Lab 2: Timers, Interrupts and System Calls

Character cannot be developed in ease and quiet. Only through experience of trial and suffering can the soul be strengthened, ambition inspired, and success achieved.

14-642, 18-349 Introduction to Embedded Systems

Checkpoint and Checkpoint demo Due: Thursday, February 22, 2018
Final Due: Tuesday, March 6, 2018
Demo by: Thursday March 8, 2018

Contents

1 Introduction and Overview

1.1 Goal

The goal of this lab is to build the first part of your kernel that provides user space isolation, interrupts, and the ability to host a simple servo motor control application. In the last lab, you implemented a cyclic-executive style (single infinite loop) program that ran entirely in supervisory mode. In this lab, you will create a kernel capable of running arbitrary programs in user space. Your new kernel will be able to handle requests from user processes via *system calls*. Finally to test your kernel, you'll write a servo motor control program, using a few custom defined system calls.

1.2 Task List

1. Get the starter code from GitHub classroom (see Section 2)
2. Implement IRQ handler and ARM timer (see Section 3)
3. Submit checkpoint with IRQ every 1 second (see Section 3.5)
4. Create a user mode (see Section 4)
5. Create servo mode system calls (see Section 5)
6. Create servo user mode program (see Section 6)
7. Demo servo to TAs
8. Submit to GitHub (see Section 8). Make sure your code has met style standards including Doxygen documentation!

1.3 Grading

Task	Points
Checkpoint 1	10
Working IRQs with ARM timer	10
User mode isolation with working SWI	25
System calls other than servo motor	15
Servo motor system calls, correctly implemented with correct IRQ handler integration	10
User space application to control servo motors using the serial console (ft-diterm)	15
Style (including Doxygen documentation, following submission protocols)	15
TOTAL	100 pts

2 Starter Code

Use the magic link to create a repo for lab 2. Much of this lab depends on your UART implementation from lab 1. You will need to merge your code from lab 1 into this lab's starter code as needed.

Compiling and running

We've added a new flag (`USER_PROJ`) to your `Makefile` that allows you to specify the loading of a user program:

```
make PROJECT=<directory of kernel> USER_PROJ=<directory of user program> gdb
```

The above command does the following:

- Compiles your kernel (specified by the `PROJECT` flag) and user program (specified by the `USER_PROJ` flag)
- Start `gdb`, and load both the kernel and user ELF files into the Raspi's memory.

You can then debug and run your program like the previous labs. For example, to load your kernel with the servo control program, run:

```
make PROJECT=kernel USER_PROJ=servo gdb
```

Once in GDB, don't forget to set breakpoints or type `continue` to launch the program.

3 IRQs and ARM timer

The first step is to implement an IRQ handler. This will require updating the ARM vector table, writing an IRQ handler, and configuring a timer that will eventually trigger the IRQ. In our case, the IRQ will be used to create servo motor control pulses.

3.1 Installing the Jump Table

By default, the RPi loads the kernel at `0x8000`. As discussed in lecture, the ARM processor responds to interrupts by setting the program counter to the corresponding entry in an interrupt vector table. This vector table starts at address `0x0`. For example, if a software interrupt occurs, the processor will set `pc` to `0x08`. The instruction at `0x08` will then load the address of the actual interrupt handler into `pc` i.e jumping to the handler. When writing interrupt handlers as an embedded designer, you will usually have to write some assembly since every CPU architecture handles interrupts differently. We have already defined an ARM jump table for you. You need to load it at `0x0`. Look inside of `kernel/src/supervisor.S`. You need to fill out the assembly for the function `install_vector_table`. This function should do the following:

- Get the address of the `interrupt_vector_table` assembly label in a register.
- Store all 8 `ldr` instructions starting at `0x0` upward. Don't worry about the order of the handlers in the table. The starter code provided already matches what the ARM CPU expects for each interrupt.
- **Fill in the remaining**, 7 soft vector table labels.

You should compile your kernel and look at `kernel.asm` to get a sense for how the C source maps to assembly. In this file, you should find the `interrupt_vector_table` label, the vector table and the soft vector table. This method is used to jump to a full 32-bit address of an interrupt handler with the limitation of the small address field offset in the `ldr` instruction.

