Lab 3: IRQs and Hardware

18–342 Fundamentals of Embedded Systems

Out: October 27, 2013, 00:05am EDT
Due: November 7, 2013, 11:59pm EDT

Contents

1 Introduction

1.1 Overview

This lab focuses on the input-output aspect of real-time embedded systems. Up until now, all of your input-ouput functionality depended on using Linux or helper routines provided to you by U-boot. In this lab, you will be given a new kernel infrastructure. We shall call this kernel **Gravel**. For this lab, Gravel will be a single process kernel, not unlike the micro-kernel you wrote for your previous lab. You will learn how to make Gravel converse with hardware modules and devices that are not directly on the ARM processor. You will also get these devices to talk back to you through the use of interrupts. To this end, you will:

- Familiarize yourself with the new kernel infrastructure.
- Port your SWI handling code to this new infrastructure.
- Port your exit, read and write syscalls to this new infrastructure.
- Write a timer driver for the built-in OS Timers.
- Write interrupt management and dispatch code for Gravel.
- Write programs that demonstrate your timer.

Please note that this lab will be significantly harder than previous labs because of time constraints and the inherent complexity of writing drivers for external devices. **Start very early — as soon as the lab comes out**. If the lab handout feels unclear at first, don't panic. Read the handout a number of times until the entire picture becomes clear. Read the entire handout completely before you begin coding. If something is still unclear, send an email to the staff or use the discussion board.

Furthermore, this lab provides an opportunity for you to earn some extra credits. You may be able to earn up to 10 extra points on this lab for producing sufficiently complex test cases. Further details are presented later in this handout. There will be plenty of opportunity to express your design skills in this lab.

1.2 Tasks

We will list all the tasks that you will have to perform to receive full credit in the lab here. You may use this as a check list while you complete this lab. In this lab, you will:

1. Port your SWI handler and syscall code to the Gravel kernel *(15 points)*.
2. Write a timer driver that correctly reports OS timer events using interrupts *(40 points)*.
3. Write two new syscall handlers to provide the kernel time to the user *(20 points)*.
4. Write a program that times how long it takes the user to type a sentence *(7 points)*.
5. Write a program that displays a spinning cursor to test your timer *(8 points)*.
6. Write documentation of your design, implementation and testing, use good programming practices and write clean, reusable code *(10 points)*.
7. Write valuable test cases *($\leq$10 points)*.

These tasks are explained in detail below.

1.3 Expectations

You and your partners need to start early to ensure that enough time is available to complete the lab.

We again emphasize that all submitted code must compile and execute properly. Code submitted that fails to compile will receive a failing grade. Therefore it is *essential* that all code is tested on the gumstix hardware prior to submission. Please also make sure that file names and cases are correct.

As in the previous lab, 10% of the lab's grade is devoted to code style. Your code must be well commented and must exhibit elements of good software practices. Please put header guards in your header files. Do not include *.c* files. Modularize your solution into a number of relevant, properly named files. If you find yourself using the same code over and over again, it should go into a function. If you find that all of your code needs to know about all other parts of your code, you may be implementing a holographic design. This is discouraged. Modularizing, abstraction and data hiding are all encouraged. Each file should contain your name and a brief description of the file at minimum. You should abstract out constants cleanly with usefully named *#defines*. Convoluted code, unnecessary and unhelpful comments and spaghetti code will be severely penalized. You will be using code from the last lab in this lab and you will be using code from this lab in the next lab. Please write clean, quality code. Use the ECE/andrew machines to make sure that files are named correctly and that your build is accurate.

We expect to be able to upload the code to the gumstix hardware (or use the cross-compiler on a linux machine), type "`make`", and be able to execute your applications without any modification or renaming of files. Any modifications (besides ones for our grading harness) that we have to make will result in grade deductions. The make infrastructure in this lab is vastly different from the previous lab. Please carefully consider the Makefiles and don't edit them unnecessarily. Any edits that you make must be carefully documented. Please do not turn in any unnecessary changes to the Makefile such as modifying compilation flags (except the `CCPREFIX` to allow the use of cross compiler[1]). As always, create files with appropriate names.

At the end of this lab, you must turn in all source code, including the support code and any new files you create, following the submission procedure specified in Section 9.1.

1.4 Lab Support Code

Plese download the support code for this lab from Blackboard. Throughout this lab we will ask you to modify the support code in order to implement your lab solution. At the end of the lab, you must turn in these files according to the submission procedure specified in Section 9.1.

