FreeRTOS porting on x86 platform

Hsuan Hsu and Chih-Wen Hsueh
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan 621, R.O.C.
{r05922020, cwhsueh}@csie.ntu.edu.tw

Abstract—**There exist some real-time solutions for x86 personal computers. Most of them involve adding real-time extensions or modifying kernels to make operating systems be able to run real-time tasks. We provide another real-time solutions for x86 personal computers by proposing a model consisting of a non-real-time operating system for non-real-time tasks, a separated real-time operating system for running real-time tasks, and a Xen hypervisor for virtualization of the two operating systems and communication between them. We choose FreeRTOS, an open-source real-time kernel, as the real-time operating system in our model. FreeRTOS has many ports, including an IA32 port. However, the IA32 port runs on a system on a chip called Galileo, and is not able to run on x86 personal computers. In this paper, we focus on modifying the Galileo port to run on x86 personal computers, and demonstrate the execution result of our modified port to show the success. Our result can be regarded as a porting example to help the people who have no porting experience to understand the porting process on bare metal as well as virtualized platform.**

Real-time operating system; x86; porting

I. Introduction

Real-time operating system (RTOS) is widely used in many aspects. Aside from various kinds of applications in embedded systems, industry also has need of RTOSes. Consider a scenario that a robot manufacturer control their robots by industrial personal computers (IPCs), which are PCs specialized for industrial purpose. They may connect an IPC and a robot together by a network cable, and develop/execute real-time programs on the IPC to control the robot. This scenario exposes the need of a PC system consisting of at least three parts: a RTOS, a development environment, and an execution user interface (UI). To meet the above requirement, there have been several solutions on different platforms. For Windows users, there is RTX, a line of real-time extensions that convert Windows into a RTOS [1]. For Linux users, there are implementations such as PREEMPT-RT kernel, the enhanced Linux kernel that has real-time ability, and dual kernel, the real-time kernel that regards the original Linux kernel as its idle task [2].

We propose a model as another real-time solution for x86 PCs. The model consists of the following parts:

- A non-real-time operating system acting as both the development environment and the execution UI, without real-time extensions nor modified kernels needed.
- A RTOS that runs on x86 bare-metal PCs.
- A Xen hypervisor to run the two operating system, and handle the communication between them.

We choose FreeRTOS as the RTOS in the model due to the following reasons. First, it is a free software under GPL, and we may distribute modified code without royalty. Second, being one of the most popular RTOSes in the market, it is well documented and well supported by the official website. Third, it has already been ported to many platforms including an IA32 port which is close to our need.

The IA32 port runs on Galileo Quark X1000 SoC rather than PCs [3]. In this paper, we focus on modifying the Galileo port to run on PCs and adding functions to support our model. We introduce the Galileo port in section II, and explains how we modify it to run on PCs in section III. The porting result is shown in section IV. Finally, in section V, we list some features that are not yet implemented but will be included in the future.

II. Original Port

A. Source Code Organization

The directory structure of FreeRTOS is introduced in the source organization page on the official web page [4]. All ports of FreeRTOS are contained in a single source code download. Under *Source/portable* exist all supported architectures. Under Demo exist all supported platforms, each consisting of some additional port files and a demo application. An architecture and a platform form a complete port.

Fig. 1 shows the directory structure of the Galileo port. The application files are *Demo/IA32_flat_GCC_Galileo_Gen_2/main.c* and *Demo/IA32_flat_GCC_Galileo_Gen_2/Blinky_Demo/main_blinky.c*. The port uses the architecture *Source/portable/GCC/IA32_flat* with additional port files in *Demo/IA32_flat_GCC_Galileo_Gen_2/Support_Files*.

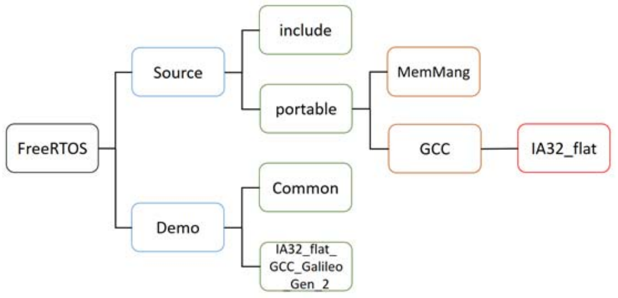


Figure 1. The directory structure of the Galileo port.