3.2 Writing an IRQ ASM handler

After you installed the vector table correctly we need to fill out the `irq_asm_handler`. This routine will be called when an ARM IRQ interrupt occurs. The handler needs the following:

- Set the stack pointer to the top of the IRQ stack. The provided linker script has defined the label `__irq_stack_top` for you to use in your code (see Section 5: Assembly Tips and Calling Conventions for more info).
- Save the register context and the address of where you need to return to on the stack.
- Branch *and link* to the `irq_c_handler` (defined in `kernel/src/kernel.c`).
- Restore register context off the stack.
- **ATOMICALLY**, set the pc where you need to return to and move the SPSR into the CPSR. If you don't do this, then you could be interrupted while trying to return from an interrupt (bad things happen).

3.3 ARM Timer

Now we need to configure the ARM timer to generate an IRQ interrupt to see if the handler is working correctly. To do this, we first need to configure the ARM interrupt controller using its MMIO interface. The description of the interrupt controller starts on page 109 of the BCM2835.pdf datasheet. In a nutshell, the interrupt controller has 3 main registers that you should focus on: IRQ basic pending, Enable Basic IRQs, Disable Basic IRQs. The MMIO map for these is on page 112, The rest of the registers handle GPU interrupts and FIQs which we are not handling in this class. In each of these registers, you should look for which bit handles enabling, disabling, and pending ARM timer IRQs.

Next, we need to look at how to set up the ARM timer to generate IRQs at the rate we want. The description of the ARM timer starts on page 196 of the BCM2835.pdf datasheet. The function definitions in `kernel/include/timer.h` describe the interface you need to implement, which is in `kernel/src/timer.c`.

After you implement this, you can put a simple `printk()` in the `irq_c_handler` to see if you can receive interrupts! You will need to call `enable_interrupts()` defined in `349libk/include/arm.h` to enable IRQ interrupts in the CPSR since they are disabled by default when the kernel boots. Just being able to generate IRQs is enough for now. Later in this lab, we will have our IRQ handler control the servo motor.

3.4 IRQ Tips and Tricks

- For loading the `interrupt_vector_table`, look at the `stm` (store multiple) and `ldm` (load multiple) ARM instructions.
- Always think about calling conventions when writing assembly (see Assembly Tips and Calling Conventions for more info)!
- Look into the `movs` ARM instruction. `movs` is "move with set" and will cause result flags to be set. It will be helpful for interrupt handlers.
- Ignore all bits in the timer control register that do not pertain to running in 32 bit mode, enabling interrupts, setting the prescaler, and enabling the timer.
- Don't forget to check the errata!
- Don't forget to clear the ARM timer interrupt in `irq_c_handler`.
- Don't forget about the ARM pc offset when figuring out where you need to return to in your `irq_asm_handler`.
- Look into the `ldr <register>, =<label>` ARM instruction for how to load an assembly label into a register.

3.5 Checkpoint

You must submit a version of your kernel that successfully handles ARM timer IRQs and simply prints `"IRQ!"` every second. See Section 8: Submission for how to tag this version of your kernel on github.

4 Supporting a User Mode

Now that we have IRQs, we can setup the infrastructure to provide a user mode. At that point, we have an actual kernel! You will implement system calls for your kernel which will allow for user mode programs access to kernel data structures. Before we continue, you will need to understand how user programs are loaded and the typical execution flow of a user program. As you learned in lecture, the ARM processor supports different modes of operation. Your kernel is loaded at memory address `0x8000` and runs in *supervisor* mode, while your user program will be loaded at memory address `0x300000` and should only run in *user* mode. This prevents user mode programs from accessing reserved registers, arbitrarily changing processor mode, etc. Whenever a user program needs access to system resources, it must make a request via a *system call* to the kernel.

Ideally, peripherals and resources like UART, I2C, and system timers should be managed by the kernel and only accessible to user programs through these system calls [1]. In ARM, these system calls will be implemented via *software interrupts* (SWI).

4.1 Entering User Mode

Remember, your kernel is loaded at address `0x8000` while user programs are loaded at address `0x300000`. The RPi CPU will first begin executing at `0x8000` i.e your kernel. Your kernel should set up user space and jump to the user mode program at `0x300000`. This process will need to be implemented in the function `enter_user_mode` defined in `code/kernel/src/supervisor.S`. Setting up user mode consists of the following:

1. Saving the current supervisor context on the kernel's stack.
2. Changing the processor mode to user mode in the CPSR and enabling IRQ interrupts.
3. Setting the user mode stack pointer to the linker defined label `__user_stack_top`.
4. Jumping to where user code is loaded, which is defined by the `__user_program` linker label.

