

Chapter 15: Hardware Insights: Clocks, Resets and Power Supply

tw rev. 28.8.2024

If you use or reference these slides or the associated textbook, please cite the original authors' work as follows:

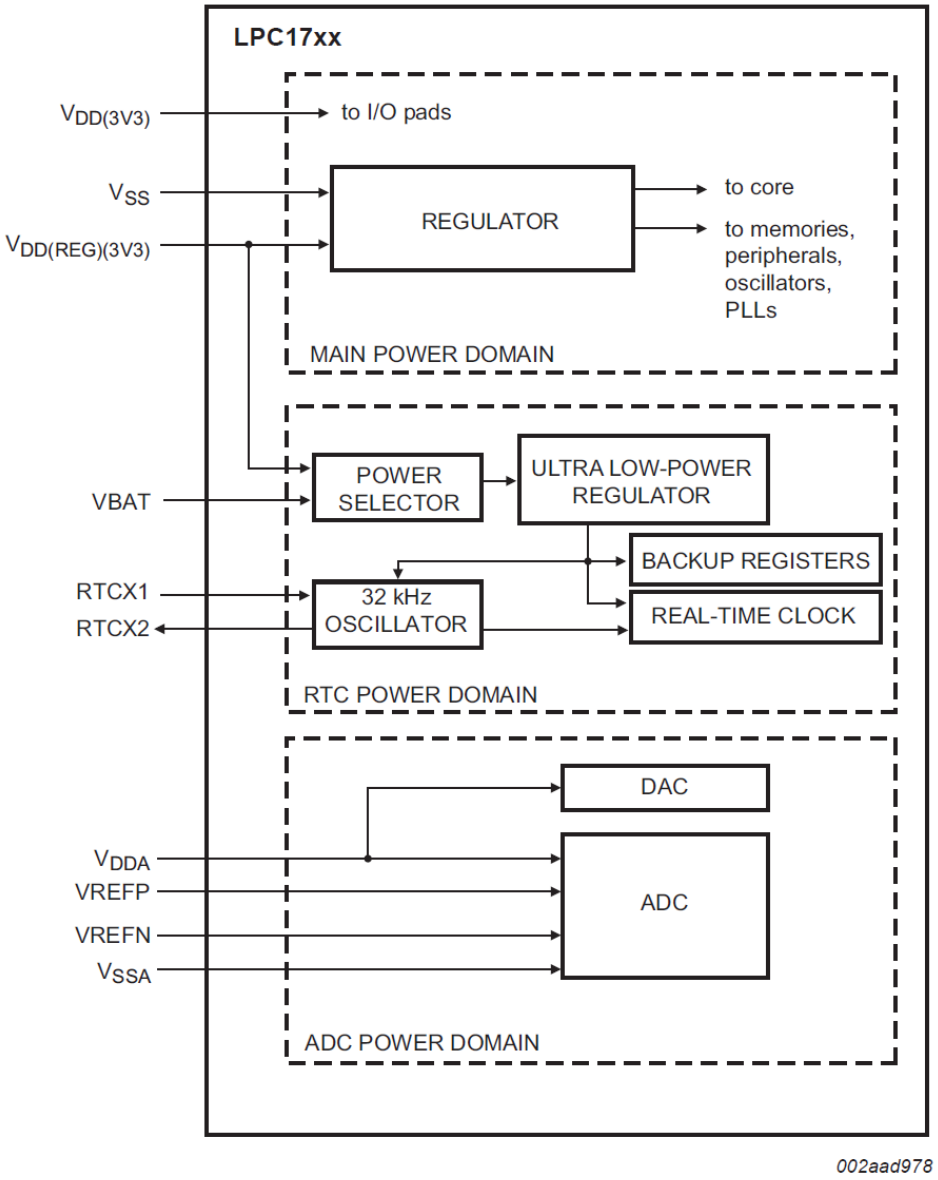
Wilmshurst, T., Spink, T. & Toulson, R. (2024). Fast and Effective Embedded Systems Design – From Bits and Bytes to IoT, with the ARM Mbed (3rd edition), Newnes, Oxford, ISBN: 978-0323951975.

www.embedded-knowhow.co.uk

Powering the LPC1768

This figure shows the internal power distribution of the LPC1768. Locate these features:

- two regular power supply inputs, labelled $V_{DD(3V3)}$ and $V_{DDREG(3V3)}$;
- V_{BAT}, which can be connected to an external battery to sustain only the Real Time Clock and backup registers;
- Supplies to ADC and DAC, V_{DDA} and V_{SSA} ;
- A voltage reference for ADC and DAC, V_{REFP} and V_{REFN}.



Powering the LPC1768

The supply voltage requirements of the LPC1768 are shown. All are centred 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 8. Static characteristics

T_{amb} = −40 °C to +85 °C, unless otherwise specified.

Symbol	Parameter	Conditions		Min	Typ[1]	Max	Unit
Supply pins							
$V_{DD(3V3)}$	supply voltage (3.3 V)	external rail	[2]	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		[3][4]	2.5	3.3	3.6	V
$V_{i(VBAT)}$	input voltage on pin VBAT		[5]	2.1	3.3	3.6	V
$V_{i(VREFP)}$	input voltage on pin VREFP		[3]	2.5	3.3	V_{DDA}	V

Supply Voltage Requirements for the LPC1768

Powering the Mbed LPC1768

The Figure shows the LPC1768 power and clock connections within the Mbed. Each connection shows the microcontroller pin number, and the name of the signal.