1.5 Debugging

One of the goals of this course is to train you to be a good embedded systems programmers. To this end, we give you hard problems with numerous solutions — all of them with varying trade-offs. During the course of completing this lab, it is highly likely that you will run into problems that need to be debugged. Please refrain from contacting your TA the moment your code fails to compile or executes incorrectly. Debugging kernel code involves intimate knowledge of the platform, tools and the code. We have the first two. Only you can possibly have the third. If you don't know why your code is crashing, we probably don't either. We will do our best to help you understand overarching issues and the overall code, but we don't know why your interrupt handler is stuck

If you have explored all options and are still having an issue, then contact the instructor or a TA. Explain everything you have done in detail and we may be able to assist with any conceptual or logistical issues that you may be having. Your instructor and TAs cannot assist you in debugging your code and will not read your code unless absolutely necessary.

[1]The compiler on verdex-pro boards will not work for this lab and you will need to use the cross compiler from now on

2 Kernel Infrastructure

Your peers in CS have to write an OS kernel not much unlike the one you are about to write. Since the kernel you are writing in this class is a lean and mean real-time kernel it is named **Gravel** – a subset of the larger Pebbles kernel that your peers in CS have to write. Gravel will eventually be a multi-threaded, shared memory, pre-emptive real time kernel. It will eventually support all the functions mentioned in the kernel API [1]. For this lab, you will be required to implement a subset of this functionality. In this lab, Gravel will only handle one thread and will not implement any memory protection.

2.1 Build Infrastructure

Before you begin implementing all the kernel functionality, a tour of the kernel architecture and its build infrastructure is in order. A new unified build system has been introduced for this project [2]. Please take a moment to read through the Makefile and familiarize yourself with its general structure. The primary files are:

- `lab3/Makefile`
- `lab3/kernel/kernel.mk`
- `lab3/kernel/arm/kernel.mk`
- `lab3/uboot/uboot.mk`
- `lab3/tasks/tasks.mk`

Here is a list of interesting variables that you may want to tweak. Note that when we compile your code, we will replace your Makefile with our own. Hence, any lasting changes may only be made in the other mk files. Note that using the mk files to subvert the variables in Makefile may result in the deduction of a significant number of points . If you feel like you desperately need to change a particular aspect of the Makefile, send an email to the staff list and we may push out a change that lets you do exactly that.

- `PACKAGES` — This variable lists the name of every program that you would like to compile for your kernel. It is being called a package instead of a program because in later labs, you will use the same infrastructure to load multiple tasks in your kernel at once. Note that if you define a package name here, a corresponding directory with the exact package name must be created in `lab3/tasks/` directory. This directory must contain a valid `pack.mk` to describe the objects in the package and how to construct them. Take a look at the three sample programs taken from the previous lab to get an idea of how to construct a valid `pack.mk`.
- `CCPREFIX` — This variable is prepended to `gcc`, `ld`, `ar` and `objcopy`. This is useful to retool the system when you are using cross compilers. For example, since your cross compiler is `arm-linux-gcc`, you need to set `CCPREFIX` to `arm-linux-`.
- `CWARNINGS_NOISY` — This variable contains warnings that are not enabled by default. These warnings are very useful and may help you track down some really nasty bugs. On the flip-side these warnings tend to generate many false positives. Unless you code very carefully, it is very easy to get swamped by warnings using these flags. You may include them to assist your debugging endeavors but we will not compile your code under these flags.
- `LOAD_ADDR` — If you wish to load your programs at a different load address (I really don't know why you would), you may use these flags.

If the target rules confuse you, please read the GNU Make manual [4] for a detailed explanation of the different features used.

[2] The original incarnation of the build infrastructure was developed by Nathaniel Filardo (`nwf@andrew.cmu.edu`) for use in 15-410.

2.2 Primary Execution Environment

The execution environment and interface to U-boot for this lab is extremely similar to that of Lab 2. Your kernel is loaded at 0xa3000000. The major difference is that the **user program is loaded at 0xa0000000**. The user stack for this lab with be a full descending stack starting at 0xa3000000. The supervisor stack is set up by U-boot and is a full descending stack starting at 0xa3ededf4. Here is a pictorial representation of the memory layout. All other stack are discussed later as they are introduced in their relevant section.

Start Address	End Address	Type
a3f00000	...	U-Boot Code
a3edf000	a3effffff	Heap (malloc)
a3edee00	a3edefff	U-Boot Global Data Struct
a3ededf4	a3ededff	Abort Stack
...	a3ededf3	Supervisor Stack
a3000000	...	Kernel start
...	a2ffffff	User Stack
a0000000	...	User program start + heap

The kernel has been configured to begin execution at `_start`. `_start` transfers control to `kmain` (instead of a `main` function, your kernel code will have a `kmain` function), where you can insert your initialization code.

