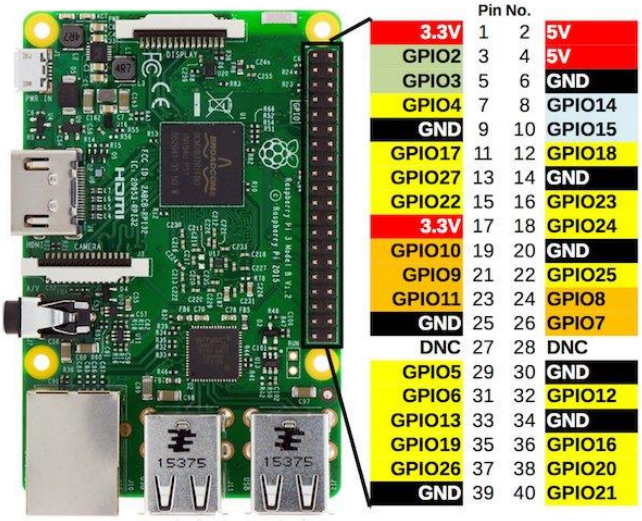
Note: register addresses specific to BVM2836/2837

Raspberry Pi GPIO Programming Version 1.0

GPIOs, an Introduction

First of all, GPIO stands for General Purpose Input Output. They are general purpose meaning that the only functionality that they possess is either being high or low; rather, they are switches that can either be driving an LED on or off or sending/receiving signals to and from another device. High is 5.5V output while low 0V output.

There is a total of 40 pins on the Raspberry Pi 3 b+ where 26 are GPIO pins; the rest being power, ground, or cryptically described as "other". Unfortunately, the GPIOs are not laid out in the most sensible order nor actually may be usable by the user. GPIO 14 (Tx) & 15 (Rx) are reserved for UART, GPIO 2 & 3 are reserved for I2C, GPIO 4 is reserved for 1-wire, and GPIO 10, 9, 11, 8, and 7 are reserved for SPI. Now, these interfaces can be disable in Raspi-Config allowing the user to gain complete control over these pins.



Using a GPIO all comes down to writing 1's and 0's to the correct register in order to switch the pin high or low. The trick is knowing how to access the correct register and write those 1's and 0's in a "low-level" programming language like C without resorting to any external libraries. After all, what's the fun in not understanding how it all works to the last detail?

The Broadcom BCM2837 ARM & Working with Peripherals

The 3b+ uses the Broadcom BCM2837 ARM as its computer processor. It can be easily mistaken as the BSM2835 processor, the processor used on older iterations of the Raspberry Pi, as it's not much different. In fact, the manual for BCM2837 is practically identical to one for BCM2835 except for a few, but certainly important, differences when it comes to physical addresses for peripherals.

The physical addresses of a processor are memory addresses represented in the form of a binary numbers on the address bus circuitry in order to enable the data bus to access a particular storage cell of main memory or a register of memory mapped I/O device. To break this statement down…

- A memory address is a reference to a specific memory location; it's the location in either software or hardware where that memory is stored. For software, could be on the stack or heap within RAM. For hardware, perhaps address registers.
- The address bus circuitry is a bus used to specify a physical address. When the processor needs to read or write to a memory location, it specifies that memory location on the address bus (the value to be read or written is sent on the data bus).

    - Bus is a communication system that transfers data between components inside/between computers and peripherals. The expression bus refers to the wires, optical fiber, etc. used as the medium for this communication system as well as the protocol employed.
- Main memory, also known as primary storage, is memory directly accessible to the CPU, i.e. RAM, processor registers, and processor cache. Main memory is connected via the address bus and data bus (collectively known as the memory bus).
- Memory mapped I/O device (MMIO) refers to two methods of performing I/O between the processor and peripheral devices in a computer (peripheral here referring to everything besides the processor). The memory and registers of the I/O devices are mapped to (and associated with) address values. Essentially, this allows CPU instructions that access memory, e.g. RAM, to also access devices. Each I/O device monitors the address bus and responds to CPU access assigned to that device to connect the data bus to the desired device's hardware register.
    - For the Pi, all hardware interactions occur using MMIO

To put it more plainly, memory addresses are locations of memory is software or hardware. Main memory is directly accessible to the processor using memory addresses and the memory (address and data) bus. I/O devices, peripherals, can also be accessed by the processor directly as well if their memory and registers are memory mapped to an address that the processor can use. Once memory mapped, peripherals are now associated with binary numbers representing addresses. The processor can now access peripherals using the same instructions it has been using to access memory locations in main memory; functionally, there is no difference between accessing main memory and peripherals at this point to the processor. These memory addresses mapped to a specific peripheral are thus referred to as physical addresses because they correspond to memory and hardware registers of said peripheral (unlike the memory locations in main memory). Each I/O device is responsible for monitoring the address bus to know when the CPU is attempting to access said device. When attempted, the device then connects the data bus to its register(s) to complete the process.

The most important take away from this discussion is the functionality of the physical addresses for peripherals: it tells us where on the BCM2837 were we need to write our 1's and 0's to gain functionality of the GPIO pins.

