10/100Mbps Ethernet controller, USB Host, Touch Screen, and Color LCD.

To apply RTAI to the system board (or target system), the development environment must first be constructed. Unlike program development in PC (or Host), program development in embedded systems can not be directly developed in the system board, hence generally used in PC. Furthermore, the executable file developed in PC are downloaded to the system board and then executed. Figure 2 shows the general development environment of the embedded systems.

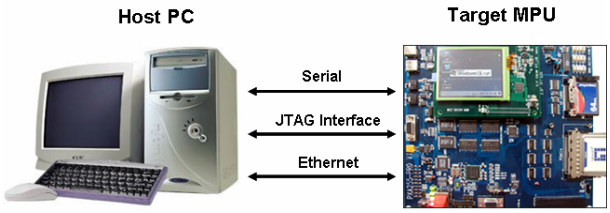

Fig. 2 Development Environment of the Embedded System

The serial port in the Fig.2 monitors the system board. In most embedded systems, monitors are not installed. Therefore, the state of system board is verified using the serial port just like a console terminal. The JTAG (Joint Test Access Group) interface downloads the boot loader, and when using JTAG emulator, it may also debug hardware in chip levels. Ethernet port is used to download kernel and file system images by using the boot loader. In addition, the cross tool-chain is required as a cross-development environment. This is because the processor used in PC and the target processor is different from each other. The sources required to build the cross tool-chain are follows.

- binutil : Assembler & loader, miscellaneous GNU tools
- glibc : Library for constructing cross compiler
- kernel : Kernel header source
- patch : Patch file for target processor

In each source, the patch for the target processor must be applied. Because binutil and gcc will be used in PC, a compiler for PC is used. On the order hand, glibc will be used for the application used in the target system so that a cross-compiler for target processor is applied for building images.

The boot loader is a program initializing the hardware, and performs functions such as Interrupt Vector and Memory Map setup. In this paper, the boot loader called Blob (Boot Loader Object) has been used. Blob is a boot loader supporting StrongARM and XScale (PXA250 and PXA255). Aside from hardware initialization, it includes the function of downloading kernel and file system images through Ethernet and storing them in Flash memory. To apply Blob to the system board, the memory map and system register, the UART and Ethernet device drivers need to be modified. This is because each system has a different memory map and peripheral devices. The modified boot loader is converted into a binary image file using the cross compiler and downloaded and stored to the 0x00 address in the Flash memory using JTAG interface. Fig. 3 shows the booting process of boot loader executed in the system board.

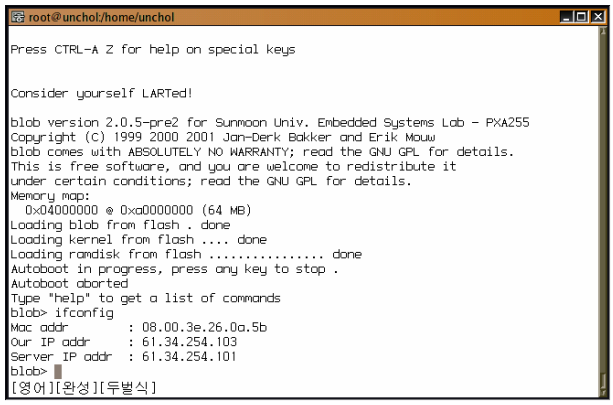

Fig. 3 Booting sequence of boot loader

3.2 Real-Time Embedded Linux Porting

Linux kernel is an application program placed on the hardware level and is divided into the hardware-dependent and the hardware-independent part. Generally, Linux porting represents the modification of processor-dependent functions, in which boot code porting, memory map initialization, system register initialization, and developing device driver are performed. The kernel porting patched with RTAI is also performed in the same process. The following shows the process of porting the RTAI patched kernel to the system board.

1) Apply ARM patch and RTAI patch to the Linux kernel. Because the system board used ARM core based SoC, ARM patch must also be applied together. RTAI patch usually modifies the timer and interrupt-related programs.

2) Modify system register, memory map, interrupt and others according to the configuration of the system board.

