Fontys University of Applied Sciences

Software & Technology - 6th Semester

Assignment: **I2C**

Elviro Pereira Junior, **2719584**
Rafal Grasman, **289290**

Assignment A:
Write a (user space) program for a HW engineer.
A HW engineer is mainly focused what he can see on this oscilloscope. So he wants to have a program where he can enter the same bytes as he expects to see on the scope.
Moreover, he wants to use this program for all i2c devices he will ever encounter (not only the PCA9532). And the amount of bytes to read or write must be flexible (so not limited to what you see in the examples below). So the program must have a generic interface.

Examples (the command line parameters represent hexadecimal values):

- for writing to a device: ***$ i2c_hw c0 16 00 ff aa 55***

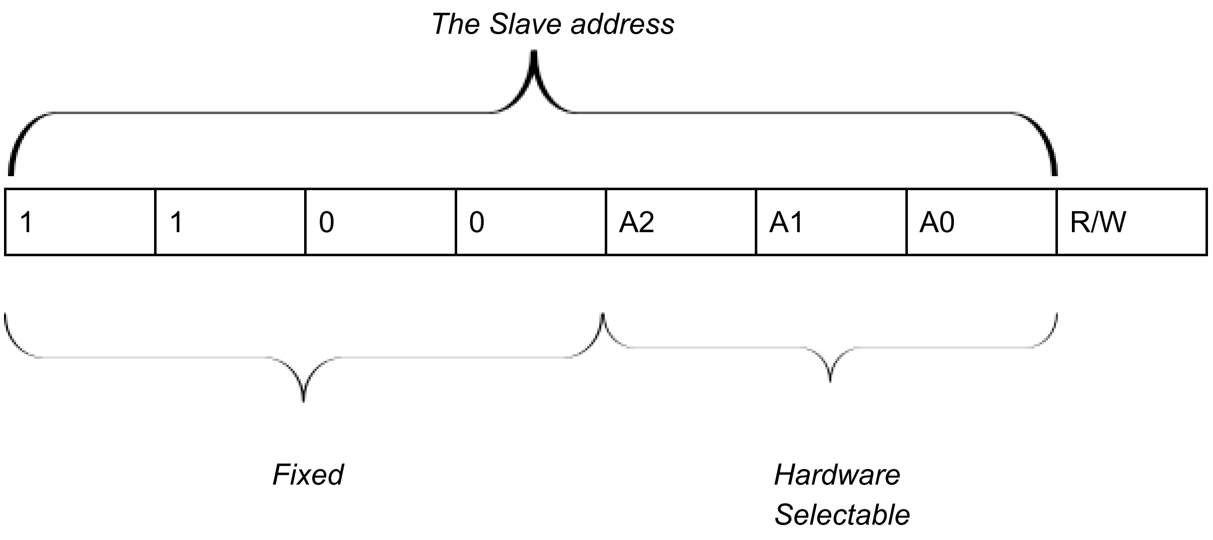
This means: i2c-slave c0 (the PCA9532) is addressed, and registers LS0, LS1, LS2, LS3 are written (check this with the PCA9532 Data Sheet).

- for reading from a device: ***$ i2c_hw c0 16 04***

This means: read 4 bytes from address 16 onwards of i2c-device c0.
The task of the user space program is to convert the user data into the correct device driver commands. (besides ioctl(), also multibyte read() and write() must be present in your implementation).
You may define yourself how the user enters his data; either on the command line (as given above),or graphically, or ....

Research Results:

Before starting the assignment the students have looked into the datasheet of PCA9532 to find out how the address is constructed, which bits can be changed and what do they mean.



The address above represents the device/slave address, in this case to save power pull-up resistors are not incorporated on the hardware selectable address pins and they must be pulled HIGH or LOW.

The last bit of the address represents an operation, if set to logic 0 it will execute the write operation and if set to logic 1 selects are read operation.