There are many peripheral devices on the BCM2837, and to make it easy to know the physical address of a device, the manual defines a base peripheral address (more accurately an address range) and an offset to a specific peripheral. For this processor in particular, page 6 and 90 of the manual reveals

- Peripheral physical address ranges from 0x3F000000 to 0x3FFFFFFF.
    - Therefore, the base peripheral address will be 0x3F000000.
- Physical bus addresses are set up to map onto the peripheral bus address range starting at 0x7E000000 (virtual memory address). Peripheral bus addresses advertised here at bus address 0x7Ennnnnn is available at physical address 0x3Fnnnnnn; in other words, convert the virtual memory address that data sheet tells you from 0x7Ennnnnn to 0x3Fnnnnnn to get the physical address.
- Virtual memory is a memory management technique that creates an illusion of very large main memory by combining active RAM and inactive memory on direct-access storage devices, e.g. SSD and HDD. Virtual memory addresses are then just memory locations of memory within this contiguous address space that hold both the application and its data.

- GPIO registers begin at virtual address 0x7E200000. As the base address is 0x7E000000 for peripherals, there is a 0x2000000 offset.
    - These GPIO registers perform specific functions like function select, output, clear, read, and event detection.

Now we have the base address of the GPIO registers: 0x3F000000 + 0x2000000. To select a specific function (select, output, read, event detect, rising edge detect, high detect, etc…) to perform on a GPIO pin, we will have to modify this address further with an additional offset to select the appropriate register; this process is detailed in the next section. In total, GPIO registers range from 0x7E200000 to 0x7E2000B0 in 0x4 increments. This increment refers to the fact that each register contains 4 bytes, or 32 bits, of data.

Diving into the Programming

Now that we have an understanding of the BCM2837, how it deals with peripherals, and functionality of the physical base address for GPIO registers, we can move on to how we employ our knowledge into developing a program in a low-level language to use GPIOs. The general procedure will be to

1. Map physical memory addresses into virtual memory
2. Develop GPIO function macros
3. LED Blink Test

Mapping Physical Memory Addresses in Virtual Memory

To access our GPIO_BASE physical address in our program, we will use a device in /dev/ called mem and a function called mmap() to map out physical addresses into virtual memory. First, let us put down what we've learned so far. We know the peripheral base address and the GPIO offset:

```
#define BCM2837_PERI_BASE    0x3F000000
#define GPIO_BASE            (BCM2708_PERI_BASE + 0x200000)
```

Next, will use a device in /dev/ called mem. /dev/mem is a character device file that is an image of the main memory of the computer. A character device file (also known as raw devices) provides unbuffered, direct access to the hardware device. Providing a byte address (in contrast to a word address) to mem will be interpreted as a physical memory address. We will open /dev/mem for reading and writing to obtain a file descriptor for this device for the eventual mmap() function call:

```
mem_fd = open("/dev/mem", O_RDWR|O_SYNC);
```

Now that file descriptor is obtained, we move onto employing mmap(). mmap() is a function from <sys/mman.h> (C standard library) that maps files or devices (such as GPIOs) into memory; mapping into memory being a technique where peripherals are treated as if they were located in main memory. In other words, we create a new mapping in the virtual address space of the calling process. This renders memory addresses associated with the GPIO registers that behave like a memory address to a variable. /dev/mem

is used directly via the file descriptor created when the device was opened as an input to mmap(). The function call where “prot” = “protection” and “fd” is “file descriptor”:

```
void* map_addr = mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset)
```

Upon success, mmap() returns a pointer to an unsigned integer representation of the first memory address in the mapped area. Pointer arithmetic can then be done on this returned pointer to access the next length/4 number of addresses in the mapped area. On error, the value MAP_FAILED ((void *) -1) is returned. This what the function call will look like for us using some variables that we’ve already and yet to define:

```
gpio_base_map_addr = mmap(
      NULL,                 // Any address in our space will do
      BLOCK_SIZE,           // Map length
      PROT_READ|PROT_WRITE, // Enable reading & writing to mapped memory
      MAP_SHARED,           // Shared with other processes
      mem_fd,               // File to map
      GPIO_BASE             // Offset to GPIO peripheral
   );
```

Function input variables that we already know are GPIO_BASE and mem_fd. What we haven’t yet discussed are the bit masking inputs PROT_READ|PROT_WRITE & MAP_SHARED, the NULL being passed as the void* addr input, and BLOCK_SIZE.

The bit masking inputs PROT_READ|PROT_WRITE and MAP_SHARED are more or less standard for a mmap() call so they are not too important to discuss in detail. The NULL being passed as the void* addr input allows the kernel to choose the (page-aligned) address at which to create the mapping. If we specified an address instead, then the kernel “takes a hint” as where to place the mapping by picking a nearby page boundary and attempting to make the mapping there. This detail isn’t too important for us as passing NULL is the most portable and easiest method of creating a new mapping.