3) Set the Linux kernel according to the target hardware, and compile it. When setting the kernel, the

setup shown in Fig. 4 must be done so that RTAI HAL can be used.

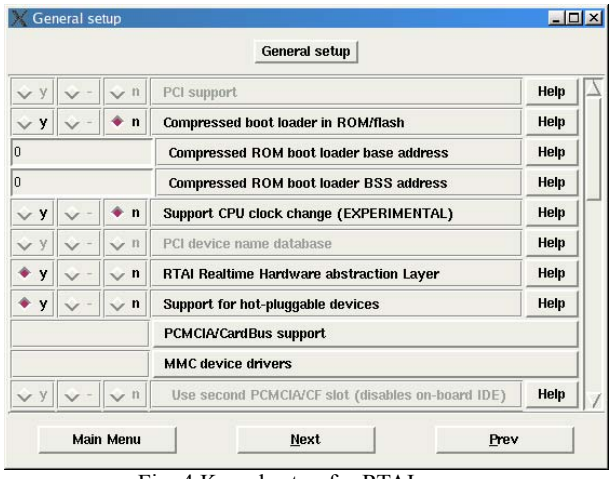

Fig. 4 Kernel setup for RTAI usage

4) Using the boot loader, download the compiled kernel image to the system board and burn it into the Flash memory

The core of the Linux operating system is known as the kernel. When an Embedded Linux system boots after doing the Linux kernel porting, the kernel is loaded into memory from a device that an embedded system's boot monitor can access, and then executed. The kernel automatically probes, identifies, and initializes as much of your system's hardware as possible, and then looks for an initial file system that it can access and load and run applications from in order to continue the boot process. The first file system mounted by Linux systems during the boot process is known as a root file system because it is automatically mounted at the Linux directory '/', which is the base of the hierarchical Linux file system. Once mounted, the root file system provides the Linux system with a basic directory structure that it can use to map devices to Linux device nodes, access those devices, and locate, load, and execute subsequent code such as system code or your custom applications. In Fig. 5, an error occur showing that it cannot find the root file system in the booting process because the file system is not placed.

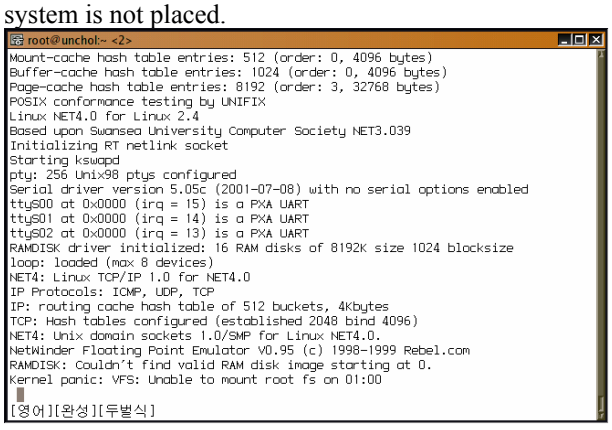

Fig. 5 Booting sequence of the RTAI patched Linux kernel

3.3 File System Construction

File system represents the physical storage space where shell, library, application programs, and others are located as shown in Fig. 6. Fig. 6 shows the types of file system used for Embedded Linux and the construction process.

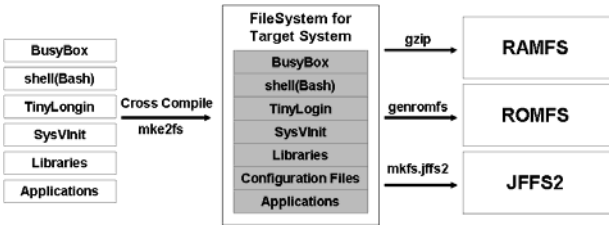

Fig. 6 Filesystem construction process

In Linux, programs such as shells, application programs and libraries are available as open source. However, these cannot be directly compiled and used. This is because the provided sources are available for x86 architecture. To make a file system, you should modify the provided setting options of each source, and then compile them by using a cross-compiler and cross-libraries. The created execution files and libraries are converted into a single file system image using file system utility.

