Low Level Programming of the Raspberry Pi in C

Submitted by Pieter-Jan on Fri, 24/05/2013 - 05:10

One of the things that disappointed me when I first got my Raspberry Pi was the fact that everybody was doing very high level programming with it. I wanted to program it like I used to do with microcontrollers, but it seemed like this was not as easy as I thought it would be. I believe however, that for embedded applications, you should be very cautious about dependencies, thus try to use as few libraries as possible. This article will show you how to program the Raspberry Pi in C code in a low level way. Tackling this can be a challenge, so let's get started!

Note that this tutorial will make use of the BCM2835 ARM Peripherals Manual. I will also refer to this as "datasheet", because it sound natural to me.

Getting Started

If you have any experience with low level programming of microcontrollers, then you know that **everything is done by writing to the registers in the memory of the device**. If you, for example, want to set a pin high, then you write a "1" to the register that corresponds to that certain pin. That means that, if we want to program our Raspberry Pi, we need to gain access to the memory of the BCM2835 first. This is by far the most complex part of programming the Raspberry Pi, so don't be discouraged!

First, we need a header file which defines some macro's.

RPI.h

```
#include <stdio.h>

#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <unistd.h>

#define BCM2708_PERI_BASE       0x20000000
#define GPIO_BASE               (BCM2708_PERI_BASE + 0x200000)  // GPIO controller

#define BLOCK_SIZE              (4*1024)

// IO Acces
struct bcm2835_peripheral {
    unsigned long addr_p;
    int mem_fd;
    void *map;
    volatile unsigned int *addr;
};

struct bcm2835_peripheral gpio = {GPIO_BASE};
```

```
extern struct bcm2835_peripheral gpio;  // They have to be found somewhere, but can't be in the header
```

So, what is this? If we look at the Broadcom BCM2835 ARM Peripherals manual at page 6, we read:

> *Physical addresses range from 0x20000000 to 0x20FFFFFF for peripherals. The bus addresses for peripherals are set up to map onto the peripheral bus address range starting at 0x7E000000. Thus a peripheral advertised here at bus address 0x7Ennnnnn is available at physical address 0x20nnnnnn.*

So the BCM2708_PERI_BASE macro contains the **physical** adress value at which the peripheral registers start. This is the adress we will need to use in our program. The **virtual** address value is 0x7E000000, and it are these virtual adresses that will be found in the datasheet.

There are a lot of different peripherals available on the BCM2835 (Timers, USB, GPIO, I2C, ...), and they will be defined by an offset to this virtual adress! For example, if we are interested in the GPIO peripheral (like in the example code above), we can find in the manual at page 90 that the virtual address is 0x7E200000. This means that the offset to the physical adress will be 0x200000 which explains the GPIO_BASE.

for every peripheral we define a struct of the type `bcm2835_peripheral`, which will contain the information about the location of the registers. Then we intialize it with the `map_peripheral()` function from the c file below. It is not so important to understand how this works in detail, as you can just copy paste the code. If you do want to know this, search for "accessing /dev/mem" on the internet. To release the peripheral, we can use `unmap_peripheral()`

RPI.c

```
#include "RPI.h"

struct bcm2835_peripheral gpio = {GPIO_BASE};

// Exposes the physical address defined in the passed structure using mmap on /dev/mem
int map_peripheral(struct bcm2835_peripheral *p)
{
   // Open /dev/mem
   if ((p->mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
      printf("Failed to open /dev/mem, try checking permissions.\n");
      return -1;
   }

   p->map = mmap(
      NULL,
      BLOCK_SIZE,
      PROT_READ|PROT_WRITE,
      MAP_SHARED,
      p->mem_fd,      // File descriptor to physical memory virtual file '/dev/mem'
      p->addr_p       // Address in physical map that we want this memory block to expose
   );

   if (p->map == MAP_FAILED) {
        perror("mmap");
        return -1;
   }

   p->addr = (volatile unsigned int *)p->map;

   return 0;
}

void unmap_peripheral(struct bcm2835_peripheral *p) {

    munmap(p->map, BLOCK_SIZE);
    close(p->mem_fd);
}
```

Now we have code to access the memory of the BCM2835, we can start to use the peripherals to do some badass projects! If you didn't fully grasp the above code, this is not a problem! Everything will start making sense after you see that code in action.

