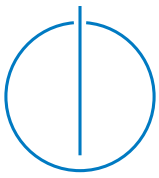
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Practical Course - Linux and L4 µ-Kernel

Documentation of Group 2

Author:	Tamas Bakos, Gurusiddesha Chandrasekhara, Moritz Dötterl, Patrick Langer, Johannes Ismair, Natalie Reppekus
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Daniel Krefft
Submission Date:	31.7.15



Contents

1 Introduction

This project has been implemented as part of the "Linux and L4" lab course of the Operating Systems Department of TUM Informatics in SS2015. The purpose of the project was to build up a setup consisting of a PandaBoard, Olimex STM32-H103 ARM development board and a LSM9DS0 multisensor and write a firmware for the Olimex, which will transmit the sensor data to the PandaBoard, develop a basic driver for Genode in the DDE, which will run on the PandaBoard, to collect and process the sensor data, and send it further to a PC. However, the initial plan of writing a driver using DDE was abandoned since the Genode environment did not offer specific components which were needed and the development would not have been possible in the given time.

The complete project was broken down into phases, with each phase having specific milestones, which contribute towards the final project. The first milestone involved developing basic functions to read the incoming sensor data using `libopencm3`, the Genode UART implementation and the Genode driver development environment setup. The second milestone was to write a simple application that should run on the Panda Board and developing a "hello_world" driver for DDE. The last milestone of this project was to develop a simple Linux USB driver and an application to read out the sensor data.

The report is organized into chapters describing the milestones in each project phase. In chapter 2 we describe the I2C connection between the sensor and the Olimex and how data is transmitted between these two, in chapter 3 we describe compiling firmware and flashing it on to Olimex. chapter 4 explains how to establish a USB connection between the Olimex and a computer, chapter 5 provides step-by-step instructions to compile Fiasco.OC, Genode, and a self written application for Genode, chapter 6 explains setting up two UART connections and the application which uses two UART, chapter 7 explains the development and usage of USB driver. Setting up a PC web application to read and visualize the sensor data is explained in chapter 8 Following all chapters are listed. To directly jump to each chapter click on the chapter number in brackets after the description.

1. Make I2C connection between sensor and `Olimex` (chapter 2).

2. Flash the Olimex with firmware using `openocd` (chapter 3).
3. Make USB connection between Olimex and Linux host system (chapter 4).
4. Load USB driver (`accel.ko`) onto host system (chapter 7).
5. Set up PC web application `teapot` in order to visualize the data from the sensor (chapter 8).

2 I2C connection to the sensor - Patrick Langer

This chapter describes the I2C connection which is used to connect the sensor to the Olimex and the functions which let you modify the sensor's settings as well as receive data from it. The code can be found in our github repository under `praktikum/src/LSM9DS0_usb/lsm9ds0.c` During the development we mostly oriented ourselves by the `srf10` source code provided in moodle, adjusting it to our needs.

2.1 Finding the Right Device Address

One of our longer lasting problems we encountered during the lab was to iniate communication with the sensor. The `LSM9DS0` has two device addresses, one for the linear acceleration and magnetic sensor and one for the gyrometer. Each of them can be modified by connecting the corresponding `SD0/SA0` pin to the voltage supply, according to the `LSM9DS0` manual. Since we did not connect these pins we used the accelerometer's device address shifted one bit to the left, logical or one to add the read bit, as illustrated in the last column of the first row in Figure 2.1, which results in an eight bit address. After that we passed this address as a parameter to the

Command	SDO_XM/SA0_XM pin	SAD[6:2]	SAD[1:0]	R/W	SAD+R/W
Read	0	00111	10	1	00111101 (3D)
Write	0	00111	10	0	00111100 (3C)
Read	1	00111	01	1	00111011 (3B)
Write	1	00111	01	0	00111010 (3A)

Source: `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00087365.pdf`

Figure 2.1: Linear acceleration and magnetic sensor SAD+read/write patterns

function `i2c_send_7bit_address` . After sending a start signal followed by calling `i2c_send_7bit_address` we did not get a response from the sensor and our program got stuck.

Our first approach to solve this was to use the address, which is used when the `SD0/SA0` pins are connected to voltage, row three and four, but as it turned out this did

not work either since our problem lay somewhere else. So after that we tried to get the right address by bruteforcing it. We sent a start signal followed by any address ranging from 0x00 to 0xFF. This attempt to solve the problem did not help us out much either since our program did not advance any further than before. My guess after thinking about it is that a stop signal has to be issued after sending the device address and not receiving a response after a certain amount of time. Thinking the sensor was simply faulty we run some arduino code which actually output some data.

Eventually our problem consisted of passing an eight bit address as a parameter instead of the requested seven which we found out after reading through the `libopencm3` documentation again. On top of that the adafruit `LSM9DS0` listens to the addresses 0x1D for the accelerometer and magnetometer and 0x6B for the gyrometer by default, which are the addresses used when the `SD0/SA0` pins are connected to voltage.

2.2 Wiring

During our device address problem mentioned above we tried out different wiring patterns, since we had a problem with the UART wiring when first connecting the `Pandaboard` to a computer. Ultimately we had to take another look at the adafruit homepage [1] to make sure everything is wired up correctly.

- `VIN` is connected to the Olimex's 3.3V power supply
- `GND` is wired to any available `GND`
- `SCL` on the sensor is connected to `SCL` on the Olimex, same goes for `SDA`

The sensor is connected to the I2C interface one on the Olimex since that is the interface we use in our main program.

2.3 I2C Connection in General

Iniating a transfer is the same for reading or writing to the sensor. It starts off by sending `i2c_send_start`, waiting until the start signal is sent and the Olimex is switched to master mode, suceeding `i2c_send_7bit_address(i2c, deviceAddress, I2C_WRITE)` where `deviceAddress` is either 0x1D or 0x6B. After that we have to wait again until the address is transferred and the address condition sequence is cleaned. `i2c_send_data( i2c, reg)` must be used to tell the sensor to which register we want to write, followed by another wait until the `I2C_SR1_BTF` flag is set.

[1] `https://learn.adafruit.com/adafruit-lsm9ds0-accelerometer-gyro-magnetometer-9-dof-breakouts/wiring-and-test`

In the case of writing a byte another call of `i2c_send_data(i2c, value)` lets us send our desired value and `i2c_send_stop` ends the transfer. The writing operation is implemented in the function `void lsm9ds0_set_register(uint32_t i2c, uint8_t sensor, uint8_t value, uint8_t reg)`.

If we want to receive a byte another call of `i2c_send_start`, as well as of `i2c_send_7bit_address(i2c, deviceAddress, I2C_READ)` is necessary. This time the last parameter is changed to `I2C_READ`. When the address is transferred again the byte can be read using `i2c_get_data`. In comparison to the writing transfer `i2c_nack_current(i2c)` has to be called before `i2c_send_stop(i2c)` to stop the sensor from sending more data and end the transfer. At which point a no master acknowledge has to be issued was a problem for us, since `libopencm3` documentation did not help us much but simply using trial and error allowed us to solve it in a reasonable amount of time. One byte can be read by calling `uint8_t lsm9ds0_read_data1B(uint32_t i2c, uint8_t sensor, uint8_t reg)`.

2.4 Receiving Sensor Data

By default the temperature sensor is enabled while the other sensors are set to power-down mode. As a result the appropriate registers have to be configured to power on all sensors before any data can be read. The functions `lsm9ds0_enable_x(uint32_t i2c)` where x is any sensor, take care of that. Receiving sensor data goes straight forward. The sensor supports auto incrementing the register address by sending the register address logical or 0x80, as can be seen in Figure 2.2. This allowed us to receive the x, y and z coordinates without having to stop and restart the connection after every byte transfer. Each of the coordinates are returned as two's complement so we cast them to a signed `int16_t`. When reading any of the data, besides the temperature, we simply use `i2c_get_data` twelve times, take care of the endiannes for every coordinate and send a stop signal in the end. Receiving the last two bytes is an exception because `i2c_nack_next` and `i2c_disable_nack` has to be called to stop the sensor from sending data after getting the second byte of the z-coordinate. The temperature sensor is a special case because the temperature value only consists of 14 bit of data, so we implemented an extra function for it and return the reading as an `int16_t`.

