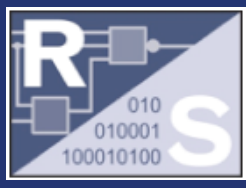
Computer Systems 2 SoC Lab

Report submitted by Julian Käuser and Fallou Galass Coly
Supervisor: M.Sc. Johanna Rohde
Begin: Nov. 13th, 2017 | Submission: Jan. 15th, 2018
Institute of Computer Engineering | Computer Systems Group | Prof. Dr.-Ing. Christian Hochberger

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Contents

1 Introduction

As a part of the *Computer Systems II* lecture, this lab deals with the application and driver development for the System-on-Chip (SoC) kit *SpartanMC*, which allows the design of an SoC on a FPGA target. In particular, the task is to generate a system configuration and develop firmware which allow distance measurements with an ultrasound sensor, a processing of the measured data, and the view of these results on a connected OLED display. This document reports about the implementation of the described tasks, and offers insights into the development processes.

The report is structured as following. First, the functions to be implemented are summarized. Afterwards, the toolchain use is reviewed. Then, the generated SoC structure is presented, and the implementations for the utilized subsystems are discussed. Before the report is closed with an evaluation of the work, a short view on the additional tasks is taken.

Task Summary

As a result of this lab, a SoC system which

- can measure a distance with an ultrasound sensor,
- processes the measured data to extinct wrong values and
- can display the data on an connected OLED Display.

shall be developed. Among the required steps are:

- Configuring the SoC hardware, including the peripherals for sensor and display connections,
- development of drivers for the SPI and I2C Master,
- implementation of the protocols of the ultrasound sensor and the OLED display driver,
- design and implementation of a filtering algorithm for the sensor data and
- integration of these sub-components into a running system.

1.1 SpartanMC Toolchain

Developing applications with a SpartanMC system requires the use of the SpartanMC toolchain. This is a set of applications which must be executed in order to define the system architecture, develop the firmware and execute it on the hardware. In general, all tools are triggered by a *make* target in a central makefile; that way, all dependency constraints are met. The main tools for the developer are:

- **JConfig**
  JConfig is a graphical tool to create a new system configuration and add components to the system.

In this task, the used components are a SpartanMC soft core, a clock generator, and peripherals conencted to an Advanced Peripheral Bus (APB). More details on the system architecture are dscribed in section 1.2. After the creation of a system configuration, it must be saved.

- **System Builder**
  The SpartanMC System Builder is invoked with the make target *make all program*, and reads in the previously designed configuration. It builds the HDL description of the system, triggers the synthesis of the hardware description, and flashes the resulting FPGA configuration onto the target board. As the synthesis and implementation have to be performed for each system individually, this step requires some time in the range of seconds to minutes. As long as the hardware is not changed (i.e. no change in JConfig), the System Builder does not need to be run again.

- **Programming the FPGA**
  With *make program*, the synthesized hardware design is flashed to the FPGA, as it is also described in the previous section. Even if the system configuration does not change, it may be necessary to re-program the FPGA, for example if the FPGA board was shut down; the configuration is lost in that case. Although *make program* also writes the firmware to the memory, this step is further described in the next section.

- **Programming the Processor**
  So far, only hardware is generated. In order to program the processor with the written firmware, the code must be compiled, linked and assembled, and be loaded into the memory of the system. Therefore, the compiler toolchain must be invoked. It compiles the user code placed in `firmware/src` and `firmware/include`, and links it with the general libraries wihtin the SpartanMC environment. With *make updateRam program*, the generated executable is transmitted to the memory of the system. The CPU core is resetted, and the execution of the user code starts. The target *updateRam* prohibits the hardware re-configuration and avoids the time-consuming reprogramming process for the FPGA target.

- **Other Tools**
  Other than the steps described previously, a serial terminal on the host pc and a logic analyzer are used. With the serial terminal, status messages and data from the program can be displayed; additionally, parameters can be sent to the SoC. The logic analyzer is helpful to debug electrical problems and can be attached to pins on the target or the connected peripherals.

As one can see, the central design steps are the configuration of the system architecture with *JConfig* and writing the user code as *C* code.

