CHAPTER 15

Hardware Insights: Clocks, Resets, and Power Supply

15.1 Hardware Essentials

In the days when cars and motorcycles were simple, i.e., before electronics and embedded systems got into them, they seemed easier to fix. If something went wrong, there were always two things to check first: was there fuel, and was there a spark? No petrol engine could run without. A microcontroller has similar needs, except they are the power supply and the clock. Any serious designer needs to know about these, and how they can be manipulated to best effect. Linked with these two basics come a number of other important features, including:

- start-up and restart features, hence the reset circuit,
- matters to do with reliability, like the watchdog timer (WDT) and brownout detect features,
- the range of low-power modes that are available in most modern microcontrollers, and their use.

This chapter starts and ends with power supply, and in its middle covers these other points. We continue to use the LPC1768 mbed, but also introduce another mbed-enabled system, the EFM32 *Zero Gecko*.

15.1.1 Power Supply and the mbed

We have already seen the mbed power distribution, in Fig. 2.2, and how a 3.3 V supply is generated from one of the incoming power sources. Fig. 15.1, taken from the LPC1768 datasheet (Ref. [4] of Chapter 2), shows that the device actually makes many uses of that 3.3 V. First of all, there are two regular power supply inputs, labeled $V_{DD(3V3)}$ and $V_{DDREG(3V3)}$. It's easy to see that $V_{DD(3V3)}$ simply supplies the I/O ports. However, $V_{DDREG(3V3)}$ goes to a further regulator, which then supplies the inner workings of the microcontroller. In addition to this, there is another possible power input, V_{BAT}; this can be connected to an external battery, to sustain only the real-time clock (RTC) and backup registers, when all other power is lost. Finally, the ADC (and DAC) can be powered

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00015-3

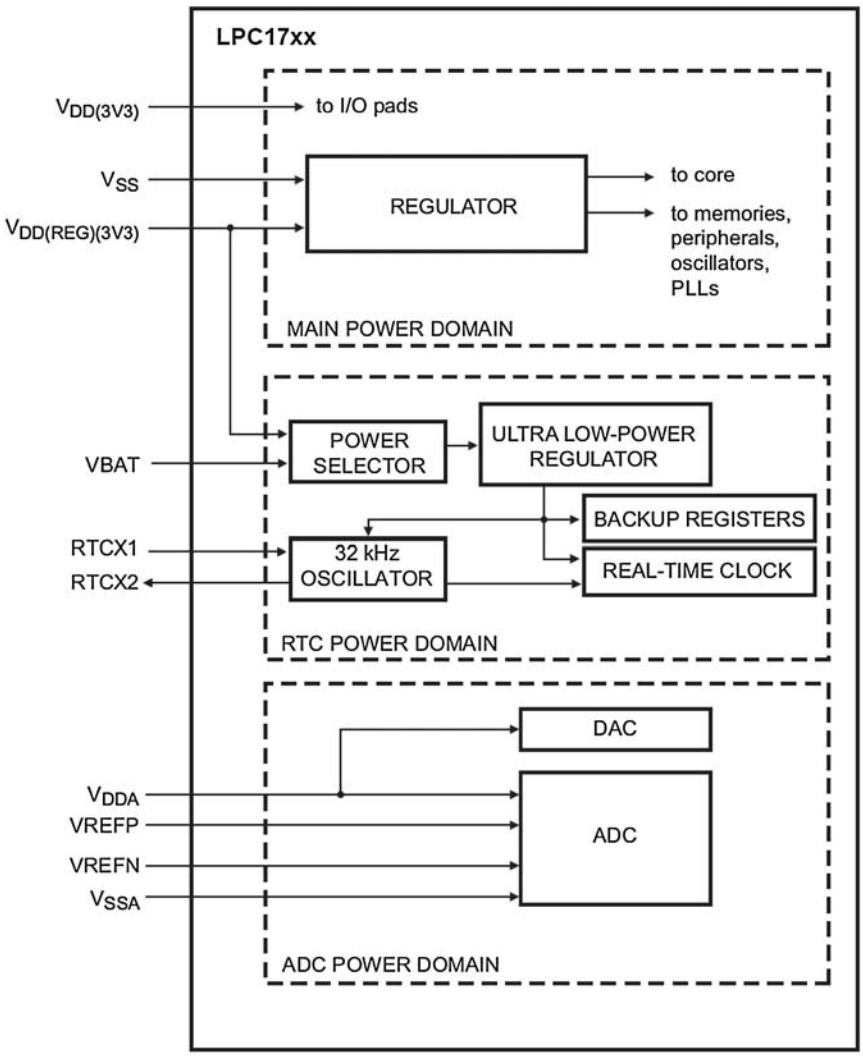


Figure 15.1
Power distribution in the LPC1768. *Image courtesy of NXP.*

separately from the rest of the microcontroller, with their own pristine power supply. A voltage reference also needs to be provided, through the positive and negative voltage pins, V_{REFP} and V_{REFN}.

The supply voltage requirements of the LPC1768 are shown in Table 15.1. All are centered around 3.3 V, with V_{BAT} having the lowest permissible value at 2.1 V. All can go up to 3.6 V, except for the ADC positive reference voltage, which must not exceed the ADC supply voltage.

Table 15.1: Supply voltage requirements for the LPC1768.

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
Supply Pins						
$V_{DD(3V3)}$	supply voltage (3.3 V)	external rail	2.4	3.3	3.6	V
$V_{DD(REG)(3V3)}$	regulator supply voltage (3.3 V)		2.4	3.3	3.6	V
V_{DDA}	analog 3.3 V pad supply voltage		2.5	3.3	3.6	V
$V_{i(VBAT)}$	input voltage on pin VBAT		2.1	3.3	3.6	V
$V_{i(VREFP)}$	input voltage on pin VREFP		2.5	3.3	V_{DDA}	V

Part of Table 8, LPC1768 datasheet Rev. 9.6. August 2015.
$T_{amb} = 40\ °C$ to $+ 85\ °C$, unless otherwise specified.

Turning now to the mbed, the diagram of Fig. 15.2 shows how the mbed designers connect these power inputs, and a number other useful things besides. The main block in the diagram is of course the LPC1768 itself, each connection shows the microcontroller pin number, and the name of the signal. It's first of all worth noting that a complex integrated circuit tends to have more than one ground connection, and similar multiple power supply connections. This is because the interconnecting wires inside the IC are so very thin that they can have significant resistance. These multiple connections are dotted around the IC interconnect pins, to enhance connectivity to the circuit chip itself. First therefore we can note that there are no less than six ground connections, labeled V_{SS}. There are four $V_{DD(3V3)}$ connections and two for $V_{DDREG(3V3)}$. The mbed designers don't take the opportunity to differentiate between these latter two connections, they just join them together. The V_{BAT} connection is, however, kept separate, and *can* be supplied via pin 3, V_B, of the mbed. Note how the V_{DD} supply is smoothed by the distributed capacitors C_{15}–C_{17} and C_{20}, each of 100 nF, and C_{21}, of value 10 μF; placed according to the guidance given. Yes, power supply distribution is a sophisticated art in a complex circuit board.

It is interesting to see how the ADC is powered and referenced. We didn't get into much detail on this in Chapter 5. The mbed designers make a happy compromise here. They don't provide a separate supply, but do filter V_{DD} with the severe, low-pass filter made up of L1 and C14, which will remove much of the high-frequency interference which V_{DD} may have picked up. The ADC uses the same voltage for its supply (V_{DDA}), and for its positive reference (V_{REFP}). This is common practice, although the use of a voltage regulator as a reference risks introducing inaccuracy into the measurement (try Ref. [1] of Chapter 4 to get into the interesting detail of this). Meanwhile, the negative side of the ADC supply and reference are connected straight to system ground.

