Experiment: FM Transmitter using Raspberry pi

Objectives:

1. Studying the principles of FM modulation
2. Studying the principles of PLL

Lab 1: Get familiar with RPi GPIO and program them using *direct register access*

Raspberry pi (RPi) is a tiny computer and affordable computer that can be used to learn programming through practical projects. RPi offers 40 GPIO, these pins have dual numbering, one for the pin number (referred as pin #) in the header and second the actual GPIO pin (referred as GPIO #) of the Broadcom chip used in pi, refer fig .1. For example, pin #40 is addressed as pin 40 in the header but it is GPIO21 from the Broadcom chip. Each GPIO may provide multiple functionality like pin #8 and #10 are also used for UART. GPIOs are one of the peripheral features provided by the Broadcom chip (referred as BCM chip). We will refer the BCM2835 ARM peripherals document[1].

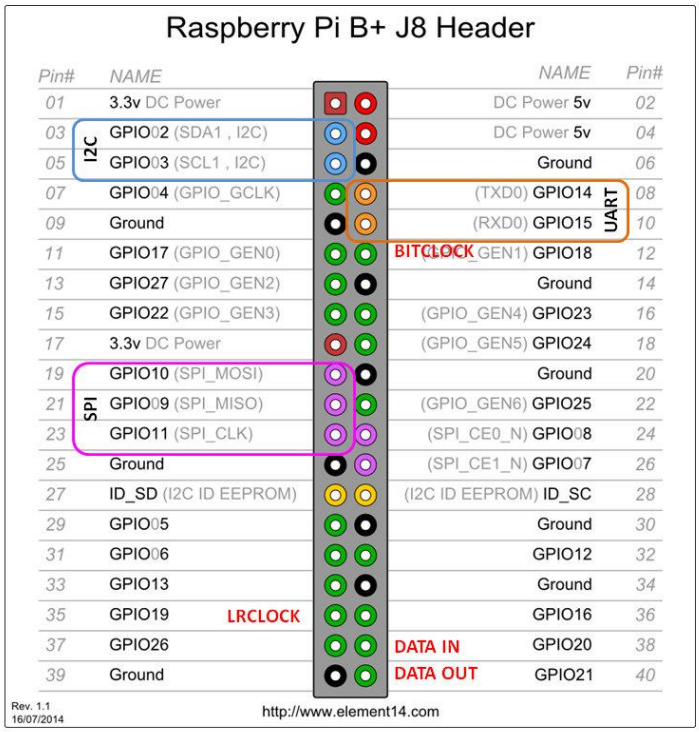


Fig 1: RPi GPIO pins

RPi 2 contains the Broadcom chip BCM2836, it is very much like BCM2835 (RPi zero), so we can use the older document, with the additional changes mentioned in ARM revision files[2] provided with the document.

If you have done any low-level programming of microcontrollers, you will know that everything is done by writing to the registers in the memory of the device. In this lab, we will try to learn how to access the registers of RPi to program the GPIO functionality.

Suggested reading here: -> chapter 1 and 6 from *BCM2835 ARM peripherals*

-> system overview from the *ARM QA7_Rev* document

-> https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md

Note: BCM2835(specific implementation) is also referred as BCM2708 (It is the family number)

If you have read through the above, by this time you will know that ARM Linux provides multiple addresses (virtual, mapped physical, mapped bus, etc.), we are going to use the bus addresses (refer section 1.2.3 - 1.2.4 and fig in page 5) as these are the address for peripherals. Chapter 6 from BCM2835 list out all the registers and functionality of GPIO, we will be programming these registers. Like GPIO, other peripherals like UART, DMA, etc. are also available, but in this experiment, we will restrict ourselves to GPIO.

Let's take an exercise and understand the direct register access to these registers.

Ex: LED Blink (the classic starting problem for any microcontroller programming) using direct register access programming in PI.

Physical address start for BCM2836 (Pi 2): *0x3F000000* (*0x20000000 for Pi zero*)

Peripheral bus address start: *0x7E000000*

Therefore, a peripheral available at address *0x7Exxxxxx* has physical address of *0x3Fxxxxxx (0x20xxxxxx in pi 0).*

From chapter 6 BCM2835 we can see the GPIO registers starts form *0x7E200000* so the GPIO offset is *0x00200000* from the peripheral base addr(*0x3F000000*) [GPIO peripheral offset = peripheral base + GPIO offset].