1.2 System Architecture

The SoC configuration consists of two major parts: the main system with the cpu core and the peripherals.

For this task, the main system may consist only of one SpartanMC soft core, a memory block, and a clock generator. This is the minimum configuration, and offers a low complexity.

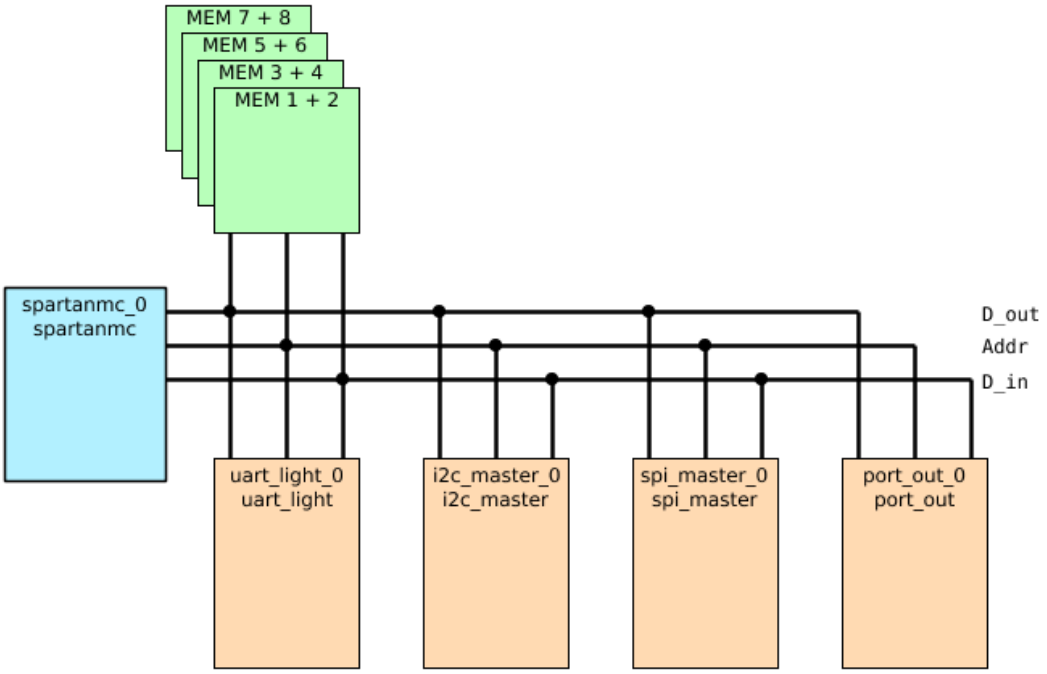


Figure 1.1: System Architecture for the SoC.

In order to implement the functions described in the task summary, the SoC requires four peripherals: a UART Light Peripheral for serial communication to the host PC, a SPI master to send data and commands to the display, an I^2C master to read out the sensor, and a General-Purpose I/O Port, requried to reset the display. These are included in the system configuration, as shown in Fig. 1.2.

1.3 General Code Structure

As a result to this task, several files of C sourcecode have been created. These can be split into application code and driver functionality.

The application code consists of the main function (in `main.c`), which calls the initializing functions for all modules, and the filter (in `filter.c/filter.h`). In the main loop, which is repeated forever, the functions to read the sensor value, filter it and send it to the display and the host processor are called.

The driver functionality created in this lab is separated into ultrasound sensor reading procedures (`ultrasound.c`), SPI sending and init procedures (`spi.c`), and GPIO and I^2C procedures (`gpio.c`, `i2c.c`). The display functions are implemented in `oled25664.c`, while a header for this given file had to be added manually.

As presented above, the code structure follows the typical approach for microcontrollers: first, all peripherals and modules are initialized; in the main loop, abstract driver functions are used, so that the concerns for each task are separated.

2 Inter-Integrated Circuit Bus

some stuff on the i2c interface here

3 Range Sensor

The SR02 is a distance sensor based on ultrasound, which allows the measurement of distances between 15 centimeters and approximately six meters. It has an I^2C interface, with which it can be started and read out. In this chapter, the communication with the sensor via this interface, as well as the processing of the measured data, is described.

3.1 Communication via I^2C

here some stuff on i2c

3.2 Measurement Processing

Every measurement performed with sensors is subject to fluctuations, and may additionally oly be taken as valid if the values are within the sensors measuring range. Hence, it is common to apply a processing procedure to correct erroneous values. Our filtering procedure is based on two steps, as presented in the following.

3.2.1 Sensor Range Check

The SRF02 sensor features a minimum measurable distance of 15 centimeters, while the maximum distance is about six meters. Consequently, any value below the lower threshold or higher than the upper range limit is may not be interpreted as valid. Before any further processing, these conditions are checked; if the currently processed value exceeds the range, it is ignored. for further processing, the previously stored value from the last measurement is used.

3.2.2 Median Filter

Before a filter algorithm for measurements can be designed, the unfiltered data must be examined for errors. Then, an algorithm capable of reducing these errors can be developed.

In our case, we observed two noteable characteristics:

1. **Rogue Results**
   In some cases, especially if the distance is near the lower bound of the sensors range, measured values are wrong because they are much too high. For example, when measuring distances between 15 and 20 centimeters, some returned values from the sensor are in the range of 200 plus centimeters. These have to be filtered out only when the previously measured distances clearly indicate that the value around 200 cm cannot be valid.

2. **No Distinction at High Error Rate**
   If many measurements are wrong (e.g. below the threshold), it is not possible to determine whether the measured value is right or wrong any more. Hence, the processing can never filter out all errors.

Thus, a filter which stores a number of previous values and determines the validity of the current value based on these must be implemented, alothough it can never be perfect with this behaviour. A *median filter* is suited well because it is capable of sorting out too high or too low, rogue measurements. The filter is based on an array, which stores the last N unfiltered values, where N is a parameter which must be found fo roptimal filter behaviour.

For the median filter, three steps are necessary:

- First, the oldest value in the array has to be deleted. In our implementation, the oldest value is overwritten by the current measured value, and the buffer for the last N values is a ring buffer. Consequently, the pointer to the oldest value is incremented, and, if it arrives at the end, reset to the initial value.

- Second, the buffer with the last N values must be sorted. We chose the *Simple Sort* algorithm, which is of $\mathcal{O}(n^2)$ complexity, iterative and works in-place. SpartanMC does not allow too much recursion, so an iterative approach is required. The complexity is acceptable for not too large buffers (see section 3.2.2 for details on the choice). As Simple Sort changes the array it is given as input, we decided to copy the buffer array and sort the copy; otherwise, the tracking of the order of the last values would be more complex. The complexity of the filter is not changed by the copying process.

- At last, the median value must be extracted. If the buffer length is odd, the value at `sorted_array[length/2 +1]` is used; else, the value `sorted_array[length/2]` is the median and thus returned.

All three steps are performed by separate functions.

Choice of the Filter Size

The number of values taken into account to determine the current filtered value is the filter size. Its choice affects the memory demand of the filter, the computation time per measurement and the quality of the output.

For our implementation of the median filter, the runtime can be neglected, since it will never be longer than the sampling interval of 65 ms. The processor is also not concerned with other demanding tasks.

In our case, two arrays (the one with the measured values and the sorted array) of the length of the filter have to be allocated. Therefore, the memory demand is linear with the filter size.

Finally, the resulting quality of the filter is the key factor for the choice of its size. With too few previous values, the filter is not able to neglect many erroneous measurements. A longer filter delays the output of the corrected value too long, because for every actual larger change of the distance, more old values have to be replaced. Thus, a larger filter is slower; for the median filter, the adaption to a change is even slower than for e.g. a moving average filter.

after the consideration of these aspects and experiments with the filter size, we chose a size of 8. In comparison with the unfiltered values, this value produced acceptably fast changes, while effectively filtering wrong measurements.

4 Serial Peripheral Interface

some stuff on spi here

5 OLED Display

some stuff on the display here

6 Additional Tasks

some stuff here why we did not do the extra stuff

7 Evaluation

some stuff here on perfoirmance, work etc