GPIO

GPIO stands for General Purpose In/Out, so this peripheral will allow us to set a pin either as in- or output and read or write (to) it. We can **add the macro's below to our header file (RPI.h)** to support this. I will explain how they work underneath.

```
// GPIO setup macros. Always use INP_GPIO(x) before using OUT_GPIO(x)
#define INP_GPIO(g)   *(gpio.addr + ((g)/10)) &= ~(7<<(((g)%10)*3))
#define OUT_GPIO(g)   *(gpio.addr + ((g)/10)) |=  (1<<(((g)%10)*3))
#define SET_GPIO_ALT(g,a) *(gpio.addr + (((g)/10))) |= (((a)<=3?(a) + 4:(a)==4?3:2)<<(((g)%10)*3))

#define GPIO_SET  *(gpio.addr + 7)  // sets   bits which are 1 ignores bits which are 0
#define GPIO_CLR  *(gpio.addr + 10) // clears bits which are 1 ignores bits which are 0

#define GPIO_READ(g)  *(gpio.addr + 13) &= (1<<(g))
```

At first sight, those macro's look terrifying! But actually they are quite easy to understand, so don't be scared!

`gpio` is a struct of the type `bcm2835_peripheral` that was initialized with `map_peripheral()` at the adress `GPIO_BASE`. This is what we discussed above. To be able to use this code, this need to be done first!

To understand properly how the macro's work, we need to take a look at the Broadcom BCM2835 ARM Peripherals manual at chapter 6 (page 89 etc.). We will go over all the macro's in the following text.

```
#define INP_GPIO(g) *(gpio.addr + ((g)/10)) &= ~(7<<(((g)%10)*3))
```

This macro sets pin "g" as an input. How?

In the datasheet at page 91 we find that the **GPFSEL registers are organised per 10 pins**. So one 32-bit register contains the setup bits for 10 pins. `*gpio.addr + ((g))/10` is the **register address** that contains the GPFSEL bits of the pin "g" (and 9 other pins we are not interested in). Remember that the result of a division on integers in C does not contain the part behind the komma. There are **three GPFSEL bits per pin** (000: input, 001: output). The location of these three bits inside the GPFSEL register is given by `((g)%10)*3` (three times the remainder, remember the modulo % operator).

GPIO Function Select Registers (GPFSELn)

SYNOPSIS The function select registers are used to define the operation of the general-purpose I/O pins. Each of the 54 GPIO pins has at least two alternative functions as defined in section 16.2. The FSEL{n} field determines the functionality of the nth GPIO pin. All unused alternative function lines are tied to ground and will output a "0" if selected. All pins reset to normal GPIO input operation.

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
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

So to set pin "g" as an input, we need to set these bits to "000". This is done with a AND-operation between the GPSEL register and a string of which everything except those bits are 1. This string is created by bitshifting 7, which is binary 111, over `((g)%10)*3` places, and taking the inverse of that.

A little example: Say that we want to use **gpio4 as an input**.

- The GPSEL address containing the function select bits for gpio4 is `*gpio.addr + 4/10` = `*gpio.addr + 0`
- Inside this 32-bit register, the bits concerning gpio4 start from bit `(4%10)*3` =12
- If we bitshift binary 111 over 12 positions we get: `00000000000000000111000000000000`
- We have to make those three bits zero in the GPFSEL register, so we have to invert that 32-bit string and perform an AND operation with the GPSEL register. (X = unknown)

```
     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1
     X X X X X X X X X X X X X X X X X X X X X X X X X X X X X   (GPFSEL old)
AND  ------------------------------------------------------------
     X X X X X X X X X X X X X X X X 0 0 0 X X X X X X X X X X   (GPFSEL new)
```

```
#define OUT_GPIO(g)   *(gpio.addr + ((g)/10)) |=  (1<<(((g)%10)*3))
```

To set pin "g" as an output, we have to set its setup bits to 001. This is done in the same way as before, but now we bitshift a 1, and use a OR operation on

the normal (not inverted) string.

