CHAPTER 14

Letting Go of the mbed Libraries

14.1 Introduction: How Much Do We Depend on the mbed API

The mbed library contains many useful functions, which allow us to write simple and effective code. This seems a good thing, but it is also sometimes limiting. What if we want to use a peripheral in a way not allowed by any of the functions? Therefore it is useful to understand how peripherals can be configured by direct access to the microcontroller's registers. In turn, this leads to a deeper insight into some aspects of how a microcontroller works. As a by-product, and because we will be working at the bit and byte level, this study develops further skills in C programming.

It's worth issuing a very clear health warning at this early stage—this chapter is technically demanding. It introduces some of the complexity of the LPC1768 microcontroller, which lies at the heart of the mbed, a complexity which the mbed designers rightly wish to keep from you. Your own curiosity, or ambition, or your professional needs, may however lead you to want to work at this deeper level.

Working with the complexity of this chapter may result in two opposing feelings. One is a sense of gratitude to the writers of the mbed libraries that they have saved you the difficulty of controlling the peripherals directly. At the opposite extreme, getting to grips with the chapter could also be a liberating experience—like throwing away the water wings after you've learned to swim. At the moment you probably think that you can't write any program unless you have the library functions at the ready. When you're thorough with this chapter you will realize that you're no longer dependent on the library; you use it when you want, and write your own routines when you want—the choice becomes yours!

This chapter may be read in sequence with all other chapters. Alternatively, it can be read in different sections as extensions of earlier chapters. We will refer to Ref. [4] of Chapter 2, the LPC1768 datasheet, and even more Ref. [5] of Chapter 2, its user manual. Because we are now working at microcontroller level, rather than mbed level, we will have to take more care about how the microcontroller pins connect with the mbed pins. Therefore, you may wish to have the mbed schematics, Ref. [3] of Chapter 2, ready.

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00014-1

14.2 Control Register Concepts

It's useful here to understand a little more about how the microcontroller CPU interacts with its peripherals. Each of these has one or more *system control registers* which act as the doorway between the CPU and the peripheral. To the CPU, these registers look just like memory locations. They can usually be written to and read from, and each has its own address within the memory map. The clever part is that each of the bits in the register is wired across to the peripheral. They might carry control information, sent by the CPU to the peripheral, or they might return status information from the peripheral. They might also provide an essential path for data. The general idea of this is illustrated in Fig. 14.1. The microcontroller peripherals also usually generate interrupts, for example, to flag when an ADC conversion is complete, or a new word has been received by a serial port.

Early in any program, the programmer must write to the control registers, to set up the peripherals to the configuration needed. This is called *initialization* and turns the control registers from a general purpose and nonfunctioning piece of hardware, to something that is useful for the project in hand. Using the mbed application programming interface, this task is undertaken in the mbed utilities; in this chapter, we move to doing this work

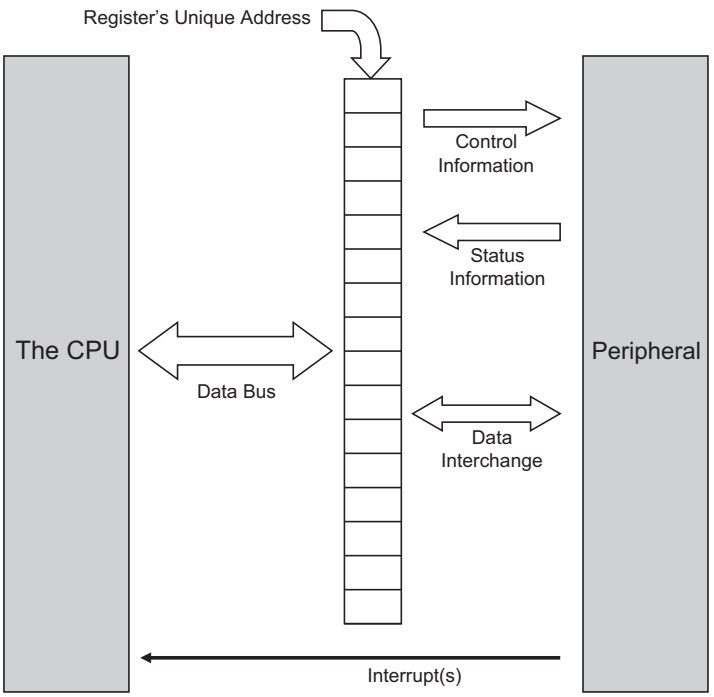

Figure 14.1
The principle of a control register.

ourselves. In all of this, an important question arises: what happens in that short period of time *after* power has been applied, but *before* the peripherals have been set up by the program? At this time, an embedded system is powered, but not under complete control. Fortunately, all makers of microcontrollers design in a *reset condition* of each control register. This generally puts a peripheral into predictable, and often inactive, state. We can look this up, and write the initialization code accordingly. On some occasions, the reset state may be the state that we need. We return to this in Chapter 15.

A central theme of this chapter is the exploration and use of some of the LPC1768 control registers. Although we draw information from the LPC1768 data, the chapter is self-contained, with all necessary data included. However, we show which table in the manual the data is taken from, so it's easy to cross-refer. We hope this gives you the confidence and ability to move on to explore and apply the other registers.

14.3 Digital Input/Output

14.3.1 LPC1768 Digital Input/Output Control Registers

The digital input/output (I/O) is a useful place to start our study of control registers, as this is simpler than any other peripherals on the LPC1768. This has its digital I/O arranged nominally as five 32-bit ports; yes, that implies a stunning 160 bits. Only parts of these are used however, for example Port 0 has 28 bits implemented, Port 2 has 14, and Ports 3 and 4 have only two each. In the end, among its 100 pins, the LPC1768 has around 70 general purpose I/O pins available. However, the mbed has only 40 accessible pins, so only a subset of the microcontroller pins actually appear on the mbed interconnect, and many of these are shared with other features.

