Lab 3: Timers, Interrupts and System Calls

Perseverance is the hard work you do after you get tired of doing the hard work you already did.

18-349 Introduction to Embedded Systems

Checkpoint Due: Thursday, March 25, 2021 at 11:59pm
Final Due: Thursday, April 1, 2021 at 11:59pm
Demo by: Thursday, April 8, 2021

Contents

1 Introduction and Overview

1.1 Goal

In the last lab, you implemented a cyclic-executive style (single infinite loop) program that ran entirely in exception mode. The goal of this lab is to build the first part of your kernel. After this lab, your kernel will support:

- **User space isolation**. The kernel will be capable of running arbitrary programs in user space.
- **Interrupts**. The kernel will be able to handle requests from user programs via *system calls*.
- **Servo motor control**. To test your kernel, you will write a servo motor control program, using a few custom system calls.

As a word of caution, this lab is significantly more challenging than past labs. Start early, plan well and come to office hours if you have questions.

1.2 Task List

1. Get the starter code from GitHub Classroom (Section 2) and (one partner only) submit your repository link to Canvas.
2. Implement SysTick handler, configure SysTick (Section 3)
3. Implement UART with interrupts (Section 4)
4. Submit checkpoint with SysTick interrupts at 1 Hz (Section 5)
5. Create a user mode, implement I/O and memory system calls (Section 6)
6. Implement the servo system calls (Section 7)
7. Implement the servo user mode program (Section 8)
8. Submit to GitHub (Section 13). Make sure your code has met style standards!
9. Demo the servo program

1.3 Grading

Task	**Points**
Checkpoint	10
SysTick interrupts	10
UART interrupts	50
I/O system calls & SVC setup	40
Servo motor system calls & user application	20
Style and Submission Protocol (see Sections 12 and 13)	20
TOTAL	150
UART DMA (bonus)	(+10)
Atomic enter user mode (bonus)	(+5)

2 Starter Code

Use the magic link to create a GitHub repository for Lab 3. First, copy over your implementation for the following files into the new repository:

```
kernel/asm/boot.S
kernel/src/uart_polling.c
kernel/src/i2c.c
kernel/src/led_driver.c
```

Then, for this lab, you will be modifying the following files:

```
kernel/src/timer.c
kernel/src/uart.c
kernel/src/syscall.c
kernel/src/syscall_servo.c
kernel/src/kernel.c
kernel/asm/asm_helpers.S
user_common/asm/svc_stubs.S
user_proj/servo/src/main.c
```

Compiling and Running

Notice above that there is now a separate `user_proj` directory. In this directory are user programs. `servo` is the user program that controls the servo by calling servo syscalls. To load this user program, use the flag (`USER_PROJ`):

```
make {flash/build} USER_PROJ=servo
```

3 Interrupts and SysTick

Your first step is to implement the system timer. This will help you get acquainted with the STM32 interrupt model. You will later use this interrupt to generate PWM signals to drive a servo. You will be using the system timer, also known as SysTick. To set it up, you will have to update the interrupt vector table (found in `kernel/src/boot.S`), write a SysTick interrupt handler, and configure the timer to provide regular interrupts. You will fill in the following functions in `kernel/src/timer.c`:

```
int timer_start(int frequency);
void timer_stop();
void systick_c_handler();
```

3.1 Interrupt Handling in STM32

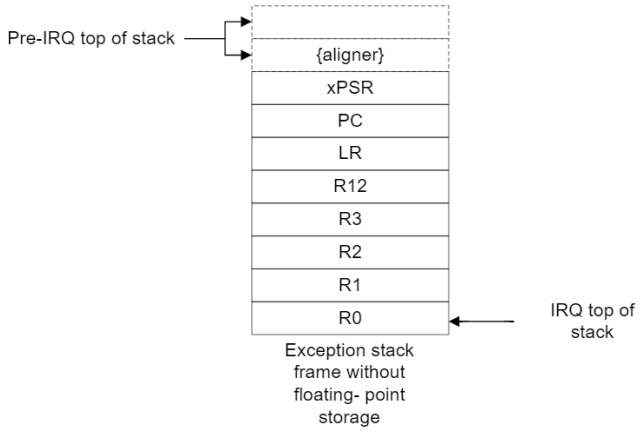
When the processor receives an interrupt, it does the following :

- It obtains the address of the interrupt handler from the vector table. For SysTick, this address is `0x800003c`, or the 15th entry in the vector table.
- The processor then pushes `r0-r3`, `r12`, `lr`, the `pc` at which it was interrupted, and the `xPSR`, all to the stack that the processor was using at the time of the interrupt.
- The processor places a special return code (see Fig. 1a) into `lr`, depending on the mode the processor was in and the stack it was using at the time of the interrupt.
- The **Main Stack Pointer** (MSP) becomes active, banking the **Process Stack Pointer** (PSP) if it was in use.