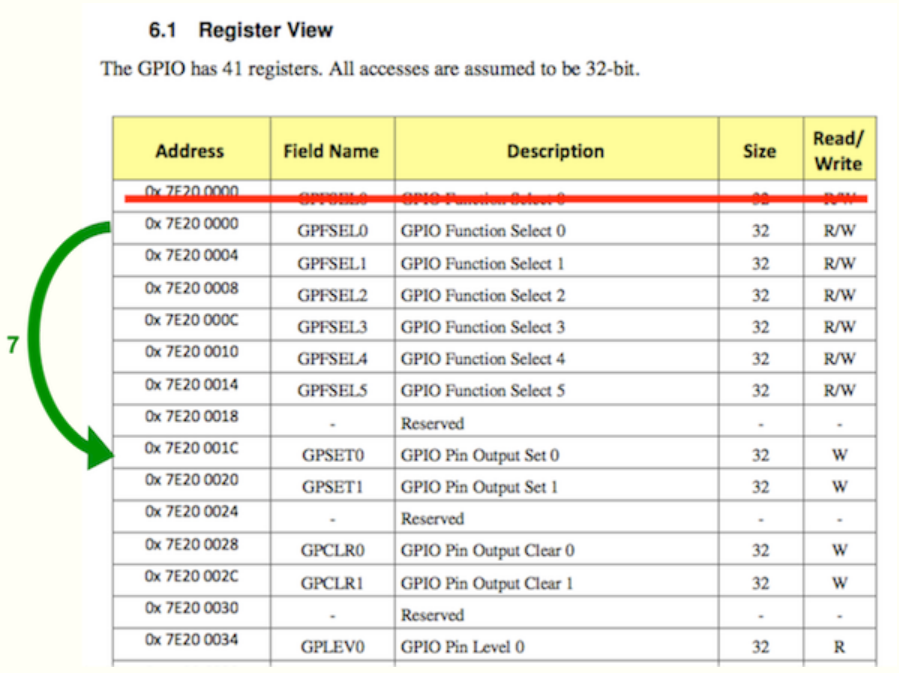
This is also the reason that the comment says to "always use INP_GPIO(x) before using OUT_GPIO(x)". This way you are sure that the other 2 bits are 0, and justifies the use of a OR operation here. If you don't do that, you are not sure those bits will be zero and you might have given the pin "g" a different setup.

```
#define SET_GPIO_ALT(g,a) *(gpio.addr + (((g)/10))) |= (((a)<=3?(a) + 4:(a)==4?3:2)<<(((g)%10)*3))
```

As you can see in the picture above, input and output are not the only functionalities available for the pins. There are 6 "alternate function" possibilities for each pin. It are these alternate functions that you will want to use if you need to use other peripherals than the gpio, like I2C and SPI. The alternate function possibilities are on page 102 in section 6.2 of the manual. With `SET_GPIO_ALT(g,a)` you can set pin g to function 0,1,... In the part about I2C below, we will use this function to set GPIO0 and GPIO1 to SDA and SCL for use with I2C.

```
#define GPIO_SET  *(gpio.addr + 7)
```

In the datasheet on page 90, we seet that the GPSET register is located 7 32-bit registers further than the gpio base register. There is a small mistake on this page of the datasheet, as the first line is printed twice.



6.1 Register View

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
0x 7E20 0030	-	Reserved	-	-
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R

`gpio.addr` is defined as a pointer to an unsigned int, which is 32-bit on the Raspberry Pi. When you add an integer to a pointer, it will know that we are working with 32-bit values, so we don't need to multiply this with 4 (7*4 = 0x1C) to obtain the address from the datasheet.

