LPC4088 Timer Interrupts

CM0506 Small Embedded Systems

Dr Alun Moon

Seminar 5

Here the module begins to separate from EN0572. The programming structure will make extensive use of interrupts to handle events, along-side a *main-loop* handling polling and time consuming actions.

Note: *No Operating System*

LED Driver

Without an operating system, there is not the need for the complexity of a Hardware Abstraction Layer. It can be argued that since the the hardware of the embedded system is fixed at design time, the flexibility of the HAL is not needed.

We still need an API abstracting the LEDs behind a device driver. That way we can keep all the hardware dependencies behind the API. A clean interface, one not exposing any knowledge of the underlying hardware still needs to be developed.

Further more the device driver needs to be efficient at run time as it will be called as part of interrupt handlers.

Driver API

A driver API needs the following main features.

- A means of initialising the devices
- A means of controlling the devices
- A means of accessing their state.

The style used has an enumerated data-type, which gives logical names by which the LEDs can be referred to in code.

Question:
Why are there both `left_green` and `LED1` names defined?

```
enum LED {
 LED1, LED2, LED3, LED4,
 left_green=LED1, right_green,
 left_blue, right_blue
};
```

The functions to control the LEDs take this as a parameter,

```
ledOn(enum LED name);
ledOff(enum LED name);
ledToggle(enum LED name);
int ledState(enum LED name);
```

These, along with the prototype for `ledInit()` can be put in the header `led.h`.

Exercise 1: Git download of initial code

Retrieve the project from the GIT repository.

```
$ git clone https://github.com/dr-alun-moon/timers
$ cd timers
$ git checkout ex.1.1
$
```

Examine the files for the LED driver `led.h` and `led.c`.

1. Can you follow the way the code is structured?
2. Why are the SET and CLR registers used to turn different LEDs on?

LED circuits and effects on code. The green (LED1 and LED2) and blue (LED3 and LED4) LEDs are connected to the processor using different circuits (figures 1 and 2 [Emb, 2014b]).

LED 1 & 2 ARE CONNECTED AS PART OF THE USB SYSTEM. The circuit in figure 1 is controlled by the PNP transistor. This transistor is "on" when the voltage to the base is low. A logic 0 (LOW) switches on the device. This is why in the code, writing to the CLR register (clearing the bit) the LED turns on.

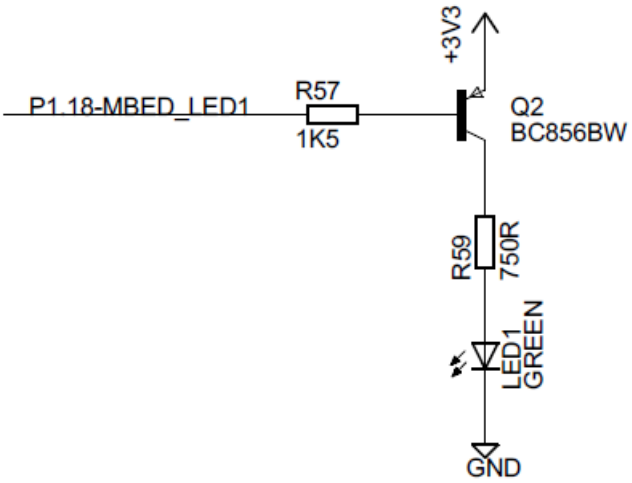

Figure 1: Connection for LED 1 & 2

LED 3 & 3 ARE CONNECTED TO OTHER GPIO PINS. The circuit is in figure 2. The LED has current flowing through it when the connection is high. A logic 1 (HIGH) switches on the device. This is why in the code, writing to the SET register (setting the bit) the LED turns on.

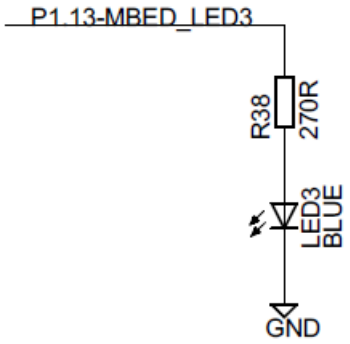

Figure 2: Connection for LED 3 & 4

Exercise 2: Extend the driver to Base-board LEDs

From the information in the Base Board schematic [Emb, 2014a], use the information to extend the driver to the Red, Green, and Blue LEDs. Things to note include:

- What Port is the LED connected to?
- Which Pin is the LED connected to?
- Is the circuit active-high or active-low?

