EV3RT: A Real-time Software Platform for LEGO Mindstorms EV3

Yixiao Li, Yutaka Matsubara, Hiroaki Takada

EV3RT is, to our knowledge, the first RTOS-based software platform for LEGO Mindstorms EV3 robotics kit. It is faster and more suitable for developing applications with real-time requirements than other existing software platforms. In practice, EV3RT has been selected as one of the officially supported platforms of ET Robocon, a popular robot competition in Japan, since 2015 and used by many participating teams to make their robots accomplish the assigned tasks more stably and precisely. In this paper, the usage and architecture of EV3RT are firstly introduced. We then explain TOPPERS/HRP2 kernel, the RTOS of EV3RT, and how to use its protection functionalities to build a reliable platform. A mechanism to support dynamic module loading in a static RTOS is proposed to implement the application loader for EV3RT. Implementation techniques like approach to reusing Linux device drivers are also described. Finally, the advantages of EV3RT are shown by evaluating and comparing its performance with other platforms.

1 Introduction

Mindstorms [26] has become one of the most popular series of robotics development kits since it was first released by LEGO Inc. in 1998. A Mindstorms kit consists of a programmable brick computer called intelligent brick that controls the whole system, and a set of modular sensors, motors and LEGO blocks that allow users to build robots flexibly. Many real-life embedded systems can be modelled with Mindstorms robots, and thus they are used as important tools in researches [24] [8] and college education like real-time control and artificial intelligence [10] [23]. Robotics competitions such as World Robot Olympiad [50] and RoboCup Junior [44] also utilize the Mindstorms robots.

LEGO Mindstorms EV3 (or just the EV3) [49] is the third and the latest (as of writing) generation of Mindstorms series. It supports many new features and is much more powerful than its predecessor, the LEGO Mindstorms NXT [17] series, as shown in Table 1. Although the NXT robots still have many users, they have been discontinued by manufacturer from the end of 2015 [25]. In the near future, the EV3 will become the most-used model in the Mindstorms series.

The standard software platform of EV3, however, has some disadvantages when developing applications with real-time requirements. It consists of an integrated development environment (IDE) and a Linux-based firmware. The IDE is based on LabVIEW [34] which uses a visual programming language as shown in Fig.1. It is proprietary software with very little extensibility. While it is friendly to beginners of computer programming, users who are already familiar with common programming languages like C or C++ may find it difficult to develop complex programs in this IDE. Multitasking features such as preemptive scheduling and synchronization are also unsupported. The Linux-based firmware (codename: lms2012) includes a GUI program to operate the intelligent brick and a virtual machine-based runtime, which takes a long time to boot up and has a huge memory footprint. Application will be compiled into an intermediate representation (a.k.a bytecode) to be ex-

---

EV3RT: LEGO Mindstorms EV3 用リアルタイム・ソフトウェア・プラットフォーム

李奕驍, 松原豊, 高田広章, 名古屋大学大学院情報科学研究科, Graduate School of Information Science, Nagoya University.

コンピュータソフトウェア, Vol.34, No.4 (2017), pp.91–115.
[ソフトウェア論文] 2016 年 10 月 31 日受付.

Table 1 Comparison of Mindstorms NXT and Mindstorms EV3

	Mindstorms NXT	Mindstorms EV3
Processor	ARM7@48MHz	ARM9@300MHz
Coprocessor	Atmel AVR@8MHz	n/a
ROM	256KB	16MB
RAM	64KB	64MB
Sensors	Analog, I2C	Analog, I2C, UART
Motors	Up to 3	Up to 4
Display	100×64 LCD	178×128 LCD
SD card	n/a	microSD (SDHC)
Bluetooth	v1.2	v2.1 + EDR
USB device	12Mbps	480Mbps
USB host	n/a	12Mbps
WiFi	n/a	USB dongle

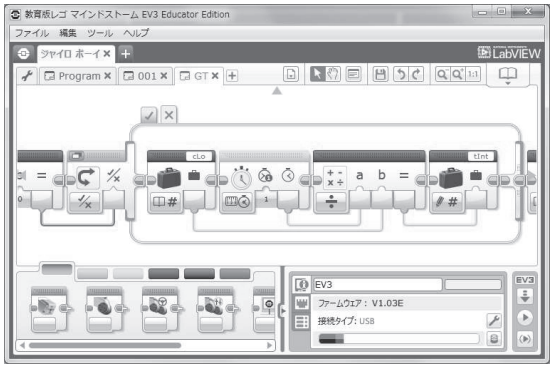


Fig. 1 Visual programming in LabVIEW

ecuted in the virtual machine. The overhead and unpredictable performance of Linux kernel and virtual machine can not provide real-time guarantees. There are some other software platforms such as leJOS EV3 [43] and MonoBrick [2], which support developing in Java or C#. However, they also use Linux and have their own runtime, and thus the disadvantages of lms2012 still remain.

In this paper, EV3RT, a real-time software platform to overcome above disadvantages, is presented. It is the first RTOS-based platform for EV3 to our knowledge and especially suitable for developing applications with hard real-time requirements such as balancing a two-wheeled robot [14]. Soft real-time and generic applications, including those for IoT devices or just entertainment, are also likely to run faster and smoother on EV3RT with its low-overhead APIs. ET Robocon (short for Embedded Technology Software Design Robot Contest) [22], a popular robot competition in Japan, has selected EV3RT as one of its officially supported platforms. Most participating teams of ET Robocon are using EV3RT to make their robots accomplish the assigned tasks more stably and precisely. Main features of EV3RT are listed as follows:

TOPPERS/HRP2 static RTOS kernel [47]. As an extended version of TOPPERS/JSP kernel, which is used by a popular RTOS platform for NXT called nxtOSEK/JSP [41], it can provide backward compatibility as long as many new features with guaranteed real time performance.

Programming in C and C++. Applications can be compiled and run as native code so the overhead of virtual machine is completely eliminated.

High reliability with enhanced protection. The base system of EV3RT is protected so it will not be affected by defects in user applications, which makes the debugging effortless for a large-scale embedded software platform like EV3RT.

Easy-to-use and low-overhead APIs. Users can easily develop applications with high real-time performance and need not learn the complex implementation details of various device drivers.

Dynamic application loading. A static RTOS generally requires the system to be rebooted when the application is updated. In the case of EV3, developers have to reset the intelligent brick and reconnect network such as Bluetooth every time after modifying an application, which will waste lots of time, especially when developing robot control

applications which are usually updated frequently to adjust control logic and parameters. To solve this problem, EV3RT provides a loader for the static TOPPERS/HRP2 kernel, which allows the dynamic updating of application without reboot.

Bluetooth and USB device support. Connectivity technologies are important in many scenarios, such as creating interactive applications or modeling an IoT device. Helpful features like uploading application wirelessly are also supported.

Ultra-fast boot process. The slow startup is one of the most annoying problems for EV3 developers using Linux-based platforms. Meanwhile, it only takes about 2 seconds for EV3RT to boot up.

Very small memory footprint. Storing or caching data in memory as possible is a common technique to improve performance and reduce latency. More than 95% of total memory can be used by applications on EV3RT.

Complete open source platform [16] [15]. Since Mindstorms robots have been used in so many different ways, the standard features and behaviors of EV3RT may not fit some users' needs. In that case, users can freely modify any part of EV3RT as all the source code is available.

Extendable software architecture. The hardware of EV3 itself is extendable by connecting devices like third party sensors. Supporting of those devices can be easily implemented with the flexible architecture provided by EV3RT.

The rest of this paper is organized as follows. An overview of EV3RT is given in Section 2 at first, by describing its architecture and usage. In Section 3, TOPPERS/HRP2 kernel, the RTOS of EV3RT, is explained, focusing on applying its protection functionalities to make EV3RT more reliable. A dynamic module loading mechanism for static OS design is proposed in Section 4, in order to implement the application loader for EV3RT. We then introduce some implementation techniques such as reusing Linux device drivers in Section 6. The performance of EV3RT is evaluated and compared with existing software platforms in Section 7. Finally, the paper is concluded with some future work discussed.

2 Overview of EV3RT

EV3RT is a developer-friendly open source software platform for real-time applications. In this section, we first give an overview of EV3RT through explaining its architecture. Thereafter, how to develop application with EV3RT in practice is also briefly described.

2.1 Platform Architecture

Application in EV3RT can be developed and built in two modes: dynamic loading mode and standalone mode. In dynamic loading mode, a base system with application loader is running on the intelligent brick. Application is separately built as a module (ELF file in fact) which can be dynamically loaded from micro SD card or Bluetooth and then executed under non-privileged mode on the base system. On the contrary, application in standalone mode will be compiled and linked together with the whole platform, to generate a boot image supported by the EV3's bootloader U-Boot [12]. Both modes have their own merits. Application in dynamic loading mode can be updated without rebooting EV3, which makes the development process very smooth. Its building is also much faster than in standalone mode since compilation of the whole platform is avoided. Robot control applications are usually updated frequently to adjust control logic and parameters, and thus we highly recommended dynamic loading mode to be used. On the other hand, application in standalone mode can be directly booted by EV3 and executed under both privileged mode and non-privileged mode. It is useful for building a platform or firmware as developer has full access to RTOS kernel and hardware resources in this mode. In fact, the base system of dynamic loading mode itself is an application in standalone mode. mruby on EV3RT + TECS [45], a platform for programming EV3 in mruby, is also developed in this mode.

EV3RT uses a layered architecture that decouples user application from the infrastructure software as shown in Fig.2, in order to achieve high flexibility, reliability and extensibility. This architecture is mainly applied to dynamic loading mode but can also be considered as a reference model for standalone mode. It consists of three layers as follows:

Core Services Layer: This layer is to provide services needed by applications and monitor the

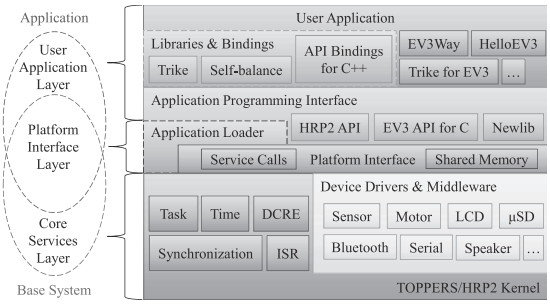


Fig. 2 Architecture of EV3RT

Fig. 3 Screenshot of EV3RT console

running application. It mainly consists of TOPPERS/HRP2 kernel, device drivers and middleware. TOPPERS/HRP2 kernel is a static RTOS kernel with protection functionalities. EV3RT uses it to provide system services with high reliability. Potential defects and bugs in user application will not harm the platform. Instead, they are monitored and handled by this layer and will be logged to provide information for debugging. Device drivers and middleware are modules to support various features of EV3, such as PWM (pulse-width modulation) motor control and file system. They are mainly running under privileged mode since they usually wish to manipulate hardware devices directly. EV3RT console, a user interface for selecting application to launch or viewing system logs as shown in Fig.3, will show at startup after all services have been initialized.