In Embedded Linux, RAMFS (RAM File System), ROMFS (ROM File System) and JFFS2 (Journaling Flash File System Version 2) are generally used. The types of root file systems that your Embedded Linux system supports during the boot process depend on the types of file systems that are supported by the kernel that you are booting. Root file system in formats such as the JFFS2 are typically used on systems with Flash memory that can be partitioned into multiple sections, usually containing the boot monitor, the loadable kernel image, and a JFFS2 file system. JFFS2 file system has been constructed including BusyBox, Tinylogin, Bash, and libraries. JFFS2 file system reduces system restart time by minimizing the chances that a file system will be left in an inconsistent state by a system crash or unplanned restart. In this paper, we also developed JFFS2 file system. Once the file system is completed, download the file system to the system board and store it into the Flash memory.

3.4 Device Driver Creation

The device driver is a standard interface used for delivering information between the operating system and the hardware. In addition, when creating the device driver, it enables different hardware to approach to a common interface. Linux device driver is divided into the character device, block device, and the network device driver. The character device driver is device that enables reading and writing without undergoing buffer,

such as serial/parallel communication, keyboard, mouse, etc. The block device driver is a device inputting/ outputting data in block units through buffer cache, including hard disk, and CD-ROM. The network device driver is a device transmitting and receiving packets, such as Ethernet. [11]

The device driver may be created using two methods. The first method is to include it in the kernel and initialize when booting, and the other method is creating it as a module and including it to the kernel, only when necessary. In the present thesis, as an example of directly registering to the kernel, the network driver creation method shall be discussed.

First, create the driver source for the Ethernet used in the Linux source's driver/net directory. However, because most device driver sources are included in the kernel source, there are few times when the total source has to be created, and in many cases only the memory map or the interrupt-related portion is modified. Also in the present thesis, the smc91x.[ch] source, included in the kernel, has been modified and used. Upon source code completion, modify the kernel setup menu script to set the source device created during kernel setup. However, if it already included in the kernel, set the menu.

Upon kernel setup completion, compile the kernel source, download it to the system board and store, followed by re-boot. Fig. 7 shows the booting results of the kernel, with the described file system stored in the system board and with device driver created.

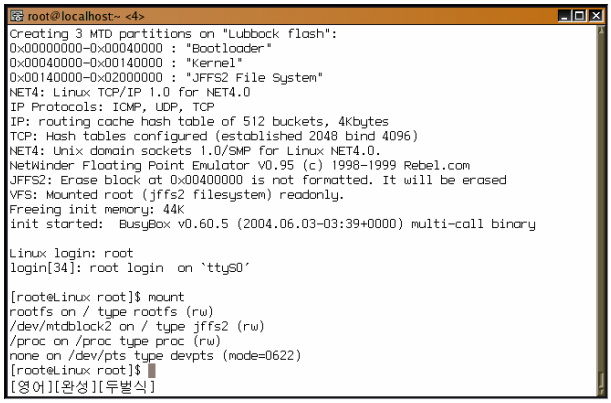

Fig. 7 Booting Sequence of Linux system with file system and device drivers

3.5 RTAI Installation & Test

To use HAL and Real-Time mechanism, provided by RTAI, module sources for each mechanism need to be compiled and added the kernel. However, compile and installation option of RTAI is prepared to us x86 compatible processors so that this option needs to be modified for PXA255. Additionally, x86 compatible processors support LXRT (Linux Real Time) and Real-Time serial drivers, but PXA255, used in this paper, does not support them. Therefore, it is necessary to build Real-Time serial drivers for PXA255.