2.3 The C Library

Gravel will be linked against U-boot's function table. User programs for this kernel will not be linked against u-boot. They will instead be setup to link against libc. It is your responsibility to port your libc over from your previous lab into `lab3/tasks/libc` correctly. Due to popular demand, a number of userland functionalities have been added to this lab's libc. These include stdio, stdlib, ctype and string functions from the standard C headers. These are provided to as-is. Please use them wisely. These library functions are meant to be used by application programs and cannot be accessed from within the kernel.

Here is a list of library functions that have been provided. Please refer to the man pages [10] and K&R [9] for more details and a formal specification of these functions [3].

2.3.1 Functions in ctype.h

```
int isascii(int);
int iscntrl(int);
int isdigit(int);
int isgraph(int);
int islower(int);
int isprint(int);
int isspace(int);
int isupper(int);
int isxdigit(int);
int isalpha(int);
int isalnum(int);
int ispunct(int);
int toupper(int);
int tolower(int);
```

[3] Some of the provided functions don't have a complete implementations. For example, printf does not support floating point numbers and some of the more esoteric format specifiers. Overflow behaviour of atoi and strtoul deviate from the ANSI C specification. You should never be actually encountering these situations in code that you write though.

2.3.2 Functions in stdarg.h

```
void va_start(va_list, name);
type va_arg(va_list, type);
void va_end(va_list);
```

2.3.3 Functions in stdio.h

```
int putchar(int);
int puts(const char *);
int printf(const char *, ...);
int vprintf(const char *, va_list);
int sprintf(char *, const char *, ...);
int snprintf(char *, size_t, const char *, ...);
int vsprintf(char *, const char *, va_list );
int vsnprintf(char *, size_t, const char *, va_list );
int sscanf(const char *, const char *, ...);
void hexdump(void *, size_t);
```

The hexdump function is an additional function that dumps words from a given location for a given length, four bytes at a time, in hexdump format.

2.3.4 Functions in stdlib.h

```
long atol(const char *);
int atoi(const char *);
long strtol(const char *, char **, int);
unsigned long strtoul(const char *, char **, int);
```

2.3.5 Functions in string.h

```
size_t strlen(const char *);
char *strcpy(char *, const char *);
char *strncpy(char *, const char *, size_t);
char *strdup(const char *);
char *strcat(char *, const char *);
char *strncat(char *, const char *, size_t);
int strcmp(const char *, const char *);
int strncmp(const char *, const char *, size_t);
char *strchr(const char *, int);
char *strrchr(const char *, int);
char *strstr(const char *, const char *);
char *strpbrk(const char *, const char *);
size_t strspn(const char *, const char *);
size_t strcspn(const char *, const char *);

void *memset(void *, int h, size_t);
int memcmp(const void *, const void *, size_t);
void *memcpy(void *, const void *, size_t);
void *memmove(void *, const void *, size_t);
```

3 Syscalls Revisited

In your last lab, you were asked to implement a system call interface. This task involved providing userland syscall stubs [4] as part of libc and also implementing a SWI installation, handling and dispatch mechanism in the kernel. In this lab we provide you with an opportunity to revisit your design and implementation from the previous lab. You will port your code to the new infrastructure. This will be simple and will require minimal coding changes from your part. You will then add additional syscall stubs and handlers. These additional syscalls will allow userland programs to exploit additional kernel functionality that you will implement later in this lab. You will continue using AAPCS and OABI [3]. Please refer to the last lab for details on the calling conventions. Please refer to the 18–342 Gravel kernel API [1] for the exact specification of each syscall.

3.1 Porting the old Syscalls

When your kernel is complete, it should support `read`, `write`, and `exit`. Stubs for these syscalls are provided in `lab3/tasks/libc/swi/`. Fill out the stubs with code from your last lab. No stubs have been provided for the kernel side SWI handler. You can copy over your old SWI dispatcher or redesign it from scratch if you didn't like your old design. The SWI argument convention remains unchanged from lab 2.

3.2 Hijacking U-boot's SWI handler

In this lab, you will again need to wire in your SWI handler. You may use code from your previous lab, but note that you may want to clean up and generalize this code. You will be using it later in the lab to wire in IRQ handlers which require a very similar operation, just with different offsets into the exception table.

3.3 Getting into user mode

Again, you will port code from the previous lab. This should be a simple task. Simply call the functions that enter user mode from lab 2. Make sure that you update the entry address as all of your user applications are now linked to enter at 0xa0000000. If you properly modularized your code from last time, this should be as simple as changing a #define. Maintain this code well as you will use this in your next lab.