2.5 Full-Scale Selection

After receiving the raw sensor data in the right format we went on to the Conversion from raw data to useable data. First of all the gyrometer, accelerometer and magnetometer offer you the choice between three to five different full-scales which can be

```
void lsm9ds0_read_data6B(uint32_t i2c, uint8_t sensor, uint8_t reg,
                         int16_t *result) {
      ...
      i2c_send_data(i2c, reg | 0x80); /* | 0x80 to auto-increment the register*/
      ...
      for (i = 0; i < 2; i++) {
       while (!(I2C_SR1(i2c) & I2C_SR1_BTF));
       result[i] = 0;
       result[i] = i2c_get_data(i2c);
       /* Now the slave should begin to send us the first byte. Await BTF. */
       while (!(I2C_SR1(i2c) & I2C_SR1_BTF));
       result[i] |= (i2c_get_data(i2c) << 8);
      }
      /*Last 2 bytes transfer is special because of NACK*/
      i2c_nack_next(i2c);

      while (!(I2C_SR1(i2c) & I2C_SR1_BTF));
      result[2] = 0;
      result[2] = i2c_get_data(i2c);
      i2c_disable_ack(i2c);
      result[2] |= (i2c_get_data(i2c) << 8);
      ...
}
```

Figure 2.2: Receiving the first four byte

set in each of the sensor's configuration register. we employed adafruit's [2] approach which reads out the current values of the register and buffers them, as other settings besides the full-scale can be set in the registers. Next the desired full-scale value is added to the buffered values with a logical or and sent back to the register. As we had problems sending the converted data via USB, the code suceeding `switch (scale)` in Figure 2.3 is not in use in the final version but can be used in combination with the out commented code in `lsm9ds0_read_data6B` to let the Olimex handle the conversion. The code basically sets a factor, depending on the selected full-scale, which is multiplied with the raw data. For the other two sensors it works analogue.

[2] `https://github.com/adafruit/Adafruit_LSM9DS0_Library`

```
void lsm9ds0_set_accel_scale(uint32_t i2c, lsm9ds0AccelRange_t scale) {
  uint8_t reg_value;
  reg_value = lsm9ds0_read_data1B(i2c, LSM9DS0_SENSOR_XM, CTRL_REG2_XM);
  reg_value &= ~(0b00111000); /*keep the current value, set current scale to 0*/
  reg_value |= scale;
  lsm9ds0_set_register(i2c, LSM9DS0_SENSOR_XM, reg_value, CTRL_REG2_XM);

  switch (scale) {
  case LSM9DS0_ACCELRANGE_2G:
    _accel_mg_lsb = LSM9DS0_ACCEL_MG_LSB_2G;
    break;
  case LSM9DS0_ACCELRANGE_4G:
    _accel_mg_lsb = LSM9DS0_ACCEL_MG_LSB_4G;
    break;
  case LSM9DS0_ACCELRANGE_6G:
    _accel_mg_lsb = LSM9DS0_ACCEL_MG_LSB_6G;
    break;
  case LSM9DS0_ACCELRANGE_8G:
    _accel_mg_lsb = LSM9DS0_ACCEL_MG_LSB_8G;
    break;
  case LSM9DS0_ACCELRANGE_16G:
    _accel_mg_lsb = LSM9DS0_ACCEL_MG_LSB_16G;
    break;
  }
}
```

Figure 2.3: Setting the accelerometer full-scale value

2.6 Conclusion

In conclusion this chapter has shown how important it is to read the documentation thoroughly when using third party libraries so you do not end up looking for an error for hours but instead pass the right parameters from the beginning. Communicating with a device using I2C can be complicated if you are new to it but once you understand the basic concept of it you will progress quickly. If we could send the converted data without problems, the full-scale could be selected with one simple change and allow easier conversion in place of letting an application handle it.

3 Flash the Olimex - Moritz Dötterl

3.1 Intorduction

The device called "the Olimex" in this document is an STM32-H103. It is a small ARM Cortex-M3 32 bit based development board, manufactured by a company called Olimex. In the project it was used to build the connection between the sensor and the BananaPi/Pandaboard/Laptop. It offers many different bus systems including USB, CAN, I^2C, ADC, UART, SPI. In our development setup we used I^2C to connect the sensor to the Olimex. To connet the Olimex to the BananaPi/Pandaboard/Laptop we used UART at first and later on USB.
The Olimex is JTAG flashable and there are many different tools and ways to flash the Olimex. But this is not only a benefit, but can also cause trouble. During the course we had some problems with the Olimex. This Chapter focuses on problems we had in the beginning of the course as we tried to find out how to flash the Olimex. It turned out that many things can be made wrong and that it needs some time to find out how everything needs to be set up.
In the end we used an ordinary text editor to write the code, using the libopencm3. We compiled the code with the GCC ARM cross compiler and flashed the Olimex manually using Telnet and OpenOCD. This setup was the only one working for us and took some time to figure out how to do it. This chapter deals with the problems we had regarding the flashing of the Olimex.

3.2 Problem 1: Find a Way to Flash the Olimex

First of all we needed to find a way to Flash the Olimex, that is working for us. There are many different ways to compile code for the Olimex and flash it into its memory. But none did really seem to work for us at first. So we spent quite a while to search for a working method and tried many different tools and approaches. Actually we used the right tool in the beginning, but did not really realized how it worked, so we where searching for a different tool. In the end we tried again OpenOCD and slowly figured out how it was working and how we could flash the file manually. This actually worked really well, but is a little bit tricky in the beginning. How this is done exactly is described in the following section 3.3.

3.3 Problem 2: Understand the Functionality of OpenOCD

The functionality of OpenOCD is rather complex at the beginning and it took us some time to realize how it works. The main questions we had were: How is the connection established? How does OpenOCD communicate with the Olimex and how do we communicate with OpenOCD? What kind of file do we need to generate to flash it to the Olimex? Figure 3.1 illustrates how the connection between the commands the

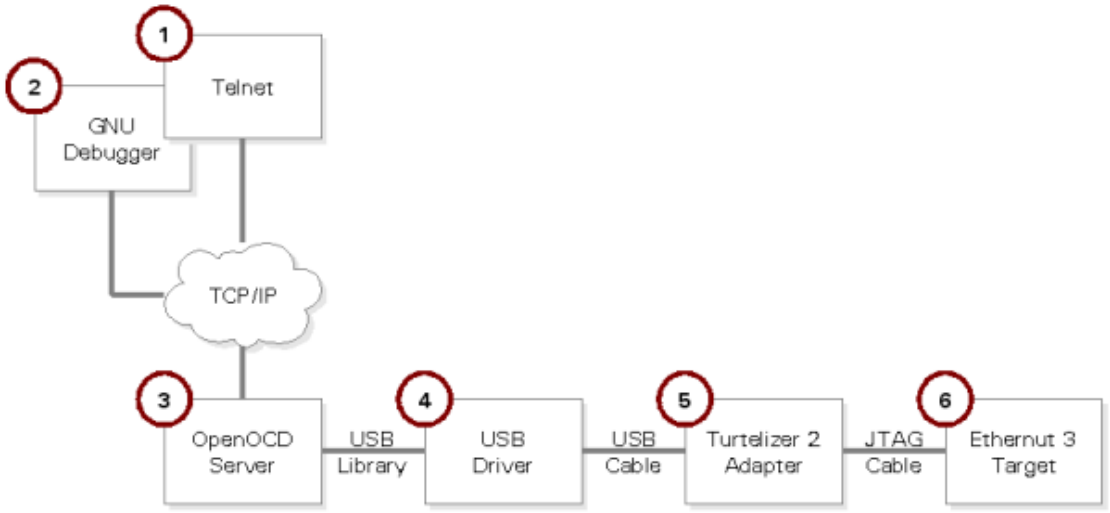


Figure 3.1: Functionality of OpenOCD

programmer runs and the Olimex itself is build up. The Target(6) in the figure is the actual Olimex, that is connected with a JTAG cable to an adapter device(5). This adapter device is connected via USB to your laptop running a Linux system, in our