Your kernel should then call this method to begin executing a user program after it has done all other initialization needed (UART, servo, IRQs, etc).

4.2 Implementing System Calls for `newlib`

Now that we have a way to launch programs in user mode, we need system calls for user mode programs to communicate with the kernel. The SWI numbers and syscalls that you need to support are defined in `code/349libk/include/swi_num.h` and `code/kernel/include/syscalls.h` respectively. Your user programs will be linked with `newlib`, a popular C standard library implementation intended for use on embedded systems. This will allow your user programs to use any libc function, such as `printf`, `malloc`, etc. These library functions will use your system calls via assembly stub functions, defined in `code/newlib/349include/swi_stubs.c`. Each stub just initiates a software interrupt with the appropriate SWI number we have defined for you in `code/349libk/include/swi_num.h` and then returns the result to the caller (see Calling Conventions for more information on return values). You need to fill in all these stubs. This is an extremely common bootstrapping process when bringing a system up on a new platform.

Now you'll need to implement the syscalls defined in `code/kernel/include/syscalls.h`. These are the kernel system call routines. We have given you a few dummy implementations for the syscalls that we do not care about (for example, ones that use a file system). We have also implemented the system call `syscall_sbrk` for you. This is used by `malloc` to increase the size of the user heap. This lab is challenging enough without having to deal with user heap issues in `malloc`. For now, you'll have to implement the following:

```
void syscall_exit(int status);
```

The `exit` syscall is called when the user program has finished executing, or wishes to exit early on purpose. An Example of this is seen in `crt0.S`. Historically, there is an assembly file called `crt0.S` which is used by the kernel to call the user program's `main()`. This file is located at `newlib/349include/crt0.S`. Notice how after the branch to user `main`, the `exit` system call is called if the user program returns with a `swi` instruction. Your system call implementation for `exit` should print out the exit status of the user program and hang with interrupts disabled. Since we are not requiring you to load user programs into memory, we will not require your kernel to handle a user program returning.

[1] We won't be implementing memory protection in our kernel, so technically your user programs will still have access to peripherals through MMIO.

```
int syscall_read(int file, char *ptr, int len);
```

Since we do not have a filesystem, your `read` syscall should only support reading from `stdin` and return -1 if this is not the case. For the Pi, `stdin` is treated as the serial console via UART. To mimic traditional behavior of reading from `stdin`, `read` should read bytes one by one and echo each byte back as they are read. It should also process certain special bytes differently:

- An end-of-transmission character (ASCII value 4) notes the closure of the stream our characters are coming from. The syscall should return immediately with the number of characters previously read.
- A backspace (ASCII value 8) or delete (ASCII value 127) character neither gets placed in the buffer nor echoed back. Instead the previous character should be removed from the buffer, and the string "\b \b" should be printed (this erases the previously written character from the console).
- A newline (ASCII value 10) or carriage return (ASCII value 13) character results in a newline being placed in the buffer and echoed back. Additionally, the syscall should return with the number of characters read into the buffer thus far (including the most recent newline).

```
int syscall_write(int file, char *ptr, int len);
```

Since we do not have a filesystem, your `write` syscall should only support writing to `stdout` and return -1 if this is not the case. For the Pi, `stdout` is treated as the serial console via UART. `write` can simply output each byte in the buffer to the serial console.

SWI Assembly Handler

After implementing these C system calls, we need only one more piece of the puzzle to make system calls work. We need an assembly SWI handler that calls a C SWI handler and figures out which system call the user wants. This process is very similar to what we did for IRQs with some subtle differences. Every system call is generated with the `swi` instruction which generates a software interrupt. The ARM CPU will then execute the assembly handler `swi_asm_handler`. This handler needs to do the following:

- Support nested SWIs. Your handler must be re-entrant. Google and infocenter.arm.com are your friends.
- Support IRQs during SWIs.
- Use the link register (plus some offset) to get the address of the `swi` assembly instruction in user mode. Then bit mask this instruction to get the SWI number the user requested. Pass this in as argument one to your `swi_c_handler`.
- Pass in the pointer to the user system call arguments as argument two for your `swi_c_handler`.

