Middle East Technical University

Electrical and Electronics Engineering Department

EE447 Introduction to Microprocessors Term Project (2017-2018 Fall Semester) Final Report

Group 40

Deniz Sayın

Berk İskender

Introduction

In this project our purpose was to construct a system able to record human voice and play it back by using the TM4C123GXL Launchpad and external hardware which includes an MAX9814 microphone module, an MCP4725 digital to analog converter module, a PAM8403 stereo speaker amplifier module. The sound sampling rate used in the project is 8 kHz, and the sound reconstruction frequency can be varied between 2 kHz and 10 kHz by the use of an external potentiometer.

This report will give a detailed description of the operation of each module included in the system as well as bonus features. Finally, the challenges faced during the development of the project will be explained along with their causes and solutions. Photos of the complete setup can be seen in Figure 1 and the pin diagram of the connections can be seen in Figure 2. The detailed flowchart of the system's algorithm can be found in the appendix at the end of the document.

We did not feel the need to include any project workload chart, since we did each and every step of the project together and did not divide the workload.

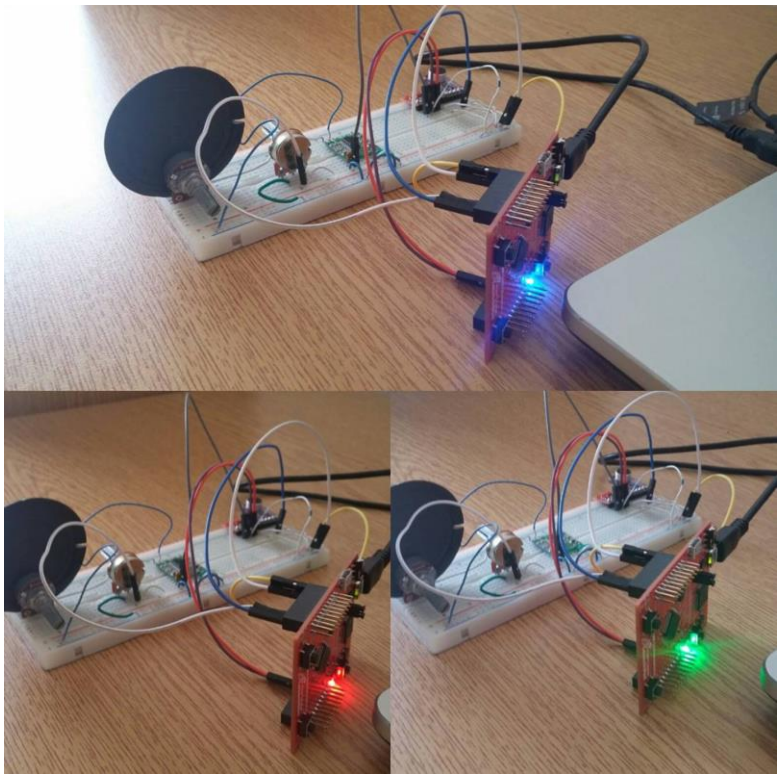

Figure 1: Photos of the Complete Setup

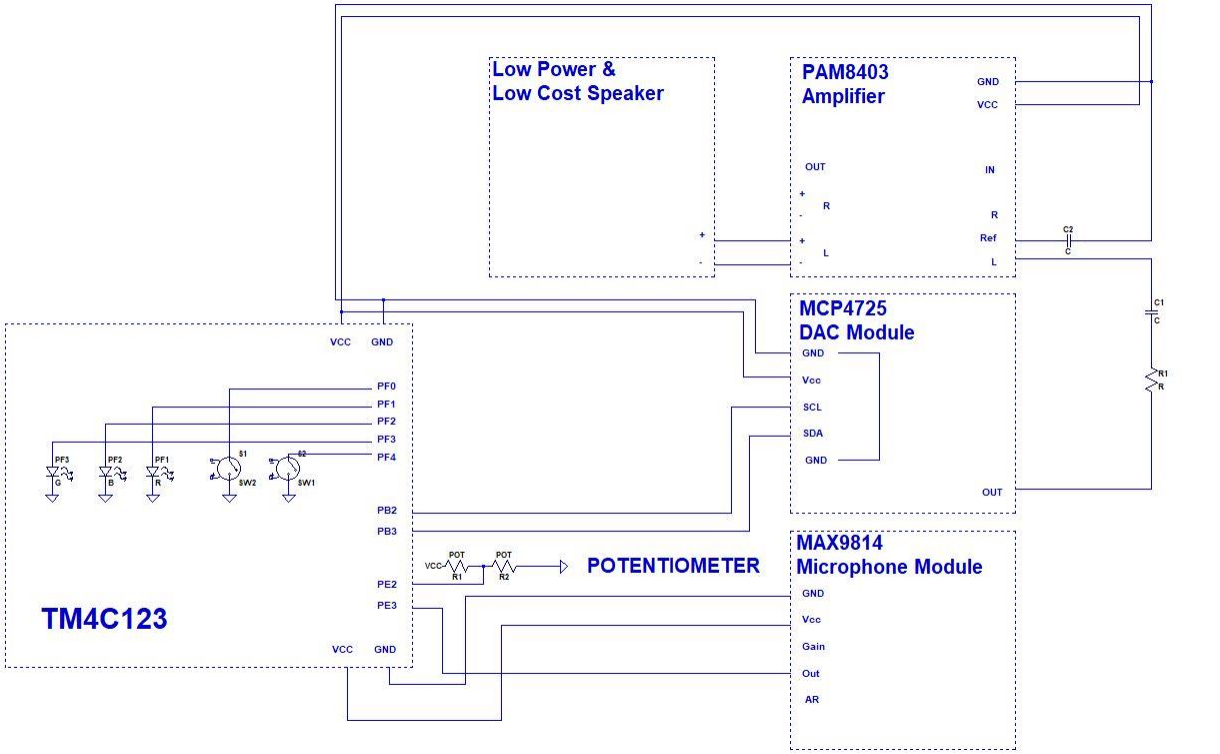

Figure 2: Hardware Scheme of the Setup including Pin Connections

Operational Description

The first concern in the system is being able to select when to record and when to playback the recording. For this purpose, the switches SW1 and SW2 built-in on the launchpad were used. These switches were connected to the General Purpose Input-Output (GPIO) module's port F pins PF4 and PF0, and therefore were configured just like an external switch: port F's clock gating register is unlocked through the RCGCGPIO register, the directions are set as inputs through the GPIO_PORTF_DIR register, the alternate function selection is cleared through the GPIO_PORTF_AFSEL register, the ports are digitally enabled through the GPIO_PORTF_DEN register and pull-up resistors are enabled through the GPIO_PORT_PUR register. Also, SW2 (PF0) has to be unlocked from the GPIO_PORTF_LOCK and GPIO_PORTF_CR registers as it is locked by default, being assigned to the WAKE function of the board.

Now that the decision process is cleared, the next step is recording the sound. A press-and-release of SW1 initiates the recording process. Since it is indicated in the specifications of the project that the sampling rate has to be a 8 kHz, a time-critical sampling approach has to be used. Therefore, a timer has to be used in order for the samples to be evenly spaced with the required frequency. In our case, the SysTick timer provided on the TM4C123GH6PM is used to time the sampling process. Since the PIOSC/4 oscillator source is used, the SysTick counting clock frequency is 4 MHz, which is 500 times 8 kHz, meaning that we need to store a reload value of 500 in the STRELOAD register. Also, the counting value is cleared in the beginning by writing in the STCURRENT register. Then, the timer is enabled through the STCTRL register.

After this, the STCTRL register is polled to await the end of the countdown. Once the countdown finished, the ADC module is used to take a sample from the microphone's output. For this purpose, ADC0 is used with channel 0 (which is connected to port E3 as an alternate function), a 125 ksps sampling rate, using sample sequencer 3 and processor controlled sampling. The sampling sequence is initiated by writing to the ADC0_PSSI register. Then, the ADC0_RIS register is polled to await the end of the sampling sequence, after which the sampled value is taken from the FIFO which can be accessed through the ADC0_SSFIFO3 register. Finally, the sampling interrupt is cleared through the ADC0_CISC register. This completes the sampling process and the sample is stored in the microprocessor's SRAM starting from address 0x20000400, which is right after the stack. The timing and sampling process repeats until 24000 samples (8 kHz times 3 seconds), unless it is interrupted. The stored samples are truncated from 12-bits, which would require around 36 KB, to 8-bits which require only around 24 KB and can fit inside the 32 KB SRAM. The quantization error caused by the truncation is negligible to the human ear. At the end, the number of samples taken is stored in R10 and the SysTick timer is disabled.

Now that the samples have been taken, the playback process can be initiated by a push-and-release of the SW2 button. By default, no playback occurs when no samples have been taken yet. However, if there are some samples stored in memory, whose amount is indicated by the register R10, the playback process begins and continues until all the samples have been communicated to the digital-to-analog converter (DAC). The communication with the DAC is done using the I2C module of the microprocessor. Just as in the sampling process, the communication process is time-critical, meaning that samples are sent to the output with a set frequency. This is why once again the SysTick timer is used to time the flow of the samples from the SRAM to the DAC. Since the frequency of the output has to be variable through a potentiometer, the ADC module is used once again to get the voltage input.

Once the playback process starts, the SysTick interrupt is enabled for the timing of the samples, and the program itself is responsible for changing the reconstruction frequency. A loop continuously takes samples from the ADC1 module using channel 1, using the same initializations and procedure as in the microphone sampling done with ADC0. The sampled analog input value is a 12-bit value between 0x000 and 0xFFF. Since the reconstruction frequency has to vary between 2 kHz and 10 kHz, which correspond to SysTick timer reload values of 2000 and 400 respectively, the input range of 0x000-0xFFF has to be mapped to the range of 400-2000. Since 0xFFF/(2000-400) is approximately equal to 0.39, the analog value is multiplied by 39 and then divided by 100, which gives a value between 0 and 1600. Then, 400 is added to the result, producing a range between 400 and 2000, as intended. This result is then stored in the STRELOAD register to update the SysTick's reload value. This process continues in a loop and the reload value is continuously updated until all the samples are sent.

The communication of the samples is managed by the SysTick timer through an interrupt service routine (ISR), which is called when the timer finishes countdown. The ISR loads a sample from the memory (if there are samples left to be sent), expands it to 12-bits and sends it to the DAC using the I2C module. The initialization of the I2C module is done by writing the constant master address and the clock divider value. Also, the data and clock outputs of the module are

connected to GPIO pins PB3 and PB2 respectively through their alternate functions. To send the data, the slave address 0x62 of the DAC is written to the I2C0_MSA register along with the R/W bit, which is set as 'write'. Then, the most significant 4 bits of the sample are written to the I2C0_MDR data register and is sent with START and RUN bits set in the I2C0_MCS register. Once this byte is sent (an ACK has been received, checked by the BUSY bit in the I2C0_MCS register), the remaining 8 least significant bits of the sample is written inside the data register and sent with a RUN bit. Then, another don't care byte is sent with a RUN bit, and finally a STOP is sent. After all four of these are sent, the sample appears at the DAC output and the ISR exits.

When done with the default system clock, the rate at which samples can be sent to the DAC does not exceed 3.5 kHz, which is not enough for the variable frequency that can go up to 10 kHz. This is why values in the SYSCTL_RCC registers have to be changed to increase the system clock frequency and obtain a maximum reconstruction frequency exceeding 10 kHz.

Bonus Features

The first bonus feature is showing the current state of the system by using the RGB LED on the launchpad. No operation is shown with the blue light, recording with the red light and playback is shown with the green light. This functionality is easily achieved by configuring the GPIO pins PF1, PF2 and PF3 which are hardwired to the RGB LED as outputs. In fact, we think that this feature has helped us greatly with debugging during the development of the project and therefore we can hardly consider it as a 'bonus' feature.

The next bonus feature is being able to interrupt the playback and recording processes by use of their respective switches. For this, we decided to use the GPIO interrupts. The GPIO port F interrupt was enabled and given a priority through the NVIC registers NVIC_EN0 and NVIC_PRI7. Pins PF4 and PF0 were set to edge-sensitive interrupt during a falling edge. This interrupt is masked by default (while the switches are being checked via polling during no operation). Once one of the processes start, the interrupts are unmasked: PF4 is unmasked for recording and PF0 is unmasked for playback. This allows the other button to remain inactive during both processes, while allowing for detection of an interrupt through its own button. When the associated button is pressed, the interrupt occurs and the interrupt handler is called. The interrupt handler sets a flag, disables and clears the interrupt and returns after the switch is released. The respective process ends in case this flag is set, allowing for early termination. This process can be seen clearly in the flowchart given in the appendix.

An extra feature that had to be added for this (already explained in the operational description) is that the amount of samples taken are stored in register R10, and only this amount of samples is sent by the playback procedure, instead of the constant 24000 samples taken/sent in the default operation.

Challenges

Most of the challenges we faced during the project were due to modules or registers previously unknown to us, which made noticing and solving the problems harder.

The first challenge we faced was PF0 being a locked pin and we kept getting hard faults while initializing it and could not figure out the reason for a while. Then, we found the solution on a tech forum where it was stated that PF0 was a locked pin, and then discovered the GPIO_LOCK and GPIO_CR registers in the datasheet and used them to unlock PF0 for use as SW2.

The next and greatest challenge we faced was the implementation of the I2C communication procedure. At first, we could not get any outputs since we used the default slave address of 0x60 instead of 0x62. This problem was solved thanks to a mail from the course coordination. The next issue was figuring out the sequence of byte transmission. It took us some time to realize that the I2C0_MCS register reacted differently to read and write operations. Also, the flowchart given in the datasheet seemed to suggest that we had to send the first byte with RUN and START bits, and the second byte with RUN and STOP bits. To be more in line with the description, we tried to send the second byte with RUN and a third transmission with STOP. We thought that this had to be the correct fashion because we were able to get single value outputs correctly. However, when trying to produce waveforms, the sequence of our outputs got mixed and we were confused. Finally we came up with the idea of adding an additional byte with a RUN, ending with a START&RUN-RUN-RUN-STOP sequence. This got the transmission working. However, the maximum rate at which we could transmit samples was still too slow. We tried to follow directions given in the description, but could not reach the required reconstruction speed. We solved this via trial and error by changing XTAL and SYSDIV values as well as the value stored in the I2C0_MTPR and attained a maximum reconstruction frequency of 12.5 kHz, at which point we left the values as they were and proceeded with the project. We have not faced any problems since those modifications.

The final challenge we faced was during the implementation of the bonus part. We thought we could do it simply by enabling switch interrupts. At first, our handler was being called immediately at the entry of the processes, even though we were not pressing the buttons. We then realized that this was due to the interrupt status being set at the initial press during the polling, and then immediately activating as soon as the interrupts were unmasked. We solved this by clearing the interrupts at the entries of the processes before unmasking. The next problem was that the processes interrupted by the button were restarted immediately. This is because we went back to the key-press part of the program immediately and the interrupting presses were also being interpreted as initial presses. We solved this by debouncing and awaiting the switch release inside the interrupt handler itself. The end result was elegant because both the processes were being interrupted by the same handler with the same procedure and flag.

All in all, we faced a variety of challenges and improved our embedded programming skills with this project, and felt the usual we-did-it happiness upon completion.

Appendix

Flowchart of the Recording/Playback System