CSCI 502 - I2C COMMUNICATION WITH RASPBERRY PI, RASPBERRY PI EMULATION WITH QEMU, CMAKE PRACTICE

Madina Yergibay, Kanagat Kantoreyeva

***Abstract*—IMU is an Inertial Measurement Unit that consists of three types of sensors, in our case, an accelerometer, gyroscope, and magnetometer sensors. The project aims to introduce students to IMU interfacing with the I2C synchronous communication bus and to create multithreaded processes in the Raspberry Pi. After that, students learn emulating Raspberry Pi using QEMU and also, using Cmake.**

I. TASK 1. IMU INTERFACING (40% OF THE TOTAL GRADE)

For this project, an L3GD20H 3-axis gyro and an LSM303D 3-axis accelerometer and 3-axis magnetometer are used to undesrtand the I²C connection for Raspberry Pi (RPi).

A. I²C Communication

1) Task 1: For this task, we studied principles of general purpose input/output (GPIO) interfacing to Raspberry PI in Chapter 6 of the Exploring Raspberry PI textbook (Fig. 1).

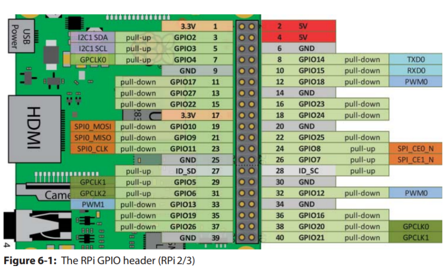

Figure 1. The GPIO of the Raspberry Pi 3b+ taken from the textbook Exploring Raspberry PI in Chapter 6.

IMU pin	RPi output/input
SCL	pin 3
SDA	pin 5
GND	pin 6
VDD	pin 2
VIN	disconnected

Table I
RULES TO CONNECT IMU TO RPI.

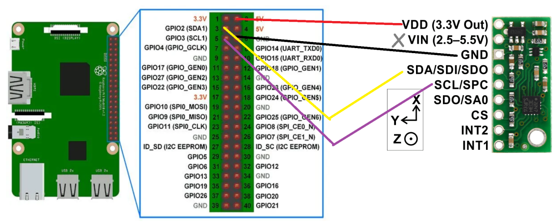

Figure 2. The GPIO of the Raspberry Pi connected to IMU (task 1.2)

2) Task 2 (10 points) : The schematics connection of the RPi GPIO to the IMU can be seen on Fig 2 along with the rules for connection in Table I.

3) Task 3 (10 points) : Steps and functions used to detect/check the sensor connected to the I2C bus of the board are:

```
$ sudo apt install i2c-tools
$ i2cdetect -l
$ i2cdetect -y -r 1
```

The book suggests to use i2cdetect, i2cdump, i2cget commands on PRi to check sensor connected to RPi via I²C busses. Using sensor datasheets we can access the data through register addresses with **i2cget -y 1 0x6B 0x28** and **i2cget -y 1 0x6B 0x29** commands, where it **gets** the value of OUT_X_L (28h), OUT_X_H (29h) or X-axis angular rate data from the device 0x6b (gyroscope) that is connected to the port number **1** out of two possible I2C ports on the RPi. (Answer to: why the value 1 is used in the above commands?)

4) Task 4 (20 points) : To change the settings of the accelerometer, magnetometer and gyroscope for special settings given in the table, we need to use i2cset command. It is done the same way as we access the data: with register addresses, but we use the **i2cset** command instead. Example, **i2cset -y 1 0x6b 0x20 0x0F**: i2cset - set command, 1-first port, 0x6b - gyroscope device, 0x20 - CTRL1 register address, 0x0F - control word meaning 00001111 value for settings.

Therefore, for the CTRL 1 (Fig. 5) to set the 50Hz frequency: Output data rate selection (with the use of table on Fig. 4 from datasheet) DR1 - 1, DR0 - 0, Bandwidth selection BW1 - 0, BW0 - 0, Power Normal Mode PD - 1, Z axis enable Zen - 1, X axis enable Xen - 1, and Y axis enable Yen - 1. Therefore, 1000 1111 is converted to 0x8F. In the same way control words for Table II are obtained.

7.2 CTRL1 (20h)

Table 19. CTRL1 register(1)

DR1	DR0	BW1	BW0	PD	Zen	Xen	Yen

1. Xen, Yen, Zen enable X, Yor Z register in level sensitive trigger mode. Once LVLen bit = 1, DEN level replaces the LSB of X, Y or Z axes and all axis are available for reading.

Figure 3. Table CTRL1 of L3GD20H

Table 21. DR and BW configuration setting

Low_ODR(1)	DR <1:0>	BW <1:0>	ODR [Hz]	Cut-Off [Hz](2)
1	00	00	12.5	n.a.
1	00	01	12.5	n.a.
1	00	10	12.5	n.a.
1	00	11	12.5	n.a.
1	01	00	25	n.a.
1	01	01	25	n.a.
1	01	10	25	n.a.
1	01	11	25	n.a.
1	1X	00	50	16.6

Figure 4. Table 21 for control word in CTRL1 of L3GD20H

Sensor name	Device	Register	Control word
Gyroscope	0x6b	0x20	0x8f (50Hz)
Gyroscope	0x6b	0x23	0x00 (±245 deg/sec)
Accelerometer	0x1d	0x20	0x57 (50 Hz)
Accelerometer	0x1d	0x21	0x00 (±2g)
Magnetometer	0x1d	0x24	0x64 (6.25 Hz)
Magnetometer	0x1d	0x25	0x20 (±4 gauss)
Magnetometer	0x1d	0x26	0x00

Table II
CONTROL WORDS DEFINED FOR SUBTASK 4.

B. TASK 2. EMULATING RASPBERRY PI USING QEMU (20 POINTS)

In this task, we will use QEMU to emulate ARM architecture of Raspberry Pi 3B+ version.

1) Task 1: The Chapters 1 and 2 of the "Embedded Programming with Modern C++ Cookbook textbook" were studied. The chapters describe different types of embedded systems and introduce us to emulators and cross-compilation tools. Embedded systems are specialized devices tightly coupling hardware and software to perform specific tasks in diverse conditions. Unlike desktops, they lack dedicated data centers and must operate independently. Hardware is precisely calculated for cost efficiency, with software maximizing resource use. Embedded systems communicate with peripherals, often lacking user interfaces, making development challenging. Emulators like QEMU facilitate development without physical boards. This recipe establishes virtualized environments for an Ubuntu build system and a QEMU-emulated ARM-based host system. It guides setting up cross-compilation tools and building a "Hello, world!" application for ARM, essential for embedded development.

2) Task 2 (10 points): We have followed the given instructions and installed QEMU with the Raspberry Pi (Raspbian Linux) OS with GUI on our virtual Ubuntu OS. The screenshot of the deployed Raspbian OS is shown below:

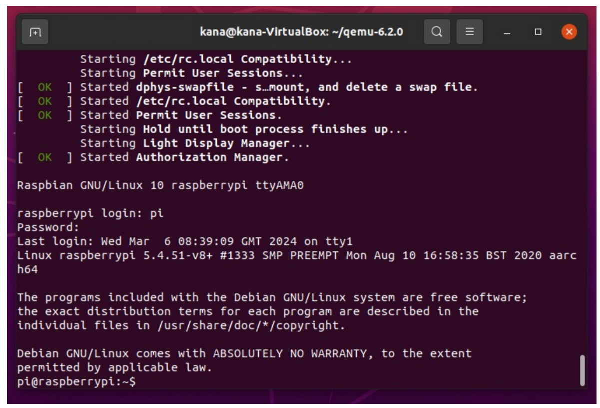

Figure 5. QEMU with the Raspberry Pi OS

3) Task 3 (10 points): Following the Cross-compilation section of the book, we have created a simple "Hello, World!" program. We have built hello.cpp using the cross-compiler (Fig. 6).

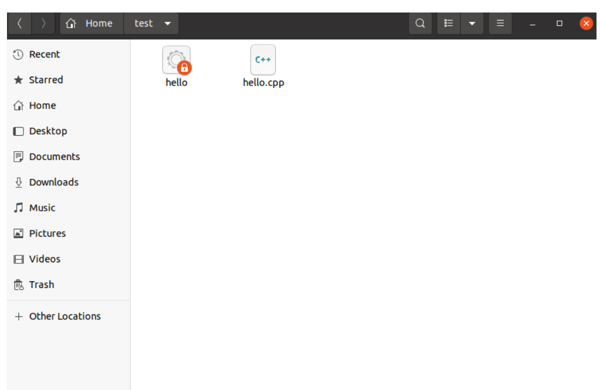

Figure 6. Hello.cpp

And then, we established ssh connection and transferred the generated executable file to the QEMU Raspbian OS through the Secure Shell (SSH) protocol (Fig 7).

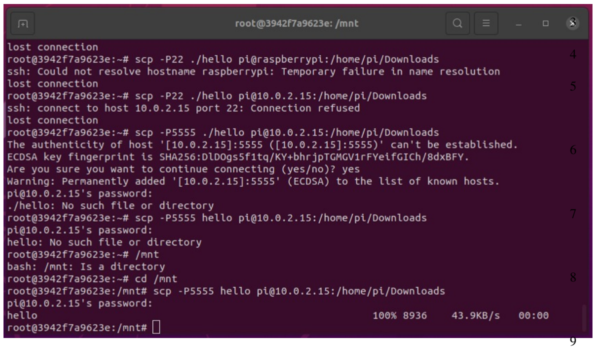

Figure 7. SSH connection and copying the file

We executed the file in QEMU Raspbian OS as seen on the Fig. 8 below.

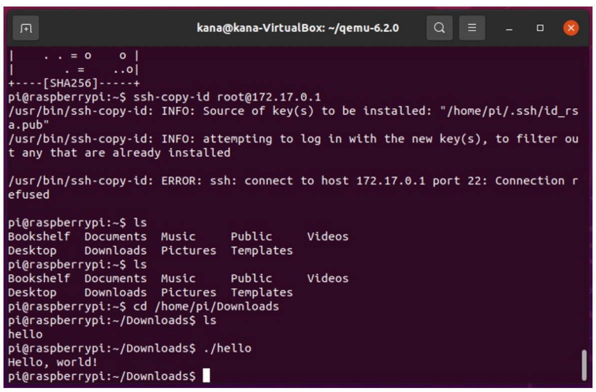

Figure 8. Executing on QEMU Raspbian OS

C. Task 3. CMake practice (30% of the total grade)

1) Task 1 (10 points): The given CMake Introduction tutorials were studied and steps were implemented in the Ubuntu 20.04 docker image virtual machine and then checked in the QEMU RPi emulator. We understood that CMakeLists.txt are needed to create the Makefile automatically.

We created CMakeLists.txt for the simple "hello" project given in the book and then run commands "cmake" and "make". Therefore, we have created "hello" build target executable from a code "hello.cpp" as seen from Fig. 9. Then we sent the executable "hello" to the RPi emulator via SSH connection (Fig. 10) and executed inside of the RPi emulator (Fig. 11).

CMakeLists.txt created for the following project "hello" is as follows:

```
cmake_minimum_required(VERSION 3.5.1)
project(hello)
add_executable(hello hello.cpp)

set(CMAKE_C_COMPILER /usr/bin/arm-linux-gnueabi-gcc)
set(CMAKE_CXX_COMPILER /usr/bin/arm-linux-gnueabi-g++)
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
```

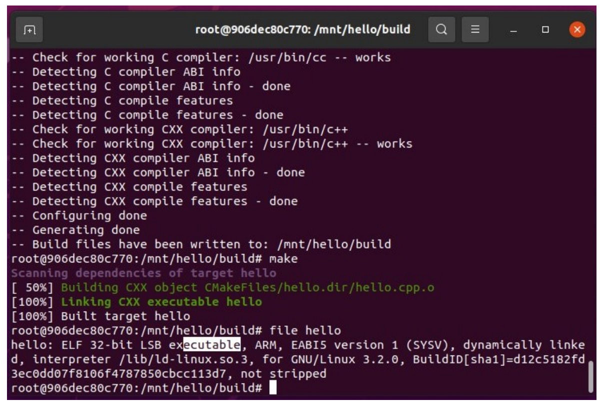

Figure 9. Executable "hello" created from cmake and make.

Figure 10. Executable "hello" sent to RPi.

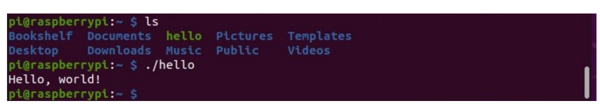

Figure 11. Hello world is executed inside of RPi. Cross compilation successful.

2) Task 2 (20 points): From the provided link for the open-source projects we chose the "Sudoku" game project (Link). It does not have CMakeFiles.txt, so we are going to create it. Firstly, we have downloaded the github repo by git cloning it to the new folder "newproject". Then, we created a build directory and wrote the following CMakeLists.txt:

```
cmake_minimum_required(VERSION
 ↪ 3.5.1)
project(sudoku)
add_executable(sudoku main.cpp
 ↪ altproj.cpp game.cpp solver.cpp
 ↪ tests.cpp altproj.hpp game.hpp
 ↪ solver.hpp tests.hpp)
```

Therefore, the game "Sudoku" has been compiled with cmake and make commands and the execution can be seen from Fig. 12 and video demonstration.

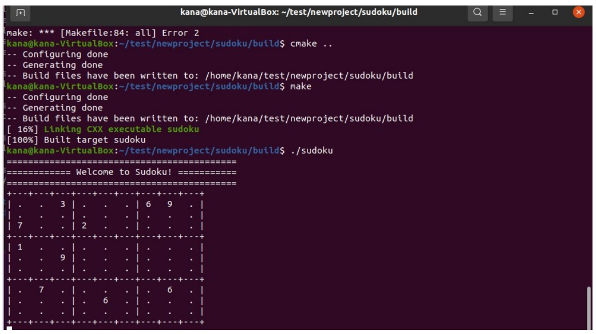

Figure 12. Sudoku game has been compiled with the implemented CMakeFiles.txt

II. Conclusion

At the end of the project, we learned the ways to work with IMUs, reading datasheets of the sensors and finding relevant information such as addresses of registers and constructing control words. Along with that, we have worked with the QEMU emulation of the Raspbian OS and the cross-compilation from the base architecture for other architectures as RPi. Finally, we have applied our knowledge of RPi emulation and cross-compilation for the projects in Task 3. We understood the CMakeFiles.txt creation and how it simplifies the work with Makefiles.