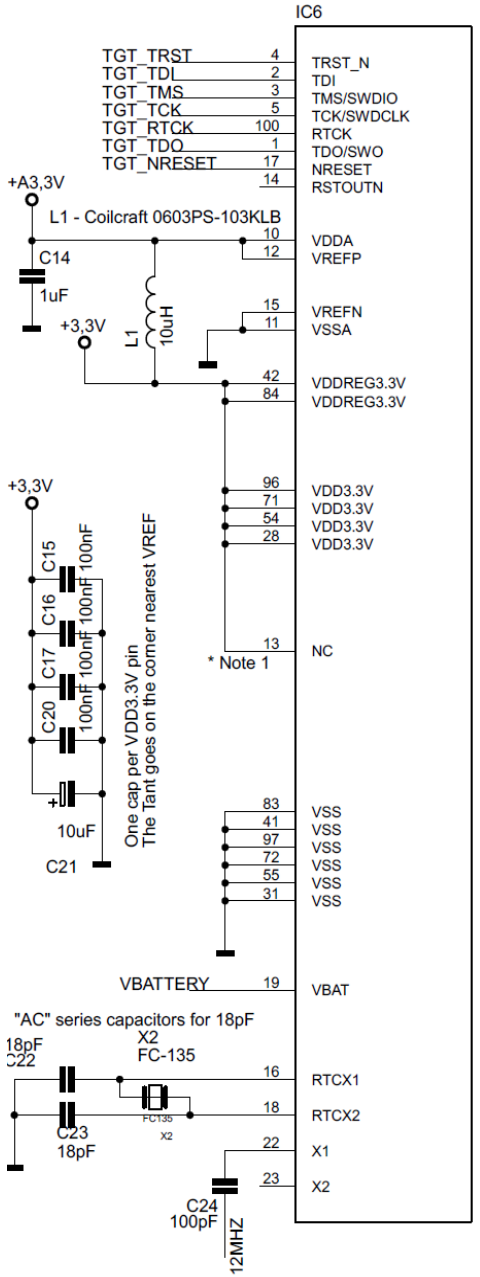
A complex integrated circuit tends to have more than one ground connection, and similar multiple power supply connections. This is because the interconnecting wires inside the IC are so very thin that they can have significant resistance.

There are six ground connections, labelled V_{SS}.

There are four $V_{DD(3V3)}$ connections and two for $V_{DDREG(3V3)}$. The mbed designers don't differentiate between these two, they just join them together.

The V_{BAT} connection is kept separate, and *can* be supplied via pin 3, V_B, of the mbed.

The V_{DD} supply is smoothed by the distributed capacitors C_{15} – C_{17} and C_{20}.



LPC1768 connections on the Mbed Board

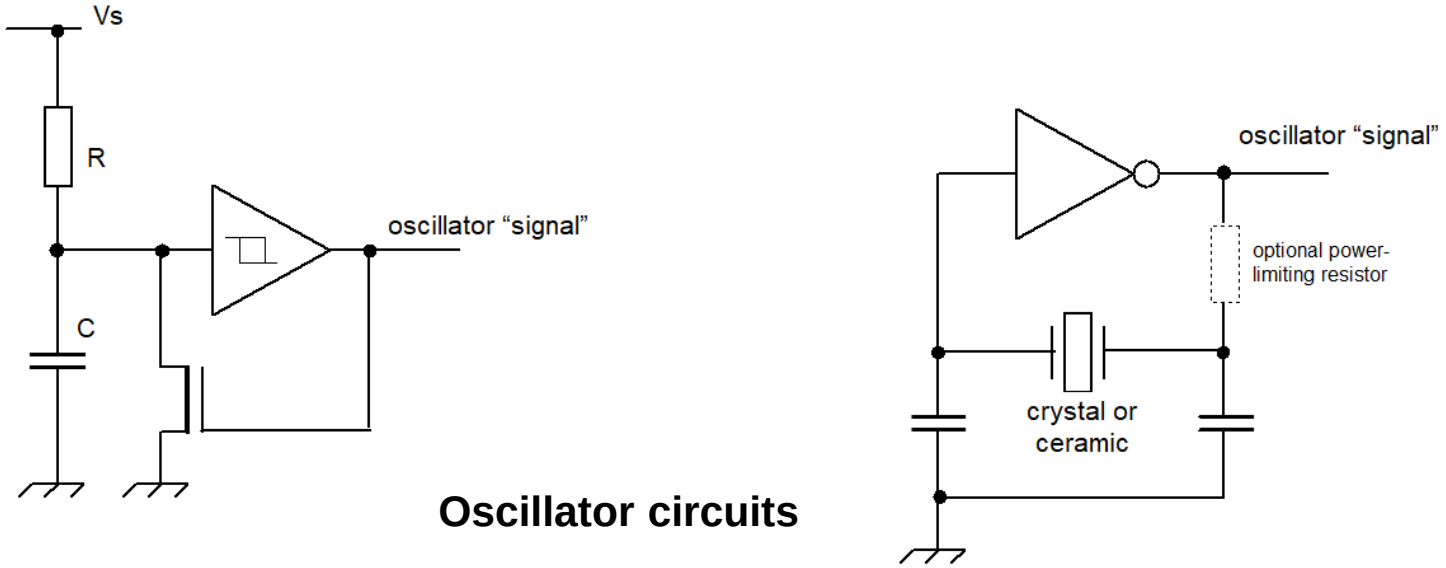
Clock Sources and their Selection

An essential part of the microcontroller system is the clock oscillator, a continuous square wave which drives forward most microcontroller action. It is also the basis of many time measurement or generation activities.

Clock oscillators can be based on resistor-capacitor (R-C) networks, or ceramic or crystal resonators; the designer should be aware of their relative advantages.

A popular R-C oscillator circuit is shown below left. The R-C circuit is the cheapest form of clock oscillator available and is widely used.

A quartz *crystal oscillator* is based on a very thin slice of crystal. This is piezoelectric. If electrical terminals are deposited on opposite surfaces of the crystal, then due to its piezoelectric property vibration can be induced and sustained electrically. A suitable circuit is shown below right.