Source/portable/GCC/IA32_flat provides architecture support to the real-time kernel (i.e. *Source/*.c*). However, instructions such as CPU initialization and I/O support, which are not a part of the real-time kernel, must be implemented somewhere. For this reason, files for boot sequence, hardware support, and some standard library support exist in *Demo/IA32_flat_GCC_Galileo_Gen_2/ Support_Files*.

From now on, we mention a file using the file name without path, once its path has been mentioned.

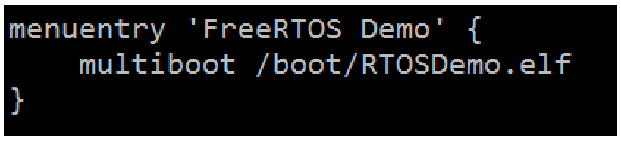
B. Building and Booting

The tool chain used to build the Galileo port is i686-elf-gcc, a GCC cross compiler which runs on Windows and produce files of ELF format. The Galileo port uses Eclipse CDT as the development environment. The internal builder in Eclipse CDT then uses i686-elf-gcc as the compiler, assembler, and linker to build the source into an ELF image. One may also extract the GCC options according to the builder setting of Eclipse CDT, and feed the compiler options to the preferred builder to build the source.

Instead of being loaded by an operating system, the built image of the Galileo port runs on x86 bare metal. The image is booted with *Multiboot Specification* [5], and *Demo/IA32_flat_GCC_Galileo_Gen_2/Support_Files/ startup.S* gives the multiboot header. A boot loader that understands *Multiboot Specification* must be installed on the bootable device containing the image, with its configuration file set up properly to load the image. At least two options should be specified in the configuration file of the boot loader as follows:

- The path of the image.
- The method to boot the image (must be multiboot).

The boot loader we use to boot the image is GRUB 2. Fig. 2 shows the content of our *grub.cfg*, the configuration file of GRUB 2. It tells GRUB 2 to boot the image using the *Multiboot Specification*.



```
menuentry 'FreeRTOS Demo' {
    multiboot /boot/RTOSDemo.elf
}
```

Figure 2. Our *grub.cfg*

III. PORT MODIFICATION

Our mission is to modify a port to run on a different but similar platform, rather than a totally different platform. Therefore, the key is that we modify the code structure as less as possible, unless the modified structure fits better to the target platform. The ability to trace the source code and learn implementation details when needed is important. Furthermore, although we just modify an existing port, it is still recommended to have enough understanding of the target platform, which includes the instruction set architecture of CPU and other hardware components within the computing system.

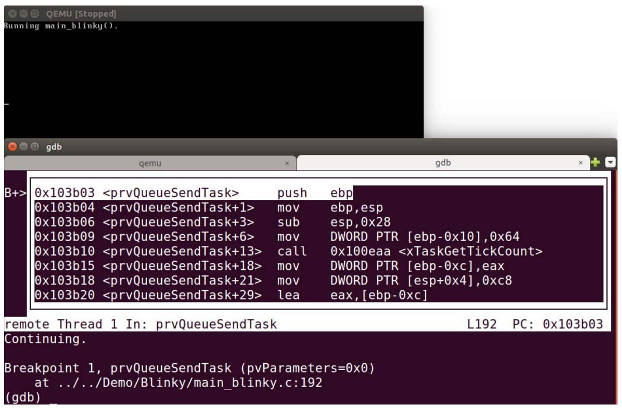
A. A Guide to Modification

The general step is described as follows. First, ensure that there exists a piece of code that will not work on the target platform (called a bug from now on). Second, starting from the entry point, trace the execution flow to locate the bug. Third, reimplement the part for the target platform. After the bug fixed, continue tracing the execution flow until another piece of code needs modification.

It is easier to locate the bug if its cause is known. For example, we know that systems on chips usually require initialization of General-purpose input/output (GPIO) to perform I/O, and that PCs do not. Starting from the entry point of the source code, we can find the expecting GPIO initialization without much effort.

However, if we have no idea what kind of the bug is, it will be a nightmare tracing the source code statically to find the bug. In this situation, two debugging methods might be applied.