Take for example a processor running in thread mode and using the PSP. After the exception, it will be in exception mode, the stack will be MSP, the value of `lr` will be `0xFFFFFFFD` (see Fig. 1a), and the process stack will look like Fig. 1b below.

EXC_RETURN	Description
0xFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.
All other values	Reserved.

(a) Return Codes



(b) Exception Stack Frame

Figure 1: Exceptions

3.2 Writing an Interrupt Handler

Start by adding the `systick_c_handler` to the interrupt vector table. Again, the SysTick handler address should be placed at address `0x800003c`, or the 15th entry, in the vector table (see Fig. 2 below).

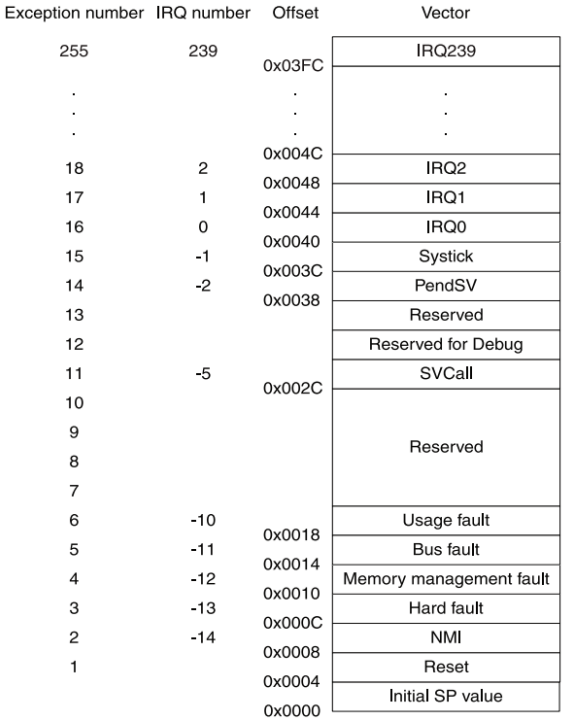


Figure 2: Exceptions

Your handler will be called when there is a SysTick interrupt. Most interrupt handlers require you to write a small assembly handler. However, since we are not passing any parameters or doing anything special with the SysTick interrupt handler, we can just put the address of the C function in the vector table, without even needing to save any registers. You should think about why this is the case and what happens when you return from this C interrupt

handler.

3.3 SysTick Interrupts

Now we need to configure SysTick to generate an interrupt to check that the handler is working correctly. To do this, we first need to configure the SysTick interrupt controller using its MMIO interface. The description of the interrupt controller starts on page 246 of the `docs/m4_programming_manual.pdf` datasheet (careful, this is the **programming manual**, not the reference manual).

The timer is relatively simple to use. In a nutshell, the interrupt controller has 2 main registers that you should focus on:

- Control register (`STK_CTRL`)
- Reload value register (`STK_LOAD`)

You only need to set the reload value, enable the timer, and enable the SysTick interrupt. Each time the timer counts to 0, the processor will receive an interrupt and jump to the function you entered into the vector table, which, for this interrupt, should be `systick_c_handler`.

Fill out the `timer_start` function to start the timer at the given `frequency` and `timer_stop` to disable the timer.

For the checkpoint, you should configure SysTick to generate interrupts once every second. See Section 5 for more details.

For the latter half of Lab 3, when you are controlling the servo, you will need to modify the interrupt frequency to something more appropriate.

In order to test SysTick, you have a couple of different options: using UART or using your LED display. Simply call `printk` or display a counter on your LED display from within your `systick_c_handler`.

3.4 SysTick Tips and Tricks

- Read the section on interrupts in the `m4_programmers_manual.pdf`, especially the section on exception return for hints on switching modes and setting the PC atomically.
- Always think about calling conventions when writing assembly! (See Assembly Tips and Calling Conventions for more info.)