The RTAI modules prepared for PXA255 are as follows.

rtai_bits.o, rtai_fifos.o, rtai_hal.o, rtai_ksched.o, rtai_math.o, rtai_mbx.o, rtai_mq.o, rtai_msg.o, rtai_sem.o, rtai_tasklets.o, rtai_up.o, rtai_usi.o, rtai_wd.o

To execute Real-Time task, the rtai_hal.o and rtai_ksched.o modules should be included in the kernel. The rtai_hal.o is the module for HAL, and rtai_ksched.o is the module for the scheduler. Other modules can be inserted to the kernel if necessary. Fig. 8 is the result of executing 4 different tasks, each with different cycles, priority, and execution orders. task1 and task2 execute in 20ms cycles, while task3 and task4 run in 30ms cycles. The order of execution is task1-> task2->task3->task4. Here, task1 and task3 have higher priority than task2 and task4. Because task1 has a higher priority than task2, it is executed first. Similarly, task3 is executed prior to task4. Out of task3 and task1, task1 is executed first by the priority, and because task2 has a lower priority than task3, it is executed after task3.

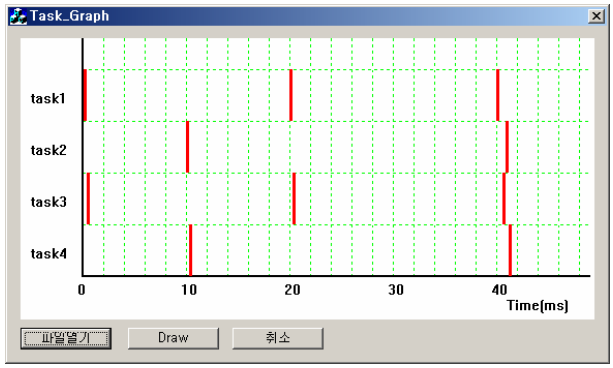

Fig. 8 Results of Real-Time task execution.

4. REAL-TIME SERIAL DRIVER

4.1 Real-Time Serial Driver Structure

In RTAI, Real-Time device drivers such as RT_COM used for serial communication, RT_NET used for Ethernet networking, and RT_CAN for CAN (Controller Area Network) are supported. In Real-Time device drivers, unlike Linux device drivers, the blocking function is excluded, and the IRQ and interrupt handler are registered to HAL. For instance, in the event of transmitting data in RT_NET, TCP communication verifying transfer status is not supported, and only UDP, which does not verify transfer status, is supported. [12, 13, 14]

In this paper, the serial driver, RT_COM has been applied. RT_COM is a Real-Time serial driver for RTLinux and RTAI, and is one of the open source projects performed in SourceForge.net.

Because the structure of Linux serial driver is as shown in Fig. 9, delay may occur handling other ISR (Interrupt Service Routine), the blocking function or other processes used in the device driver.

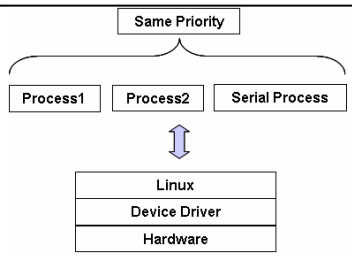

Fig. 9 Linux serial communication program structure

On the other hand, when utilizing RT_COM and Real-Time task, the structure is as shown in Fig. 10. The task priority of serial_task becomes higher than other tasks, and it may eliminate delay occurred due to other tasks. Real-Time is guaranteed because the blocking function is not included even in the device driver.

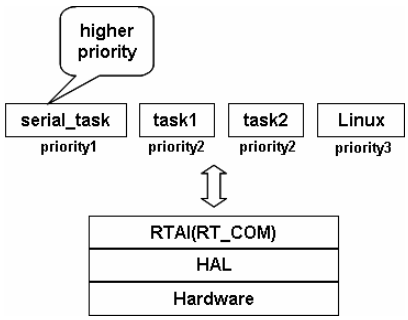

Fig. 10 Structure of serial communication using RT_COM

RT_COM stores information and data of each serial port to the data structure of the rt_buf_struct and rt_com_struct. The rt_buf_struct is a structure to integrate S/W FIFO (First-In First-Out), and is connected with H/W FIFO. The start address of serial port, IRQ, ISR, and serial port-related setup values are stored in the rt_com_struct. Because RT_COM is implemented with module structures, it is loaded to kernel when it is required. IRQ and ISR stored in the rt_com_struct are registered to HAL, and the interrupt is activated. In HAL, if an interrupt corresponding to IRQ occurs, the registered ISR is invoked.