So to use the macro GPIO_SET to set e.g. gpio 4 pin we can set the 4'th bit of this register to 1 using a bitshift:

```
GPIO_SET = 1 << 4;
```

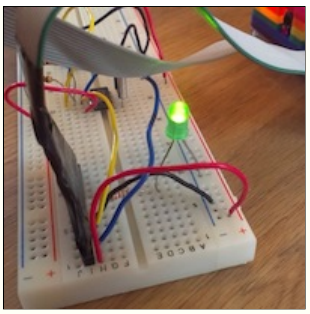
It's important to note that **writing a "0" to the GPIO_SET register will not reset that pin**. You have to write a "1" to the GPIO_CLR register to reset a pin. This is very useful, as this means that we can use the above bitshift operation to write to all pins, without taking the risk of accidentally resetting a pin we forgot that was on. Awesome.

The gpio clear is ofcourse similar.

```
#define GPIO_READ(g)  *(gpio.addr + 13) &= (1<<(g))
```

To read a pin we simply perform a bitwise AND on the GPLEV register and the pin we are interested in (g). The result is the value of that pin.

Example - Blink a LED



Hopefully the above analysis helped you understanding how to use the GPIO functionality of the Raspberry Pi. It really is nothing more than writing 1's and 0's to the correct registers ... As a little example, here is the code to blink a led. Note that pin 7 corresponds to gpio 4 if you look a the pinout. To run this, make sure you added the GPIO macro's to your header file!

Don't try this at home (without a resistor in series with the led)!

main.c

```
#include "RPI.h"

int main(
{
  if(map_peripheral(&gpio) == -1)
  {
    printf("Failed to map the physical GPIO registers into the virtual memory space.\n");
    return -1;
  }

  // Define pin 7 as output
  INP_GPIO(4);
  OUT_GPIO(4);

  while(1)
  {
    // Toggle pin 7 (blink a led!)
    GPIO_SET = 1 << 4;
    sleep(1);

    GPIO_CLR = 1 << 4;
    sleep(1);
  }

  return 0;
}
```

I2C

The peripheral we will need to use the I2C functionality of the Raspberry Pi is the BSC (Broadcom Serial Controller). Information about this can be found in de datasheet chapter 3 (p. 28 and further). There are two BSC's: BSC0 and BSC1. We will use BSC0. First we map all the registers by **adding the following macro's to our RPI.h file**.

```
#define BSC0_BASE     (BCM2708_PERI_BASE + 0x205000)  // I2C controller

extern struct bcm2835_peripheral bsc0;

// I2C macros
#define BSC0_C          *(bsc0.addr + 0x00)
#define BSC0_S          *(bsc0.addr + 0x01)
#define BSC0_DLEN     *(bsc0.addr + 0x02)
#define BSC0_A          *(bsc0.addr + 0x03)
#define BSC0_FIFO     *(bsc0.addr + 0x04)

#define BSC_C_I2CEN     (1 << 15)
#define BSC_C_INTR      (1 << 10)
#define BSC_C_INTT      (1 << 9)
#define BSC_C_INTD      (1 << 8)
#define BSC_C_ST        (1 << 7)
#define BSC_C_CLEAR     (1 << 4)
#define BSC_C_READ      1

#define START_READ      BSC_C_I2CEN|BSC_C_ST|BSC_C_CLEAR|BSC_C_READ
#define START_WRITE     BSC_C_I2CEN|BSC_C_ST

#define BSC_S_CLKT  (1 << 9)
#define BSC_S_ERR     (1 << 8)
#define BSC_S_RXF     (1 << 7)
#define BSC_S_TXE     (1 << 6)
#define BSC_S_RXD     (1 << 5)
#define BSC_S_TXD     (1 << 4)
#define BSC_S_RXR     (1 << 3)
#define BSC_S_TXW     (1 << 2)
#define BSC_S_DONE    (1 << 1)
```

```
#define BSC_S_TA      1

#define CLEAR_STATUS    BSC_S_CLKT|BSC_S_ERR|BSC_S_DONE

// I2C Function Prototypes
void i2c_init();
void wait_i2c_done();
```

Not much special here, except maybe for the `START_READ`, `START_WRITE` and `CLEAR_STATUS`. Of course, these are just the **combinations** (OR) of bits in the status register that need to be set for that action to occur. It's just easier to do it like that than to set all the bits seperately every time.

