Bare Metal Programming

IIT Bombay

By Ravi Kushawaha, Saqib Azim and Sharan Kumar Pujari

Any mistake in the following? Help us improve.

References:

University of Cambridge: https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/introduction.html

Peripheral Manual: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf

What is bare metal programming?

Some geeky people wouldn't like to use an OS to program their device and hence they prefer using so called 'bare metal Programming'. This is done using a language called assembly language. An appendix is attached regarding the language's commands. You might need it when you start writing code.

Getting Ready

We are going to program on **Raspberry Pi-1 Model-B**. This model has **Broadcom 2835 (BCM2835)** chip safely lying below Samsung's memory chip. Knowing the above information is very important since whatever we are going to do later might not work on other versions on R-Pi. You should have a proper working Raspbian OS installed on your Pi. If you don't know how to get to it, Google might save you. Also note that we are using a **Linux system** to work on. Now download the GNU tool-chain. Some Linux distributions including Ubuntu offer the ARM GNU Tool chain via apt-get. Run the following command:

sudo apt-get install gcc-arm-none-eabi

For other OSs visit the Downloads page of the reference link given in at the last.

Also you need to download the OS Template for Linux given on the above said page. Extract the contents of the OS template into a folder with name of your choice (like 'build'). I placed this folder on Desktop. It should look something like this:

build (folder) ->

1. Source (folder) ->

   (Empty)

2. Kernel.ld (file)

3. LICENSE (file)

4. Makefile (file)

Whatever code we write in **.s** file will be saved in **Source** folder. Don't mess up with any other content.

Creating the .img file

After writing the code in **.s** files, open Linux terminal and change directory to that folder where you saved the **build** folder (Desktop in my case). Then type **make** and press enter this will create three files in **build** folder viz., kernel.img, kernel.list and kernel.map. We mostly deal with kernel.img file. Now in the SD card containing the image of Raspbian, you will find a boot folder in which you will find a file with name kernel.img. This is around 4 Mb. Save this file with another name like kernel.imgRaspbian in the same folder. Now copy and paste the kernel.img file you have created using make command in the same folder without changing name. After copying the file, reload the SD card back in Pi. Pi will read the file with name kernel.img while booting up. To switch back to the previous state, just delete your kernel.img file and rename the original file back to kernel.img.

Lesson 1

In this lesson we will learn how to light up ACT LED or OK LED. You might want to locate it on your Pi board. Create **main.s** file in Source folder. Open it and let's start writing the code!

Copy and paste the following code.

```
.section .init
.globl _start
_start:
```

The above set of instructions are for assembler. In assembly code, each line is a code. Spaces before and after code lines to aid readability. In first line of code, **.section** puts all the code following it to a place where we specify; in this case we asked it to put in .**init**. **.section** puts all the code following it until next **.section** comes up or code ends. Putting code in **.init** section will tell assembler which code to run first as we will see later, we will be using .section multiple times.

The next two lines are there to stop a warning message from the assembler. They aren't all that important.

Copy the following code below the above three lines of code:

```
ldr r0,=0x20200000
```

This command line asks the processor to store the hexadecimal number 20200000 into register r0. We write **0x** before a hexadecimal number to inform the processor that the number is in hexadecimal. Similarly 0b before a binary number.

There are 13 general purpose registers (r0 - r12) which you can use for calculations. All these are of 32 bit. So a register can store numbers as large as 4,29,49,67,295.

ldr is command and stands for load register. **ldr x,=y** commands the processor to store the number **y** in the register **x.**