case most of the time a KUbuntu 15.4. The USB driver(4) in the Linux system handles the communication with the adapter and provides a device file. Once the USB cable is connected to the laptop OpenOCD needs to be started in a terminal. The OpenOCD program(3) uses USB Libaries and this device file to establish a connection to the Olimex. On the other side the OpenOCD program provides a Telnet and a GDB server on the localhost. There should be a way to automatically flash the compiled code, but none of those methods worked for us, we always got errors. So after a while and after testing many different methods we found a instruction how the Olimex can be flashed manually. In the following this method is described. In a different terminal window a Telnet connection to localhost on a predefined port(in our case 4444) has to be established. In this Telnet connection the actual commands can be run. At first you should make sure you are in the correct directory, by default the Telnet directory is the same directory where OpenOCD was started. To change the directory the normal *"cd"* command can be used, to print out the whole path of the current directory there is the *"pwd"* command. Unfortunately there is no command to list all files in the current directory which is quite annoying. To begin with the flashing procedure the processor of the Olimex first needs to be stopped with the command *"reset halt"*. Now the actual flashing can happen. For this a .hex file needs to be created by the compiler. This is a binary file (in a hexadecimal representation) of the code that should run on the Olimex board. How this file is created can be seen in section 3.5. In the Telnet terminal the command *"flash write_image erase foobar.hex"* needs to be executed, where *"foobar.hex"* is the name of the binary .hex file that should be flashed on the Olimex. When this command was executed without any error the Olimex is ready to use. However unless it looses the power connection the processor of the Olimex needs to be started again. This can be done by either pressing the reset button on the board or by entering the command *"reset"* in the Telnet session. Now the Olimex is ready to go and starts executing the code.

3.4 Problem 3: Config File

A big problem we had was the config file. If you try to flash the Olimex and you are actually doing everything right but the config file is wrong it would not work. In fact OpenOCD will not even start if you do not provide a valid config file. The mean thing is that even if OpenOCD starts it does not mean that the config file is correct. This makes it harder to troubleshoot whether the config file is wrong or the flashing itself is somehow not working or wrong.
We tried different config files we found in the internet, that should work with our setup. In the config file several things are set up. At first the ports for the Telnet and

```
# daemon configuration
telnet_port 4444
gdb_port 3333
# interface configuration
interface ftdi
ftdi_device_desc "Olimex OpenOCD JTAG ARM-USB-OCD-H"
ftdi_vid_pid 0x15ba 0x002b
ftdi_layout_init 0x0c08 0x0f1b
ftdi_layout_signal nSRST -oe 0x0200
ftdi_layout_signal nTRST -data 0x0100 -noe 0x0400
ftdi_layout_signal LED -data 0x0800
# board configuration
set WORKAREASIZE 0x5000
reset_config none
# target configuration
source [find target/stm32f1x.cfg]
```

Figure 3.2: OpenOCD Config File

the GDB daemons are defined. After this the interface is configured, in our case we use a ftdi interface. The device descriptor is set up, and the vendor and product id of the device we want to connect to is specified. After that some signals are defined. Following this the board specific configurations are made: Since we are using a board with 20KB of RAM (here called Work-area size) we need to set the WORKAREASIZE to 0x5000. On other boards with e.g. 64KB RAM this would needed to be set to 0x10000 or a different appropriate value. The *"reset_config none"* produces a single reset via the JTAG command SYSRESETREQ at the reset pin. In the last line OpenOCD is asked to use target configurations as described in another file called stm32f1x.cfg. We use this file, because our Olimex is a STM32-H103 which belongs to the family called f1.

3.5 Problem 4: Compile

Also the compilation of the produced source code for the Olimex made some problems. In the example code indeed a Makefile is provided, but it makes some trouble as well. First of all it is divided in three different files: The Makefile, a Makefile.include and a Makefile.rules. The Makefile includes the Makefile.include which again includes the Makefile.rules. These three files are located in different folders and are a little bit annoying when trying to put them all in one directory because every file needs to be

adopted. To compile the source code using make a GCC ARM cross compiler is needed. However if you run *"make"* it will compile the code, but it will not generate the .hex file that is needed to flash the compiled code on the Olimex. It took us some time to figure out how the .hex file is created, but in the end the solution is rather simple. *"make flash"* will compile the code and generate the needed .hex file. The annoying thing about this is that the *"make flash"* command also tries to directly flash the .hex file on the device. The problem is that this automatic flashing is not working and therefore will always print out an error, even if the compilation itself was successful. This can get quiet confusing, especially in the beginning.

3.6 Problem 5: libopencm3

libopencm3 is a project that aims to create a open-source firmware library for ARM Cortex-M based microcontrollers. In this project it was used to establish and utilize the I^2C, USB and UART connections. Of course we needed some time to understand what this library is, how it is used and what it can do for us. After following the install guide we thought we could use the library out of the box. But that did not work. We had some problems with it as well. The source code would not compile, mainly due to the following two problems.
First of all we had to adjust some variables in the Makefile.rules file, since the directory structure of the installed libopencm3 is slightly different than the directory structure expected in the original Makefile.rules. Figure 3.3 shows the modified variables. In the original Makefile.rules all of this three variables where set to *$(OPENCM3_DIR)*.

```
INCLUDE_DIR = $(OPENCM3_DIR)/include
LIB_DIR  = $(OPENCM3_DIR)/lib
SCRIPT_DIR = $(OPENCM3_DIR)/share/libopencm3/scripts
```

Figure 3.3: Changes in Makefile.rules

This brings us direct to the next problem we had, which actually was not too bad, but for sure a little bit annoying.
The environment variable *OPENCM3_DIR* needs to be set to the path where the libopencm3 is installed, in our case this would look like this:
"export OPENCM3_DIR=/usr/local/arm-none-eabi". Unfortunately this environment variable was not set during the installation of the libopencm3. Trying to compile code using make will fail if the variable is not set manually. Since I never managed to export a environment variable permanently we always needed to export the variable every

time we used a new terminal window to compile the code. Later on I figured out how to tell my terminal to automatically export a variable on startup of every new window. This deviantly made things easier.

3.7 Conclusion

To sum this up: Many things can be made wrong, without even writing a single line of code, but instead rather trying to compile and flash a actually working example. As soon as you figured out how to do it there is no problem whatsoever. But until you know how things are working everything always seems complicated and it is a time intense job to get to know how everything works. But in the end it is interesting to see how happy a simple blinking LED can make you, because you finally where able to compile a simple example code and flash it on the Olimex.
This chapter described how the source code for the Olimex is compiled and loaded into the flash memory of the Olimex board. Also all the problems we had and that took us some time in the beginning of this course where described. Later on in the course we needed the knowledge of this chapter often, since we often compiled code and flashed it on the Olimex. At least every time we meet in the lab we had to flash our code on the Olimex. So even thou it took some time to get to know how this works it was totaly worth it because we used it often.