3.5 Debugging Interrupts

Debugging interrupts can be tricky, especially with timers. To make things a little easier for you, the timer is disabled when you hit a break point. Additionally, we also provide you with the GDB function, `armex`. If you call this function, in GDB, just after the processor receives an interrupt, it will case on the exception return code and print the context that was saved onto the stack. The behavior of this function is undefined if the value in `lr` is not a valid return code. Furthermore, you will get inaccurate results if you push more values onto the stack after the interrupt.

Caution! GDB is sometimes inaccurate with breakpoints. For example, if you place a breakpoint at the symbol `kernel_main`, it may not put a breakpoint exactly at the start of `kernel_main` because GDB "helpfully" skips over some of the mundane stacking that happens in the beginning. This can be cumbersome with interrupts. To get around this, place a breakpoint at the address of the interrupt handler.

4 UART Again?

In the previous lab, you implemented UART using polling. As you have probably realized, polling is inefficient. Furthermore, in a real-time system, we need to provide hard real-time guarantees on operation completion. When

we use polling, the number of bytes written is not always known at compile time. This makes it difficult to analyze and place bounds on.

In order to get around this, we use interrupts. As you have learned in the lectures, interrupts can be modeled as long as we can place an upper bound on the time taken to service the interrupt. You will fill in the following files in `kernel/src/uart.c` with your new interrupt-based UART:

```
void uart_init(int baud);
int uart_put_byte(char c);
int uart_get_byte(char *c);
void uart_irq_handler();
void uart_flush();
```

You will also need to modify `kernel/src/printk.c` to make use of this new UART implementation. Currently, it uses `uart_polling_put_byte`. To switch to the interrupt-based version, simply comment out line 27 and uncomment line 28.

4.1 Interrupt I/O System

For this discussion, we will talk about `uart_put_byte`, but this also applies to `uart_get_byte` without loss of generality.

Interrupt I/O systems typically have a "top-half" and a "bottom-half". The top-half is `uart_put_byte`, which is called by the user and is typically very fast. All it does is put the byte in some sort of kernel buffer.

The bottom-half is the interrupt handler, which runs when the hardware signals via an interrupt that the UART transmit buffer is empty. The interrupt handler will take up to 16 bytes from your kernel buffer and transmit them via the UART transmit buffer. By transmitting only a set number of bytes on each interrupt, we can place an upper bound on the time to complete the interrupt and thus provide a real-time guarantee.

4.2 Implementing UART with Interrupts

You will now need to implement an interrupt I/O system using the four functions described below.

This is a little different from what you've done before. With I2C, the ADC and so on, you only needed to write to registers and give instructions to the peripherals, which were able to keep track of their state on their own. However, for interrupt I/O you will have to manage the state in the kernel[1]. This will require designing data structures in addition to the send/receive buffers and maintaining a persistent state. These can be stored in global variables in the kernel.

NOTE: The send and receive buffers themselves must not be larger than 512 bytes each. (This does not include other data structures.)

Once you have designed your data structures, you can implement the following four functions. Don't forget to enable UART interrupts via the control register!

- `int uart_init(int baud)` Initialization of the interrupt-based UART largely the same as with polling UART. You should begin by copying your previous implementation over. There are a couple of modifications you will need to make, however. First, as with the timer interrupt handler, you should fill in the entry in the vector table corresponding to UART with your `uart_irq_handler`.

[1]This is one of the purposes of the kernel, to act as the intermediary between a user program and the hardware. The kernel's objective to allow the user program to call a function like `printf` without having to worry about waiting for the UART to be ready, the size of the UART buffer, UART configuration, etc.

Then, you should enable the corresponding interrupt in the Nested Vector Interrupt Controller (NVIC). The NVIC is used to enable and disable interrupts as well as support dynamic changes in interrupt priority. A library has been provided to you in `kernel/include/nvic.h`. You can use `nvic_irq(<UART irq number>, IRQ_ENABLE)` to enable the UART interrupt. You will need to look at the vector table to find the correct UART irq number.

