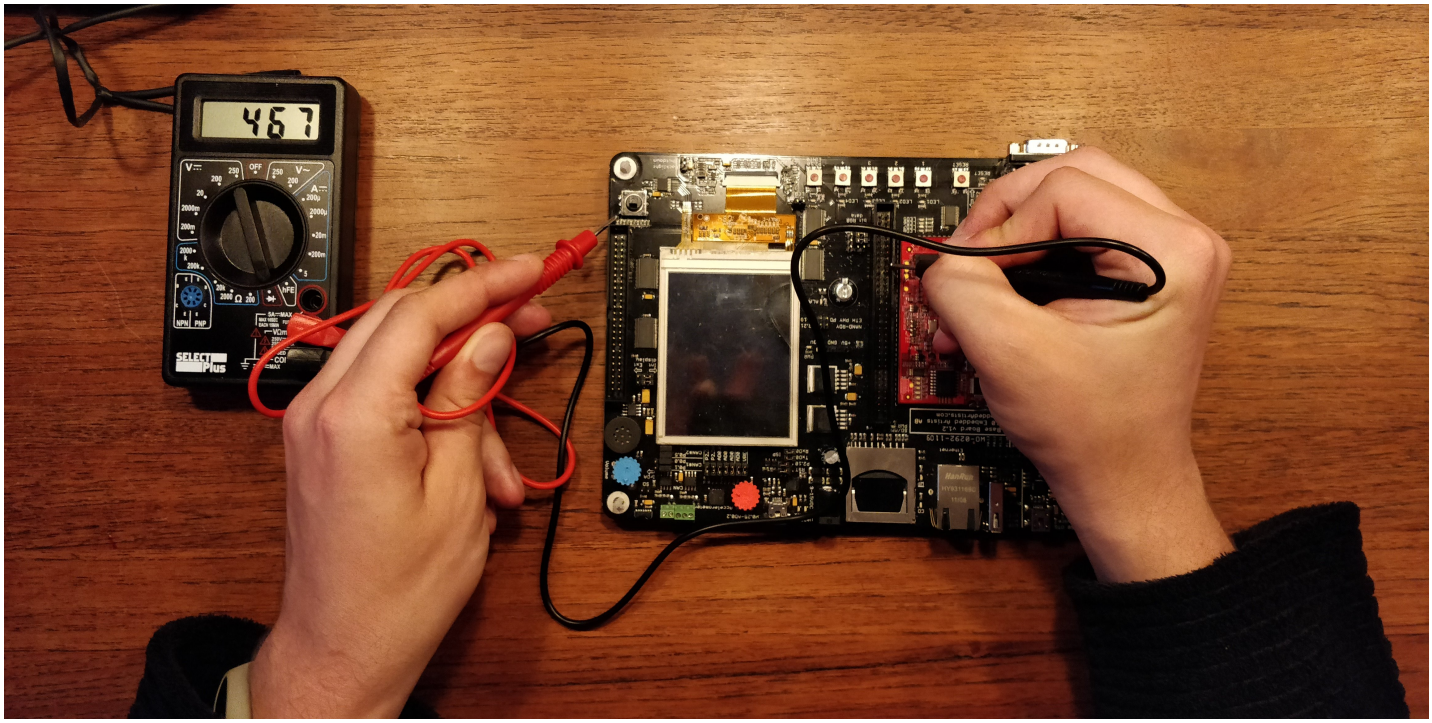
Controlling the GPIO pins on the LPC3250