Design Decisions:

The program can be executed by:

./ass2-hw <bus> <r/w (read/write)> <slave address> <register>

And at the end, in case of a read:

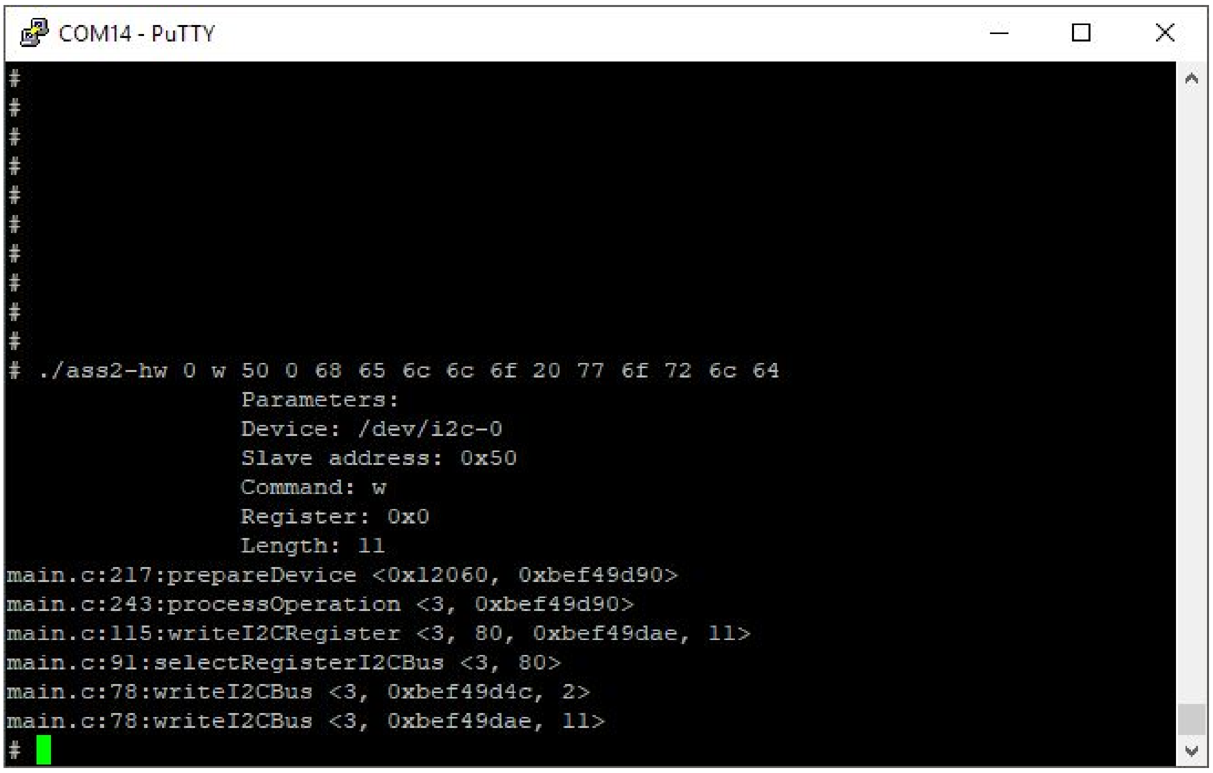
<the length>

In case of a write:

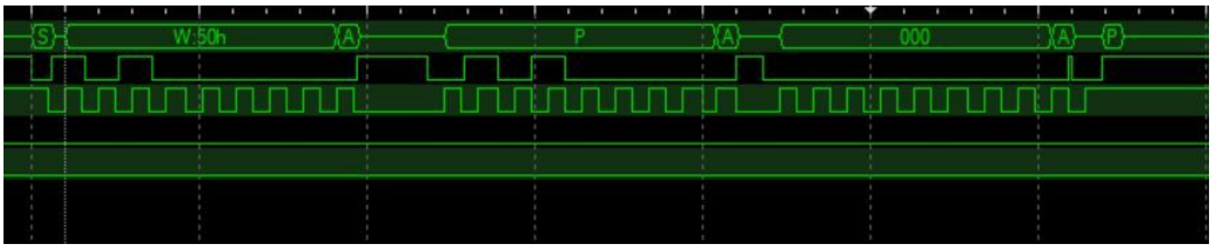
<byte1, byte2, …>

ScreenShots:

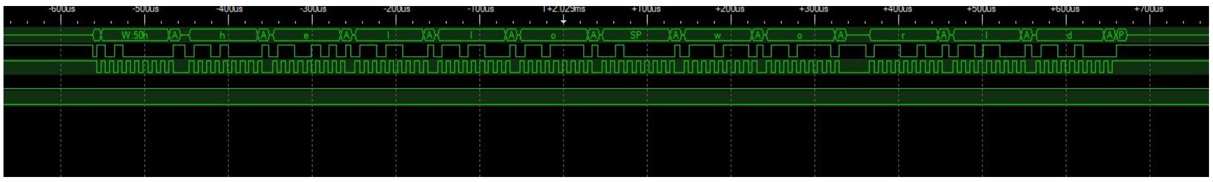
Sending “hello world” on bus 0 (/dev/i2c-0) to slave 0x50 and register 0x00:


COM14 - PuTTY

```
#
#
#
#
#
#
#
#
#
#
# ./ass2-hw 0 w 50 0 68 65 6c 6c 6f 20 77 6f 72 6c 64
                Parameters:
                Device: /dev/i2c-0
                Slave address: 0x50
                Command: w
                Register: 0x0
                Length: 11
main.c:217:prepareDevice <0x12060, 0xbef49d90>
main.c:243:processOperation <3, 0xbef49d90>
main.c:115:writeI2CRegister <3, 80, 0xbef49dae, 11>
main.c:91:selectRegisterI2CBus <3, 80>
main.c:78:writeI2CBus <3, 0xbef49d4c, 2>
main.c:78:writeI2CBus <3, 0xbef49dae, 11>
# 
```

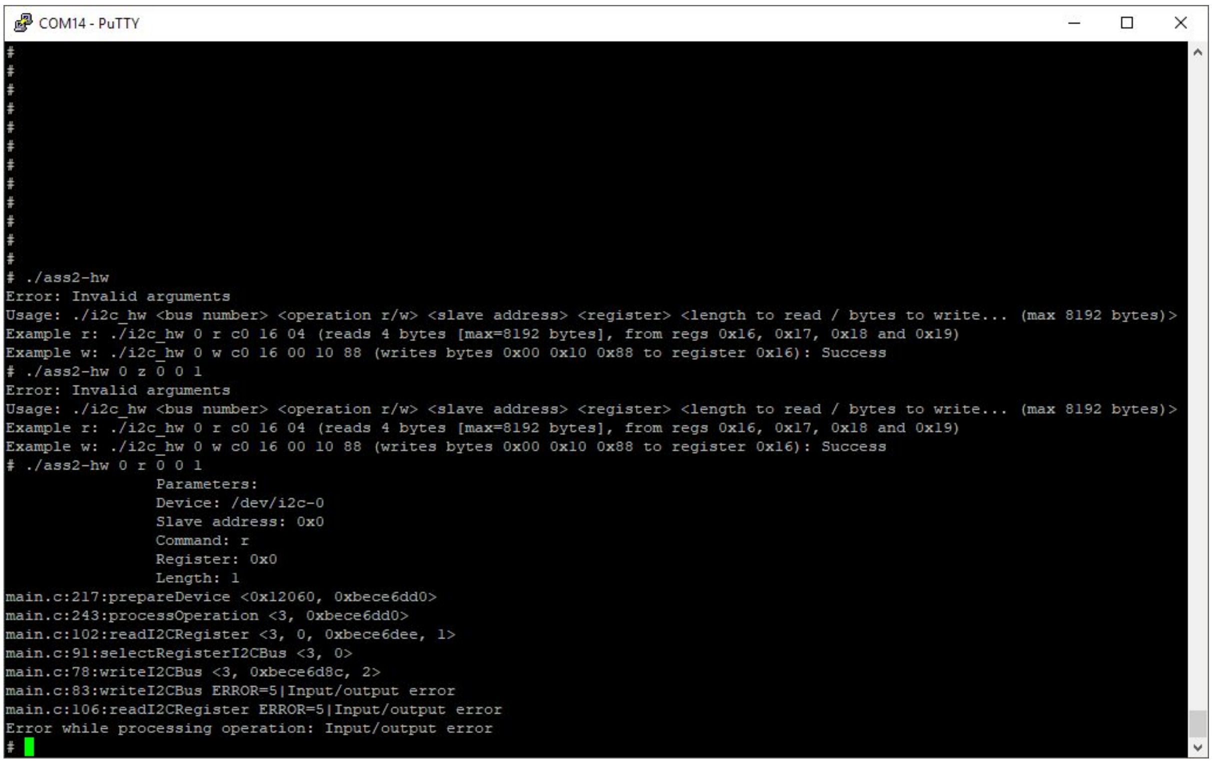


On the screen above we can see that we start our communication in the address 50 which is acknowledge and then the register address is selected a 0x00 which is also acknowledge. This indicates that we can select a slave and a register in the slave to which we want to read or write.