Please copy over your code for `crt0` and the kernel code to pass arguments to the user. You will implement argument passing using the specification from lab 2. The only difference is in the top of the user stack. The user stack for this lab grows down from 0xa3000000. The argument passing convention remains the same as lab 2.

3.4 New Syscalls

If you read `lab3/tasks/libc/include/bits/swi.h`, you will notice that a few new syscall numbers have been added. These syscalls are documented in the Gravel kernel API. For this lab, you are implementing the `time` syscall and the `sleep` syscall. The former is used to retrieve the time in milliseconds since the kernel booted up. The latter is used to suspend the execution of the user program for a certain number of milliseconds. Please implement the libc syscall wrappers for these two syscalls.

3.5 Progress Check

If you completed the tasks in this section, you will now have a kernel that supports executing test programs from lab2. Take a peek at how `lab3/tasks/hello/pack.mk` is written. See if you can copy over rot13 from your previous lab to the `lab3/tasks/` folder and give it a correct `pack.mk`. Do not forget to add the package

[4] There is a lot of confusing nomenclature in industry and in the literature. When one usually talks about stubs or wrappers, one is merely talking about a small piece of code that interfaces two larger bodies of code in some manner. They are also called shim-layers in some situations. In this document, whenever you read syscall "stub" or syscall "wrapper", please understand that this refers to the userspace libc assembly wrappers that perform SWIs. The kernel side assembly wrapper is called the SWI wrapper or the SWI dispatcher.

name to the PACKAGES variable in the main Makefile. The TAs are going to replace your Makefile when we test your code. Hence, we are not worried about how many of your own packages you add to PACKAGES. The test programs from your previous labs should all work flawlessly in your current environment. Follow the instructions from lab 2 to get U-boot to load your kernel at 0xa3000000. Load your user program of choice at 0xa0000000. Run your kernel at 0xa3000000. You should see the program run and complete successfully. Congratulations! You now have a working kernel.

4 Communicating with Devices

4.1 Introduction

Up until now, the code you have written primarily dealt with subsystems of the ARM processor. You had to deal with mode changes, privilege management, software interrupts and a host of other details. Regardless, all of this activity was to fundamentally manage features of a single device — the ARM processor. In the real world, your embedded system is not going to run code in isolation. You will need to communicate with other devices and with code running on those devices. There are a number of techniques used in industry to cleanly, consistently and uniformly communicate with devices. In this lab, we shall learn about some of them — namely, memory-mapped registers and hardware interrupts.

A lot of the general theoretical aspects of these techniques have been covered in lectures. This section will give you details on how these general concepts are applied to the ARM processor, and in particular, to the Intel XScale PXA255 or the PXA270 (for verdex-pro boards) processor architecture that your gumstix uses. This introduction is in no way complete. The authoritative specifications are the Intel PXA255/PXA270 Processor Developer's Manual [6, 7], the Intel XScale Architecture Manual [8], and the ARM Architecture Reference Manual [2].

4.2 Memory Mapped Registers

One of the standard ways to uniformly communicate with devices is to use memory mapped registers. Suppose you have a device that exports a number of attributes, each of which can be controlled through some registers. One way a programmer could modify these registers is to treat these registers as if they were locations in memory. Then the programmer could read from them, write to them, move them around, so on and so forth. Architectures usually facilitate this by setting aside some region in the physical address space. Hardware engineers can then hook up devices in such a way that accesses to these reserved memory locations will be correctly routed to the appropriate device that is mapped at that location. The ARM architecture does precisely this. Here is a reproduction of the memory map of the PXA255. A similar but much more detailed memory map for the PXA27x processor on the verdex-pro board is shown Figure 28-2 on page 28-2 of the PXA270 Manual [7].

Start Address	End Address	Type
a0000000	a3ffffff	SDRAM (64 MB)
48000000	4bffffff	Memory Controller (MMIO)
40000000	43ffffff	Memory Mapped Registers
00000000	00ffffff	StrataFlash ROM (16 MB)

Writes and reads to the region between 0x40000000 and 0x44000000 are intercepted by the memory subsystem of the processor and rerouted to particular device registers. A detailed description of all the registers and their addresses is provided in Table 28-8 starting on page 28.13 of the PXA270 Processor Developer's Manual [7]. *Please read these sections of the manual. It is of paramount importance that you understand the underlying system before you start churning out code.* Please make a note of the register maps for OS Timer and Interrupt Controller as these will be used in the next section. Reads and writes to these locations will change the appropriate values in the device registers.