Therefore, first we need to access the memory and then do the mapping. Luckily, we have C functions to do these. ***mmap(2) and open(3)***

```
int open(const char *pathname, <--- open the file - "/dev/mem" in our case)
                int flags);     <--- flags (read/write, etc.)

void *mmap (
                void *addr,     <--- address where to place mapping
                size_t length,  <--- mapping length
                int prot,       <--- protocol (read/write/both)
                int flags,      <--- flags (shared/private)
                int fd,         <--- file to map(return of above function)
                off_t offset    <--- offset to GPIO peripherals
        );
```

So to setup the Pi for direct register access (through memory mapping) we call a function **`Setup()`**

```
void setup(void) {

      //open the memory as file

      //do the register mapping to the file

      //assign a pointer to the memory (Why? What will be its type qualifier?)

}
```

Why? – we are mapping the registers to the memory, the safest and best way to access any memory content is using pointers. Please go through pointers in C, its use and application.

What will be its type qualifier? – I want you to understand this by trying. A Good explanation about what is the solution, is available in

https://barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword

Now since we are done setting up the GPIO, lets access the pins to do a LED blinking. Let say I want to connect the LED to the GPIO 7 (pin #26).

Pi GPIO provide multiple functionality, so first select functionality as i/o. refer GPFSELn (0-5) registers, it allows you to choose different functionality for each GPIO pin.

29-27	FSEL9	FSEL9 - Function Select 9 000 = GPIO Pin 9 is an input 001 = GPIO Pin 9 is an output 100 = GPIO Pin 9 takes alternate function 0 101 = GPIO Pin 9 takes alternate function 1 110 = GPIO Pin 9 takes alternate function 2 111 = GPIO Pin 9 takes alternate function 3 011 = GPIO Pin 9 takes alternate function 4 010 = GPIO Pin 9 takes alternate function 5	R/W	0
26-24	FSEL8	FSEL8 - Function Select 8	R/W	0
23-21	FSEL7	FSEL7 - Function Select 7	R/W	0
20-18	FSEL6	FSEL6 - Function Select 6	R/W	0
17-15	FSEL5	FSEL5 - Function Select 5	R/W	0
14-12	FSEL4	FSEL4 - Function Select 4	R/W	0
11-9	FSEL3	FSEL3 - Function Select 3	R/W	0
8-6	FSEL2	FSEL2 - Function Select 2	R/W	0
5-3	FSEL1	FSEL1 - Function Select 1	R/W	0
2-0	FSEL0	FSEL0 - Function Select 0	R/W	0

Table 6-2 – GPIO Alternate function select register 0

Here is a snippet, in this GPFSEL0 allows to select functionality for GPIO pin 1 through 9. Refer chapter 6 in BCM2835.

To select GPIO 7 as output we set FSEL7 with output (001).

Generally, all the pins remain in floating state, so **it is advisable to first make the pin as input (by pulling down) and then make them output.**

Now, to set/clear the pin we use the GPSETn/GPCLRn register.

To summarize,

Setup GPIO for register access

Make GPIO 7 output by setting appropriate bit in GPFSEL register (make it first i/p and then o/p)

Set / clear (i.e. send high or low to the pin) the GPIO using GPSETn/GPCLRn to toggle LED

(pi GPIO 7) ------> resistor(1 K ohm) ------> LED

Can you now write a program to blink LED by accessing the GPIO registers in Pi?

Some source for reading:

https://www.raspberrypi.org/

http://elinux.org/RPi_GPIO_Code_Samples

https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md


BCM2835-ARM-Peripherals.pdf

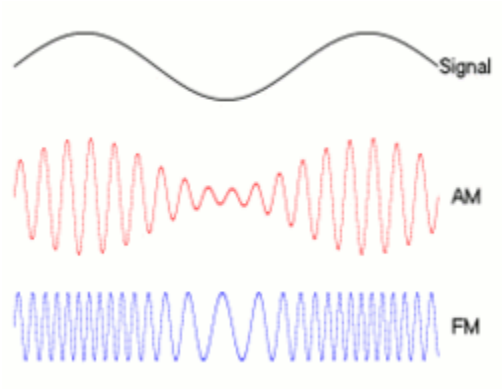

QA7_rev3.4.pdf


_blink.c

Lab 2/3: Creating an FM transmitter using Pi

Let's visit some FM modulation basics,

Frequency modulation(FM) is the encoding of information in a carrier wave by varying the instantaneous frequency wave. The instantaneous frequency deviation, the frequency difference between the carrier and the central frequency, is proportional to the message or modulating signal. FM is widely used FM radio broadcasting.



A typical FM modulation is shown here with comparison to AM.

Direct FM modulation can be achieved by directly feeding the message to the input of a VCO (voltage controlled oscillator).

Few terminology and equation,

FM wave: $s(t) = A_c\cos[\omega_c t+\theta_m(t)]$

Bandwidth: $B_T = 2(\Delta f + f_m)$ Hz

Δf is peak frequency deviation, f_m is maximum frequency of message signal

So, to get an FM wave we have the following requirement:

A high frequency carrier wave.