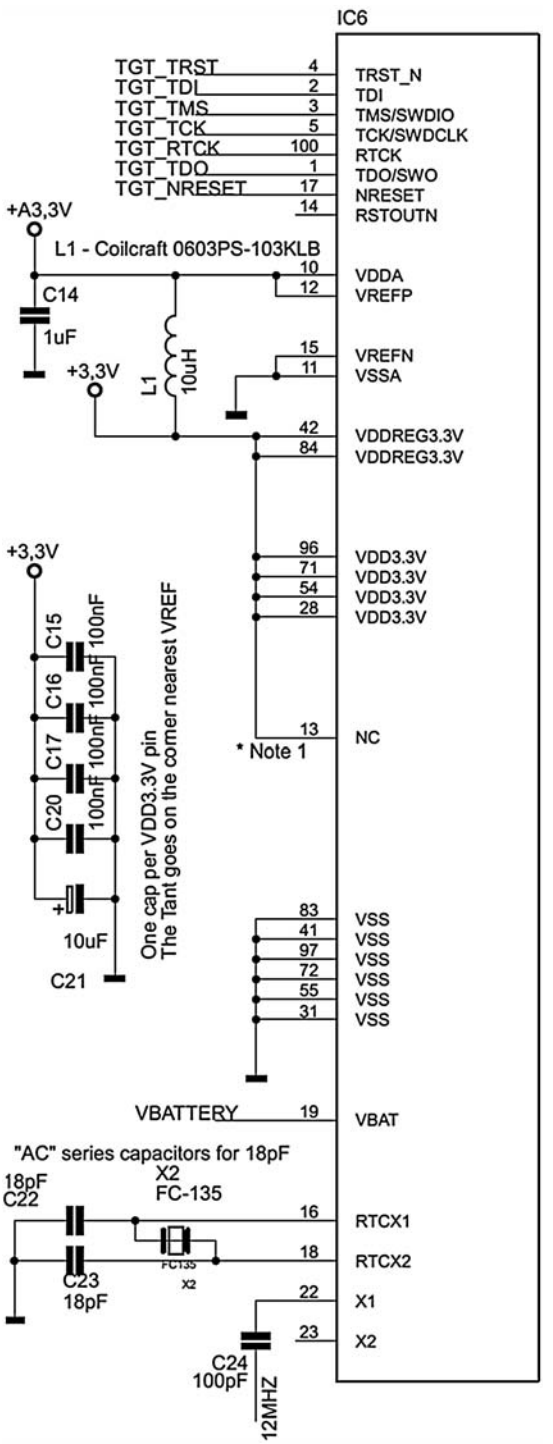


Figure 15.2
LPC1768 internal connections on the mbed. *Reproduced with permission from ARM Limited. Copyright © ARM Limited.*

15.2 Clock Sources and Their Selection

15.2.1 Some Clock Oscillator Preliminaries

An essential part of the microcontroller system is the clock oscillator, a continuous square wave which relentlessly drives forward most microcontroller action. Besides this, it is also the basis of any accurate time measurement or generation. Clock oscillators can be based on resistor–capacitor (R-C) networks, or ceramic or crystal resonators; the designer should always be aware of the options that are available, and their relative advantages.

A popular R-C oscillator circuit is shown in Fig. 15.3A. Here, the capacitor C is charged from the supply rail through resistor R. The voltage at the R-C junction connects to the input of a logic gate. When the input threshold of the logic gate is passed, its output goes high and switches on a transistor. This rapidly discharges the capacitor. The gate output goes low again, and the capacitor restarts its charging. This action continues indefinitely. Operating frequencies depend on values of R and C, and the input thresholds of the gate. This incidentally has a *Schmitt trigger* input stage, shown by that little symbol inside the gate symbol. A Schmitt trigger tidies up a poorly defined logic signal, such as appears at the R-C junction, and its *hysteresis* action ensures good discharge of the capacitor.

The R-C circuit is the cheapest form of clock oscillator available and is widely used. Moreover, all components can be integrated on-chip and it is resistant to mechanical shock. Because of this, an on-chip R-C oscillator is highly reliable. With some microcontrollers, the resistor and capacitor can be placed externally (though not with the LPC1768), so the operating frequency can be approximately selected. In this case, only one IC interconnection pin is required. The main disadvantage of the circuit, which in

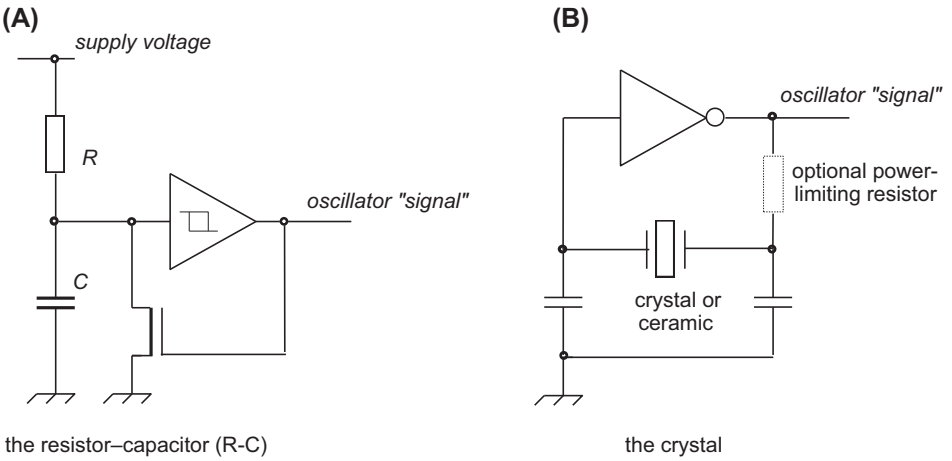


Figure 15.3
Oscillator circuits (A) the resistor–capacitor (R-C) (B) the crystal.

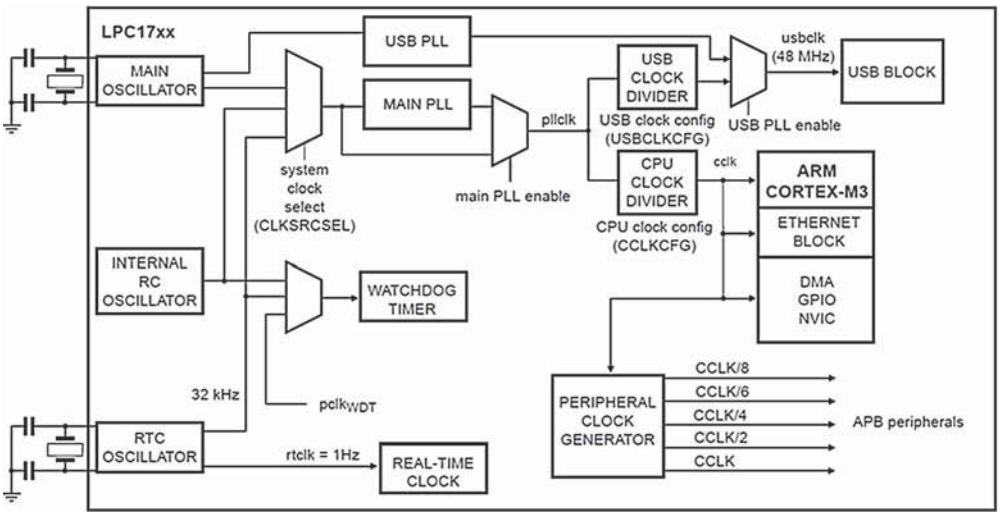
many situations is decisive, is that the precise frequency of oscillation is unpredictable, and that it drifts with temperature and time. Recent technical advances have improved this situation, and many on-chip R-C oscillators are now surprisingly stable, though never as good as a crystal oscillator, described next.

A quartz *crystal oscillator* is based on a very thin slice of crystal. The crystal is piezoelectric, which means that if it is subjected to mechanical strain then a voltage appears across opposite surfaces, and if a voltage is applied to it, then it experiences mechanical strain, i.e., it distorts slightly. The crystal is cut into shape, polished, and mounted so that it can vibrate mechanically, the frequency depending on its size and thickness. The resonant frequency of vibration is very stable and predictable. If electrical terminals are deposited on opposite surfaces of the crystal, then due to its piezoelectric property, vibration can be induced and then sustained electrically, by applying a sinusoidal voltage at the appropriate frequency. A suitable circuit is shown in Fig. 15.3B. A crystal can never be integrated on an IC, so must always be external, though the logic gate can be on-chip. It is very common to see an external crystal connected through two terminals to a microcontroller, with the two associated low-value capacitors. You can see this applied for the mbed RTC oscillator in Fig. 15.2, connected to pins RTCX1 and RTCX2.