As we saw in Chapter 3, it is possible to set each port pin as an input or as an output. Each port has a 32-bit register which controls the direction of each of its pins. These are called the **FIODIR** registers. To specify which port the register relates to, the port number is embedded within the register name, as shown in Table 14.1. For example, **FIO0DIR** is the direction register for port 0. Each bit in this register then controls the corresponding bit in the I/O port, for example, bit 0 in the direction register controls bit 0 in the port. If the bit in the direction register is set to 1, then that port pin is configured as an output; if the bit is set to 0, the pin is configured as an input.

It is sometimes more convenient not to work with the full 32-bit direction register, especially when we might just be thinking of one or two bits within the register. For this reason, it is also possible to access any of the bytes within the larger register, as single-byte registers. These registers have a number code at their end. For example, **FIO2DIR0** is byte 0 of the Port 2 direction register, also seen in Table 14.1. From the table you can see that the address of the whole word is shared by the address of the lowest byte.

Table 14.1: Example digital input/output control registers.

Register Name	Register Function	Register Address
FIO*n*DIR	Sets the data direction of each pin in Port *n*, where *n* takes value 0 to 4. A port pin is set to output when its bit is set to 1, and as input when it is set to 0. Accessible as word. Reset value = 0, i.e., all bits are set to input on reset.	–
FIO0DIR	Example of above for Port 0.	0x2009C000
FIO2DIR	Example of above for Port 2.	0x2009C040
FIO*n*DIR*p*	Sets the data direction of each pin in byte *p* of Port *n*, where *p* takes value 0 to 3. A port pin is set to output when its bit is set to 1. Accessible as byte.	–
FIO0DIR0	Example of above, Port 0 byte 0.	0x2009C000
FIO0DIR1	Example of above, Port 0 byte 1.	0x2009C001
FIO2DIR0	Example of above, Port 2 byte 0.	0x2009C040
FIO0PIN	Sets the data value of each bit in Port 0 or 2. Accessible as word. Reset value = 0.	0x2009C014
FIO2PIN		0x2009C054
FIO0PIN0	Sets the data value of each bit in least significant byte of Port 0 or 2. Accessible as byte. Reset value = 0.	0x2009C014
FIO2PIN0		0x2009C054

A second set of registers, called **FIOPIN**, hold the data value of the microcontroller's pins, whether they have been set as input or output. Again, two are seen in Table 14.1, with the same naming pattern as the **FIODIR** registers. If a port bit has been set as an output, then writing to its corresponding bit in its **FIOPIN** register will control the logic value placed on that pin. If the pin has been set as input, then reading from that bit will tell you the logic value asserted at the pin. The **FIODIR** and **FIOPIN** registers are the only two register sets we need to worry about for our first simple I/O programs.

14.3.2 A Digital Output Application

As we well know, a digital output can be configured and then tested simply by flashing an LED. We did this back at the beginning of the book, in Program Example 2.1. We will look at the complete method for making this happen, working directly with the microcontroller registers. In this first program, Program Example 14.1, we will use Port 2 Pin 0 as our digital output. You can see from the mbed schematic (Ref. [2] of Chapter 2) that this pin is routed to the mbed pin 26.

Follow through Program Example 14.1 with care. Notice that for once we're *not* writing `#include "mbed.h"`! At the program's start, the names of two registers from Table 14.1 are defined to equal the contents of their addresses. This is done by defining the addresses as pointers (see Section B8.2), using the * operator. The format of this is not entirely simple, and for our purposes we can use it as shown. Further information can however be found from [1]. Having defined these pointers, we can now refer to the register names in the code rather than worrying about the addresses. The C

keyword **volatile** is also applied. This is used to define a data type whose value may change outside program control. This is a common occurrence in embedded systems, and is particularly applicable to memory-mapped registers, such as we are using here, which can be changed by the external hardware.

Within the **main( )** function the data direction register of Port 2, byte 0 is set to output, by setting all bits to Logic 1. A **while** loop is then established, just as in Program Example 2.1. Within this loop pin 0 of Port 2 is set high and low in turn. Check the method of doing this, if you are not familiar with it, as this shows one way of manipulating a single bit within a larger word. To set the bit high, it is ORed with logic 1. All other bits are ORed with logic 0, so will not change. To set it low, it is ANDed with binary 1111 1110. This has the effect of returning the LSB to 0, while leaving all other bits unchanged. The 1111 1110 value is derived by taking the logic inversion of 0x01, using the C ~ operator.

The delay function should be self-explanatory, and is explored more in Exercise 14.1. Its function prototype appears early in the program.

```
/*Program Example 14.1: Sets up a digital output pin using control registers, and
flashes an led.
                                                                              */
// function prototypes
void delay(void);

//Define addresses of digital i/o control registers, as pointers to volatile data
#define FIO2DIR0       (*(volatile unsigned char *)(0x2009C040))
#define FIO2PIN0       (*(volatile unsigned char *)(0x2009C054))

int main() {
  FIO2DIR0=0xFF;    // set port 2, lowest byte to output
  while(1) {
    FIO2PIN0 |= 0x01;    // OR bit 0 with 1 to set pin high
    delay();
    FIO2PIN0 &= ~0x01;  // AND bit 0 with 0 to set pin low
    delay();
  }
}
//delay function
void delay(void){
  int j;                      //loop variable j
  for (j=0;j<1000000;j++) {
    j++;
    j--;                      //waste time
  }
}
```

Program Example 14.1: Manipulating control registers to flash an LED

Connect an LED between an mbed pin 26 and 0 V, and compile, download, and run the code. Press reset and the LED should flash.

Exercise 14.1

With an oscilloscope, carefully measure the duration of the delay function in Program Example 14.1. Estimate the execution time for the `j++` and `j--` instructions. Adjust the delay function experimentally so that it is precisely 100 ms. Then, create a new 1 s delay function, which works by calling the 100 ms one 10 times. Now, write a library delay routine, which gives a delay of *n* ms, where *n* is the parameter sent.

