Lab Work 2 - Bare-metal Programming

The things I have here are now available at GitHub under project my1barepi. I want to show that we do not necessarily need an OS - sometimes a 'simple' bare-metal code is good enough for simple applications.

The things I put here are mostly based on what I gather from the internet. Among the notable source of information are:

- http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/index.html
- http://www.valvers.com/embedded-linux/raspberry-pi/step01-bare-metal-programming-in-cpt1 (**DEAD LINK?**)
- https://github.com/dwelch67/raspberrypi

Compiling baremetal codes for Raspberry Pi

If you really want to start from the beginning, start from ACT I. But, if you are only interested in getting things running a.s.a.p., go straight ahead to ACT III, where we use example codes from *my1barepi*.

ACT I: Blinking LED

The Hello world! of embedded systems (which usually has no display by default) is possible with the availability of the ACT LED which is generally used to indicate (micro-)SD card access by an OS. Since this is bare-metal, all your base are belong to us! 😎

The GPIO that is used for the ACT LED on R-Pi B+ is GPIO47 (was on GPIO16 previously). To access the GPIO, some GPIO register information from the peripherals documentation introduced earlier:

The GPIO has 41 registers. All accesses are assumed to be 32-bit.

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W

Notice that the addresses shown are the ones assigned by the MMU. Since we will not be writing any codes to communicate with the MMU (at least for now), we need to access them using physical memory. So, instead of accessing 0x7E200000 for GPFSEL0, we will use address 0x20200000 (from the memory map introduced earlier).

A look into GPFSELx registers:

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL19	FSEL19 - Function Select 19 000 = GPIO Pin 19 is an input 001 = GPIO Pin 19 is an output 100 = GPIO Pin 19 takes alternate function 0 101 = GPIO Pin 19 takes alternate function 1 110 = GPIO Pin 19 takes alternate function 2 111 = GPIO Pin 19 takes alternate function 3 011 = GPIO Pin 19 takes alternate function 4 010 = GPIO Pin 19 takes alternate function 5	R/W	0
26-24	FSEL18	FSEL18 - Function Select 18	R/W	0
23-21	FSEL17	FSEL17 - Function Select 17	R/W	0
20-18	FSEL16	FSEL16 - Function Select 16	R/W	0
17-15	FSEL15	FSEL15 - Function Select 15	R/W	0
14-12	FSEL14	FSEL14 - Function Select 14	R/W	0
11-9	FSEL13	FSEL13 - Function Select 13	R/W	0
8-6	FSEL12	FSEL12 - Function Select 12	R/W	0
5-3	FSEL11	FSEL11 - Function Select 11	R/W	0
2-0	FSEL10	FSEL10 - Function Select 10	R/W	0

Table 6-3 – GPIO Alternate function select register 1

These should be enough for us to write a software to blink that LED!

Doing it in Assembly

Note: All codes/files in this section is available in my1barepi repository

A code to blink the ACT LED in assembly:

main.s

```
.section .boot
boot:
    ldr r0,=0x20200000
@set gpio as output
    mov r1,#1
    lsl r1,#21
    str r1,[r0,#16]
loop:
@clr gpio
    mov r1,#1
    lsl r1,#15
    str r1,[r0,#44]
@loop delay
    mov r2,#0x3F0000
wait1:
```

```
    sub r2,#1
    cmp r2,#0
    bne wait1
@set gpio
    mov r1,#1
    lsl r1,#15
    str r1,[r0,#32]
@loop delay
    mov r2,#0x3F0000
wait2:
    sub r2,#1
    cmp r2,#0
    bne wait2
@infinite loop
    b loop
```

We also need a linker:

kernel.ld

```
SECTIONS {
    . = 0x8000;
    .text : { /** code segment */
        *(.boot)
        *(.text)
    }
    .data : { /** data segment */
        *(.data)
    }
    /DISCARD/ : { /** discard all other... */
        *(*)
    }
}
```

A makefile to build bare-metal codes (assembly) for R-Pi:

Makefile

```
# to generate raspberry pi bare-metal code (kernel.img)

ifeq ($(OS),Windows_NT)
TOOLPATH ?= /c/users/public/tool/xtool-arm/bin/
else
TOOLPATH ?= /home/share/tool/xtool-arm/bin/
endif
TOOLPFIX ?= $(TOOLPATH)arm-none-eabi-

LINKER = kernel.ld
TARGET = kernel.img
```

```
LST = kernel.lst
MAP = kernel.map

AFLAGS +=
LFLAGS += --no-undefined

help:
    @echo "Targets: help clean pi <filename>.img"

pi: main.img

clean:
    rm -rf *.img *.lst *.map
# *.elf *.o deleted by compiler once done with

new: clean pi

%.img: %.elf
    $(TOOLPFIX)objcopy $< -O binary $@
    $(TOOLPFIX)objcopy $< -O binary $(TARGET)

%.elf: %.o $(LINKER)
    $(TOOLPFIX)ld $(LFLAGS) $< -Map $(MAP) -o $@ -T $(LINKER)
    $(TOOLPFIX)objdump -d $@ > $(LST)

%.o: %.s
    $(TOOLPFIX)as $(AFLAGS) $< -o $@
```