The huge number you see above is the base address of GPIO (acronym for General Purpose Input Output) which controls the status of GPIO pins. There are 54 GPIO peripherals in total out of which 26 are physically accessible. These 26 pins can be seen emerging out of the board. Now to control all 54 GPIO peripheral, the processor has 24 bytes reserved for the same. Every 4 bytes has control over 10 GPIO pins. This means GPIO 0 to GPIO 9 are in control of first 4 bytes. GPIO 9 to GPIO 19 are in control of next 4 bytes and so on. This makes 6 sets of 4 bytes with last set having control of only 4 GPIO pins as there are only 54 GPIO pins. Now every 3 bits of these bytes has control of 1 GPIO pin. Consider 1st set of 4 bytes that controls GPIO 0 to GPIO 9. 4 bytes means 32 bits. Now first 3 bits (0th bit, 1st bit and 2nd bit) controls GPIO 0. Next 3 bits controls GPIO 1 and so on. As you see, last 2 bits are of no use. They are left unused. These 3 bits of each pin can represent 8 possible numbers and hence 8 possible functions. 001 means **Output** (refer datasheet if curious).

NOTE: Addresses given in peripheral datasheet of BCM 2835 are in the form 0x7Ennnnnn, where n is an integer. You have to take care to replace 7E in the addresses by 20.

You must know that the ACT/OK LED is connected to 16th GPIO pin. So this falls in 2nd set of 4 bytes. Of these bits 20th, 19th and18th bits represent 16th pin. We have to set these to 001.

Copy the following code:

```
mov r1,#1

lsl r1,#18

str r1,[r0,#4]
```

mov x,y means move or store the number **y** in the register **x.** We should put 1 in 18th place of the 32 bit number (all other bits are zero by default). So left shifting this 1 from 0th place to 18th place would do the job. **lsl x,y** means left shift the number in register **x** by **y** places. **lsl** stands for logical shift left. **str x,y** stores the number in register x in register y. **str x,[y]** means store the number in register **x** in the place whose address is given by the number in stored in **y**. As you see, #4 is written in 3rd line. This is because, we want the number to be stored in 2nd set of 4 bytes. So this is an offset given to address. Suppose we

want to use GPIO 22, then instead of #4, write #8.

We just set it to output. Next we want to pull the pin high. There are **SET** registers and **CLEAR** Registers that pull up or pull down a GPIO pin.

Copy the following code:

```
mov r1,#1

lsl r1,#16

str r1,[r0,#28]
```

We store 1 at 16th place in r1 as we want to pull up 16th GPIO pin. Set registers address is after 28 bytes from base address 0x20200000. This information is sent to that address using last command line in the above code.

Lastly copy the code:

```
loop$:

b loop$
```

After LED is lit up, what is the next job to the processor? To keep it engaged or in other words, to prevent a crash, the above code is used. **random** is the name of a label. Name of a label ends with **$** symbol. Whenever a label is used (e.g., **random$:**), all the code following it (until a branch comes) will come under this label. Later in the code when the name of this label is used, the code following that label is executed. **b** is used to indicate a branch. This makes the processor struck in the loop created by the code. Make sure to put an empty line at the end of any assembly code to avoid warning messages from the GNU tool chain.

Lesson 2

It's time to blink the LED. 0x20003000 is the base address of System Timer. There's 64 bit Timer Register at 4 Bytes of offset from this base address. It increments every 1 microsecond. Since it's of 64 bit we need two registers to keep a track of time. Note that these registers should be consecutive. And also the first register should be even one. For e.g., r10, r11 would do but not r5, r6. In a new **main.s**, Copy the following code:

```
.section .init

.globl _start

_start:

gpio .req r0
```

```
timer .req r1

delay .req r2

ldr gpio,=0x20200000

ldr timer,=0x20003000

ldr delay,=500000
```

The 4th line of code assigns register r0 the name 'gpio'. Same with 5th and 6th lines. The number stored in *delay* register is 0.5 sec. Copy the following code:

```
mov r3,#1

lsl r3,#18

str r3,[gpio,#4]

blink$:

mov r3,#1

lsl r3,#16

str r3,[gpio,#28]
```

This is something you are familiar with. Let's go ahead for something new. We will use difference between two time points to count the time. Copy the following code:

```
ldrd r4,r5,[timer,#4]

waitloop1$:

ldrd r6,r7,[timer,#4]

sub r8,r6,r4

cmp r8,r2

bls waitloop1$
```

