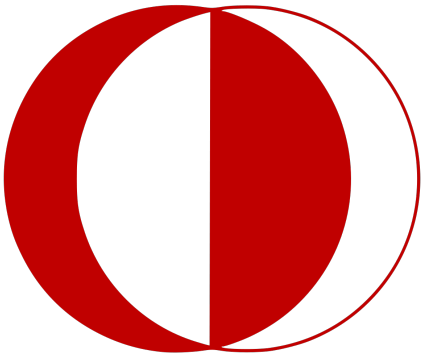

EE447

INTRODUCTION TO MICROPROCESSORS

LABORATORY PROJECT

FINAL REPORT

Başar Kütükcü - 2031110

Faruk Volkan Mutlu - 2031136

Group ID: 8

January 22, 2018

Contents

1 Overall System **2**

2 Modules **4**

2.1 GPIO 4

2.2 ADC 4

2.3 I2C 4

2.4 SysTick Timer 5

2.5 General Purpose Timer 6

3 Conclusion **6**

1 Overall System

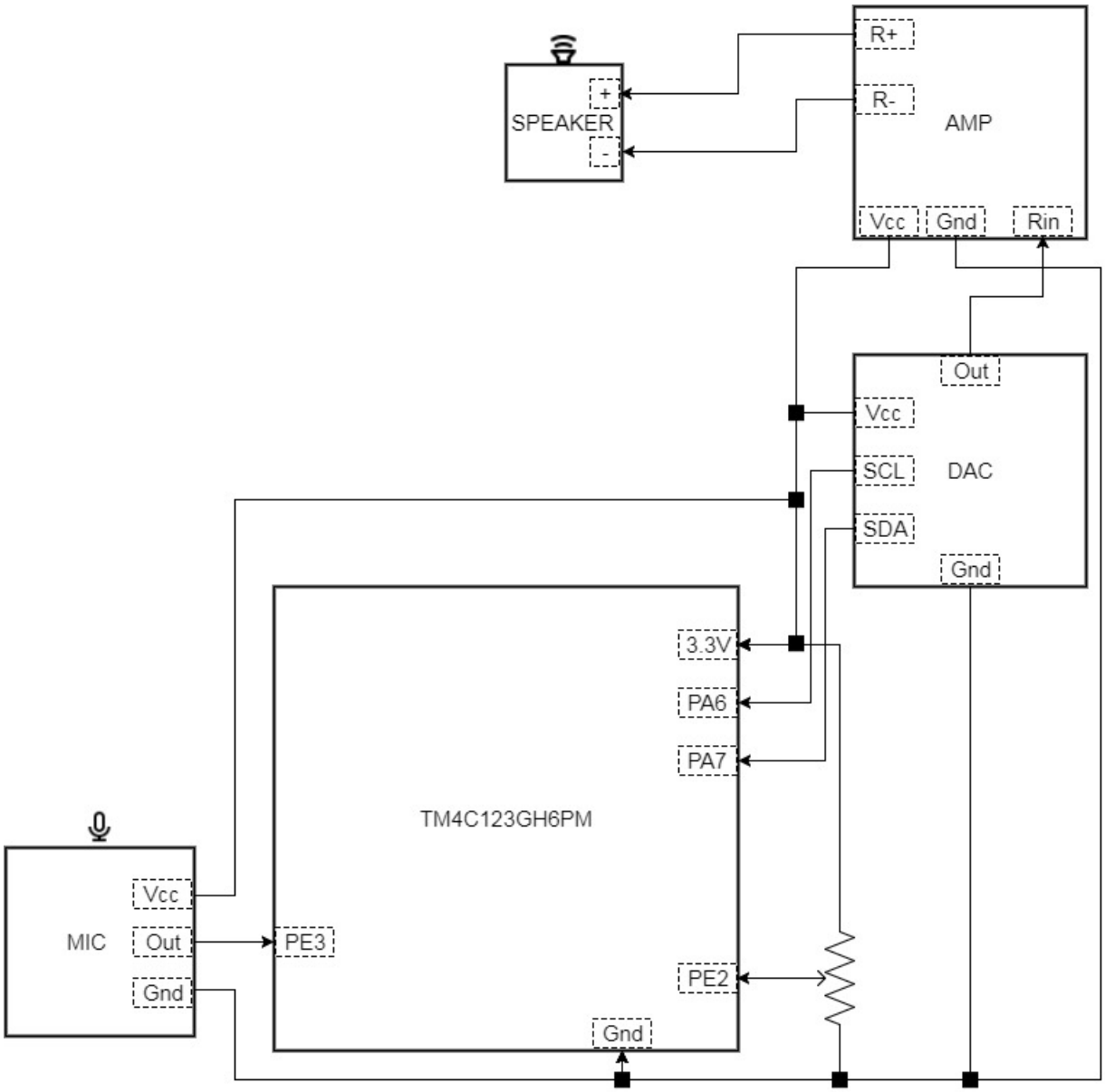

Figure 1: Hardware scheme

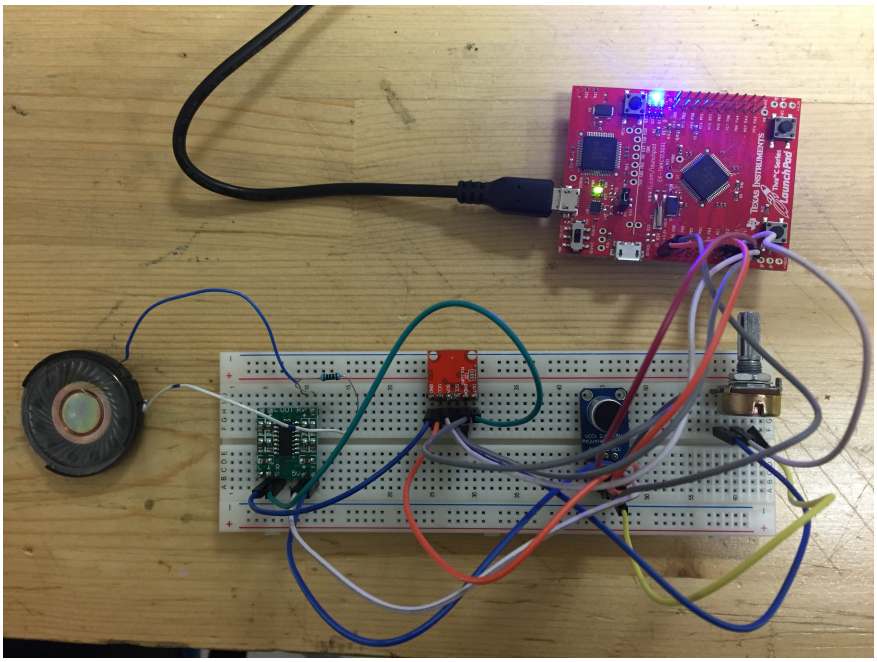

Figure 2: Photo of the setup

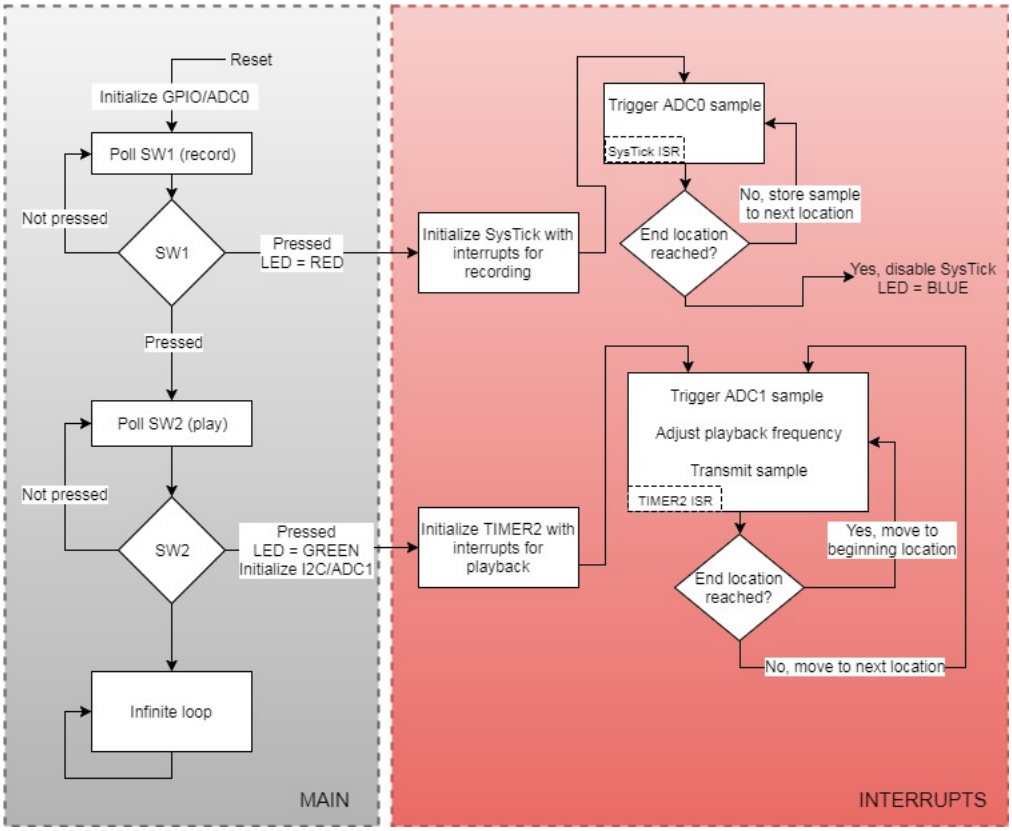

Figure 3: Flowchart of the software

2 Modules

2.1 GPIO

There are two switches (SW1 and SW2) on the board that we use to control the state of our system. These switches are connected to PF4 and PF0 pins respectively so we configured those pins as inputs. We had to unlock PF0 to be able to utilize since it is locked by default. We used pull-ups in these pins' configurations so that when pressed, their input pins go low. Utilizing this, we employed a polling procedure for both switches. We poll SW1 and when pressed, we start recording. We then poll SW2 and when pressed, we start playing the recorded audio.

There is also the on-board LED we use as a state indicator. This LED is connected to PF1, PF2 and PF3 pins which correspond to red, blue and green respectively. If these pins are configured as outputs and are outputting 1, they will drive the LED to their respective colors. We use red to indicate recording is in progress, blue to indicate the recording has finished and green to indicate that audio is being played. For each state we drive the other two pins low so that the colors don't mix.

We configured multiple other pins for other functions, obviously, however we think those can be explained under their respective modules' sections.

2.2 ADC

We use the ADC module of the microcontroller to capture audio from the microphone and to adjust the frequency of the playback. We configured ADC0 with AIN0 for the former and ADC1 with AIN1 for the latter.

We initialized the third sequencer (SS3) in ADC0 to convert the input from pin PE3 which we initialized with ADC function; when configured like this, PE3 acts as the AIN0 channel. In our implementation, SS3 is software triggered and raises the conversion complete flag in its raw interrupt status register after it converts one sample. It samples at the rate of 8 kHz, which is adjusted by the SysTick timer; we'll cover that process bit later on.

We initialized SS3 for ADC1 as well, this time to convert the input from pin PE2 configured as AIN1 channel. PE2 is connected to the middle leg of a 22 kΩ potentiometer which has its other legs connected to 3.3V and GND pins of the board. In our implementation, the voltage level on the middle leg is reflected to the interval of the timer module so we mapped this 3.3V interval to a scale that our timer module can use to adjust the rate at which data is sent to the DAC; we will talk about how the timer handles this a little further in the report.

2.3 I2C

To transmit the recorded audio data to DAC, we used the I2C module, as the project requires. We needed the DAC to convert our digital data to analog signals at 2kHz - 10kHz. However, even the standard speed of I2C protocol is too high for these frequencies. There is an I2C register called Master Timer Period in Cortex M4 to adjust the frequency of SCL and therefore transmission speed. However,

even by writing the smallest possible value to this register, it is not possible to achieve rates of 2kHz-10kHz. Therefore, we make use of a feature of DAC and Timer module of Cortex M4. The feature of DAC is that it keeps the last converted output as long as there is no new input. So, we have configured one of the general purpose timers to have time-outs with adjustable (2kHz-10kHz) periods and made sure the time-out interrupts were sent to the controller. In the ISR of the timer, we transmitted a single data chunk and waited for the next time-out interrupt. This is explained in more detail in the following section.

We used I2C Module 1 which has PA6 as the SCL pin and PA7 as the SDA pin. We initialized the GPIO A port by configuring PA6 and PA7 as digital with their alternative functions enabled and selected as SCL and SDA by writing 3 to their fields in Port Control Register. The first important thing here is that these pins are pulled up since I2C requires the buses to stay high in the IDLE state. We knew that DAC board has built-in pull-ups on its pins, however not every slave device has pull-ups so we aimed to make our code more general. By doing so, the resistances of DAC and MCU were connected parallel, so they became smaller. The operation was not affected so we kept the pull-ups on MCU. The second important thing is we configured the SDA line as open-drain as the manual stated.

We then continued with the configuration of I2C registers. We enabled master mode in Master Configuration since we were going to use our MCU in master mode and DAC in slave mode. We wrote the Master Timer Period Register with 0x02 which is 333kbps in fast mode. Finally the slave address was written as 0x62 with send bit 0 (they became 0xC4 when concatenated).

2.4 SysTick Timer

We configured the SysTick timer to count from its reload value to zero in 125 microseconds so that a rate of 8 kHz could be achieved.

We implemented an interrupt service subroutine (ISR) for the SysTick timer, which is triggered every time the timer count reaches zero. The ISR initiates an ADC0 sample, polls for the conversion complete flag and, when the flag is detected, reads from the SS3 FIFO into a register. Before storing the result to SRAM, there are two conditional stages the ISR goes through.

First, it checks if the sample count is odd or even. If the count is odd, the current sample is just read to the register. If the count is even, the current sample is shifted 12 bits to the left, added to the register that holds the latest odd-count sample and finally the two sample (24-bit) result is stored to the current SRAM location with post-indexing so that in the next iteration the samples will be placed in the following SRAM location. The 12 bit shift for even-count samples makes sure the results are stored back-to-back which is required to completely utilize the SRAM while keeping all 12 bits of each sample.

As the second conditional stage, the ISR checks to see if the location register has the end location of SRAM (0x2000.7FFE in our case since we store in 3-byte chunks) and simply returns if that is not the case. If, however, the end location has been reached, the ISR disables SysTick and drives PF2 high to turn the LED blue.

2.5 General Purpose Timer

We used TIMER2 general purpose timer and configured it as a periodic countdown timer with time-out interrupts. We needed the interrupts to be sent to the interrupt controller, therefore we configured the appropriate NVIC registers (with Priority 2). We also used the prescaler, which acts as a true prescaler in periodic countdown mode. Timer Interval Load Register is adjusted by the samples taken from ADC1. This takes place in our timer ISR, operation of which is detailed below.

When the ISR is called, the following steps are executed. First of all, timer is disabled. Since we had stored our data as 2 samples in 3 bytes previously, in the ISR we check which location we're reading in the SRAM. According to this information, we do one of two things. If we are reading from an odd-count sample location, we read a half-word and clear the most significant 4 bits of the reading, which gives us the sample result. We then increment the read location by one byte. If we are reading from an even-count sample location, we read a half-word and this time, shift the least significant 4 bits of the reading right. We then increment the read location by two bytes to reach the next odd-count sample's location. Remember that we stored data as 2 samples, in 3 bytes. Then, MDR is loaded with 4 most significant bits as leading zeros. The first two zeros are used since we are transmitting in fast mode and the other two indicate the power down select as stated in the datasheet of DAC. Then we write to the MCS register to give commands start and run. We then poll the busy bit of the MCS to wait for the bus to be free and once it is, we send the least significant 8 bits with just the run command. Lastly, we again poll the busy bit and then when free, send an EOT character with just a stop command.

After the transmission is complete, we initiate an ADC1 and we change the Timer interval value according to the result's mapped value. At the end, we check whether we've reached the end location of SRAM. If so, we reload the first location to play our record in a loop. Lastly, we clear the timeout interrupt flag of the timer, enable the timer and leave the ISR.

3 Conclusion

As we finalize our report, we want to take a brief look at the process. There were numerous challenges we faced along the way, but the most troublesome ones stand out to us as deserving of an appearance in this report:

- The project requires a 3 second recording to be stored in the SRAM. With the given sampling rate, this is only possible by utilizing all available locations in the SRAM. Storing all 12 bits together was only possible via storing a half-word, which left 4 bits unused once every two locations since the last 4 bits can't be directly accessed. At first we thought of dumping the least significant 4 bits of each sample but we decided that would result in some loss of quality and although it would be minor, we did not want that in our system. So we decided to store 24 bits in 3 locations for every two samples. Our method to do so was explained previously.

- The project description stated the I2C slave address of the DAC we used to be 0x60. However,

after being unable to transmit any sort of data to the DAC, we decided to see if this was true. We used an Arduino and a premade slave address finder (found online) to detect the slave address of the DAC. It turned out to be 0x62 which solved a crucial problem.

- However, even after solving the above problem, we were not completely sure of the accuracy of the data we were sending to the DAC. Since we were not seeing the expected outputs, we decided to check if our routines worked properly. In order to do so, we enabled the loopback feature on the TM4C123GH6PM I2C module and checked our transmission. As a result we figured out some minor adjustments we needed to make to solve our problems.

- Once we were able to get some recognizable audio from our speaker, we realized the quality was not as we expected. In the project manual, there is a "device addressing byte" which is explained as a requirement for the first byte of the transmission. Since we were transmitting in a loop, we thought that sending this byte at every iteration might have been affecting the quality. We decided to try removing it and observed a significant improvement in quality.

Having stated these, we are happy with how we overcame these problems and also with the end results. We believe that we learned a few valuable lessons during the course of the project but more importantly, we had the chance to put into practice the preparatory work we had been doing in labs throughout the semester, which was a rewarding experience.