Then we need to **add the following functions to our RPI.C file**.

```
struct bcm2835_peripheral bsc0 = {BSC0_BASE};

// Initialize I2C
void i2c_init()
{
    INP_GPIO(0);
    SET_GPIO_ALT(0, 0);
    INP_GPIO(1);
    SET_GPIO_ALT(1, 0);
}

// Function to wait for the I2C transaction to complete
void wait_i2c_done() {

        int timeout = 50;
        while((!((BSC0_S) & BSC_S_DONE)) && --timeout) {
            usleep(1000);
        }
        if(timeout == 0)
            printf("Error: wait_i2c_done() timeout.\n");
}
```

Notice how we access the memory of the BSC0 peripheral with a struct of the type `bcm2835_peripheral` called bsc0. We initalize it so that the address points to BSC0_BASE.

The classic I2C delay is implemented with `wait_i2c_done()`. It constantly polls the "DONE" bit of the BSC0 S register (status). This bit becomes 1 when the transfer is completed. See datasheet p. 32.

Now we need to make sure that the GPIO pins are set up properly for I2C. For this we need one pin to behave as **SDA**, and one pin to behave as **SCL**. If we look at the pinout again, we see that GPIO0 and GPIO1 are the proper pins for this. They have to be enabled first though. For this we use the `i2c_init()` function. Inside you will find the `SET_ALT_GPIO()` macro used to set these pins to alternate functionality 0. The reason we set these to alternate functionality 0 is because the datasheet shows the following table at page 102:

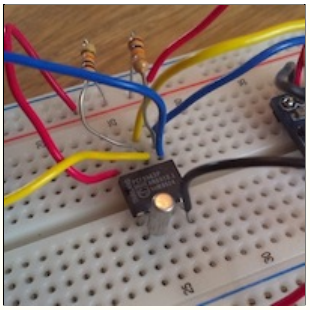
	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	<reserved>			
GPIO1	High	SCL0	SA4	<reserved>			
GPIO2	High	SDA1	SA3	<reserved>			
GPIO3	High	SCL1	SA2	<reserved>			

Ok, so after initializing the I2C we are very close to start sending data over the I2C bus! To send or write some data, we need to take the following steps:

- Write the **I2C slave address** of the device you want to communicate with (you can find this in the datasheet) to the BSC0_A adress (slave adress register).
- Write the **amount/quantity of bytes** you want to send/receive to the BSC0_DLEN register (Data Length).
- If you want to write to the slave, write the bytes you want to write in the BSC0_FIFO. Remember: FIFO = First In First Out!
- Clear the status register: `BSC0_S = CLEAR_STATUS`
- Start write or start read by writing to the BSC0 C register (Control): `BSC0_C = START_WRITE` or `BSC0_C = START_READ`

Are you totally confused? Me too! Let's take a look at some examples!

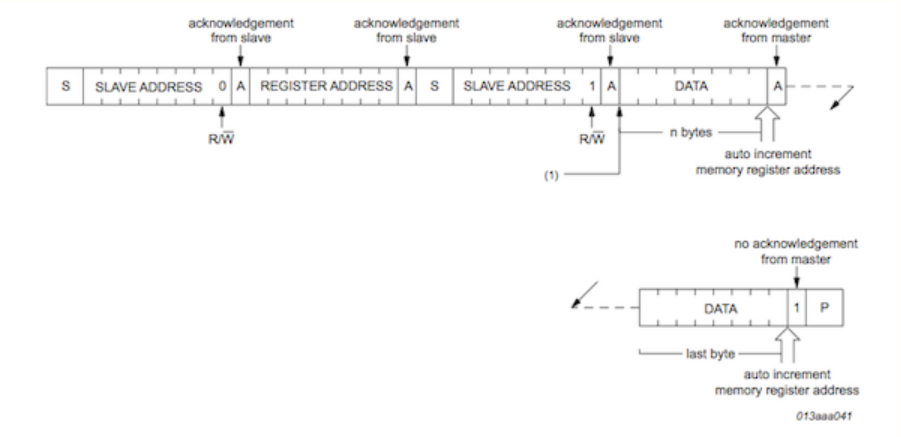
Example 1 - Real Time Clock (RTC)



In this example we will read a Real Time Clock (RTC) with a Raspberry Pi using I2C. The RTC we are using is the PCF8563, a very classic device. Normally you would want to be able to write the RTC as well for initializing, but I want to start with a simple example.

