CHAPTER 15

I^{2}C Bus

The I^{2}C bus, also known as the two-wire interface (TWI), was developed by Philips circa 1982 to allow communication with lower-speed peripherals.[17] It was also economical because it only required two wires (excluding ground and power). Since then, other standards have been devised, building upon this framework, such as the SMBus. However, the original I^{2}C bus remains popular as a simple, cost-effective way to connect peripherals.

I^{2}C Overview

Figure 15-1 shows the I^{2}C bus in the Raspberry Pi context. The Raspberry Pi provides the bus using the BCM2835 device as the bus master. Notice that the Pi also provides the external pull-up resistors R_1 and R_2, shown inside the dotted lines. Table 15-1 lists the two I2C bus lines that are provided on the header strip.

Table 15-1. *I2C Bus Connections*

Connection	GPIO	Description
P1-03	GPIO-2	SDA1 (serial bus data)
P1-05	GPIO-3	SCL1 (serial bus clock)

W. Gay, *Advanced Raspberry Pi*, https://doi.org/10.1007/978-1-4842-3948-3_15

The design of the I2C bus allows multiple peripherals to attach to the SDA and the SCL lines. Each slave peripheral has its own unique 7-bit address. For example, the MCP23017 GPIO extender peripheral might be configured with the address of 0x20. Each peripheral is referenced by the master by using this address. Nonaddressed peripherals are expected to remain quiet.

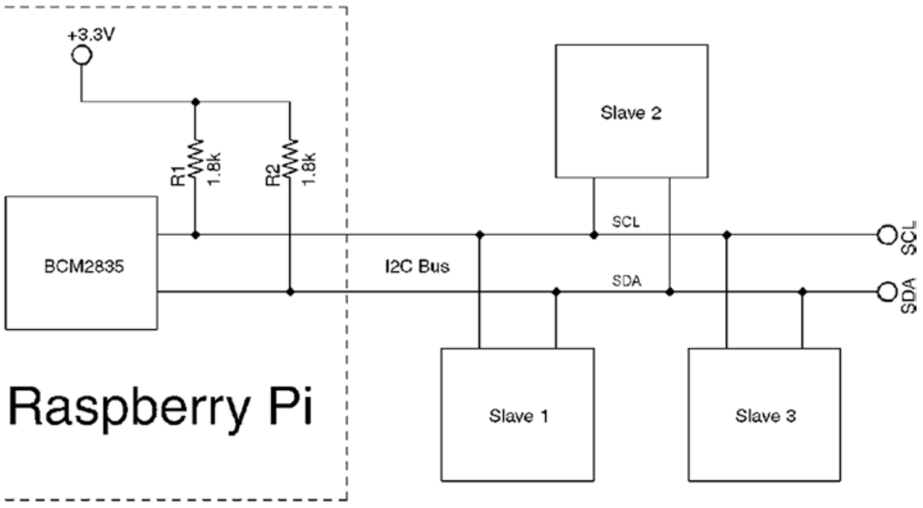

Figure 15-1. *The I^2C bus on the Raspberry Pi*

SDA and SCL

Both masters and slaves take turns at "grabbing the bus" at various times. Master and slave use open-drain transistors to drive the bus lines. It is because all participants are using open-drain drivers that pull-up resistors must be used (provided by the Pi). Otherwise, the data and clock lines would float between handoffs.

The open-drain driver design allows all participants to drive the bus lines—just not at the same time. Slaves, for example, turn off their line drivers, allowing the master to drive the signal lines. The slaves just listen, until the master calls them by address. When the slave is required to answer, the slave will then assert its driver, grabbing the line. It is assumed by the slave that the master has already released the bus at this point. When the slave completes its own transmission, it releases the bus, allowing the master to resume.

The idle state for both lines is high. The high state for the Raspberry Pi is +3.3 V. Other systems may use +5 V signaling. When shopping for I^{2}C peripherals, choose ones that will operate at the 3.3 V level. Sometimes 5 V peripherals can be used with careful signal planning or using adapters.