Suffixing **d** to **ldr** command lets us to deal with 2 words i.e., 64bits. Similarly **h** for half-word (16 bits) and **b** for a byte. Firstly time **t1** is loaded into r4 and r5 and then inside the **waitloop** we again sample time **t2** and load it in r6 and r7. We will use only lower 32 bits to deal with time. Subtracting t1 from t2 would give us a value in microsecond taking **t1** as zeroth microsecond (or a reference). This loop helps us to compare the increment in time with the desired delay in microseconds. **sub r8,r6,r4** saves the value **r6** minus **r4** in **r8**. **cmp r8,r2** compares the values in r8 and r2 and the output result is stored somewhere. This result can be further used by other commands by adding a suffix. **ls** is a suffix added to branch **b**. Now the branch executes only if the previous comparison is less than or equal to. Copy the following

code:

```
mov r3,#1

lsl r3,#16

str r3,[gpio,#40]

ldrd r4,r5,[timer,#4]

waitloop2$:

ldrd r6,r7,[timer,#4]

sub r8,r6,r4

cmp r8,delay

bls waitloop2$

b blink$
```

This time we want to switch OFF the LED. Hence **Clear** register is used which at 40 Bytes offset from the base address. We want to keep the LED blinking; so put the code in a loop.

Something New That You Should Know

Look at the example code below:

```
.section .data

.align 4

myData:

.int 242526

.int 0b131517
```

As explained earlier, **.section** command lets us categorize our whole code. Writing **.data** after this command will store the information following it in the memory. **.align x** (**x** being always multiple of 2) will arrange the data in groups of 2^x bytes. In the above example, data is stored in groups of 16 bytes. **.int y** will store the integer **y** as a data. To all your data, you must give a name such as **myData**.

Lesson 3

It's time to search a screen for your Pi. Yeah, it's time to know about the GPU (Graphical Processing Unit) now. We must tell the GPU about all the characteristics of the display we want to use viz., resolution, pixel bit-depth, etc. This information that we want to convey to the GPU will be stored in memory separately. Code that helps us do this will also be in separate **.s** file. Make a new file and name it **FrameBufferInfo.s**.

Copy the following code into it:

```
.section .data
.align 4
.globl FrameBufferInfo
FrameBufferInfo:
.int 1024 /* #0 Physical Width */
.int 768 /* #4 Physical Height */
.int 1024 /* #8 Virtual Width */
.int 768 /* #12 Virtual Height */
.int 0 /* #16 GPU - Pitch */
.int 16 /* #20 Bit Depth */
.int 0 /* #24 X */
.int 0 /* #28 Y */
.int 0 /* #32 GPU - Pointer */
.int 0 /* #36 GPU - Size */
```

Copy the following code in **main.s:**

```
.section .init

.globl _start

_start:

b main

.section .text

main:

mov sp,#0x8000
```

The last three lines are to include a stack pointer that helps us push information into it and pull out of it as and when required. After this we have to write a code to send the above frame data to the GPU (mailbox writing).

```
mov r0,#1024
mov r1,#768
mov r2,#16
ldr r3,=FrameBufferInfo
str r0,[r3,#0]
str r1,[r3,#4]
str r0,[r3,#8]
str r1,[r3,#12]
str r2,[r3,#20]
/*mailbox write*/
mov r0,r3
add r0,#0x40000000
mov r1,#1
tst r0,#0b1111
bne error$
```

```
ldr r2,=0x2000B880
wait1$:
ldr r5,[r2,#0x18]
tst r5,#0x80000000
bne wait1$
add r0,r1
str r0,[r2,#0x20]
```