A property to remember when using memory mapped registers is that device registers are, in a fundamental way, not regular memory. In a lot of situations, they do not guarantee the idempotence of their read and write operations — the property of memory wherein multiple sequential reads or writes of the same

value from or to the same memory location is equivalent to a single read or write of that value to the same location. They also introduce new optimization hazards which is of relevance to compiler designers, processor architects and system programmers. The compiler is not free to assume that write or read reordering is acceptable when dealing with memory mapped registers. Hence, when dealing with memory mapped regions, one must inform the compiler of these special properties. In the C language for the gcc compiler, you can do this by using the **volatile** keyword. One must also disable write coalescing, write re-ordering and caching in the processor for those particular memory regions. Since we have not enabled caching in our processor, we do not need to worry about the latter issues. Be aware that in the real world, using a virtual memory system, multiple processors, out-of-order execution or an aggressive compiler can all bring their own demands when dealing with memory mapped registers. These issues are solved by using a mixture of cross processor interrupts, memory barriers, stringent compilation flags, ritual sacrifices and lots of prayers.

4.3 Interrupts

Memory mapping is a technique used to transfer information to and from the primary processor, but this transfer is always initiated by the primary processor, sometimes in a tight loop. Tight looping (also called polling) on a condition can be very wasteful in terms of processor resources and bus bandwidth resources. In some situations, it can even slow down the target device. Interrupts are an asynchronous method that devices can use to signal to the primary processor that it needs attention. A device can have no interrupts, one interrupt or a number of different interrupts to signal different events.

When an interrupt is fired by an external device, the ARM processor vectors to the IRQ handler (assuming the device was configured to generate and IRQ) and executes the handler in IRQ mode. sp and lr are banked out, and cpsr is saved into the banked spsr. The interrupt handler must now appropriately acknowledge the interrupt, pacify the signalling device and then return to the original caller without disrupting its register state. The exact details of the IRQ vectoring process is described in section A2.6.8 on page A2-24 of the ARM Architecture Manual for more details [2].

When the ARM vectors to an interrupt handler, IRQs are disabled until the programmer enables them. For performance reasons, a lot of real time systems enable interrupts as soon as possible. This allows for nested delivery of interrupts and decreases system interrupt latency. Allowing nested interrupts significantly complicate a multi-threaded system. Since we are only dealing with a single interrupt on a single threaded system, there is no benefit to building a nested interrupt system. Hence, we will not require you to support nested interrupts and you may run your interrupt handlers with IRQs disabled for the entire duration. On the flip side, since we are allowing you to run non-preemptible interrupt handlers, you must be careful to not perform long computations inside the context of an interrupt handler. As a rule of thumb, do not perform anything that doesn't have a constant bounded computation time ($\mathcal{O}(1)$ time complexity).

4.4 The Interrupt Controller

You may have noticed that nowhere in the above description is there a mention on any way to determine which device raised an interrupt. This is deliberately not mentioned in the ARM architecture so as to make it implementation dependent. On the PXA255 and PXA270 processors, there is an interrupt controller that takes care of arbitering interrupt assertions from multiple devices. All devices are connected to the interrupt controller instead of the ARM processor, in level sensitive mode. When the interrupt controller notices an external device signalling an interrupt and knows that this interrupt is not masked (masking can also be controlled by writing to the interrupt controller's memory mapped registers), it asserts an IRQ/FIQ with the ARM processor. The ARM processor can then read the status registers in the interrupt controller to determine which device actually caused the interrupt to be fired. This interrupt controller is described in Section 25.4.1 and Section 25.5 for the Intel PXA270 Processor [7]. Here is a synopsis. The interrupt controller has a number of registers that are of importance to us. These are the ICMR, ICPR, and ICLR (verdex-pro users should ignore ICMR2, ICPR2 and ICLR2 for this lab).

ICMR The Interrupt Controller Mask register is a 32 bit read-write register that contains a mask bit per interrupt source. It controls whether a particular device interrupt generates a processor IRQ/FIQ or

not. If an ICMR bit is set, the corresponding interrupt is delivered. If an ICMR bit is not set, the corresponding interrupt is masked and not delivered to the ARM processor.

ICPR The interrupt Controller Pending register is a 32 bit read-only register that shows all active interrupts in the system. The bits in this register are not affected by ICMR bit masks and hence, reading the ICPR gives the true state of interrupts in the system. This register can be used to determine the cause of the current interrupt. The bits in this register are automatically cleared when the corresponding interrupt is cleared at the source. A list of which interrupt corresponds to which device is given in the developer's manual.