Bus Signaling

The start and stop bits are special in the I^{2}C protocol. The start bit is illustrated in Figure 15-2. Notice the SDA line transition from high to low, while the clock remains in the high (idle) state. The clock will follow by going low after 1/2 bit time following the SDA transition. This special signal combination informs all connected devices to "listen up," since the next piece of information transmitted will be the device address.

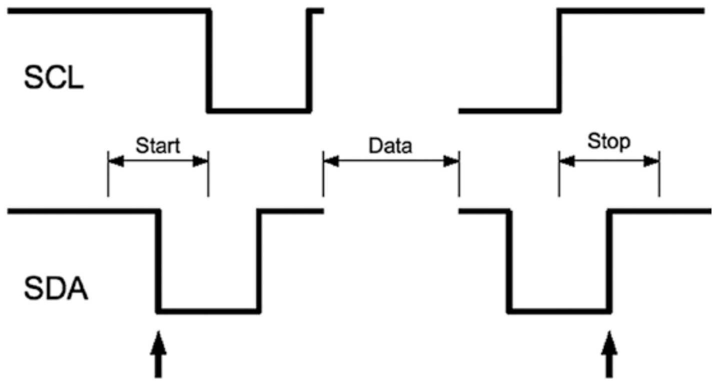

Figure 15-2. *I2C start/stop signaling*

The stop bit is also special in that it allows slave devices to know whether more information is coming. When the SDA line transitions from low to high midway through a bit cell, it is interpreted as a *stop bit*. The stop bit signals the end of the message.

There is also the concept of a *repeated start*, often labeled in diagrams as *SR*. This signal is electrically identical to the start bit, except that it occurs within a message in place of a stop bit. This signals to the peripheral that more data is being sent or required as part of another message.

Data Bits

Data bit timings are approximately as shown in Figure 15-3. The SDA line is expected to stabilize high or low according to the data bit being sent, prior to the SCL line going high. The receiver clocks in the data on the falling edge of SCL, and the process repeats for the next data bit. Note that most significant bits are transmitted first (network order, or big endian).

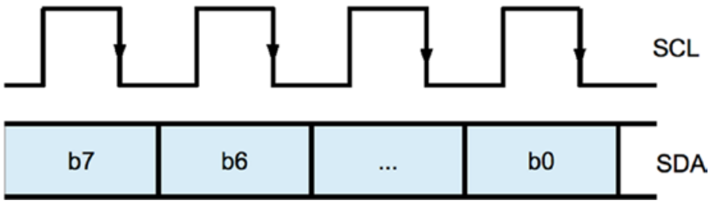

Figure 15-3. *I2C Data bit transmission*

Message Formats

Figure 15-4 displays two example I2C messages that can be used with the MCP23017 chip. The simplest message is the write register request.

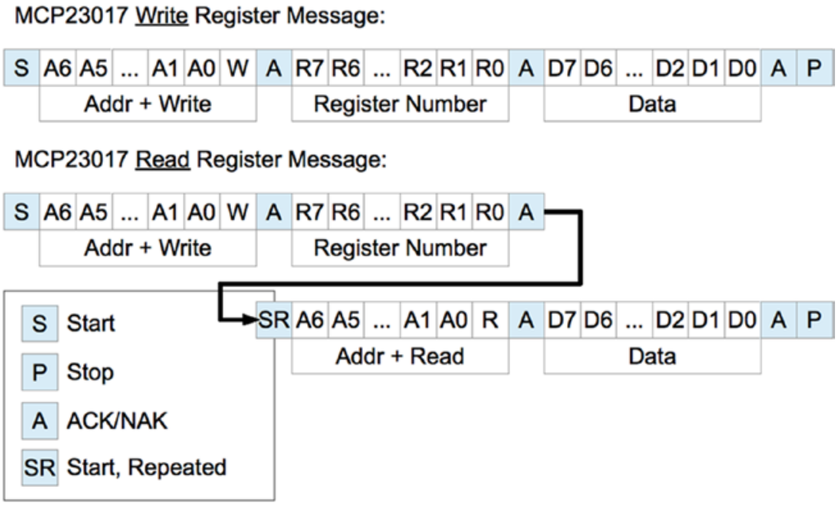
MCP23017 Write Register Message:
S A6 A5 ... A1 A0 W A R7 R6 ... R2 R1 R0 A D7 D6 ... D2 D1 D0 A P
Addr + Write
Register Number
Data
MCP23017 Read Register Message:
S A6 A5 ... A1 A0 W A R7 R6 ... R2 R1 R0 A
Addr + Write
Register Number
SR A6 A5 ... A1 A0 R A D7 D6 ... D2 D1 D0 A P
Addr + Read
Data
S Start
P Stop
A ACK/NAK
SR Start, Repeated

Figure 15-4. *Example I2C messages*

The diagram shows each message starting with the S (start) bit and ending with a P (stop) bit. After the start bit, each message begins with a byte containing the 7-bit peripheral address and a read/write bit. Every peripheral must read this byte in order to determine whether the message is addressed to it.

The addressed peripheral is expected to return an ACK/NAK bit after the address is sent. If the peripheral fails to respond for any reason, the line will go high due to the pull-up resistor, indicating a NAK. The master, upon seeing a NAK, will send a stop bit and terminate the transmission.

When the addressed peripheral ACKs the address byte, the master then continues to write when the request is a write. The first example shows the MCP23017 8-bit register number being written next. This indicates which of the peripheral's registers is to be written to. The peripheral will then ACK the register number, allowing the master to follow with the data byte to be written, into the selected register. This too must be ACKed. If the master has no more data to send, the P (stop) bit is sent to end the transmission.

The second example in Figure 15-4 shows how a message may be composed of both write and read messages. The initial sequence looks like the write, but this only writes a register number into the peripheral. Once the register number is ACKed, the master then sends an SR (start, repeated) bit. This tells the peripheral that no more write data is arriving and to expect a peripheral address to follow. Since the address transmitted specifies the *same* peripheral, the same peripheral responds with an ACK. This request is a read, so the peripheral continues to respond with 8 bits of the requested read data, with the master ACKing. The master terminates the message with a P (stop) to indicate that no more data is to be read.

Many peripherals will support an *auto-increment* register mode. This is a feature of the peripheral, however. Not all devices support this. Once a peripheral's register has been established by a write, successive reads or writes can occur in auto-increment mode, with the register being incremented with each byte transferred. This results in efficient transfers.

I^{2}C Bus Speed

Unlike the SPI bus, the I^{2}C bus operates at a fixed speed within Raspbian Linux. The SoC document claims I^{2}C operation up to 400 kHz, but the default is 100 kHz.

To use I^{2}C, you must enable it in your `/boot/config.txt` file by uncommenting it. You can also optionally specify the clock rate by specifying the `i2c_arm_baudrate` parameter. The following enables I^{2}C and sets the clock to 400 kHz:

```
dtparam=i2c_arm=on,i2c_arm_baudrate=400000
```

The default clock rate is equivalent to:

```
dtparam=i2c_arm=on,i2c_arm_baudrate=100000
```

Save the config.txt file and reboot. You can confirm that the clock rate was accepted as follows:

```
# xxd -g4 /sys/class/i2c-adapter/i2c-1/of_node/clock-frequency
00000000: 00061a80                                    ....
# gdb
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
...
(gdb) p 0x00061a80
$1 = 400000
(gdb) quit
#
```

- The `xxd` command reports a group of 4 bytes (`-g4`) as 00061a80.
- The `gdb` command is used to print (`p` command) this value in decimal (don't forget to prefix the reported number with `0x` to indicate the value is hexadecimal).

