Fall 2020 ECSE 444: Microprocessors Final Project Dynamically Sensitive Musical Instrument Group 6 Project Final Report

Mai Zeng
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
mai.zeng@mail.mcgill.ca

Shaluo Wu
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
shaluo.wu@mail.mcgill.ca

Wenhao Geng
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
wenhao.geng@mail.mcgill.ca

Yi Zhu
Department of Electrical and Computer Engineering
McGill University
Montreal, Quebec, Canada
yi.zhu6@mail.mcgill.ca

I. Introduction

In this project, a dynamically sensitive musical instrument is implemented using the B-L475E-IOT01A development board. The target design outcome is that the board can play various musical notes as users interact with it, and it can play all notes at once upon request. Meanwhile, the notes played are displayed in the terminal output graphically. A small animation running with the music melody will show on the terminal. Finally, as an exploration, research of neural network development for the MNIST dataset and CIFAR10 dataset was conducted.

II. Design Problem and Solution

In an effort to create a musical instrument program controllable through user actions, we decided to make use of the accelerometer sensor and small speakers. Ideally, by moving the board in different directions, which is detected by the accelerometer, the board can change the note or volume of the music accordingly. Flash memory and DAC output are utilized to store the prepared sound and output the music data respectively. Furthermore, to enhance interactions with users, the blue push-button on the board and UART display are also included. Specifically, when the music is playing, the corresponding musical notation will be printed on the serial monitor. Once the button is pressed, all previously played notes will appear in the same UART display line, forming a piece of beautiful music sheet.

The diagram in Figure 1 shows the software architecture of our music player. The *Reading Accelerometer* thread keeps detecting the movement every 100ms to decide the note and volume. On the side, it also detects the button press. The *Playing Notes* thread sees the current note and volume and then reads the predefined beats from the Flash memory to play them through DAC. The *UART Transmission* thread will send the string-format music notation according to the note to the exposed port. We have a python script launched simultaneously as our microprocessor project to listen to the exposed port from UART. As a result, it can print the musical

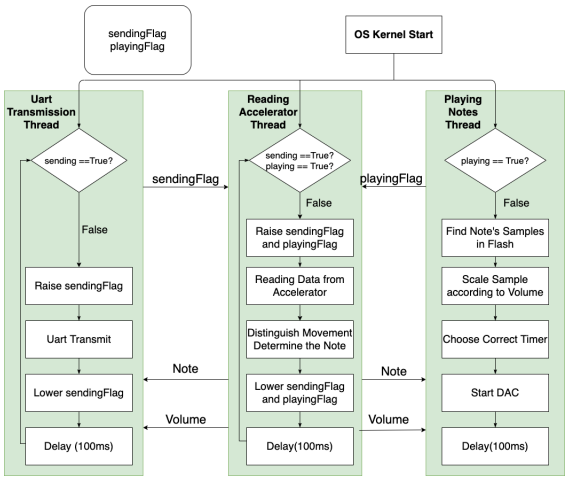

Fig. 1. Flow Chart of Dynamically Sensitive Musical Instrument

notation in the terminal. A list is defined in the script, which saves all the received string from the port. In such a design, when the *Reading Accelerator* thread detects a button press, the *UART Transmission* thread will notify information about the button press to the port in the next cycle. Thus the python script can print all the history notations received in one line to the terminal.

Apart from the aforementioned design of the music player, we also tried to deploy a neutral network model (NN model) to the board. To implement more entertaining functionalities, we utilized a canvas program where users can draw a digit on the interface. The digit will then be passed to the NN model for recognition of which number it is. Each number is associated with one note of music. Thus, users can also change the playing note by drawing a specific number. We performed several experimental deployments for the NN models and successfully implemented a model that can recognize the animal in the input image. However, due to the limited RAM size of the board and the instability of UART-transferred weights, our NN

TABLE I
TEST RESULTS FOR ACCELEROMETER MOVEMENT DIRECTION

Movement Directions	Marginal Values of (X, Y, Z)		
	Test#1	Test#2	Test#3
+X	(**1074**, -158, 948)	(**1294**, -208, 934)	(**824**, -147, 906)
-X	(**-889**, -180, 950)	(**-813**, -243, 966)	(**-826**, -122, 987)
+Y	(233, **641**, 843)	(-175, **815**, 776)	(-212, **473**, 957)
-Y	(71, **-1070**, 1021)	(339, **-608**, 1015)	(136, **-787**, 1039)
+Z	(-147, 174, **1790**)	(251, -244, **1951**)	(-126, 127, **1998**)
-Z	(220, -247, **-149**)	(-178, -329, **-45**)	(225, -158, **-265**)

model for recognizing numbers failed to deploy successfully. We will discuss our NN model deployment discoveries in detail in section V.