Stefan Grimminck & Skip Geldens T62

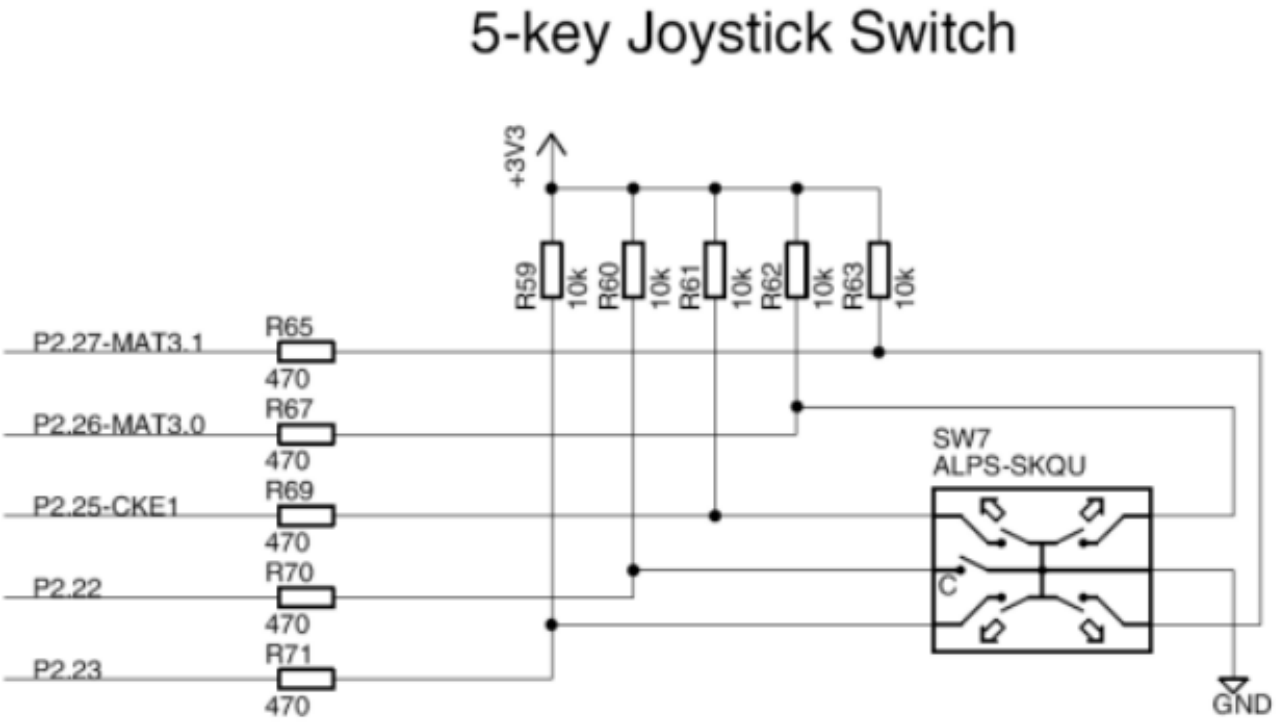
Part 1: Discovery & reading joystick values

We set out to make a kernel module for customer that wants to set the GPIO state, write values to the GPIO pins and read them. The customer doens't want to work with the registers itself, so our program is going to map the pins located on the board to the corresponding registers

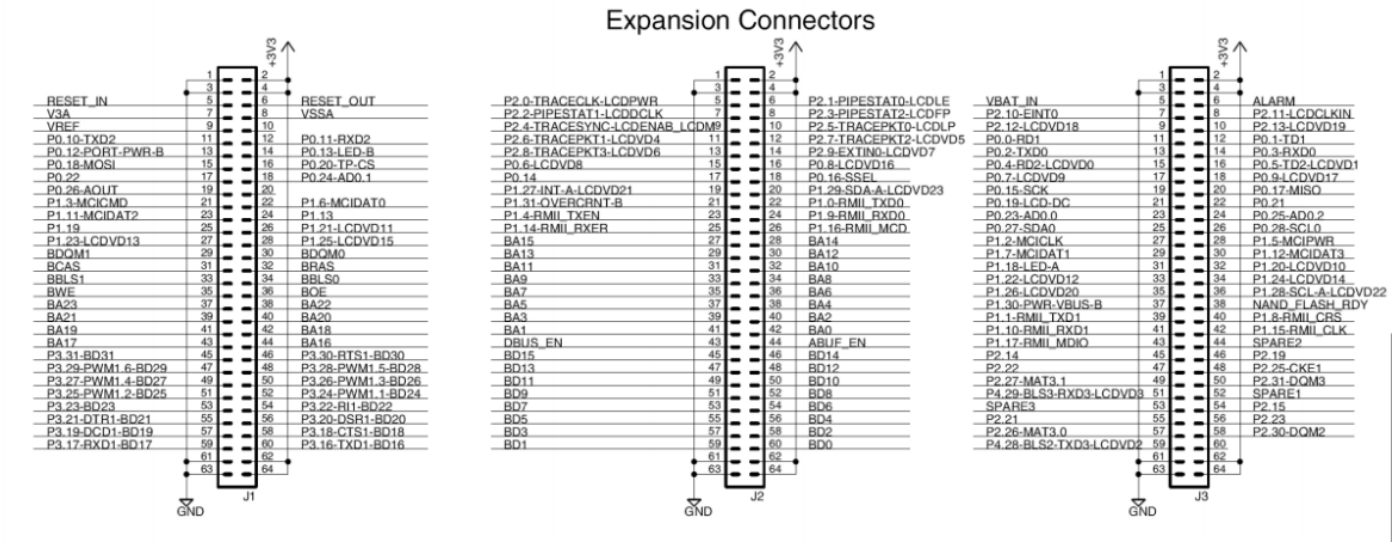
First, we start with connecting the joystick pins to the pins on the jumpers on the board to see where they're connected to:



As you can see in the image, the resistance between the joystick pin and the header pin is +/- 470 ohm, which corresponds with the following chart on sheet 4/8 of the "LPC3250 OEM Board User Guide":



The mapping of the ports are visualised in the diagram below:



Now we can determine that:

Port	pin	jumper	Joystick state
P2.26-MAT3.0	57	J3	R UP
P2.27-MAT3.1	49	J3	R DOWN
P2.25-CKE-1	48	J3	L UP
P2.23	56	J3	L DOWN
P2.22	47	J3	PUSH

Now we know where the joystick output is mapped to we can read the corresponding registers. According to Table 662 (Port 2 Multiplexer Set Register) in the UM10326 datasheet we need to set bit 3 to tell the multiplexer we want to use pins 31:19 as GPIO. Now that the pins can be used as GPIO, we need to set the P2*DIR*CLR register so that the GPIO pins are set as input. After we've done that we can read the corresponding pins:

OEM_PORT	OEM_PIN	CONNECTOR	PIN
2	27	3	49
2	26	3	57
2	25	3	48
2	23	3	56
2	22	3	33

When we read the values of these registers we can see that the register value changes:

```
# echo "r 3 49" > /dev/gpio
# cat /dev/gpio
Pin: 49 on connector 3 has been set to direction 0, with value 16
#
# cat /dev/gpio
Pin: 49 on connector 3 has been set to direction 0, with value 0
#
# echo "r 3 57" > /dev/gpio
# cat /dev/gpio
Pin: 57 on connector 3 has been set to direction 0, with value 8
#
# cat /dev/gpio
Pin: 57 on connector 3 has been set to direction 0, with value 0
#
# echo "r 3 48" > /dev/gpio
# cat /dev/gpio
Pin: 48 on connector 3 has been set to direction 0, with value 4
#
# cat /dev/gpio
Pin: 48 on connector 3 has been set to direction 0, with value 0
#
# echo "r 3 56" > /dev/gpio
# cat /dev/gpio
Pin: 56 on connector 3 has been set to direction 0, with value 2
# cat /dev/gpio
Pin: 56 on connector 3 has been set to direction 0, with value 0
```

note: initially this work was done with peek & poke

Part 2: Creating a kernel module for reading and writing from and to the GPIO pins

The LPC3250 has GPIO aswel as GPO and GPI port. The program we created only uses the GPIO pins. These pins are described in Table 613, 614 and 615 of the UM10326 datahsheet:

Table 613. Port 0 GPIO pin description

Pin name	Type	Description
p0[7:0]	I/O	General purpose input/outputs P0.0 through P0.7.

Table 614. Port 1 GPIO pin description

Pin name	Type	Description
P1[23:0]	I/O	General purpose input/outputs P1.0 through P1.23. (multiplexed with EMC_A[23:0])

Table 615. Port 2 GPIO pin description

Pin name	Type	Description
P2[12:0]	I/O	General purpose input/outputs P2.0 through P2.12. (multiplexed with EMC_D[31:19]

Port	GPIO Pins
P0	P0.0 - P0.7
P1	P1.0 - P1.23
P2	P2.0 - P.12

Before we can work with the pins we first have to map our physical pins to their port and corresponding register. When doing this, we noticed that not all GPIO pins are handled the same way. First we have to disable the LCD for P0.0 to P0.7 to work. We do this by setting the LCDCLK_CTRL to 0. We also noticed that the GPIO ports on P1 are not usable, because these are used as address bus of the RAM (see table 91 in UM10326 datasheet).

