Vol.2014-SLDM-165 No.33
Vol.2014-EMB-32 No.33
2014/3/16

An RTOS-based Platform for LEGO Mindstorms EV3

Yixiao Li[1,a)] Takuya Ishikawa[1] Yutaka Matsubara[1] Hiroaki Takada[1]

Abstract: In this paper, we describe the design and implementation of a platform for developing real-time applications running on LEGO Mindstorms EV3. The platform is based on the TOPPERS/HRP2 kernel and the porting of the kernel is also described. Many device drivers such as PWM controlling and Bluetooth are developed for this platform. The development work has been reduced a lot by proposing an approach to reuse the Linux kernel-space device drivers. An API for C language is implemented and its performance is evaluated. At last, we show the development process of our platform by developing a sample program for a self-balancing robot.

1. Introduction

Lots of researches and publications have shown that using robots is playing an important role in the graduate level research and college education such as computer science [1], [2], [3], [4], [5]. For example, Kumar and Meeden showed [6] that building a robotics laboratory can provide a pedagogical tool for teaching artificial intelligence (AI) courses more effectively. Similarly, Larsen and Wang Yi [7] developed a tool box called UPPAAL for modeling, simulation and verification of real-time systems by using a robotics kit to create the platform for experiments.

Mindstorms [8] is a series of robotics kits released by LEGO Inc. since 1998. They include a programmable brick computer called Intelligent Brick that control the whole system and a set of modular sensors and motors, and LEGO blocks to allow users to build robots flexibly. The latest generation of Mindstorms series is called LEGO Mindstorms EV3 [8]. The capability of Mindstorms EV3 has increased greatly comparing to its prior generation model, the LEGO Mindstorms NXT which runs on a 48MHz ARM7 CPU and has only 64KB RAM. Mindstorms EV3 is equipped with a 300MHz ARM9 CPU and 64MB RAM which allows it to run a Linux-based firmware. It also supports modern wireless technologies such as Wi-Fi and Bluetooth v2.1 Enhanced Data Rate (EDR).

Mindstorms EV3 provides a standard development environment based on the Laboratory Virtual Instrument Engineering Workbench (LabVIEW). LabVIEW uses a graphically dataflow programming language whose programming is done by dragging and dropping icons into a line in order to form commands. The development environment is shown in **Fig. 1**. Although the development environment is friendly to those who are not familiar with computer programming, there are some disadvantages for the developers who are already familiar with common programming languages such as C or C++ as described below:

- Hard to write complex programs: The graphical programming language is too simple to write programs with complex logic. Even if it was done in some way, the source program will become almost unreadable and unmaintainable. A small change in the functions can lead to a lot of modifications.
- Lack of real multi-tasking support: The standard development environment does support multi-tasking to some extent. However, it doesn't support priority-based scheduling and task synchronization primitives such as semaphores.
- No real-time guarantee: All programs developed by the standard development environment are running on a virtual machine which doesn't provide any real-time guarantees for various control operations.
- No third party devices support: The standard development environment is not open source and doesn't provide interfaces to add the support for third party devices.

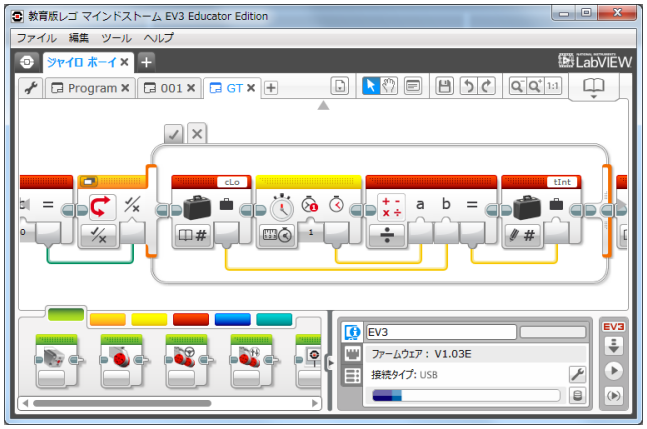

Fig. 1 The standard development environment for Mindstorms EV3

Therefore, we decided to build a development platform for those developers who are suffering from the disadvantages mentioned above. The main objectives of our platform are described as below:

- Real-time guarantees: The platform should work on a real-time operating system (RTOS) to provide real-time guarantees. The use of RTOS also gives a chance to boot Mind-

[1] Graduate School of Information Science, Nagoya University
[a)] liyixiao@ertl.jp

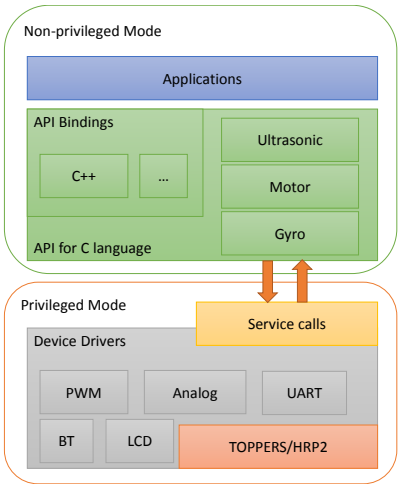

Fig. 2 The architecture for our platform

storms EV3 up much more quickly than the default Linux-based firmware which takes more than 30 seconds to boot up.

- Programming with common languages: The platform should allow developers to develop with common languages. At least, an Application Programming Interface (API) for C language should be provided. Moreover, an easy approach to add the support for other languages should be available.
- Assistance for development: Features to assist the process of development should be provided. For example, features such as viewing task log through Bluetooth and transferring applications wirelessly are preferred.
- Support for third party devices: The architecture of the platform should be open to support the third party devices easily.

It is worth noting that this research was conducted as part of the Education Network for Practical Information Technologies (enPiT) [19].

2. Design of the Platform Architecture

We designed the architecture of our platform to achieve the objectives mentioned above. The architecture is shown in **Fig. 2**.

We chose the TOPPERS/HRP2 (HRP stands for "High Reliable system Profile", and 2 is the version number) [9] kernel as the kernel of our platform. HRP2 is an RTOS that satisfies high reliability and safety requirements of large-scale embedded systems by introducing the protection functionality such as memory protection, object access protection and extended service call functionality. With the protection functionality of HRP2, we are able to protect the kernel from defects in applications.

Above the HRP2 kernel are the device drivers. Since Mindstorms EV3 supports lots of features, there are many device drivers to be provided, such as pulse-width modulation (PWM) control driver for controlling the motors, universal asynchronous receiver transmitter (UART) communication protocol driver for UART sensors and Bluetooth protocol stack. Device drivers are running in privileged mode and should provide extended service calls which can be called in non-privileged mode.

Above the device drivers are the API for C language. It is implemented with extended service calls provided by device drivers. However, unlike the device drivers which are mainly focusing on supporting the communication protocols, the API is focusing on providing interfaces to operate particular devices such as servo motor, color sensor, gyro sensor and ultrasonic sensor. The supporting of other languages such as C++ can be implemented relatively easily by wrapping the API for C language. The API should be able to run in non-privileged mode.

The topmost are applications. Applications are developed with APIs and running in non-privileged mode. Therefore, exceptions and crashes of applications won't affect the kernel.

To implement the whole architecture, there are some key issues to be solved:

- HRP2 kernel has not supported AM1808, the processor of Mindstorms EV3 yet. We are required to port the kernel to AM1808 at first.
- The scale of device drivers is too large. The source code of the stock Linux-based firmware has been released by LEGO [18]. We analyzed it with CLOC (Count Lines of Code), a statistics utility to count lines of code. The result shows that there are still approximately 60000 lines of code even after removing all the headers, blank lines and comments. And it is important to note that this is just a statistics on the device drivers exclusive for Mindstorms EV3 such as PWM control driver. If we also counted the generic device drivers like USB or Bluetooth protocol stacks, the scale of device drivers will be approximately half a million lines. Since it is very difficult, if not impossible, to implement the device drivers from scratch, we must find a way to effectively reuse the existing source code such as the Linux-based firmware or other open source project that can be applied to our platform.