III. PRODUCT COMPONENTS

A. Accelerometer

For motion detection of the board in this project, the accelerometer was implemented. To determine the threshold parameter, the UART was utilized in experiments to display the values of X, Y, and Z read by the accelerometer under movements in specific directions. The testing was conducted using the controlled variable method, where the board was moved in only one direction while keeping other parameters unchanged as much as possible. The movement in each direction has been performed three times, and the marginal values, parameters with the highest deviation, are recorded. The detailed testing result is illustrated in Table I. By comparing the data acquired in the most relevant movement direction, bold in each bracket in Table I, with data collected during movement in other directions, the most significant parameters for movement in X, Y, and Z directions have been chosen for threshold values. Finally, ± 400 was selected for movements in X and Y direction, while 1400 and 400 for movement in $\pm Z$ direction. All these parameters are implemented in the *Reading Accelerator* thread for movement determination.

Besides, in order to reduce the sensor sensitivity, the *osDelay* function was called to stop the sensor from reading the data. Since our sensor is detecting some specific movements to determine which note the music player should play, one potential problem is that an additional note/volume change might happen after the user performs an intended note/volume change movement. To avoid this issue, we conducted several tests to find a suitable delay that avoids detecting undesirable movements. The delay of 50-100ms is sufficient to erase the effect of those extra movements.

B. Flash Memory

In Cortex M4, the SRAM total size is 128 Kbytes, which is split into two parts: SRAM1 is 96 Kbytes starting from address 0x20000000, and SRAM2 is 32 Kbytes starting from address 0x10000000. SRAM1 is located in the usual ARM memory space for RAM, while SRAM2 can be directly accessed through Data code and Instruction code buses with 0 wait states and can be used for code execution [1]. Figure 2 taken from [1] illustrates the RAM size of Cortex M4. Since we cannot store all of our music notes into RAM, we need some other types of memory to store all the data.

TABLE II
CRITICAL SECTION BETWEEN DIFFERENT THREADS

	UartTran	ReadAcc	PlayNote
UartTran	-	-	-
ReadAcc	Note, Volume	-	-
PlayNote	Note, Volume	Note, Volume	-

Fortunately, we have a size of Flash memory up to 1 MB on the board. We then could store six different notes into the Flash memory with the same number of samples when the program sets up. However, we store only the sine waves without scaling them to the audible range. More details on the utilization of the Flash memory will be discussed in the latter sections. As the accelerometer readings determine the note getting played and the volume, we multiply the note number with the sample number to know the address of which we wrote the notes into the Flash memory. After retrieving the note, we will scale it according to the volume and play the music through DMA.

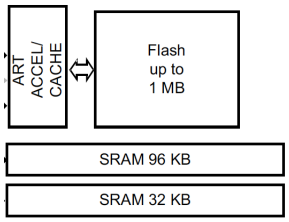

Fig. 2. Flow Chart of Dynamically Sensitive Musical Instrument

C. OS

Table II shows the race conditions and the critical section between different threads. The *UartTran* thread, *ReadAcc* thread and *PlayNote* thread would have a race condition on the variable *Note* and *Volume* since all of them are either reading or writing the variables. We introduced two semaphores "sendingFlag" and "playingFlag" to gate on *UartTran* thread and *PlayNote* thread respectively, thus *ReadAcc* thread can be protected by grabbing both semaphores at the same time. Comparing to simply defining a variable to be used as a mutex, this built-in semaphore (SemaphoreHandle_t) works more stably. Whenever a thread is activated, the threads which are reading or writing from the same memory location (same buffer) or using the same peripheral would stop and wait for the current thread to finish operating on this buffer.

The stack size of each thread is 128kb, which is sufficient for each thread since the buffer that we are using in the threads are declared as a global variable.

D. UART Output on Terminal

We have developed a python script to display the UART output on the terminal. The program on the board will send the corresponding music note symbol, which is essentially an ASCII code string, to the terminal through the UART.

On the terminal side, it will display the note symbol on the serial monitor. Figure 3 shows the G clef ASCII code string displayed on the terminal. The program on the board side will store each note ID inside an array. When a button is pressed, the board will send the symbol of the notes that have been played to the python script. Since each note symbol consists of a complete string, if we want to display all the notes on the terminal like a music score, we need to split the long string into each note and concatenate them together. The reformat algorithm is showing in the latter section, and the music score after the reformatting is shown in Figure 4.

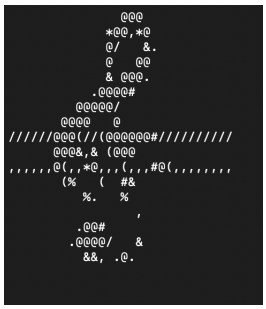

Fig. 3. ASCII code characters array for G clef displayed on terminal

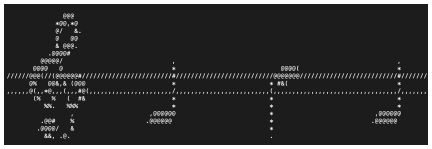

Fig. 4. ASCII code characters array for G clef displayed on terminal

E. Reformatting Algorithm

```
N = number of '\n' characters in the string
n = number of notes has been sent
for i in n:
    for i in N/n:
        musicScore += musicNotes[j*C+i]
        if j = N/n - 1:
            musicScore += '\n'

return musicScore
```

The code snippet above shows the algorithms of reformatting the music note string. Note here *musicNotes[]* is a list sent from the board side. The final String that need to display on the terminal is *musicNotes[]*.

F. Memory Usage

For this project, we need one buffer for playing music notes and one buffer for UART transmission, which totally counts for 22050+31*19*2=22kb of size approximately. This requirement indicates that this project needs at least 23 kb for RAM.

IV. Final Performance

For the final performance of our dynamically sensitive musical instrument, when the program is compiled, our board will play a pre-stored tone, and a treble clef will be displayed on the serial monitor. After users move the board in a particular direction, the board will change the tone and display a note on the terminal accordingly. Users can also press on the blue button to display a piece of visualized melody that combines with all notes performed previously on the terminal. The visual effect of the melody displayed on the terminal of the treble clef and the visualized melody is shown in Figure 3 and Figure 4 respectively.

V. Design Alternatives

A. UART Receive

We tried using UART to transmit data to the board, and this eventually is proven infeasible. We did different experiments to test whether the UART Receive is reliable. Since our objective is to transmit at least 3072 bytes to the board, we first performed an experiment on transmitting all the 3072 bytes to the board with a 2000s timeout. We found out that the UART transmission stops at sending the 384th byte shown in Figure 5. We believe that the array is too long to send, so we then modified the packet size to 383 bytes per packet. By sending multiple packets with 383 bytes per packet to the board, we can send a total of 3065 bytes to the board. However, when we later noticed that some bytes that we sent to the board are disordered. Figure 6 illustrates the result of this experiment. The UART receives either the image data (32*32*3=3072bytes) or the weights (32*32*5*5=25600bytes) from the board. Since it does not provide a mechanism to check the error in the bytes stream sending in, we have to abandon this part of this project. The UART interface values promptness over reliability, and it is not an appropriate solution to our problem.

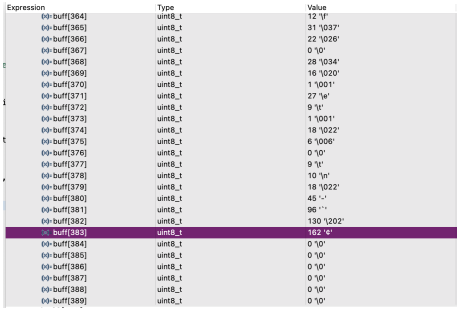

Fig. 5. Result of sending 3072 bytes together

B. Neural Network

In the initial report, we planned to implement a NN model as the core of our project. While this is not included in the final version, we were able to implement it for most parts, and there are valuable thoughts and takeaways. The original blueprint was to utilize the model for recognizing the user's painting on the canvas program. If the user draws an Arabic number

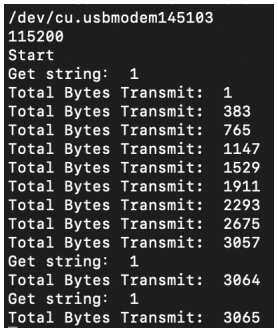

Fig. 6. Result of sending multiple packets

TABLE III
LAYER PARAMETERS AND MEMORY USAGE FOR CNN CIFAR-10 DATASET

	Layer	Layer Weights Size	Output shape
Layer1	Convolution	3x5x5x32 kb	32x32x32 kb
Layer2	Max Pooling	-	16x16x32 kb
Layer3	Convolution	32x5x5x32 kb	16x16x32 kb
Layer4	Max Pooling	-	8x8x32 kb
Layer5	Fully connected	8x8x32x10 kb	10 kb

three, the board would recognize what was written and process the information. Several compromises were made during development. To begin with, the algorithm appeared to work well with complicated figures. Arabic numbers have too few characteristics to be recognized precisely. We then changed the plan to identifying animals, to be specific, cats, dogs, and deer. Table III adopted from [2] shows the layer parameters and the memory usage for the convolutional neural network for the CIFAR-10 dataset. The NN model behaved fine when it was tested separately on our computer. Nevertheless, we had trouble integrating this model into the project. The program always runs out of RAM during runtime. The total RAM size required by weights is 57 KB, and we need two buffers in which each can store the largest size of outputs, and the other can store the largest size of the input, which also required an extra 32*32*32=32kb and 32*32*3=3kb. We need 23 kb for making all four features working so the total that we need for RAM is 57kb+32kb+3kb+23kb=115kb if we want to add the CNN to our project.

To tackle this problem, as mentioned in the previous section, we then tried sending the data through UART to save memory space on the board. As we experimented with this method, we noticed that the output of image recognition sometimes diverts from expectation. The original algorithm performs well, and it is the transmission process that tempered the result. There is no checksum or transmission quality guarantee embedded in the UART function.

In the end, considering the on-time delivery and completeness of the project, the NN model was excluded from the final version. If time permits, it would be worthwhile exploring more options to cope with the memory limit.

C. Microphone Recording

In the initial versions of the project, much effort was put into developing the microphone recording functionality under OS. The idea was to record and to store the samples onto Flash memory directly. However, numerous experiments failed. The program could never save anything into the array. The technical difficulty of this function was later confirmed by the lecture given on Interrupts.

Therefore, this component has been abandoned in the final deployment.

VI. REFLECTION AND FUTURE IMPROVEMENT

During the development stage, we had multiple versions of the program for the dynamically sensitive musical instrument with various functionalities. Some seemed reasonable but ended up being barely feasible, such as the recording function; some appeared ambitious but was eventually proven possible. We selectively kept and omitted certain functions for the integrity of the final deployment, but the scope of our project is not limited to what we chose to present.

If we were able to start the project over, there are several things that we would not try implementing. Firstly, utilizing the MEMS microphone for recording is not reliable under OS. Secondly, in an effort to feed the NN model to the board, we tried using the UART Receive method, which was found to be an unreliable means of transmission. There was no error-detecting algorithm in UART, which makes it unsuitable for essential data transmission. However, we found it fascinating that a small microprocessor chip could almost handle our Machine Learning model. We wish to use a chip with larger RAM and higher peripheral performance to explore the NN model further if possible. On the other hand, we could also look for other digits dataset with higher resolution or more distinct features for current NN model recognition.

REFERENCES

[1] "STM32L4 Series," *STMicroelectronics*. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l4-series.html. [Accessed: 07-Dec-2020].

[2] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," *ArXiv*, vol. abs/1801.06601, 2018.