Oscillator circuits

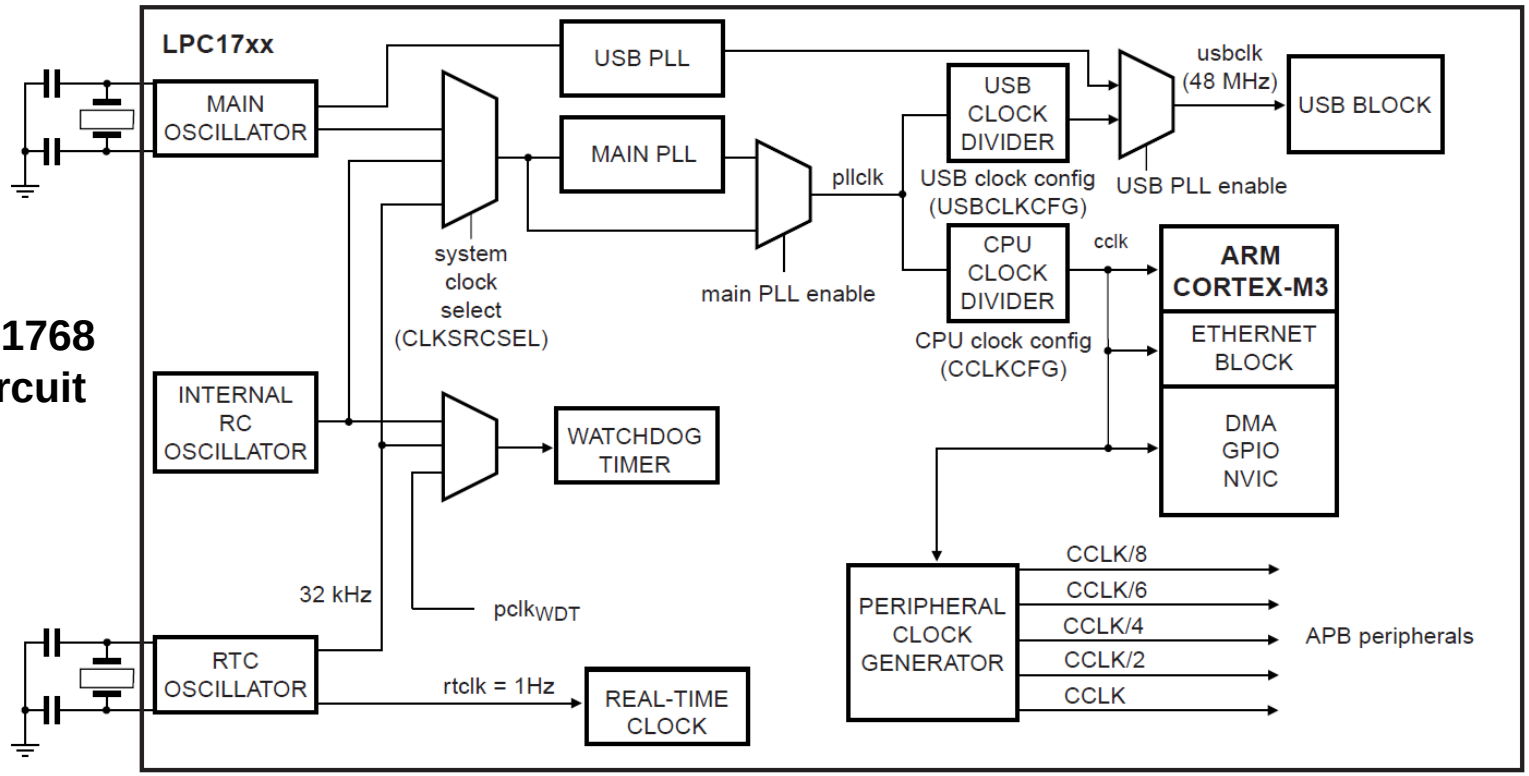
LPC1768 Clock Oscillators and the Mbed Implementation

The Figure shows how clock sources within the LPC1768 are distributed.

There are three sources, seen to the left of the diagram:

- the main oscillator (an external crystal), which can operate between 1 and 25 MHz. It can also act in *slave mode* - an external clock signal can be connected via a capacitor to the XTAL1 pin;
- the internal R-C oscillator, running at a nominal frequency of 4 MHz;
- the RTC oscillator - another crystal, but of low frequency.

The LPC1768 clock circuit



Setting the LPC1768 Clock Configuration Register

The CPU Clock Divider block, seen in the previous Figure, is controlled by the Clock Configuration register **CCLKCFG,** seen below.

The frequency of **pllclk** is divided by the number held in the lower 7 bits of this register, plus one; this produces **cclk**.

Table 38. CPU Clock Configuration register (CCLKCFG - address 0x400F C104) bit description

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

LPC1768 Clock Configuration register CCLKCFG

Adjusting the Clock Configuration Register

```
/*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=0x05; // divider divides by this number plus 1
  while(1) {
    myled = 1;
    delay();
    myled = 0;
    delay();
  }
}
void delay(void){          //delay function.
  //loop variable j, volatile to stop compiler optimisation
  volatile int j;
  for (j = 0;j < 5000000;j++) {
    j++;
    j--;                   //waste time
  }
}
```

This Program Example approximately replicates the original “blinky” program, using its own **delay( )** function. It starts by resetting the value of **CCLKCFG**, so the program then runs with a changed clock frequency.

Adjusting the LPC1768 Phase Locked Loop

The phase locked loop is 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 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.

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 sub-system is complex, and requires a very careful reading of relevant sections of the LPC1768 user manual.