I^{2}C Tools

Working with I^{2}C peripherals is made easier with some utilities. These may be preinstalled in your Raspbian Linux but are otherwise easily installed if necessary.

```
$ sudo apt–get install i2c–tools
```

The `i2c-tools` package includes the following utilities:

`i2cdetect`: Detects peripherals on the I2C line

`i2cdump`: Dumps values from an I2C peripheral `i2cset`: Sets I2C registers and values

`i2cget`: Gets I2C registers and values

Each of these utilities has a man page available for additional information.

MCP23017

The MCP23S17 is the I2C chip that provides 16 expansion GPIO ports. At startup, the pins default to inputs, but can be configured as outputs like the native Pi GPIO ports. The MCP23S17 is the companion chip for SPI bus.

The chip allows eight different I2C addresses to be wire configured. Like the native Pi GPIOs, the ports can be configured active high or low. The chip operates from a supply voltage of 1.8 to 5.5 V, making it perfect for Pi 3.3 V operation.

The output mode GPIO can sink up to 8 mA of current and source 3 mA. This should be taken into account when driving loads, even LEDs.

For input GPIOs, it has an interrupt capability, signaling an input change on the INTA (GPIOA0 to GPIOA7) or INTB pins (GPIOB0 to GPIOB7). The chip can be configured to report all changes on INTA, which is the way it will be used here. This is important for inputs because otherwise you would need to continuously poll the device.

Perhaps the best part of all is that a kernel driver is available for it. This makes it very convenient to use.

Driver Setup

The first thing that must be configured is that I^2C must be enabled, if you have not done it already. The `/boot/config.txt` file must have the following line uncommented:

```
dtparam=i2c_arm=on,i2c_arm_baudrate=100000
```

Next you must enable the driver in `config.txt`:

```
dtoverlay=mcp23017,gpiopin=4,addr=0x20
```

- Optional parameter `gpiopin=4` specifies that GPIO4 will be used for sensing interrupts in the chip. GPIO4 is the default.
- Optional parameter `addr=0x20` specifies the I^{2}C address for the MCP23017 chip. 0x20 is the default.

After editing these changes, reboot:

```
# sync
# /sbin/shutdown -r now
```

After the Pi boots back up, log in and check for suspicious error messages using the `dmesg` command. You can skip that if you're feeling lucky.

If all went well, you should see something like the following in the `/sys/class/gpio` directory:

```
# ls /sys/class/gpio
export  gpiochip0  gpiochip128  gpiochip496  unexport
```

If you used I2C address 0x20 and you have your MCP23017 wired up to the bus, you should see the subdirectory name `gpiochip496` (higher for other addresses). If you don't see the chip listed, then:

- Scrutinize the `dmesg` log for errors.
- Check the configuration and wiring.
- Make sure that the MCP23017 chip's $\overline{RESET}$ pin is wired to +3.3 V.

Wiring

The wiring used in this example is illustrated in Figure 15-5. A few things are worth noting about the circuit:

- Supply the MCP23017 chip from +3.3 V (not 5 V).
- No resistors are required for the bus since the Raspberry Pi already provides R_1 and R_2.
- Important! Wire the $\overline{RESET}$ line to +3.3 V. Otherwise random or complete failure will occur.
- Wiring the INTA line is not required if you only plan to use *output* mode GPIOs. However, the driver will consume the configured GPIO, whether used or not.