BLOCK_SIZE is defining the length of the mapping: the number of bytes starting from void* addr. This length must result in a page aligned mapped region of memory in order for the function call to be successful, so there is a certain length the you can pass to mmap() depending on the system. Most default page table sizes are 4096 kb for each process, and this is what we will define our BLOCK_SIZE as:

```
#define BLOCK_SIZE    4096
```

GPIO Function Macros

GPIO Register Assignments

We now have memory address in the form of a pointer named gpio_base_map_addr that maps directly to the GPIO register base physical memory address 0x3F200000. If you take a look at the BCM2837 datasheet, you will see that on page 90 that this address corresponds to the peripheral bus memory address of 0x7E200000. Remember, this memory address is a virtual memory address of the peripheral bus but translates to a physical address of 0x3F200000. For readability, I'll use both addresses to allow easy comparison between the manual and this reference guide. Looking on page 90, we see that our address 0x3F200000/0x7E200000 is the "GPIO Function Select 0" register. GPIO Function Select 1-5 occupies the next five addresses.

Address	Field Name	Description	Size	Read/Write
0x3F200000/0x7E200000	GPFSEL0	GPIO Function Select 0	32	R/W
0x3F200004/0x7E200004	GPFSEL1	GPIO Function Select 1	32	R/W
0x3F200008/0x7E200008	GPFSEL2	GPIO Function Select 2	32	R/W
0x3F20000C/0x7E20000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x3F200010/0x7E200010	GPFSEL4	GPIO Function Select 4	32	R/W
0x3F200014/0x7E200014	GPFSEL5	GPIO Function Select 5	32	R/W

Each of the five addresses in the table above are the 32-bit transistors that allow users to select the function of specific GPIO pins. Function being input, output, or alternate function 0-5. On page 91-92, we see more discussion on the GPIO function select registers (GPFSEL). The big take away here is that GPFSEL registers are organized per 10 pins and each pin is provided 3 bits in the 32-bit register. Our Pi only has 26 GPIO pins meaning that we will only be using the GPFSEL0 to GPFSEL2 registers, but we can see that the BCM2837 can support up to 53 pins in its 6 total GPFSEL registers. Here is the table for GPFSEL0 register's bit-field assignment.

Bits	Field Name	Description	Type	Reset
31-30	----	Reserved	R	0
29-27	FSEL9	FSEL9 – Function Select 9 (GPIO 9)	R/W	0
26-24	FSEL8	FSEL8 – Function Select 8 (GPIO 8)	R/W	0
23-21	FSEL7	FSEL7 – Function Select 7 (GPIO 7)	R/W	0
20-18	FSEL6	FSEL6 – Function Select 6 (GPIO 6)	R/W	0
17-15	FSEL5	FSEL5 – Function Select 5 (GPIO 5)	R/W	0
14-12	FSEL4	FSEL4 – Function Select 4 (GPIO 4)	R/W	0
11-9	FSEL3	FSEL3 – Function Select 3 (GPIO 3)	R/W	0
8-6	FSEL2	FSEL2 – Function Select 2 (GPIO 2)	R/W	0
5-3	FSEL1	FSEL1 – Function Select 1 (GPIO 1)	R/W	0
2-0	FSEL0	FSEL0 – Function Select 0 (GPIO 0)	R/W	0

Setting bits 29-27 (FSEL9) to 000 will set GPIO pin 9 to an input while 001 makes it an output. The same is true for every FSEL fields in GPFSEL registers. GPFSEL1 register bits 2-0 is for pin 10, GPFSEL2 register bits 2-0 is for pin 20, and so on and so forth.

We will be using GPFSEL, GPSET, GPCLR, and GPLEV registers to set input or output function, high or low state, and read the actual value of a pin respectfully. The summary of addresses for all of these registers except GPFSEL is tabulated below.

Address	Field Name	Description	Size	Read/Write
0x3F20001C/0x7E20001C	GPSET0	GPIO Pin Output Set 0	32	R/W
0x3F200020/0x7E200020	GPSET1	GPIO Pin Output Set 1	32	R/W
0x3F200028/0x7E200028	GPCLR0	GPIO Pin Output Clear 0	32	R/W
0x3F20002C/0x7E20002C	GPCLR1	GPIO Pin Output Clear 1	32	R/W
0x3F200034/0x7E200034	GPLEV0	GPIO Pin Level 0	32	R
0x3F200038/0x7E200038	GPLEV1	GPIO Pin Level 1	32	R

Pages 95 and 96 will be useful in determining how to perform the desired operation to a specific GPIO pin. Summarily, each operation (GPSET, GPCLR, and GPLEV) uses only two registers meaning that each bit in then one GPIO pin. Bits 0 to 31 in GPSET0/GPCLR0/GPLEV0 and bits 32 to 53 in GPSET1/GPCLR1/GPLEV1 are GPIO pins 0 to 31 and 32 to 53 respectfully. Setting 0 to the bit has no effect while setting 1 does the desired operation.

There are a total of 41 registers assigned to GPIOs. Additional operations that you can perform include event, rising/falling edge, and high/low detection. The next section, developing GPIO function macros, will focus on implementing GPFSEL, GPSET, GPCLR, and GPLEV, but it should be easy enough to add the other operations using the following as a guide.

Developing GPIO Macro Functions

The macros we define will accomplish one thing: changing the value of one or more bits in the appropriate register. Accomplishing this action can be thought of as a twostep process.

1. Given the desired GPIO and function we want to perform, determine the memory address of the appropriate register.
2. Perform a bitwise operation to that memory address to set one or more bits in that memory address to either 1 or 0.

GPIO Function Select

Let's first focus on the GPIO Function Select to develop two macros called `GPIO_INP(g)` and `GPIO_OUT(g)` to set the pin to either input or output respectfully. I will first define the macros and then explain how they work.

```
#define GPIO_INP(g) *(gpio_base_map_addr + ((g)/10)) &= ~(7 << (((g) % 10)*3))
#define GPIO_OUT(g) *(gpio_base_map_addr + ((g)/10)) |=  (1 << (((g) % 10)*3))
```

There is quite of bit going on here… Many strange symbols, cryptic bitwise logic, and a whole lot of parentheses makes the macros look obfuscated. If you feel confused, just remember that these macros are simply bipartite and it will become easier to understand. To recap these two parts of the macros, we first build a register memory address based on the given GPIO pin *g* and then perform a bitwise operation

on said memory address to make the three bits either 000 or 001. I'll start making my way through these macros by highlighting the part where the register memory address is built based upon the given pin *g*.

```
#define GPIO_INP(g) *(gpio_base_map_addr + ((g)/10)) &= ~(7 << (((g) % 10)*3))
#define GPIO_OUT(g) *(gpio_base_map_addr + ((g)/10)) |=  (1 << (((g) % 10)*3))
```

`gpio_base_map_addr` is the function return of mmap(). This is the pointer the points to the first memory address of the mapped area of virtual memory represented as an unsigned integer (32 bits on the Raspberry Pi). For our purpose of understanding, let us substitute 0x3F200000/0x7E200000 in for `gpio_base_map_addr`. This will help in explaining why we are adding g/10 to this address in the above macros. Note that this is not the address that `gpio_base_map_addr` points to.

From our previous discussion, recall that each of the GPFSEL registers are organized per 10 pins. Pins 0-9 are assigned to GPFSEL0 (0x3F200000/0x7E200000), pins 10-19 are assigned to GPFSEL1 (0x3F200004/0x7E200004), and so on and so forth. This means that we want to add an offset to the base address only when we are dealing with pin 10-19, 20-29, etc. but not for 0-9. Adding *g*/10 satisfies this offset requirement as the result of divisions on integers in C does not carry what's behind the comma (i.e. the remainder). That means to say that for g = 4, g/10 = 0 and for *g* = 26, g/10 = 2.

Base Address	*g*	*g*/10	New Address	Register
0x3F200000/0x7E200000	0-9	0	0x3F200000/0x7E200000	GPFSEL0
	10-19	1	0x3F200004/0x7E200004	GPFSEL1
	20-29	2	0x3F200008/0x7E200008	GPFSEL2
	30-39	3	0x3F20000C/0x7E20000C	GPFSEL3
	40-49	4	0x3F200010/0x7E200010	GPFSEL4
	50-59	5	0x3F200014/0x7E200014	GPFSEL5

One question you might have is why we are only adding g/10 to the base address given that physical addresses are in increments of 4. It would make more sense to add (g/10)*4 to reflect this 0x04 interval separating adjacent addresses, but recall that `gpio_base_map_addr` is a pointer to an unsigned integer representation of a memory address. An unsigned integer is 32 bits (4 bytes) in length meaning that each memory address takes up 4 bytes in memory. Adding an integer to a pointer, also 32 bits in length, results in a new memory address that is exactly 4 bytes time the value of the integer offset from the original one. Pointer arithmetic is designed in such a way that you always end up at the starting byte of a memory address and not in the middle of it. This is automatically handled for us, so we don't have to worry about obtaining memory addresses that are 4 bytes away from its neighbor. A clear way to write pointer arithmetic involving integers:

```
new_address = base_address + sizeof(int)*int
```

Now that we have the correct memory address selected, we next need to perform a bitwise operation to said address. Remember from out previous discussion that each pin gets three bits, pin 0/10/20/30/40 is assigned bits 2-0, pin 9/29/39/49/59 is assigned bits 29-27, setting these bits to 000 means input, and setting these bits to 001 mean output. The bitwise operation that will need to perform to set all bits to

000 will be a bitwise AND and assignment (&=) while setting bits to 001 requires a bitwise OR and assignment (|=). Let's first start by highlighting these two operations in our macros

```
#define GPIO_INP(g) *(gpio_base_map_addr + ((g)/10)) &= ~(7 << (((g) % 10)*3))
#define GPIO_OUT(g) *(gpio_base_map_addr + ((g)/10)) |=  (1 << (((g) % 10)*3))
```

Focusing first on similarities, each macro seems to have at least one thing in common: ((g) % 10)*3). This expression is specifying the first bit location of the input GPIO pin g. First, calculate the remainder after division of *g* and 10 by using the modulo operator (%) then multiply by three to get the starting bit value. Remained after the division of 10 and its multiplication by 3 reflects that the GPIO pins are organized per 10 pins and are each given 3 bits. Examples of this calculation are tabulated below.

GPIO Pin	(g % 10)	(g % 10)*3	Starting Bit Listed in Data Sheet
0	0	0	0
4	4	12	12
13	3	9	9
26	6	18	18

GPIO_INP function, after determining the starting bit location for the input GPIO pin *g*, does some bitwise logic in order to make the target 3 bits 000. The macro accomplishes this by bitshifting, string inverting, and performing an AND and assignment operation with the GPSEL register represented by *(gpio_base_map_addr + ((g)/10)). The process can be thought of in these steps.

1. Bitshift binary 111 (7 in decimal) over (g % 10)*3 places to the left to get 3 sequential bits starting at (g % 10)*3 equal to 111.

   ```
   (7 << (((g) % 10)*3))
   ```

2. String Invert to invert all bits in constructed 32-bit string.

   ```
   ~(7 << (((g) % 10)*3))
   ```

3. Perform an AND and assignment with the GFSEL register and constructed 32-bit string.

   ```
   *(gpio_base_map_addr + ((g)/10)) &= ~(7 << (((g) % 10)*3))
   ```

Let's do an example to better understand this process. Say we want to use GPIO 4 (g = 4) as an input.

- Inside the GPFSEL register, the bits concerning GPIO 4 start from bit

  ```
  (4 % 10)*3 = 12
  ```

- If we bitshift binary 111 (decimal 7) 12 positions to the left, we get this resulting implied 32-bit string

  ```
  (7 << 12) = 00000000000000000111000000000000
  ```

- Inverting this 32-bit string results in

  ```
  ~(7 << 12) = 11111111111111111000111111111111
  ```

- We now AND and assignment with our 32-bit string and GPFSEL register to set 111 starting at bit position 12. “X” here refers to the unknown bit values of the register.

```
11111111111111111000111111111111
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
&= --------------------------------
XXXXXXXXXXXXXXXXX000XXXXXXXXXXXX
```

And there we go, we successfully set a GPIO pin as an input. Setting the pin as an output is slightly different as we are now concerned with setting 3 bits to 001. The process of `GPIO_OUT` can be broken down into these steps.

1. Bitshift binary 1 (1 in decimal) over `(g % 10)*3` places to the left.

```
(1 << (((g) % 10)*3))
```

2. Perform an OR and assignment with the GFSEL register and constructed 32-bit string.

```
*(gpio_base_map_addr + ((g)/10)) |= (7 << (((g) % 10)*3))
```

Let’s do an example to better understand this process. Say we want to use GPIO 4 (`g = 4`) as an output.

- Inside the GPFSEL register, the bits concerning GPIO 4 start from bit

```
(4 % 10)*3 = 12
```

- If we bitshift binary 1 (decimal 1) 12 positions to the left, we get this resulting implied 32-bit string

```
(1 << 12) = 00000000000000001000000000000
```

- We now OR and assign our 32-bit string with the GPFSEL register to set 001 starting at bit position 12. “X” here refers to the unknown bit values of the register.

```
00000000000000001000000000000
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
|= --------------------------------
XXXXXXXXXXXXXXXXXXX1XXXXXXXXXXXX
```

Did we succeed in setting 001 starting at bit position 12? Can you see why we didn’t? I’ve left bits 13 and 14 unknown, and as you recall, these bits need to be 0 to set a GPIO as an output. This is a consequence of using the OR operation to set bit 12 to 1: we cannot know for certain what the final bit values will be if we are OR’ing a 0. For this reason, we always use `GPIO_INP` before attempting `GPIO_OUT` to ensure that bits 13 and 14 are set to 0; otherwise, there is a chance that our GPIO will not be set as an output.

We have now successfully set a GPIO as an input or output using our own macros `GPIO_INP` and `GPIO_OUT`. Thankfully, the remaining macros we will develop are going to be easier to understand and walk through.

GPIO Set & Clear

The next two macros to discuss are GPIO_SET and GPIO_CLR. These macros will be using the GPSET and GPCLR registers respectively. GPSET registers are used to set a GPIO pin to high while GPCLR registers set a GPIO pin to low. These operations will only take immediate effect if the pin is set as an output. However, if the pin is subsequently defined as an output then the pin will be set according to the last set/clear operation. Like we did previously, let's first define the macros and explain later.

```
#define GPIO_SET(g)   *(gpio_base_map_addr + 7  + ((g)/31))) = (1 << g)
#define GPIO_CLR(g) *(gpio_base_map_addr + 10 + ((g)/31))) = (1 << g)
```

The above macros are composed of two parts: setting the register memory address and it equal to a 32-bit string where the bit a position g is equal to 1. Dissimilar to the previous macros we've defined is that register address is the base plus a constant offset and input pin g divided by 31. The constants 7 and 10 for GPIO_SET and GPIO_CLR respectively are gotten right off of the manual and reflect that addresses of GPSET0 and GPCLR0 registers. (g)/31 is included to allow GPSET1/GPCLR1 to be addressed too for when pins 32 to 53 are to be set/cleared; however, given that the 3b+ only has 26 pins, including (g)/31 is not necessary so it can be left off if desired.

Base Address	Constant	g	(g)/31	New Address	Register
0x3F200000/0x7E200000	7	0-31	0	0x3F20001C/0x7E20001C	GPSET0
	10			0x3F200028/0x7E200028	GPCLR0
	7	32-53	1	0x3F200020/0x7E200020	GPSET1
	10			0x3F20002C/0x7E20002C	GPCLR1

The second part of each macro is the same. We are setting the register equal to a 32-bit string created by bitshifting binary 1 (decimal 1) over g places to the left with our selected register. Recall that each bit in GPSETn and GPCLRn is associated with one GPIO pin such that bit 0 is GPIO 0 pin and bit 31 is GPIO 31 pin. All we have to do in this case is create a 32-bit string where the bit a position g is 1 and assign it to the register. To fully understand, let's do an ambiguous example with pin 4 such that either GPIO_SET or GPIO_CLR is being called.

- Inside the GPSET0/GPCLR0 register, the bit we are concerned with is bit 4.

$$g = 4$$

- If we bitshift 1 (decimal 1) over 4 places to the left, we end up with this implied 32-bit string

$$(1 >> 4) = 00000000000000000000000000010000$$

- We now assignment our 32-bit string to the GPSET/GPCLR register setting 1 at bit position 4.

$$\begin{array}{l} 00000000000000000000000000010000 \\ \text{XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX} \\ = \text{----------} \\ 00000000000000000000000000010000 \end{array}$$

Remember that setting a 0 in either register causes no clearing or setting of pins. Only setting a bit position equal to 1 will cause a change of state of a pin pin associated with that bit position. This is the reason why we are only assigning the register to our 32-bit string and not performing any bitwise logic to preserve memory in register like we did for GPIO_INP and GPIO_OUT.

GPIO Read

Last GPIO function to cover reads the current value—high or low—of a pin. Unlike the previous two macros, we are not looking to write values to a register but rather read from one. GPLEV0 and GPLEV1 registers return the actual value of a GPIO pin. Just like the set and clear registers, each bit in the register is associated with one pin meaning that we are only again concerned with one bit. Our macro GPIO_LEV will output the memory address of a GPLEV register bitmasked with a 32-bit string where the bit at position g is 1. This value will either be 0, indicating the pin is low, or (g << 1), indicating the pin is high.

```
#define GPIO_LEV(g) (*(gpio_base_map_addr + 13 + ((g)/31))) & (1 << g))
```

The constant 13 is the offset from gpio_base_map_addr to GPLEV0 and (g)/31 is included to allow GPLEV1 to be addressed to in the case pins 31 to 53 are read. Given that our Pi only has 26 GPIO pins, including (g)/31 is not necessary.

Base Address	Constant	g	(g)/31	New Address	Register
0x3F200000/0x7E200000	7	0-31	0	0x3F200034/0x7E200034	GPLEV0
		32-53	1	0x3F200038/0x7E200038	GPLEV1

Unlike all of the macros we previously developed, we are only performing bitwise AND operation and not AND and assignment with the GPLEV register and our 32-bit string. This reflects that we are not writing data to the register but just reading from it to determine whether a pin is high or low. We are specifically reading the unsigned integer at the GPLEVn memory address after bitmasking (1 << g) resulting in either 0 or (1 << g). The reason for this output is evident in the following example where we want to read the value of GPIO pin 4.

- Inside the GPLEV0 register, the bit we are concerned with is bit 4.

  g = 4

- If we bitshift 1 (decimal 1) over 4 places to the left, we end up with this implied 32-bit string

  (1 >> 4) = 00000000000000000000000000**1**0000

- We now AND with our 32-bit string and GPLEV register to return an output integer where bit 4 equals 1 and all other bits equal 0. “X” here refers to the unknown bit values in the register.
  - Assuming GPIO 4 is high:

    00000000000000000000000000**1**0000

    XXXXXXXXXXXXXXXXXXXXXXXXXXX1XXXX

    & ----------

    00000000000000000000000000**1**0000

- Assuming GPIO 4 is low:

```
  00000000000000000000000000010000
  XXXXXXXXXXXXXXXXXXXXXXXXXXX0XXXX
& --------------------------------
  00000000000000000000000000000000
```

When GPIO 4 is high in the above example, the return integer will be equal to decimal 16 (or its binary equivalent to the string produced by bitshifting 1 by 4 places to the left, i.e. (1 << 4)). This is the reason that high GPIO pins will return with an integer equal to its binary equivalent (1 << g) and 0 when low.

Led Blink Test

Let's now apply our knowledge of GPIOs to create a simple program that will blink an LED off and on. This program will set GPIO 11 high and then low once a second for a total of 10 iterations. At the end of each iteration, we will write the value of the pin out so that we can confirm that we successfully set or cleared the pin supplemental to the visual indication of the LED. The procedure of our program will the following:

1. Above our main() program:
    a. Define the address constants as macros
    b. Define the GPIO function macros
2. In our main() program
    a. Open /dev/mem/ for reading and writing
    b. Memory map GPIO physical addresses into virtual memory using mmap()
    c. Close /dev/mem/
    d. Use our GPIO macros GPIO_INP(g) and GPIO_OUT(g) to set GPIO 11 as input and then output in that order
    e. In a for loop for 10 iterations
        i. Use our GPIO macro GPIO_SET(g) to set GPIO 11
        ii. Use our GPIO macro GPIO_LEV(g) to get current value of GPIO 11 and then print to the screen
        iii. Wait 1 second
        iv. Use our GPIO macro GPIO_CLR(g) to clear GPIO 11
        v. Use our GPIO macro GPIO_LEV(g) to get current value of GPIO 11 and then print to the screen
        vi. Wait 1 second
    f. Unmap memory map using munmap()

```
// How to access GPIO registers from C-code on the Raspberry-Pi using only
// C standard and POSIX libraries

// Include C standard libraries:
#include <stdio.h>    // C Standard I/O libary
#include <stdlib.h>   // C Standard library
#include <fcntl.h>    // C Standard file control library
#include <time.h>     // C Standard get and manipulate time library

// Include C POSIX libraries:
#include <sys/mman.h> // Memory management library
#include <unistd.h>   // Symbolic constants and types library

// Define constants:
#define BCM2807_PERI_BASE 0x3F000000 // Base physical address
#define GPIO_BASE         (BCM2807_PERI_BASE + 0x200000) // Base plus offset
#define PAGE_SIZE         4096 // Memory page size in kilo bytes

// Declare GPIO base mapped memory address:
// (Memory address returned my mmap)
volatile unsigned *gpio; // Volatile as address may change during runtime

// GPIO function macros:
#define GPIO_INP(g) *(gpio + ((g)/10)) &= ~(7 << (((g) % 10)*3)) // Input
#define GPIO_OUT(g) *(gpio + ((g)/10)) |=  (1 << (((g) % 10)*3)) // Output
#define GPIO_SET(g) *(gpio + 7)  = (1 << g)
#define GPIO_CLR(g) *(gpio + 10) = (1 << g)
#define GPIO_LEV(g) (*(gpio + 13) & (1 << g))

// LED Blink Test
int main() {
    // Define variables:
    int mem_fd, i;
    void* gpio_base_map_addr;

    // Open /dev/mem for reading and writing
    mem_fd = open("/dev/mem", O_RDWR|O_SYNC);

    // Check success:
    if (mem_fd == -1) {
        // Print:
        printf("/mem/dev/ device could not be opened!");

        // Exit:
        return -1;
    }
```

```
 // Memory map GPIOs into virtual memory:
 gpio_base_map_addr = mmap(
     NULL,                 // Any adddress in our space will do
     PAGE_SIZE,            // Map length
     PROT_READ|PROT_WRITE, // Enable reading & writting to mapped memory
     MAP_SHARED,           // Shared with other processes
     mem_fd,               // File to map
     GPIO_BASE             // Offset to GPIO peripheral
);

 // Check success:
 if (gpio_base_map_addr == MAP_FAILED) {
     // Print:
     printf("Memory map failed!\n");

     // Exit:
     return -1;
 }

 // Set gpio_base_map_addr to gpio by type casting:
 gpio = (volatile unsigned*) gpio_base_map_addr;

 // Set GPIO to input than output:
 // (Order is important here)
 GPIO_INP(11);
 GPIO_OUT(11);

 // Blink LED:
 for (i = 0; i < 10; i++){
     // Set GPIO:
     GPIO_SET(11);

     // Get value of pin and print:
     printf("Current value of GPIO pin 11: %d\n",GPIO_LEV(11));

     // Wait one second:
     sleep(1);

     // Clear GPIO pin:
     GPIO_CLR(11);

     // Get value of pin and print:
     printf("Current value of GPIO pin 11: %d\n",GPIO_LEV(11));

     // Wait one second:
     sleep(1);
 }

 // Unmap:
 munmap(gpio_base_map_addr,PAGE_SIZE);

 // Exit:
 return 0;
```

Additional GPIO Pin Functions

Additional GPIO pin functions available on the BCM-2837 include event detection and pull-up/down clock. These are summarized in the register view table below.

Address	Field Name	Description	Size	Read/Write
0x3F200040/0x7E200040	GPEDS0	GPIO Pin Event Detect Status 0	32	R/W
0x3F200044/0x7E20004	GPEDS1	GPIO Pin Event Detect Status 1	32	R/W
0x3F20004C/0x7E20004C	GPREN0	GPIO Pin Rising Edge Detect Enable 0	32	R/W
0x3F200050/0x7E200050	PGREN1	GPIO Pin Rising Edge Detect Enable 1	32	R/W
0x3F200058/0x7E200058	GPFEN0	GPIO Pin Falling Edge Detect Enable 0	32	R/W
0x3F200058/0x7E200058	GPFEN1	GPIO Pin Falling Edge Detect Enable 1	32	R/W
0x3F200064/0x7E200064	GPHEN0	GPIO Pin High Detect Enable 0	32	R/W
0x3F200064/0x7E200064	GPHEN1	GPIO Pin High Detect Enable 1	32	R/W
0x3F200070/0x7E200070	GPLEN0	GPIO Pin Low Detect Enable 0	32	R/W
0x3F200070/0x7E200070	GPLEN1	GPIO Pin Low Detect Enable 1	32	R/W

GPIO Event Detect Status

GPIO event detection is used to record level and edge event on the 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 GPRENn/GPFENn.
2. Level is detected that matches the type of level programmed in the high/low level detect enable registers GPHENn/GPLENn.

Each bit in the event detect status registers is associated with one GPIO pin. Bit 0 in GPEDS0 is GPIO 0 pin and bit 31 is GPIO 32 pin; bit 0 in GPEDS1 is GPIO pin 33 and bit 21 is GPIO 54. Recording whether an event has occurred will be a function of reading the current value of the relevant bit within the appropriate GPEDSn register. A “1” indicates that an event has occurred while “0” says that no event was detected. To reset the event detection and record a future event, the relevant bit is cleared by writing a “1” to it.

Detecting an event and resetting after one has occurred uses two functions that we have already defined: `GPIO_SET/GPIO_CLR` and `GPIO_LEV`. The only change we have to make to these two functions is the address the 32-bit word is operated to.

```
#define GPIO_EVT_DET(g) *(gpio_base_map_addr + 16 + ((g)/31)) = (1 << g)
#define GPIO_EVT_CLR(g) *(gpio_base_map_addr + 16 + ((g)/31)) & (1 << g)
```

`gpio_base_map_addr + 16` results in an address of 0x3F200040/0x7E200040, the address of GPEDS0. Note that including (g)/31 is for completeness only as the Raspberry Pi has 26 GPIO pins.

Falling and Rising Edge Enable

The rising and falling edge detect enable registers define the pins for which an edge transitions sets a bit in the event detect status registers. When the relevant bits are set in both the GPRENn and GPFENn registers, any transition (1 to 0 and 0 to 1) will set a bit in the GPEDSn registers. Note that the rising and falling edge registers use synchronous edge detection meaning the input signal is sampled using the system clock. The sampled signal is searched for the pattern "001" or "100" depending on the event.

Each bit in the event detect status registers is associated with one GPIO pin. Bit 0 in GPRED0/GPFED0 is GPIO 0 pin and bit 31 is GPIO 32 pin; bit 0 in GPRED1/GPFED1 is GPIO pin 33 and bit 21 is GPIO 54. Setting the relevant bit to "1" in either register will enable event detection while clearing the bit disables it.

```
#define GPIO_RED_DIS(g) *(gpio_base_map_addr + 19 + ((g)/31)) &= ~(1 << g)
#define GPIO_RED_ENA(g) *(gpio_base_map_addr + 19 + ((g)/31)) |=  (1 << g)

#define GPIO_FED_DIS(g) *(gpio_base_map_addr + 22 + ((g)/31)) &= ~(1 << g)
#define GPIO_FED_ENA(g) *(gpio_base_map_addr + 22 + ((g)/31)) |= (1 << g)
```

The bitwise logic here ensures that only the bit we are interested in changes state. For disabling event detection, we are writing a 32-bit word where only the bit in position g is set to "0". The opposite is true for enabling event detection: the bit at position g in our 32-bit word is the only bit set to "1". Bitwise Or'ing and And'ing is used for enabling and disabling respectively to ensure that no other bits changes its value.

High and Low Enable

The high and low detect enable registers define the pins for which a high or low level sets a bit in the event detect status registers. When the relevant bits are set in both the GPHENn and GPLENn registers, any level (0 and 1) will set a bit in the GPEDSn registers. If the pin is still high or low if event detection is enabled, an attempt to clear the status bit in GPEDSn while result in the status bit to be remain set.

Each bit in the event detect status registers is associated with one GPIO pin. Bit 0 in GPPHEN0/GPLEN0 is GPIO 0 pin and bit 31 is GPIO 32 pin; bit 0 in GPHEN1/GPHEN1 is GPIO pin 33 and bit 21 is GPIO 54. Setting the relevant bit to "1" in either register will enable event detection while clearing the bit disables it.

```
#define GPIO_HED_DIS(g) *(gpio_base_map_addr + 25 + ((g)/31)) &= ~(1 << g)
#define GPIO_HED_ENA(g) *(gpio_base_map_addr + 25 + ((g)/31)) |= (1 << g)

#define GPIO_LED_DIS(g) *(gpio_base_map_addr + 28 + ((g)/31)) &= ~(1 << g)
#define GPIO_LED_ENA(g) *(gpio_base_map_addr + 28 + ((g)/31)) |= (1 << g)
```

Important Note

You must add “dtoverlay=gpio-no-irq” in /boot/config.txt to use `GPIO_EVT_DET` and `GPIO_EVT_CLR` without causing problems on the Pi. The kernel assumes that GPIO event detection will only be enabled in order to generate interrupts managed by the kernel (or user). Enabling event detection without the kernel driver to monitor for said event kills the interrupt system; there is nothing to handle the interrupt and clear the bit within GPEDSn. The machine will continue to run after this occurs but access to I/O devices (USB, network, etc…) stops. If you do not add this overlay into config.txt, then you will see problems whenever you enable event detection and that event occurs.

Adding the boot overlay (also know as a “device tree overlay”) gpio-no-irq to config.txt will reconfigure the kernel such that it does not use GPIOs as interrupt request lines. These device tree overlays are used on the Pi to support many hardware configurations while booting to the same kernel. Once you add the line to the configuration file and reboot, you will then have unrestricted access to the event registers. Just remember that you cannot use GPIOs for interrupts now. If you do not want to rely upon an overlay, use GPIO interrupts instead. There are many implementations in kernel or userspace that can be done to achieve this.

Note that GPIO interrupts on a non-real-time Pi typically see a 50 - 100µs latency from event to a polling userspace process. Preempt-RT, a Linux RTOS patch, will improve that latency by about 50%.