Once you write this assembly handler, you can write the `swi_c_handler` (in `kernel/src/kernel.c`) as one huge switch statement on the user's SWI number and call your kernel C system call functions. If all goes well, you should have a fully functional kernel, capable of running user programs that use standard libc functions! Try playing around in `code/servo/src/main.c` and using C library functions like `printf`, `read`, `write`, etc. to verify that your kernel and system calls are working correctly.

User Mode and SWI Tips and Tricks

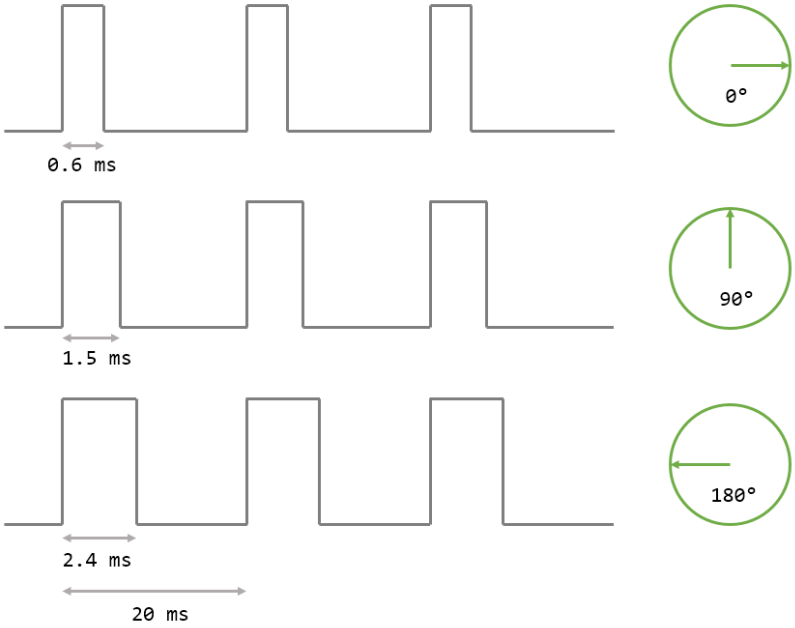
- You can't directly operate on status registers (CPSR, SPSR) like you can with normal registers. You'll need to use the `msr` and `mrs` instructions instead.
- When debugging system calls, having IRQs running will just make your life harder. It would be best to leave interrupts disabled during this.
- To handle nested SWIs, you will need to save the SPSR on the kernel stack along with the register context.

5 Servo Motor Control

Now that we have working IRQs and system calls, we need to add system calls to allow a user program to control the position of two servo motors independently.

5.1 Servo Motor Operation

Hobby servo motors such as the SG90 are designed to be inexpensive and simple to control. Unlike normal DC motors, servos have a limited range of motion, usually about 180°. The servo includes an internal feedback controller that will hold the motor at a user specified position. We specify the desired position by adjusting the width of a periodic pulse on the servo control line. In this lab, we are using the SG90 servo motor, which expects a pulse every 20 ms. By varying the width of the pulse from 0.6 ms to 2.4 ms[2], we can set the servo position from 0°to 180°.



The BCM2836 (and almost any embedded microcontroller) has a hardware pulse width modulation (PWM) module that can be used to generate these types of pulses. However, for this lab we will be using the timer interrupt support you have implemented to manually generate the servo control signal.

5.2 Configuring the IO Board for Servo Control

The IO board you have been provided with includes two three-pin headers for connecting servo motors. Besides connecting the control line (the orange wire on the servo motor) to one of the GPIO pins on the Pi, the header includes a 5v power and a ground connection. Take a look at the schematic on page 23 of `docs/RPi-io-spec-sheet.pdf`, you should see headers `P605` and `P606` located near the top middle of the schematic, along with `SW603`. Note that both headers have the following pin mapping:

[2]Ignore the range of 1ms to 2ms specified in the datasheet (`docs/SG90.pdf`).

Pin 1	`GND`
Pin 2	`VDD_SERVO`
Pin 3	`PWM0 / PWM1`

Locate `SW603` on your IO board and ensure that **only switch number 2 is on**. This will connect `VDD_SERVO` to 5v power, which the servos require. When plugging in the servo motor, ensure that `GND` (the brown wire on the servo) is connected to pin 1, which is labeled with a box around it and is located directly below the `P605/P606` label. We will use `P605` as servo channel 1 and `P606` as servo channel 2. Use the tables on pages 6 and 9 of `docs/RPi-io-spec-sheet.pdf` to figure out which GPIO pins correspond to channels 1 and 2.