Using this makefile, all you have to do is type make and a kernel.img file will be created. Simply copy that file to the (micro-)SD card and you get yourself a system that blinks an LED! Cool, right?!

Doing it in C

Note: All codes/files in this section is available in my1barepi repository

The equivalent code to blink the ACT LED in C:

main.c

```
#define GPIO_BASE 0x20200000
#define GPIO_FSEL 0x00
#define GPIO_FSET 0x07
#define GPIO_FCLR 0x0A
#define GPIO_ACT_LED 47

/**  needs to be global, coz local needs stack => stack pointer! */
unsigned int *gpio, loop;
```

```
void main(void)
{
    /** point to gpio access register */
    gpio = (unsigned int*) GPIO_BASE;
    /** configure gpio as output */
    gpio[GPIO_FSEL+(GPIO_ACT_LED/10)] = 1 << (GPIO_ACT_LED%10)*3;
    /** main loop */
    while(1)
    {
        /** clear pin - on led! */
        gpio[GPIO_FCLR+(GPIO_ACT_LED/32)] = 1 << (GPIO_ACT_LED%32);
        /** delay a bit to allow us see the light! */
        for(loop=0;loop<0x3F0000;loop++);
        /** set pin - off led! */
        gpio[GPIO_FSET+(GPIO_ACT_LED/32)] = 1 << (GPIO_ACT_LED%32);
        /** delay a bit to allow us see the blink! */
        for(loop=0;loop<0x3F0000;loop++);
    }
}
```

A slightly modified linker file:

kernel.ld

```
SECTIONS {
    . = 0x8000;
    .text : { /** code segment */
        KEEP(*(.text.startup))
        *(.text)
    }
    .data : { /** data segment */
        *(COMMON)
        *(.data)
    }
    /DISCARD/ : { /** discard all other... */
        *(*)
    }
}
```

And, a slightly modified makefile:

Makefile

```
# to generate raspberry pi bare-metal code (kernel.img)

ifeq ($(OS),Windows_NT)
TOOLPATH ?= /c/users/public/tool/xtool-arm/bin/
else
TOOLPATH ?= /home/share/tool/xtool-arm/bin/
```

```
endif
TOOLPFIX ?= $(TOOLPATH)arm-none-eabi-

LINKER = kernel.ld
TARGET = kernel.img
LST = kernel.lst
MAP = kernel.map

CFLAGS += -mfpu=vfp -mfloat-abi=hard -march=armv6zk -mtune=arm1176jzf-s
CFLAGS += -nostdlib -nostartfiles -ffreestanding -Wall
LFLAGS += --no-undefined

help:
    @echo "Targets: help clean pi <filename>.img"

pi: main.img

clean:
    rm -rf *.img *.lst *.map
# *.elf *.o deleted by compiler once done with

new: clean pi

%.img: %.elf
    $(TOOLPFIX)objcopy $< -O binary $@
    $(TOOLPFIX)objcopy $< -O binary $(TARGET)

%.elf: %.o $(LINKER)
    $(TOOLPFIX)ld $(LFLAGS) $< -Map $(MAP) -o $@ -T $(LINKER)
    $(TOOLPFIX)objdump -d $@ > $(LST)

%.o: %.c
    $(TOOLPFIX)gcc $(CFLAGS) -c $< -o $@
```

And you get a bigger binary of the same thing (as discussed in lecture)! Also, you should notice that even though both programs (the main.s and main.c uses the same loop value (0x3f0000), the kernel.img created from main.c will result in a slower blinking LED.

