LPC4088 Button Device Driver

CM0506 Small Embedded Systems

Dr Alun Moon

Seminar 5b

Button Device Driver

If we take the user board button[1] by itself, the task of designing an API becomes fairly simple. We need to be able to initialise the hardware to have the button as an input, and we need to be able to read the button state.

[1] The one on the red PCB

The API definitions in the header file will look something like.

```
void button_init(void);
int button_pressed(void);
```

With only one button, the API is simply a case of configuring the GPIO port for input on the appropriate pin, and reading the state of the pin.

Internals

The internals of the driver merit some examination and discussion. There are a number of features that the processor has around the port that are worth looking at in detail as they can make life easier for the programmer.

Initialisation

For the most part the initialisation of the port is familiar. The pin is put into its GPIO mode and set for input.

```
void button_init(void){
  LPC_IOCON->P2_10 = PULLUP|INVERT; /* use pullup resistor and signal inversion */
  LPC_GPIO2->DIR &= ~buttonpin; /* input - clear direction bit */
}
```

The IOCON register is worth examining, its value is no longer 0.

The User manual [NXP, 2014] lists the behaviour of the IOCON register in Chapter 7. The button is connected to Port-2 pin-10, table 79 on page 127 lists this as a type D pin. Tables 83 and 84 describe this in more detail.

Bits 3:4 are described as MODE bits, controlling an internal resistor. Without going into the details of the electronics. This can be set to no resistor (00) or a pull-up resistor (10). Setting this as a pull-up

resistor ensures that a high value is read if the input is not connected to another signal.[2]

[2] We can get away with no resistor, but I prefer to use the pull-up resistor as it is available and ensures a *High* or *Low* signal.

Bit 6 controls an inverter [NXP, 2014, 7.3.4]. Setting this bit turns the inversion on. A signal level of *Low* on the pin now reads as a logic 1, and a *High* signal now reads as 0. This is useful here as the bit now reflects the state of the button, *pushed*(1), or *not-pushed*(0).

By setting the ICON register to `PULLUP|INVERT`. We are guaranteed a 0 or 1, with 1 being the button pressed.

Reading the button state

Reading the button state is now a simple matter of testing bit-10 in port-2.

```
int button_pressed(void){
  int state = 0;
  LPC_GPIO2->MASK = ~buttonpin;
  state = LPC_GPIO2->PIN;
  LPC_GPIO2->MASK = nil;
  return state;
}
```

The built-in MASK register performs a similar action to the masking operations we have already seen in code. Its operation is subtly different.

A value of 1 marks the bit as not to update the value of the PIN register on read [NXP, 2014, 8.5.1.2].

For convenience I read the line as

```
  LPC_GPIO2->MASK = ~buttonpin;
```

Mask (*hide*) **not** the button pin.

When we then read the value of the PIN register, we will get 0s everywhere *except* for the value of bit-10, which reflects the state of the button.

Since C treats 0 as false, non-zero as true and we have a value of 0 or 1024 (2^{10}), then we can just return this value[3].

[3] you could test this and return true/false as required, or shift the bits down 10 places to give 0 or 1

Interrupts

The User Button is connected to Port-2, which can generate interrupts. We can add to the API to initialise and configure an interrupt handler for the button.

Rising and Falling edge triggers

Section 8.2.2 of the manual, says that the port can generate interrupts on rising and falling edges. A rising-edge is where the value goes from 0-to-1, and a falling-edge where the value goes from 1-to-0.

These values are taken from the PIN register, so after the effects of the inverter on the pin. So given the configuration of IOCON we have used, we get

Button changes from *not*-pushed to pushed	rising edge
Button changes from pushed to *not*-pushed	falling edge

Table 1: Button interrupts

Initialisation

There is one interrupt generated for GPIO registers, IRQ 38. The interrupt controller has two registers, one for disabling the interrupt and one for enabling the interrupt. Because the number of interrupts is greater than 32, CMSIS defines an array, which is why the code for enable/disable looks a little odd. IRQ 38 is bit 6 in interrupt register 1.

To select the rising or falling edge to trigger the interrupt, we set the appropriate bit (10) in the Port interrupt control registers.

```
NVIC->ICER[1] = (1UL<<(GPIO_IRQn-32)); /* disable interrupt */

LPC_GPIOINT->IO2IntEnR = buttonpin;    /*  rising edge triggers interrupt */
LPC_GPIOINT->IO2IntEnF = buttonpin;    /* falling edge triggers interrupt */

NVIC->ISER[1] = (1UL<<(GPIO_IRQn-32)); /* enable interrupt */
```

In the interrupt handler `GPIO_IRQHandler` we can look at the status registers for rising and falling edges to identify the cause of the interrupt.

In the initialisation of the interrupt handler we pass in a pointer to a function for the custom code for the application. If this function takes the appropriate **enum** type (see the header file), we can pass the type of edge to this when called from the interrupt handler.

```
if( LPC_GPIOINT->IO2IntStatR ) button_event(Risingedge);
if( LPC_GPIOINT->IO2IntStatF ) button_event(Fallingedge);
```

where

```
static void (*button_event)(enum edge_t);
```

References

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