Observe addition of 0x40000000 to the address in r0. If we just send the address, the GPU will write its response, but will not make sure we can see it by flushing its cache. The cache is a piece of memory where a processor stores values its working on before sending them to the RAM. By adding 0x40000000, we tell the GPU not to use its cache for these writes, which ensures we will be able to see the change. The whole 32 bit information is stored in two pieces. First 28 bits is in r0 and last 4 bits is stored in r1. Later we will add these two. Therefore is important to check that the last 4 binary digits of value in r0 should be zero. The 4 bit information if is 1, instructs to mail write. 0 means mail read. To test it we use **tst x,y** which ANDs the value **y** with the value present in the register **x** and compares the results with zero. This comparison result is stored. If it's not zero then it goes to that part of code where I have defined error$ at the end. There is something called status field after 18 bytes offset of the base address of the GPU. Unless the 31st bit (top most bit) here becomes zero we shouldn't proceed. This is the reason why the wait loop is used. After all this, we add and send it to 20 bytes offset from base.
Time to mail read. Copy the following code after the previous code:

```
/*mailbox read*/
mov r1,#1
rightmail$:
wait2$:
ldr r5,[r2,#0x18]
tst r5,#0x20000000
bne wait2$
ldr r0,[r2,#0]
and r5,r0,#0b1111
teq r5,r1
bne rightmail$
and r2,r0,#0xfffffff0
```

There is status field to mail read also. Observe in the above code. To extract information, 8th line of the above code is used and is stored in r0. r0 is ANDed with 15 to extract the last 4 bits. This information is compared with what we sent as the channel number in r1. First 28 bits are extracted to r2.
A symbol on the screen is drawn by exciting appropriate set of pixels. A pixel is lit up by filling colour to it. [gpu pointer+ (x + y * width) * pixel size] will lit up (x,y) pixel. Note that origin is top left corner of the screen. Copy the following code:

```
mov r0,[r3,#32]  /*gpu pointer*/
y .req r11
x .req r12
colour .req r9
ldr colour,=#0b11111000000111111
mov x,100
mov y,200
mov r10,y
lsl r10,#10   /* multiply by 1024 is left-shifting by 10 places*/
```

```
add r10,x
lsl r10,#1    /* multiply by 2 i.e., 2 bytes pixel size(16bitdepth)*/
add r0,r10
str colour,[r0]
```

Also note that colour is 16 bit information as we have chosen bit depth as 16. Most significant 5 bits make the red colour, the next 6 the green colour and the last 5 the blue colour. This code lits up a pixel at (100,200).
Copy the following code:

```
error$:
/* lED on code*/
ldr r0,=0x20200000
mov r1,#1
lsl r1,#18
str r1,[r0,#4]
mov r1,#1
lsl r1,#16
str r1,[r0,#40]
loop$:
b loop$
```

You are encouraged to draw straight lines or any other random patters which use appropriate algorithms such as Bresenham's algorithm for drawing lines. You could also use font.bin file which helps draw alphanumeric characters.

Appendix

The following is a list of all the instruction boxes in the courses in order.

`ldr reg,=val` puts the number **`val`** into the register named **`reg`**.
`mov reg,#val` puts the number **`val`** into the register named **`reg`**.
`lsl reg,#val` shifts the binary representation of the number in **`reg`** by **`val`** places to the left.
`str reg,[dest,#val]` stores the number in **`reg`** at the address given by **`dest`** + **`val`**.
`name:` labels the next line **`name`**.
`b label` causes the next line to be executed to be **`label`**.
`sub reg,#val` subtracts the number **`val`** from the value in **`reg`**.
`cmp reg,#val` compares the value in **`reg`** with the number **`val`**.
Suffix **`ne`** causes the command to be executed only if the last comparison determined that the numbers were not equal.
`.globl lbl` makes the label **`lbl`** accessible from other files.
`mov reg1,reg2` copies the value in **`reg2`** into **`reg1`**.
Suffix **`ls`** causes the command to be executed only if the last comparison determined that the first number was less than or the same as the second. Unsigned.
Suffix **`hi`** causes the command to be executed only if the last comparison determined that the first number was higher than the second. Unsigned.
`push {reg1,reg2,...}` copies the registers in the list **`reg1,reg2,...`** onto the top of the stack. Only general purpose registers and lr can be pushed.
`bl lbl` sets **`lr`** to the address of the next instruction and then branches to the label **`lbl`**.
`add reg,#val` adds the number **`val`** to the contents of the register **`reg`**.
Argument shift **`reg,lsl #val`** shifts the binary representation of the number in **`reg`** left by **`val`** before using it in the operation before.
`lsl reg,amt` shifts the binary representation of the number in **`reg`** left by the number in **`amt`**.
`str reg,[dst]` is the same as **`str reg,[dst,#0]`**.
`pop {reg1,reg2,...}` copies the values from the top of the stack into the register list **`reg1,reg2,...`**. Only general purpose registers and pc can be popped.
`alias .req reg` sets **`alias`** to mean the register **`reg`**.
`.unreq alias` removes the alias **`alias`**.
`lsr dst,src,#val` shifts the binary representation of the number in **`src`** right by **`val`**, but stores the result in **`dst`**.
`and reg,#val` computes the Boolean and function of the number in **`reg`** with **`val`**.
`teq reg,#val` checks if the number in **`reg`** is equal to **`val`**.
`ldrd regLow,regHigh,[src,#val]` loads 8 bytes from the address given by the number in **`src`** plus **`val`** into **`regLow`** and **`regHigh`**.
`.align num` ensures the address of the next line is a multiple of 2^{num}.
`.int val` outputs the number **`val`**.
`tst reg,#val` computes **`and reg,#val`** and compares the result with 0.
`mla dst,reg1,reg2,reg3` multiplies the values from **`reg1`** and **`reg2`**, adds the value from **`reg3`** and places the least significant 32 bits of the result in **`dst`**.
`strh reg,[dest]` stores the low half word number in **`reg`** at the address given by **`dest`**.

Code for drawing line:

Note: The following code is to be put in main.s. Make sure you include framebufferinfo file also. You might want to change initial and final points.

```
.section .init
.globl _start
_start:
b main
.section .text
main:
mov sp,#0x8000
mov r0,#1024
mov r1,#768
mov r2,#16
ldr r3,=FrameBufferInfo
str r0,[r3,#0]
str r1,[r3,#4]
str r0,[r3,#8]
str r1,[r3,#12]
str r2,[r3,#20]
/*mailbox write*/
mov r0,r3
add r0,#0x40000000
mov r1,#1
tst r0,#0b1111
bne out$
ldr r2,=0x2000B880
wait1$:
ldr r5,[r2,#0x18]
tst r5,#0x80000000
bne wait1$
add r0,r1
str r0,[r2,#0x20]
/*mailbox read*/
mov r1,#1
rightmail$:
wait2$:
ldr r5,[r2,#0x18]
tst r5,#0x20000000
bne wait2$
ldr r0,[r2,#0]
and r5,r0,#0b1111
teq r5,r1
bne rightmail$
and r2,r0,#0xfffffff0
/*line drawing algo*/
x0 .req r0
x1 .req r1
y0 .req r2
y1 .req r4
sx .req r7
sy .req r8
```

```
dx .req r9
dyn .req r10
err .req r11
colour .req r12
mov x0,#25            /*put your favourite initial x coordinate */
mov y0,#50
mov x1,#500
mov y1,#600           /*put your favourite final y coordinate */
ldr colour,=#0b11111111111
cmp x0,x1
subgt dx,x0,x1
movgt sx,#-1
suble dx,x1,x0
movle sx,#1
cmp y0,y1
subgt dyn,y1,y0
movgt sy,#-1
suble dyn,y0,y1
movle sy,#1
add err,dx,dyn
add x1,sx
add y1,sy
pixelLoop$:
teq x0,x1
teqne y0,y1
beq out$
ldr r5,[r3,#32]
mov r6,y0
lsl r6,#10
add r6,x0
lsl r6,#2
add r5,r6
strh colour,[r5]
sub r5,r6
cmp dyn, err,lsl #1
addle err,dyn
addle x0,sx
cmp dx, err,lsl #1
addge err,dx
addge y0,sy
b pixelLoop$
out$:
/* lED on code*/
ldr r0,=0x20200000
mov r1,#1
lsl r1,#18
str r1,[r0,#4]
mov r1,#1
lsl r1,#16
str r1,[r0,#40]
loop$:
b loop$
```

Code for writing the symbol A:

Note: The following code is to be put in main.s. Make sure you include framebufferinfo file and a font.bin file also. Link to font.bin file:
https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads/font0.bin

```
.section .init
.globl _start
_start:
b main
.section .text
main:
mov sp,#0x8000
mov r0,#1024
mov r1,#768
mov r2,#16
ldr r3,=frameBufferInfo
str r0,[r3,#0]
str r1,[r3,#4]
str r0,[r3,#8]
str r1,[r3,#12]
str r2,[r3,#20]
/*mailbox write*/
mov r0,r3
add r0,#0x40000000
mov r1,#1
tst r0,#0b1111
bne out$
ldr r2,=0x2000B880
wait1$:
ldr r5,[r2,#0x18]
tst r5,#0x80000000
bne wait1$
add r0,r1
str r0,[r2,#0x20]
/*mailbox read*/
mov r1,#1
rightmail$:
wait2$:
ldr r5,[r2,#0x18]
tst r5,#0x20000000
bne wait2$
ldr r0,[r2,#0]
and r5,r0,#0b1111
teq r5,r1
bne rightmail$
and r2,r0,#0xfffffff0
/*DrawCharacter*/
gpuPointer .req r0
ldr r0,[r3,#32]
fontAddr .req r1
ldr r1,=font
```

```
add r1,#1040 /* 65*16=1040 has 'A' */
x0 .req r2
y0 .req r3
x .req r4
y .req r5
colour .req r6
mov x0,#200
mov y0,#200
mov x,#200
mov y,#200
ldr colour,=#0b1111111111100000
mov r9,#16

dc$:
teq r9,#0
subne r9,#1
beq engage1$
ldrb r7,[r1]
add r1,#1
teq r7,#0
addeq y,#1
moveq x,x0
beq dc$

and r8,r7,#0b1
teq r8,#1
moveq r10,y
lsleq r10,#10
addeq r10,x
lsleq r10,#1
addeq r0,r10
streqh colour,[r0]
subeq r0,r10
add x,#1

and r8,r7,#0b10
teq r8,#2
moveq r10,y
lsleq r10,#10
addeq r10,x
lsleq r10,#1
addeq r0,r10
streqh colour,[r0]
subeq r0,r10
add x,#1

and r8,r7,#0b100
teq r8,#4
moveq r10,y
lsleq r10,#10
addeq r10,x
lsleq r10,#1
```

```
addeq r0,r10
streqh colour,[r0]
subeq r0,r10
add x,#1

and r8,r7,#0b1000
teq r8,#8
moveq r10,y
lsleq r10,#10
addeq r10,x
lsleq r10,#1
addeq r0,r10
streqh colour,[r0]
subeq r0,r10
add x,#1

and r8,r7,#0b10000
teq r8,#16
moveq r10,y
lsleq r10,#10
addeq r10,x
lsleq r10,#1
addeq r0,r10
streqh colour,[r0]
subeq r0,r10
add x,#1

and r8,r7,#0b100000
teq r8,#32
moveq r10,y
lsleq r10,#10
addeq r10,x
lsleq r10,#1
addeq r0,r10
streqh colour,[r0]
subeq r0,r10
add x,#1

and r8,r7,#0b1000000
teq r8,#64
moveq r10,y
lsleq r10,#10
addeq r10,x
lsleq r10,#1
addeq r0,r10
streqh colour,[r0]
subeq r0,r10
add x,#1

and r8,r7,#0b10000000
teq r8,#128
moveq r10,y
```

```
lsleq r10,#10
addeq r10,x
lsleq r10,#1
addeq r0,r10
streqh colour,[r0]
subeq r0,r10
add x,#1



mov x,x0
add y,#1
b dc$

engage1$:
b engage1$

/* lED on code*/
out$:
ldr r0,=0x20200000
mov r1,#1
lsl r1,#18
str r1,[r0,#4]
mov r1,#1
lsl r1,#16
str r1,[r0,#40]
engage2$:
b engage2$
```