- Insert debugging messages to places where bugs might exist.
- Perform run-time debugging with a virtual machine and a debugger. We use QEMU-KVM as the virtual machine and GDB as the debugger. QEMU-KVM runs the RTOS in a virtual machine, with the option -s to let GDB connect into and monitor the software running in the virtual machine (i.e. the RTOS). Fig. 3 shows how we debug the RTOS.



```
Running main_blinky().
```

```
B+> 0x103b03 <prvQueueSendTask>       push   ebp
    0x103b04 <prvQueueSendTask+1>     mov    ebp,esp
    0x103b06 <prvQueueSendTask+3>     sub    esp,0x28
    0x103b09 <prvQueueSendTask+6>     mov    DWORD PTR [ebp-0x10],0x64
    0x103b10 <prvQueueSendTask+13>    call   0x100eaa <xTaskGetTickCount>
    0x103b15 <prvQueueSendTask+18>    mov    DWORD PTR [ebp-0xc],eax
    0x103b18 <prvQueueSendTask+21>    mov    DWORD PTR [esp+0x4],0xc8
    0x103b20 <prvQueueSendTask+29>    lea    eax,[ebp-0xc]
remote Thread 1 In: prvQueueSendTask                    L192   PC: 0x103b03
Continuing.

Breakpoint 1, prvQueueSendTask (pvParameters=0x0)
    at ../../Demo/Blinky/main_blinky.c:192
(gdb) _
```

Figure 3. A screenshot of debugging

B. Porting Input/Output

As mentioned above, the Galileo port utilizes GPIO to perform I/O. In addition, a serial cable is used to connect the Galileo board to the host PC. To find the initialization code of GPIO and serial port, We first trace the source code statically to understand the roles of some source files and the control transfer between them.

Taking a look at the linker options, we know that *Demo/IA32_flat_GCC_Galileo_Gen_2/elf_ia32_efi.lds* is the linker script file, and the entry point is _start defined in *startup.S*. From the address _start, after some CPU and memory initializations, the execution flow jumps to main() in *main.c*. At the start of main(), prvLoopToWaitForDebugConnection() and prvSetupHardware() are called. The two functions then call the functions that initialize GPIO and serial port, which are exactly what we search for. We simply remove the GPIO and serial port initialization functions to avoid unpredictable errors.

It is noticeable that in *main.c*, there are many functions defined for other files and are not called in *main.c* itself. For example, vApplicationIdleHook() is the idle task hook called by portTASK_FUNCTION() in *Source/tasks.c*, a part of the real-time kernel. Many of these hooks contain functions printing text onto the terminal. The only thing to know is that these hooks will be called at some time during the execution flow. Understanding the purposes of these hooks and their callers is not necessary (but helps however).

To port the Galileo print functions, it is required to have knowledge of printing characters onto PC screens. Basic Input/Output System (BIOS) sets up a memory region 0xa0000 to 0xbffff as the VGA display memory. Accessing the address with base 0xb8000 and offset less then 80 * 25 * 2 to read/write characters on a 80 * 25 screen. We implement our printf() and replace the Galileo print functions with it.

The Galileo port only calls an input function in vAssertCalled(), the assert hook defined in *main.c*, to wait for any key being pressed. We simply remove the input function and rewrite vAssertCalled().

C. Fixing the Tick Interrupt

The Galileo port application creates two tasks: one sends heartbeats to a queue every 200 ms, and the other waits for data from the queue and displays confirm messages once it receives heartbeats. Once the I/O is ported, the image should have run with output messages displaying periodically. The execution result shows output messages displaying only once, however. To find the problem, we first check whether the tick interrupt raises periodically. The tick interrupt handler may be located by tracing the execution flow: main() in *main.c*, main_blinky() in *main_blinky.c*, vTaskStartScheduler() in *tasks.c*, xPortStartScheduler() in *Source/portable/GCC/IA32_flat/port.c*, and prvSetupTimerInterrupt() in *port.c*. Taking a look at the arguments of prvSetInterruptGate() in prvSetupTimerInterrupt(), we know that the handler of portAPIC_TIMER_INT_VECTOR is vPortTimerHandler() in *Source/portable/GCC/IA32_flat/portASM.S*. Since vPortTimerHandler() is responsible for calling vTaskSwitchContext() which performs the context switch, we ensure that vPortTimerHandler() is the tick interrupt handler. In vPortTimerHandler(), we insert assembly code which calls printf() to display messages. After rebuilding and rebooting the image, the messages does not show as expected, which implies that the tick interrupt is not set up correctly.

Before handling the bug, we find out where the execution flow goes. By performing run-time instruction-level debugging, we trace the execution flow after every application task yells the CPU. Since the control transfer at low level involves not only jumps and calls but also some complicated interrupts which transfer to their interrupt service routines, we just give the conclusion as follows. The kernel creates an idle task that runs on the lowest priority. The idle task will be preempted if there exist other ready tasks. Since the system tick count does not increase, the application tasks never become ready once they yell the CPU. As the result, the idle task loops forever.

All ports with IA32 architecture use Advanced Programmable Interrupt Controller (APIC) to handle interrupts [6]. The system tick is driven by the timer provided by APIC [7]. Although the Galileo port has code for APIC initialization, it does not work correctly on PCs. We then read the pages about APIC and APIC timer on OSDev Wiki to learn the details about using APIC and its timer. The bug is fixed after the following modifications:

- There is a chip called the 8259 Programmable Interrupt Controller (PIC), which is the predecessor of APIC. To use APIC, one must disable 8259 PIC by masking out interrupts from 8259 PIC. The Galileo port does have a function which masks out the 8259 PIC, named vInitialize8259Chips() in *Demo/IA32_flat_GCC_Galileo_Gen_2/Support_Files/galileo-support.c*. The Galileo port, however, does not call the function. We add the call in prvSetupHardware() in *main.c* to solve this issue.
- APIC timer works like many other hardware timers: given an initial count, the timer counter decreases, and generates an interrupt when it reaches zero. Several modes such as one-shot mode and periodic mode may be applied. To drive the system tick, periodic mode must be chosen to generate interrupts periodically. The Galileo port misses this step in prvSetupTimerInterrupt() in *port.c*. After toggling the mode flag to periodic mode, the modified port now displays output messages periodically.

D. Timer Calibration

Although the tick interrupt raises periodically, the interval between ticks still needs calibration. FreeRTOS requires users (i.e. application developers) to define configTICK_RATE_HZ as the tick rate in *FreeRTOSConfig.h*, a user-provided header containing macros that must be defined by users. In addition, the Galileo port also requires users to define configCPU_CLOCK_HZ. In prvSetupTimerInterrupt() in *port.c*, the Galileo port then takes configCPU_CLOCK_HZ as the decreasing rate of the APIC timer counter, and calculates the correct initial count of the counter to generate the tick interrupt at the rate configTICK_RATE_HZ.

In fact, the decreasing rate of APIC timer counter depends on the external frequency of a CPU, which varies between machines. Instead of defining configCPU_CLOCK_HZ, a more portable way is to measure the decreasing rate at run time. To achieve this, another timer which must be CPU-rate-independent is required. For simplicity, we use Programmable Interval Timer (PIT), which works at roughly 1193182 Hz [8]. Fig. 4 shows the pseudo code of our calibration process.

IV. EXECUTION RESULT

Our PC port now runs correctly. For demonstration, we add a task displaying messages every second to the original application. In addition, we add code which prints output messages after a heartbeat is sent. Hence, there are three tasks in total, each displaying output messages. The code can be found at https://github.com/kugwa/freertos_x86_gcc_pc.

Putting the built image and *grub.cfg* together, we use grub-mkrescue to produce a bootable ISO. The ISO may be used to create a bootable device to boot on bare metal, or fed to a virtual machine manager to boot in a virtual machine. Since the PC port is designed for our model, we boot it as a guest domain of Xen. Fig. 5 and fig. 6 shows the GRUB 2 menu and a screenshot of our application.

```
/* Since the PIT conuter has only 16 bits,
the longest time PIT can delay is about 0.05
(i.e. 65535 / 1193182) second. In other
words, the wrap frequency must be greater
than or equal to 19. */

pit_wrap_hz = 19
pit_init_count = 1193182 / pit_wrap_hz
high8 = pit_init_count & 0xff00

/* APIC timer counter starts from the maximum
count. */

APIC_Timer.init_count = 0xffffffff
APIC_Timer.start()

PIT.init_count = pit_init_count
PIT.mode = PERIODIC
PIT.start()

/* Use two empty loops to wait until PIT
counter wraps. */

/* wait until high-8-bits decreases */
while PIT.current_count & 0xff00 == high8

/* wait until high-8-bits recharges */
while PIT.current_count & 0xff00 != high8

remain = APIC_Timer.current_count
APIC_Timer.init_count = (0xffffffff - remain)
     * pit_wrap_hz / configCPU_CLOCK_HZ
APIC_Timer.mode = PERIODIC
APIC_Timer.restart()
```

Figure 4. Calibrate APIC timer against PIT.

V. FUTURE WORK

Being the RTOS part in our model, a successful FreeRTOS port on PCs is not enough. Here we describe some features that have not been supported by the current PC port but are required by our model.

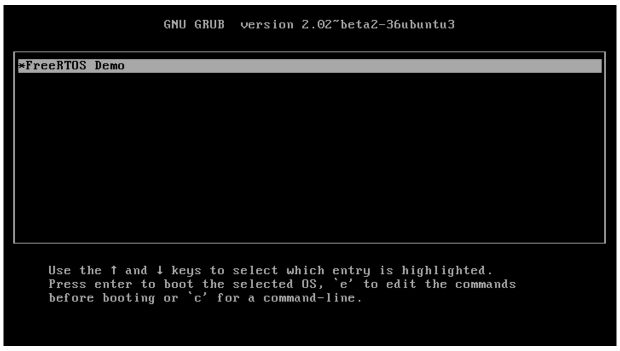
A. *User Input*

The PC port currently has no functions for input. In our model, we do not send input to the RTOS directly via a keyboard; however, the execution UI would send data to the RTOS. To implement this feature, we plan to establish shared memory between the execution UI and the RTOS via Xen's hypercalls. The RTOS then creates a task for receiving data from the shared memory.

B. *High Resolution Timer*

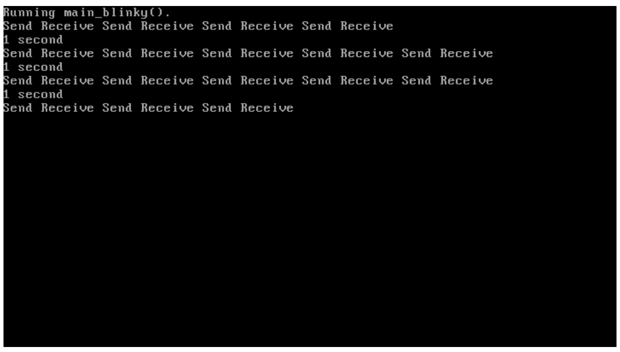
FreeRTOS provides delay functions for tasks to block themself. However, the granularity of delay is the period of ticks. FreeRTOS requires configTICK_RATE_HZ to be less than or equal to 1000, or the cost of context-switch becomes too expensive. If we want a task to delay a period of time shorter than 1 ms, the task must poll the APIC timer counter by itself until the number of counts reaches. We consider two approaches for the polling method:

- The task polls without preparation. The disadvantage is that another task might preempt the task during polling. After the task regain the control, it might find itself delaying too much.
- Before polling, the task may call taskENTER_CRITICAL(), which tells FreeRTOS to disable interrupt, to ensure not to be preempt by other tasks. After the polling is done, the task calls taskEXIT_CRITICAL() to enable interrupt. The whole RTOS might miss a tick, however, causing all task delaying 1 ms.



```
Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, `e' to edit the commands
before booting or `c' for a command-line.
```

Figure 5. The GRUB 2 menu



```
Running main_blinky().
Send Receive Send Receive Send Receive Send Receive
1 second
Send Receive Send Receive Send Receive Send Receive Send Receive
1 second
Send Receive Send Receive Send Receive Send Receive Send Receive
1 second
Send Receive Send Receive Send Receive
```

Figure 6. A screenshot of the PC port application

REFERENCES

[1] "RTX (operating system)," retrieved from https://en.wikipedia.org/wiki/RTX_(operating_system)

[2] Robert Berger, "Getting real (time) about embedded GNU/Linux," retrived from http://www.embedded.com/design/operating-systems/4204740/Getting-real--time--about-embedded-GNU-Linux

[3] "FreeRTOS running on the Intel Galileo (x86/IA32 Quark SoC X1000)," retrieved from http://www.freertos.org/RTOS_Intel_Quark_Galileo_GCC.html

[4] "Free RTOS Source Code Directory Structure," retrieved from http://www.freertos.org/a00017.html

[5] "Multiboot Specification version 0.6.96," retrieved from https://www.gnu.org/software/grub/manual/multiboot/multiboot.html

[6] "APIC," retrieved from http://wiki.osdev.org/APIC

[7] "APIC timer," retrieved from http://wiki.osdev.org/APIC_timer

[8] "Programmable Interval Timer," retrieved from http://wiki.osdev.org/PIT