Adjusting the Phase Locked Loop

The PLL is controlled by four registers, outlined in the Table. The PLL is enabled and connected through **PLL0CON**, with multiply and divide values set through **PLL0CFG**.

The Feed Register 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.

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 pre-divider, 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 have 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

PLL0 Control Registers

Adjusting the Phase Locked Loop

This program illustrates PLL0 control. The **main( )** function immediately disconnects and turns 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 int *)(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.

  //continued over…
```

Adjusting the Phase Locked Loop

```
  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
  }
}
```

Adjusting the Phase Locked Loop

This code fragment can be inserted in the previous program to introduce clock frequency changes with PLL0.

```
// 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
```

Selecting the LPC1768 Clock Source

If the clock source is to be changed, through the input multiplexer, it must be done with PLL0 shut down. This change is controlled by the Clock Source Select Register, **CLKSRCSEL,** with details shown.

Table 17. Clock Source Select register (CLKSRCSEL - address 0x400F C10C) bit description

Bit	Symbol	Value	Description	Reset value
1:0	CLKSRC		Selects the clock source for PLL0 as follows:	0
		00	Selects the Internal RC 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 baudrates > 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

Clock Source Select Register, CLKSRCSEL

Reset

At certain times, any microcontroller needs to start its program from the beginning. 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*.

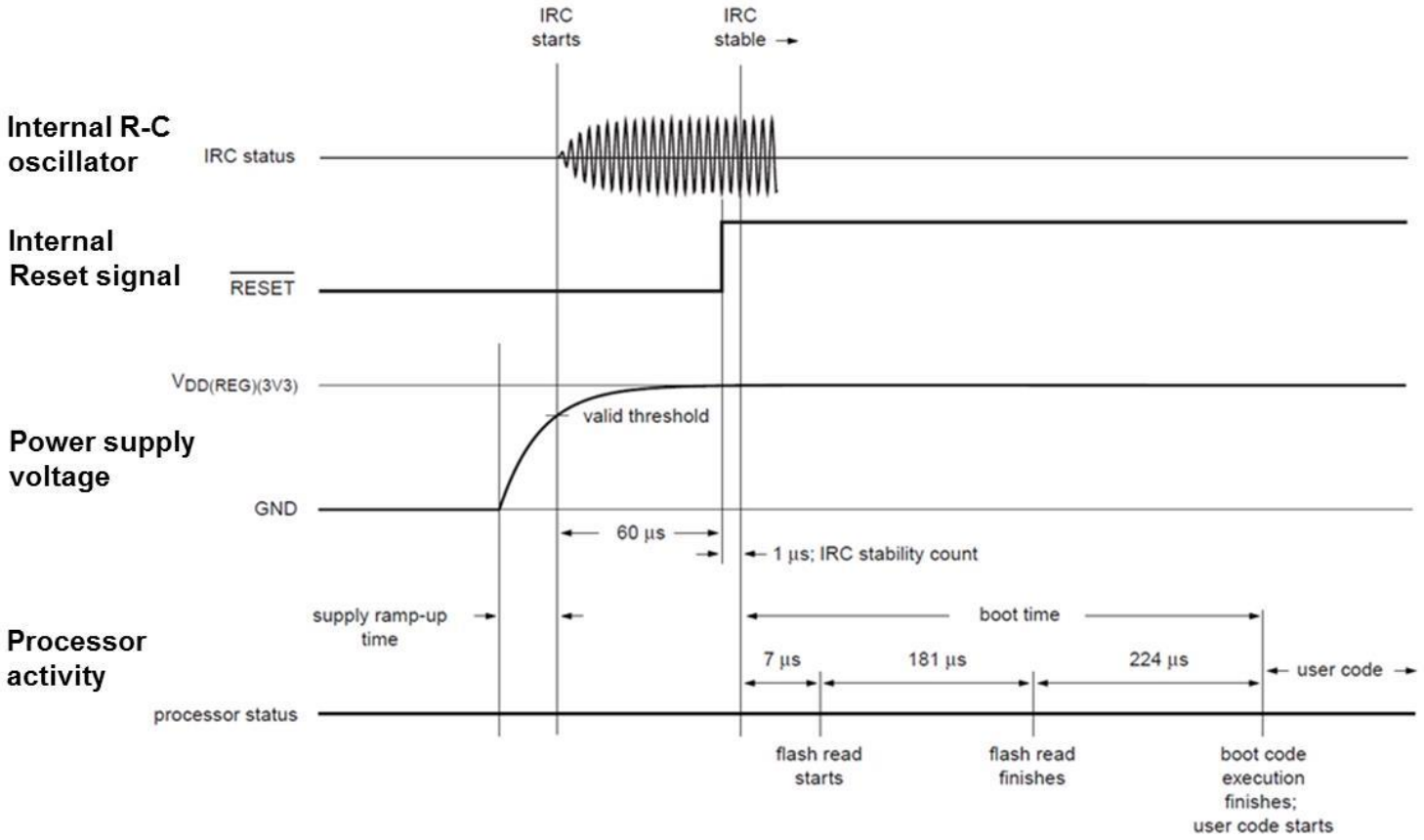
The CPU starts running its program when it leaves the reset condition.

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.

Apart from power-up, there are other times when reset is needed, including the possibility that the user may want to force a reset if a system locks or crashes.

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 stabilise, and in a complex system power to different parts of the circuit may stabilise at different times.



LPC1768 internal signals during start-up

Other Sources of Reset

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.

The mbed has a reset button. This is however connected to the interface microcontroller. This can force a reset to the LPC1768 itself.

Watchdog timer

A common failure mode of any computer-based system is for the computer to lock up. For most embedded systems this is unacceptable.

An uncompromising solution is the Watchdog Timer (WDT), which *resets the processor if the WDT is ever allowed to overflow*. The WDT runs continuously, counting towards its overflow value.

The programmer must ensure that this overflow doesn't happen. 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.

A WDT overflow causes a reset very similar to the power-on reset described above.

Watchdog Timer API

The Mbed OS provides a useful API for the WDT, summarised in Table below. This allows the programmer to start, load, check and refresh (“kick”) the WDT. We use some of these features in Program Example 15.4.

Functions	Usage
`Watchdog::get_instance`	Get a reference for the WDT (establishes link to the hardware WDT).
`bool start()`	Start the WDT with the maximum timeout value available for the target. Returns true if WDT started successfully.
`bool start(uint32_t timeout)`	Start the WDT with value timeout, in ms. Returns true if WDT started successfully.
`void kick()`	Reset the WDT.
`uint32_t get_timeout() const`	Returns refresh value of the WDT.
`bool stop()`	Stops the WDT. Returns true if WDT stopped successfully.

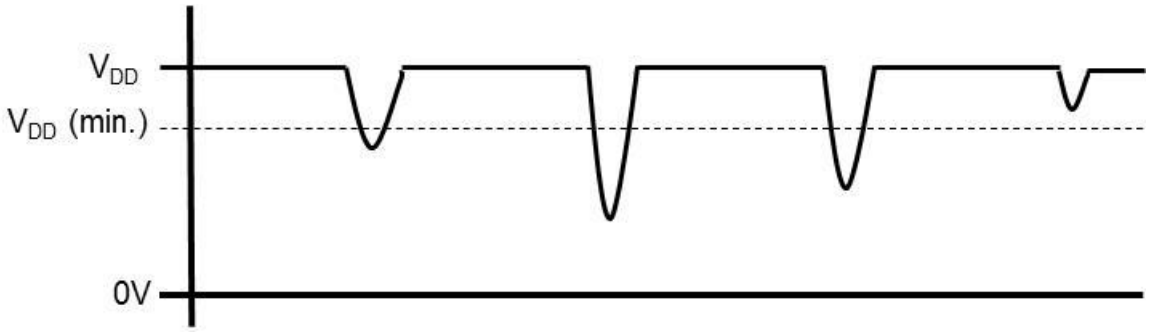
Other Sources of Reset

Brownout detect

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

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.



Voltage "brownouts"

Reason for Reset

Sometimes it is useful to know why a system has reset. Did the power dip, did the watchdog time out, was a physical reset pressed?

There is a useful OS API, called **ResetReason**, which allows the reason for the most recent reset to be acquired for evaluation. The API accesses those internal registers which hold the reason. Program Example 15.4, in following slides, which adapts and combines two examples from the Mbed web site, demonstrates this. It also shows use of the WDT.

Checking details of the program, **main( )** starts immediately with an access to **ResetReason,** thereby seeking to report on what reset event has caused the fresh start of program execution. It then launches the WDT, preloading with an equivalent value of 5 s. It establishes an interrupt input linked to the pushbutton, with ISR which resets the WDT. The program loops, printing a countdown value to screen. If the button *isn't* pressed, the WDT times out, the processor resets, the program relaunches and the reset reason (WDT timeout in this case) is extracted and displayed. Other forms of reset can also be tested.

WDT Example

```
/* Program Example 15.4. Implements Watchdog Timer, and tests for reset reason.
Works on Mbed LPC1768 or F401RE with no additional components.
*/

#include "mbed.h"
#include "ResetReason.h"

#include <string>                    //will be manipulating strings

const uint32_t TIMEOUT_MS = 5000;  //Timeout value loaded to WDT
InterruptIn button(p5);      //Replace p5 with BUTTON1 for F401
volatile int countdown = 9;

std::string reset_reason_to_string(const reset_reason_t reason){
  switch (reason) {
    case RESET_REASON_POWER_ON:
      return "Power On";
    case RESET_REASON_PIN_RESET:
      return "Hardware Pin";
    case RESET_REASON_SOFTWARE:
      return "Software Reset";
    case RESET_REASON_WATCHDOG:
      return "Watchdog";
    default:
      return "Other Reason";
  }
}
```

```
//ISR response to button push, resets Watchdog
void trigger(){
  Watchdog::get_instance().kick();  //Reset WDT
  countdown = 9;
}

int main(){
  //Acquire then print reset reason
  const reset_reason_t reason = ResetReason::get();
  printf("Last system reset reason: %s\r\n", reset_reason_to_string(reason).c_str());
  Watchdog &watchdog = Watchdog::get_instance();
  watchdog.start(TIMEOUT_MS);  //Start WDT with value of TIMEOUT_MS
  button.rise(&trigger);       //Link rise on Interrupt input to ISR
  uint32_t watchdog_timeout = watchdog.get_timeout();
  printf("Watchdog initialized to %lu ms.\r\n", watchdog_timeout);
  printf("Press BUTTON1 at least once every %lu ms to kick the "
         "watchdog and prevent system reset.\r\n", watchdog_timeout);

  while (true) {
    printf("\r%3i", countdown--);
    fflush(stdout);          //Flush the buffer
    ThisThread::sleep_for(TIMEOUT_MS/10); //Sleep for 500ms
  }
}
```

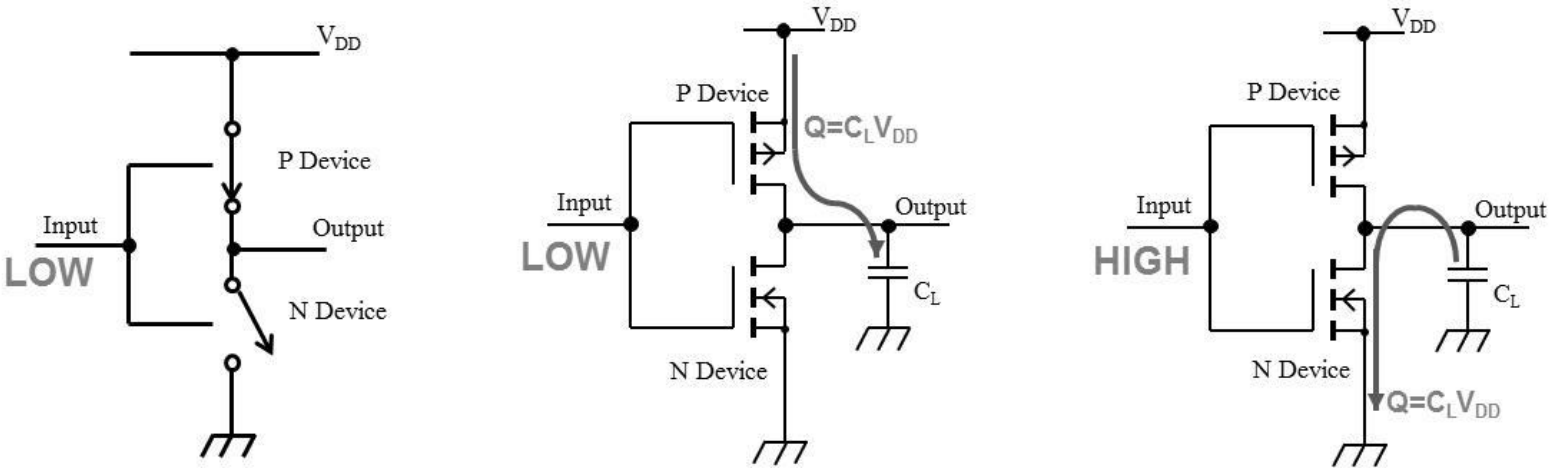
Testing Reason for Reset Program

An example screen output from the previous program is shown below, with three different sources of reset in turn.



Towards Low Power: How Power is Consumed in a Digital Circuit

Complementary Metal Oxide Semiconductor (CMOS) digital technology is the basis for the mobile phone, laptop, and any other portable electronic device. To understand how to minimise the power consumption these devices, it is useful to have some understanding as to how CMOS consumes power.



(a) Idealised circuit **(b) actual circuit, input going low** **(c) actual circuit, input going high**

Power consumption in a CMOS inverter (Eq. 15.1)

$$I_T = I_Q + \{ V_{DD} \times f\, C_{eq} \}$$

I_T : total current taken
f : switching frequency
V_{DD} : supply voltage
I_Q : quiescent current (due to leakage in the device)
C_{eq} equivalent capacitance of load and “shoot through” effect

Cells and Batteries

Cells are classified either as primary (non-rechargeable), 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.





Cells and Batteries

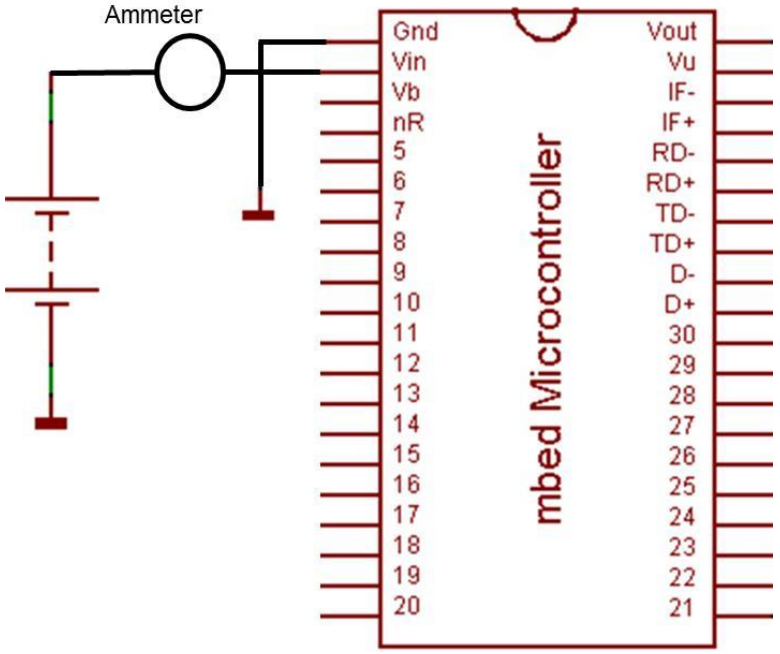
Two important electrical characteristics of a cell are terminal voltage and capacity; the latter generally measured in Amp-hours (Ah), or milliamp hours (mAh).

The mAh rating of a cell or battery can be converted to the fundamental unit of energy, Joules, by multiplying by terminal voltage and (60x60)/1000, or 3.6. The Table gives some examples.

Manufacturer	Technology	Shape/ Package	Nominal Terminal Voltage (V)	Capacity (mAh)	Capacity (Joules)
Varta	Silver Oxide	V301, Button	1.55	96	536
Varta	Silver Oxide	V303, Button	1.55	160	893
Procell	Alkaline	AAA cylinder	1.5	1175	6,345
Procell	Alkaline	AA cylinder	1.5	2700	14,580
Procell	Alkaline	PP3	9.0	550	17,820

Exploring LPC1768 Mbed Power Consumption

The LPC1768 Mbed was not designed for low power, yet it provides a good opportunity to study power supply in an embedded system, and to explore ways of reducing that power. It's interesting to set up a simple current measurement circuit as shown, and monitor current in the different examples and exercises.



LPC1768 Current Consumption Characteristics

The current consumption characteristics of the LPC1768 are shown, along with the LPC1769, which can run at a slightly higher clock frequency. The Table also refers to Sleep and Power-down modes that we shall meet soon. Table entries reinforce the message that clock frequency dominates current consumption.

The LPC1768 Mbed runs with a clock frequency of 96 MHz, with PLL enabled. Hence we estimate that the LPC1768 on the Mbed takes a supply current just under 42 mA, say 40 mA. But the values shown are with all peripherals disabled, so in practice it will be higher.

LPC1768/9 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

Controlling Power to Microcontroller Peripherals

This Program demonstrates power consumed by the microcontroller peripherals. Register **PCONP** controls the power to many of these. The program starts by switching all peripherals that are controlled through **PCONP** *on*. It then blinks an LED 10 times, during which time a current measurement can be made. Then all peripherals are switched off, and the program sits indefinitely in a loop. A further current measurement can be made. A practical application is likely to sit somewhere between the “all or nothing” extremes represented here.

```
/*Program Example 15.5
Powers down peripherals through PCONP register. Mbed LPC1768 only.     */

#include "mbed.h"
DigitalOut led1(LED1);

#define PCONP (*(volatile unsigned long *)(0x400FC0C4))

int main() {
  PCONP = 0xFFFFFFFF;  //switch all peripherals ON. For simplicity, write 1s
                       //even to reserved bits, tho normally not good practice.
    for(int i=10; i>0; i--){
      led1=1;
      wait_us(100);   //tiny LED flashes to limit ave current
      led1=0;
      wait_us(499900);
    }
  //Turn all peripherals controlled through PCONP OFF
  PCONP = 0x00000000;
  while (true) {
  }
}
```

Manipulating the Clock Frequency

The LPC1768 has extensive capabilities to vary the clock frequency, e.g. through the **CCLKCFG** and **PLL0CFG** registers already shown.

This means 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.

It is attractive to imagine that clock speeds can be reduced at will to reduce current consumption. While this is true in principle, take care! Many of the peripherals depend on that clock frequency as well, along with their Mbed API libraries, for example for the setting of a serial port bit rate.

LPC1768 Low Power Modes

Aside from varying clock frequency, there are other, even more effective power-saving 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.

The LPC1768 uses the modes summarised below, with power consumptions given in the earlier Table.

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 be achieved by an enabled Interrupt, or a Reset.

Deep Sleep mode: Here the main oscillator 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 wakeup. The 32 kHz RTC continues, and can generate an interrupt. 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.

Applying Operating System Sleep Modes

This Program Example tests OS support for low power operation. The program moves through a series of clearly identifiable phases, during which each of which a current measurement can be made.

It starts with 5 seconds of sleep. This is placed here because the OS guide claims that deep sleep is always enabled as **main()** starts, but may subsequently be disabled by mechanisms not always obvious to the programmer. Following a text introduction to screen, the program flashes the test LED 10 times, while using **wait_us()** for delays, thus the CPU is kept “busy”. A test for deep sleep permission follows, and sleep is re-entered. Deep sleep is then disabled, and a further 5 seconds of sleep follows.

```
/*Program Example 15.6
Testing Sleep modes using OS functions
Runs on either LPC1768 or F401RE without adjustment
*/

#include "mbed.h"
DigitalOut testled(LED1);
//int i;
```

```
int main(){
  ThisThread::sleep_for(5000);   //An initial sleep, at start of program
  printf("TESTING CURRENT CONSUMPTION\r\n");
  printf("_____________________________\r\n");
  //Normal operation for 5s
  printf("Normal continuous operation\r\n");
  for(int i = 10; i > 0; i--){
    testled = 1;
    wait_us(100);  //Keep the CPU busy
    testled = 0;
    wait_us(499900);
  }

  // Deep sleep for 5 seconds
  printf("Deep sleep allowed: %i\r\n", sleep_manager_can_deep_sleep());
  //Disable input from STDIN, will release deep sleep lock (Ref 15.2)
  mbed_file_handle(STDIN_FILENO)->enable_input(false);
  ThisThread::sleep_for(5000);

  //Lock deep sleep
  printf("Locking deep sleep\r\n");
  sleep_manager_lock_deep_sleep_internal();

  // Sleep for 5 seconds
  printf("Deep sleep allowed: %i\r\n", sleep_manager_can_deep_sleep());
  ThisThread::sleep_for(5000);
}
```

Reviewing the Nucleo F401RE Clock and Power Features

Like the LPC1768, the F401RE can accommodate two external crystals, as main and RTC clock sources. It also has a high speed (16 MHz) internal RC oscillator, and a low speed (32 kHz) internal RC oscillator; this can drive the WDT and/or RTC. A PLL is available to multiply up the main clock frequency.

A selection of current consumption figures for the Nucleo board is given in the Table. While these are measured under different operating conditions, so cannot be directly compared with the LPC1768, they display similar patterns. As expected, current consumption is highly dependent on clock frequency, as well as number of peripherals in use.

Three low power modes are available, with example current consumption figures appearing in the Table. They are summarised here:

Sleep mode: The CPU is stopped, all peripherals continue to operate. Exit is through interrupt or reset.

Stop mode: Clocks (except for RTC) and PLL are stopped. The contents of SRAM and registers are retained. Exit can be by external interrupt or RTC wakeup.

Standby mode: Clocks (except for RTC) are switched off, SRAM and register contents are lost. Exit can be by reset or RTC wakeup. This gives the lowest power consumption.

The Mbed OS maps its sleep mode to the F401RE Sleep mode, and deep sleep to Stop.

Overview: F401RE Current Consumption Characteristics

Condition	Clock frequency (MHz)	Supply Current
External clock, all peripherals enabled, code running from Flash memory	84	23.2 mA
External clock, all peripherals enabled, code running from Flash memory	40	10.8 mA
External clock, all peripherals disabled. code running from Flash memory	84	12.3 mA
External clock, all peripherals disabled. code running from Flash memory	40	6.0 mA
Sleep Mode: External clock, all peripherals enabled	84	16.6 mA
Sleep Mode: External clock, all peripherals disabled	84	5.3 mA
Stop Mode: all oscillators off, main and low power regulator usage.	-	153 uA
Standby Mode: all oscillators (inc. RTC) off.	-	1.8 uA

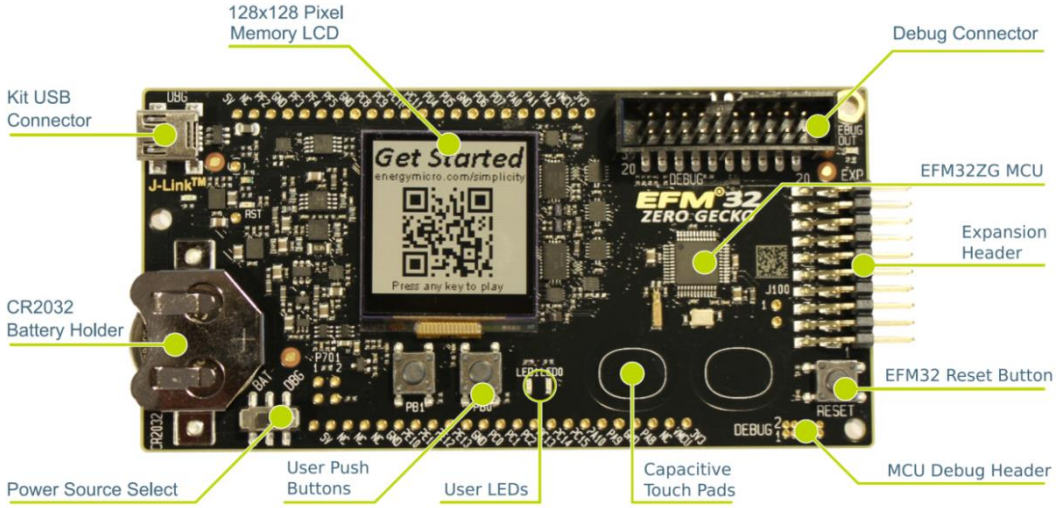
the EFM32 Zero Gecko Starter Kit

The simplest Cortex cores are the M0 and M0+. These are specifically designed for small scale, low power and low cost applications.

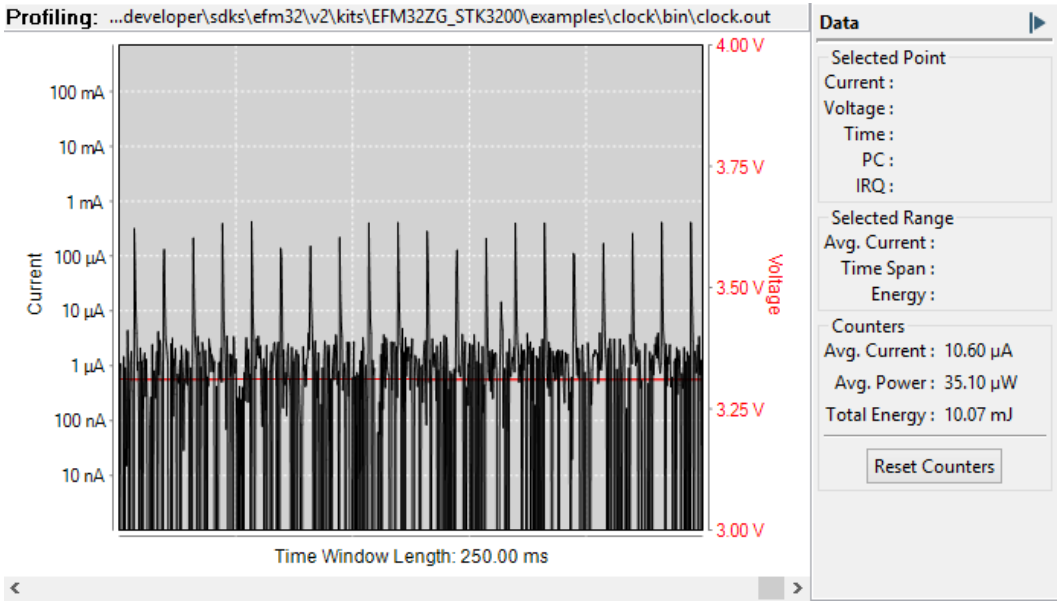
The EFM Zero Gecko Starter Kit is a development board based around the Silicon Labs Zero Gecko microcontroller.

It forms a useful example of a truly low-power mbed-enabled design. It is based on an M0+ core, and is designed for extreme low power.

It has an on-board *Energy Profiler*, which can measure current consumption from 0.1 uA to 50 mA.



The EFM32 Zero Gecko Starter Kit



The EFM32 Zero Gecko Energy Profiler

Chapter Review

- Distribution of power supply and clocks in microcontrollers can be understood from studying data sheets and user manuals; this leads to valuable insights for the programmer and circuit designer.
- Power consumption mechanisms in digital electronics are well understood. Microcontroller consumption data is generally available in good detail, including for both the LPC1768 and F401RE microcontrollers.
- Sophisticated techniques, both hardware and software, exist to minimise power consumption. In the Mbed OS, software techniques centre around use of *Sleep* and *Deep Sleep* modes.
- Power consumption in the Mbed LPC1768 can be optimised, but this development board is not ideal for low power applications.
- Power consumption in the Nucleo F401RE can be monitored and optimised, with the possibility of energy efficiency which is of interest for low power applications.
- For competitive low power design, circuits must be designed specifically and rigorously for that purpose. The EFM32 Zero Gecko, applying the M0+ Cortex core, is one example where this has been achieved.