Solution

A solution is in Git

```
$ git checkout s.1.1
```

Question 1: enum and switch

In the code I've used a combination of `enum` for LED names, and then a `switch` to select which code to use. Why?

- What does the compiler say if the `switch` does not check all of the `enum` values?
- What does the compiler warn about in the interrupt handler `ledone`?
- Are these useful features?
- How efficient is the compiled code for a switch?

If you want to have a go at optimising the code, try it out. You will find that there is a tension between compiled code side and ease of writing the C code. In reality hand optimising is a tricky buiseness, and rarely done well.

Timers and Interrupts

ATTENTION CAN NOW BE TURNED TO THE TIMERS AND CORRESPONDING INTERRUPTS. These can be used to trigger periodic events in some fashion. The LPC4088 has four timers, each of which operate in the same way. See pages 9–13 for details of how the processor's internals work.

Timer Init

The code used for initialising a timer. Most of these have the required values, some are changed for different uses.

```
LPC_SC->PCONP |= (1UL << 1);
LPC_TIM0->TCR = 0;
LPC_TIM0->PR = 59999;
LPC_TIM0->CTCR = 0;
LPC_TIM0->MR0 = PeripheralClock / tickHz - 1;
LPC_TIM0->MCR = 0x03UL;
timer0UserDefinedHandler = handler;
LPC_TIM0->IR = 0x3F;
NVIC_EnableIRQ(TIMER0_IRQn);
LPC_TIM0->TCR |= (1UL << 0);
```

A line-by-line discussion of the values used follows. Table 539 on page 691 of the User manual [NXP, 2014] is a good summary of the register functions.

```
LPC_SC->PCONP |= (1UL << 1);
```

The PCONP register controls power to the peripheral devices. Bit 1 is the power supply for Timer-0 (see [NXP, 2014, Table 14, pg.30])

```
LPC_TIM0->TCR = 0;
```

The Timer control register [NXP, 2014, 24.6.2] switches the timer on and off. Disabling the timer while configuring it is important for correct behaviour to occur.

```
LPC_TIM0->PR = 59999;
```

The prescale register [NXP, 2014, 24.6.4] controls the rate at which the Timer ticks. The Timer counts one tick for each (PR+1) ticks of the peripheral clock (60 MHz). Setting the value to 59999 makes the timer count for every 60000 ticks of the Peripheral clock, or $60000 \times 60\,\text{MHz} = 1\,\text{ms}$.

```
LPC_TIM0->CTCR = 0;
```

The count-control-register [NXP, 2014, 24.6.11] selects whether the system is used as a clock (Peripheral Clock) or a counter. By setting it to 0 we are using the device as a counter.

```
LPC_TIM0->MR0 = period_ms - 1;
```

Match Register 0, is one of the 4 Match Registers for the timer [NXP, 2014, 24.6.7]. An interrupt occurs every (MR+1) ticks of the Timer. Since we have make the clock tick once every 1 ms, we just need the period in ms minus 1.

```
LPC_TIM0->MCR = 0x03UL;
```

The Match Control Register [NXP, 2014, 24.6.6] determines what actions occur, when the timer count (CR) equals one of the match registers (MR*n*). Setting bit 0 causes an interrupt to be generated, setting bit 1 causes a timer reset.

```
timer0UserDefinedHandler = handler;
```

This saves a user defined function for use in the ISR.

```
LPC_TIM0->IR = 0x3F;
```

The Interrupt Register [NXP, 2014, 24.6.1] indicates which interrupt has occurred. Writing 1s to this register resets the corresponding interrupt.

```
NVIC_EnableIRQ(TIMER0_IRQn);
```

Enables the timer-0 interrupts (see [NXP, 2014, Chapter 5].

```
LPC_TIM0->TCR |= (1UL << 0);
```

The Timer control register [NXP, 2014, 24.6.2] switches the timer on and off. Setting bit 0 enables the timer, *note* bit 1 is kept clear.

ISR

The Interrupt-Service-Routine is show below.

```
void TIMER0_IRQHandler(void) {
        timer0UserDefinedHandler();
        LPC_TIM0->IR |= (1UL << 0);
}
```

The first line calls the function stored from the Timer init function. The second line writes a 1 to bit 0 resetting the interrupt. If this bit is *not* reset, then no further interrupts can occur.

The Timing Diagram for timer interrupts is shown in table 1. It is shown for a hypothetical case with the prescale register set to 2 (PR=2) and the match register 1 (MR*n*=1).

Simple Flashing

Example 1.2 just flashes one LED in a simple fashion.

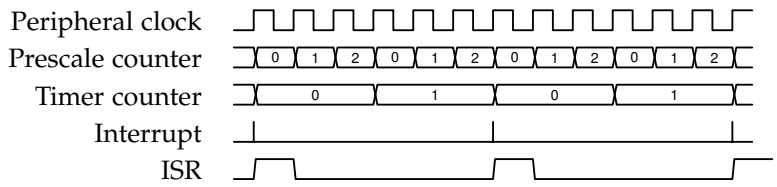

Table 1: Timing diagram

```
$ git clone https://github.com/dr-alun-moon/timers
$ cd timers
$ git checkout ex.1.2
```

Exercise 3: Add a second LED

Add code to use a second timer (Timer-1) to flash a second LED at a different rate.
If you use a multiple of the rate on Timer-0, it is easier to check the operation is as you'd expect.

Solution

A solution is in Git

```
$ git checkout s.1.2
```

A possible timing diagram is shown in table 2, assuming the LED state is toggled in each ISR.

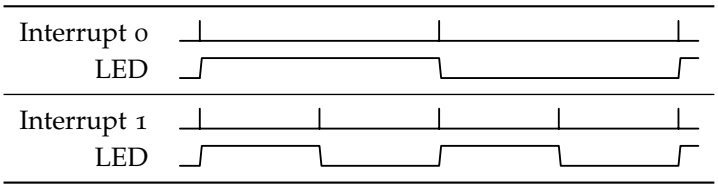

Table 2: Timing diagram for two LEDs

Multiple Interrupts

EACH TIMER CAN HAVE UP TO 4 INTERRUPTS. Generated by each of the 4 match registers. One will have to be configured to reset the timer, which gives the timer period (time between interrupts). The other three will have values less than the timer period to cause interrupts at (up to) 3 points within the timer period. The Match Control Register [NXP, 2014, 24.6.6], has bits to enable the interrupts for each of the match registers.

These share the same interrupt routine. The interrupt handler will have to interrogate the Interript Register (IR) [NXP, 2014, 24.6.1] to determine which match register caused the interrupt.

Setting up MR1

TO SET UP AN INTERRUPT ON MR1. We need to write a value into it, and set the appropriate flag in the MCR. The value needs to be a fraction of the value in MR0, or else the interrupt will never occur. A fraction can be passed to `timer0Init` as a `float`, and then used to multiply MR0. Bit 3 in the match control register needs to be set to enable the interrupt. Finally the interrupt handler needs to test the interrupt register (IR) to see which match register caused the interrupt.

A 20% duty cycle means that the LED is on for the first 20% of the flashing period.

Exercise 4: LED1 on a 20% duty cycle

The code in branch ex.1.3 has been modified to set up timer-0 to flash one LED on a 20% duty cycle.

```
$ git clone https://github.com/dr-alun-moon/timers
$ cd timers
$ git checkout ex.1.3
```

Examine `main.c`, `timer.c`, and `timer.h` to check how the changes have been made.

1. Add similar changes to enable a second match register on Timer-1
2. Flash another LED with an 80% duty cycle

Solution

A solution is in Git

```
$ git checkout s.1.3
```

Question 2

Are there any behaviours that would need:

- 3 interrupts per Timer?
- 4 interrupts per Timer?

Question 3

How could you stop timer-0 after 4 flashes?

References

Cortex-M4 Devices, Generic User Guide. ARM, 2010.

LPC4088 Experiment Base Board rev A. Embedded Artists, September 2014a.

LPC4088 Quickstart Board rev B. Embedded Artists, August 2014b.

UM10562 LPC408x/407x User manual. NXP, rev. 3 edition, March 2014.

Timer Behavior

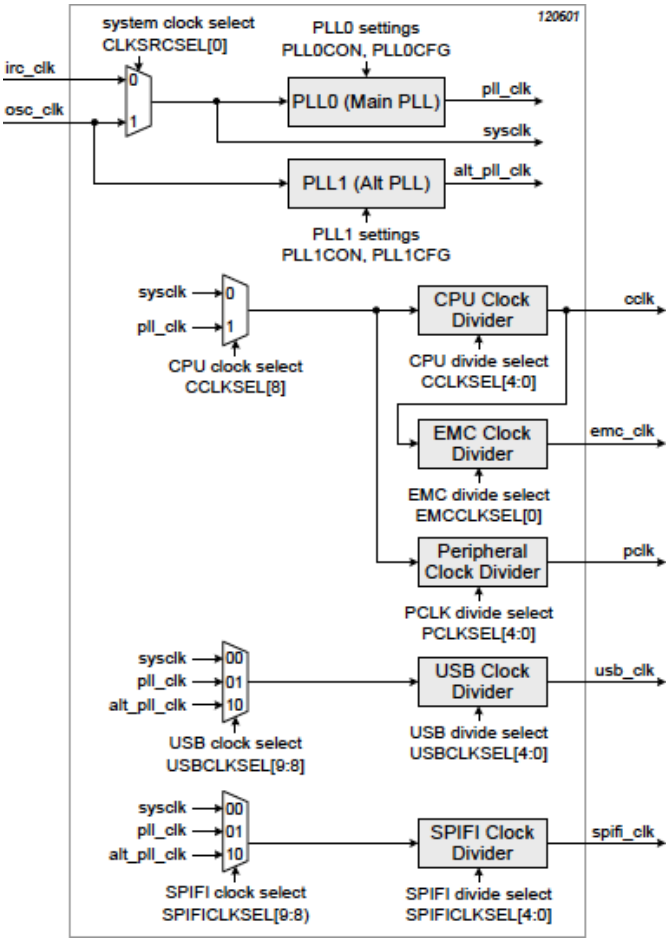

Figure 3: Clock generation

Figure 3 from [NXP, 2014, pg. 21] shows the various clock generation circuits. fFigure 4 [Emb, 2014b, sheet 2] shows the external crystal used as the external oscillator.

The PLL (Phase Locked Loop) is used to boost the input clock signal to higher frequencies.

The Register `PLL0CFG` controls the behaviour of the PLL [NXP, 2014, sec 3.10.4]. In the startup files `system_LPC407x_8x177x_8x.c` and `system_LPC407x_8x177x_8x.h` the values for the PLL and timers are defined. (see [NXP, 2014, table 47] for mapping register values to M and P)

```
#define PLL0CFG_Val           0x00000009
```

Quantity		Source	Value	
M	PLL multiplier	`PLL0CFG` bits 4:0	10	
P	PLL divider	`PLL0CFG` bits 6:5	1	
f_{in}	input frequency	external crystal	12	MHz