■

14.3.3 Adding a Second Digital Output

Following the principles outlined above, we can add further digital outputs. Here, we add a second LED and make a flashing pattern. Port 2 Pin 1 is used, which connects to mbed pin 25. Add a second LED to this pin, preferably of a different color. We have already defined the direction control and pin IO registers for Port 2, so we don't need to add any new registers. We do, however, add a variable to allow us to generate an interesting flashing pattern.

Copy Program Example 14.1 to a new program, and add into it the following variable declaration, prior to the main program function:

```
char i;
```

Also add the **for** loop of Program Example 14.2 at the end of the **while** loop. Notice that now we are toggling bit 1 of the **FIO2PIN0** register. We do this as before by ORing with 0x02 and then ANDing with the inverse of 0x02. This is where the second LED is connected.

```
for (i=1;i<=3;i++){
  FIO0PIN0 |= 0x02;        // set port 2 pin 1 high (mbed pin 25)
  delay();
  FIO0PIN0 &= ~0x02;       // set port 2 pin 1 low
  delay();
}
```

Program Example 14.2 (code fragment): Controlling a second LED output

Run the program. It should flash the first LED once, followed by three flashes of the second. If you have different color LEDs, this will give you a pattern something like

green—red—red—red—green—red—red—red—green—…

Exercise 14.2

In the mbed circuit diagrams (Ref. [3] of Chapter 2), identify the LPC1768 pins that drive the onboard LEDs. Rewrite Program Example 14.1/14.2 so that the onboard LEDs are activated, instead of using external ones.

■

14.3.4 Digital Inputs

We can create digital inputs simply by setting a port bit to input, using the correct bit in an **FIODIR** register. Program Example 14.3 now develops the previous example, by including a digital input from a switch. The state of the switch changes the loop pattern, determining which LED flashes three times and which flashes just once in a cycle. This then gives a control system which has outputs that are dependent on particular input characteristics. It uses the outputs already used, and adds bit 0 of Port 0 as a digital input. Looking at the schematic, we can see that this is pin 9 of the mbed.

It should not be difficult to follow Program Example 14.3 through. As before, we see the necessary register addresses defined. Directly inside the **main( )** function Port 0 byte 0 is set as a digital input, noting that a Logic 0 sets the corresponding pin to input. We have set all of byte 0 to input by sending 0x00; we could of course set each pin within the byte individually if needed. Moreover, this setting is not fixed; we can change a pin from input to output as a program executes. After all this, a reading of Table 14.1 reminds us that the reset value of all ports is as input. Therefore, this little bit of code isn't actually necessary—try removing it when you run the program. However, it is good practice to reassert values which are said to be in place due to the reset; it gives you the confidence that the value is in place, and it's a definite statement in the code of a setting that you want.

The **while** loop then starts, and at the beginning of this, we see the **if** statement testing the digital input value. The **if** condition uses a bit mask to discard the value of all the other pins on Port 0 byte 0 and simply returns a high or low result dependent on pin 8 alone. The variables **a** and **b** will hold values which will change depending on the switch position. A little later we see the **a** and **b** values used to define the port values for the green and red LEDs. If **b** has been set to 0x01 before the **for** loop, then the red LED will flash three times; if it has been set to 0x02, then the green LED will flash three times.

```
/* Program Example 14.3: Uses digital input and output using control registers, and
flashes an LED. LEDS connect to mbed pins 25 and 26. Switch input to pin 9.
                                                */
// function prototypes
void delay(void);
```

```
//Define Digital I/O registers
#define FIO0DIR0 (*(volatile unsigned char *)(0x2009C000))
#define FIO0PIN0 (*(volatile unsigned char *)(0x2009C014))
#define FIO2DIR0 (*(volatile unsigned char *)(0x2009C040))
#define FIO2PIN0 (*(volatile unsigned char *)(0x2009C054))
//some variables
char a;
char b;
char i;

int main() {
   FIO0DIR0=0x00;             // set all bits of port 0 byte 0 to input
   FIO2DIR0=0xFF;             // set port 2 byte 0 to output
   while(1) {
     if (FIO0PIN0&0x01==1){   // bit test port 0 pin 0 (mbed pin 9)
       a=0x01;                // this reverses the order of LED flashing
       b=0x02;                // based on the switch position
     }
     else {
       a=0x02;
       b=0x01;
     }
     FIO2PIN0 |= a;
     delay();
     FIO2PIN0 &= ~a;
     delay();

     for (i=1;i<=3;i++){
       FIO2PIN0 |= b;
       delay();
       FIO2PIN0 &= ~b;
       delay();
     }
   }                          //end while loop
}

//delay function
void delay(void){
  int j;                      //loop variable j
  for (j=0;j<1000000;j++) {
    j++;
    j--;                      //waste time
  }
}
```

Program Example 14.3: Combined digital input and output

To run this program, set up a circuit similar to Fig. 3.6, except that the green and red LEDs should connect to mbed pins 25 and 26 respectively, and the switch input is on pin 9.

Compile and run. You should see that the position of the switch toggles the flashing LEDs between the patterns

green—red—red—red—green—red—red—red—green—…

and

green—green—green—red—green—green—green—red—green—…

Exercise 14.3

Rewrite Program Example 14.3 so that it runs on the exact circuit of Fig. 3.6.

■

14.4 Getting Deeper Into the Control Registers

To continue with this chapter, we need to get further into the use of control registers. In this section, we look therefore at some of the registers which control features across the microcontroller—we could call them informally "global" registers—which we will need to use in the later sections. These relate to the allocation of pins, setting clock frequency, and controlling power. This is by no means a complete survey, and we will not even see or make use of many of the features of this microcontroller.

14.4.1 Pin Select and Pin Mode Registers

One of the reasons that modern microcontrollers are so versatile is that most pins are multifunctional. They can be allocated to different peripherals, and used in different ways. With the mbed library, this flexibility is (quite reasonably) more or less hidden from you; the libraries tidily make the allocations for you, without you even knowing. If they didn't, you would be faced with a bewildering choice of possibilities every time you tried to develop an application. As our expertise grows, however, it's good to know that some of these possibilities are available.