The data from the PCF8563 starts at register 2 (seconds) and the 6 sequential adresses contain minutes, hours, ... So in total there are 7 registers we need to read, starting from adress 2.

Reading data from the RTC is described in the PCF8563 datasheet and should be performed like this:



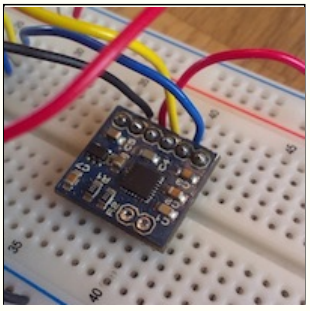
The first step, writing the slave address to the bus, will be done automatically for us if we write this address to the BSC0_A register. So we have to do this before starting communication. After this we need to write the register address of the RTC we want to start reading from to the bus. In our case, this is "2". After that, it will start sending the data, so we have to start reading data from the bus.

The code example below nicely illustrates how this is done on the Raspberry Pi.

```
#include "RPI.h"
#include <stdio.h>
#include <time.h>
#include <fcntl.h>

struct tm t;

/* Functions to do conversions between BCD and decimal */
unsigned int bcd_to_decimal(unsigned int bcd) {
    return ((bcd & 0xf0) >> 4) * 10 + (bcd & 0x0f);
}
unsigned int decimal_to_bcd(unsigned int d) {
    return ((d / 10) << 4) + (d % 10);
}

int main(int argc, char *argv[]) {

    /* Gain access to raspberry pi gpio and i2c peripherals */
    if(map_peripheral(&gpio) == -1) {
        printf("Failed to map the physical GPIO registers into the virtual memory space.\n");
        return -1;
    }
    if(map_peripheral(&bsc0) == -1) {
        printf("Failed to map the physical BSC0 (I2C) registers into the virtual memory space.\n");
        return -1;
    }

    /* BSC0 is on GPIO 0 & 1 */
    i2c_init();

    /* I2C Device Address 0x51 (See Datasheet) */
    BSC0_A = 0x51;

    /* Write operation to restart the PCF8563 register at index 2 ('secs' field) */
    BSC0_DLEN = 1;            // one byte
    BSC0_FIFO = 2;            // value 2
    BSC0_S = CLEAR_STATUS;    // Reset status bits (see #define)
    BSC0_C = START_WRITE;     // Start Write (see #define)
```

```
    wait_i2c_done();

    /* Start Read of RTC chip's time */
    BSC0_DLEN = 7;
    BSC0_S = CLEAR_STATUS;  // Reset status bits (see #define)
    BSC0_C = START_READ;    // Start Read after clearing FIFO (see #define)

    wait_i2c_done();

    /* Store values in struct */
    t.tm_sec = bcd_to_decimal(BSC0_FIFO & 0x7f);
    t.tm_min = bcd_to_decimal(BSC0_FIFO & 0x7f);
    t.tm_hour = bcd_to_decimal(BSC0_FIFO & 0x3f);
    t.tm_mday = bcd_to_decimal(BSC0_FIFO & 0x3f);
    t.tm_wday = bcd_to_decimal(BSC0_FIFO & 0x07);
    t.tm_mon = bcd_to_decimal(BSC0_FIFO & 0x1f) - 1; // 1-12 --> 0-11
    t.tm_year = bcd_to_decimal(BSC0_FIFO) + 100;

    printf("%02d:%02d:%02d %02d/%02d/%02d (UTC on PCF8563)\n",
        t.tm_hour,t.tm_min,t.tm_sec,
        t.tm_mday,t.tm_mon + 1,t.tm_year - 100);

    /* Unmap the peripheral */
    unmap_peripheral(&gpio);
    unmap_peripheral(&bsc0);decimal_to_bcd
}
```

Example 2 - IMU (MPU6050)



For reading the MPU6050 IMU I will just provide the source code and hope that you understand what to do with it after the previous examination. If there still is something that is not clear, please contact me, and I will add more explanation here.