ICLR The Interrupt Controller Level register is a 32 bit read-write register that controls whether pending interrupts generate FIQs or IRQs. If an interrupt is masked, then the corresponding ICLR bit has no effect. Otherwise, the corresponding interrupt is routed as an IRQ if the ICLR bit is low and routed as an FIQ if the ICLR bit is high.

Note that we are only using the timer interrupt and routing it as an IRQ. You should configure your interrupt controller accordingly.

5 Writing a Timer Driver

In this section of the lab, you will be writing a timer driver that exposes functionality of the OS Timer built into the PXA255/270 processor. The PXA255/270 has a two major timers — the real-time timer and the OS timer. We will be using the OS timer in this lab. The OS timer is clocked by a 3.6864 MHz oscillator for basix boards and 3.25 MHz for verdex-pro boards (verdex-pro users should ignore the 8-channel timer for this lab) (*please memorize this number as you will be asked to recall this number on the exam!*). The OS timer contains four match registers (OSMR0-OSMR3), a counter register (OSCR), a status register (OSSR) and an interrupt-enable register (OIER). You will be writing a driver in your kernel that will utilize these resources to keep track of time within your kernel. You will also implement two system calls that allow user programs to ask for the time and to sleep for a certain period of time.

5.1 Register Description

We shall now provide a brief overview of the OS Timer registers. Authoritative details are present in Section 22.5 of the Intel PXA270 Processor Developer's Manual [7].

OSCR The OS Timer Count register is a 32-bit count-up counter that increments on the rising edge of the source 3.25 MHz (for verdex-pro boards) clock. It is a read write register. There may be a latch delay from the time the register is written to and the time the register is actually updated.

OIER The OS Timer Interrupt Enable register has four non-reserved bits. Each bit controls whether the corresponding OS Match register is active. If a bit is high in this register, then a match between the OSCR and the corresponding OSMR will result in the corresponding bit in OSSR to be set.

OSMRx The OS Timer Match register(#) is a set of four registers. Each register is a 32-bit read-write register that holds a timer counter target. If the target in the match register matches the current value in OSCR and the corresponding interrupt enable bit is set, then the corresponding OSSR bit is set. OSMR3 can be used as a watch dog timer, but we shall not exercise this functionality in this lab.

OSSR The OS Timer Status register holds bits indicating that a match had occurred between the OSCR and the corresponding OSMR. This register is hooked up to the system interrupt controller. Writing a zero to this register has no effect. Writing a 1 to a bit acknowledges the match and hence, clears that bit.

5.2 Driver Requirements

In this lab, you are required to set up the interrupt controller and the OS timer to deliver interrupts. You are required to "wire in" an IRQ handler. This is very similar to wiring in a SWI handler. You will use the same technique (and possibly the same code) to check if the ARM vector table contains an `ldr pc, pc, #x` instruction and use that to hijack an 8-byte region in U-boot code sitting in RAM. If you need to recall how to do this, please consult the Lab 2 handout. The IRQ handler must check that a timer interrupt has occurred. If one has occurred, it must make a note of this event and reload the match registers with the appropriate values. If the interrupt is not a timer interrupt (which should not be the case if you masked all other interrupts), the IRQ handler must ignore it. In this lab, we are only going to use OSMR0. Please store your timer match targets in OSMR0 only. Furthermore, try not to accumulate any artificial drift. Load your match registers so as to not introduce drift in your timing. The more meticulous you are, the more style points you get. You may perform a one time initialization of the other OSMRs if you feel like it, but you may not update them in your interrupt handler.

Please note that the IRQ handler uses a different sp than the supervisor sp. You are responsible for initializing the IRQ sp such that it points to a legal location. Where you decide to place the IRQ stack is a design decision that you will have to make and subsequently document. Possible approaches involve cordoning off a region of DRAM at some fixed address to act as the IRQ stack. Another one involves statically allocating storage in the program data/bss sections. Other approaches may be appropriate as well. Please make sure that you use a timer resolution of at least 10 ms. Your IRQ handler should under no circumstance corrupt any user memory or registers.

5.3 Support Code

Significant support code has been provided to help you with this lab. You may choose to completely ignore the code given, but you may not delete the files given to you. Look in the `lab3/kernel/include/arm/` directory for arm related helper definitions and memory-mapped register read/write functions. You have also been given debug utilities and some assembly macros in `lab3/kernel/include/`.

5.4 Syscalls

Once you have the driver working, you can now integrate it will the time and sleep syscall. Follow the kernel API and implement those syscalls. It should be a simple matter of leveraging the functionality of your timer driver. Do not implement these with calls to U-boot.

5.5 Progress Check

If you have completed this lab, you will have a kernel that now handles timer interrupts. You may now write a simple test program to exercise the syscalls to sanity check your kernel. Congratulations, you have finished most of this lab. The only part remaining is a test program that will help you exercise your kernel. We recommend that you write tests other than the one that we require you to write.

6 Test programs

Now that the kernel is complete, you will test this kernel by writing two small application that exercise its features. The first one displays a spinning cursor and the second one times the user's typing speed. Here is what these program must do:

6.1 Splat

- The program must display a spinning cursor. The cursor must transition between the following states: `|/-\` and must transition between them once every 200 ms. In other words, the glyph must look like a spinning bar that completes one rotation every 1.6 s.

- The program should never terminate.
- The program should not read user input.

6.2 Typo

- The program must present the user with a prompt to type characters. Feel free to choose what your prompt looks like.
- The user should be able to type a line of characters and press return.
- The program now echoes what has already been typed again.
- On the next line, the program should print out the time it took for the user to type that line (accurate to a tenth of a second) starting from when the prompt was displayed.
- The program must present the prompt again.
- The program should never terminate.

You may assume a maximum line length, but your program **may not crash** if this line length is exceeded. It may ignore characters, drop them or perform something different, but it may not crash.

Here is some sample output where the > character was used as the prompt:

```
> Hello World
Hello World
2.5s
> My hovercraft is full of eels.
My hovercraft is full of eels.
3.8s
```

6.3 Progress Check

You are now code complete. Test! Test! Test! Go back and document your code. Clean up any crud and polish your code. You are now ready to submit your code. Follow the submission procedure. Celebrate (until the next lab comes out)!

7 Extra Credit

You may earn up to 10 points extra in this lab if you submit a working kernel and write extra test cases. We will examine your tasks directory to see if you have written any interesting packages that stress your kernel. You may write straight out corner-case testers. You may write elaborate programs that do something interesting with the given syscalls (like a whack-a-mole game that draws 9 squares and shows when moles pop-up at random and get hit). Note that all of these programs must adhere to the given kernel interface and must themselves be correct. **They must also not be trivial.** Handing in 20 programs, all of which display the month, day or second since reboot in different formats is not going to get you any extra credit. Although we do strive to not make you work against your class mates, the conditions for awarding extra credit are a bit subjective due to the difficult task of defining a "non-trivial" test case. An interesting (note: interesting and complicated are independent concepts), unique test case will get 5 points. Providing a battery of tests that show some degree of thorough co-ordination and meticulousness will result in even more points. You may earn up to 10 points and no more. If you feel that you can do something very interesting but need an extra syscall or some modification to the spec, send an email to the course staff. These requests will be evaluated on a case by case basis.

8 Plan of Attack

If you are feeling completely lost as to how to approach this lab, here is a rough timeline to keep you on track.

1. Read through the Makefiles and provided header files.
2. Copy over your `read.S`, `write.S` and `exit.S` and `crt0.S` from your previous lab into this lab's libc.
3. Port the hijacking/wire-in code over – generalize it if possible.
4. Port the "go to user mode" functionality.
5. Port the SWI handlers.
6. Test your kernel with `hello`.
7. Write the syscall wrappers for the two new syscalls.
8. Change your hijacking code to now hijack IRQs.
9. Write the interrupt controller setup code. Hardcode values in at first and generalize it later once you are sure the code works.
10. Write the timer setup code.
11. Make a dummy IRQ handler that prints "ababab" alternately. Modify the timer interval to a couple times every second. This way you are not swamped with output. Make sure you change these back when you are done debugging.
12. Make sure your timer now works.
13. Go in and clean up your code and add checks.
14. Implement global state variables to maintain time.
15. Implement the syscalls to return time and sleep.
16. Test, test, test!
17. Polish and turn in.

9 Completing the Lab

9.1 What to Turn In

When finished with the lab, please submit the following source code and project files in an archive lab3-group-XX.tar.gz to blackboard, where XX is your lab group number (maintain the directory paths in the archive if you donot want to lose points). Only one member per group needs to submit the code on Blackboard (please follow the same instructions as Labs 1 and 2).