Two important sets of registers used in the LPC1768 are called **PINSEL** and **PINMODE**. The **PINSEL** register can allocate each pin to one of four possibilities. An example of part of one register which we will be using soon, **PINSEL1**, is shown in Table 14.2. This controls the upper half of Port 0. The first column shows the bit number within the register; each line details two bits. The second column shows the microcontroller pin which is being controlled. The two bits under consideration can have four possible combinations; each of these connects the pin in a different way. These are shown in the next four columns. Don't worry if some of the abbreviations shown have little meaning to you; we'll pick out the ones we need, when we need them.

Table 14.2: PINSEL1 register (address 0x4002 C004).

PINSEL1	Pin Name	Function When 00	Function When 01	Function When 10	Function When 11	Reset Value
1:0	P0.16	GPIO Port 0.16	RXD1	SSEL0	SSEL	00
3:2	P0.17	GPIO Port 0.17	CTS1	MISO0	MISO	00
5:4	P0.18	GPIO Port 0.18	DCD1	MOSI0	MOSI	00
7:6	P0.19[a]	GPIO Port 0.19	DSR1	Reserved	SDA1	00
9:8	P0.20[a]	GPIO Port 0.20	DTR1	Reserved	SCL1	00
11:10	P0.21[a]	GPIO Port 0.21	RI1	Reserved	RD1	00
13:12	P0.22	GPIO Port 0.22	RTS1	Reserved	TD1	00
15:14	P0.23[a]	GPIO Port 0.23	AD0.0	I2SRX_CLK	CAP3.0	00
17:16	P0.24[a]	GPIO Port 0.24	AD0.1	I2SRX_WS	CAP3.1	00
19:18	P0.25	GPIO Port 0.25	AD0.2	I2SRX_SDA	TXD3	00
21:20	P0.26	GPIO Port 0.26	AD0.3	AOUT	RXD3	00
23:22	P0.27[a,b]	GPIO Port 0.27	SDA0	USB_SDA	Reserved	00
25:24	P0.28[a,b]	GPIO Port 0.28	SCL0	USB_SCL	Reserved	00
27:26	P0.29	GPIO Port 0.29	USB_D+	Reserved	Reserved	00
29:28	P0.30	GPIO Port 0.30	USB_D−	Reserved	Reserved	00
31:30	—	Reserved	Reserved	Reserved	Reserved	00

[a]Not available on 80-pin package.
[b]Pins P0[27] and P0[28] are open-drain for I^2C-bus compliance.
From Table 81 of LPC1768 User Manual.

Let's take as an example the line showing the effect of bits 21:20, i.e., line 11 of the table; these control bit 26 of Port 0. Column 3 shows that if the bits are 00, the pin is allocated to Port 0 bit 26, i.e., the pin is connected as general purpose I/O. Importantly, the final column shows that this is also the value when the chip is reset. In other words, as long as we only want to use digital I/O, we don't need to worry about this register at all, as the reset value is the value we want. If the bits are set to 01, the pin is allocated to input 3 of the ADC. If set to 10, the pin is used for analog output (i.e., the DAC output). If set to 11, the pin is allocated as receiver input for UART 3.

Turning to the **PINMODE** registers, partial details of one of these is shown in Table 14.3. This is **PINMODE0**, which controls the input characteristics of the lower half of Port 0. The pattern is the same for every pin, so there's no need for repetition. It's easy to see that pull-up and pull-down resistors (as seen in Fig. 3.5) are available, we explore this in Exercise 14.4. The repeater mode is a neat little facility which enables a pull-up resistor when the input is a Logic 1, and pull-down if it's low. If the external circuit changes so that the input is no longer driven, then the input will hold its most recent value.

Table 14.3: PINMODE0 register (address 0x4002 C040).

PINMODE0	Symbol	Value	Description	Reset Value
1:0	P0.00MODE		Port 0 pin 0 on-chip pull-up/down resistor control.	00
		00	P0.0 pin has a pull-up resistor enabled.	
		01	P0.0 pin has repeater mode enabled.	
		10	P0.0 pin has neither pull-up nor pull-down.	
		11	P0.0 has a pull-down resistor enabled.	
3:2	P0.01MODE		Port 0 pin 1 control, see P0.00MODE.	00
	Continued to P0.15MODE			

From Table 88 of LPC1768 User Manual.

Exercise 14.4

Change the hardware for Program Example 14.3 so that you use an SPST (single pole, single throw) switch, for example, a push button, instead of the toggle (single pole, double throw (SPDT)) switch. Connect it first between pin 9 and ground, and run the program with no change. This should run as before, because you are depending on the pull-up resistor being in place due to the reset value of the **PINMODE** register. In diagrammatic terms, you have moved from the switch circuit of Fig. 3.5A, to that of Fig. 3.5B. Now change the setting of **PINMODE0** so the pull-down resistor is enabled, and connect the switch between pin 9 and 3.3 V, i.e., applying Fig. 3.5C. The program should again work, but with the changed input mode selection.

■

14.4.2 Power Control and Clock Select Registers

As Chapter 15 will describe in detail, power control and clock frequency are very closely linked. Every clock transition causes the circuit of the microcontroller to take a tiny pulse of current; the more transitions, the more the current taken. Hence a processor or peripheral running at a high clock speed will cause high power consumption; one running at a low clock frequency will consume less power. One with its clock switched off, even if it is powered up, will (if a purely digital circuit using CMOS technology) take negligible power. To conserve power, it is possible to turn off the clock source to many of the LPC1768 peripherals. This power management is controlled by the **PCONP** register, seen in part in Table 14.4. Where a bit is set to 1, the peripheral is enabled, when set to 0 it is disabled. It is interesting to note that some peripherals (like the serial peripheral interface (SPI)) are reset in the enabled mode, and others (like the ADC) are reset disabled.