15.2.2 LPC1768 Clock Oscillators and the mbed Implementation

Fig. 15.4 shows how the clock sources within the LPC1768 are distributed. There are three sources available, each of which can be used in many ways. They are the main oscillator (an external crystal), the internal R-C oscillator, and the RTC oscillator—another crystal, but of low frequency. All three sources are seen to the left of the diagram. The main oscillator, based on the circuit of Fig. 15.3B, can operate between 1 and 25 MHz. However, this oscillator circuit can also act in *slave mode*, in which case an external clock signal can be connected via a capacitor to the XTAL1 pin. The internal R-C oscillator runs at a nominal frequency of 4 MHz. Fig. 15.4 shows that it *can* be used as the main clock source, though it will lack the precision required for certain time-based activity. It is also available to drive the WDT. The RTC oscillator is generally expected to be 32.768 kHz. This is 2^{15}, which divides tidily down to a 1-Hz, one pulse per second, signal. This is useful as the basis for RTC applications, where seconds, minutes, hours, and days are counted.

The three clock sources all enter a multiplexer, one of several "wedge-shaped" symbols in the circuit. This is effectively a selector circuit, controlled by a couple of bits from the Clock Source Select Register **(CLKSRCSEL)**; we see more of this soon. The output of this multiplexer can go through a *phase-locked loop* (PLL), a clever circuit which can *multiply* frequencies. Hence it could take a 10-MHz oscillator input, multiply by 8, to give an output of 80 MHz. This is useful—indeed essential, because it allows an internal clock



Note: MAINPLL is also called PLL0

Key
APB: Advanced Peripheral Bus
DMA: Direct memory Access
GPIO: General Purpose Input/Output
NVIC: Nested Vectored Interrupt Controller
PLL: Phase Locked Loop
RTC: Real Time Clock
USB: Universal Serial Bus

Figure 15.4
The LPC1768 clock circuit. *Image courtesy of NXP.*

frequency *higher* than the frequency a crystal can supply. The main PLL is called PLL0; we return to it in Section 15.2.4. Another multiplexer then selects whether the PLL is used or not. The resulting signal, **pllclk**, is available for both the USB and the CPU. Following the CPU path, it can then be *divided,* producing a signal called **cclk**. Why divide the frequency, when we have just been talking about multiplying it? Different combinations of multiplication and division allow a very wide range of oscillator frequencies to be selected. The **cclk** signal goes to four further system blocks, including the Cortex core itself, and the peripherals. Details of the peripheral clock generator have already been covered Section 14.4.2. Note that the maximum permissible frequency for the LPC1768 CPU is 100 MHz.

How does the mbed use the three clock sources available to the LPC1768? Fig. 15.2 will help to answer this question. It's easy to see the RTC crystal, connected to pins 16 and 18. A quick check of the data shows that the FC-135 crystal, identified in the diagram, runs at 32.768 kHz, as expected. The internal R-C oscillator is entirely on-chip, so has no visibility here; yet Section 15.3 will show what an important role it has. The main oscillator is connected to pin 22. The mbed has a 12-MHz oscillator source, shared by several devices, which links to the

LPC1768 through C24. We know, however, that the mbed itself runs at 96 MHz (Appendix C). We must deduce that the incoming 12 MHz is multiplied by 8 to achieve the 96 MHz.

Would it be possible to change the operating frequency of the mbed? Answers to this are explored in the next few sections. Fig. 15.4 suggests that there are three ways to manipulate the frequency of the main clock: by selecting a different clock source, by changing the PLL setting, or by changing the divider. Control of the PLL carries some complexities, so let's leave that to a later section. The simplest clock frequency change can be made by changing the CPU clock divider; let's make that our next topic of investigation.

15.2.3 Adjusting the Clock Configuration Register

The CPU Clock Divider block, seen in Fig. 15.4, is controlled by the Clock Configuration register **CCLKCFG,** shown in Table 15.2. It's easy to see that the frequency of **pllclk** is divided by the number held in the lower 7 bits of this register, plus one; this produces **cclk**. The value held in this register is accessed and adjusted, in Program Example 15.1.

Like Program Example 14.1, Program Example 15.1 approximately replicates the original "blinky" program. However, it starts by resetting the value of **CCLKCFG**, so that the

Table 15.2: LPC1768 clock configuration register CCLKCFG.

Bit	Symbol	Value	Description	Reset Value
7:0	CCLKSEL		Selects the divide value for creating the CPU clock (CCLK) from the PLL0 output.	0x00
		0 to 1	Not allowed, the CPU clock will always be greater than 100 MHz.	
		2	PLL0 output is divided by 3 to produce the CPU clock.	
		3	PLL0 output is divided by 4 to produce the CPU clock.	
		4	PLL0 output is divided by 5 to produce the CPU clock.	
		255	PLL0 output is divided by 256 to produce the CPU clock.	
31:8	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

program then runs with a changed clock frequency. The program uses techniques introduced in Chapter 14 to access the microcontroller control registers.

```
/*Program Example 15.1 Adjusts clock divider through register CCLKCFG,
with trial blinky action                              */

#include "mbed.h"     //keep this, as we are using DigitalOut
DigitalOut myled(LED1);
#define CCLKCFG (*(volatile unsigned char *)(0x400FC104))

// function prototypes
void delay(void);

int main() {
  CCLKCFG=0x00000005; // divider divides by this number plus 1
  while(1) {
    myled = 1;
    delay();
    myled = 0;
    delay();
  }
}

void delay(void){         //delay function.
  int j;                     //loop variable j
  for (j=0;j<5000000;j++) {
    j++;
    j--;                 //waste time
  }
}
```

Program Example 15.1: Changing the CPU Clock Divider settings

Exercise 15.1

Compile and download Program Example 15.1 to an mbed, first with the line `CCLKCFG=0x00000005;`commented out. The clock frequency will not be changed. Carefully record how many times the led flashes in 30 s. Now enable the divider code line, and run the program several times, with different values entered for CCLKCFG, initially in the range 2 to 9. For each record how many times the led flashes in 30 s.

1. Deduce what is the approximate duration of the delay function.
2. Which setting of CCLKCFG most closely matches your original reading?

You *may* want to look forward and do Exercise 15.7 at the same time; this applies the same program.

■

15.2.4 Adjusting the Phase-Locked Loop

The PLL has already been mentioned as a circuit which can multiply frequencies. The main PLL of the LPC1768, PLL0, is actually made up of a divider followed by the PLL. Hence (perhaps strangely), it can divide frequencies as well as multiply them. Different combinations of multiply and divide give a huge range of possible output frequencies, which can be extremely useful in some situations. As its name suggests, a PLL needs to "lock" to an incoming frequency. However, it only locks if conditions are right, and it may take finite time to do this. These conditions include the requirement that the input to PLL0 must be in the range 32 kHz to 50 MHz. It's interesting to see therefore that the PLL can be used to multiply up the RTC frequency if required.

Full use of the PLL0 subsystem is complex and requires a very careful reading of relevant sections of the LPC1768 user manual (Ref. [5] of Chapter 2). However, we can still gain some useful insights by accessing its features in a limited way. The PLL is controlled by four registers, outlined in Table 15.3. We can readily see that the PLL can be enabled and connected through **PLL0CON**, with multiply and divide values set through **PLL0CFG**. Because it sits in the path of the main oscillator, and because PLLs sometimes act in a way which can be described as temperamental, there are two further important registers. The Feed Register, **PLL0FEED**, is a safety feature which blocks accidental changes to **PLL0CON** and **PLL0CFG**. A valid "feed sequence" is required before any update can be configured. Once implemented, changes can be tested in **PLL0STAT**. This further carries the important **PLOCK0** bit, which tests whether successful lock has been achieved.

The setup sequence for PLL0 is defined in the User Manual; it must be followed precisely. Program Examples 15.2 and 15.3 illustrate aspects of this. The **main( )** function immediately sets about disconnecting and turning off the PLL, "feeding" the PLL control as required. The mbed then runs with the PLL disabled and bypassed.

```
/*Program Example 15.2 Switches off PLL0, with blinky action
                                                    */
#include "mbed.h"
DigitalOut myled(LED1);
#define CCLKCFG (*(volatile unsigned char *)(0x400FC104))
#define PLL0CON (*(volatile unsigned char *)(0x400FC080))
#define PLL0FEED (*(volatile unsigned char *)(0x400FC08C))
#define PLL0STAT (*(volatile unsigned char *)(0x400FC088))

// function prototypes
void delay(void);

int main() {
  // Disconnect PLL0
  PLL0CON &= ~(1<<1); // Clears bit 1 of PLL0CON, the Connect bit
```

```
  PLL0FEED = 0xAA;    // Feed the PLL. Enables action of above line
  PLL0FEED = 0x55;    //
  // Wait for PLL0 to disconnect. Wait for bit 25 to become 0.
  while ((PLL0STAT & (1<<25)) != 0x00);//Bit 25 shows connection status
  // Turn off PLL0; on completion, PLL0 is bypassed.
  PLL0CON &= ~(1<<0); //Bit 0 of PLL0CON disables PLL
  PLL0FEED = 0xAA;    // Feed the PLL. Enables action of above line
  PLL0FEED = 0x55;
  // Wait for PLL0 to shut down
  while ((PLL0STAT & (1<<24)) != 0x00);//Bit 24 shows enable status

  /****Insert Optional Extra Code Here****
         to change PLL0 settings or clock source.
  **OR** just continue with PLL0 disabled and bypassed*/

  //blink at the new clock frequency
  while(1) {
    myled = 1;
    delay();
    myled = 0;
    delay();
  }
}

void delay(void){         //delay function.
  int j;                     //loop variable j
  for (j=0;j<5000000;j++) {
    j++;
    j--;                  //waste time
  }
}
```

Program Example 15.2: Switching off the main PLL

Adapted from "PLL0 config script", Hugo Zijlmans, https://developer.mbed.org.

Exercise 15.2

Compile and download Program Example 15.2 to an mbed. Carefully measure how many times the LED flashes in one minute. It will be very slow. Can you explain the beats per minute that you measure for this, comparing with the first value recorded in Exercise 15.1?

You *may* want to look forward and do Exercise 15.7 at the same time, this applies the same program.

■

Table 15.3: PLL0 control registers.

Name	Description	Access	Address
PLL0CON	Control Register. Holding register for updating PLL0 control bits. Values written to this register do not take effect until a valid PLL0 feed sequence has taken place. There are only 2 useful bits: bit 0 to enable; PLL0, bit 1 to connect. Connection must only take place after the PLL is enabled, configured, and locked.	Read/Write	0x400F C080
PLL0CFG	Configuration Register. Holding register for updating PLL0 configuration values. Bits 14:0 hold the value for the frequency multiplication, less one; Bits 23:16 hold the value for the predivider, less one. Values written to this register do not take effect until a valid PLL0 feed sequence has taken place.	Read/Write	0x400F C084
PLL0STAT	Status Register. Read-back register for PLL0 control and configuration information. If **PLL0CON** or **PLL0CFG** has been written to, but a PLL0 feed sequence has not yet occurred, they will not reflect the current PLL0 state. Reading this register provides the actual values controlling PLL0, as well as the PLL0 status. Bits 14:0 and bits 23:16 reflect the same multiply and divide bits as in **PLL0CFG**. Bits 24 and 25 reflect the two useful bits of **PLL0CON**. When either is zero, PLL0 is bypassed. When both are 1, PLL0 is selected. Bit 26, **PLOCK0**, gives the lock status of the PLL.	Read Only	0x400F C088
PLL0FEED	Feed Register. Correct use of this register enables loading of the PLL0 control and configuration information from the **PLL0CON** and **PLL0CFG** registers into the shadow registers that actually affect PLL0 operation. The required feed sequence is 0xAA followed by 0x55.	Write Only	0x400F C08C

Based on Table 18 and following, of LPC1768 User Manual.

Exercise 15.3

Adjust Program Example 15.2 to set a multiply value for the PLL. To do this, apply the code fragment of Program Example 15.3, inserting it before the **while ( )** loop in Program Example 15.2. Basic information on setting **PLL0CFG** is given in Table 15.3. You can try your own experimental values, using information supplied in this chapter. To work with a deeper understanding of the limits and possibilities, refer to Section 4.5.10 of Ref. [2] of Chapter 2. In each case, measure the LED blink rate, and try to correlate it to the setting you have made.

■

```
// Set PLL0 multiplier
PLL0CFG = 07;  //arbitrary multiply value, divide value left at 1
PLL0FEED = 0xAA; // Feed the PLL
PLL0FEED = 0x55;
// Turn on PLL0
PLL0CON |= 1<<0;
PLL0FEED = 0xAA; // Feed the PLL
PLL0FEED = 0x55;
// Wait for main PLL (PLL0) to come up
while ((PLL0STAT & (1<<24)) == 0x00);
// Wait for PLOCK0 to become 1
while ((PLL0STAT & (1<<26)) == 0x00);
// Connect to the PLL0
PLL0CON |= 1<<1;
PLL0FEED = 0xAA; // Feed the PLL
PLL0FEED = 0x55;
    while ((PLL0STAT & (1<<25)) == 0x00); //Wait for PLL0 to connect
```

Program Example 15.3: Code fragment to set PLL0 multiplier

Adapted from "PLL0 config script", Hugo Zijlmans, https://developer.mbed.org.;

15.2.5 Selecting the Clock Source

If the clock source is to be changed, through the input multiplexer top left of Fig. 15.4, it must be done with PLL0 shutdown. This change is controlled by the Clock Source Select Register, **CLKSRCSEL**, with details shown in Table 15.4.

Exercise 15.4

Write a program which turns off PLL0, changes the clock source, and starts up PLL0 again. To do this, combine Program Examples 15.2 and 15.3, and apply the information in Table 15.4. Try this for both IRC and RTC sources.

■

Table 15.4: Clock source select register, CLKSRCSEL.

Bit	Symbol	Value	Description	Reset Value
1:0	CLKSRC		Selects the clock source for PLL0 as follows:	0
		00	Selects the internal R-C oscillator as the PLL0 clock source (default).	
		01	Selects the main oscillator as the PLL0 clock source. **Remark:** Select the main oscillator as PLL0 clock source if the PLL0 clock output is used for USB or for CAN with baud rates > 100 kBit/s.	
		10	Selects the RTC oscillator as the PLL0 clock source.	
		11	Reserved, do not use this setting.	
		Warning: Improper setting of this value, or an incorrect sequence of changing this value may result in incorrect operation of the device.		
31:2	—	0	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

15.3 Reset

At certain times in its use, any microcontroller is required to start its program from the beginning, the most obvious being when power is applied. At this moment, it also needs to put all of its control registers into a known state, so that peripherals are safe and initially disabled. This "ready-to-start" condition is called *reset*. Apart from power-up, there are other times when this is needed, including the possibility that the user may want to force a reset if a system locks or crashes. The CPU starts running its program when it leaves the reset condition. In an advanced processor like the LPC1768, the user code is preceded by some "boot code," hard-wired into the processor, which undertakes preliminary configuration. How reset is implemented in any microcontroller, and what it actually does, is an important part of any microcontroller-based design.

15.3.1 Power-On Reset

The moment that power is applied is a critical one for any embedded system. Both the power supply and the clock oscillator take finite time to stabilize, and in a complex system power to different parts of the circuit may stabilize at different times. Clearly, this takes some careful handling. How can the start of program execution be delayed until power has settled? How can a complex system be kept in a safe state while all its subsystems initialize? This will only happen if explicit circuitry is built in to detect power-up, preset control registers, and force a program restart.

The LPC1768 has an interesting but complex circuit to manage its reset routine, which you can see in the user manual (Fig. 4 of Ref. [5] of Chapter 2). The effect of that circuit, when power is applied, is summarized in Fig. 15.5. Here, we see the power supply voltage rising from 0 to 3.3 V; as a consequence, the internal R-C oscillator starts oscillating. An internal timer starts measuring a fixed delay of 60 µs from when the supply voltage reaches a valid threshold; this gives time for the oscillator to stabilize. The internal reset signal initially stays low, disabling all activity, and forcing all registers to their reset condition. On completion of the 60 µs delay, however, this reset signal is set high, and processor activity can start. Summing all the delays, one

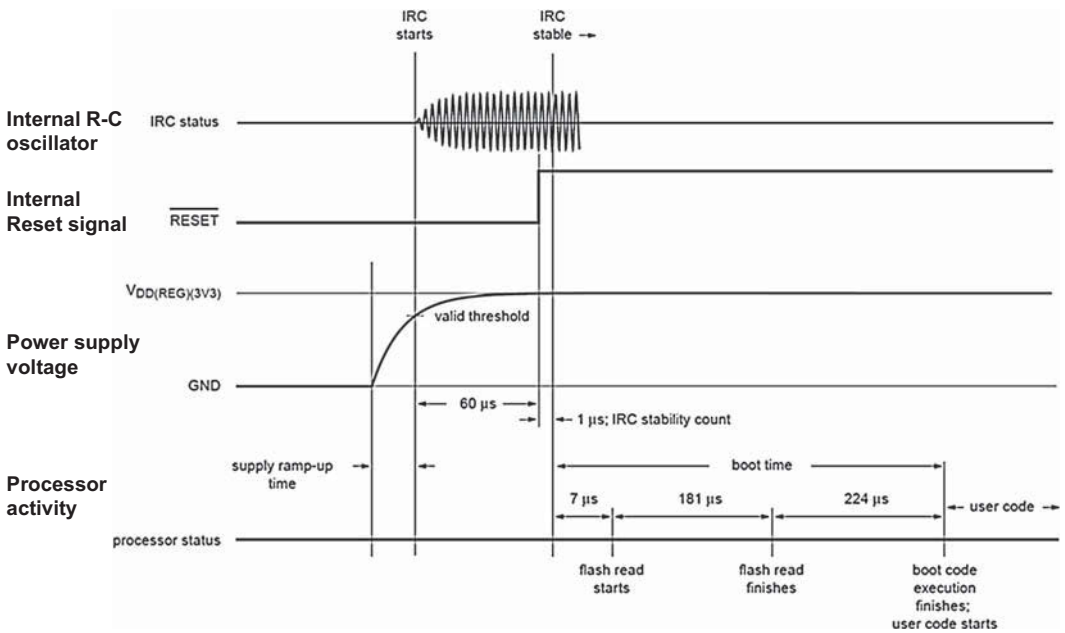


Figure 15.5
LPC1768 internal signals during start-up. *Image courtesy of NXP.*

can see that it's around 0.5 ms after power is switched on, that the user code starts to execute.

15.3.2 Other Sources of Reset

Three further sources of reset are described in overview here. The first is implemented in the mbed, as outlined. We describe the other two in overview only, and don't attempt any experimentation.

External reset

The LPC1768 has an external reset input. As long as this is held low, the microcontroller is held in reset. When it is taken high, a sequence is followed very similar to the power-on reset described above. If the pin is taken low while the program is running, then program execution stops immediately and the microcontroller is forced back into Reset mode. This allows an external push-switch to be connected, so that a user can force a reset if needed. For a product on the market, it's unlikely that this facility would be used—it's almost an expression of mistrust in the design if a reset is made available. In a prototype environment, however, it can be useful, as program crashes are more likely to occur. As we know, the mbed has a reset button. This is, however, connected to the interface microcontroller (Fig. 2.2), which can force a reset to the LPC1768 itself (pin 17, Fig. 15.2).

Watchdog timer

A common failure mode of any computer-based system is for the computer to lock up, and cease all interaction with the outside world. For most embedded systems, this is unacceptable. An uncompromising solution to the problem is the WDT, which *resets the processor if the WDT is ever allowed to overflow*. The WDT runs continuously, counting toward its overflow value. It is up to the programmer to ensure that this overflow never ever happens (in normal program operation). This is done by including periodic WDT resets throughout the program. *If* the program crashes, then the WDT overflows, the controller resets, and the program starts from its very beginning, with the program counter set to its reset value. A WDT overflow causes a reset very similar to the power-on reset described above. The WDT for the LPC1768 is shown in Fig. 15.4, with its three possible clock sources. Two of these are direct from R-C and RTC oscillators. The third, $\mathbf{pclk_{WDT}}$, is derived from the Peripheral Clock Generator, and has already appeared in Table 14.5.

Brownout detect

An awkward failure condition for an embedded system is when the power just dips, and then returns to normal. This is called a *brownout*, and is illustrated in Fig. 15.6. A

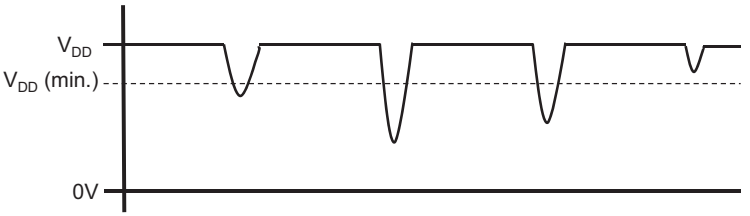


Figure 15.6
Voltage "brownouts."

brownout won't be detected as a full loss of power, and may not be noticed at all. However, that momentary loss of full power could cause partial system failure, for example, some settings becoming corrupted, resulting in possible malfunction later. Brownouts can be due to many things, including the switching on of a motor or other actuator, a supply dip due to poor power supply design, or severe interference picked up on the supply lines. Like many microcontrollers, the LPC1768 has a brownout detect capability, which must be enabled by the user. It is not normally enabled in the mbed. Detection of brownout, when enabled, causes a reset very similar to the power-on reset described above.

15.4 Toward Low Power

15.4.1 How Power Is Consumed in a Digital Circuit

Since the large-scale adoption of digital electronics, a number of logic families have competed for the designer's attention. Each provides modular electronic circuits which implement the essential logic functions of AND, NAND, OR, NOR, and so on; and each has its own advantages, for example, in terms of speed or power consumption. The only logic family that is suitable for low-power circuits is called *complementary metal oxide semiconductor* or CMOS for short. CMOS technology has transformed our lives, as it is the basis for the mobile phone, laptop, and any other portable electronic device. To understand how to minimize the power consumption of CMOS, it is useful to have some understanding as to how it consumes that power. Let's look at a simple circuit. An inverting CMOS buffer is shown in different guises in Fig. 15.7. All other CMOS logic gates are extensions of this simple circuit, and a microcontroller is essentially made up of millions of these circuits, grouped together to form all the essential subsystems. So if we can understand power consumption in one, we have a key to what goes on in much larger systems.

The CMOS buffer is simply made up of two transistors, one fabricated on n-type and the other on p-type semiconductor substrates (that's the "complementary" bit). Most of the time, each transistor acts as a switch; to aid understanding, the circuit can first be

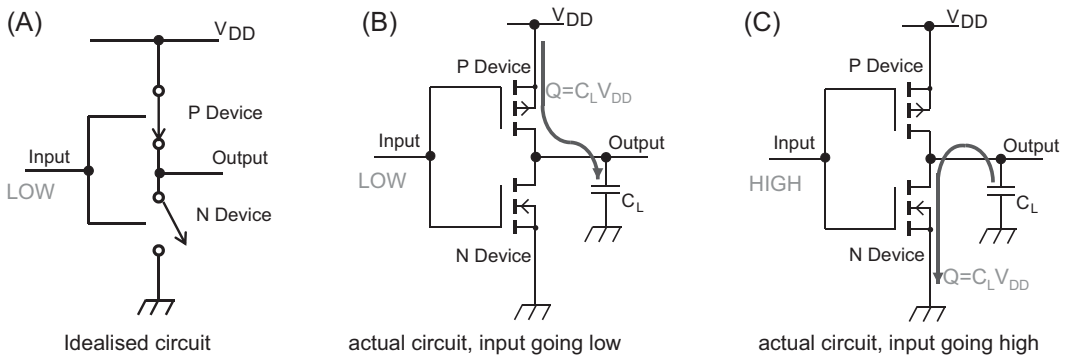


Figure 15.7
Power consumption in a complementary metal oxide semiconductor inverter. (A) Idealized circuit (B) actual circuit, input going low (C) actual circuit, input going high.

simplified to Fig. 15.7A. When—as is shown—the input is low (i.e., at Logic 0), then the lower transistor is switched *off* (i.e., the switch is open), and the upper one is closed, or switched *on*. The output is therefore connected to the supply voltage, i.e., it is high, or at Logic 1. If the input flips to Logic 1, then the transistors change accordingly. What is important is that in either of these states there is no current path from the supply to ground. This, in a nutshell, is why CMOS consumes such little power. The actual circuit is shown in Fig. 15.7B and C; don't worry about the curvy arrows just yet.

It's when the input is changing state that awkward things happen, as far as power consumption is concerned. First of all, there's always "stray" capacitance in the circuit. This is symbolized as C_L in circuits Fig. 15.7B and C. This capacitance is due to the interconnecting wires, and also the very structure of the CMOS transistors themselves (the parallel lines in the symbol give the clue). So on every change of input, the stray capacitance is charged or discharged, hence taking current from the supply. This is symbolized by the curvy arrows, which show C_L being charged as the input goes low (hence output goes high), and discharged as the output goes low. This may not be too bad for one logic gate going through one transition, but if millions of them are doing it millions of times a second, a lot of charge can get dumped to ground. This process is called *capacitive* power consumption. To add to that, as it changes state, the input voltage passes a middle region where both devices are momentarily partially turned on. A tiny pulse of current can flow straight through them at this instant; this is called *shoot-through* power consumption. Finally, there's a tiny bit of leakage current which flows continuously, often so small as to be negligible; this is called *quiescent* power consumption.

It is beyond the scope of this book to fully analyze this power consumption behavior (for more detail try Chapter 10 of Ref. [1] of Chapter 1), but it is useful to look briefly at Eq. (15.1), which describes it. Here the total supply current I_T is made up of the small and

moderately constant quiescent current I_Q, plus a term which depends on supply voltage V_{DD}, clock frequency f, and a capacitive term, C_{eq}. This "equivalent" capacitance is a complex thing, which in brief lumps together interconnection capacitance in the IC, capacitance due to external interconnections, plus an equivalent capacitance which represents the shoot-through behavior. To convert this current consumption to power consumption, simply multiply each side by V_{DD}.

$$I_T = I_Q + \{V_{DD} \times fC_{eq}\} \tag{15.1}$$

Neglecting for a moment the very small quiescent current, this equation shows that supply current is effectively proportional to supply voltage, switching frequency, and equivalent capacitance. If we can tame these things, we're on our way to taming power consumption!

15.4.2 A Word on Cells and Batteries

If we are considering power optimization, then it is almost certainly because the plan is to run from battery power. Battery technology and behavior is in itself is a huge topic, but we introduce some key aspects here. This helps to evaluate current consumption in a microcontroller. Correctly speaking, a battery is made up of a collection of cells.

Cells are classified either as primary (nonrechargeable), or as secondary (rechargeable). They are based on a variety of metal/chemical combinations, each of which has special characteristics in terms of energy density, whether rechargeable or not, and other electrical characteristics. Primary cells like alkaline tend have the highest energy density, so are widely used for applications of greatest power demand. They are also used for low-power or occasional applications, where replacement is infrequent. Of course in applications like the mobile phone or laptop, it is unthinkable to consider primary cells, and a range of sophisticated and specialist rechargeable batteries exist. Cells are available in a wide variety of packages. These include the more traditional cylinder formats, now most seen in AA or AAA version, a range of button cells, and numerous specialist batteries for laptop and mobile phones. Two familiar types are shown in Fig. 15.8, with some basic characteristics given in Table 15.5.

The two most important electrical characteristics of a cell are terminal voltage and capacity; the latter generally measured in Amp-hours (Ah) or milliamp hours (mAh). The inference is that a battery of 500 mAh can sustain a 500 mA current for 1 h or a 1 mA current for 500 h. In reality the situation is not so simple; batteries do tend to recover between periods of use, and display somewhat different capacities depending on load current, temperature, battery age, and a number of other things. Multiplying the Amp-hour capacity by the battery terminal voltage gives an approximate value for the actual energy stored, in Watt-hours. Thus the PP3 cell in Table 15.5, which appears to have a small

Figure 15.8
Example cells.

capacity of only 550 mAh, has an energy capacity of 4.95 Watt-hours, compared to the 4.05 Watt-hour capacity of the AA cell.

As a simple example rather relevant to this book, suppose the battery-powered circuit of Fig. 4.10B is drawing an average current of 160 mA, and the cells shown are Procell type MN1500. Each cell is delivering the same current, so an approximate value for the cell life would be given the mAh capacity, divided by the load current, also in mA; i.e., 2700/160, or just under 17 h.

Suppose instead that a device was to be powered from a V303 cell, and a year's continuous operation is required. This cell has a much smaller capacity, of 160 mAh. There are 365 × 24 h in a year, so the maximum allowable current is 160/(365 × 24) mA, or 18.3 μA.

Simple calculations like these allow useful estimations to be made relating battery life to circuit current consumption, and give a feel of magnitudes that can be expected. The calculations can with care be adapted to more realistic situations, where the battery use is intermittent, and/or varying.

Table 15.5: Example cell data.

Manufacturer	Technology	Shape/Package	Nominal Terminal Voltage (V)	Capacity (mAh)
Varta	Silver Oxide	V301, Button	1.55	96
Varta	Silver Oxide	V303, Button	1.55	160
Procell	Alkaline	AAA cylinder	1.5	1175
Procell	Alkaline	AA cylinder	1.5	2700
Procell	Alkaline	PP3	9.0	550

15.5 Exploring mbed Power Consumption

The designers of the mbed would never claim that the device was designed for low power, yet it does provide a good opportunity to study power supply in an embedded system, and to explore ways of reducing that power. It's interesting and revealing to set up a simple current measurement circuit, and to do the exercises which follow.

Exercise 15.5

Download Program Example 2.1 (or indeed any mbed program which does not require external connections) to an mbed. Disconnect the USB cable and power your circuit as shown in Fig. 15.9. Use a battery pack or bench DC supply, set at around 6 V. The supply should link to the VIN mbed pin (pin 2), with an ammeter—for example, a digital multimeter on its 200 mA range—inserted between supply positive and the VIN pin. The mbed will draw an approximately constant current, so the precise supply voltage does not matter, only that it lies in the specified range of 4.5 to 9 V.

Now measure and record the current supplied to the mbed. You should find this somewhere in the region of 140 mA.

Complete the calculation of Quiz question 5.

■

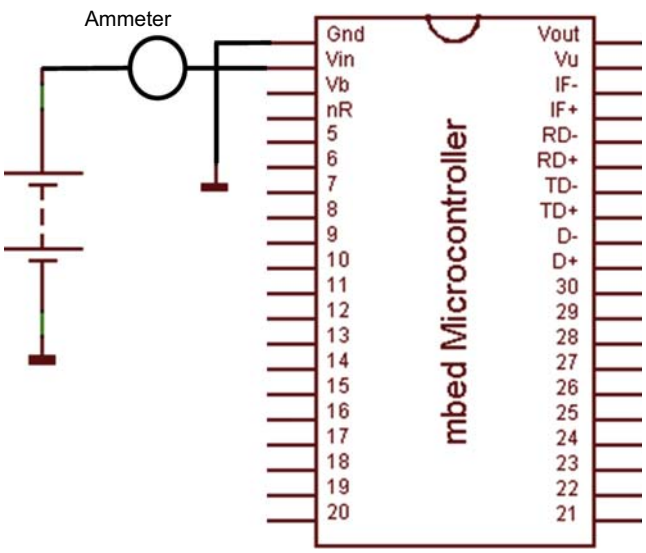


Figure 15.9
Measuring mbed current consumption

15.5.1 LPC1768 Current Consumption

The current consumption characteristics of the LPC1768 itself are shown in Table 15.6, along with the LPC1769, which has the ability to run at a slightly higher clock frequency. We shall return to this table several times. The entries are self-explanatory, and reinforce the message that clock frequency dominates current consumption. The table also refers to Sleep and Power-down modes that we shall meet soon. Applying this table to the mbed, we know that it runs with a clock frequency of 96 MHz, with PLL enabled. Hence, we could estimate that the mbed LPC1768 is taking a supply current just under 42 mA, let's say 40 mA. However, the values shown are with all peripherals disabled, so in practice the consumption will be somewhat higher.

The information just acquired, and the result of Exercise 15.5, are sobering indeed. If our estimate of 40 mA for the LPC1768 is true (and we have still to confirm this), then the circuitry external to the microcontroller is taking around 100 mA! These are big figures,

Table 15.6: LPC1768 current consumption characteristics.

$I_{DD(REG)(3V3)}$	regulator supply current (3.3 V)	active mode; code `while(1){}` executed from flash; all peripherals disabled; PCLK = $^{CCLK}/_{8}$				
		CCLK = 12 MHz; PLL disabled	[6][7]	-	7	-	mA
		CCLK = 100 MHz; PLL enabled	[6][7]	-	42	-	mA
		CCLK = 100 MHz; PLL enabled (LPC1769)	[6][8]	-	50	-	mA
		CCLK = 120 MHz; PLL enabled (LPC1769)	[6][8]	-	67	-	mA
		sleep mode	[6][9]	-	2	-	mA
		deep sleep mode	[6][10]	-	240	-	µA
		power-down mode	[6][10]	-	31	-	µA
		deep power-down mode; RTC running	[11]	-	630	-	nA
I_{BAT}	battery supply current	deep power-down mode; RTC running					
		$V_{DD(REG)(3V3)}$ present	[12]	-	530	-	nA
		$V_{DD(REG)(3V3)}$ not present	[13]	-	1.1	-	µA

The footnotes referenced provide the fine detail of how the measurements apply or are made, and may be accessed from Reference 2.4.

Reproduced from Table 8 of LPC1768 datasheet.

and far removed from what would be required for realistic battery supply. Let us explore how this power consumption can be reduced.

15.5.2 Switching Unwanted Things Off!

A brief glance at an mbed, or its circuit diagram, or the block diagram of Fig. 2.2, reminds you how much there is in the overall mbed circuit. And all of this is continuously powered, whether you use it or not!

Program Example 15.4 allows you to explore some aspects of managing mbed power consumption. To set up the program you need to import the **EthernetPowerControl** library file from the mbed Cookbook site, accessed through Ref. [1]. You can then power down the Ethernet physical interface (sometimes called the PHY). To switch on or off individual peripherals in the LPC1768 itself, we access the **PCONP** register, already introduced in Table 14.4.

```
/*Program Example 15.4
Powers down certain elements of the mbed, when not in use.
                  */

#include "mbed.h"
 //import next from mbed site
#include "PowerControl/EthernetPowerControl.h"

DigitalOut myled1(LED1);
DigitalOut myled4(LED4);
#define PCONP          (*(volatile unsigned long *)(0x400FC0C4))

Ticker blinker;
void blink() {
  myled1=!myled1;
  myled4=!myled4;
}

int main() {
  myled1=!myled4;
  PHY_PowerDown();  //**comment this in and out

  //Turn all peripherals OFF, except repetitive interrupt timer,
  PCONP = 0x00008000;        //which is needed for Ticker

  blinker.attach(&blink, 0.0625);
  while (1) {
    wait(1);
  }
}
```

Program Example 15.4: Switching off unused circuit sections

Adapted from “Blink_LED_with_Power_Management”, Ref. [2].

Exercise 15.6

Create a new program using Program Example 15.4. Download to an mbed, and measure supply current as in Fig. 15.9. First comment out both the lines

```
PHY_PowerDown();
```

and

```
PCONP = 0x00008000;.
```

Run the program and measure current consumption. This should be the same as found in Exercise 15.5. Enable each and then both of these lines, in each case recompiling the program, downloading, running, and reading the current consumption. Record the values measured.

You will see a significant power reduction with the PHY switched off, and a lesser reduction with the peripherals all off. We are beginning to see that power consumption can be managed.

■

Program Example 15.4 shows the benefit that can be gained by removing power to unused circuit elements. You can take it further, at the cost of a little more program complexity, for example by powering down the interface microcontroller, starting with Ref. [2].

15.5.3 Manipulating the Clock Frequency

We saw earlier in this chapter that the LPC1768 has extensive capabilities to vary the clock frequency. Now we understand one of the reasons why—we can trade off speed of execution with power consumption. A program with significant computational demands will need to run fast, and will consume more power; one with low computational demands can run slowly, and consume less power.

Exercise 15.7

Rerun both Program Examples 15.1 and 15.2 in turn. Now measure current supply to the mbed for a range of different clock frequencies. Record your results.

■

Exercise 15.8

Write a program which both powers down unwanted peripheral devices, and slows down the clock. How low can you get the current consumption?

■

It is attractive to imagine that clock speeds can be reduced at will to reduce current consumption. While this is true in principal, take care! Many of the peripherals depend on that clock frequency as well, along with their mbed application programming interface libraries, for example, for the setting of a serial port bit rate. If you're using any such peripheral, you may need to adjust its clock frequency setting to compensate.

15.5.4 LPC1768 Low-Power Modes

Manipulating the clock frequency is a simple yet effective way to influence microcontroller power consumption. However, there are other, even more effective techniques. These involve either switching off the clock altogether at times when nothing needs doing, or switching it off to certain parts of a microcontroller. For example, the CPU could be switched off, while certain peripherals were left running, if active computation wasn't needed at that time. These low-power modes are often call Sleep or Idle; though the terminology varies somewhat between different microcontrollers. The modes are entered by special instructions in the processor instruction set, and generally exited through an interrupt occurring, or a system reset. The LPC1768 uses the modes summarized below, with power consumptions given in Table 15.5.

Sleep mode: The clock to the core, including the CPU, is stopped, so program execution stops. Peripherals can continue to function. Exit from this mode can again be achieved by an enabled interrupt, or a reset. For example, the processor could put itself to sleep, and be programmed to wake when a serial port receives a new byte of data, or an external button is pressed by a user.

Deep Sleep mode: Here the main oscillator (Fig. 15.4) and PLL0 are powered down, and the output of the internal R-C oscillator disabled, so no internal clocks are available. The internal R-C oscillator can continue running, and can run the WDT, which can cause a wake-up. The 32-kHz RTC continues, and can generate an interrupt. The SRAM and processor keep their current contents, and the flash memory is in standby, ready for a quick wake-up. Wake-up is by reset, by the RTC, or by any other interrupt which can function without a clock.

Power-Down mode: This is similar to Deep Sleep, but the internal R-C oscillator and flash memory are turned off. Wake-up time is thus a little longer.

Deep Power-Down mode: In this mode, all power is switched off, except to the RTC. Wake-up is only through external reset, or from the RTC.

Exercise 15.9

Run Program Example 15.4, but replace the final code line, `wait(1);`, with `Sleep( );`. Measure the current again.

Try then replacing the `Sleep( );` line with `DeepSleep( );`. Observe what happens, and measure the current again.

■

When trying the second part of Exercise 15.8, the program will compile and download, but will then lock after the first LED is lit. The microcontroller is asleep, and we can't wake it up! Reading the description of Deep Sleep above, we can see that this program doesn't provide a mechanism to exit from this mode, as all internal clocks are disabled. The measurement of current consumption is still of interest.

Applying low-power modes other than Sleep does involve some further complexity, beyond the scope of this book. To take the topic further, and it is a very important topic, explore Refs. [2,3], and the other sources to which they lead.

15.6 Getting Serious About Low Power; the M0/M0+ Cores and the Zero Gecko

We have seen that the mbed has a comparatively high power consumption, but that it is possible to manipulate that consumption, given a knowledge of which variables have an influence. But where do we turn if a really low-power design is needed?

15.6.1 The M0 Cortex Core

As mentioned in Chapter 1, there are a number of versions of the Cortex core.
The simplest are the M0 or M0+ cores. These are specifically designed for small-scale, low-power and low-cost applications, through the use of a very simple core, with the gate count highly optimized. The formidable low-power/low-cost/low-size combination that this leads to gives many advantages in the embedded world, for example, in intelligent or networked sensors, or any small-scale or portable device. A full reference work on this core can be found in Ref. [4].

15.6.2 The EFM32 Zero Gecko Starter Kit

The EFM Zero Gecko Starter Kit is a development board based around the Silicon Labs Zero Gecko microcontroller, and forms a useful example of a truly low-power mbed-enabled design. It is based on an M0+ core, and is mbed enabled. It is designed for extreme low-power and—of great interest to us—has an onboard *Energy Profiler*, which can measure current consumption from 0.1 μA to 50 mA. The board layout is shown in Fig. 15.10.

The Silicon Labs Simplicity Studio Development Environment can be downloaded free from the company website [5]. This has a number of demo programs available on it for the kit, which can be directly downloaded to the device. A user's guide to the board is available [6].

Once a program is running, the Energy Profiler gives a power consumption reading, as shown in Fig. 15.11. This shows the energy profiler at work, running the "Analog and Digital Clock Example" from the range of demos available. A quick glance at this shows

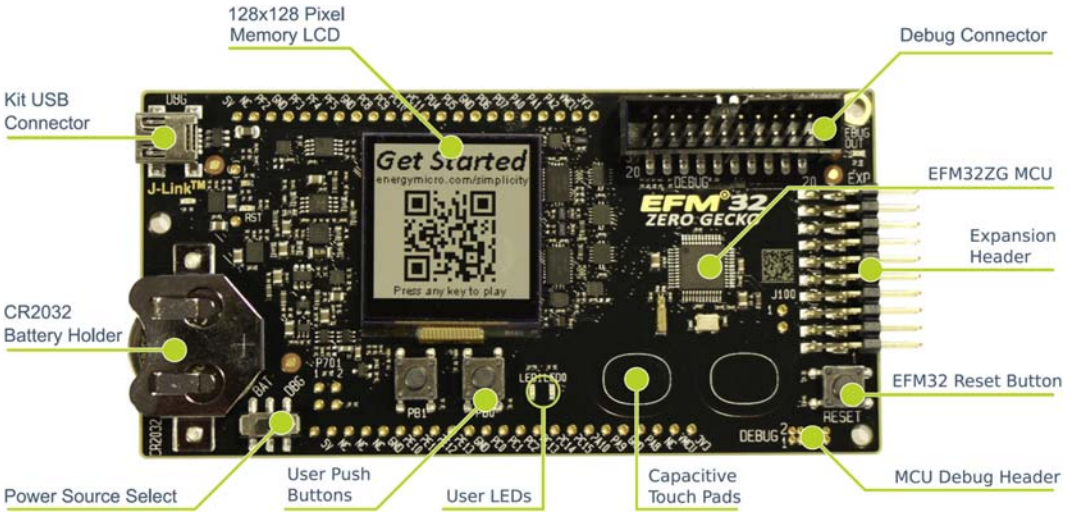


Figure 15.10
The mbed-enabled EFM32 Zero Gecko Starter Kit. *Image courtesy of Silicon labs.*

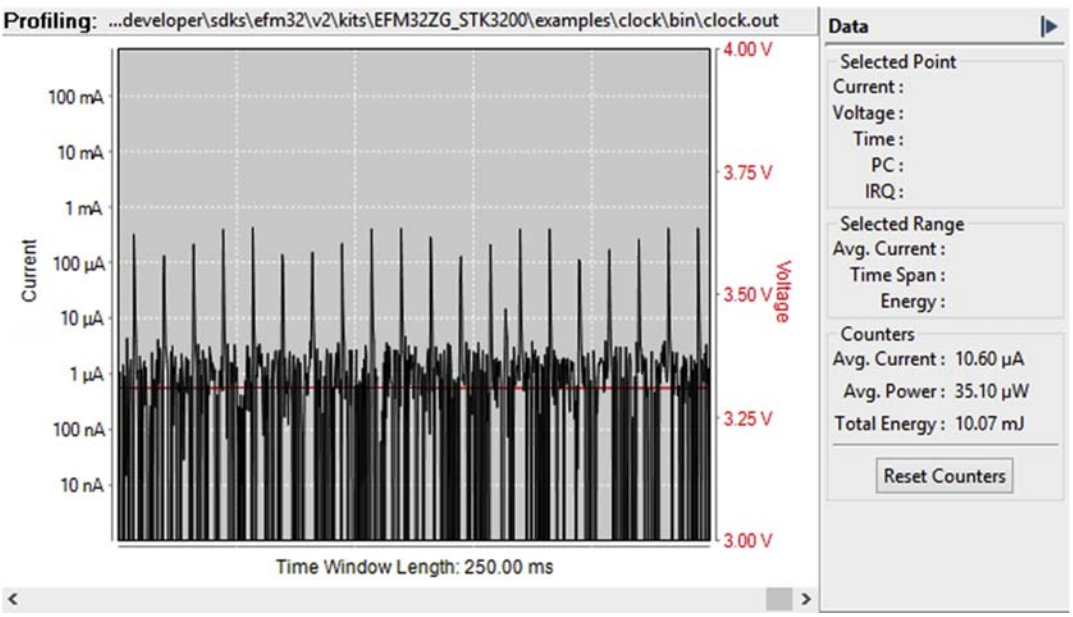


Figure 15.11
The EFM32 Zero Gecko Energy Profiler.

the diagnostic power available to the developer. Supply current (left vertical axis) and supply voltage (right vertical axis) appear. It is interesting to see even the current peaks sitting well below 1 mA. Compare this to the current measurements earlier in this chapter! In addition to this, the data block on the right shows average current and power. Most important perhaps, it shows total energy transferred; this is the ultimate piece of information we need, when running from a battery. This simple example doesn't implement *code correlation*, an option whereby power consumption can be displayed against the actual line of code being executed. However, this important capability can be implemented.

If you're working with this development board, you have the choice of doing so from the Simplicity Studio environment, from the mbed compiler, or from the range of other development environments that are on offer. It provides an excellent pathway for study in genuinely low-power applications.

Chapter Review

- The LPC1768 has clearly defined power supply demands, in terms of supply voltages and currents, with the potential for power-conscious applications.
- The mbed designers have satisfied these power supply requirements, but without the intention of minimizing power consumption.
- A range of clock sources are available to the LPC1768 and other microcontrollers; they can be applied in different ways and for different purposes, and their frequency multiplied or divided.
- The mbed power consumption can be improved, by switching off unnecessary circuit elements, adjusting clock frequency, and applying the special low-power modes, e.g., *Sleep* and *Deep Sleep*. Even with this optimization, current consumption remains too high for effective battery-powered applications.
- For competitive low-power design, circuits must be designed specifically and rigorously for that purpose. The EFM32 Zero Gecko is one example where this has been achieved, to good effect.

Quiz

1. Name two advantages and two disadvantages of R-C and quartz oscillators.
2. Following reset, the LPC1768 always starts running from the internal R-C oscillator. Why is this?
3. In a certain application, the main oscillator in an LPC1768 application is running at 18.000 MHz, the PLL multiplies by 8, and the lower 7 bits of register **CCLKCFG** are set to 5. What is the frequency of **cclk**?

4. A certain logic circuit is powered from 3.0 V. It has a quiescent current of 120 nA, and an "equivalent capacitance" in the circuit of 56 pF. Applying Eq. (15.1), what is its current consumption when the clock frequency is 1 kHz, 1 MHz, and when it is not clocked at all?
5. An mbed is found to draw 140 mA, when powered from 4 AAA cells in series, each of capacity 1175 mAh. Approximately how long will the cells last if they run continuously?
6. Access these answers from the LPC1768 User Manual:
   a. Explain why only the main oscillator may be used as clock source for the USB.
   b. There is a required range of *output* frequency for PLL0. What is it?
7. Propose situations where each of the LPC1768 low-power modes (Sleep, Deep Sleep, and so on) can be used effectively.
8. Power consumption to a digital circuit is being carefully monitored. It is found that connecting a long cable to a digital interface marginally increases the power consumption, even though nothing is connected at the far end of the cable. Why is this?
9. Through Internet search or otherwise, identify the main differences between the M0 and the M0+ processor cores.
10. A designer wishes to estimate power consumption characteristics of a new product based on the LPC1769, and applies the current consumption data of Table 15.5. A 3.3-V battery is available, with capacity 1200 mAh. She anticipates a behavior whereby the processor will need to wake once per minute to perform a task, made up of two parts. In the first part, the processor must run with a clock speed of 120 MHz, PLL enabled for 200 ms. It then runs at 12 MHz, PLL disabled, for 400 ms. For the rest of the time, it is in power-down mode. For the purposes of this question, the current taken by the rest of the circuit can be assumed to be negligible.
    a. Estimate the average current drawn from the battery.
    b. Estimate the battery life.

References

[1] M. Wei, Power Control Page, mbed website. https://developer.mbed.org/users/no2chem/code/PowerControl/.
[2] Power Management page, mbed web site. https://developer.mbed.org/cookbook/Power-Management#questions.
[3] Using the LPC1700 power modes. AN10915. Rev. 01−25 February 2010. NXP.
[4] J. Yiu, The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors, second ed., Elsevier, 2015.
[5] Silicon Labs web site. http://www.silabs.com/.
[6] EFM® 32 Starter Kit EFM32ZG-STK3200 User Manual, Silicon Labs, 2013.