3. Porting of the TOPPERS/HRP2 kernel

We ported the TOPPERS/HRP2 kernel to the processor AM1808 used by Mindstorms EV3 by following the porting guide provided by the TOPPERS project [9]. The porting includes the following parts:

- Driver for ARM Interrupt Controller (AINTC): AM1808 is based on the ARMv5 architecture which doesn't define a common interrupt controller programming interface like the Generic Interrupt Controller (GIC) used in the ARMv7 architecture and after. Instead, it uses an interrupt controller called AINTC defined by the processor [10]. Therefore, we implemented the driver for it.
- Handling IRQ with priority and nesting: AINTC has the feature of hardware prioritization and nesting of interrupts. We supported handling IRQ with this feature.
- Driver for the serial port: We implemented the driver for the serial port so that it can be used through the TOPPERS serial interface to provide functions such as outputting the logs for debugging.
- Support for ARMv5 MMU (Memory Management Unit): An MMU or MPU (Memory Protection Unit) should be supported to enable the memory protection functionality. AM1808 is equipped with both MMU and MPU [10]. Although the implementation of supporting an MPU can be done more easily, we believe that the lack of capability to

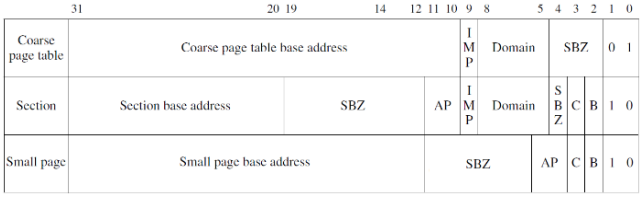

Fig. 3 The format of page table entries in ARMv5 architecture

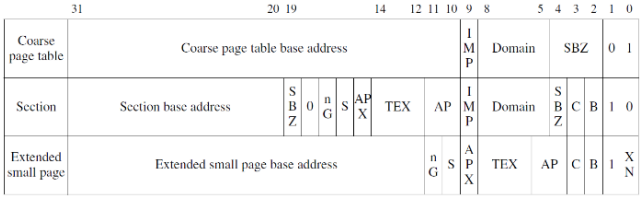

Fig. 4 The format of page table entries in ARMv6 architecture

protect memory space used by special registers would be fatal for our platform to provide complete protection of the kernel against applications. Therefore, we chose the MMU solution.

Unlike the Linux which modifies and manages page tables dynamically, the HRP2 kernel requires page tables for each protection domain, which has its own page tables like a process in the Linux, generated statically. The kernel switches among protection domains to provide the memory protection functionality. The hardest part is to implement the generations of page tables. The HRP2 kernel already supported the MMU of ARMv6 architecture and later. Since there are lots of similarities between ARMv6 MMU and ARMv5 MMU, we decided to modify the implementation for ARMv6 to support the ARMv5 architecture.

At first, we designed the translation rules from ARMv6 page tables to ARMv5 page tables. The formats of page table entries are shown in Figure 3 and Figure 4. The definition of coarse page table entry is exactly the same. However, for the section entry and small page entry, a lot of fields are introduced in ARMv6 architecture. The S (shared) bit for sharing memory with multiple processors, APX (access permission extension) bit for restricting writing in privileged mode and the XN (execute-never) bit for restricting executing are not supported in ARMv5 but can be ignored in our platform. In ARMv6 architecture, the cache policy for a page table entry is determined by TEX (type extension) bit, C (cacheable) bit and B (bufferable) bit altogether. Since only C bit and B bit are supported in ARMv5 architecture, we analyzed the cache policy on both the architectures, and then defined a translation table between them as shown in **Table 1**.

The most important difference between ARMv5 architecture and ARMv6 architecture is the nG (non-global) bit. A feature called application space identifier (ASID) is introduced in ARMv6 architecture. With this feature, an entry in translation lookaside buffers can be associated with a specific identifier, called ASID, to eliminate the requirement for TLB flushes on most context switches. A page which will be associated with an ASID is called non-global page. The nG bit determines whether a page is non-global. Unfortunately, the ASID feature is not supported in ARMv5 architecture. Therefore, the TLB flushes becomes necessary when switching among protection domains.

Flushing the entire TLB at each time of protection domain switching is very expensive and may have a negative effect for real-time applications. An instruction to invalidate a single TLB entry is provided in ARMv5 architecture. With this instruction, we made that only non-global pages will be invalidated on protection domain switching. Since the non-global pages are determined statically, we optimized the TLB flushes further by generating the assembly code for invalidating non-global pages statically, rather than managing a list for non-global pages and traversing it at each time of protection domain switching. We measured the performance of the method which flushes the entire TLB and our implementation. The results show that flushing the entire TLB takes 1232 CPU cycles and our implementation takes only 25 CPU cycles per non-global page. That means if the number of non-global pages is small, our implementation can achieve a very low overhead on protection domain switching.

			ARMv6	ARMv5	
TEX	C	B	Cache Policy	C	B
000	0	0	Strongly order	0	0
000	0	1	Shared device	0	0
001	0	0	Outermost cache non-cacheable Innermost cache non-cacheable	0	1
000	1	0	Outermost cache write through Innermost cache write allocate	1	0
001	1	1	Outermost cache write back Innermost cache write allocate	1	1

Table 1 Translation rules for cache policy between ARMv6 and ARMv5 architecture

4. Porting of Device Drivers

As mentioned earlier, the scale of device drivers for Mindstorms EV3 is too large to implement from scratch. We must find a way to reuse the existing source code to reduce our coding work as more as possible. We proposed an approach to reuse the kernel-space Linux device drivers on the TOPPERS/HRP2 kernel and evaluated its effectiveness. We also ported an open source Bluetooth protocol stack called BTstack [16] to our platform for supporting wireless communication.

4.1 Reusing Linux device drivers

We performed an investigation on the development of Linux device drivers [12]. It shows that Linux device drivers can be divided into two types, the kernel-space device driver and the user-space device driver. The kernel-space device drivers are developed with the Linux kernel API [13] which only provides basic management functionality for the kernel. On the other hand, the user-space device drivers are developed with the Linux kernel user-space API which is based on the Single UNIX Specification version 4 (also known as POSIX.1-2008) [14].

The Linux kernel user-space API is so complex that it is almost impossible to implement it on a non-UNIX-like operating system in a short time. It will apparently cost much more time even than implementing the device drivers by ourselves. Therefore we de-

cided not to reuse the user-space drivers. There are many similarities between the Linux kernel API and the TOPPERS API, which gives us a chance to reuse the kernel-space device drivers by implementing a part of the Linux kernel API with the TOPPERS API. The Linux kernel API we implemented are described below:

- Kernel-space memory management: Functions such as `kmalloc()` and `kfree()` are implemented. The Linux uses multiple virtual address spaces while the HRP2 kernel uses a flat memory model. Therefore it is unnecessary to perform the address translation between kernel space address and user space address. In the case of `kmalloc()`, it can be implemented easily by wrapping the `malloc()` function.
- Interface for interrupt handling: Functions such as `request_irq()` which is used to register an interrupt handler for an interrupt number, `request_gpio_irq()` which is used to register an interrupt handler for a GPIO pin, and `free_irq()` which is used to remove an registered interrupt handler are implemented. It should be noted that the dynamic creation of an interrupt handler is not supported by the TOPPERS/HRP2 kernel yet. We chose to define the interrupt handlers statically and let the functions like `request_irq()` or `free_irq()` simply perform enabling or disabling of the corresponding interrupt. The interrupts of GPIO pins are grouped into banks. To implement the `request_gpio_irq()` function, we implemented a GPIO interrupt dispatcher which dispatches interrupt to the corresponding handler.
- Semaphore API: Functions such as `down_trylock()` and `up()` are implemented. These functions can't be implemented by wrapping the semaphore management functions in the TOPPERS API directly because these operations are not permitted when CPU is locked (i.e. all the interrupts are masked) according to the TOPPERS specification. We implemented these functions natively by referencing the implementation of corresponding functions in the TOPPERS API.
- Spinlock API: Functions such as `spin_lock_irqsave()` and `spin_unlock_irqrestore()` are implemented by wrapping the `SIL_LOC_INT()` and `SIL_UNL_INT()` macros which are used to control whether all the interrupts are masked.
- High-resolution timer API: The high-resolution timers have become the standard time framework in Linux since version 2.6.16 [15]. The Linux device drivers for Mindstorms EV3 use the high-resolution timer API for handling events periodically. Unfortunately, the HRP2 kernel has not supported the high-resolution timer feature yet. The HRP2 kernel handles periodic events by kernel objects called cyclic handlers. The period of cyclic handlers can only be set in milliseconds. However, the HRP2 kernel does allow us to provide high-resolution periodic ticks. By defining the `TIC_NUME` and `TIC_DENO` macros, we can set the period of system ticks to `(TIC_NUME/TIC_DENO)` milliseconds. We decided to set this period to 200 microseconds and implemented an interface for high-resolution cyclic handlers by handling system ticks. We then implemented the high-resolution timer API with this interface.

```
// Include the common part for a driver
#include "driver_common.h"

// Hacks for this module
#define InitGpio PWM_InitGpio
static void SetGpioRisingIrq(...) {
    ...
}

// Include the source file to reuse
#include "d_pwm.c"

// Interfaces
void pwm_command(...) {
    ...
    Device1Write(...);
    ...
}
```

Fig. 5 An example for reusing Linux device driver

With the Linux kernel API implemented above, the core parts of the Linux device drivers can already work on our platform. To allow the interactions between device drivers and applications, we are also needed to adapt the device driver model for our platform. In Linux, a special file is used as an interface for a device driver. It allows user-space applications to interact with a device driver using the standard file operation system calls. There are no file systems in the HRP2 kernel by default. Instead, it allows us to add new system calls called extended service calls for a device driver. We decided to wrap the file operation functions as extended service calls.

Fig. 5 uses an example to show how to reuse a device driver in particular. At first, the header file `driver_common.h` is included, which contains the Linux kernel API we implemented. Then, some hacks are used to make the device driver compiled and working properly. The source file of the device driver is included after the hacks. To keep the maintainability, this source file should not be changed. We just comments out the unneeded code in it. At last, the interfaces for applications are implemented as extended service calls by wrapping the file operation functions.

We reused the following device drivers with this approach successfully:

- PWM control driver for motors
- Analog I/O driver for analog sensors
- UART communication protocol for UART sensors
- Soft UART ports driver

We used CLOC to evaluate the effectiveness of our approach. The results show that we reused 14254 lines of code by writing only 669 lines of code. This approach has saved more than 90% of the coding work for these device drivers.

4.2 Porting of the BTstack

The standard Bluetooth protocol stack for the Linux is BlueZ. BlueZ is a user-space device driver which is almost impossible to port to our platform. We chose to port the BTstack, an open source operating system independent Bluetooth protocol stack

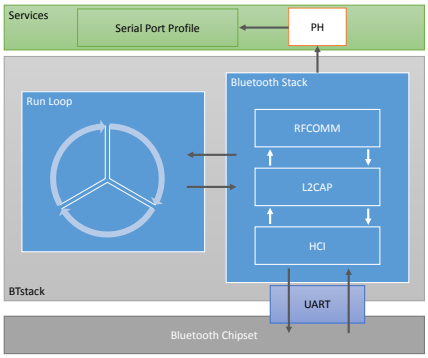

Fig. 6 The architecture of BTstack

[16]. The architecture of BTstack is shown in **Fig. 6**.

We implemented the hardware initialization for the Bluetooth chipset by referencing the Linux device driver. The chipset and BTstack are communicating with each other via a UART port. We implemented the method stubs in the UART hardware abstraction layer defined by BTstack. BTstack uses a run loop to handle incoming data and schedule work. By default, BTstack only provides two types of run loops, one for POSIX system and the other one for OS-less system. The run loop for OS-less system runs as a busy loop. We modified it to run as a periodic task in our platform. At last, we implemented the packet handlers (PH) to provide the Bluetooth Serial Port Profile (SPP) which emulates a serial port over air. We also made a serial port driver for SPP that allows outputting task logs via Bluetooth.

It should be noted that the Bluetooth can work at a high data rate up to 3 Mbps. Handling the data transferring with interrupts may have a negative effect for real-time applications. We decided to handle it with a low priority task working in polling mode.

5. Application Programming Interface

Providing an API can let developers write programs much more easily than use the device drivers directly. We implemented an API for C language by using the extended service calls provided by device drivers so that it can be used in non-privileged mode. Some functions provided by our API are listed as below:

- `ev3_motor_set_speed()`
  Control the speed and direction of a motor
- `ev3_motor_sync()`
  Steer or synchronize with two motors
- `ev3_motor_get_counts()`
  Get the angular position of a motor (a.k.a. rotary encoder)
- `ev3_gyro_sensor_get_angle()`
  Detect the rotation of a robot with a gyro sensor
- `ev3_gyro_sensor_get_rate()`
  Measure the angular velocity of a robot with a gyro sensor
- `ev3_ultrasonic_sensor_get_distance()`
  Measure the distance to an object with an ultrasonic sensor
- `ev3_touch_sensor_is_pressed()`
  Detect whether the button of a touch sensor is pressed
- `ev3_color_sensor_get_color()`
  Distinguish between 8 different colors with a color sensor
- `ev3_uart_sensor_get_raw()`
  Read the raw value of a UART sensor

Function	Average time	Maximum
ev3_motor_set_speed()	4.54 us	40 us
ev3_uart_sensor_get_raw()	5.55 us	33 us

Table 2 Execution time of the representative functions chosen from our API

- `ev3_led_set_color()`
  Set the color of LED on the body of Mindstorms EV3
- `ev3_button_set_on_clicked()`
  Register a handler for the button click event

It should be noted that a button click event is triggered by a GPIO pin interrupt so the registered handler will be called during handling the interrupt. It means that the handler will run in privileged mode which can't provide any protection. Using a dedicated task running in non-privileged mode might be a workaround but we didn't choose it. We decided to leave this problem for the future and wish to deal with it more elegantly.

By wrapping the API for C language, other programming languages such as C++ can be supported easily. In particular, we have made our platform able to be compiled with C++ compiler. The API design for C++ language is still under discussion.

We did a simple performance evaluation on the API to show the efficiency of our platform. We chose two representative functions `ev3_motor_set_speed()` and `ev3_uart_sensor_get_raw()` from our API. The `ev3_motor_set_speed()` function is implemented by sending a message to the PWM control driver. This can be done by simply calling the extended service call `motor_command()` provided by the PWM control driver in our platform. However, in Linux, the message is passed through the file system by calling the file operation system call `write()`, which may lead to a huge overhead. We measured the execution time of each of the functions for 10000 times and the results are shown in **Table 2**.

To figure out how fast the results represent, we would like to evaluate the performance of the standard development environment also. However, the standard development environment is not open source so it is hard for us to measure it accurately. Instead, we decided to measure the Linux device drivers. We implemented the `ev3_motor_set_speed()` function on Linux with the `write()` system call. We used `strace`, a debugging tool for Linux to monitor the system calls, to measure the function for 40000 times. The results show that calling `write()` costs 596 microseconds averagely and 3424 microseconds at most. It means the speed of our API can be almost 100 times of the standard development environment in this case.

6. A Sample Program for Self-balancing Robot

On our platform, an application can be developed and executed with the following steps:

(1) Write code with the API
(2) Build the binary image `hrp2` by the `make` command
(3) Generate the boot image `uImage` from `hrp2` by the `mkimage` command
(4) Copy `uImage` to the root of a microSD card
(5) Insert the microSD card into the body of Mindstorms EV3
(6) Push the center button on the body of Mindstorms EV3 and

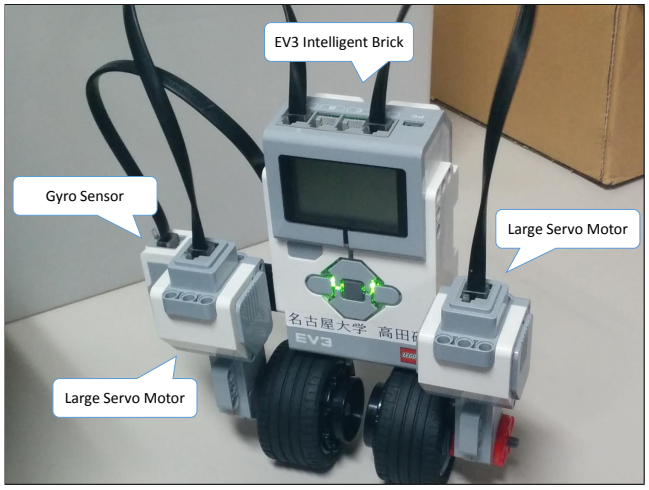

Fig. 7 A self-balancing two-wheeled robot

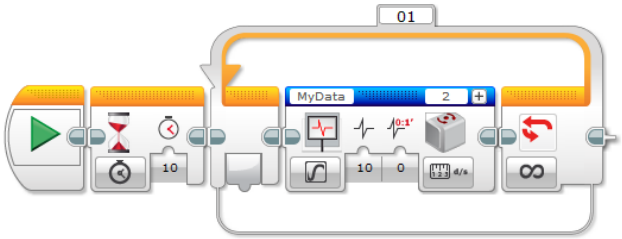

Fig. 8 A task to log the value of gyro sensor in the standard development environment

the application will get executed

To validate the implementations of our platform, we made a sample program for a self-balancing two-wheeled robot, which has real-time requirements. The construction of the robot is shown in **Fig. 7** . We implemented the self-balancing algorithm by referencing the HTWay [17]. We made the self-balancing algorithm work as a task. There is another task in our sample program. The task is to output the value of gyro sensor continuously via Bluetooth and works in a lower priority than the self-balancing task. The result shows that the robot can boot up in 5 seconds and both the tasks can work flawlessly.

On the other hand, the standard development environment provided by Mindstorms EV3 also includes a sample program for self-balancing called GyroBoy. Although it takes more than 35 seconds to boot up, GyroBoy itself works correctly. However, if we add a task like shown in **Fig. 8** to log the value of gyro sensor simultaneously, the robot will fall down very easily. This comparison shows that our platform can boot up much faster and is more suitable for the applications with real-time requirements than the standard development environment.

7. Conclusions and Future Work

The most important goal of this research is to build a convenient development platform with real-time guarantees for Mindstorms EV3. We ported the TOPPERS/HRP2 kernel to provide real-time guarantees and protection functionality. For the device drivers, we proposed an approach to reuse the Linux kernel-space device drivers. We also made a Bluetooth protocol stack called BTstack work on our platform. We implemented an API for C language and evaluated its performance. The results have shown that our platform is much more faster than the standard development environment. At last, we tested the whole platform by making a sample program for a self-balancing robot. Our platform is proven to be more suitable for real-time applications by a comparison between this sample program and the standard development environment.

In the future, we plan to support more features to assist the development such as transferring and loading applications wirelessly. An elegant solution for the non-privileged mode interrupt handlers described in the 5th section should be found. For now, the data transferring of Bluetooth is working as a low priority in polling mode. A DMA-based implementation should be done to increase the efficiency. The device drivers for other important functions such the LCD should also be implemented.

References

[1] Fagin, B. S., Merkle, L. D., and Eggers, T. W.: Teaching computer science with robotics using Ada/Mindstorms 2.0, *ACM SIGAda Ada Letters*, Vol. 21, No. 4, pp. 73-78 (2001).
[2] Klassner, F.: A case study of LEGO Mindstorms' suitability for artificial intelligence and robotics courses at the college level, *ACM SIGCSE Bulletin*, Vol. 34, No. 1, pp. 8-12 (2002).
[3] Klassner, F. and Anderson, S. D.: Lego MindStorms: Not just for K-12 anymore, *IEEE Robotics & Automation Magazine*, Vol. 10, No. 2, pp. 12-18 (2003).
[4] Fagin, B. and Merkle, L.: Measuring the effectiveness of robots in teaching computer science, *ACM SIGCSE Bulletin*, Vol. 35, No. 1, pp. 307-311 (2003).
[5] Sell, R., and Seiler, S.: Combined Robotic Platform for Research and Education, *Proceedings of SIMPAR*, pp. 522-531 (2010).
[6] Kumar, D. and Meeden, L.: A robot laboratory for teaching artificial intelligence, *ACM SIGCSE Bulletin*, Vol. 30, No. 1, pp. 341-344 (1998).
[7] Larsen, K. G., Pettersson, P. and Yi, W.: UPPAAL in a nutshell, *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 1, No. 1, pp. 134-152 (1997).
[8] LEGO Group: LEGO.com Mindstorms, available from ⟨http://www.lego.com/en-us/mindstorms/⟩ (accessed 2014-02-12).
[9] TOPPERS Project Inc.: TOPPERS/HRP2 kernel, available from ⟨http://www.toppers.jp/en/hrp2-kernel.html⟩ (accessed 2014-02-12).
[10] Texas Instruments Inc.: *AM1808/AM1810 ARM Microprocessor Technical Reference Manual* (2011).
[11] ARM Ltd.: *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* (2012).
[12] Venkateswaran, S.: *Essential Linux device drivers*, Prentice Hall Press (2008).
[13] Linux Kernel Organization: Documentation for Linux kernel, available from ⟨https://www.kernel.org/doc/⟩ (accessed 2014-02-12).
[14] The Open Group: The Single UNIX Specification, Version 4, available from ⟨http://www.unix.org/version4/⟩ (accessed 2014-02-12).
[15] Mauerer, W.: *Professional Linux kernel architecture*, John Wiley & Sons (2010).
[16] BlueKitchen: BTstack, available from ⟨https://code.google.com/p/btstack/⟩ (accessed 2014-02-12).
[17] HiTechnic Products: HTWay - A Segway type robot, available from ⟨http://www.hitechnic.com/blog/gyro-sensor/htway/⟩ (accessed 2014-02-12).
[18] LEGO Group: LEGO MINDSTORMS EV3 source code, available from ⟨https://github.com/mindboards/ev3sources/⟩ (accessed 2014-02-12).
[19] INOUE, Katsuro and KUSUMOTO, Shinji and GOTO, Atsuhiro and UBAYASHI, Naoyasu and KITAGAWA, Hiroyuki: "Peta-gogy" for Future : Education Network for Practical Information Technologies enPiT, *Journal of Information Processing*, Vol. 55, No. 2, pp. 194-197 (2014). (in Japanese).