Finally, you will need to enable interrupts within the UART control registers to tell your UART peripheral to generate interrupts. You have the freedom to enable and disable whichever UART interrupts you feel are necessary. Furthermore, you can decide when and where to enable/disable them to suit your needs. For example, you can choose not to enable interrupts in `uart_init` and instead enable them after `int uart_put_byte`, when there is a new symbol to transmit in the kernel buffer.

- `int uart_put_byte(char c)`

  This function places the byte into the kernel buffer. It returns 0 if it was successful and -1 if the kernel buffer is full.

- `int uart_get_byte(char *c)`

  This function takes a byte out from the kernel buffer. It returns 0 on success and -1 if there are no bytes in the buffer. Note that because of this, we can no longer directly return the byte. Instead, it puts the byte at the address passed in the function argument.

- `void uart_irq_handler()`

  You will only have one handler to handle both both receive and transmit. When you receive an interrupt, you should check to see whether you are able to transmit, receive, or both (hint: you may need to check the status register). During transmission, you should ensure that there is space in the UART transmit buffer, before sending up to 16 bytes from your kernel buffer. During reception, you should read anything available in the UART receive buffer (up to 16 bytes) into your kernel buffer. Remember that we are placing a limit on the number of bytes that can be sent or received so that we can place an upper bound on the execution time of the interrupt handler.

  Upon handling the interrupt, you should clear the interrupt pending bit. Again, a function to do so has been provided for you in `nvic.h`: `nvic_clear_pending`

  WARNING! Moving the remaining bytes to the front of the buffer every time you consume data from the buffer is incredibly inefficient and unacceptable. You should think about how you can achieve first-in first-out behavior with a statically allocated buffer in a simple and efficient way. Keep in mind that the ARM processor does divide and mod operations in software, so avoid using them in the interrupt handler.

- `void uart_flush()`

  Polling is bad and polling inside the kernel is worse. However, there is one exception to this rule - if your program has exited, you don't have other programs to run, and you don't need to care about real-time guarantees anymore. We are not having you implement re-entrant interrupt handlers this semester, so when a system call is running you will not be able to receive interrupts. This means that after your user program has exited, the kernel will no longer receive UART TX empty interrupts and your kernel buffers will not be flushed. This function flushes the buffers and is required to make tests deterministic. You may only call it within `syscall_exit`, to flush any kernel buffers you have, and may use polling to achieve this. (If this seems confusing, you may want to read all the way to Section 6.3.)

4.3 Interrupt I/O Tips and Tricks

- Don't be afraid to change your design in the process of trying to implement it. Realizing there's a better way to design something while implementing it is fairly common and normal.
- If you are unsure about your interrupt I/O system design, you can come to office hours and discuss it with a TA.

5 Checkpoint

You must submit a version of your kernel that successfully handles SysTick interrupts and prints `"Interrupt!"` every second. **You will also need to modify `kernel/src/printk.c` to make use of this new UART implementation. Currently, it uses `uart_polling_put_byte`. To switch to the interrupt-based version, simply comment out line 27 and uncomment line 28.**

See Section 13 for how to submit this version of your kernel on Github. You do not need to demo the checkpoint.

6 Supporting a User Mode

Now that we have timer interrupts, we can set up the infrastructure to provide a user mode. At that point, we will have a functional kernel! You will implement system calls for your kernel which will provide an interface for user-mode programs to access 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.

6.1 Processor Modes

As you learned in lecture, the processor supports different modes of operation. Your kernel runs in *Handler* mode, while your user program will only run in *Thread* mode. This prevents user programs from accessing reserved registers, arbitrarily changing processor mode, and otherwise wreaking havoc. 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. In ARM, these system calls will be implemented via *software interrupts* (SVC).

6.2 Entering User Mode

Your board always starts up in kernel mode. Once the system resources have been initialized, your kernel is responsible for setting up user space and jumping to the user mode program. This process will need to be implemented in the function `enter_user_mode` defined in `kernel/boot/asm_helpers.S`. Setting up user mode consists of the following:

1. Load the starting address for the process stack pointer (used by user mode) into the PSP register. See the linker script for the details about the process stack pointer.
2. Branch to the symbol `_crt0` (found in `user_common/src/crt0.c`). We have provided this function for you. It deescalates privileges and switches to the PSP. It will then obtain `argc` and `argv` and jump to main. (As a side note, `argv` can be specified with the compiler flag `USER_ARG="your string here")`
3. Your usermode programs can be found in the `user_proj` directory. To build a specific user project, use: `make <flash/build> USER_PROJ=<user_project_name>`

If you have done the above correctly, you will be in Thread mode without privileges and the stack pointer will be PSP. Your kernel should only call the `enter_user_mode` method to begin executing a user program after it has initialized the necessary resources (UART, servo, timer, etc.).