Table 3: Values for PLL

The output frequency of the PLL is given by

$$f_{\text{out}} = M f_{\text{in}} \tag{1}$$

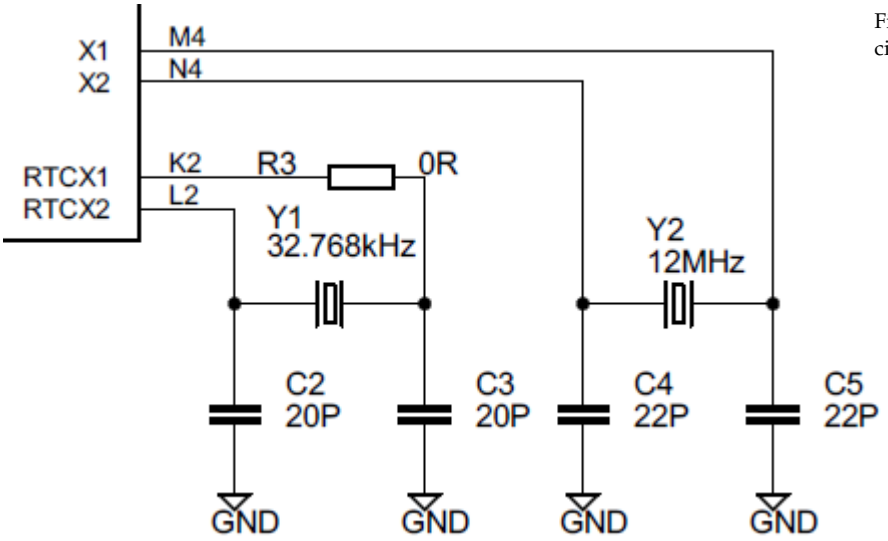

Figure 4: Quickstart board Crystal circuit

The current controlled oscillator (internal to the PLL) has a frequency of

$$f_{CCO} = 2MPf_{in} \tag{2}$$

Putting the values into these we get

crystal	12	MHz
f_{in}	12	MHz
f_{out}	120	MHz
f_{CCO}	240	MHz

Table 4: Clock and oscilator values

CPU Clock

The output from the PLL is selected as the CPU clock frequency by the `CCLKSEL` register, which also defines the divider value in bits 4:0 [NXP, 2014, Table 20, sec3.3.3.2].

```
#define CCLKSEL_Val             0x00000101
```

The CPU clock divider is 1, so the PLL ouput is used directly as the clock signal This 120 MHz frequency is made available from `system_LPC407x_8x_177x_8x.c` line 368 as `SystemCoreClock`.

Peripheral Clock

The Peripheral Clock signal drives most of the on chip peripherals. The register `PCLKSEL` [NXP, 2014, sectin 3.3.3.5, table 23] controls the divider value to use

```
#define PCLKSEL_Val             0x00000002
```

This gives the divider value as 2. The peripheral clock is therefore 60 MHz. This frequency is made available from `system_LPC407x_8x_177x_8x.c` line 369 as `PeripheralClock`.

SysTick

The SysTick timer is driven by the 120 MHz CPU clock signal. The timers uses a 24bit count down timer, the reset value is loaded from `STRELOAD`. Given 24 bits can hold a maximum value of $2^{24} - 1$ or

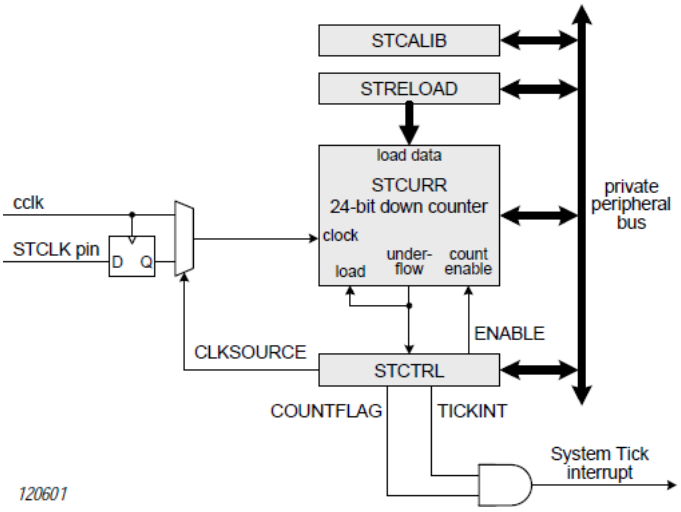

Figure 5: SysTick timer circuit

16 777 215 we get a maximum SysTick period of 2^{24} CPU ticks of 0.139 810 s

$$\frac{16777216}{120000000} = 0.139810$$

Conversely the minimum period is 8 ns! The Cortex M4 User Guide [ARM, 2010] has a good description of the SysTick timer operation.

Timer 0/1/2/3

The timers are covered in [NXP, 2014, Chapter 24]. Figure 6 shows the timer circuit. The peripheral clock signal drives the timer through the prescale counter.

Operation – prescale and count

Timer Counter (TC), Prescale Register (PR) and prescale counter (PC). The 32 bit TC is incremented every PR+1 cycles of PCLK. Prescale Register. When the Prescale Counter (PC) is equal to this value, the next clock increments the TC and clears the PC. Prescale Counter. The 32 bit PC is a counter which is incremented to the value stored in PR. When the value in PR is reached, the TC is incremented and the PC is cleared.

The prescale register therefore is used to scale the CPU clock down to give the timer clock tick.

The timer register, counts up on each tick from the prescaler. When it reaches the value in one of the match registers, the timer reacts acording to the values in the match control register.