4 USB Connection to the Olimex - Natalie Reppekus

In this chapter the concept of establishing a USB connection from the Olimex to a computer is described. This allows to transmit the data, arriving at the Olimex from the LSM9DS0 sensor, to a computer. First, this was intended for debugging purposes. After the change in the task to not write a driver that should run on the Panda Board but on a PC with a Linux operating system installed, this connection is now needed for the final setup. In Figure 4.1 the setup referred to in this chapter is shown. The functionality is implemented in the C-file `main.c` using `libopencm3` [1], which can be found under the following path in the git repository of group 2: `praktikum/src/LSM9DS0_usb`

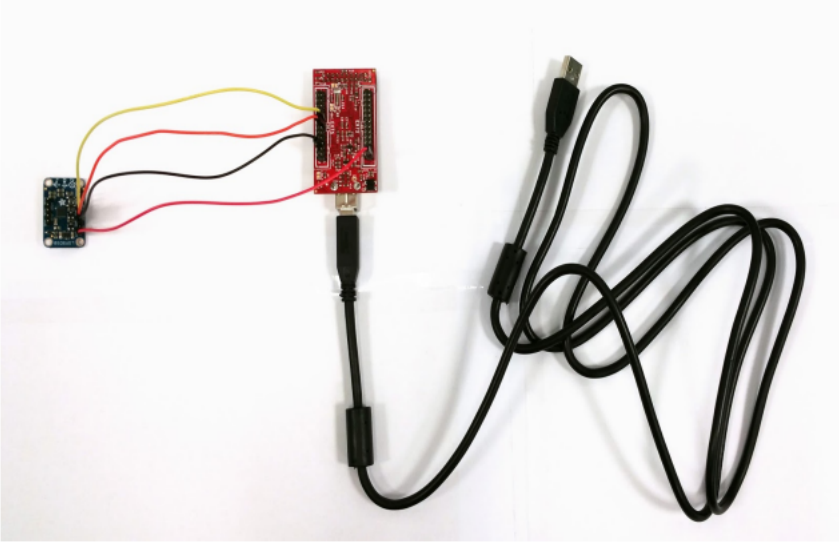
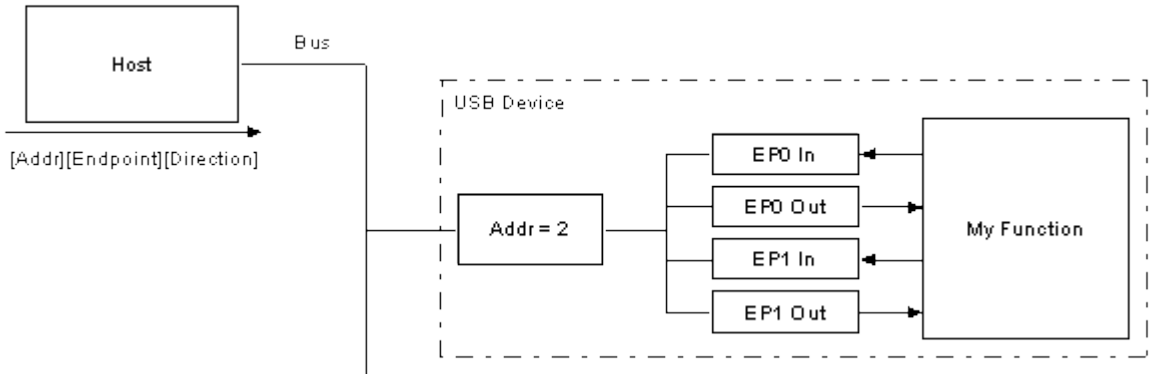


Figure 4.1: LSM9DS0 connected to the Olimex, Olimex with USB cable for connection to PC

[1] `http://libopencm3.org`

4.1 Basic USB Connection

Our first problem was to get a basic understanding of USB connections. Without this it is not possible to understand any example code, not to mention writing ones own code. For this reason a short introduction into the concept of USB is given hereafter.
A USB function is the function of the USB device that receives and sends data. Typically



Source: http://www.beyondlogic.org/usbnutshell/usb3.shtml#USBProtocols

Figure 4.2: General USB connection setup

it has a series of buffers which belong to an endpoint. In Figure 4.2 a functions can be seen that has several endpoints set up. How many endpoints are used is dependent on the the hardware and the needs of the function. In our case we initialize three endpoints, which will be further discussed in section 4.2.
Whenever the host sends a request the function hardware will read the packet to determine from the address field whether the packet is for itself. If the address matches the device address, the function will copy the payload to the appropriate endpoint buffer and generates an interrupt to signify that the endpoint has received a packet. This is usually done in hardware.
The software only needs to initialize the endpoints as well as implementing appropriate send and receive functions. The important concept in the USB system is that no functions may transmit as soon as they are ready to do so. They must wait to be queried, and then send the appropriate request via a reply or interrupt transfer.

4.2 Endpoints

In the section above endpoints were mentioned. To use them in our code we had to find out what kinds of endpoint exist and how to initialize them.
Endpoints can be described as sources or sinks of data. As the bus is host (PC) centric,

the endpoints occur at the end of the communication channel on the device (Olimex) side. Each endpoint is either an IN or an OUT endpoint. Whenever the device driver sends a packet to the device it will end up in the OUT buffer of this endpoint, because the data is flowing out from the host. There it can be read from the device. The other way round, if the device wants to send data it writes it to the endpoint IN buffer of its choice, where it stays until the host sends a IN packet corresponding to that endpoint. Note that all devices must support EP0, because all control and status requests are send to this endpoint. This is automatically done by the used library.
In general four types of endpoints can be declared.

- Control Transfers
- Interrupt Transfers
- Isochronous Transfers
- Bulk Transfers

For the use case of continuously sending sensor data from the Olimex to the computer a bulk transfer is needed.
Figure 4.3 shows the IN and OUT endpoint with their attributes. The flag `USB_ENDPOINT_ATTR_BULK`

```
static const struct usb_endpoint_descriptor data_endp[] = {{
    .bLength = USB_DT_ENDPOINT_SIZE,
    .bDescriptorType = USB_DT_ENDPOINT,
    .bEndpointAddress = 0x01,
    .bmAttributes = USB_ENDPOINT_ATTR_BULK,
    .wMaxPacketSize = 64,
    .bInterval = 1,
}, {
    .bLength = USB_DT_ENDPOINT_SIZE,
    .bDescriptorType = USB_DT_ENDPOINT,
    .bEndpointAddress = 0x82,
    .bmAttributes = USB_ENDPOINT_ATTR_BULK,
    .wMaxPacketSize = 64,
    .bInterval = 1,
}};
```

Figure 4.3: Bulk transfer endpoints

defines the type of the endpoint. Notice that the maximal packet size is set to 64, which

is the maximal size for high and full speed devices. An endpoint address consists of 8 Bits. Bits 0-3 are the endpoint number, Bits 4-6 are reserved and set to Zero and Bit 7 signifies the direction (0 = OUT, 1 = IN). The addresses used for the endpoints are

$$\text{EP1 OUT} = 0 \times 01_{16} = 00000001_2$$
$$\text{EP2 IN} = 0 \times 82_{16} = 10000010_2.$$

We have seen that it is possible to set the endpoints by using an endpoint descriptor and how the endpoint address is chosen. Both is needed to perform the basic setup of the USB connection.

4.3 Callback Functions

The next problem our group was facing, was accessing the IN and OUT buffers. To process incoming data in the OUT buffer and write data to the IN buffer that should be send, the corresponding endpoints have to be connected with so called callback functions. These functions are called when the device reads or writes data. Because we want to write the sensor data to the USB bus, but don't need to receive any data from the computer, we only implemented a function that reads the data from the sensor and writes it to the IN buffer using library functions (compare Figure 4.4). This function

```
static void cdcacm_data_rx_cb(usbd_device *usbd_dev, uint8_t ep)
{...
  usbd_ep_write_packet(usbd_dev, data_endp[1].bEndpointAddress,
    &data, sizeof(data));
...}
```

Figure 4.4: Callback function writing to IN buffer of EP2

is linked to the OUT endpoint EP1, meaning that if an IN packet is send from the computer it is written to the OUT buffer of the device and `cdcacm_data_rx_cb` is called, which writes the data from the sensor to the IN buffer, the bus respectively. To establish a continuous data transfer from the device to the computer, the computer has to send IN packets regularly. This functionality is provided by a poll function which is called with a given frequency in the device driver of the computer.