6.3 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 SVC numbers and syscalls that you need to support are defined in `kernel/include/svc_num.h` and `kernel/src/syscall.c` 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 `user_common/include/svc_stubs.S`. Each stub simply initiates a software interrupt with the `svc` instruction and the appropriate SVC number (review your Lecture notes). The stub then returns the result to the caller (see Section 10 for more information on return values). We have defined the mapping between SVC numbers and system call functions for you in `user_common/include/svc_num.h`.

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 `kernel/src/syscall.c`. 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, functions that use a file system). For now, you'll have to implement the following:

Exiting

```
void sys_exit(int status);
```

`sys_exit` is called when the user program has finished executing, or wishes to exit early on purpose. An example of this is seen in `crt0.c`. Historically, this is a file called `crt0.c` or sometimes `crt0.S` which is used by the kernel to call the user program `main()`. This file is located at `user_common/src/crt0.c`. Notice how after the branch to the user `main()`, the `exit` system call is called via a `svc` instruction if the user program returns. Your system call implementation for `sys_exit` should print out the exit code, flush the UART buffer, display the exit code on the LED screen and sleep 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.

Console I/O

```
int sys_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 STM32, `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 \b 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 \n 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). NOTE: Windows machines send the bytes \r \n when you hit enter. This should be handled appropriately.

```
int sys_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 STM32, `stdout` is treated as the serial console via UART. `write` can simply output each byte in the buffer to the serial console. Be sure to use the interrupt version of UART for read and write.

Memory

```
void* sys_sbrk(int incr);
```

`sys_sbrk()` is used by programs to increment the program data space by incr bytes. On success, `sys_sbrk()` returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, `(void*)-1` is returned.

This function gives chunks of memory to malloc, which then divides up these chunks and gives them to the user. If you look in the linker script, you will see several heaps declared. For this system call, you must use the heap created between the `__heap_low` and `__heap_top` symbols. The operation of `sys_sbrk()` is fairly straightforward:

- You need to keep track of where you are in the heap (where the current program break is).

- Each time the user calls `sys_sbrk()`, move the current program break by `incr` to make more space, and give the user (i.e. return a pointer for) the memory they requested.
- If you run out of memory, return `-1`.

We provide you with the stubs for a library `k_malloc`, you may fill these in and and use the library to implement `sys_sbrk`. Using this library is not required; feel free to manage the heap some other way.

6.4 SVC Assembly Handler

When a user program performs a SVC, all the system calls generate the same interrupt regardless of which system call it is (the assembly instruction is different but the interrupt is the same). As such, we need a software interrupt handler to determine which system call the user wants to run. Therefore, you need to write an assembly SVC handler and a C SVC handler that figures out which function the user called.

Every system call is generated with the `svc <num>` instruction. The instruction operand is the SVC number. The mapping of functions to SVC numbers is defined in `svc_num.h`. Upon receiving a SVC, the ARM CPU will execute the assembly handler `svc_asm_handler` which has been placed in the interrupt vector table. You have to implement the `svc_asm_handler` and `svc_c_handler`. Use the `pc` saved on the process stack (accessed using PSP with offset) to obtain the the `svc` instruction that triggered this interrupt, which contains the SVC number. You can then case on this and call your appropriate kernel-side function.

Remember to check calling conventions (Section 10) so you know how to transition from a assembly function to a C function, especially with regard to providing arguments and obtaining the return value. Make sure the user program eventually gets the return value of the system call in register `r0`.