Platform Interface Layer: This layer acts as an interface between core services layer (CSL) and user application layer (UAL). It specifies a list of functions that must be implemented in CSL. These functions are wrapped as service calls so that they can be called from user application which runs under non-privileged mode. There are some service calls to obtain the pointer of a shared memory area. Shared memory can be used as a simple interface to eliminate the overhead of calling the same service call repeatedly. Examples of shared memory include read-only sensor data and read-writable LCD frame buffer. Data structures and macros shared between CSL and UAL are also defined in this layer. User applications are only dependent on this layer so they can be compiled and linked as loadable modules without any detail and code of CSL. Modifications to source code of UAL are unnecessary as long as this layer does not change, even if implementation of CSL changed a lot. Therefore, platform interface layer can be considered as the application binary interface (ABI) of EV3RT and is given a version number called PIL version. This layer also includes an application loader which can be used to update the running application dynamically. On launching an application, the loader checks the PIL version of the base system and the application at first to make sure they are compatible (with the same version). The running application can be manually terminated using EV3RT console, and if it is crashed, the loader will automatically unload it.

User Application Layer: This layer consists of software components for building user applications and the user application itself. Application programming interfaces (APIs) for C language are provided, which will be introduced later in the next subsection. Runtime and API bindings for C++ have been implemented to support object-oriented programming. In addition, libraries for common robot functionalities such as self-balancing are also available. Some sample applications are included to help developers get started with EV3RT. All code in this layer runs under non-privileged mode inside a determined protection domain called `TDOM_APP`, which will be explained later in Section 3. Currently, only one user application can be running at the same time on EV3RT.

2.2 Application Development in EV3RT

Users can develop applications for EV3RT under Linux, Mac OS X or Windows. EV3RT uses GNU Tools for ARM Embedded Processors [5] as its toolchain. All the required software packages can be easily installed by following the guide on our website [16] [15]. Installation of EV3RT to the intelligent brick can be done by just putting boot image of the base system, which can be built with one sin-

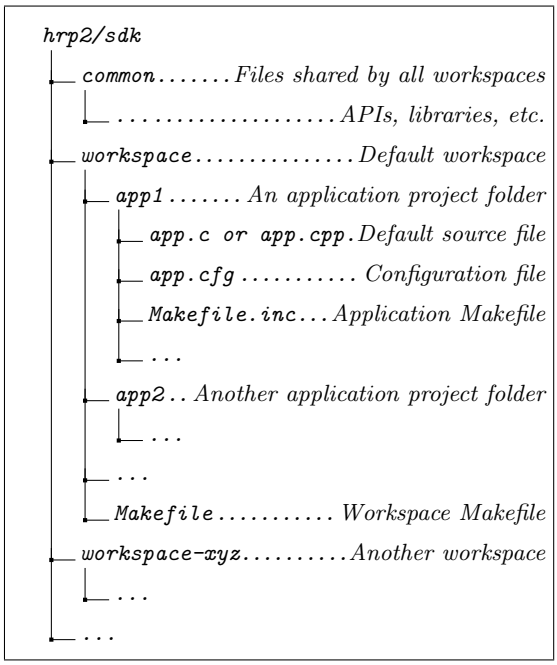


Fig. 4 Structure of the sdk folder

gle command, into the root directory of EV3's microSD card. In this subsection, some important information about developing applications in EV3RT are described.

2.2.1 Application Project Management

In EV3RT, user application projects are managed inside the `sdk` (short for software development kit) folder. The `sdk` folder includes a `common` folder holding common files such as source files of APIs and libraries, a default workspace folder named `workspace` and optional workspace folders which can be created by users, as shown in Fig.4. A workspace folder is a folder used to hold multiple application projects. Each application project corresponds to a dedicated folder under some workspace folder. All the sample applications can be found in the default workspace.

With the workspace feature, application projects can be managed in an easy way. A new application project can be simply created by duplicating a existing project folder with a new name. Deleting a project is also straightforward: just remove the folder. An application can be built in two modes, as mentioned above, with its folder name. For example, for an application project folder named `app1`, developer can build it in dynamic loading mode by executing `make app=app1` command under its workspace folder. After building successfully, a loadable module named `app` will be generated. Further, developer can also use `make img=app1` to build it in standalone mode. In that case, a boot image file named `uImage` will be generated.

```
APPL_COBJS += balancer.o \
    balancer_param.o

APPL_CXXOBJS +=

SRCLANG := c++

ifdef CONFIG_EV3RT_APPLICATION

# Include libraries
LDIR = $(EV3RT_SDK_LIB_DIR)
include $(LDIR)/libcpp-ev3/Makefile

endif
```

Fig. 5 Example of application-specific Makefile

An application project has three files by default, `app.c` (or `app.cpp` when developing in C++), `app.cfg` and `Makefile.inc`. `app.c` (or `app.cpp`) is the default source file of an application which will be compiled automatically. `app.cfg` is the configuration file of an application. An application can be built in dynamic loading mode, only if all kernel objects defined in `app.cfg` belong to the predetermined application protection domain `TDOM_APP`. `Makefile.inc` is the application-specific Makefile. Application build configuration such as source files and programming language (only C and C++ language are supported currently) can be customized by modifying this file. See Fig.5 for an example of `Makefile.inc`.

2.2.2 Development Process

The process to develop an application in standalone mode is listed as follows.

1. Power on the intelligent brick
2. Write code for the application
3. Generate the boot image `uImage`
4. Connect EV3 to PC with a USB cable
5. Copy `uImage` to the root of the microSD card
6. Restart the intelligent brick
7. The new application will get executed
8. Application needs modification, go to step 2

It should be noted that every time the application is modified, it is required to reboot the EV3 brick and write the microSD card with a PC. This procedure is a bit annoying and, if the application uses Bluetooth, the need of reconnection will make it even worse. Therefore, the standalone mode is not recommended except for developing a firmware or platform.

Developing an application in dynamic loading mode can be much more smooth. After the EV3 is booted, EV3RT console will show up on the display. Developers can use the loader in EV3RT console to receive and execute a new application wirelessly transferred from Bluetooth. Besides, applications can also be put into microSD card for execution via USB. If an application is running, user can press the back button for about 0.5 second to show the EV3RT console to terminate it and load a new application. The process is listed as follows.

1. Power on the intelligent brick
2. Write code for the application
3. Generate the loadable application module
4. Upload application via Bluetooth or USB
5. Start the new application
6. Application needs modification, go to step 2

2.2.3 APIs for User Application

EV3RT currently provides three types of application programming interface (API) as follows:

TOPPERS/HRP2 Kernel API: This API allows user applications to access real-time operating system services provided by HRP2 kernel. It includes μITRON-like [48] static APIs and service calls, whose details are described in the TOPPERS new generation kernel specification [46]. Static APIs are used in the configuration file to configure the kernel objects of an application statically. On the other hand, service calls, which may be called system calls in other operating systems, are used in source files to request services at run time. In the HRP2 kernel specification, cyclic handlers can be only created in the kernel domain, which run in interrupt context under privileged mode without protection. Since cyclic handlers are also very useful in user applications, EV3RT extends the original kernel API with user domain cyclic handler functionality.

C/C++ Standard Library: Newlib [36], a C/C++ standard library implementation for embedded systems, has been ported to EV3RT. With the standard library, users can develop applications for EV3RT as easily as developing a normal C/C++ application. The support of dynamic memory management is required for most functions in Newlib. However, the default memory allocator in Newlib only supports one single heap, which is not compatible with the protection model of HRP2 kernel, and can not provide any real-time guarantee. We replaced it with the TLSF (Two-Level Segregated Fit) memory allocator [32], a memory allocator designed for real-time embedded systems. Since memory objects in HRP2 kernel are configured statically, a fixed-size memory pool is preallocated for Newlib. EV3RT also integrates communication services into the standard file operations. Special files for communication devices such as Bluetooth or serial port can be obtained. Developers can use these files to do communications just like accessing a file, by calling functions such as `fprintf()` and `fgetc()`.

EV3RT C Language API: This API provides C language functions to support hardware devices such as sensors and other platform-specific features. Since it is used to control the robot actually, most of its functions provide high real-time performance, except those involving microSD card operations. The supported devices and features are listed as follows:

- EV3 brick (buttons, LED, LCD and speaker)
- Memory file and microSD card
- Bluetooth and serial port
- Servo motors and rotary encoder
- Various sensors (ultrasonic, color, etc.)

These APIs allow users to develop applications easily without any knowledge about the implementation details of core services layer (e.g. device drivers) in EV3RT. If a developer wants to access device drivers (including middleware) directly, the standalone mode must be used, since device drivers run under the privileged mode. In that case, the developer will also need to learn the usage of device drivers (for example, from their documents).

Besides above APIs, optional static libraries like API bindings for C++ are also available.

2.2.4 Sample Applications

Several sample applications come with EV3RT to help developers get started. Two representative samples are introduced as follows:

`HelloEV3` is a program which can be used to test the features of EV3RT thoroughly. It provides a simple graphical user interface which can be operated with buttons on the intelligent brick. This graphical menu contains tests for all functions provided by EV3RT C language API and itself is also implemented with those functions. Further, user is allowed to reconfigure the connection of devices such as sensors and motors dynamically, without recompilation or reboot. With this sample, the implementation of EV3RT can be checked easily and developers can learn how to use EV3RT APIs by referring to its source code.

`EV3Way` is a sample application to control a two-wheeled self-balancing robot. The researches on two-wheeled self-balancing robot (a.k.a two-wheeled inverted pendulum mobile robot) have gained momentum over the past decades and shown that high real-time performance is required [35].

This application is developed by the executive committee of ET Robocon in C++ to control a robot called EV3Way-ET. The reference construction of EV3Way-ET is available on GitHub [14]. There are two tasks, the balancing task and the communication task, in this application. The balancing task runs in a high priority to control the robot with the self-balancing algorithm and the communication task runs in a lower priority to communicate with user via Bluetooth. Both of the tasks can work stably, which shows that EV3RT is suitable for developing applications with high real-time requirements.

3 TOPPERS/HRP2 RTOS Kernel

TOPPERS/HRP2 kernel (or just HRP2 kernel) is used by EV3RT as its RTOS kernel. It is a μITRON-like [48] open source RTOS kernel created by the TOPPERS project [47]. HRP2 is short for High Reliable system Profile version 2, and by the way the first version of HRP kernel has been adopted by the HII-A and HII-B rockets [21] and proven its excellent real-time performance and reliability.

Since nxtOSEK/JSP [41], a popular RTOS platform for NXT, also uses RTOS kernels from the TOPPERS project, developers of NXT should find it easy to migrate to EV3 with EV3RT. Aside from the consideration of migration, the most important reason for choosing HRP2 kernel is that it is a static RTOS supporting various protection functionalities. Mindstorms EV3 is a relatively large-scale embedded system, and if its software platform is not protected, a bug in user application may easily break the whole platform, which makes debugging very difficult.

In this section, we focus on explaining how features of HRP2 kernel are applied to make EV3RT a reliable software platform. For a thorough description of HRP2 kernel, check the TOPPERS new generation kernel specification [46].

3.1 Static Configuration Approach

A static operating system kernel is a kernel whose kernel objects (or resources), such as tasks or memory areas, are configured at design time and statically allocated at compile time. Both EV3RT and the user applications on it are developed following this static configuration approach.

In HRP2 kernel, developers statically configure the kernel by writing configuration files using static APIs. See Fig.6 for an example of configuration file. A tool called configurator will parse these configuration files. By using the parsing results as input, the configurator interprets some template files to generate necessary C source files or linker scripts. Template files are written in a template language defined by HRP2 kernel for generating files. Common template files such as generating source files for kernel objects are included in HRP2 kernel. Target-dependent template files like generating linker scripts or translation tables to support memory protection are also available for many popular targets. However, the processor of EV3 was not supported by HRP2 kernel, and thus we implemented those target-dependent template files for it.

Research has shown that a static OS design is more reliable and can provide a significantly better resilience to soft errors than the dynamic ones [20], as many potential software defects can be found during the configuration process. Besides, a static OS also tends to have higher and more stable performance and consume less memory.

3.2 Kernel Object Access Control

A kernel object is a system resource managed by the RTOS kernel. Kernel objects are defined in and will be generated from configuration files. Each

```
KERNEL_DOMAIN {
  /* create a system task */
  CRE_TSK(MAIN_TASK,{ TA_ACT, 0, main_task, 6, 1024, NULL });
  /* register a memory object */
  ATT_MOD("sample_kernel.o");
  /* define an extended service call */
  DEF_SVC(SVC1, { TA_NULL, svc1_entry, 64 });
}
/* a user domain named DOM1 */
DOMAIN(DOM1) {
  /* create a user task */
  CRE_TSK(TASK1, { TA_ACT, 0, task1, 6, 1024, NULL });
  /* register a memory object */
  ATT_MOD("sample1.o");
}
/* a user domain named DOM2 */
DOMAIN(DOM2) {
  /* create a user task */
  CRE_TSK(TASK2, { TA_NULL, 0, task2, 6, 1024, NULL });
  /* create a semaphore named SEM1 */
  CRE_SEM(SEM1, { TA_NULL, 0, 1 });
  /* configure access rights of SEM1 */
  SAC_SEM(SEM1,{TACP(DOM2),TACP(DOM1)|TACP(DOM2),TACP(DOM2),TACP(DOM2)};
}
/* define a shared memory object with access rights configured */
ATA_SEC(".appheap", { TA_NULL, "RAM" }, \
    { TACP_SHARED, TACP_SHARED, TACP_SHARED, TACP_SHARED });
```

Fig. 6 Example of a configuration file

kernel object created will be associated with an ID which can be used to access it. Tasks, semaphores and data queues are typical examples of kernel objects.

The concept of protection domain is introduced to support access control of kernel objects. There are two types of protection domains in HRP2 kernel: kernel domain and user domain. The kernel domain is unique while there can be multiple user domains. A task must belong to some protection domain to determine its permissions. Tasks in the kernel domain, called system tasks, are executed under privileged mode which has full access rights to all kernel objects. Tasks in a user domain, called user tasks, are executed under non-privileged mode with limited access rights. Other runnable kernel objects, such as alarm handlers, are always executed under privileged mode and thus must belong to the kernel domain. A protection domain is the finest granularity to grant access rights of a kernel object, which means that tasks in the same protection domain have the same access rights. By default, a kernel object allows user tasks in the same protection to access it. A non-runnable kernel object like semaphore can also be shared by all protection domains without belonging to one of them, which allows any task to access it.

In order to configure access permissions more flexibly, operations to each type of kernel objects are divided into four groups: type 1 operations, type 2 operations, management operations and reference operations. For example, in the case of semaphores, type 1 operations are for signaling, type 2 operations are for waiting, management op-

erations are for configuring access rights, and reference operations are for acquiring status. The access rights of each group of operations for a kernel object can be configured individually.

In EV3RT, the core services layer works in the kernel domain while all kernel objects in the user application layer belong to a determined user domain called `TDOM_APP`. The core services layer does not allow user application to access its objects directly. Instead, service calls and shared memory defined in the platform interface layer must be used. In this way, kernel objects in the base system are protected from illegal or undesired access caused by the user application.

3.3 Memory Protection Support

Memory protection is necessary because user application can still harm the base system by operations such as writing improper values to a hardware register or changing data in the kernel, even if kernel objects are protected. What is worse, if user application breaks the file system, all data on the microSD card might be corrupted. As a high reliable RTOS, HRP2 kernel does support memory protection, with static configuration of memory objects.

In HRP2 kernel, a memory object represents an area of memory controlled by the kernel. Lots of information is associated with a memory object, including base address, size and attributes of that area, and access rights granted to protection domains. An attribute is a property that always holds regardless of which protection domain it is accessed from. For example, if a memory object has the attribute `TA_NOWRITE`, its memory area cannot be written even by a system task. There are also attributes like `TA_UNCACHE` to control cache behavior. If base address and size of a memory object is fixed, such as a hardware register, developer can use the `ATT_MEM` static API to create it explicitly. In the case of text and data section in an object file whose base address and size are unknown at design time, the `ATT_MOD` static API can be used to generate memory objects from an object file with specified access rights. Configuring attributes and permissions for a named section defined in the source code is also supported, by using the `ATT_SEC` static API with the section name. Memory objects should never overlap, otherwise an error message will show up during configuration.

Although HRP2 kernel is designed to support memory protection with both memory protection unit (MPU) and memory management unit (MMU), the ARMv5 MMU used by EV3 was not supported. In our previous work [29], we have supported the ARMv5 MMU by designing an algorithm to generate ARMv5 page tables for each protection domain, and proposed a method to optimize the TLB flushing overhead. With that optimization, ultra-low latency of context switching can be achieved.

For memory objects in the base system of EV3RT, only those defined as shared memory in the platform interface layer can be accessed from the user application layer. Most of the shared memory areas are read-only, such as sensor values and font data. Only memory areas that will not break the base system even if they are corrupted, like LCD frame buffer for user application, can be configured as read-writable shared memory. The base system has its own frame buffer which is protected from user application, for displaying interface of core services like EV3RT console. In this way, shared memory can be used as an efficient and safe method of passing data in EV3RT.

3.4 Extended Service Call

Although shared memory is very efficient, it is not enough because most user applications will also wish to execute code under privileged mode, such as using PWM device to control a motor, in a safe way. HRP2 kernel provides the extended service call functionality to support this case.

The term service call in HRP2 kernel has a similar meaning to system call in Linux. Service calls act as an interface between user domains and the kernel. This interface is essential to provide system services for a user task which is usually prevented from directly manipulating the kernel's memory. When a service call is called, it is executed under privileged mode and aware of the caller's protection domain, so illegal operations can be blocked by checking the access rights.

As a general-purpose operating system, Linux has a stable system call interface, and device drivers are abstracted as special files to be accessed with file I/O system calls. Access permissions of device drivers in Linux are also controlled using file

```
KERNEL_DOMAIN {
DEF_SVC(TFN_MOTOR_COMMAND,{ TA_NULL,
  extsvc_motor_command, 1024 });
}
```

a. define extended service call in configuration file

```
ER_UINT extsvc_motor_command(
  intptr_t cmd, intptr_t size,
  intptr_t par3, intptr_t par4,
  intptr_t par5, ID cdmid) {
  ...
}
```

b. implement service call in kernel domain

```
ercd = cal_svc(TFN_MOTOR_COMMAND,
  cmd, size, 0, 0, 0);
```

c. call service call from user domain

Fig. 7 Example of extended service call

system. Although this approach brings excellent portability, it also introduces overhead of the file system which influences the real-time performance. In HRP2 kernel, instead of providing a universal interface for device drivers, the functionality of defining new service calls, called extended service calls, conveniently is supported.

Fig.7 is an example to show how extended service calls are used in device drivers and middleware of EV3RT to provide services safely. Firstly, a new service call is defined with the static API `DEF_SVC`. A unique number (`TFN_MOTOR_COMMAND` in the example), called function code, is associated with each service call. The function name and stack size for the service call must also be specified. Thereafter, function body of the service call, which will be executed under privileged mode, is implemented. The function has 6 parameters: 5 of them are passed from the caller and the last one (`cdmid`) holds the protection domain ID of the caller. These parameters are carefully checked and an error code will be returned if any problem occurs. In user application, the `cal_svc()` API is used to call a service call with its function code and arguments. The context will enter privileged mode to execute corresponding function and switch back with return value after the function has completed.

3.5 Task Priority Protection

With above policies applied, the space of base system can be protected from defects in user application. However, a user application can still influence the base system in the aspect of CPU time. HRP2 kernel uses priority-based preemptive scheduling algorithm. If a user application changes the priority of a task, for example, of an infinite loop to the highest priority, the intelligent brick will be stuck forever until a physical power-off is performed, because all tasks in the base system cannot be scheduled.

In order to solve this issue, we added the `LMT_DOM` static API to the HRP2 kernel specification. This API can limit the highest priority that can be set at run time for tasks in a protection domain. It does not limit the initial priority of tasks since they can be checked during configuration, which allows developers to control the system more flexibly. EV3RT uses this API to limit task priority in `TDOM_APP`, and thus the base system will not be preempted by tasks in user application.

Besides, Bluetooth service in the base system has another kind of issue on task priority. The CPU utilization of task to process Bluetooth packets depends on the speed of traffic. If the task has higher priority than user application and packets come in a very high rate, it will have a negative effect on the real-time performance of user application. On the other hand, if the task has lower priority than user application, it may never get scheduled, which will break the Bluetooth service. EV3RT copes with this issue by using a QoS (quality of service) task to control the priority of Bluetooth task dynamically. The QoS task will periodically raise the priority of Bluetooth task, which usually works in the lowest priority, to a priority higher than user application for a short time.

By default, the Bluetooth task will run in high priority for 1 ms in every 20 ms. These timing parameters are decided by an experimental approach to find the minimum required CPU resource for a relatively stable Bluetooth connection. During the experiment, a task of infinite busy loop is running as user application. The exection time of Bluetooth task in high priority is fixed at a single system tick, which is 1 ms. We then change the period to control CPU resource given to the Bluetooth task, and check whether the Bluetooth connection is stable.

A connection is considered to be stable if it can complete pairing process and stay connected for over one hour. After having tested and failed with 100 ms (≈1% CPU), 50 ms, 30 ms, 25 ms periods, we found that 20 ms period can provide a stable connection. Meanwhile, as a representative application with high real-time requirements, the balancing task of `EV3Way` has an execution time of 1 ms with 5 ms period. Therefore, Bluetooth service can be kept alive with this policy while has very limited influence on real-time performance. Users are also allowed to change the timing parameters of QoS policy or disable it as necessary.

4 Dynamic Module Loading

Dynamic updating mechanisms of software system or applications without reboot are becoming significantly necessary to reduce the time to market and development costs, with the rapid growth of the complexity of embedded systems. System reboots can be slow and disruptive and will drop all current status like network connections, which are very expensive for some safety-critical applications such as avionics [40]. For Linux-based software platforms, new applications can be transferred and executed on-the-fly within a few seconds via network protocols such as Secure Shell (SSH) [51]. While the static design of HRP2 kernel provides high robustness, the lack of dynamic updating support can be one of its main drawbacks.

In this section, we present a dynamic module loading mechanism for HRP2 kernel. The application loader in EV3RT is implemented by referring to this mechanism. Main characteristics of our original mechanism are listed as follows:

Excellent portability. The design of our mechanism, per se, does not depend on any specific hardware, although the motivation is to support dynamic application loading in EV3RT. It is able to support most of targets where HRP2 kernel works. Further, developers can apply it to existing systems very easily, without implementing any complex function such as dynamic memory or page table management.

Keep it simple, static. Our mechanism is to support dynamic updating of modules rather than to support development in a dynamic way. HRP2 kernel benefits a lot from the simplicity of its static design, especially in the aspects of performance and robustness, and our design is able to keep those advantages. All loadable modules are still configured and developed in the same way of developing a normal HRP2 application, which is static, with some restrictions and minor changes. Thus, resources required by a module are statically determined at compile time and can be checked at load time.

Reliable and fault-tolerant. Modules may contain bugs and could lead to incorrect or unexpected behaviors after loaded. Our mechanism has the feature to guarantee that the modules loaded will not harm the whole system. The loader will verify the configuration information of a module before loading, and the unloading of a module can be performed in an elegant way too. Further, protection functionalities introduced in previous section can also be applied to save the system from run-time failures of modules.

High real-time performance. High real-time performance is required as a dynamic module loading mechanism for hard real-time operating systems like HRP2 kernel. Since loadable modules in our mechanism are developed in a static way and generated as native code, the real-time performance can be easily guaranteed after loaded. Moreover, loading and unloading operations are provided as service calls working under task context and will only lock the CPU for a very few cycles. That is to say, using our mechanism has very little influence on the real-time performance of the whole system.

4.1 Dynamic Creation Extension

Although HRP2 kernel requires all kernel objects and resources to be allocated statically according its static design, it does provide a kernel extension, called Dynamic CReation Extension (DCRE), that allows kernel objects to be dynamically created or deleted at run time.

DCRE still follows the static methodology of HRP2 kernel by using the object pool design pattern [39]. The maximum number of each kind of kernel object that can be created must be predefined in configuration files at design time. Objects pools for these kernel objects will be generated at compile time and the resources (e.g. data structures like task control blocks) required by them will be allocated statically. During the startup process of kernel, all kernel objects in pools will be set to

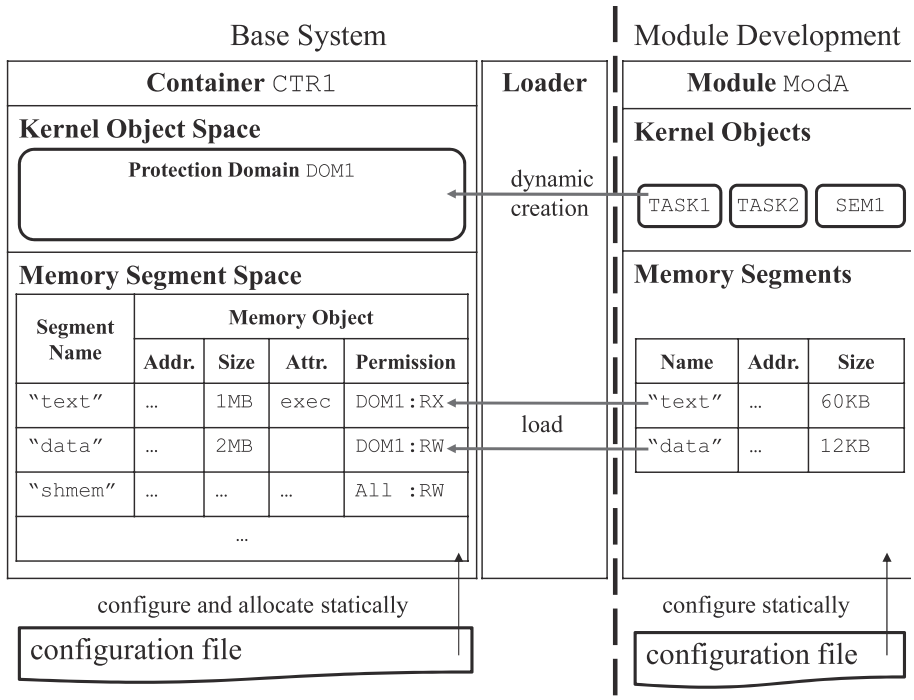


Fig. 8 Proposed Dynamic Loading Mechanism

the 'not in use' state (or the invalid state). An object can be created by obtaining a 'not in use' object from the pool and initializing it to 'in use' state (or the valid state). When an object created dynamically is no longer needed, it can be deleted and returned to its pool by setting back to the 'not in use' state with necessary clean-up process performed. In this way, kernel objects can be created and deleted without dynamic allocation of memory or other resources.

DCRE is designed in the spirit of minimality principle that it does not depend on any prediction of hardware or application. Consequently, the mechanisms for dynamic memory management are not defined by DCRE. However, dynamic memory management is necessary for dynamically creating tasks whose stacks have different size, address and access permissions which can not be determined and allocated statically. Thus, dealing with dynamic memory management is required for designing dynamic module loading mechanisms based on DCRE.

4.2 Design of Proposed Mechanism

In this subsection, design of the reference model of our dynamic module loading mechanism is described. Our mechanism mainly consists of three kinds of concepts: loadable modules, containers and the loader. Fig.8 provides an overview of these concepts.

Loadable modules are developed by module developers and can be built as binary files containing information for loading and execution. Containers, whose protection policy, memory layout and resource limit can be configured, are kernel objects used to hold a loadable module. The loader is a system service that is responsible for checking and placing a loadable module into a container, preparing it for execution and unloading it when it is no longer needed. System developers can create and configure containers in the base system and use the loader to load module into specific container dynamically. The reference model of these concepts is explained in following subsections.

4.2.1 Loadable Module

A loadable module (or just module) is a software component that can be dynamically installed into

the existing system to provide extended features such as new services, applications or even device drivers. A module installed can be unloaded in order to free the resources allocated to it such as memory, when the functionality provided by it is no longer required.

Logically, a module is a collection of kernel objects and data. All the kernel objects must be defined in a configuration file of the module at design time. Module developers are not allowed to perform any dynamic creation or deletion of kernel objects during the execution of a module. In this way, the resources required by a module can be statically determined at compile time and will be provided by the loader at load time.

Unlike similar concepts in general-purpose operating systems such as loadable kernel modules (LKM) in Linux [37] or kernel-mode drivers in Windows NT [7], which are running in privileged mode without protection, our modules are supported to work in both privileged mode and non-privileged mode with protection.

In order to achieve high reliability while keeping the simplicity, there are some constraints on the development of modules:

Kernel objects must belong to the same protection domain. Access permissions of kernel objects are granted in the granularity of protection domain. A module will be loaded into a container, which is associated with a single protection domain, in the base system for execution. Permissions granted to a module depend on the protection domain of its container. Therefore, all kernel objects in a module must be configured to belong to the same protection domain at design time. The protection domain of a module is usually omitted, in which case the protection domain of container will be used at loading. If the protection domain is specified explicitly, the base system is responsible for choosing a suitable container. It should be noted that the protection domain of a module must not be decided freely by the module developers themselves, which can make any protection policy become meaningless. If the base system is intended to support explicitly specified protection domain of a module, verification schemes such as storing digital signatures in the module for checking trusted developers should be introduced at the implementation level of our mechanism. The loader in EV3RT does not support signature verifying since its user applications will only be loaded into the predetermined protection domain (i.e. `TDOM_APP`).

Configurations of memory objects are not allowed. Implementing dynamic memory management with protection is not required to support our mechanism for the simplicity of porting. All the memory protection information such as page tables will be statically generated at compile time of the base system. Since configuring memory objects may result in changing of the memory protection information, it is not allowed in the development of modules. Instead, we provide an approach, which will be described later in the section of container, to controlling memory protection for a module by putting memory contents into segments with access permissions specified by a container.

Dynamic creation extension (DCRE) cannot be used. The main purpose of DCRE is to support developing middlewares such as a dynamic loader, rather than to support developing user applications in a dynamic way. Therefore, DCRE does not provide fine-grained access control on managing objects. That is to say, if a protection domain has the permission to use DCRE, tasks in that domain can create objects belonging to any protection domain even the system domain, which may be used to bypass existing protection policy. Further, the allocation of kernel object pools will introduce configurations of memory objects implicitly, which will violate the second constraint. Hence, while the implementation of our mechanism is based on DCRE, using dynamic creation extension is not supported in the development of modules.

Physically, a module is an object file that contains the module information and multiple segments with code or data. The module information consists of a module configuration table and other implementation-specific or application-specific information. An example of the module configuration table is shown Fig.9. Each entry in the module configuration table corresponds to a configuration entry in the configuration file, which represents a static API call such as `CRE_TSK`. An entry in the module configuration table has three members: a static API code, a pointer to the argument of that call and a pointer to store the return value. The loader will create kernel objects according to the

```
// Content in the configuration file of a module:
// CRE_TSK(TASK1,{TA_ACT,0,task1,TMIN_TPRI,STACK_SIZE,NULL});

ID _module_id_TASK1 __attribute__((section(".module.text")));
static STK_T _module_ustack_TASK1[COUNT_STK_T(STACK_SIZE)];

/* Structure of information used to create a task */
typedef struct t_ctsk {
    ...
} T_CTSK;

/* Array of configuration information for tasks */
static const T_CTSK _module_ctsk_tab[1] = {
    { TA_ACT, 0, task1, TMIN_TPRI, ROUND_STK_T(STACK_SIZE),
        _module_ustack_TASK1, DEFAULT_SSTKSZ, NULL },
};

/* Structure of a module configuration entry */
typedef struct {
    FN          sfncd; /* Static API code */
    const void *argument;
    void       *retvalptr;
} MOD_CFG_ENTRY;

/* Array of a module configuration entry */
const MOD_CFG_ENTRY _module_configuration_entries[] = {
        { TSFN_CRE_TSK, &_module_ctsk_tab[0],
        &_module_id_TASK1 },
};

/* Structure of the module configuration table */
typedef struct {
    const MOD_CFG_ENTRY *entries;
    const SIZE           entry_number;
} MOD_CFG_TAB;

/* The module configuration table */
const MOD_CFG_TAB _module_configuration_table =
    { _module_configuration_entries, 1 };
```

Fig. 9 Example of a module configuration table

static API code and the argument. The return value is used to pass information to the module. For example, in Fig.9, the ID of TASK1 is unknown at compile time and will be set by the loader through the return value pointer after the kernel object of TASK1 is created.

The memory contents inside a module are divided into segments. There are two segments required to support our mechanism, the text segment and the data segment. The text segment is used to store

```
DML_CRE_CTR(CTR1, DOM1, "text", "data");
DML_ATA_SEG(CTR1, "text", { TA_EXEC, NULL, 1 * 1024 * 1024, NULL }, \
    { TACP(DOM1), TACP_KERNEL, TACP_KERNEL, TACP_KERNEL });
DML_ATA_SEG(CTR1, "data", { TA_NULL, NULL, 2 * 1024 * 1024, NULL }, \
    { TACP(DOM1), TACP(DOM1), TACP_KERNEL, TACP_KERNEL });
```

Fig. 10 Example of configuring a container and its segments

instructions and read-only data, and the module is not permitted to write to the memory space specified by it. The module information must also be placed into the text segment for safety concerns. Static variables such as global variables and static local variables are contained in the data segment which is writable by the module. Stacks of tasks are generated as static variables and hence will be put into the data segment, as shown in Fig.9. A module may include other segments, specified by its container, like shared memory segment for finer memory management. Most compilers support to place specific memory contents into a particular segment. For example, the GNU Compiler Collection (GCC) can use section attribute to do this [18].

4.3 Container

Dynamic resource allocation is not supported in HRP2 specifications: all resources must be configured at design time and allocated at compile time for generating the base system. This means that the available resources of a module must also be defined statically at the design time of the base system. A new concept called container is introduced to support this.

A container is a kernel object acting as the placeholder of a module. Configuring a container allows system developers to control the characteristics of modules loaded into it such as their protection policy, memory layout and resource limit (e.g. the maximum number of each kind of kernel objects that can be created). We define three kinds of static APIs to configure a container:

DML_CRE_CTR(ctrid, domid, text_segment_name, data_segment_name): A container can be created with a specific container ID (`ctrid`) using this API. It is assocaited with a protection domain (`domid`). All kernel objects created by loading a module into the container will belong to that domain. In such a way, system developers can control the access permissions of a module by configuring the protection domain of its container. This API also specifies the name of text segment (`text_segment_name`) and data segment (`data_segment_name`) in a module. These two segments are mandatory for a module.

DML_ATA_SEG(ctrid, segment_name, { mematr, base, size, paddr }, { acptn1, acptn2, acptn3, acptn4 }): A memory segment can be attached to a container (`ctrid`) using this API. Each segment in the same container must have a unique name (`segment_name`) which can be used in the development of modules. A memory object with specific attributes (`mematr`), address (`base` and `paddr`), size (`size`) and access permissions (`acptn1-acptn4`) will be created for a segment. In this way, the memory protection information of each segment can be configured at design time of the base system. See the TOPPERS new generation kernel specification [46] for details about configuring a memory object (`ATA_PMA`). It means that the memory area for a segment is preallocated at compile time and has a maximum size. When loading a module into a container, segments in that module will be loaded to corresponding memory area with the same segment name. See Fig.10 for an example of configuring segments with this API. In this example, the text segment is 1MB size, executable (`TA_EXEC`) and read-only to `DOM1` which is the protection domain of its container. The data segment is 2MB size and both readable and writable to `DOM1`. It should be noted that the addresses (both virtual and physical) of these segments are omitted in the example. In this case, the base addresses of text and data segment will not be fixed and loadable module must be compiled to position-independent code that data segment has an unfixed offset with text segment [38]. By this

API, the memory layout with protection information of modules supported by a container can be configured flexibly and the required resources for a container to load a module will be allocated statically.

DML_MAX_yyy(ctrid, max_number): For each kind of kernel objects, their maximum number inside a container can be limited by this series of static APIs. For example, `DML_MAX_TSK(CTR1, 10)` means that the container `CTR1` can only load a module with no more than 10 tasks. Using this API is optional, and if a kind of kernel object is not limited by this API, its maximum number will depend on the status of the kernel object pool.

4.4 Loader

The loader provides services to load a loadable module into a container and unload a module when it is no longer needed. The process of loading a module is listed as follows:

1. Verify the loadable module
2. Copy segments into the container
3. Perform relocation when necessary
4. Do hardware-dependent post-processing
5. Create kernel objects and resolve their IDs

The resources required by the module to be loaded can be known from its configuration table. The loader must verify it before doing any actual work for loading. If the module has a memory segment whose name does not exist in the segments of the target container, it cannot be loaded into that container. The size of every segment in the module must not exceed the maximum size of the corresponding segment in the container too. If the container has limits on the number of kernel objects that can be created, it must also be checked. It must be noted that the creation of kernel objects may still fail, depending on the status of the base system (e.g. available resources in kernel object pools). The loader should also make sure the module information is located in the text segment since it will be used for unloading and must not be changed by the module.

After verification, the loader copies each memory segment of the module into the corresponding segment in the container. Since memory for segments of containers is allocated statically, no dynamic memory allocation is needed in this step.

The base address of segment may be not fixed, according to the configuration of container. The loader must relocate the module if its segments have different base address with the container's. In this case, the module must include the relocation table created by the compiler [28].

Modifications of text segment, including copying and relocating, involve indirect self-modification code, and may require post-processing on some hardware. For example, on ARM architecture [3], data and instruction cache are isolated. The cache synchronization such as flushing data cache and invalidating instruction cache for the modified text segment must be explicitly performed by the loader before further processing.

After handling memory segments, the loader can finally create kernel objects according to the configuration table of the module. For each kernel object created, the loader will pass its actual ID to the module through the return value pointer in the module configuration entry. The implementation of loader must guarantee atomicity for this step: if a kernel object is not created successfully, all kernel objects created previously must be deleted before any code in the module is executed.

IDs of all the kernel objects created for a module are stored in its configuration table, which is read-only to the module, after loaded. If the module is no longer needed, the loader can safely unload it by using information in that table to delete all its kernel objects.

5 Application Loader in EV3RT

Application loader in EV3RT is developed as a prototype implementation of the proposed dynamic module loading mechanism. It can dynamically load application from microSD card or Bluetooth without rebooting EV3. User can also put new applications into microSD card for loading by connecting EV3 to PC with a USB cable.

Fig.11 shows the configuration of the application container in EV3RT. In EV3RT, the application container `APP_CTR` is associated with the application domain `TDOM_APP`. It has only two segments, the text segment and the data segment. Both of these segments do not have a fixed base address so relocation is required. The text segment is set to be read-only for all user domains (`TACP_SHARED`), not

```
DML_CRE_CTR(APP_CTR, TDOM_APP, "text", "data");
DML_ATA_SEG(APP_CTR, "text", \
    { TA_EXEC, NULL, TMAX_APP_TEXT_SIZE /* 1MB */, NULL }, \
    { TACP_SHARED, TACP_KERNEL, TACP_KERNEL, TACP_KERNEL });
DML_ATA_SEG(APP_CTR, "data", \
    { TA_NULL, NULL, TMAX_APP_DATA_SIZE /* 2MB */, NULL }, \
    { TACP_SHARED, TACP_SHARED, TACP_KERNEL, TACP_KERNEL });
DML_MAX_TSK(APP_CTR, TMAX_APP_TSK_NUM /* 32 */);
DML_MAX_SEM(APP_CTR, TMAX_APP_SEM_NUM /* 16 */);
DML_MAX_FLG(APP_CTR, TMAX_APP_FLG_NUM /* 16 */);
DML_MAX_DTQ(APP_CTR, TMAX_APP_DTQ_NUM /* 16 */);
DML_MAX_PDQ(APP_CTR, TMAX_APP_PDQ_NUM /* 16 */);
DML_MAX_MTX(APP_CTR, TMAX_APP_MTX_NUM /* 16 */);
```

Fig. 11 Configuration of the application container in EV3RT

just the application domain. The data segment is shared between all user domains too. This configuration is based on the fact that EV3RT has only one user domain, the application domain. Therefore, all memory objects in user domains can be considered as global. The number of non-global entries in address translation tables can be reduced to optimize the TLB flushing by setting them to be shareable, as shown in our previous work [29]. Maximum number for each kind of kernel objects is also defined.

Applications in EV3RT are loadable modules. They start from a task instead of main function. In fact, an initialization task will be created for each user application. It is responsible for executing functions such as constructors of global C++ objects before starting other tasks in the user application. Since the base addresses of its segments are not fixed, applications will be compiled to generate complete position-independent code (with `-mno-pic-data-is-text-relative` option in GCC compiler) into Executable and Linkable Format (ELF) [33] files. Applications only use service calls and shared memory areas whose pointer is acquired by service calls to interact with the platform so they do not depend on any address of the base system. In this way, recompilation of applications is not needed even if configuration of the application container (e.g. segment maximum size) is changed.

The loader is implemented to partially support ELF files for the ARM architecture [4], and configuration of the application container is hard-coded instead of parsed from configuration file. That is why we call it a 'prototype'. Although it can only properly relocate applications compiled with complete position-independent code, it is sufficient for supporting dynamic application loading in EV3RT. At present, the unloading of a module is done by killing all tasks in it forcedly and then deleting all kernel objects created for it.

In the future, we are planning to extend the loader for EV3RT into a more elegant and flexible implementation. For example, automatically generating necessary kernel objects for containers from configuration file will be supported. Refactoring like spliting target-dependent part from existing source code to improve reusability is also in progress. Further, the loader will allow a module to provide a termination routine for clean unloading, instead of just releasing all allocated resources forcedly.

6 Device Drivers and Middleware

Both hardware and software of Mindstorms EV3 have been improved vastly compared to its predecessor Mindstorms NXT. As a result, the scale of device drivers and middleware became one of the major challenges in developing EV3RT.

In this section, the strategy for implementing necessary device drivers and middleware for EV3RT is disscused firstly. Thereafter, the ap-

Table 2 Software metric comparison of Mindstorms NXT and Mindstorms EV3

		nxtOSEK	lms2012
Lines of Code	Mindstorms	LCD	157	489
		Speaker	389	597
		Motor	137	3,176
		Sensor	235	5,620
		Soft I2C	421	1,598
		Soft UART	n/a	16,889
	SoC	SPI	135	919
		AVR	257	n/a
		SD	n/a	983
	Generic	FAT	n/a	5,375
		USB	664	16,746∼
		Bluetooth	330	14,186∼
		WiFi	n/a	34,763∼

proach to reuse Linux drivers and how some middleware are integrated into EV3RT are explained.

6.1 Analysis and Strategy Decisions

To discuss the issue of scale in a quantitative way, we compared software metric of nxtOSEK [41] and lms2012 [27]. nxtOSEK is a popular platform for NXT while lms2012 is the code name of the stock Linux-based firmware of EV3. A utility called CLOC (Count Lines of Code) [1] is used to measure the amount of code. The statistics include both headers and source files with all comments and blank lines omitted. We have analyzed most of the important modules in these platforms and the results are listed in Table 2.

Software modules in Mindstorms can be mainly classified into three groups:

Mindstorms exclusive: This group includes drivers for those special-purpose devices such as motors and sensors. EV3 also uses its own software-based implementation for I2C and UART communication because its processor cannot provide adequate hardware resources. Although source code is available, detailed information for those devices is very limited.

SoC peripherals: This group includes drivers for those peripherals included in the SoC such as SPI and MMC/SD controllers. Specification and usage of those devices can usually be found in the data sheet and user manual released by the SoC maker.

Generic middleware: This group includes middleware of those common functions which can be found in various systems, such as file system and network. Functions supported by them are relatively advanced and it can be a heavy burden for users to control the low-level device directly. Thus, those middleware components are usually provided as a protocol stack or framework with a hardware abstraction layer to hide differences of hardware.

Mindstorms exclusive drivers of EV3 are about 20 times larger than NXT in the terms of total lines of code. Implementing these device drivers from scratch is extremely difficult and, if even possible, will take too much time. To reduce the workload of developing, we must reuse the existing source code as possible. Since lms2012 is a Linux-based firmware, we proposed an approach to reuse kernel-space Linux device drivers on HRP2 kernel.

For the SoC peripherals, we decided not to reuse the drivers from lms2012. Instead, the AM1808 StarterWare [42] released by Texas Instruments, the maker of EV3's SoC, is used. StarterWare is a free software development package including libraries and example applications for peripherals on the TI processors. It is designed for no-OS platform and can be easily integrated with EV3RT.

As to those generic middleware in lms2012, all of them are especially developed for a Linux system, which means that a near-complete compatibility layer will be required to reuse them directly. However, there are some open source alternatives for bare metal environment, since the functionalities they provide are actually very common even in simple embedded systems without a kernel. We ported BTstack [9] and FatFS [11] to HRP2 kernel so EV3RT can support Bluetooth communication and FAT file system.

6.2 Reusing Linux Device Drivers

Linux device drivers can be divided into two types by the execution modes: the user-space device drivers and the kernel-space device drivers. We analyzed their characteristics at first in order to reuse them in EV3RT.

A user-space device driver is an application or library whose code is, just as its name implies, running in user space. It is developed with the system call interface of the Linux kernel [31], which consists of about 380 system calls. Many user-space device

drivers tend to use complicated system calls such as socket() and mmap(). Supporting those system calls on an RTOS kernel like HRP2 can be very difficult. User-space device drivers in lms2012 include Bluetooth protocol stack (BlueZ) and communication libraries between its virtual machine runtime and low-level drivers. EV3RT does not reuse any user-space device driver from lms2012.

A kernel-space device driver is, on the other hand, a module that will be loaded into and then become part of the kernel. Since its code is running in kernel space, the user-space system call interface can not be used. Instead, it is developed with the Linux in-kernel API (a.k.a the Linux device driver interface) [30]. Although the Linux in-kernel API is not a stable interface, it is relatively simple compared to the system call interface. Except for those interfaces for generic middleware like USB and file system, the Linux in-kernel API just provides management functions for basic hardware resources such as timers and interrupts. Mindstorms exclusive drivers are mainly developed with those management functions as the devices they support are for special purposes. HRP2 kernel, like most RTOS kernels, also provide functions to manage the basic hardware resources and the similarities give us a chance to reuse Mindstorms exclusive drivers in EV3RT.

We have implemented a simplified compatibility layer of Linux in-kernel API, which enables the core parts of Mindstorms exclusive drivers to work on HRP2 kernel. Header files are needed in order to create a compatibility layer. We noticed that headers in Linux kernel will not interfere with headers in HRP2 kernel, and can be compiled easily. Thus, we just used those header files from Linux kernel. Functions supported in our compatibility layer, whose implementation details have been described in our previous work [29], are list as follows:

- Memory management
- IRQ (Interrupt ReQuest) handling
- Kernel locking: spinlock and semaphore
- High-resolution timer

As mentioned in section 3.4, device drivers in Linux use file system as their interface, which has relatively high overhead. Moreover, Mindstorms exclusive drivers are developed as loadable kernel modules but HRP2 kernel does not support dynamic loading of device drivers. Therefore, a device driver must be adapted for reusing in EV3RT. We describe the methodology to reuse a Linux driver with an example shown in Fig.12. At first, the header file `driver_common.h`, which contains the compatibility layer we implemented, is included. Then, hacks needed to make the driver compiled or working properly are defined. For example, there are multiple modules in Mindstorms exclusive drivers having a symbol named `InitGpio`, we renamed them with a macro to avoid name conflicts. The original source file of the driver to be used is included after hacks. In principle, we do not modify the original source file for maintainability, but unneeded code like data structures used in dynamic loading is commented out. The interfaces of this driver are implemented at last. Basically, the file operation functions are wrapped to implement extended service calls defined in the platform interface layer. In such a way, a Linux device driver can be integrated into EV3RT and provide high real-time performance with overhead of file system eliminated.

```
// Include the compatibility layer
#include "driver_common.h"

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

// Include source file to be reused
#include "d_pwm.c"

// Interfaces
ER_UINT extsvc_motor_command(...) {
    ...
    Device1Write(...);
    ...
}
...
```

Fig. 12 Example of reusing Linux device driver

Mindstorms exclusive drivers that have been successfully reused are listed as follows:

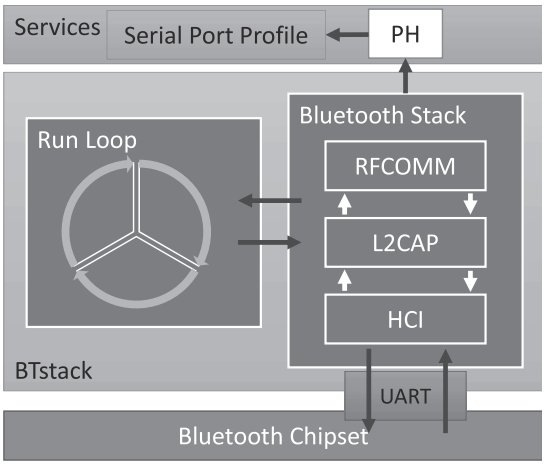


Fig. 13 Architecture of BTstack

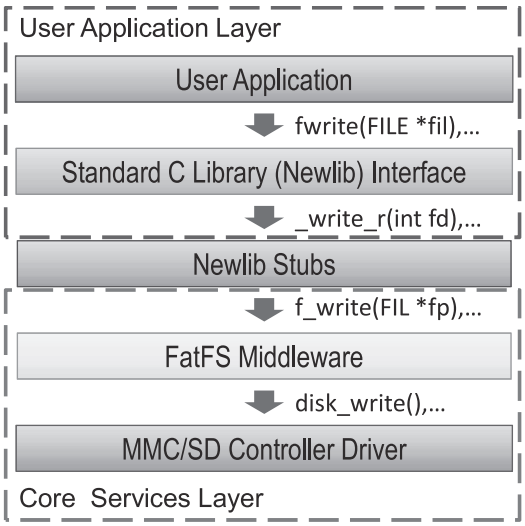


Fig. 14 Call flow of a standard file operation

- PWM control for servomotors
- A/D converter for analog sensors
- Soft-based I2C ports for I2C sensors
- Soft-based UART ports for UART sensors
- UART sensor communication protocol
- LCD driver for graphics
- Speaker driver for sound

We used CLOC [1] to evaluate the effectiveness of our approach. The results show that 32,183 lines of code are reused by writing only 1,033 lines of code including the compatibility layer. This approach saved about 97% of the coding work for supporting Mindstorms exclusive devices in EV3RT.

6.3 Bluetooth Support with BTstack

The middleware of Bluetooth protocol stack is needed in order to support Bluetooth communication in EV3RT. BlueZ, the protocol stack for Linux, is hard to be used in EV3RT as explained above. Instead, we integrated BTstack [9], an open source Bluetooth protocol stack for embedded systems into EV3RT.

Since BTstack can even work without an OS, we believe it can be ported to our platform easily. The architecture of BTstack is shown in Fig.13. 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 defined in the UART hardware abstraction layer of 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 is executed as a busy loop. We modified it to run as a task in the core services layer. A QoS task is used to dynamically control the priority of the run loop task, as described in Section 3.5. At last, we implemented the packet handlers (PH) to provide the Bluetooth Serial Port Profile (SPP) which can emulate a serial port wirelessly.

EV3RT hides BTstack from user applications since developers have to understand the Bluetooth protocol, which might be very difficult, in order to use it. An API to open the RFCOMM serial channel as a file is provided. With that file obtained, developers can easily do communications by using functions such as `fprintf()` and `fgetc()`. The serial I/O interface in HRP2 kernel is also supported. For firmware or platform developers using standalone mode, BTstack API can be used directly.

6.4 File System Support with FatFS

File system support is also very important since many applications fetch and store data from files in the microSD card. The bootloader of EV3 requires the microSD card using FAT file system. Therefore, we integrated FatFS [11], a generic FAT file system middleware, into EV3RT. FatFs is com-

Table 3 Basic characteristics of EV3RT, leJOS and MonoBrick

	EV3RT	leJOS	MonoBrick
Boot time	2.0s	65.5s	105.4s
Memory footprint	2,230 KiB (3.4%)	58,888 KiB (89.9%)	43,224 KiB (66.0%)
CPU utilization	1%	6%	5%

Table 4 Execution time of HaWe Brickbench benchmark in ms

	EV3RT			leJOS			MonoBrick		
	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.
Integer +−	2	2	3	1	2	151	1	1	19
Integer ×÷	12	12	13	9	10	69	25	28	50
Float math	26	26	26	40	43	174	191	209	275
Random number	1	1	3	4	5	79	12	12	24
Matrix algebra	5	5	6	28	33	265	39	43	286
Array sort	72	74	78	80	86	255	141	154	279
LCD text	56	56	58	155	163	356	181	217	346
LCD graphics	178	178	180	305	312	661	144	164	493

pletely written in ANSI C (C89) and has no platform dependence. It requires the implementation of a hardware abstract layer, called media access interface, which only includes six functions to access the physical devices. We implemented the media access interface with the MMC/SD controller driver to support the access of microSD card.

However, as FatFS has its own platform-independent API, it cannot cooperate with the standard C library directly. That is to say, user cannot use functions like `fscanf()` with a FatFS file. We created a bridge between Newlib and FatFS, which associates every Newlib file descriptor with a FatFS file, to overcome this limitation. The call flow of a standard file I/O function is shown in Fig.14. Thus, to access files in the microSD card, users do not have to learn the usage of FatFS at all. In fact, they are not allowed to use FatFS directly when developing in dynamic loading mode, in order to avoid conflicts.

7 Performance Evaluation

Performance of EV3RT is evaluated and compared with two existing software platforms, leJOS [43] and MonoBrick [2], in this section. leJOS supports developing applications for EV3 in Java while MonoBrick allows users to program in C# with an open source .NET framework called Mono. Both of them are based on Linux kernel and use virtual machine as runtime to execute applications. EV3RT version Beta 6-2, leJOS EV3 version 0.9.1, and MonoBrick with firmware version 1.2.0.39486 are used in the comparison. According to the results, dynamic loading mode and standalone mode in EV3RT have almost the same execution performance. We only show the performance under dynamic loading mode in this section since it is recommended for developing user applications.

At first, we measured some basic but important characteristics, including boot time, memory footprint and CPU utilization, of these platforms. The amount of used memory and CPU usage is acquired using `top` command in leJOS and MonoBrick. In EV3RT, memory footprint is known at compile time, and CPU usage can be calculated by monitoring task dispatcher. The results are shown in Table 3. EV3RT can boot over 30 times faster than others, due mainly to the static design of whole platform. It only uses about 2MB memory (3.4% of the whole available RAM) for the base system. The very small memory footprint of EV3RT allows developers to store and process bigger data sets in their application. All these platforms consume very little CPU resource and EV3RT is the best of them.

We then used HaWe Brickbench [19] version 1.09.2, a benchmark test for Mindstorms software platforms, to evaluate the overall performance. The benchmark has been performed for 1,000 times and

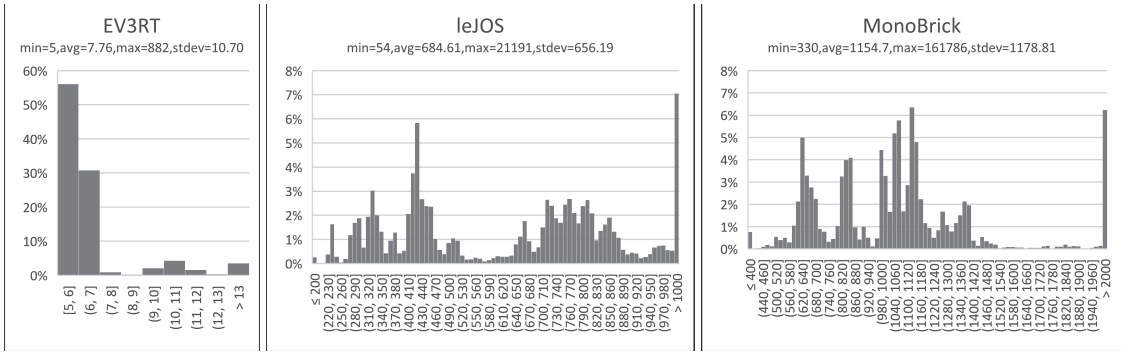


Fig. 15 Histograms of observed thread switching latency in μs

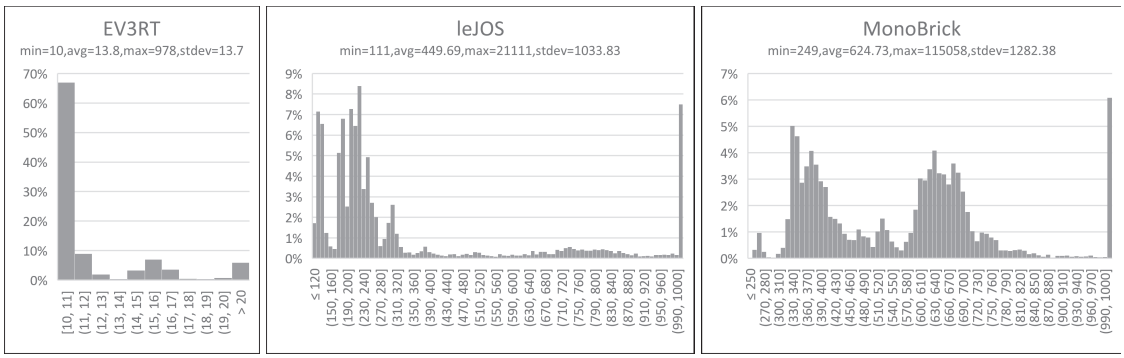


Fig. 16 Histograms of motor control API's overhead in μs

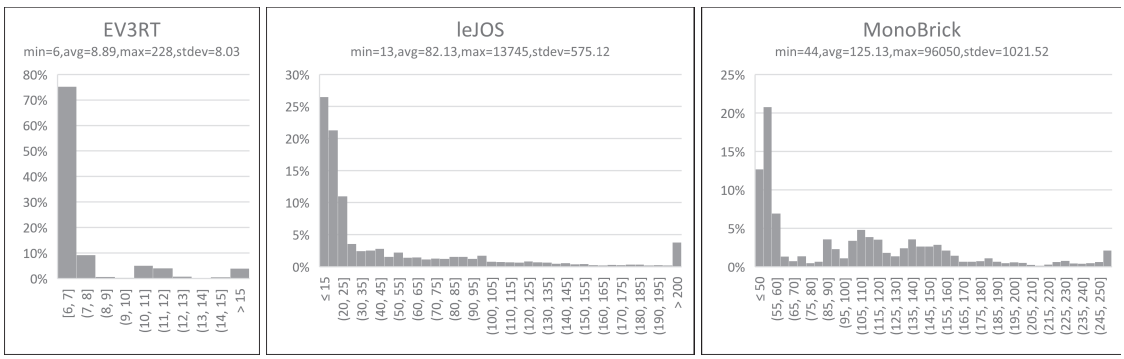


Fig. 17 Histograms of sensor access API's overhead in μs

the execution time of each test in milliseconds are shown in Table 4. EV3RT shows much more stable results and it also outperforms leJOS and MonoBrick in most cases. The average performance of LCD graphics on EV3RT is a little worse than MonoBrick because we have not optimized the implementation of graphics.

HaWe Brickbench only tests the computational performance of intelligent brick. In order to evaluate the performance in real-time robot control, we also measured thread switching, motor control and sensor access on these platforms. The thread

switching latency is critical in multi-threaded robot control. It is also very important in single thread program with real-time requirements, since there are always some system threads running in the background. Overhead of motor control is measured by calling API to set power of a servomotor, and sensor access is measured by calling API to read distance from an ultrasonic sensor. We measured them for 10,000 times on each platform, and the results are shown in Fig.15, Fig.16 and Fig.17. Averagely, EV3RT is about 100 times faster in thread switching, 40 times faster in motor control and 10 times faster in sensor access. Further, its performance is much more predictable than others, which makes it the most suitable platform for real-time applications.

8 Conclusions and Future Work

In this paper, EV3RT, a novel real-time software platform for LEGO Mindstorms EV3 robotics development kits, has been presented. It is a developer-friendly open source platform with an extendable layered architecture and easy-to-use APIs. EV3RT supports developing in two modes: dynamic loading mode for user application, and standalone mode for firmware and platform. Sample applications to show the usage of EV3RT are also provided. We based EV3RT on the TOPPERS/HRP2 static RTOS kernel, and applied the protection functionalities of HRP2 kernel to make EV3RT a reliable platform. A dynamic module loading mechanism for a static OS design has been proposed, in order to provide a smooth development process with dynamic application loading support in EV3RT. The proposed mechanism has a hardware-independent design and does not require dynamic memory management, which can be easily implemented on many targets. Existing open source software including Linux device drivers, BTstack and FatFS are reused to save the cost of developing EV3RT. We evaluated the performance of EV3RT and compared it with two existing software platforms called leJOS and MonoBrick. The results show that EV3RT can startup much faster, consume less resources and is the most suitable software platform for developing application with high real-time requirements.

As an ongoing open source project, the beta version of EV3RT has been selected by ET Robocon [22] as one of its officially supported platforms since 2015. In fact, it has become the most-used software platform among participating teams of ET Robocon. The future work will mainly focus on improving usability and connectivity, since the performance requirements have already been achieved. For example, we would like to support developing and debugging with Eclipse CDT [13] which is a fully functional C and C++ integrated development environment. We also plan to integrate the mbed IoT device platform [6] into HRP2 kernel, to support middleware like USB host, WiFi communication and TCP/IP protocol in EV3RT. Besides, TOPPERS/HRP3 kernel, the third version of HRP kernel series, is currently being developed, with features including time protection, high resolution timer and Ruby configurator added. We have experimentally supported HRP3 kernel in EV3RT and will release the source code in the near future.

References

[1] AlDanial: Count Lines of Code (CLOC), https://github.com/AlDanial/cloc.

[2] Anders: MonoBrick EV3 Firmware, http://www.monobrick.dk.

[3] ARM Ltd.: ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406-.

[4] ARM Ltd.: ELF for the ARM Architecture, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044f/IHI0044F_aaelf.pdf.

[5] ARM Ltd.: GNU Tools for ARM Embedded Processors, https://launchpad.net/gcc-arm-embedded.

[6] ARM Ltd.: mbed IoT Device Platform, https://www.mbed.com.

[7] Baker, A.: *Windows NT Device Driver Book: A Guide for Programmers, with Disk with Cdrom*, Prentice Hall PTR, 1996.

[8] Bedi, P., Qayumi, K. and Kaur, T.: Home security surveillance system using multi-robot system, *International Journal of Computer Applications in Technology*, Vol. 45, No. 4(2012), pp. 272–279.

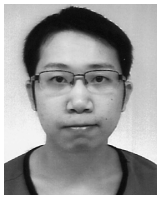
[9] BlueKitchen: BTstack, http://btstack.org.

[10] Bradley, P. J., Juan, A., Zamorano, J. and Brosnan, D.: A Platform for Real-Time Control Education with LEGO MINDSTORMS, *IFAC Proceedings Volumes*, Vol. 45, No. 11(2012), pp. 112–117.

[11] ChaN: FatFS, http://elm-chan.org/fsw/ff/00index_e.html.

[12] DENX Software Engineering: Das U-Boot – the Universal Boot Loader, http://www.denx.de/wiki/U-Boot.
[13] Eclipse Foundation: Eclipse CDT (C/C++ Development Tooling), https://eclipse.org/cdt.
[14] ET Robocon Executive Committee: Reference construction of EV3way-ET robot, https://github.com/ETrobocon/etroboEV3/wiki.
[15] EV3RT: English home page, http://ev3rt-git.github.io.
[16] EV3RT: Japanese home page, http://dev.toppers.jp/trac_user/ev3pf/wiki/WhatsEV3RT.
[17] Ferrari, M. and Ferrari, G.: *Building robots with LEGO Mindstorms NXT*, Syngress, 2011.
[18] GNU Project: GNU Compiler Collection, https://gcc.gnu.org.
[19] HaWe: HaWe Brickbench Benchmark Test, http://www.mindstormsforum.de/viewtopic.php?f=71&t=8095.
[20] Hoffmann, M., Dietrich, C. and Lohmann, D.: Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience, in *2nd GI W'shop on Software-Based Methods for Robust Embedded Systems* (*SOBRES '13*), 2013.
[21] Japan Aerospace Exploration Agency: Space Transportation Systems, http://global.jaxa.jp/projects/rockets.
[22] Japan Embedded System Technology Association: Embedded Technology Software Design Robot Contest, http://www.etrobo.jp.
[23] 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, (2002), pp. 8–12.
[24] Lee, T.: Real-time face detection and recognition on LEGO mindstorms NXT robot, in *International Conference on Biometrics*, Springer, 2007, pp. 1006–1015.
[25] LEGO Group: Discontinuing LEGO MINDSTORMS NXT, https://education.lego.com/en-au/learn/middle-school/mindstorms-nxt/faqs.
[26] LEGO Group: History of the Mindstorms series, http://www.lego.com/en-us/mindstorms/history.
[27] LEGO Group: LEGO MINDSTORMS EV3 source code, https://github.com/mindboards/ev3sources.
[28] Levine, J. R.: *Linkers and Loaders*, Morgan Kaufmann, 2000.
[29] Li, Y., Ishikawa, T., Matsubara, Y. and Takada, H.: A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support, in *OSPERT 2014*, 2014, p. 51.
[30] Linux Kernel Organization, Inc.: Linux Kernel Documentation, https://www.kernel.org/doc/htmldocs.
[31] Linux Programmer's Manual: syscalls - Linux system calls, http://man7.org/linux/man-pages/man2/syscalls.2.html.
[32] Masmano, M., Ripoll, I., Crespo, A. and Real, J.: TLSF: A new dynamic memory allocator for real-time systems, in *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, IEEE, 2004, pp. 79–88.
[33] Matz, M., Hubicka, J., Jaeger, A. and Mitchell, M.: System V Application Binary Interface, *AMD64 Architecture Processor Supplement*, Vol. 99(2013).
[34] National Instruments: Laboratory Virtual Instrument Engineering Workbench (LabVIEW), http://www.ni.com/labview.
[35] Nawawi, S., Ahmad, M. and Osman, J.: Real-time control of a two-wheeled inverted pendulum mobile robot, *World Academy of Science, Engineering and Technology*, Vol. 39(2008), pp. 214–220.
[36] Red Hat: Newlib, a C standard library for embedded systems, https://sourceware.org/newlib.
[37] Salzman, P. J., Burian, M. and Pomerantz, O.: The Linux kernel module programming guide, http://tldp.org/LDP/lkmpg/2.4/html/book1.htm.
[38] Schwarz, B., Debray, S. and Andrews, G.: Disassembly of executable code revisited, in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, IEEE, 2002, pp. 45–54.
[39] Shalloway, A. and Trott, J. R.: *Design patterns explained: a new perspective on object-oriented design*, Pearson Education, 2004.
[40] Sinha, S., Koedam, M., Van Wijk, R., Nelson, A., Nejad, A. B., Geilen, M. and Goossens, K.: Composable and predictable dynamic loading for time-critical partitioned systems, in *Digital System Design (DSD), 2014 17th Euromicro Conference on*, IEEE, 2014, pp. 285–292.
[41] Takashi, C.: nxtOSEK/JSP, http://lejos-osek.sourceforge.net.
[42] Texas Instruments Inc.: Starterware for AM1808 SoC, http://processors.wiki.ti.com/index.php/StarterWare.
[43] The leJOS project: leJOS EV3, http://www.lejos.org.
[44] The Robocup Federation: RoboCup Junior, http://rcj.robocup.org.
[45] TOPPERS Project Inc.: mruby on EV3RT + TECS, https://www.toppers.jp/tecs.html.
[46] TOPPERS Project Inc.: TOPPERS new generation kernel specification, http://www.toppers.jp/documents.html.
[47] TOPPERS Project Inc.: TOPPERS/HRP2 kernel, http://www.toppers.jp/en/hrp2-kernel.html.
[48] TRON Forum: ITRON Specification, http://www.tron.org/specifications.
[49] Valk, L.: *LEGO MINDSTORMS EV3 Discovery Book: A Beginner's Guide to Building and Programming Robots*, No Starch Press, 2014.
[50] WRO Advisory Council (AC): World Robot Olympiad, http://www.wroboto.org.
[51] Ylonen, T. and Lonvick, C.: The secure shell

(SSH) authentication protocol, http://tools.ietf.org/html/rfc4252.



Yixiao Li

received his B.E. degree in Software Engineering from East China Normal University in 2012, and the degree of Master of Information Science from Nagoya University in 2015. He is a Ph.D. candidate at the Graduate School of Information Science, Nagoya University. His research interests include real-time operating systems, embedded multi/many-core systems, and software platforms for embedded systems.

Yutaka Matsubara

is an Assistant Professor at the Center for Embedded Computing Systems (NCES), the Graduate School of Informatics, Nagoya University. He received his Ph.D. degree in Information Science from Nagoya University in 2011. He was a Researcher from 2009 to 2011 at Nagoya university, and was a designated Assistant Professor in 2012. His research interests include real-time operating systems, real-time scheduling theory, and system safety and security for embedded systems. He is a member of IEEE, USENIX and IPSJ.

Hiroaki Takada

is a professor at Institutes of Innovation for Future Society, Nagoya University. He is also a professor and the Executive Director of the Center for Embedded Computing Systems (NCES), the Graduate School of Informatics, Nagoya University. He received his Ph.D. degree in Information Science from University of Tokyo in 1996. He was a Research Associate at University of Tokyo from 1989 to 1997, and was a Lecturer and then an Associate Professor at Toyohashi University of Technology from 1997 to 2003. His research interests include real-time operating systems, real-time scheduling theory, and embedded system design. He is a member of ACM, IEEE, IPSJ, IEICE, and JSAE.