The main functions of RT_COM are composed of rt_com_setup, rt_com_write, rt_com_read. The rt_com_setup is a function initializing the communication speed, parity bit, data size, FIFO trigger level, etc, while rt_com_read is a function reading data from the serial port. However, the rt_com_write and rt_com_read are not functions which directly handle the hardware FIFO, and they either read data stored in the S/W FIFO, or write data into this FIFO. Real data input/output is performed in the ISR, called by the occurred interrupt when writing the set data amount to FIFO, from the FIFO trigger level. This function either writes the data into the hardware FIFO, or reads the data stored in this FIFO. [15]

4.2 Implementation of Real-Time Serial Driver

RT_COM has been prepared for x86 compatible processor, with built-in UART controller. Therefore, it is available for any processor included built-in 16550 compatible UART controllers through source code modification.

The portion to be modified in order to use RT_COM in PXA255 are hardware dependent stuffs that are the start address of the UART controller, IRQ allocated from the processor, and the internal register of the UART controller. The start address of PXA255 UART controller has been modified to 0xf8200000, and IRQ to 14. Also, the clock cycle used for UART controller, FIFO trigger level, and interrupt-related registers have been modified.

4.3 Comparison between Real-Time Serial Driver and Non Real-Time Serial Driver

To perform the experiment comparing RT_COM and Linux serial driver, we apply them to motor speed control, in which the speed command is transferred by serial communication.

The communication speed between the motor module and system board is 57600bps. The encoder used in this paper has 4000 pulses in one cycle. The system board counts the encoder value with 4-multiple method. In this paper, the traveling distance of one rotation cycle is around 308mm, and the maximum velocity of the motor is 500mm/s. The motor speed command is transferred on every 20ms.

The experiment procedure is as follows.

1) Increase speed with 0.2m/s^2 acceleration during 2 seconds.
2) Maintain constant velocity during 2 seconds with constant motor acceleration.
3) For 4 ~ 6 seconds, reduce the speed with the motor acceleration of 0.2m/ s^2

Fig. 11 is the result of using RT_COM, and Fig. 12 is the result of using the Linux serial driver. The traveling distance of Fig. 11 is 1.71m, and the traveling distance of Fig. 12 is 2.06m.

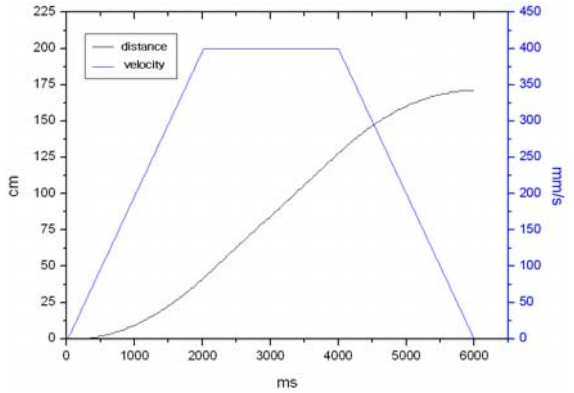

Fig. 11 Motor control using RT_COM

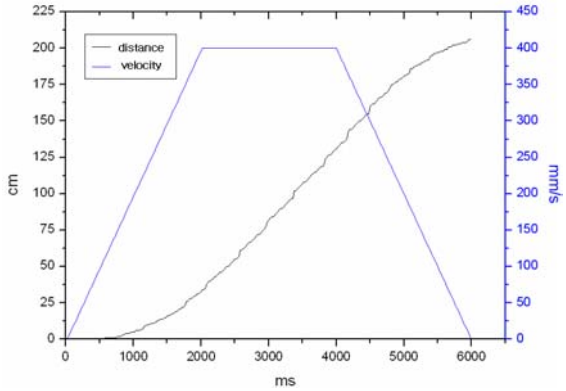

Fig. 12 Motor control using Linux serial driver

As shown in Figs. 11 and 12, the velocity of the mobile robot has increased and decreased in fixed ratio when using RT_COM. However, when using the Linux serial driver, the velocity of the mobile robot has increased and decreased irregularly because the velocity command is irregularly transferred to the motor controller. Consequently, the control period is not guaranteed and delayed. Therefore, the traveling distance in the case of the Linux device driver resulted in 0.35m greater than when using RT_COM. The reason for such result is because when using Linux serial driver, speed could not be increased and decreased with exact cycles due to other ISR or processes.