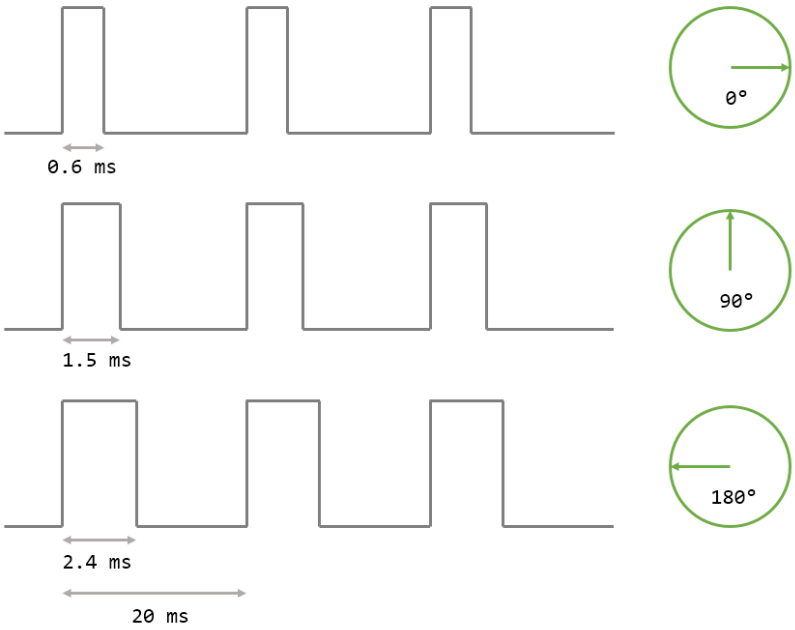
Now you should have a fully functional kernel, capable of running user programs that use standard `libc` functions! Try playing around in `servo/src/main.c` and using C library functions like `printf`, `read`, `write`, etc. to verify that your kernel and system calls are working correctly.

7 Servo Motor Control

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

7.1 Servo Motor Operation

Hobby servo motors such as the SG90 are designed to be inexpensive and simple to control. Unlike normal DC motors, servo motors 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[2] from 0.6 ms to 2.4 ms, we can set the servo position from 0°to 180°.



The STM32F401 (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 (i.e. SysTick) you have already implemented to manually generate PWM pulses to control the servo motor.

7.2 Configuring the IO Board for Servo Control

On the I/O board you built there are two three-pin headers for connecting servo motors. The brown wire on the servo is ground and the pwm line is orange. Check your schematic to make sure you line these up properly when connecting your servo. You should once again make use of the `gpio.c` library to control the `PWM0/1` pins.

BROWN	`GND`
RED	`VDD_SERVO`
ORANGE	`PWM0 / PWM1`

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

7.3 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 SVC handlers. The two system calls are as follows (defined in `code/kernel/src/syscall_servo.c`):

```
/**
 * @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 sys_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 sys_servo_set(uint8_t channel, uint8_t angle);
```

`sys_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. `sys_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 `sys_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**.

These syscalls will require you to do a little bit of design! You will need to figure out what frequency you want the timer to generate interrupts at and how you want to generate the pulse width. There are multiple ways that you can make these syscalls work, and some are simpler than others.

8 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:

```
Welcome to Servo Controller!
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
```

9 Bonus

9.1 Bonus: UART with DMA (10 points)

Read the documentation and the manual to figure out how the DMA engine works and implement UART with DMA. If you do so, include in your README.txt how you configured the DMA engine and how you are handling the DMA interrupts. You must also demonstrate that your implementation is not dropping any bytes. Points will not be awarded for partial solutions. Solutions will be subjected to private tests and you will only know if you received any points after the lab has been graded.

9.2 Bonus: Atomic Enter User Mode (5 points)

Read the comment at the top of the crt0.c file. Implement your own version of enter user mode that is able to jump to the user program, de-escalate privilege and swap the stack atomically. If you choose to do this, include in your README.txt how you achieved atomicity. Submissions will be hand graded, you will only know if you got the bonus points after the grade for the lab has been graded. For correctness, you must ensure that you are in main, with the correct argv and argc.

10 Assembly Tips and Calling Conventions

1. 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.

11 Hints for Lab 3

1. `printf()` may not print to your serial console if you don't append a newline (`\n`) to your format strings. You could also call `fflush()` to flush output to the serial console.

2. We're only handing out one servo, but you can test supporting multiple servo motors by enabling both channels, setting each channel to a different angle, and moving the servo between channels (unplugging it from channel 1 and plugging it into channel 2). We'll likely demo your code this way although we might use two motors. Don't worry – if your code works when moving one servo motor between channels, it's extremely likely that your code will work with two servo motors.

3. Oscilloscopes are extremely helpful for debugging issues with PWM! Every bench in HH 1303 should have an oscilloscope and cables can be found on the rack in the lab. If you don't remember how to use oscilloscopes from 18-100 (or if you never took that class) come to office hours! Any TA should be able to point you in the right direction.
4. Sometimes when plugging in the a servo, the LED driver will shut off. This is OK, to get the driver back up and running just restart the board.

12 Style

12.1 Doxygen

Doxygen is a framework that allows you to automatically generate documentation from comments and markup tags inserted directly into the source. This style of embedding documentation in source is often used in industry. We will be using `doxygen` for code documentation for this course moving forward. A tutorial is available here if unfamiliar with it:`http://www.doxygen.nl/manual/docblocks.html`. Steps for setup and use of Doxygen are below:

```
Installation in the VM:
$ sudo apt-get install doxygen

Generating documentation:
$ make doc
```

The handout code contains a default configuration file called `doxygen.conf`. While most of the tags provided in the file already have the correct value, you are responsible for making sure that the `INPUT` tag specifies the correct source files (for this lab and future labs). **Specifically, you will need to have zero doxygen warnings for `kernel`.**

If the above command runs successfully, then you should have a `/doxygen_docs` directory with an `index.html` file. View it locally in a browser to see the documentation created from the code you wrote. When running this command, a file called `doxygen.warn` should have been created in the directory you ran `make doc`. Open this file to inspect any warnings. If there are any documentation warnings in the file about code you have written, then fix them. The TAs will check this file and take off style points if there are *any* warnings in this file. Please see the TA written code in `349libk/` for example doxygen code documentation.

12.2 Good Documentation

Good code should be mostly self-documenting: your variable names and function calls should generally make it clear what you are doing. Comments should not describe what the code does, but why; what the code does should be self-evident. (Assume the reader knows C better than you do when you consider what is self-evident.)
There are several parts of your code that do generally deserve comments:

- File header: Each file should contain a comment describing the purpose of the file and how it fits in to the larger project. This is also a good place to put your name and email address.
- Function header: Each function should be prefaced with a comment describing the purpose of the function (in a sentence or two), the function's arguments and return value, any error cases that are relevant to the caller, any pertinent side effects, and any assumptions that the function makes.
- Large blocks of code: If a block of code is particularly long, a comment at the top can help the reader know what to expect as they're reading it, and let them skip it if it's not relevant.
- Tricky bits of code: If there's no way to make a bit of code self-evident, then it is acceptable to describe what it does with a comment. In particular, pointer arithmetic is something that often deserves a clarifying comment.
- Assembly: Assembly code is especially challenging to read – it should be thoroughly commented to show its purpose. This does not mean simply stating what the instruction does in the comment. You should explain what the instruction is doing in the context of your program (ex. "Increment the counter").

12.3 Good Use of Whitespace

Proper use of whitespace can greatly increase the readability of code. Every time you open a block of code (a function, "if" statement, "for" or "while" loop, etc.), you should indent one additional level.

You are free to use your own indent style, but you must be consistent: if you use four spaces as an indent in some places, you should not use a tab elsewhere. If you would like help configuring your editor to indent consistently, please feel free to ask the course staff. Make sure that whatever editor(s) you and your partners are using are configured the same way.

12.4 Good Variable Names

Variable names should be descriptive of the value stored in them. Local variables whose purpose is self-evident (e.g. loop counters or array indices) can be single letters. Parameters can be one (well-chosen) word. Global variables should probably be two or more words.

Multiple-word variables should be formatted consistently, both within and across variables. For example, "hashtable_array_size" or "hashtableArraySize" are both okay, but "hashtable_arraySize" is not. And if you were to use "hashtable_array_size" in one place, using "hashtableArray" somewhere else would not be okay.

12.5 Magic Numbers

Magic numbers are numbers in your code that have more meaning than simply their own values. For example, if you are reading data into a buffer by doing `fgets(stdin, buf, 256)`, 256 is a "magic number" because it represents the length of your buffer. On the other hand, if you were counting by even numbers by doing `for (int i = 0; i < MAX; i += 2)`, 2 is not a magic number, because it simply means that you are counting by 2s.

You should use `#define` to clarify the meaning of magic numbers. In the above example, doing `#define BUFLEN (256)` and then using the `BUFLEN` constant in both the declaration of `buf` and the call to `fgets`.
This is especially important when putting constants in memory-mapped registers.

12.6 No "Dead Code"

"Dead code" is code that is not run when your program runs, either under normal or exceptional circumstances. These include `printk` statements you used for debugging purposes but since commented out. Your submission should have no "dead code" in it.

12.7 Modularity of Code

You should strive to make your code modular. On a low level, this means that you should not needlessly repeat blocks of code if they can be extracted out into a function, and that long functions that perform several tasks should be split into sub-functions when practical. On a high level, this means that code that performs different functions should be separated into different modules; for example, if your code requires a hashtable, the code to manipulate the hashtable should be separate from the code that uses the hashtable, and should be accessed only through a few well-chosen functions.

12.8 Consistency

This style guide purposefully leaves many choices up to you (for example, where the curly braces go, whether one-line "if" statements need braces, how far to indent each level). It is important that, whatever choices you make, you remain consistent about them. Nothing is more distracting to someone reading your code than random style changes.

13 Submission

13.1 Canvas

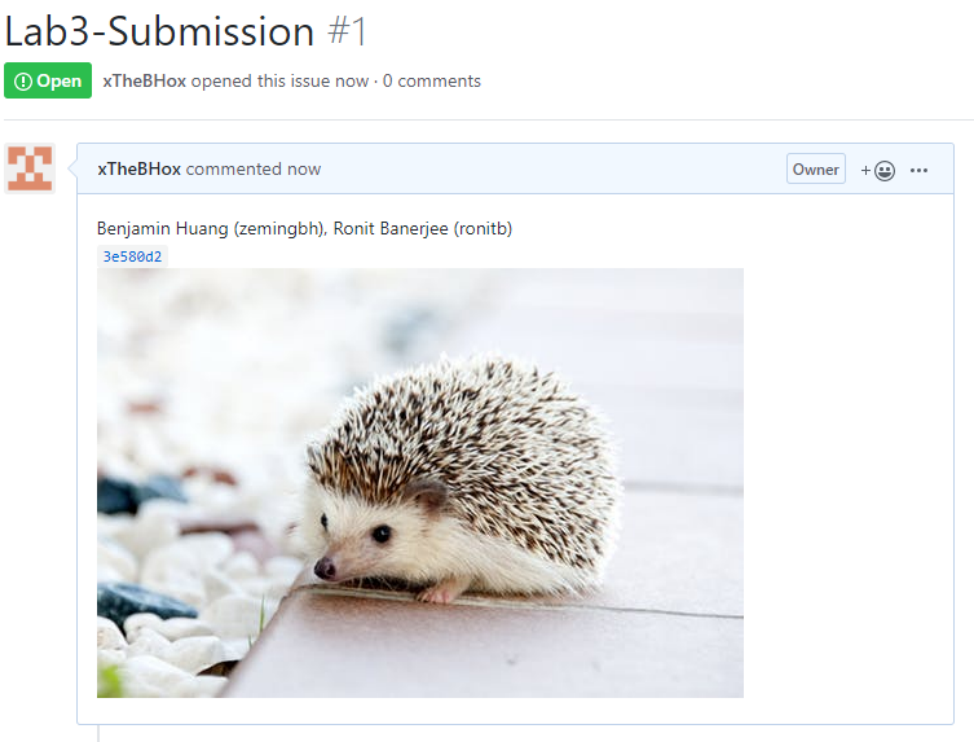
Submit the link to your repository to Canvas (you can do this right when you start the lab). Only one partner needs to do this.

13.2 Github

If you have any comments or suggestions, please feel free to let us know in README.txt.

1. To submit your checkpoint, create an issue on GitHub titled `Lab3-Checkpoint`. In the comments section include your and your partner's name, AndrewID, and the commit-hash you want to submit.
2. To submit the completed lab, create an issue on GitHub titled `Lab3-Submission`. In the comments section include your and your partner's name, AndrewID, and the commit-hash you want to submit.
3. If you completed the bonus, please submit an additional issue, titled `Lab3-SubmissionBonus`, with the bonus commit hash.

When you are done, it should look like the screenshot below. Pictures of cute animals are not needed but are recommended.



13.3 Demo

You do not need to demo the checkpoint, but your TA may check in with you about it.

To demo the final submission, meet a TA (in OH or a separate meeting) and show your usermode program successfully controlling the servo.