On the above screenshot we can see the data being write to register we picked previously. After every bit sent there’s an acknowledge to make sure the data is received by the slave.

Testing:



```
#
#
#
#
#
#
#
#
#
#
#
#
# ./ass2-hw
Error: Invalid arguments
Usage: ./i2c_hw <bus number> <operation r/w> <slave address> <register> <length to read / bytes to write... (max 8192 bytes)>
Example r: ./i2c_hw 0 r c0 16 04 (reads 4 bytes [max=8192 bytes], from regs 0x16, 0x17, 0x18 and 0x19)
Example w: ./i2c_hw 0 w c0 16 00 10 88 (writes bytes 0x00 0x10 0x88 to register 0x16): Success
# ./ass2-hw 0 z 0 0 1
Error: Invalid arguments
Usage: ./i2c_hw <bus number> <operation r/w> <slave address> <register> <length to read / bytes to write... (max 8192 bytes)>
Example r: ./i2c_hw 0 r c0 16 04 (reads 4 bytes [max=8192 bytes], from regs 0x16, 0x17, 0x18 and 0x19)
Example w: ./i2c_hw 0 w c0 16 00 10 88 (writes bytes 0x00 0x10 0x88 to register 0x16): Success
# ./ass2-hw 0 r 0 0 1
                Parameters:
                Device: /dev/i2c-0
                Slave address: 0x0
                Command: r
                Register: 0x0
                Length: 1
main.c:217:prepareDevice <0x12060, 0xbece6dd0>
main.c:243:processOperation <3, 0xbece6dd0>
main.c:102:readI2CRegister <3, 0, 0xbece6dee, 1>
main.c:91:selectRegisterI2CBus <3, 0>
main.c:78:writeI2CBus <3, 0xbece6d8c, 2>
main.c:83:writeI2CBus ERROR=5|Input/output error
main.c:106:readI2CRegister ERROR=5|Input/output error
Error while processing operation: Input/output error
#
```

To make sure are design rules are working we tried to pass wrong arguments and test the handling of errors cause by those arguments.

Assignment B:
Write a (user space) program for an application engineer. An application engineer (working with the PCA9532) talks in terms of leds (1 till 8), blinking frequencies (in Hz) and dimming rates (in %). Write a user space program for him so he doesn't need to know anything about internal registers or hexadecimal values. Define the interface yourself. This might be on the command line, or interactive; as long as it is convenient for the target audience.

Research
The PCA9532 i2c slave has a few registers that are important to controlling the leds. First to be able to control which leds are on or off, there are two registers (from PCA9532 datasheet) connected on the development board (Note, on the board leds are named starting from 1, LED1 on board corresponds to LED0 here, and so on):

LS2: 0x08

00 (LED3)	00 (LED2)	00 (LED1)	00 (LED0)

LS3: 0x09

00 (LED7)	00 (LED6)	00 (LED5)	00 (LED4)

In each register, two bits determine what happens to a LED:

00 (0): OFF
01 (1): ON
10 (2): Blink with PWM0 frequency
11 (3): Blink with PWM1 frequency

This can be controlled easily by OR'ing a byte value with such a define (with LED4 being also index 0, LED5 index 1, and so on):

```
#define PCA9532_SET_LED(index, mode) ((mode) << ((index) * 2))
```

We need to control the power the leds illuminate at, this can be simulated with PWM by making the PWM frequency high (PSC0 register at 0x02) and setting the desired duty cycle (PWM0 register at 0x03).