4.4 Setting up a USB Connection

We now have all necessary information about our device, so that we are able to set up a connection between the Olimex and a computer. To find out how this is done using

libopencm3 was one of the problems of our group. Every USB device must respond to setup packets on the default pipe. The setup packets are used for detection and configuration of the device and carry out common functions such as setting the USB device's address, requesting a device descriptor or checking the status of a endpoint. In order to enable our device to answer with the requested data of a setup packet several functions have to be implemented and registered. After the device is powered up and is initialized with usbd_init the function usbd_register_set_config_callback is called. It is a library function that accepts an USB device and a configuration function as input arguments. In Figure 4.5 our implementation of this configuration function is shown. The main task of it is to associate the endpoints with their callback functions with usbd_ep_setup. Note that only the OUT endpoint is linked with a function.

```
static void cdcacm_set_config(usbd_device *usbd_dev, uint16_t wValue)
{
      (void)wValue;
      (void)usbd_dev;
      usbd_ep_setup(usbd_dev, 0x01, USB_ENDPOINT_ATTR_BULK,
          64, cdcacm_data_rx_cb);
      usbd_ep_setup(usbd_dev, 0x82, USB_ENDPOINT_ATTR_BULK,
          64, NULL);
      usbd_ep_setup(usbd_dev, 0x83, USB_ENDPOINT_ATTR_INTERRUPT,
          16, NULL);

      usbd_register_control_callback(
              usbd_dev,
              USB_REQ_TYPE_CLASS | USB_REQ_TYPE_INTERFACE,
              USB_REQ_TYPE_TYPE | USB_REQ_TYPE_RECIPIENT,
              cdcacm_control_request);
}
```

Figure 4.5: Configuration Function

4.5 Poll Function

This section focuses on poll functions. In section 4.3 it was mentioned that the computer has to poll regularly to allow the device to send data. Theory says that because the USB bus is host centric, the device is not able to start a transfer without the permission of the

computer and therefore does not have to poll the bus. In practice this assumption does not hold. In the main function of the Olimex a loop is needed that calls `usbd_poll()` an infinite number of times. Otherwise the program does not work, more precisely no data is send. Online we found two statements that state the necessity of this function:

> "You need to call the function (...) at least once every millisecond or in response to USB interrupt, which is the preferred way. If do not use interrupts and forgot to call (the function) regularly then your code will hang if it is waiting for a data transfer to complete[2]."

> "Just when we have data to send, the data is put to the bulk IN EP. When no data is written to the IN EP, no more EP interrupt is generated. Then, we have to poll the number of data on the TX buffer periodically out side of the EP ISR, too. SOF interrupt is often used for this purpose[3]."

The reason for polling according to the first quote is, that otherwise the program will wait for the completion of a data transfer. The second quote is more precise and gives the argument that if more data is expected than is actually ready to send, we need to poll the remaining amount of data by hand, to prevent the program of waiting for all IN packets to be served. Therefore the poll function does not poll the bus but the IN buffer of EP2.

4.6 The Sensor Data

Finally, we need to access the data on the computer for debugging purposes. Our USB device is handled by the default cdcacm device driver of the linux kernel, which creates an ACM device file. In this file all received data is stored. Our first approach to read out the data was to use minicom [4] as we did for the UART connection as well. This does not work because minicom is a tool for serial communication. After that we wrote a python script that reads out the device file and prints its content to the terminal to check if data is send and if we used the right format. The result was that for most sensor data our overall setup and code was working fine, except fast movements.

To solve this problem we have to have a look at the sensor's output. The LSM9DS0 sensor is able to return the measurements from a gyroscope, an accelerometer, an magnetometer and a thermostat. All values are integers and have to be converted

[2] `http://www.sparetimelabs.com/usbcdcacm/usbcdcacm.php`
[3] `http://www.keil.com/forum/13504/usb-cdc-acm-function-class`
[4] `http://linux.die.net/man/1/minicom`

into decimal numbers after certain procedures, in order to get meaningful values. The conversion took place on the Olimex, which then send packets of floating points to the computer. For to high measurements these floating points cause a buffer overflow, which explains our observation for fast movements of the sensor. The problem was solved by sending the raw data to the computer and perform the conversion there.

Another misbehavior we observed was that our packets arrived in arbitrary order when sending the four measurements in different packets. A quick fix by sending all values in only one packet solved the problem.

4.7 Conclusion

In conclusion this chapter describes how to establish an USB connection from the device's point of view. To do so IN and OUT endpoints are needed as well as a configuration function that provides information used during the setup of the connection. Also the necessity of the poll function was discussed, which is needed to take care of IN packets from the computer that cannot be served with data. Last but not least we had a look on the sensor data, how to read the incoming packets on the computer and that the conversion cannot be done on the Olimex because of the great size of floating point numbers. To sum it up a overview of the code for the USB connection from the Olimex to a computer and the problems group 2 had to deal with was given.

5 Compiling Fiasco.OC for BananaPi and Genode for PandaBoard - Tamas Bakos

5.1 Introduction

For this part of the project, we have compiled a Fiasco.OC based Genode for our development board, to be able to run a small application on it, which transmits data from the sensor to a PC. In this chapter we describe firstly what Fiasco.OC is and how it should be compiled and ran on the BananaPi in order to get familiar with the OS and embedded systems, and secondly what Genode is, and how it should be compiled for the PandaBoard, along with building an application.

5.2 Fiasco.OC for BananaPi

This part is optional for the final result, however, it is recommended in order to get comfortable with Fiasco, and running your own OS on an embedded system.

5.2.1 What is Fiasco.OC?

Fiasco.OC is a third generation microkernel developed by the OS group of the TU-Dresden, which evolved from its predecessor L4/Fiasco. Fiasco.OC is capability based, supports multi-core systems. The completely redesigned user-land environment running on top of Fiasco.OC is called L4 Runtime Environment (L4Re). It provides the framework to build multi-component systems, including a client/server communication framework, common service functionality, a virtual file-system infrastructure and popular libraries such as a C library, libstdc++ and pthreads.[1]

5.2.2 What is the BananaPi?

Banana Pi is a single-board computer based on an Allwinner A20 ARM Cortex-A7 dual-core processor, and has 1GB of DDR3 SDRAM, a Gigabit ethernet port, and a SATA Socket.

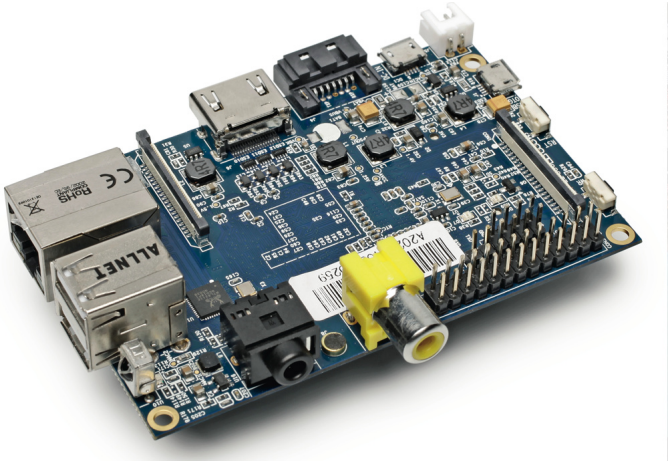


Figure 5.1: BananaPi Board

[1] https://en.wikipedia.org/wiki/L4_microkernel_family#Other_research_and_development

5.2.3 Compile Fiasco.OC for BananaPi[2]

Download necessary packages and source code

Install subversion and download source code for Fiasco.OC. Inside the main folder you will find the directories "kernel" which contains the l4 kernel, and "l4" which contains the L4 Runtime Environment.

```
sudo apt-get install subversion
svn co http://svn.tudos.org/repos/oc/tudos/trunk fiasco.oc
```

Get further necessary packages:

```
gcc-arm-linux-gnueabi \ libc6-dev libncurses5-dev
g++-arm-linux-gnueabi \ build-essential u-boot-tools
```

Build the kernel

Create directory, where the kernel will be built:

```
cd fiasco.oc/kernel/fiasco/
make BUILDDIR=build-cubieboard2
```

Go to new directory and start configuring the kernel:

```
cd build-cubieboard2
make menuconfig
```

Set the following options and click Save:

```
Target configuration
|-> Architecture (ARM processor family)
|-> Platform (Allwinner (sunxi))
|-> CPU (ARM Cortex-A7 CPU)

Kernel options
|-> [*] Enable multi processor support
```

[2] http://lautgedacht.org/posts/2015-02-21-cubieboard2-fiasco.html

Before starting the build, the file "src/Makeconf.arm" has to be adjusted:

```
# -*- makefile -*-
# vi:se ft=make:
#OPT_SHARED_FLAGS += $(call CHECKCC,-finline-limit=10000,)
SYSTEM_TARGET += arm-linux-gnueabi-
```

Execute the make command in the build directory, to build the kernel.

Build the Runtime Environment

Create directory, where the RE will be built:

```
cd fiasco.oc/l4
make B=build-cubieboard2
```

Adjust the file "mk/Makeconf":

```
SYSTEM_TARGET_arm = arm-linux-gnueabi-
```

Start configuring the RE:

```
make menuconfig O=build-cubieboard2/
```

Set the following options and click "Save":

```
Target Architecture (ARM architecture)
CPU variant (ARMv7A type CPU)
Platform Selection (CubieBoard 2)
```

Execute the make command in the build directory, to build the RE.

5.2.4 Configure target system

Create uImage

For the bootloader (U-Boot) to be able to execute it, we need Fiasco.OC in uImage format. In order to prepare it, execute the following commands in the "l4/build-cubieboard2" folder:

```
make E=<image_name> uimage \ # <image_name> has to be consistent everywhere!
MODULE_SEARCH_PATH=<abs path>/fiasco.oc/kernel/fiasco/build-cubieboard2/
```

Choose the starting configuration in the appearing menu. The built image will be at "bin/arm_armv7a/".

uImage onto the SD-card

Prepare a partition on the SD-card of the target system. For this we format it to ext2 using gparted or the following command:

```
sudo mkfs.ext2 /dev/<sd_card_name>
```

Copy the uImage onto the SD-card.

Configure bootloader (U-Boot)

Create the script file "boot.cmd", with the following content: The first command tells

```
setenv kernel_addr_r 0x40000000
ext2load mmc 0 ${kernel_addr_r} /<image_name>.uimage
bootm ${kernel_addr_r}
```

the bootloader to load the kernel image to the specified address. The "ext2load" command loads the kernel image from the first partition, and "bootm" initializes the boot process. The bootloader can be configured via the "boot.scr" binary, which sets up the booting parameters. This binary can be created from the "boot.cmd" with the following command:

```
mkimage -C none -A arm -T script -d boot.cmd boot.scr
```

Copy the boot.src onto the SD-card, and plug the SD-card into the embedded board. **Start your system!**

5.3 Compile Genode for PandaBoard

5.3.1 What is Genode?

The Genode OS Framework is a tool kit for building highly secure special-purpose operating systems. It scales from embedded systems with as little as 4 MB of memory to highly dynamic general-purpose workloads. [3]
Genode is open source and commercially supported by Genode Labs. In this project we are using Genode based on the Fiasco.OC.

5.3.2 What is a PandaBoard?

The PandaBoard is a low-power, low-cost single-board computer development platform based on the Texas Instruments OMAP4430 system on a chip. It features a dual-core 1 GHz ARM Cortex-A9 MPCore CPU, a 304 MHz PowerVR SGX540 GPU, IVA3 multimedia hardware accelerator with a programmable DSP, and 1 GiB of DDR2 SDRAM.

Figure 5.2: PandaBoard

[3] `http://genode.org/about/`

5.3.3 Compiling Genode

Get the latest release here: `http://genode.org/download/latest-release`, and unzip the downloaded archive.

Create a toolchain:

```
cd <genode-dir>
./tool/tool_chain arm
```

Prepare the libports, the DDE, and the base-foc repositories:

```
cd $(GENODE_DIR)/repos/libports
make prepare
cd $(GENODE_DIR)/repos/dde_linux
make prepare
cd $(GENODE_DIR)/repos/base-foc
make prepare
```

Create and prepare your build directory:

```
cd <genode-dir>
./tool/create_builddir foc_panda BUILD_DIR=~/genode-build.foc_panda
cd <build-dir>
nano etc/build.conf
```

Uncomment the following lines:

```
REPOSITORIES += $(GENODE_DIR)/repos/libports
REPOSITORIES += $(GENODE_DIR)/repos/dde_linux
```

Build with the following command:

```
make -C build.okl4 run/<build_name>
```

5.3.4 Compiling a program for Genode

Create directory "<program_name>/src/<module_name>". Write the source code to your Genode application, based on the tutorial at "`http://genode.org/documentation/developer-resources/client_server_tutorial`"

Create a file named "target.mk" in "<program_name>/src/<module_name>", with the following content:

```
TARGET = <_module_name>
SRC_CC = main.cc
LIBS = base
```

Add a start entry to the config file, like this:

```
<start name="<_module_name>">
<resource name="RAM" quantum="1M"/>
<provides><service name="Hello"/></provides>
</start>
```

Copy the "<program_name>" folder to "$(GENODE_DIR)/path/to/the/<program_name>"

Add the following line to "etc/build.conf":

```
REPOSITORIES += $(GENODE_DIR)/path/to/the/<program_name>
```

Download skeleton .img file from "`https://github.com/downloads/nfeske/genode/two_linux_panda.img.gz`" and unzip it.
Insert SD card and execute command:

```
sudo dd if=two_linux_panda.img of=/dev/<sd_card_name> bs=1M
```

Replace image.elf, modules.list and genode/ with your own build.

5.4 Problem encountered

We only encountered one problem at this part of the project: when compiling the Fiasco.OC for the BananaPi, we just did not use the correct filename for the uImage in the "boot.cmd", therefore the bootloader (U-Boot) was not able to find the kernel image, and thus could not boot.

5.5 Conclusion

In this chapter it is explained step-by-step how to compile Fiasco.OC, Genode and a program for Genode. Anyone wanting the same result has only to copy the instructions from this documentation, and adjust the template names marked by <>. As for this part it was only necessary to follow exact instructions, these are all that had to be described. Because of the straightforwardness of this part, we only encountered one problem during the Genode compilation, and we solved it after the first laboratory session. Due to the fact that this part of the project was skipped in later stages, unfortunately this chapter did not contribute to the final result.

6 2-Uart-Component PandaBoard - Johannes Ismair

In this chapter we explain how to set up a PandaBoard with the intention to receive data from our sensor and to send the data to another computer. Both connections will be established via UART. Figure 6.1 shows the setup described in this chapter.

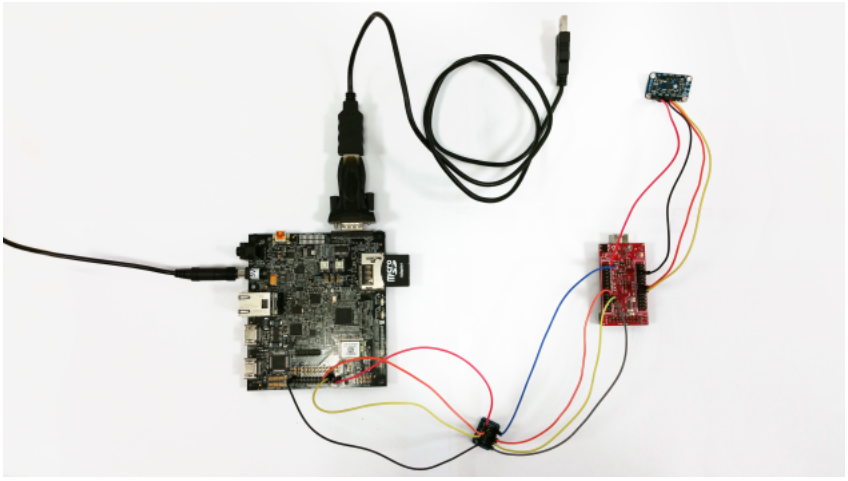


Figure 6.1: The basis setup for the UART application

6.1 UART Interfaces on the Pandaboard

The documentation of the PandaBoard[1] lists a total of four UART interfaces. The Olimex gets connected to the PandaBoard through the level shifter by using the UART4 interface on the J3 expansion (TX = Pin 6, RX = Pin 8). For the second UART connection we used the standard UART3 RS232 interface, which is primarily used for debug outputs.

[1] http://pandaboard.org/sites/default/files/board_reference/pandaboard-a/panda-a-manual.pdf

6.2 The Configuration File

The Genode architecture is based on a strictly hierarchic and recursive structure. At boot time, the init process gets started by the core and gets assigned all the physical resources and controls for the execution of all further processes. Its policy is driven by a XML file named config. It declares a number of children, their relationships, and resource assignments.

```
<config>
    ...
            <start name="uart_drv">
                    <resource name="RAM" quantum="1M"/>
                    <provides>
                            <service name="Uart"/>
                            <service name="Terminal"/>
                    </provides>
                    <config>
                            <policy label="panda_uart2_server" uart="3"
                             baudrate="115200" />
                    </config>
            </start>
            <start name="panda_uart2_server">
                    <resource name="RAM" quantum="1M"/>
            </start>
            ...
    </config>
```

Figure 6.2: The main part of the configuration file

In the configuration file of our application (see Figure 6.2) you can see that the UART driver is loaded with the UART4 interface (enumartion starts with index 0) and a baudrate of 115200. The config also defines that the process named panda_uart2_server will use it.

6.3 The Application

The starting point of the UART application is the basic example that you can find in the genode folder `genode/repos/os/src/test/uart/`.
The UART connection can be accessed in the program by creating an object of

`Uart::Connection`. Genode automatically assigns the proper interface which is declared in the configuration file for this process. The object offers basic read and write operations which then are used to read the data in a buffer. It is important to check if any data is received because the read function is non-blocking.
The received data is sent to the UART3 interface by just calling printf. This works as aforementioned because it is the standard debug interface.

```
static Uart::Connection uart;
static char read_buffer[READ_BUFFER_SIZE];

for (;;) {
        int num_bytes = uart.read(read_buffer, sizeof(read_buffer));
        read_buffer[num_bytes] = '\0';
        if (num_bytes != 0)
                printf("%s", read_buffer);
        /* ... */
}
```

Figure 6.3: The main part of the application

6.4 Problems

We encountered some problems during our work on the application:

1. The application was not working in the beginning. Another group proposed to set the baudrate manually. So we added it to the configuration file and the warning of the driver disappeared (which complains when the default baudrate is used). Apart from that it had no impact on the program.

2. The application was only working on the last revision of the PandaBoard we got to use. It remains to be analyzed where the problem with the other revision was.

3. As we got the first output of our program, we just got the 'd' of "Hello World". We forgot to reset the output on the terminal by adding a \r after each \n we used.

6.5 Conclusion

This chapter described how to setup a basic application in Genode for the PandaBoard which reads data via UART and sends it by using a second UART connection. To get it working, it is important to connect the sensor with the right pins and to use a level shifter as well as set it up correctly in the configuration file. Last but not least some problems were discussed which we encountered during our work on the application.

7 Linux USB Device Driver - Gurusiddesha Chandrasekhara

In this chapter the development of USB device driver is described. The Linux USB device driver is used to communicate between Linux host and LSM9DS0 sensor through Olimex. This driver was implemented and tested on Linux kernel 3.13. The driver follows the standards of Linux USB driver specification. Therefore, it can be compiled and tested on recent versions of kernel as well. The functionality is implemented in the C header `usb_read.h` and source file `usb_read.h`. The source files can be found under the following path in the git repository of group 2: `praktikum/src/usb_driver/usb_driver_final`

7.1 Building and Loading

The driver is implemented as a single, loadable Linux Kernel Module(LKM).

7.1.1 Building

To build the driver simply enter the source folder and issue the command `make`. This will build the driver, a file named `accel.ko` should be generated.

7.1.2 Loading

The driver can be loaded by issuing a command `insmod accel.ko` as `root`. One of the problems that can occur while loading a custom driver is that, there might be some default driver that is already taking care of this device. In our case the `cdc_acm` driver was handling `LSM9DS0` device. Our challenge was to make sure that `accel.ko` takes care of LSM9DS0.

The temporary solution was to unload the `cdc_acm` driver. `cdc_acm` gets loaded when the device is connected. Unload the cdc_acm using `rmmod cdc_acm` and load the accel.ko using `insmod accel.ko`. But this was a repeated task to everytime unload the driver. Fortunately there is an option in Linux to blacklist a module so that it will not

get loaded. It can be done by adding the following line to `/etc/modprobe.d/blacklist` file.

```
blacklist cdc_acm
```

7.2 USB Device Basics

In order to write a USB device driver understanding the USB subsystem was important. We went through chapter 13 of the Linux device driver book [1], which explains the USB driver in detail.

The USB communication happens through `endpoints`. USB endpoints are bundled up into `interfaces`. USB interfaces handle only one type of logical connection, such as keyboard or a mouse. USB interfaces are again bundled up into something known as `configurations` and a device might have multiple configurations to switch between them. USB drivers bind to USB interfaces not the entire USB device which is illustrated in Figure 7.1.

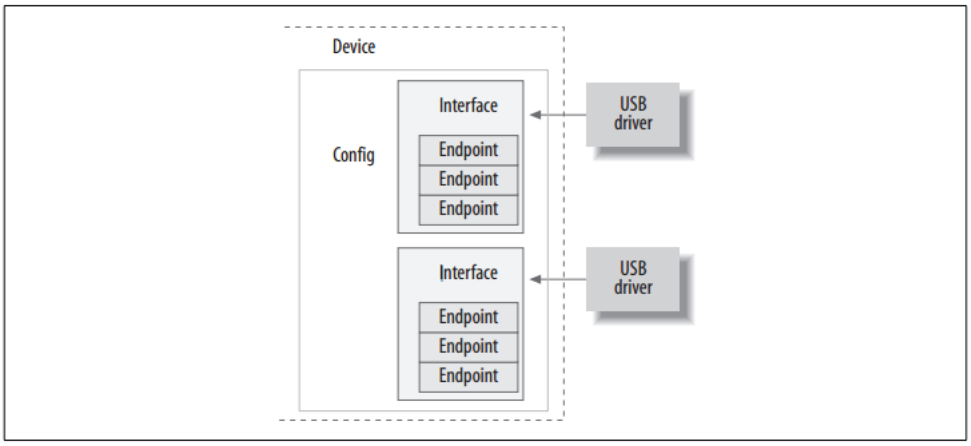


Figure 7.1: USB device overview

7.3 Implementation

This section discusses the implementation details of the driver along with the challenges encountered and the resolutions for them as well. We took the USB skeleton

[1] `http://lwn.net/images/pdf/LDD3/ch13.pdf`

driver provided by the Linux kernel source [2] as a reference to design our driver for LSM9DS0. All the declarations(except static ones) are declared in header file and all other definitions and driver specific data is in source file. A structure `usb_skel` was defined to hold all the device specific data.