5. CONCLUSION

Embedded systems such as information appliance and control device are being small-sized and light-weighted while providing various high performances. These devices usually employ the SoC. Although SoC provides diverse development advantages, it is limited when being developed without an operating system because of its complex functions. Therefore, embedded operating system and Real-Time operating systems are being applied in the embedded systems. However, embedded operating systems cannot support Real-Time, we have to use expensive Real-Time operating systems.

In this paper, we use a Real-Time Embedded Linux, RTAI to utilize the advantages of Embedded Linux, such as open source licensing and a strong developer community. RTAI, by choosing the method of providing interface for Real-Time tasks using Linux as the basic operating system, has minimized kernel modification. Therefore, it does not only provide Real-Time with simple modification operations in the Embedded Linux-applied system. And it supports average latency time of 3 ~ 4us, it does not fall back to commercial operating systems (1 ~ 20us). Furthermore, we developed Real-Time serial driver for PXA255 target system, and applied motor control system to show the advantages of the Real-Time system.

The control system platform developed in this paper is a SoC based embedded system, use RTAI and Real-Time serial device driver to guarantee periodic response time. We also verify the feasibility of control system platform using RTAI with motor control system which is widely used for mobile robots. We believe that the results of this paper are useful to build Real-Time control platforms for intelligent mobile robots.

REFERENCES

[1] Electronic Tread Publications, System on chip Market and Trends, April, 2003.

[2] Deok Yeon Cho, Byoung-Wook Choi, "Commercial Inverter's Web-based Remote Management Using Embedded Linux", Control & Automated System 제어 및 Engineering Association, 9th book 4th issue, 340-346, 2003.

[3] Byoung-Wook Choi, Eun-Cheol Shin, Soo Young Lee, "Web-based Building Automated System Using Embe dded Linux", Control & Automated System Engineering Association, 10th book 4th issue, 335-341, 2004.

[4] Byoung-Wook Choi, Hyun Ki Kim, Eun-Cheol Shin, "Connection Application Between Embedded Linux Based Multiple Protocol Controller Development & Building Automated System", Control & Automated System Engineering Association, 10th book 5th issue, 428-433, 2004.

[5] Byoung-Wook Choi, "Embedded Operating System and It's Applications in Development for Networked Robot", Korea Robot Engineering Association, 1st Workshop, 2004.

[6] Huins Technology Research Center Co, "Intel PXA255 and Embedded Linux Application", 2004, Heung Neung Science Publisher.

[7] Ismael Ripoll, "RTLinux versus RTAI", www.linux devices.com, 2002.

[8] Kevin Dankwardt, "Comparing real-time Linux alternatives", www.linuxdevices.com, 2000.

[9] L.Dozio, P.Mantegazza, "Linux Real Time Application Interface(RTAI) in low cost high performance motion control", Motion Control 2003, a conference of ANIPLA, Associazione Nazionale Italiana per I'Automazione, Milano, Italy, 27-28, March, 2003

[10] Herman Bruyninckx, "Real Time and Embedded Guide", December, 2002

[11] Written by Alessandro/Translated by In Seong Kim, Tae Joong Ryu, "Linux Device Driver", February, 2000, Hanvit Media.

[12] Hard Real-Time Networking for Linux/RTAI, www.rts.uni-hannover.de/rtnet.

[13] Real-Time CAN on Linux, sourceforge.net/project/rtcan

[14] Real-Time Linux driver for the serial port, rt-com.sourceforge.net.

[15] P.N. Daly and Kupper, "The serial port driver of Real-Time Linux", Real Time Documentation Project, Vol 1, 2000