Aside from being able to switch the clock to a peripheral on or off, there is some control over the peripheral's clock frequency itself. This controls the peripheral's speed of operation

Table 14.4: Power control register PCONP (address 0x400F C0C4).

Bit	Symbol	Description	Reset Value
0	—	Reserved.	NA
1	PCTIM0	Timer/Counter 0 power/clock control bit.	1
2	PCTIM1	Timer/Counter 1 power/clock control bit.	1
3	PCUART0	UART0 power/clock control bit.	1
4	PCUART1	UART1 power/clock control bit.	1
5	—	Reserved.	NA
6	PCPWM1	PWM1 power/clock control bit.	1
7	PCI2C0	The I^2C0 Interface power/clock control bit.	1
8	PCSPI	The SPI interface power/clock control bit.	1
9	PCRTC	The RTC power/clock control bit.	1
10	PCSSP1	The SSP 1 interface power/clock control bit.	1
11	—	Reserved.	NA
12	PCADC	A/D converter (ADC) power/clock control bit	0
Continued to bit 28			

From Table 46 of LPC1768 User Manual.

Table 14.5: Peripheral Clock Selection register PCLKSEL0 (address 0x400F C1A8).

Bit	Symbol	Description	Reset Value
1:0	PCLK_WDT	Peripheral clock selection for WDT.	00
3:2	PCLK_TIMER0	Peripheral clock selection for TIMER0.	00
5:4	PCLK_TIMER1	Peripheral clock selection for TIMER1.	00
7:6	PCLK_UART0	Peripheral clock selection for UART0.	00
9:8	PCLK_UART1	Peripheral clock selection for UART1.	00
11:10	—	Reserved.	NA
13:12	PCLK_PWM1	Peripheral clock selection for PWM1.	00
15:14	PCLK_I2C0	Peripheral clock selection for I^2C0.	00
17:16	PCLK_SPI	Peripheral clock selection for SPI.	00
19:18	—	Reserved.	NA
21:20	PCLK_SSP1	Peripheral clock selection for SSP1.	00
23:22	PCLK_DAC	Peripheral clock selection for DAC.	00
25:24	PCLK_ADC	Peripheral clock selection for ADC.	00
27:26	PCLK_CAN1	Peripheral clock selection for CAN1.[a]	00
29:28	PCLK_CAN2	Peripheral clock selection for CAN2.[a]	00
31:30	PCLK_ACF	Peripheral clock selection for CAN acceptance filtering.[a]	00

[a]PCLK_CAN1 and PCLK_CAN2 must have the same PCLK divide value when the CAN function is used.

From Table 40 of LPC1768 User Manual.

Table 14.6: Peripheral clock selection register bit values.

PCLKSEL0 and PCLKSEL1 Individual Peripheral's Clock Select Options	Function	Reset Value
00	PCLK_peripheral = CCLK/4	00
01	PCLK_peripheral = CCLK	
10	PCLK_peripheral = CCLK/2	
11	PCLK_peripheral = CCLK/8, except for CAN1, CAN2, and CAN filtering when "11" selects = CCLK/6	

From Table 42 of LPC1768 User Manual.

as well as its power consumption. This clock frequency is controlled by the **PCLKSEL** registers. Peripheral clocks are derived from the clock that drives the CPU, which is called **CCLK** (or **cclk**). For the mbed, **CCLK** normally runs at 96 MHz. Partial details of **PCLKSEL0** are shown in Table 14.5. We can see that two bits are used per peripheral to control the clock frequency to each. The four possible combinations are shown in Table 14.6. This shows that the **CCLK** frequency itself can be used to drive the peripheral. Alternatively, it can be divided by 2, 4, or 8. **CCLK** is derived from the main oscillator circuit and can be manipulated in a number of interesting ways, as described in Chapter 15.

14.5 Using the DAC

We now turn to controlling the DAC through its registers, trying to replicate and develop the work we did in Chapter 4. Remind yourself of the general block diagram of the DAC, as seen in Fig. 4.1. It's worth mentioning here that the positive reference voltage input to the LPC1768, shared by both ADC and DAC, is called V_{REFP}, and is connected to the power supply 3.3 V. The negative reference voltage input, called V_{REFN}, is connected directly to ground. We explore this further in Chapter 15.

14.5.1 LPC1768 DAC Control Registers

As with all peripherals, the DAC has a set of registers which control its activity. In terms of the "global" registers which we have just seen, the DAC power is always enabled, so there is no need to consider the **PCONP** register. The *only* pin that the DAC output is available on is Port 0 pin 26, so we must allocate this pin appropriately through the **PINSEL1** register, as we saw in Table 14.2. The DAC output is labeled AOUT here. It is no surprise to see in the mbed schematics that this pin is connected to mbed pin 18, the mbed's only analog output.

The only register specific to the DAC that we will use is the **DACR** register. This is comparatively simple to grasp, and is shown in Table 14.7. We can see that the digital

Table 14.7: The DACR register (address 0x4008 C000).

Bit	Symbol	Value	Description	Reset Value
5:0	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
15:6	VALUE		After the selected settling time after this field is written with a new VALUE, the voltage on the AOUT pin (with respect to V_{SSA}) is VALUE x $((V_{REFP} - V_{REFN})/1024) + V_{REFN}$.	0
16	BIAS	0	The settling time of the DAC is 1 μs max, and the maximum current is 700 μA. This allows a maximum update rate of 1 MHz.	0
		1	The settling time of the DAC is 2.5 μs and the maximum current is 350 μA. This allows a maximum update rate of 400 kHz.	
31:17	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined	NA

From Table 540 of LPC1768 User Manual.

input to the DAC must be placed in here, in bits 6 to 15. Most of the rest of the bits are unused, apart from the bias bit, explained in the table.