- `lab3/Makefile`
- `lab3/kernel/arm/kernel.mk`
- `lab3/kernel/arm/psr.c`
- `lab3/kernel/arm/reg.c`
- `lab3/kernel/assert.c`
- `lab3/kernel/include/arm/exception.h`

- lab3/kernel/include/arm/interrupt.h
- lab3/kernel/include/arm/psr.h
- lab3/kernel/include/arm/reg.h
- lab3/kernel/include/arm/timer.h
- lab3/kernel/include/asm.h
- lab3/kernel/include/assert.h
- lab3/kernel/include/bits/errno.h
- lab3/kernel/include/bits/fileno.h
- lab3/kernel/include/bits/swi.h
- lab3/kernel/include/config.h
- lab3/kernel/include/inline.h
- lab3/kernel/include/stdarg.h
- lab3/kernel/include/types.h
- lab3/kernel/kernel.mk
- lab3/kernel/main.c
- lab3/kernel/start.S
- lab3/mount.sh
- lab3/tasks/exit/exit.S
- lab3/tasks/exit/pack.mk
- lab3/tasks/hello/hello.c
- lab3/tasks/hello/pack.mk
- lab3/tasks/libc/crt0.S
- lab3/tasks/libc/include/asm.h
- lab3/tasks/libc/include/bits/errno.h
- lab3/tasks/libc/include/bits/fileno.h
- lab3/tasks/libc/include/bits/swi.h
- lab3/tasks/libc/include/bits/types.h
- lab3/tasks/libc/include/ctype.h
- lab3/tasks/libc/include/errno.h
- lab3/tasks/libc/include/inline.h
- lab3/tasks/libc/include/stdarg.h
- lab3/tasks/libc/include/stdio.h
- lab3/tasks/libc/include/stdlib.h

- lab3/tasks/libc/include/string.h
- lab3/tasks/libc/include/sys/types.h
- lab3/tasks/libc/include/unistd.h
- lab3/tasks/libc/libc.mk
- lab3/tasks/libc/stdio/*
- lab3/tasks/libc/stdlib/*
- lab3/tasks/libc/string/*
- lab3/tasks/libc/swi/exit.S
- lab3/tasks/libc/swi/libc.mk
- lab3/tasks/libc/swi/read.S
- lab3/tasks/libc/swi/sleep.S
- lab3/tasks/libc/swi/time.S
- lab3/tasks/libc/swi/write.S
- lab3/tasks/skyeye.conf
- lab3/tasks/tasks.mk
- lab3/tasks/typo/pack.mk
- lab3/tasks/typo/typo.c
- lab3/tasks/splat/pack.mk
- lab3/tasks/splat/splat.c
- lab3/uboot/include/_exports.h
- lab3/uboot/include/exports.h
- lab3/uboot/stubs.c
- lab3/uboot/uboot.mk

Please also submit any additional source files that you may have created and that are required for successful compilation of your project code. Feel free to organize your project in any way you see fit as long as the above layout is maintained and any extra files are submitted.

9.2 Where to Get Help

Please read the relevant sections of documents mentioned in the reference section. By now, you should be reasonably comfortable with using the GNU assembler and Makefiles. If you need help, consult the GAS manual [5] and the Make manual [4].

Come to office hours. Ask the instructor conceptual questions and questions about specification.

Post a thread on the discussion board for all questions regarding the lab. This is the fastest way to get help with the lab outside of office hours.

References

[1] 18-342 Course Staff. *Gravel Kernel API*, Oct. 2011.

[2] ARM Limited. *ARM Architecture Reference Manual*, June 2000. Available from: `http://www.arm.com/community/university/eulaarmarm.html`.

[3] ARM Limited. *ARM Architecure Procedure Calling Standard (AAPCS)*, Apr. 2008. Available from: `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042b/IHI0042B_aapcs.pdf`.

[4] Free Software Foundation. GNU Make [online]. Available from: `http://www.gnu.org/software/make/`.

[5] Free Software Foundation. *Using as*, Aug. 2007. Available from: `http://sourceware.org/binutils/docs-2.18/as/`.

[6] Intel Corporation. *Intel PXA255 Processor Developer's Manual*, Jan. 2004. Available from: `http://pubs.gumstix.com/documents/PXADocumentation/PXA255/PXA255ProcessorDevelopersManual[278693-002].pdf`.

[7] Intel Corporation. *Intel PXA27x Processor Developer's Manual*, Oct. 2004. Available from: `http://mmpod.googlecode.com/files/Intel_PXA270_Developers_Manual.pdf`.

[8] Intel Corporation. *Intel XScale Core Developer's Manual*, Jan. 2004. Available from: `http://pubs.gumstix.com/documents/PXADocumentation/XScale/XScaleCoreDevelopersManual.pdf`.

[9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[10] Name.net. Linux man pages [online]. Available from: `http://www.linuxmanpages.com/`.