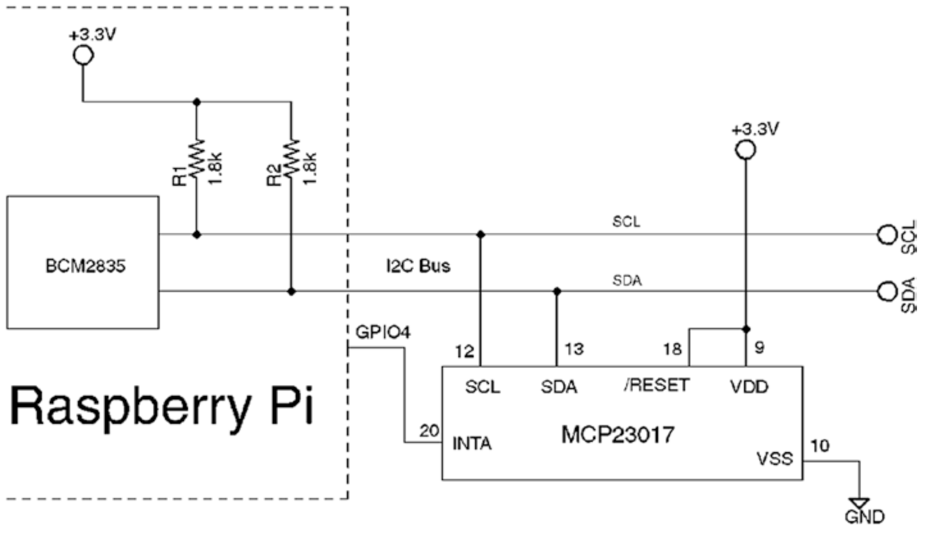

Figure 15-5. *Wiring for the MCP23017 to the Raspberry Pi*

When the $\overline{RESET}$ line is not wired to +3.3 V, the input to the chip will float. Sometimes the CMOS input will float high and sometimes low (causing a chip reset). I ran into this when I initially wired up the circuit. The worst part was that the driver and chip worked for a while but later developed problems.

The purpose of the INTA line (and GPIO4 in Figure 15-5) is to notify the Pi that an input GPIO port has changed state. This informs the mcp23017 driver to send an I^{2}C request to read the inputs. Without this notification, the driver would have to busy the I^{2}C bus with repeated read requests to see if there is new input.

Testing GPIO Output

With the circuit wired up and the configuration set and the system rebooted, you should see the driver report its presence in `/sys/class/gpio` as `gpiochip496`, if you used the I^{2}C address of 0x20.

In the same way that native GPIOs were accessed in Chapter 12, we can export this GPIO. But first we need to determine which GPIO number corresponds to each MCP23017 GPIO port. There are two pseudo files for this purpose:

1. gpiochip496/base lists the starting GPIO number for this device (496).
2. gpiochip496/ngpio lists how many GPIOs are supported (16).

The following shows an example discovery session:

```
# cd /sys/class/gpio
# ls
export   gpio503  gpiochip0  gpiochip128  gpiochip496  unexport
# ls gpiochip496
base  device  label  ngpio  power  subsystem  uevent
# cat gpiochip496/base
496
# cat gpiochip496/ngpio
16
#
```

This information permits the creation of the chart in Table 15-2.

***Table 15-2.** GPIO Associations for Gpiochip496 (I^2C Address 0x20)*

GPIO	Pin	MCP23017	GPIO	Pin	MCP23017
GPIO496	21	A0	GPIO504	1	B0
GPIO497	22	A1	GPIO505	2	B1
GPIO498	23	A2	GPIO506	3	B2
GPIO499	24	A3	GPIO507	4	B3
GPIO500	25	A4	GPIO508	5	B4
GPIO501	26	A5	GPIO509	6	B5
GPIO502	27	A6	GPIO510	7	B6
GPIO503	28	A7	GPIO511	8	B7

To use the MCP23017 GPIO A7 as an output, we do:

```
# pwd
/sys/class/gpio
# echo out >gpio503/direction
# cat gpio503/direction
out
# echo 1 >gpio503/value
# cat gpio503/value
1
```

If you have an LED wired to A7 in active high configuration, it should be lit. Otherwise measure it with your DMM and you should see +3.3 V on pin 28.

```
# echo 0 >gpio503/value
# cat gpio503/value
0
```

After the above, GPIO A7 should now go low.

Testing GPIO Input

With the GPIO A7 still configured as an output, configure MCP23017 GPIO A6 as an input:

```
# ls
export   gpio503  gpiochip0  gpiochip128  gpiochip496  unexport
# echo 502 >export
# ls
export
gpio502  gpio503  gpiochip0  gpiochip128  gpiochip496  unexport
# echo in >gpio502/direction
```

Place a jumper wire from A7 (pin 28) to A6 (pin 27). Now let's see if input A6 agrees with output A7:

```
# cat gpio502/value
0
# cat gpio503/value
0
# cat gpio502/value
0
# echo 1 >gpio503/value
# cat gpio502/value
1
```

As expected, as we changed A7, the input A6 followed.

Test GPIO Input Interrupt

Being able to read a GPIO alone is often not enough. We need to know *when* it has changed so that it can be read at that point in time. In Figure 15-5, the MCP23017 chip has its INTA pin wired to the Pi's GPIO4. The MCP23017 will activate that line whenever an unread change in inputs occurs, alerting the driver in the Pi. Only then does the driver need to read the chip's current input status.

To test that this is working, we'll reuse that evinput program to monitor gpio502 (GPIO input A6):

```
$ cd ~/RPi/evinput
$ ./evinput -g502 -b
```

Changing to the root terminal session, let's toggle A7 a couple of times:

```
# pwd
/sys/class/gpio
# ls
export
gpio502  gpio503  gpiochip0  gpiochip128  gpiochip496  unexport
# echo 1 >gpio503/value
# echo 0 >gpio503/value
# echo 1 >gpio503/value
# echo 0 >gpio503/value
```

Switch back to the evinput session, and see if we got any edges (the -b option monitors for both rising and falling edges):

```
$ ./evinput -g502 -b
Monitoring for GPIO input changes:

GPIO 502 changed: 1
GPIO 502 changed: 0
GPIO 502 changed: 1
GPIO 502 changed: 0
^C
```

Indeed, this confirms that the interrupt facility works. Note that we monitored GPIO502 (A6) rather than GPIO4. Only the driver needs to monitor GPIO4.

Limitations

The driver support for the MCP23017 provides a very convenient way to add sixteen GPIOs to your Raspberry Pi. As great as this is, here are a few points to consider:

- The extended GPIOs are not as fast as the native Pi GPIOs.
- You may need to do some homework to add more than one MCP23017 chip. While the bus supports up to eight uniquely addressed MCP23017 chips, the device driver might not. It may be possible with added nodes to the device tree.
- I/O performance is directly related to the I^{2}C clock rate.
- The GPIOs are accessed through the sysfs pseudo file system, further impacting performance.

The main thing to keep in mind is that all GPIO interaction occurs over the I^{2}C bus at the clock rate (100 kHz or 400 kHz). Each I/O may require several bytes of transfer because the MCP23017 has a large set of registers. Each byte transferred requires time. At the default of 100 kHz, a one-byte transfer takes:

$$\begin{aligned} t &= \frac{1}{100kHz} \times 8bits \\ &= 80\mu s \end{aligned}$$

To read one GPIO input register requires a start bit, three bytes of data, and a stop bit. This results in a minimum transaction time of 260 μs. That limits the number of GPIO reads to approximately 3,800 reads/s. This doesn’t account for sharing the bus with other devices.

In the end, the suitability depends upon the application. By shifting the highest rate GPIO transactions to the Pi’s native GPIOs and the slower I/Os to the extension GPIOs, you might find that the arrangement works well enough.

I^{2}C API

The bare-metal C language API for the I^{2}C bus transactions will be introduced in this section. Using this API you can program your own interface with another GPIO expander such as the PCF8574, for example. That chip provides eight additional GPIOs but is economical and +3.3 V friendly. It has only one configuration register making it easy to use directly.

Kernel Module Support

Access to the I2C bus is provided through the use of kernel modules. If you have enabled I2C in the config.txt as discussed earlier, you should be able to list the bus controller:

```
# i2cdetect -l
i2c-1 i2c       bcm2835 I2C adapter             I2C adapter
```

Access to the driver is provided by the following nodes:

```
# ls -l /dev/i2c*
crw-rw---- 1 root i2c 89, 1 Jul  7 16:23 /dev/i2c-1
```

Header Files

The following header files should be included in an I^{2}C program:

```
#include <sys/ioctl.h>
#include <linux/i2c.h>
#include <linux/i2c–dev.h>
```

open(2)

Working with I^{2}C devices is much like working with files. You open a file descriptor, do some I/O operations with it, and then close it. The one difference is that the `ioctl(2)` calls are used instead of the usual `read(2)` and `write(2)`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(const char *pathname,int flags,mode_t mode);
```

where

`pathname` is the name of the file/directory/driver that you need to open/create.

`flags` is the list of optional flags (use `O_RDWR` for reading and writing).

`mode` is the permission bits to create a file (omit argument, or supply zero when not creating).

returns `-1` (error code in `errno`) or open file descriptor >=0.

Error	Description
EACCES	Access to the file is not allowed.
EFAULT	The pathname points outside your accessible address space.
EMFILE	The process already has the maximum number of files open.
ENFILE	The system limit on the total number of open files has been reached.
ENOMEM	Insufficient kernel memory was available.

To work with the I^{2}C bus controller, your application must open the driver, made available at the device node:

```
int fd;

fd = open("/dev/i2c-1",O_RDWR);
if ( fd < 0 ) {
    perror("Opening /dev/i2c-1");
```

Note that the device node (`/dev/i2c-1`) is owned by root, so you'll need elevated privileges to open it or have your program use `setuid(2)`.

ioctl(2,I2C_FUNC)

In I^{2}C code, a check is normally performed to make sure that the driver has the right support. The `I2C_FUNC ioctl(2)` call allows the calling program to query the I^{2}C capabilities. The capability flags returned are documented in Table 15-3.

```
long funcs;
int rc;

rc = ioctl(fd,I2C_FUNCS,&funcs);
if ( rc < 0 ) {
    perror("ioctl(2,I2C_FUNCS)");
    abort();
}

/* Check that we have plain I2C support */
assert(funcs & I2C_FUNC_I2C);
```

***Table 15-3.** I2C_FUNC Bits*

Bit Mask	Description
I2C_FUNC_I2C	Plain I2C is supported (non SMBus)
I2C_FUNC_10BIT_ADDR	Supports 10-bit addresses
I2C_FUNC_PROTOCOL_MANGLING	*Supports:*
	I2C_M_IGNORE_NAK
	I2C_M_REV_DIR_ADDR
	I2C_M_NOSTART
	I2C_M_NO_RD_ACK

The `assert()` macro used to check that at least plain I2C support exists. Otherwise, the program aborts.

ioctl(2,I2C_RDWR)

While it is possible to use `ioctl(2,I2C_SLAVE)` and then use `read(2)` and `write(2)` calls, this tends not to be practical. Consequently, the use of the `ioctl(2,I2C_RDWR)` system call will be promoted instead. This system call allows considerable flexibility in carrying out complex I/O transactions.

The general API for any `ioctl(2)` call is as follows:

```
#include <sys/ioctl.h>

int ioctl(int fd,int request,argp);
```

where

`fd` is the open file descriptor.

`request` is the I/O command to perform.

`argp` is an argument related to the command (type varies according to `request`).

returns -1 (error code in errno), number of msgs completed (when request = I2C_RDWR).

Error	Description
EBADF	fd is not a valid descriptor.
EFAULT	argp references an inaccessible memory area.
EINVAL	request or argp is not valid.

When the request argument is provided as I2C_RDWR, the argp argument is a pointer to struct i2c_rdwr_ioctl_data. This structure points to a list of messages and indicates how many of them are involved.