5.3 349libc Servo Motor Custom System Calls

You will add two custom system calls to your kernel to allow a user program to control the position of the servo. This simple interface will allow users to enable and disable servo control for each channel and independently set the position of the connected servo. You may have already noticed these functions while implementing your SWI handlers. The two system calls are as follows (defined in `code/kernel/include/syscalls.h`):

```
/**
 * @brief Enable or disable servo motor control
 *
 * @param channel  channel to enable or disable
 * @param enabled  true to enable, false to disable
 *
 * @return 0 on success or -1 on failure
 */
int syscall_servo_enable(uint8_t channel, uint8_t enabled);

/**
 * @brief Set a servo motor to a given position
 *
 * @param channel   channel to control
 * @param angle     servo angle in degrees (0-180)
 *
 * @return 0 on success or -1 on failure
 */
int syscall_servo_set(uint8_t channel, uint8_t angle);
```

`syscall_servo_enable` should enable the periodic pulse signal on the given channel if `enabled` is `true`, and set the channel output to low (no pulse) if `enabled` is `false`. When a channel is disabled, you should be able to spin the connected servo motor with your fingers, since it is not driving to a specific angle. `syscall_servo_set` should set the pulse width on the given channel to the value corresponding to the given angle. If the channel is disabled, this system call should **not** enable it, but when the channel is enabled with `syscall_servo_enable`, the servo should spin to the newly set angle. **The servo should be stable – the motor shouldn't be whining, the servo shouldn't be shaking, and the servo should quickly stabilize to the set angle**.

6 User mode program

Now, you will create a user mode servo control interface to test the kernel functionality that you just implemented. The user application will use the serial console to allow users to enable, disable, and control the position of the servo motor. Your console should support the following commands: `enable <channel>`, `disable <channel>`, and `set <channel> <angle>`. Don't forget to print an error message if the command is invalid. An example console session is shown below:

```
opening first available ftdi device...
--- Miniterm on None: 115200,8,N,1 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Commands
  enable <ch>:  Enable servo channel
  disable <ch>: Disable servo channel
  set <ch> <angle>: Set servo angle

> enable 1

> set 1 90

> set 2 45

> enable 2

> disable 1

> badcommand
Invalid command

> set 2 270
Invalid command
```

7 Assembly Tips and Calling Conventions

1. To load 32-bit constants (such as memory addresses) into a register, you can use the `ldr` pseudo instruction. See the ARM reference for details on how to use this: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/Babbfdih.html`

2. For help understanding what `.global`, `.section`, etc mean in ARM assembly files, see the `gnu_arm_ref.pdf` in the `docs` directory.

3. For help with ARM assembly, see the `arm_isa_ref.pdf` in the `docs` directory.

4. ARM calling conventions are defined by ACPS (ARM Procedure Call Standard). This standard is defined here: `https://en.wikipedia.org/wiki/Calling_convention#ARM_.28A32.29`. If you have any questions, take a look at the ARM assembly generated for your kernel in `kernel.asm`. This will help you see how arguments/return values are passed around by the compiler and which registers are callee or caller saved. If you are still confused, ask a TA.

5. Variables defined in the linker script (`kernel/349util/kernel.ld`) such as `__irq_stack_top`, are defined at compile time by the linker. You can think of these as C global variables from the compiler telling you where some sections of a given code binary begin or end. So we can define a variable in C like the following: `extern uint32_t __irq_stack_top;` and have it be automatically filled with the address from the linker script by the compiler as it makes our ELF file. You don't need to understand how the linker script does this. However, you will need to understand how to use these labels to complete this lab.

8 Submission and Demos

1. To submit the checkpoint, use the tag `lab2-checkpoint1`. We are looking for a kernel that prints "IRQ!" every second. Doxygen style will not be graded for the checkpoint.

2. To submit the final, use the tag `lab2-submit`. The demo for the final will be a demo of the servo control console. You should display the ability to control the servo independently on two channels (you can just unplug it and plug it into the other channel to test both) using the serial interface.