Applying Eq. (4.1) to this 10-bit DAC, its output is given by:

$$V_0 = (V_{REFP} \times D)/1024 = (3.3 \times D)/1024 \tag{14.1}$$

where D is the value of the 10-bit number placed in bits 15 to 6 of the **DACR** register.

14.5.2 A DAC Application

Let's try a simple program to drive the DAC, accessing it through the microcontroller control registers. Program Example 14.4 replicates the simple saw tooth output, which we first achieved in Program Example 4.2. The program follows the familiar pattern of defining register addresses, and then setting these appropriately early in the **main( )** function. The **PINSEL1** register is set to select DAC output on port bit 0.26. An integer variable, called **dac_value**, is then repeatedly incremented and transferred to the DAC input, in register **DACR**. It has to be shifted left six times, to place it in the correct bits of the **DACR** register. Between each new value of DAC input, a delay is introduced. We explore the effect of this in Exercise 14.5.

```
/* Program Example 14.4: Sawtooth waveform on DAC output. View on oscilloscope.
Port 0.26 is used for DAC output, i.e., mbed Pin 18
                                                    */
```

```
// function prototype
void delay(void);
// variable declarations
int dac_value;             //the value to be output
//define addresses of control registers, as pointers to volatile data
#define DACR (*(volatile unsigned long *)(0x4008C000))
#define PINSEL1 (*(volatile unsigned long *)(0x4002C004))

int main(){
  PINSEL1=0x00200000; //set bits 21-20 to 10 to enable analog out on P0.26
  while(1){
    for (dac_value=0;dac_value<1023;dac_value=dac_value+1){
      DACR=(dac_value<<6);
      delay();
    }
  }
}

//delay function
void delay(void){
  int j;                       //loop variable j
  for (j=0;j<1000000;j++) {
    j++;
    j--;                       //waste time
  }
}
```

Program Example 14.4: Saw tooth output on the DAC

Compile the program and run it on an mbed; no external connections are needed. View the output from the mbed pin 18 on an oscilloscope.

Exercise 14.5

Measure the period of the saw tooth waveform. How does it relate to the delay value you measured in Exercise 14.1? Try varying the period by varying the delay value, or removing it altogether. Can you estimate how long a single digital-to-analog conversion takes?

■

14.6 Using the ADC

We now turn to controlling the ADC through its registers, trying to replicate and develop the work we did in Chapter 5. It is worth glancing back at Fig. 5.1, as this represents many of the features that we need to control. This includes selecting (or at least knowing the value of) the voltage reference, clock speed and input channel, starting a conversion, detecting a completion, and reading the output data. The LPC1768 has eight inputs to its

ADC, which appear—in order from input 0 to input 7—on pins 9–6, 21, 20, 99, and 98 of the microcontroller. A study of the mbed circuit shows that the lower six of these are used, connected to pins 15 to 20 inclusive of the mbed.

14.6.1 LPC1768 ADC Control Registers

The LPC1768 has a number of registers which control its ADC, particularly in its more sophisticated operation. However, we will only apply two of these, the ADC control register, **ADCR**, and the Global Data Register, **ADGDR**. These are detailed in Tables 14.8 and 14.9.

Table 14.8: The AD0CR register (address 0x4003 4000).

Bit	Symbol	Value	Description	Reset Value
7:0	SEL		Selects which of the AD0.7:0 pins is (are) to be sampled and converted. For AD0, bit 0 selects Pin AD0.0. and bit 7 selects pin AD0.7. In software-controlled mode, only one of these bits should be 1. In hardware scan mode, any value containing 1 to 8 ones is allowed. All zeroes is equivalent to 0x01.	0x01
15:8	CLKDIV		The APB clock (PCLK_ADC0) is divided by (this value plus one) to produce the clock for the A/D converter, which should be less than or equal to 13 MHz. Typically, software should program the smallest value in this field that yields a clock of 13 MHz or slightly less, but in certain cases (such as a high-impedance analog source) a slower clock may be desirable.	0
16	BURST	1	The A/D converter does repeated conversions at up to 200 kHz, scanning (if necessary) through the pins selected by bits set to ones in the SEL field. The first conversion after the start corresponds to the least-significant 1 in the SEL field, then higher numbered 1 bits (pins) if applicable. Repeated conversions can be terminated by clearing this bit, but the conversion that's in progress when this bit is cleared will be completed. **Remark:** START bits must be 000 when BURST = 1 or conversions will not start.	0
		0	Conversions are software controlled and require 65 clocks.	
20:17	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
21	PDN	1	The A/D converter is operational.	0
		0	The A/D converter is in power-down mode.	
23:22	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
26:24	START		When the BURST bit is 0, These bits control whether and when an A/D conversion is stared:	0
		000	No start (this value should be used when clearing PDN to 0).	
		001	Start conversion now.	

Note: further more advanced options for START control are available, see full Table.
From Table 532 of LPC1768 User Manual.

Table 14.9: The AD0GDR register (address 0x4003 4004).

Bit	Symbol	Description	Reset Value
3:0	—	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined	NA
15:4	RESULT	When DONE is 1, this field contains a binary fraction representing the voltage on the AD0[n] pin selected by the SEL field, as it falls within the range of V_{REFP} to V_{REFN}. Zero in the field indicates that the voltage on the input pin was less than, equal to, or close to that on V_{REFN}, while 0x3FF indicates that the voltage on the Input was close to, equal to, or greater than that on V_{REFP}.	NA
23:16	—	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined	NA
26:24	CHN	These bits contain the channel from which the RESULT bits were converted (e.g., 000 identifies channel 0, 001 channel 1...).	NA
29:27	—	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
30	OVERRUN	This bit is 1 in burst mode if the results of one or more conversions was (were) lost and overwritten before the conversion that produced the result in the RESULT bits. This bit is cleared by reading this register.	0
31	DONE	This bit is set to 1 when an A/D conversion completes. It is cleared when this register is read and when the ADCR is written. If the ADCR is written while a conversion is still in progress, this bit is set and a new conversion is started.	0

From Table 533 of LPC1768 User Manual.

As we have seen, the ADC can also be powered down, indeed on microcontroller reset it is switched off. Therefore, to enable it, we will have to set bit 12 in the **PCONP** register, seen in Table 14.4.

14.6.2 An ADC Application

Program Example 14.5 provides a good opportunity to see many of the control registers in action. Channel 1 of the ADC is applied, which connects to pin 16 of the mbed.

The configuration of the ADC must be done with care, so read the program with diligence, starting from the comment "initialize the ADC." The ADC channel we want is multiplexed with bit 24 of Port 0, so first we must allocate this pin to the ADC. This is done through bits 17 and 16 of the **PINSEL1** register, **seen in** Table 14.2. We then enable the ADC, through the relevant bit in

the **PCONP** register, Table 14.4. There follows quite a complex process of configuring the ADC, through the **AD0CR** register. We could set this register by transferring a single word. Instead the relevant bits are shifted in, in turn. Check each with Table 14.8.

The data conversion then starts in the **while** loop which follows. The comments contained in the program listing should give you a good picture of each stage.

```
/* Program Example 14.5: A bar graph meter for ADC input, using control registers
to set up ADC and digital I/O
                                                                                */

// variable declarations
char ADC_channel=1;     // ADC channel 1
int  ADCdata;           //this will hold the result of the conversion
int DigOutData=0;       //a buffer for the output display pattern

// function prototype
void delay(void);

//define addresses of control registers, as pointers to volatile data
//(i.e. the memory contents)
#define PINSEL1        (*(volatile unsigned long *)(0x4002C004))
#define PCONP          (*(volatile unsigned long *)(0x400FC0C4))
#define AD0CR          (*(volatile unsigned long *)(0x40034000))
#define AD0GDR         (*(volatile unsigned long *)(0x40034004))
#define FIO2DIR0       (*(volatile unsigned char *)( 0x2009C040))
#define FIO2PIN0       (*(volatile unsigned char *)( 0x2009C054))

int main() {
  FIO2DIR0=0xFF;// set lower byte of Port 2 to output, this drives bar graph

//initialise the ADC
  PINSEL1=0x00010000; //set bits 17-16 to 01 to enable AD0.1 (mbed pin 16)
  PCONP |= (1 << 12);                // enable ADC clock
  AD0CR =   (1 << ADC_channel)       // select channel 1
          | (4 << 8)       // Divide incoming clock by (4+1), giving 4.8MHz
          | (0 << 16)      // BURST = 0, conversions under software control
          | (1 << 21)      // PDN = 1, enables power
          | (1 << 24);     // START = 1, start A/D conversion now

  while(1) {                     // infinite loop
    AD0CR = AD0CR | 0x01000000;   //start conversion by setting bit 24 to 1,
                                                          //by ORing
    // wait for it to finish by polling the ADC DONE bit
    while ((AD0GDR & 0x80000000) == 0) {  //test DONE bit, wait till it's 1
    }
    ADCdata = AD0GDR;            // get the data from AD0GDR
    AD0CR &= 0xF8FFFFFF;         //stop ADC by setting START bits to zero
```

```
   // Shift data 4 bits to right justify, and 2 more to give 10-bit ADC
   // value - this gives convenient range of just over one thousand.
    ADCdata=(ADCdata>>6)&0x03FF;    //and mask
    DigOutData=0x00;                //clear the output buffer
    //display the data
    if (ADCdata>200)
      DigOutData=(DigOutData|0x01);  //set the lsb by ORing with 1
    if (ADCdata>400)
      DigOutData=(DigOutData|0x02);  //set the next lsb by ORing with 1
    if (ADCdata>600)
      DigOutData=(DigOutData|0x04);
    if (ADCdata>800)
      DigOutData=(DigOutData|0x08);
    if (ADCdata>1000)
      DigOutData=(DigOutData|0x10);

    FIO2PIN0 = DigOutData;        // set port 2 to Digoutdata
    delay();      // pause
   }
}

    //delay function
void delay(void){
  int j;                      //loop variable j
  for (j=0;j<1000000;j++) {
    j++;
    j--;                      //waste time
 }
}
```

Program Example 14.5: Applying the ADC as a bar graph

Connect an mbed with a potentiometer between 0 and 3.3 V with the wiper connected to mbed pin 16. Connect five LEDs between pin and ground, from pin 22 to pin 26 inclusive. Compile and download your code to the mbed, and press reset. Moving the potentiometer should alter the number of LEDs lit, from none at all to all five.

Exercise 14.6

Add a digital input switch as in the previous example to reverse the operation of the analog input. With the digital switch in one position, the LEDs will light from right to left; with the switch in the other position, the **DigOutData** variable is inverted to light LEDs in the opposite direction, from left to right.

■

Exercise 14.7

Extend the bar graph so that it has eight or ten LEDs.

■

14.6.3 Changing ADC Conversion Speed

One of the limitations of the mbed library is the comparatively slow speed of the ADC conversion. We explored this in Exercise 5.7. Let's try now to vary this conversion speed by adjusting the ADC clock speed.

Table 14.8 tells us that the ADC clock frequency should have a maximum value of 13 MHz, and that it takes 65 cycles of the ADC clock to complete a conversion. A quick calculation shows that the minimum conversion time possible is therefore 5 μs. It takes a very careful reading of the LPC1768 user manual, Ref. [5] of Chapter 2, to get a full picture of how the ADC clock frequency is controlled. The ADC clock is derived from the main microcontroller clock; there are several stages of division that the user can control in order to set up a frequency as close to 13 MHz as possible. The first is through register **PCLKSEL0**, detailed in Tables 14.5 and 14.6. Bits 25 and 24 of **PCLKSEL0** control the ADC clock division. We have seen that for most peripherals, including the ADC, the clock can be divided by one, two, four, or eight. On power-up, the selection defaults to divide-by-four. The clock may be further divided through bits 15 to 8 of the **AD0CR** register, seen in Table 14.8.

Program Example 14.6 replicates Program Example 5.5, with some interesting results. It is also a useful example, as it combines ADC, DAC, and digital I/O, therefore illustrating how these can be used together. It is made up of elements from programs earlier in this chapter, sometimes with adjustments; it should be possible to follow it through without too much difficulty. As sections of the program repeat from earlier examples, only the newer parts are reproduced here. The full program listing can be downloaded from the book website.

```
/* Program Example 14.6: Explore ADC conversion times, programming control
registers directly. ADC value is transferred to DAC, while an output pin is strobed
to indicate conversion duration. Observe on oscilloscope
                                                                              */
....
....

int main() {
  FIO2DIR0=0xFF;                      // set lower bits port 2 to output
  PINSEL1=0x00210000;  //set bits 21-20 to 10 for analog output (mbed p18)
         //and bits 17-16 to 01 to enable ADC channel 1 (AD0.1, mbed pin 16)

//initialise the ADC.
....
```

```
....
  while(1){         // infinite loop
    // start A/D conversion by modifying bits in the AD0CR register
    AD0CR &= (AD0CR & 0xFFFFFF00);
    FIO2PIN0 |= 0x01;             // OR bit 0 with 1 to set pin high
    AD0CR |= (1 << ADC_channel) | (1 << 24);
    // wait for it to finish by polling the ADC DONE bit
    while((AD0GDR & 0x80000000) == 0) {
    }
    FIO2PIN0 &= ~0x01;            // AND bit 0 with 0 to set pin low

    ADCdata = AD0GDR;              // get the data from AD0GDR
    AD0CR &= 0xF8FFFFFF;           //stop ADC by setting START bits to zero

    // shift data 4 bits to right justify, and 2 more to give 10-bit ADC value
    ADCdata=(ADCdata>>6)&0x03FF;   //and mask
    DACR=(ADCdata<<6);     //could be merged with previous line,
                              // but separated for clarity
    //delay();               //insert delay if wished
  }
}
...
```

Program Example 14.6: Applying ADC, DAC, and digital output, to measure conversion duration

You can use the same mbed configuration for this as you did for Program Example 14.5, although only the LED on pin 26 is necessary. Compile and run the program. First put an oscilloscope probe on pin 18, the DAC output. The voltage on this pin should change as the potentiometer is used. This confirms the program is running. Now move the probe to pin 26; you will see the pin pulsing high for the duration of the ADC conversion. If you measure this, you should find it is 14 μs, or just under.

To calculate the ADC clock frequency, and hence conversion time, remember that the mbed **CCLK** frequency is 96 MHz. We have not touched the **PCLKSEL0** register, so the clock setting for the ADC will be 96 MHz divided by the reset value of 4, i.e., 24 MHz. This is further divided by 5 in the **ADC0CR** setting seen in the program example, leading to an ADC clock frequency of 4.8 MHz, or period of 0.21 μs. Sixty-five cycles of 0.21 μs leads to the measurement duration mentioned in the previous paragraph.

Exercise 14.8

1. Adjust the setting of the **CLKDIV** bits in **AD0CR** in Program Example 14.6 to give the fastest permissible conversion time. Run the program, and check that your measured value agrees with the predicted.

2. Can you now account for the value of ADC conversion time you measured in Exercise 5.7?

■

14.7 A Conclusion on Using the Control Registers

In this chapter, we have explored the use of the LPC1768 control registers, in connection with use of the digital I/O, ADC, and DAC peripherals. We have demonstrated how these registers allow the peripherals to be controlled directly, without using the mbed libraries. This has allowed greater flexibility of use of the peripherals, at the cost of getting into the tiny detail of the registers, and programming at the level of the bits that make them up. Ordinarily, we probably wouldn't want to program like this; it's time-consuming, inconvenient, and error-prone. However, if we need a configuration or setting not offered by the mbed libraries, this approach can be a way forward. While we've only worked in this way in connection with three of the peripherals, it's possible to do it with any of them. It's worth mentioning that the three that we have worked with are some of the simpler ones; others require even more attention to detail.

Chapter Review

- In this chapter, we have recognized a different way of controlling the mbed peripherals. It demands a much deeper understanding of the mbed microcontroller, but allows for much greater flexibility.
- There are registers which relate just to one peripheral, and others which relate to microcontroller performance as a whole.
- We have begun to implement features that are not currently available in the mbed library, for example, in the change of the ADC conversion speed.
- The chapter only introduces a small range of the control registers which are used by the LPC1768. However, it should have given you the confidence to look up and begin to apply any that you need.

Quiz

1. How many I/O ports does the LPC1768 have? Hence, theoretically, how many bits does this lead to, how many I/O bits does it actually have, and how many does the mbed have?
2. What is the name of the register which sets the data direction of Port 1, byte 2?
3. Explain the function of the **PINSEL** and **PINMODE** register groups.
4. The initialization section of a certain program reads:

```
FIO0DIR0=0xF0;
PINMODE0=0x0F;
PINSEL1=0x00204000;
```

Explain the settings that have been made.

5. An LPC1768 is connected with a 3.0-V reference voltage. Its DAC output reads 0.375 V. What is its input digital value?
6. A designer is developing a low power application, where high speed is not essential. Which aspect of the DAC control, excluding clock frequency considerations, allows trade-off between speed of conversion and power consumption?
7. On an LPC1768, the ADC clock is set at 4 MHz. How long does one conversion take?
8. A user wants to sample an incoming signal with an mbed ADC at 44 kHz or greater. What is the minimum permissible ADC clock frequency?
9. Describe how the ADC clock frequency calculated in Question 8 can be set up.
10. Which feature of the ADC allows a set of inputs to be selected and scanned, under hardware control? Which feature detects if use of this mode leads to an error?

References

[1] ARM Technical Support Knowledge Articles: Placing C Variables at Specific Addresses to Access Memory-Mapped Peripherals. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka3750.html.