MPU6050.h

```
#ifndef _INC_MPU6050_H
#define _INC_MPU6050_H

#include "RPI.h"
#include <stdio.h>

#define MPU6050_ADDR 0b01101001 // 7 bit adres
#define CONFIG 0x1A
#define ACCEL_CONFIG 0x1C
#define GYRO_CONFIG 0x1B
#define PWR_MGMT_1 0x6B

void MPU6050_Read(short * accData, short * gyrData);
void MPU6050_Init(void);
void MPU6050_SetRegister(unsigned char regAddr, unsigned char regValue);

#endif
```

MPU6050.c

```
#include "MPU6050.h"

void MPU6050_Init(void)
{
//  MPU6050_SetRegister(PWR_MGMT_1, 0x80);  // Device Reset
    MPU6050_SetRegister(PWR_MGMT_1, 0x00);  // Clear sleep bit
    MPU6050_SetRegister(CONFIG, 0x00);
    MPU6050_SetRegister(GYRO_CONFIG, 0x08);
    MPU6050_SetRegister(ACCEL_CONFIG, 0x08);
}

void MPU6050_SetRegister(unsigned char regAddr, unsigned char regValue)
{
    // See datasheet (PS) page 36: Single Byte Write Sequence

    // Master:   S  AD+W       RA       DATA       P
    // Slave :            ACK      ACK        ACK
```

```
    BSC0_A = MPU6050_ADDR;

    BSC0_DLEN = 2;
    BSC0_FIFO = (unsigned char)regAddr;
    BSC0_FIFO = (unsigned char)regValue;

    BSC0_S = CLEAR_STATUS;  // Reset status bits (see #define)
    BSC0_C = START_WRITE;     // Start Write (see #define)

    wait_i2c_done();
}

void MPU6050_Read(short * accData, short * gyrData)
{
    // See datasheet (PS) page 37: Burst Byte Read Sequence

    // Master:   S  AD+W       RA       S  AD+R          ACK        NACK  P
    // Slave :            ACK      ACK          ACK DATA       DATA

    BSC0_DLEN = 1;        // one byte
    BSC0_FIFO = 0x3B;     // value of first register
    BSC0_S = CLEAR_STATUS;  // Reset status bits (see #define)
    BSC0_C = START_WRITE;     // Start Write (see #define)

    wait_i2c_done();

    BSC0_DLEN = 14;

    BSC0_S = CLEAR_STATUS;  // Reset status bits (see #define)
    BSC0_C = START_READ;      // Start Read after clearing FIFO (see #define)

    wait_i2c_done();

    short tmp;

    int i = 0;
    for(i; i < 3; i++)    // Accelerometer
    {
  tmp = BSC0_FIFO << 8;
  tmp += BSC0_FIFO;
  accData[i] = tmp;
    }

    tmp = BSC0_FIFO << 8;   // Temperature
    tmp += BSC0_FIFO;

    i = 0;
    for(i; i < 3; i++)    // Gyroscope
    {
  tmp = BSC0_FIFO << 8;
  tmp += BSC0_FIFO;
  gyrData[i] = tmp;
    }
}
```

GitHub Repository

All this code can also be found in my GitHub repository: https://github.com/Pieter-Jan/PJ_RPI. You will need CMake (`sudo apt-get install cmake`) and git (`sudo apt-get install git`) to use it.

To use it, create a folder on the pi where you want to put the files in and extract the git repository in there:

```
git clone git://github.com/Pieter-Jan/PJ_RPI.git
```

Then you have to install the library, so cd into the PJ_RPI directory and:

```
mkdir Build
cd Build
cmake ../
make
sudo make install
```

Now the library is installed and you can build the examples that are in the "Examples" folder using CMake.

If you want to build the MPU6050 example you will also need ncurses for the visualization (`sudo apt-get install ncurses-dev`).

Tags:
Raspberry Pi