7.3.1 Registering the USB Driver

The first thing Linux USB driver needs to do is register itself to USB subsystem. The driver passes a structure `usb_driver` with the information about which devices the driver supports and the functions to call when the device is inserted or removed. The structure is defined as below,

```
    static struct usb_driver accel_driver = {
     .name =        "accel_driver",
     .probe =       accel_prtobe,
     .disconnect =  accel_disconnect,
     .id_table =  accel_table,
};
```

The registration and deregistration of the driver is done in `module_init` and `module_exit` function respectively as shown below.

```
static int __init usb_module_init(void)
{
int Major = usb_register(&accel_driver);
if (Major < 0)
  printk(KERN_ALERT "Registering Accel sensor device failed with %d\n", Major);

return Major;
}

static void __exit usb_module_cleanup(void)
{

usb_deregister(&accel_driver);
}

module_init(usb_module_init);
module_exit(usb_module_cleanup);
```

[2] `http://lxr.free-electrons.com/source/drivers/usb/usb-skeleton.c`

7.3.2 Vendor and Product id

The `struct usb_device_id` structure provides a list of USB devices that this driver supports. This struct is used by USB core to decide which driver supports which device. Vendor and product id are defined in `accel_table` as shown below.

```
#define USB_ACCEL_VENDOR_ID 0x0042
#define USB_ACCEL_PRODUCT_ID 0x0007

static struct usb_device_id accel_table [] = {
    { USB_DEVICE(USB_ACCEL_VENDOR_ID,
      USB_ACCEL_PRODUCT_ID) },
    { }                     /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, accel_table);
```

7.3.3 Probe and Disconnect calls

One of the major problems faced by our team while writing the driver is to get the probe call working and setting up the correct endpoint for receiving the data. As one can select `BULK_IN` or `Interrupt` endpoint for this type of application. We decided to use `BULK_IN` endpoint.

`probe`: `probe` function is called when a device is connected to the system. It performs the checks on the information passed to it about the device and finally register the device. We setup the `BULK_IN` endpoint to receive the data and allocate space for `bulk_in_urb` which are later used in `device_read` function. After setting up these data we finally register our device using `usb_register_dev` as shown below.

```
int retval = usb_register_dev(interface, &skel_class);
```

`USB Urbs`: The USB code communicates with all USB devices using urb(USB request block). A urb is used to send or receive the data to or from a specific endpoint in an asynchronous manner. Multiple urbs can be created to a single enpoint. However, we needed only one urb which is created in `probe` and assigned to BULK_IN endpoint to recieve the data. Using of urbs is explained in section 7.3.5.

`disconnect`: `disconnect` function is called when the device is disconnected from the system and can do some clean-up. We give back the minor number and call an helper function `skel_delete` is called to deallocate all the memory.

7.3.4 File Operations

File operations are used to communicate between user space program and the driver. `struct file_operations` holds pointers to functions which performs various operations on the device.

```
static struct file_operations fops = {
.read = device_read,
.write = device_write,
.open = device_open,
.release = device_release
};
```

- `device_open`: Called when some application tries to open the device.
- `device_close`: Called when the file structure is being released.
- `device_write`: Used to send the data to the device. This functionality is currently not implemented by our driver since our goal was to only read the data from the device.
- `device_read`: Used to retrieve data from device.

This is a class driver, `struct usb_class_driver` is populated with device file name `magic0` and device file operations. This structure is used in probe function to register as explained above.

```
static struct usb_class_driver skel_class = {
.name = "magic%d",
.fops = &fops,
.minor_base = USB_SKEL_MINOR_BASE,
};
```

7.3.5 Read Function in Detail

Another challenge we had while writing the driver is to get the read function correct inorder for us to retrieve the data. Our initial plan was to read the data through BULK_IN endpoint without using urbs in order to avoid complexity of creating and maintaining Urbs. The `usb_bulk_msg` function creates a USB bulk urb and sends data to specified device. This method was not successful for us since it was not able to receive any data from the BULK_IN endpoint and returning with error number -110. Therefor, we decided to use Urbs.

The read function call is organized this way. If there is no ongoing read, a helper function `skel_do_read_io` is called. In this helper function urb read is done via `usb_fill_bulk_urb` and urb is submitted by `usb_submit_urb` function call. Once the data is available, it is copied to user space buffer using copy_to_user function call.

7.4 Conclusion

As explained in this chapter the accel driver is a decent driver to read the data from the device file. There are, however, further improvements that can be done to increase the effectiveness. The driver creates two device files when connected, this can be eliminated. A write function can be implemented to send the data to LSM9DS0 sensor.

8 PC Application - Johannes Ismair

In this chapter we explain how we set up our PC web application in order to visualize the data from the sensor. The application uses the standard model in 3d computer graphics - the teapot - and rotates it corresponding to the data of the accelerometer. Additionally, the teapot changes its color from blue when the sensor measures a low temperature to red when the temperature is higher. Figure 8.1 shows the running application.

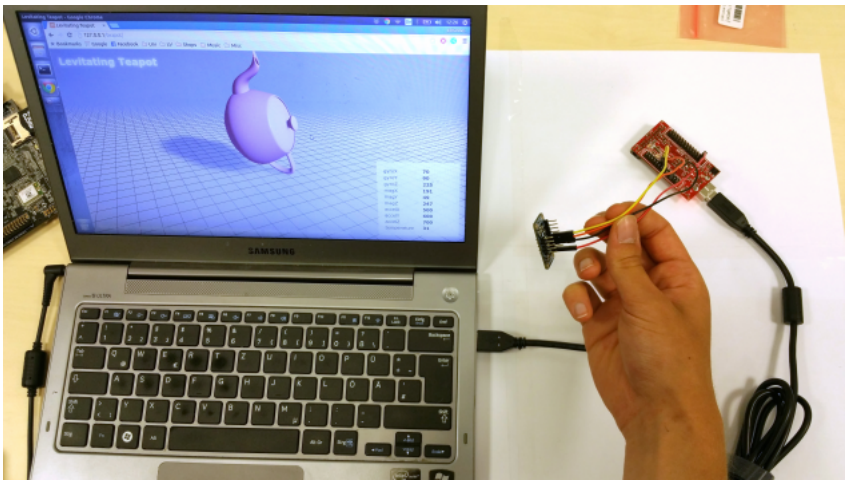


Figure 8.1: The teapot application

8.1 The Data Pipeline

As soon as the USB driver is installed and loaded and the Olimex with the sensor is connected to the computer, a device file called `/dev/magic0` will appear.
A simple C program reads the device file using standard file operations and casts the received data to a struct which contains ten integer values. The data is then converted (next section) and printed out in JSON format.
The backend uses an Apache webserver running a PHP installation, the frontend of our application uses ThreeJS which is a JavaScript wrapper library for WebGL. Every 100ms

Figure 8.2: Data Pipeline

the frontend sends update requests to a PHP script on the server which then executes the C program with a simple `shell_exec()` and sends the data back. It is important that the permissions on the device file are configured correctly. It may be necessary to change them with a chmod command or by editing the file `/etc/udev/permissions.rules`.

8.2 Convertion of the Raw Data

You can find explanations on how to interprete the sensor data in the official documentation[1] of the sensor or in the source code of the Arduino port of our sensor[2].
Each sensor value (except the temperature) has to be multiplied with a constant corresponding to their calibration (look in the documentation of the firmware) and be divided by 1000.
To calculate the temperature, the integer value has to be divided by 8 and added to an offset. The documentation did not contain the exact value and the Arduino port guessed a value of 21°C. Based on our experience, this value was way too high. We would recommend 15°C, but we had no thermometer to validate this value.
The web application shows the orientation of the sensor by rotating the teapot correspondingly. The magnetometer data was too unprecise to achieve sufficiently stable results; also the measurements were sporadically disturbed by magnetic fields of computers and other electronic devices nearby. Because of that the application uses the accelerometer data.

[1] `http://www.adafruit.com/datasheets/LSM9DS0.pdf`
[2] `https://github.com/adafruit/Adafruit_LSM9DS0_Library`

The following formular is used to calculate the roll and pitch values, where X, Y and Z are the values of the accelerometer[3]. As pointed out by the referenced document, we used the atan2 function to get correct values even when exceeding 45°:

Figure 8.3: Roll, pitch and yaw

$$\phi = atan2(X, Z) \cdot \frac{180°}{\pi}, \theta = atan2(-X, \sqrt{Y^2 + Z^2}) \cdot \frac{180°}{\pi}$$

8.3 Conclusion

This chapter described the steps to run our application for the LSM9DS0 sensor which makes it possible to rotate a teapot in a 3d application in the same way the sensor is rotated. Additionally we explained how to calculate the orientation in degrees by using the accelerometer data.

[3] `http://www.freescale.com/files/sensors/doc/app_note/AN3461.pdf`, p. 10, Eqn. 25/26

List of Figures

List of Tables