```
struct i2c_rdwr_ioctl_data {
    struct i2c_msg   *msgs;   /* ptr to array of simple messages */
    int               nmsgs;   /* number of messages to exchange */
};
```

The individual I/O messages referenced by the preceding structure are described by struct i2c_msg:

```
struct i2c_msg {
    __u16     addr;   /* 7/10 bit slave address */
    __u16     flags;  /* Read/Write & options */
    __u16     len;    /* No. of bytes in buf */
    __u8     *buf;    /* Data buffer */
};
```

The members of this structure are as follows:

addr: Normally this is the 7-bit slave address, unless flag I2C_M_TEN and function

I2C_FUNC_10BIT_ADDR are used. Must be provided for each message.

flags: Valid flags are listed in Table 15-4. Flag I2C_M_RD indicates the operation is a read. Otherwise, a write operation is assumed when this flag is absent.

buf: The I/O buffer to use for reading/writing this message component.

len: The number of bytes to read/write in this message component.

Table 15-4. *I2C Capability Flags*

Flag	Description
I2C_M_TEN	10-bit slave address used
I2C_M_RD	Read into buffer
I2C_M_NOSTART	Suppress (Re)Start bit
I2C_M_REV_DIR_ADDR	Invert R/W bit
I2C_M_IGNORE_NAK	Treat NAK as ACK
I2C_M_NO_RD_ACK	Read will not have ACK
I2C_M_RECV_LEN	Buffer can hold 32 additional bytes

An actual ioctl(2,I2C_RDWR) call would be coded like the following. In this example, a MCP23017 *register* address of 0x15 is being written out to peripheral address 0x20, followed by a read of 1 byte:

```
int fd;
struct i2c_rdwr_ioctl_data msgset;
struct i2c_msg iomsgs[2];
static unsigned char reg_addr[] = {0x15};
unsigned char rbuf[1];
int rc;
```

```
iomsgs[0].addr   = 0x20;              /* MCP23017—A */
iomsgs[0].flags  = 0;                 /* Write operation. */
iomsgs[0].buf    = reg_addr;
iomsgs[0].len    = 1;

iomsgs[1].addr   = iomsgs[0].addr;  /* Same MCP23017-A */
iomsgs[1].flags  = I2C_M_RD;          /* Read operation */
iomsgs[1].buf    = rbuf;
iomsgs[1].len    = 1;

msgset.msgs      = iomsgs;
msgset.nmsgs     = 2;

rc = ioctl(fd,I2C_RDWR,&msgset);
if ( rc < 0 ) {
    perror("ioctl (2, I2C_RDWR)");
```

The example shown defines `iomsgs[0]` as a write of 1 byte, containing a register number. The entry `iomsgs[1]` describes a read of 1 byte from the peripheral. These two messages are performed in one `ioctl(2)` transaction. The flags member in `iomsgs[x]` determines whether the operation is a read (`I2C_M_RD`) or a write (0).

Note Don't confuse the peripheral's internal register number with the peripheral's I2C address.

Each of the `iomsgs[x].addr` members must contain a valid I^2C peripheral address. Each message can potentially address a different peripheral. The `ioctl(2)` will return an error with the first message failure. For this reason, you may not always want to combine multiple messages in one `ioctl(2)` call, especially when different devices are involved.

The returned value, when successful, is the number of `struct i2c_msg` messages successfully performed.

Summary

From this chapter you saw that adding sixteen GPIOs to your Pi can be realized with the addition of one chip and a little wiring. Considering the cost of add-on boards, this can save your project considerably. With the driver support for the MCP23017, using these extension GPIOs is just as simple as the native ports.

For the developer that wants to interact directly over I^2C with his devices, the C API for doing so was presented. Whether though the driver or through direct the C API, no Pi developer is left wanting for access to GPIO ports.