So the relevent relations are shown in table 5. A simple scheme is to set the prescaler to 0 (divide by 1). The period of the timer,

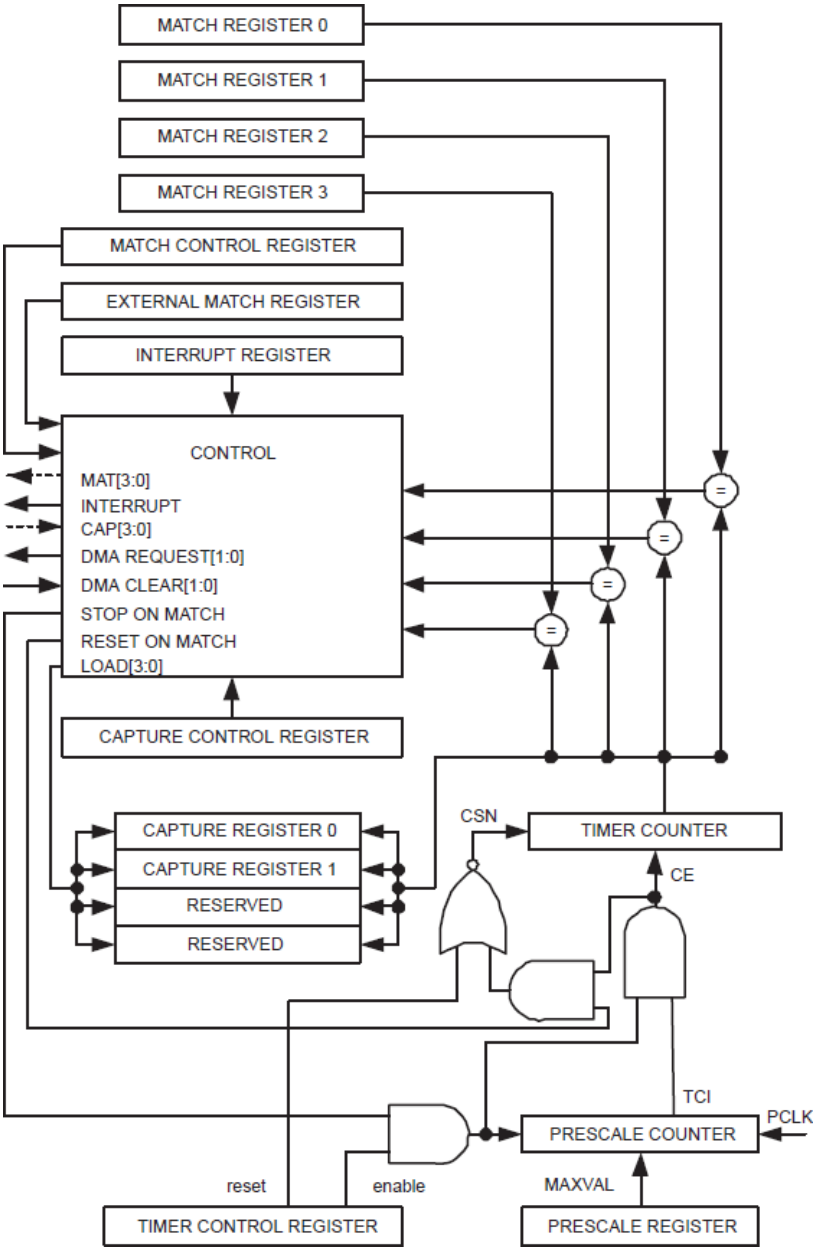

Figure 6: Timer circuit

the value to use in the match register is given by (`tickHz` is the rate wanted for the timer interrupt).

```
PeripheralClock / tickHz - 1
```

Peripheral Clock rate	f_P	60 MHz
Prescale Register	PR	
Timer tick rate	$f_T = \frac{f_P}{\text{PR}+1}$	
Match Register	MR	
Interupt tick rate	$\frac{f_T}{\text{MR}+1}$	

Table 5: Timer relationships

The highest rate the timer can run at is one interrupt every tick of the Peripheral Clock 60 MHz or once every 16.7 ns. With the prescaler to 0 (divide by 1), the timer interrupt can occur at a maximum of every 2^{32} ticks, or 71.6 s

$$\frac{2^{32}}{60 \times 10^6} = 71.6$$

With the prescaler and the match register set to their maximum values, an interrupt occurs every 307 445 734 561 s or 9749 years!