So now we are left with the following ports

Port	Connector	Pin		Port	Connector	Pin
P0.0	2	40		P2.10	1	51
P0.1	2	24		P2.11	1	52
P0.2	2	11		P2.12	1	53
P0.3	2	12		P2.2	3	48
P0.4	2	13		P2.3	3	57
P0.5	2	14		P2.4	3	49
P0.6	3	33		P2.5	3	58
P0.7	1	27		P2.6	3	50
P2.0	3	47		P2.7	3	45
P2.1	3	56		P2.8	1	49
P2.9	1	50				
GPIO_0	3	54				
GPIO_1	3	46				
GPIO_5	3	12				

These ports have to be mapped with the corresponding input, output & direction registers. In our code we've done this by using a structure named Pinfo;

```
typedef struct Pinformation{
    int pin;
    int jumper;
    struct DIR dir;
    struct OUTP output;
    struct INP input;
    int LOC_IN_REG;
}Pinfo;
```

This structure maps the pin on the board to the corresponding registers. As you can see this struct contains 3 other structures with registers for setting the direction, output and getting the input state of the ports. These are structured as follows:

```
struct DIR {
    unsigned int set;
    unsigned int clr;
    unsigned int state;
};

struct INP {
    unsigned int state;
};

struct OUTP {
    unsigned int state;
    unsigned int set;
    unsigned int clr;
};
```

By storing our data this way we can easily edit the registers of a physical pin, for example:

```
 *(unsigned int*)(io_p2v(pinformatie.dir.set)) =
 PIN_TO_BIT(pinfo.loc_in_reg.input_bit);
```

Here we set the direction of the pin to input by setting the correspoding bit (`PIN_TO_BIT(pinfo.loc_in_reg.input_bit)`) in register `Px_DIR_SET` .

How to use our software

The sys filesystem is used to set the io state of our pins, so that a GPIO pin can be set as input and as output. This is done in the following way:

```
echo "[i/o] [pin] [connector]" > /sys/kernel/gpio/data
```

- i means set the pin as an input pin
- o means set the pin as an output pin

The input value of the pins can be read or output value can be set by using the dev filesystem. This is done in the following way:

```
echo "[r/l/h] [pin] [connector]" > /dev/gpio
```

- r means to read
- l means set the pin low
- h means set the pin high

Testing

Ofcourse our kernel module has to be tested before we can give the application to the customer.

We tested this by pin 24 on connector output and setting it to high via the following commands:

```
# echo "o 2 24" > /sys/kernel/buffer/data
Pin: 24 on connector 2 has been set to output
#
# echo  "r 2 24" > /dev/gpio
#
# cat /dev/gpio
Pin: 24 on connector 2 has been set to direction 1, with value 0
# echo  "h 2 24" > /dev/gpio
BIT: 2
INSIDE REG: 0x40028044
Pin: 24 on connector: 2 is set
# echo  "l 2 24" > /dev/gpio
BIT: 2
INSIDE REG: 0x40028048
Pin: 24 on connector: 2 is cleared
```

which generated the following output: Youtube video

We also tested the reading capability by connecting +3.3v to a this pin. This resulted in a bit in the register set to high.

```
# cat /dev/gpio
Pin: 24 on connector 2 has value 0
# cat /dev/gpio
Pin: 24 on connector 2 has value 2
```

We also tested the other pins on the board in the same way.

References

Walls, C. (2014, July 27). Device registers in C. Retrieved May 14, 2018, from https://www.embedded.com/design/programming-languages-and-tools/4432746/Device-registers-in-C

Wells,K (2010).Linux kernel release 4.x GPIO-LPC32XX.c https://github.com/torvalds/linux/blob/master/drivers/gpio/gpio-lpc32xx.c

Wells,K (2010).Linux kernel release 4.x PLATFORM.h https://github.com/torvalds/linux/blob/5924bbecd0267d87c24110cbe2041b5075173a25/arch/arm/mach-lpc32xx/include/mach/platform.h

The Linux Kernel Programming Guide, visited on May 14, 2018, from https://fhict.instructure.com/courses/5839/pages/lkmpg-4-character-devices?module*item*id=219190

The Linux Kernel Programming Guide, visited on May 14, 2018, from https://fhict.instructure.com/courses/5839/pages/lkmpg-6-sysfs?module*item*id=219188