A message signal.

Carrier wave:

Since we are familiar with accessing the GPIO registers and program them from lab exercise, if you refer section 6.2 (page 102) in BCM2835 it provides all the alternate functions of GPIO pins. So GPIO 4 can provide GPIOCLK0 (a clock), Can we use this as carrier wave? Yes, since the clock is connected to the GPIO, we can directly get an analog output from the GPIO4, once set for that function.

Message signal:

We will use an audio file as message signal here. You will be knowing there are lot of different audio format like mp3, aac, wma, wav, etc. Find out the differences in these formats.

We will be choosing WAV audio format here, reason it is a lossless and typically uncompressed audio format with a small ~40 Byte header. We can use other audio formats, then appropriately audio decoding needs to be done, which is out of scope of this experiment.

Experiment 1:

- Select the GPCLK0 function in GPFSEL0 register
- Read section 6.3 in BCM2835
- Initialize the CM_GP0CTL (page 107) register
    - For clock sources refer the below

0	0 Hz	Ground
1	19.2 MHz	Oscillator
2	0 Hz	testdebug0
3	0 Hz	testdebug1
4	0 Hz	PLLA
5	1000 MHz	PLLC (changes with overclock settings)
6	500 MHz	PLLD
7	216 MHz	HDMI Auxiliary
8-15	0 Hz	Ground

- Only PLLD and oscillator are stable clock source, so choose PLLD
- The clock is ready for operation, set the GP clock divisor register (page 108) CM_GP0DIV to get the desired frequency
    1) Ex1: PLLD frequency = 500 MHz
       Required frequency 100 MHz
       Clock divisor = 500/100 = 5 (put this in CM_GP0DIV be aware of the DIVI and DIVF)
       DIVI – it divides the frequency in integral MHz value (ex: 1, 5, 100, 250, etc. MHz)
       DIVF – it divides the frequency in non-integral MHz range (ex: 1.5, 10.2, 200.7, etc. MHz)
       Here DIVI = 5 DIVF=0
    2) Ex2: Required frequency = 250.5
       Clock divisor = 1.99 (DIVI = 1 DIVF=990)
- Change the value of DIVI and measure the frequency in CRO, vary it from 1 to 500 MHz, observe the waveform, note your observation, **can you explain your observation?**

Experiment 2:

- Since we are now ready with our carrier wave from experiment 1, let's modulate a message wave with it.
- We choose your favorite audio file and convert it into WAV audio (16 bit) using any audio converter available, or download one WAV audio file.
- Let say I choose the carrier frequency = 100 MHz (DIVI = 5)
- As per FM wave I need to create variation in this 100 MHz frequency based on my audio file. Can I use DIVF to achieve that? yes.
- DIVF is 12-bit that is why we use 16-bit audio file (remove first 44 bytes from file they are WAV header). Normalize the audio file and scale it to the bandwidth required.
    (Audio_value/ (2^16))*bandwidth
    This step make sure that the audio file values are limited to a range, i.e. if bandwidth is 8 kHz then audio_values are normalized as 8 to -8, this is the frequency deviation.

- We add/subtract the frequency deviation from the clock divisor register value (basically changing the DIVF), to vary the frequency up or down.
  Ex: for audio range 8 to -8 (8kHz Bandwidth)
  Min frequency 100 - 0.008 = 99.002 MHz (DIVI = 4, DIVF = 4088) (4088? 2^{12} - 8)
  Max frequency 100 + 0.008 = 100.008 MHz (DIVI = 5, DIVF = 8)
- Please note here raspberry pi can process audio file way much faster than human ears, so before writing the next audio value to the clock divisor give appropriate delay.
- Now we are transmitting the FM wave. Connect a wire which will act as an antenna at GPIO4, you can now listen your favorite song in your mobile radio at the frequency you have chosen.
- Vary the bandwidth from 1 to 100 kHz and observe the quality of audio? Can you explain why this difference in quality?
- Vary the delay (referred as speed now on) between two successive load of clock divisor register, what do you observe?
- Present your observation of all the experiment.

A pseudo algorithm for the experiment 1 and 2.

```
/*main*/
{
    // setup GPIO
    //setup clock control register
    //initialize carrier (first DIVI for clock divisor, ex. 5 in case of 100 MHz)
    //modulate the clock
}
/*modulate clock*/
{
    //read audio values from the audio file (read as short 16-bit)
    //remove header,
    //for each audio value from the audio file
        //Normalize to 65536 and scale to bandwidth
        //Add the value to the clock divisor (full register) (-ve value will take care of subtraction)
        //provide delay(speed)
}
```

mypiFm.c