Utilizing C Macro Definition

The C code can be rewritten to utilize C macro (function macro).

main.c

```
/*-------------------------------------------------------------------------
-------*/
#define GPIO_BASE 0x20200000
#define GPIO_FSEL 0x00
```

```
#define GPIO_FSET 0x07
#define GPIO_FCLR 0x0A
#define GPIO_ACT_LED 47
/*-------------------------------------------------------------------------
-------*/
/** using macros :p */
#define gpio_output(x) gpio[GPIO_FSEL+(x/10)]=1<<(x%10)*3
#define gpio_clr(x) gpio[GPIO_FCLR+(x/32)]=1<<(x%32)
#define gpio_set(x) gpio[GPIO_FSET+(x/32)]=1<<(x%32)
/*-------------------------------------------------------------------------
-------*/
/** volatile coz -O2 compiler option would otherwise kill them? */
volatile unsigned int *gpio, loop;
/*-------------------------------------------------------------------------
-------*/
void main(void)
{
    /** point to gpio access register */
    gpio = (unsigned int*) GPIO_BASE;
    /** configure gpio as output */
    gpio_output(GPIO_ACT_LED);
    /** main loop */
    while(1)
    {
        /** clear pin! */
        gpio_clr(GPIO_ACT_LED);
        /** delay a bit to allow us see the blink! */
        for(loop=0;loop<0x200000;loop++);
        /** set pin! */
        gpio_set(GPIO_ACT_LED);
        /** delay a bit to allow us see the blink! */
        for(loop=0;loop<0x200000;loop++);
    }
}
/*-------------------------------------------------------------------------
-------*/
```

ACT II: From Switch to LED

A system is rarely complete without an input. Let us now try to write a software that reads the status of a button (a.k.a. reset switch) and drive an LED accordingly.

Something About Input

Here are some information that is needed to read GPIO pin status.

Some relevant registers

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R
0x 7E20 003C	-	Reserved	-	-
0x 7E20 0040	GPEDS0	GPIO Pin Event Detect Status 0	32	R/W
0x 7E20 0044	GPEDS1	GPIO Pin Event Detect Status 1	32	R/W
0x 7E20 0048	-	Reserved	-	-
0x 7E20 004C	GPREN0	GPIO Pin Rising Edge Detect Enable 0	32	R/W
0x 7E20 0050	GPREN1	GPIO Pin Rising Edge Detect Enable 1	32	R/W
0x 7E20 0054	-	Reserved	-	-
0x 7E20 0058	GPFEN0	GPIO Pin Falling Edge Detect Enable 0	32	R/W
0x 7E20 005C	GPFEN1	GPIO Pin Falling Edge Detect Enable 1	32	R/W

So, to actually check the pin status we need to check the following.

GPIO Pin Level Registers (GPLEVn)

SYNOPSIS The pin level registers return the actual value of the pin. The LEV{n} field gives the value of the respective GPIO pin.

Bit(s)	Field Name	Description	Type	Reset
31-0	LEVn (n=0..31)	0 = GPIO pin *n* is low 0 = GPIO pin *n* is high	R/W	0

Table 6-12 – GPIO Level Register 0

This is a simple program that detects an input level and updates an LED accordingly.

main.c

```
/*------------------------------------------------------------------------
-------*/
#define GPIO_BASE 0x20200000
#define GPIO_FSEL 0x00
#define GPIO_FSET 0x07
#define GPIO_FCLR 0x0A
#define GPIO_FGET 0x0D
//*-----------------------------------------------------------------------
--------*/
#define MY_LED 47
#define MY_SWITCH 3
/*------------------------------------------------------------------------
```

```
-------*/
/** using macros :p */
#define gpio_output(x) gpio[GPIO_FSEL+(x/10)]=1<<(x%10)*3
#define gpio_clr(x) gpio[GPIO_FCLR+(x/32)]=1<<(x%32)
#define gpio_set(x) gpio[GPIO_FSET+(x/32)]=1<<(x%32)
#define gpio_get(x) (gpio[GPIO_FGET+(x/32)]&(1<<(x%32)))
/*-------------------------------------------------------------------------
-------*/
volatile unsigned int *gpio;
/*-------------------------------------------------------------------------
-------*/
void main(void)
{
    /** base register address */
    gpio = (unsigned int*) GPIO_BASE;
    /** configure gpio as output, by default it is an input pin */
    gpio_output(MY_LED);
    /** main loop */
    while(1)
    {
        if(gpio_get(MY_SWITCH)) gpio_set(MY_LED);
        else gpio_clr(MY_LED);
    }
}
/*-------------------------------------------------------------------------
-------*/
```

Something About Input (Events)

Notice that we can also detect events (and edges)!

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0060	-	Reserved	-	-
0x 7E20 0064	GPHEN0	GPIO Pin High Detect Enable 0	32	R/W
0x 7E20 0068	GPHEN1	GPIO Pin High Detect Enable 1	32	R/W
0x 7E20 006C	-	Reserved	-	-
0x 7E20 0070	GPLEN0	GPIO Pin Low Detect Enable 0	32	R/W
0x 7E20 0074	GPLEN1	GPIO Pin Low Detect Enable 1	32	R/W
0x 7E20 0078	-	Reserved	-	-
0x 7E20 007C	GPAREN0	GPIO Pin Async. Rising Edge Detect 0	32	R/W
0x 7E20 0080	GPAREN1	GPIO Pin Async. Rising Edge Detect 1	32	R/W
0x 7E20 0084	-	Reserved	-	-
0x 7E20 0088	GPAFEN0	GPIO Pin Async. Falling Edge Detect 0	32	R/W
0x 7E20 008C	GPAFEN1	GPIO Pin Async. Falling Edge Detect 1	32	R/W
0x 7E20 0090	-	Reserved	-	-
0x 7E20 0094	GPPUD	GPIO Pin Pull-up/down Enable	32	R/W
0x 7E20 0098	GPPUDCLK0	GPIO Pin Pull-up/down Enable Clock 0	32	R/W
0x 7E20 009C	GPPUDCLK1	GPIO Pin Pull-up/down Enable Clock 1	32	R/W
0x 7E20 00A0	-	Reserved	-	-
0x 7E20 00B0	-	Test	4	R/W

Info on event detect status registers and edge detect enable registers

GPIO Event Detect Status Registers (GPEDSn)

SYNOPSIS The event detect status registers are used to record level and edge events on the GPIO pins. The relevant bit in the event detect status registers is set whenever: 1) an edge is detected that matches the type of edge programmed in the rising/falling edge detect enable registers, or 2) a level is detected that matches the type of level programmed in the high/low level detect enable registers. The bit is cleared by writing a "1" to the relevant bit.

The interrupt controller can be programmed to interrupt the processor when any of the status bits are set. The GPIO peripheral has three dedicated interrupt lines. Each GPIO bank can generate an independent interrupt. The third line generates a single interrupt whenever any bit is set.

Bit(s)	Field Name	Description	Type	Reset
31-0	EDSn (n=0..31)	0 = Event not detected on GPIO pin n 1 = Event detected on GPIO pin n	R/W	0

Table 6-14 – GPIO Event Detect Status Register 0

GPIO Asynchronous rising Edge Detect Enable Registers (GPARENn)

SYNOPSIS The asynchronous rising edge detect enable registers define the pins for which a asynchronous rising edge transition sets a bit in the event detect status registers (GPEDSn).

Asynchronous means the incoming signal is not sampled by the system clock. As such rising edges of very short duration can be detected.

Bit(s)	Field Name	Description	Type	Reset
31-0	ARENn (n=0..31)	0 = Asynchronous rising edge detect disabled on GPIO pin *n*. 1 = Asynchronous rising edge on GPIO pin *n* sets corresponding bit in EDSn.	R/W	0

Table 6-24 – GPIO Asynchronous rising Edge Detect Status Register 0

The event(s) status would be very useful, but we will pin this up and revisit the topic later.

ACT III : Using my1barepi

Using my1barepi makes things easier.

Starting with the basics...

- T00 Intro
- T01 GPIO - Manage the I/O pins
- T02 Timer - Manage time
- T03 Interrupt - Manage events

Revisiting Basic Digital Interfacing

Unlike boolean logic, digital electronics has a third-state condition (aptly named tri-state condition) which cannot be fit into the common `VDD = logic 1` and GND = `logic 0` presumptions. A more suitable definition for logic 1 in digital electronics is node charging/discharging capabilities: logic 1 is the ability to charge a node to voltage VDD (actually over a certain threshold, but that is for another discussion), and logic 0 is the ability to discharge a node to GROUND reference voltage.

Logic 1 - Node charged to VDD	Logic 0 - Node discharged to GND
Also known as sourcing current	Also known as sinking current

Using this, the tri-state condition is when a node is NOT being charged or discharged (i.e. floating, left for other device to pull the node to VDD or GND). This is very useful since there are times when a tri-state condition is required (e.g. I2C interface, where open collector/drain circuit is commonly used).

Things To Tinker

Thing 1 Using the given example code, try to blink an external LED (off the Pi, on a breadboard). You will have to select a GPIO pin for that. Then, try to do 2 alternate-blinking LEDs. Investigate the issue when the two selected GPIO number have the same digit on the 10s (e.g. 11 and 16, 24 and 22).

Thing 2 Based on the GPIO input example, try to write a code that blinks an external LED ONLY when the input is at logic 'LO'. Otherwise, the LED should be turned OFF.

Thing 3 Use a seven segment. Create a simple up (OR down) counter.

Thing 4 Upgrade the previous Thingy - use reset switches to start/stop the counting and control the direction.