To be able to blink the leds visually to a human, the program will turn the LEDs ON/OFF/... repeatedly at the desired blinking frequency (if it's 0 the leds stay on). This can be done by implementing a while loop which sends the correct on or off commands to the slave, and sleeping for a determined amount of time to achieve the desired frequency:

```
bool turn_on = true;
do
{
        for (int i = 0; i < MAX_LED_SELECTORS; ++i)
        {
                i2c_smbus_write_byte_data(*device_handle_ptr,
                                      params->selectors[i].reg,
                        turn_on ?
                        params->selectors[i].val :
                        PCA9532_LEDMODE_OFF
                );
        }

        turn_on ^= 1;

        nanosleep(&params->frequency_spec, NULL);
} while (params->frequency != 0 && stop == 0);
```

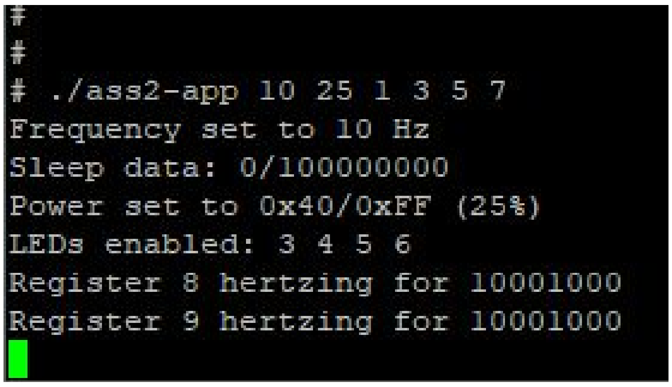
Design Decisions
To make the program easy to use it can be simple used as:

Usage: ./ass2-app <hz> <power %> [<leds>...]
Example: ./ass2-app 1.5 10 0 2 4 6
To blink leds 0 2 4 and 6 at 1.5 Hz at 10% duty cycle

Without providing LEDs (only frequency and power), all LEDs will be affected.

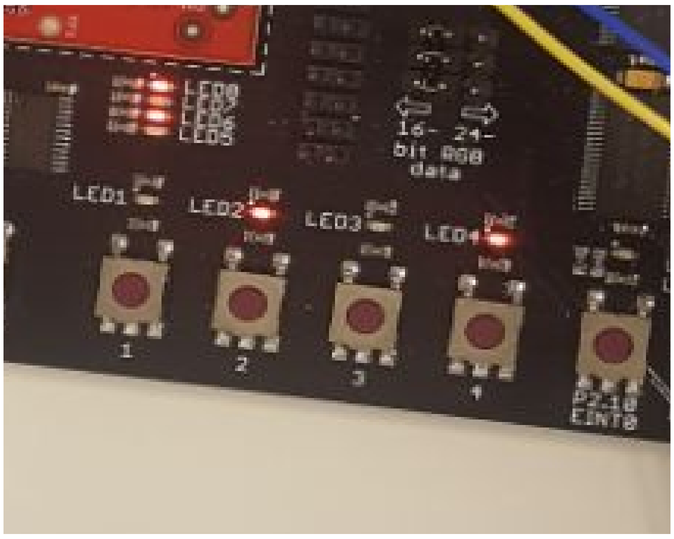
The application provides comprehensive information as to what has been input as can bee seen in testing.

Testing
Running the application (10 hz, 25% power):

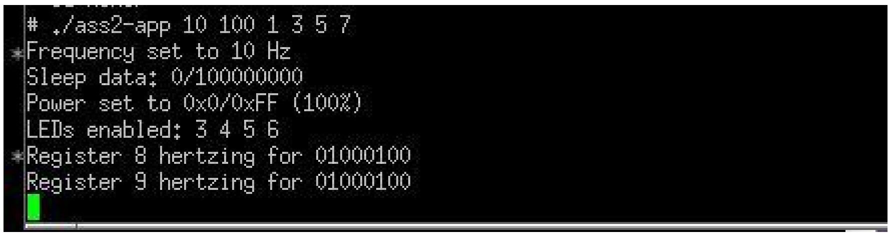


```
#
#
# ./ass2-app 10 25 1 3 5 7
Frequency set to 10 Hz
Sleep data: 0/100000000
Power set to 0x40/0xFF (25%)
LEDs enabled: 3 4 5 6
Register 8 hertzing for 10001000
Register 9 hertzing for 10001000
```

The LEDs blink at a 10hz rate and are illuminated at 25% power:



Compared to 100% power:
Running the application (10 hz, 100% power):



```
# ./ass2-app 10 100 1 3 5 7
Frequency set to 10 Hz
Sleep data: 0/100000000
Power set to 0x0/0xFF (100%)
LEDs enabled: 3 4 5 6
Register 8 hertzing for 01000100
Register 9 hertzing for 01000100
```

The LEDs blink at a 10hz rate and are illuminated at 100% power: