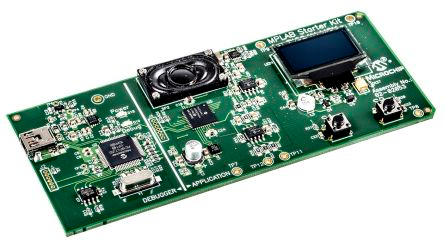
Final Report



H62EDM

Thana Leewongjaroen

Saran Kritsanaviparkporn

**Introduction**

The final report will be discussing about the work done in developing the software for the microcontroller unit “PIC24H”. The task is to develop and design a real-time software, a tool to aid musician. The project features three main function: metronome, tuning utility, and drum kit. Our task is to interpret, design, and implement efficient coding so that the program reach and go beyond the expected functionality given. Since the project is group-based, many skills are essential to complete the task. Such skills, to name a few, are teamwork skills, planning skills, debugging skills, and problem-solving skills.

**General Plan for the Work**

The working hours are divided into three sections: before, during, and after Easter break. The general workload division is as follow:

|  |  |  |
| --- | --- | --- |
|  | Contributor | Finish Date |
| Main Menu | Thana L. | Before Easter break |
| Metronome | Saran K. | Before Easter break |
| Tuner | Saran K. | During Easter break |
| Drum Kit 1 & 2 | Thana L. | During Easter break |

By the time this report is written (during Easter break, 21st of April), the program is already finished. Everything is all according to what is scheduled.

**Get Key Function**

Get Key function is the backbone of the program as it is the function that every function, main menu, metronome, tuner, and drum kit, will be needing to use. For simplicity, the output of the function is defined as in Figure 1.

|  |
| --- |
| #define S1\_SHORT 1  #define S1\_LONG 10  #define S2\_SHORT 2  #define S2\_LONG 20 |

Figure 1

The snippet code from figure 1 serves the purpose as when checking for a function, it is not necessary to remember the each possible output of the get key function, but rather if the programmer wanted to use S1 or S2, long or short, press. It is more intuitive this way.

Many function in the program required a very responsive input, therefore the get key function must be very responsive and accurate. It is necessary that when users let go of the button, the timer must stop immediately and the get key function must return the value in that very instant. This is done with flagging technique as shown in Figure 2.

|  |
| --- |
| while (TMR1 < PR1 )  {  //WAIT FOR USER INPUT  if(SWITCH\_S1 == 0 && SWITCH\_S2 == 1 && flag != 2 && hold < 20){  hold++;  flag = 1;  Delay(100);  }  ….  else if(SWITCH\_S1 == 1 && SWITCH\_S2 == 1 && flag != 0){  break; // This is immediately exit the timer function when user release a  // button after pressing it.  }  ….  } |

Figure 2

When the user press a button, the “flag” will be given a corresponding value. If the button is not release, the value of hold will keep increasing (until the timer time out). However, if the user releases early, the value of the flag is obviously not zero and so does the very action of pressing the button, so the program recognize that and immediately stop the timer. After that, the value of output will be determine by two things, the value of flag and the value of hold.

**Main Menu**

Main menu is the face of the program. It is the first display screen shown when the microcontroller unit is boot up. It contains a pathway to three other functions. In the main menu, user will be able to select each of the three main function, metronome, tuner, and drum kit. User will be using S1 and S2 button to navigate the “pages” of the main menu. If the user wishes to select the navigated menu, they can simply hold S1 button. Holding S2 button will do nothing as it is currently unassigned.

The value “page” will be assigned when a key is press, either to increment it, decrement it, or change its value to the opposite end. The value of page will persist even after entering and exiting the function, so if the user is done using a function, let's say drum kit as an example, the main menu will show drum kit selection interface after the user exited it. The main menu interface corresponded to that very same value as shown in Figure 3 below.

|  |
| --- |
| update\_display(page); |

Figure 3

Navigation through the main menu is very simple via changing the value of *page*. How it is done is shown in Figure 4 below.

|  |
| --- |
| flag = getKey();  if(flag == S1\_SHORT){  switch(page){  case 0: page = 2; break;  case 1:  case 2: page--; break;  }  update\_display(page);  }  if(flag == S2\_SHORT){  ….  }  if(flag == S1\_LONG){  switch(page){  case 0: play\_metronome(); break;  case 1: PlayTune(); break;  case 2: play\_drum(); break;  }  } |

This way the main menu is very stable and robust. Main menu screen withstand abusive testing (repeatedly press S1 and S2 button) without crashing or getting noticeable delay.

**Metronome**

Approach

Since a metronome is used to keep beats, it is important that each metronome ticks are played at a constant tempo. There are numerous ways that can be used to implement the constant tempo, including the use of delay and timer. Since delays will slow down the program and make it hard to for user to input information while the metronome is being played, timer is used instead. A flowchart of how the metronome function operates is shown in figure 5 below.

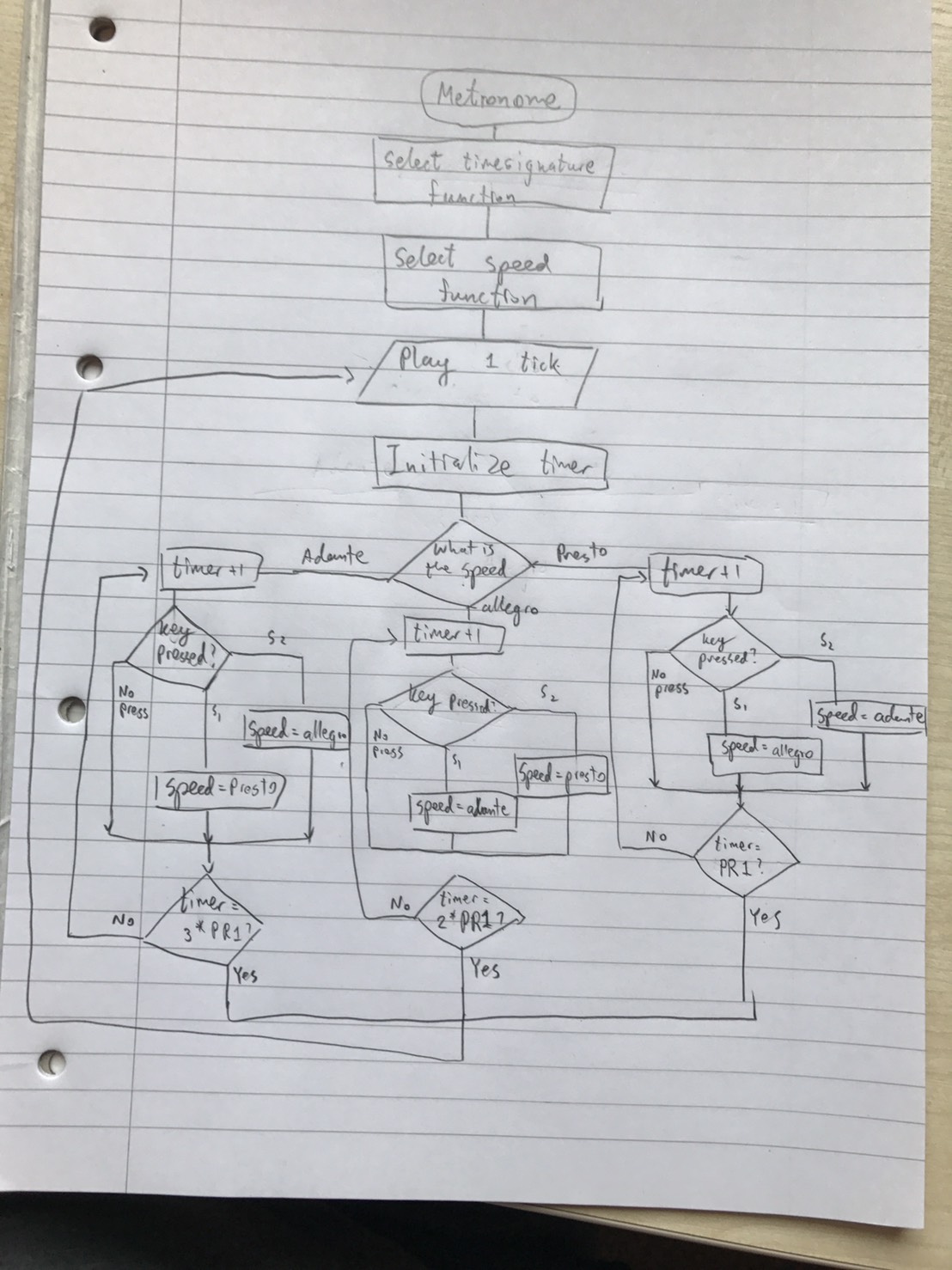


Figure 5

Implementation

The two setup functions at the start of the metronome program, select time signature and select speed, asks the user how they want the metronome to function. The most important elements inside those 2 functions are the loop that listen to what the user inputs and the switch case to aid the user on choosing their setting. Figure 4 is an extract taken from the select key signature function, while figure 5 is an extract taken from the select speed function.

|  |
| --- |
| while(1){  if(press\_flag == 0){  x = getKey();  press\_flag = 1;  }  if(x==S1\_SHORT&& press\_flag == 1){  y--;  press\_flag = 0;  }      if(x==S2\_SHORT&& press\_flag == 1){  y++;  press\_flag = 0;  }  … |

Figure 4

The variable y is used to represent either time signature or speed. Since the user is only allowed to select 2/4, 3/4, or 4/4 as their time signature, and Andante, Allegro, and Presto as the speed, there are conditions that keep the value of y to correspond to those possible options.

|  |
| --- |
| switch (y){  case 2: Display\_ClearScreen();  Display\_Printf(“Speed\nPresto”); break;    case 3: Display\_ClearScreen();  Display\_Printf(“Speed\nAllegro”); break;    case 4: Display\_ClearScreen();  Display\_Printf(“Speed\nAndante”); break;  } |

Figure 5

The switch case in figure 5 is actually inside the loop of both select key signature and speed functions so that the user knows what they are choosing. The reason why the switch case starts from 2 has to do with the speed control in the metronome function, which will be further explained later on.

As the setup completed, the program then enters a loop that plays metronome ticks. As shown in the flowchart, this loop also listens to the user whether the user wants to change the speed. Unfortunately, the loop is not programed to enable key signature change.

When the user selects the time signature, the function passes an integer back to the metronome function. The y value is then assigned to be equal to the variable timesignature, which is used to control how the primary tick is played as shown in figure 6.

|  |
| --- |
| If(tick == 0){  speakerActivate(SPEECH\_ADDR\_METRONOME1, SPEECH\_SIZE\_METRONOME1);  Delay(50);  tick++;  }  else {  speakerActivate(SPEECH\_ADDR\_SELECT, SPEECH\_SIZE\_SELECT);  Delay(50);  tick++;  }    if(tick == timesignature){  tick=0;  } |

Figure 6

Timer polling is used as a wait time before each tick is played. In this wait time, the program is also able to wait for user input in case the user wants to change the speed. The y value from the select speed function is also assigned to be equal to the variable speed, as shown in figure 7.

|  |
| --- |
| For(i=0;i<speed;i++){  TMR1=0;  while(TMR1<PR1/3){  if(SWITCH\_S1== 0&&mark ==0){  change = getKey();  mark =1;  tick=0;  }  if(SWITCH\_S2== 0&&mark==0){  change = getKey();  mark =1;  tick=0;  } |

Figure 7

Calculations are made to find out the speed that the metronome will tick for each tempo mark, using the equation. Since PR1 is equal to 61900, it will take timer1 0.4 second to run the loop, which is too slow for the metronome’s tempo. Hence, PR1 value is divided by 3 so that it will take 0.133 second instead. Indeed the variable speed is used to control how fast each tempo mark is. The corresponding value of the variable speed to each the tempo marks are: 7 for andante (which will give approximately 64bpm), 4 for allegro (gives approximately 112 bpm), and 3 for presto (gives approximately 150 bpm). Note that the length of the tick sound files are not used in the calculation.

Testing

To test if the metronome is accurate, it is compared to the real metronome. Upon comparing the ticks from the 2 metronomes, it is noticeable that the primary beats do not always land on the same time, and after a while, the metronomes go off sync. It is possible that this problem is caused by the unequal length of the tick sound files.

**Tuner**

Approach

A tuner plays sound of a certain frequency that matches with a note in the musical instrument. Since the PIC is capable of playing sound files, the sound of each notes should be stored inside the PIC and is playable when selected. To explain how the tuner will work, a flowchart is draw in figure 8.

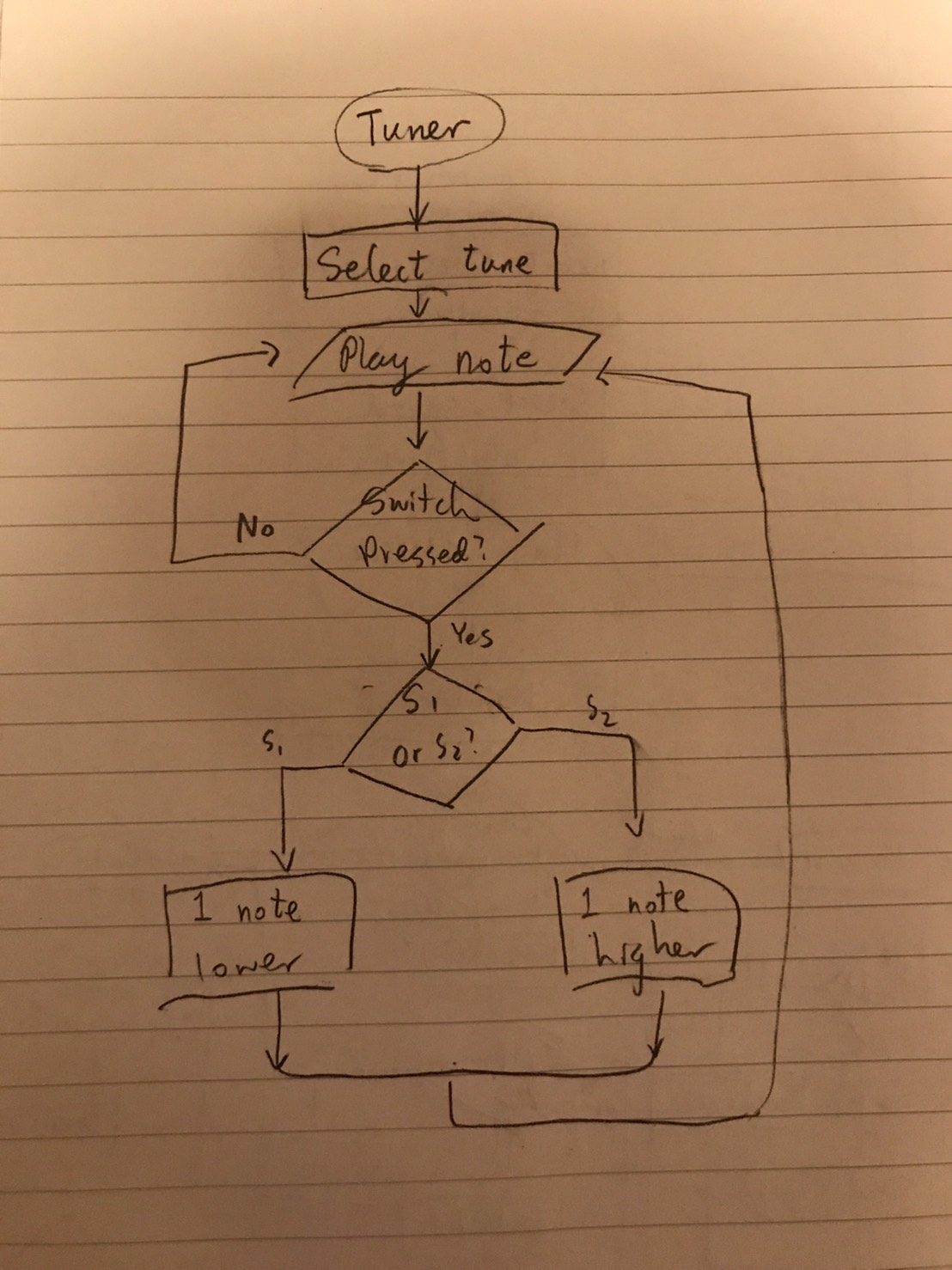


Figure 8.

Implementations

The setup stage of the tuner is very similar to that of the metronome. However, it is written a little differently. At the start, the program asks the user to select a tune the function SelectTune. Instead of having a loop inside the function, a loop is pre-declared in the main function as shown in figure 9.

|  |
| --- |
| while(1){  note = SelectTune(note);  … |

Figure 9

Inside the function SelectTune, getKey function is called to receive the user input. After receiving the input, the function will translate the returned integer from the getKey function into a command know in the function. Indeed, short presses means changing the nots and long presses means select or exit. The process is shown in figure 10. Note that each note corresponds to a number, since there are 7 possible tunes, the value of variable note will only be from 1 to 7.

|  |
| --- |
| key = getKey();  if(key==S1\_SHORT)  note--;    if(key==S2\_SHORT)  note++;  … |

Figure 10

Each time a player chooses to change the select note, the note displayed on the screen will change accordingly. The switch case in figure 11 is used to complete that process.

|  |
| --- |
| switch(note){  case 1: Display\_ClearScreen(); Display\_Printf("C"); break;  case 2: Display\_ClearScreen(); Display\_Printf("D"); break;  case 3: Display\_ClearScreen(); Display\_Printf("E"); break;  case 4: Display\_ClearScreen(); Display\_Printf("F"); break;  case 5: Display\_ClearScreen(); Display\_Printf("G"); break;  case 6: Display\_ClearScreen(); Display\_Printf("A"); break;  case 7: Display\_ClearScreen(); Display\_Printf("B"); break;  } |

Figure 11

After all the setups, the program comes back to the main function and goes around a loop playing the tunes. Since the each playable tune has a corresponded value, another switch case is used. However, instead of displaying the screen, it is used to play the tune, as shown in figure 12.

|  |
| --- |
| switch(tune){  case 1: speakerActivate(SPEECH\_ADDR\_CTUNE, SPEECH\_SIZE\_CTUNE); break;  … |

Figure 12

Since the tunes played are from sound file, there should be delays in between each time the file is played to prevent possible errors. The tuner should be able to change the tune while it is playing, hence, instead of using delay, timer is used instead. The process is illustrated in figure 13 below. Note that once the program detects user input while it is inside the timer loop, the loops are exited immediately, so that the new tune can be played.

|  |
| --- |
| for(i=0;i<2;i++){  TMR1=0;  while(TMR1<PR1){    if(SWITCH\_S1 == 0 || SWITCH\_S2 == 0){  note = SelectTune(note);  i=2;  break;  }  }  } |

Figure 13

**Drum Kit**

Part 1

The first part of drum kit is very simple. We just need the board to recognize the input and returns a sound. If the S1 input is long, then process to part 2. If the S2 input is long, back to main menu. First part of drum kit is all about key detection. Flowchart (Figure 5) is shown below.

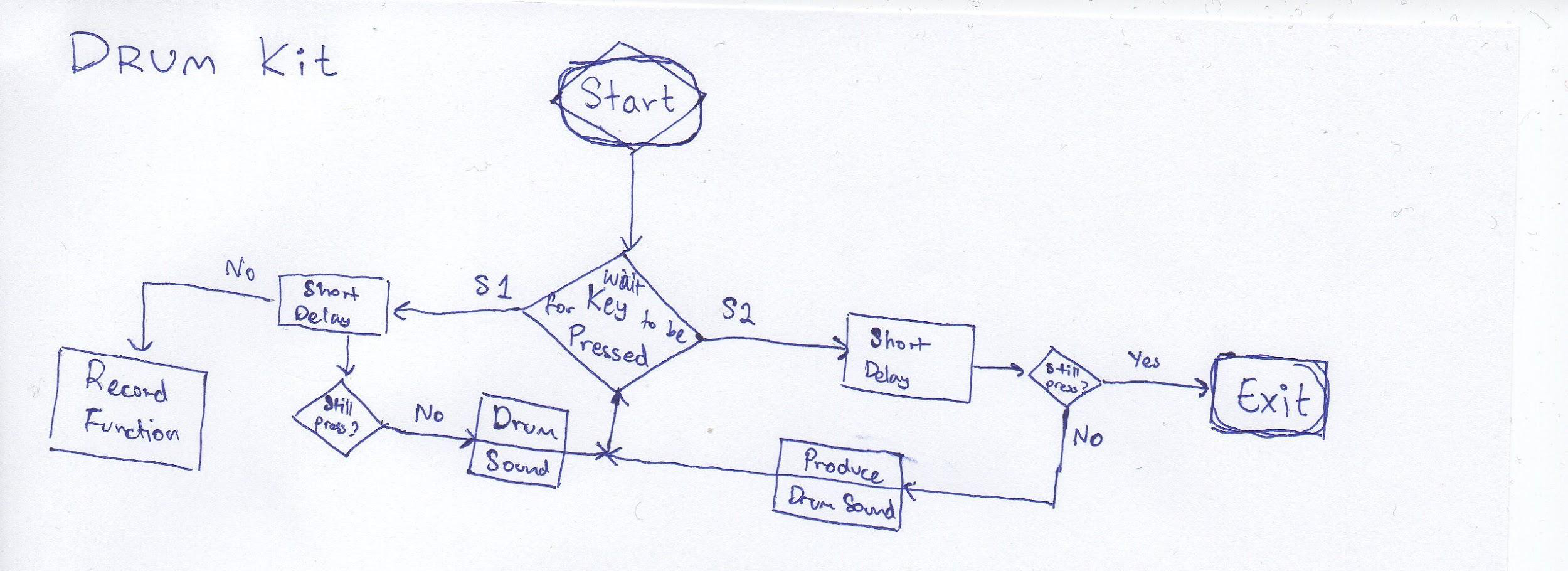


Figure 5

As mentioned before, first part of drum kit is about key detection. The code snippet of the function is shown below in Figure 6.

|  |
| --- |
| while(Loop){    if(SWITCH\_S1 == 0 || SWITCH\_S2 == 0){  flag = getKey();  if(flag <= S2\_SHORT){  speakerActivate(SPEECH\_ADDR\_SELECT, SPEECH\_SIZE\_SELECT);  }  if(flag == S1\_LONG){  record\_drum(); //proceed to part 2  Display\_Printf("\n\nPLAY DRUM KIT"); //after returning from part 2 func.  }  if(flag == S2\_LONG){  Loop = 0;  }  }  } |

Figure 6

Because the getKey function is written to be very responsive (see Figure 2), this function, therefore, is also very responsive. There were no problem or program failing when repeatedly press S1 and S2.

Part 2

The general idea of how it works is best explain using the following flowchart (figure 7)

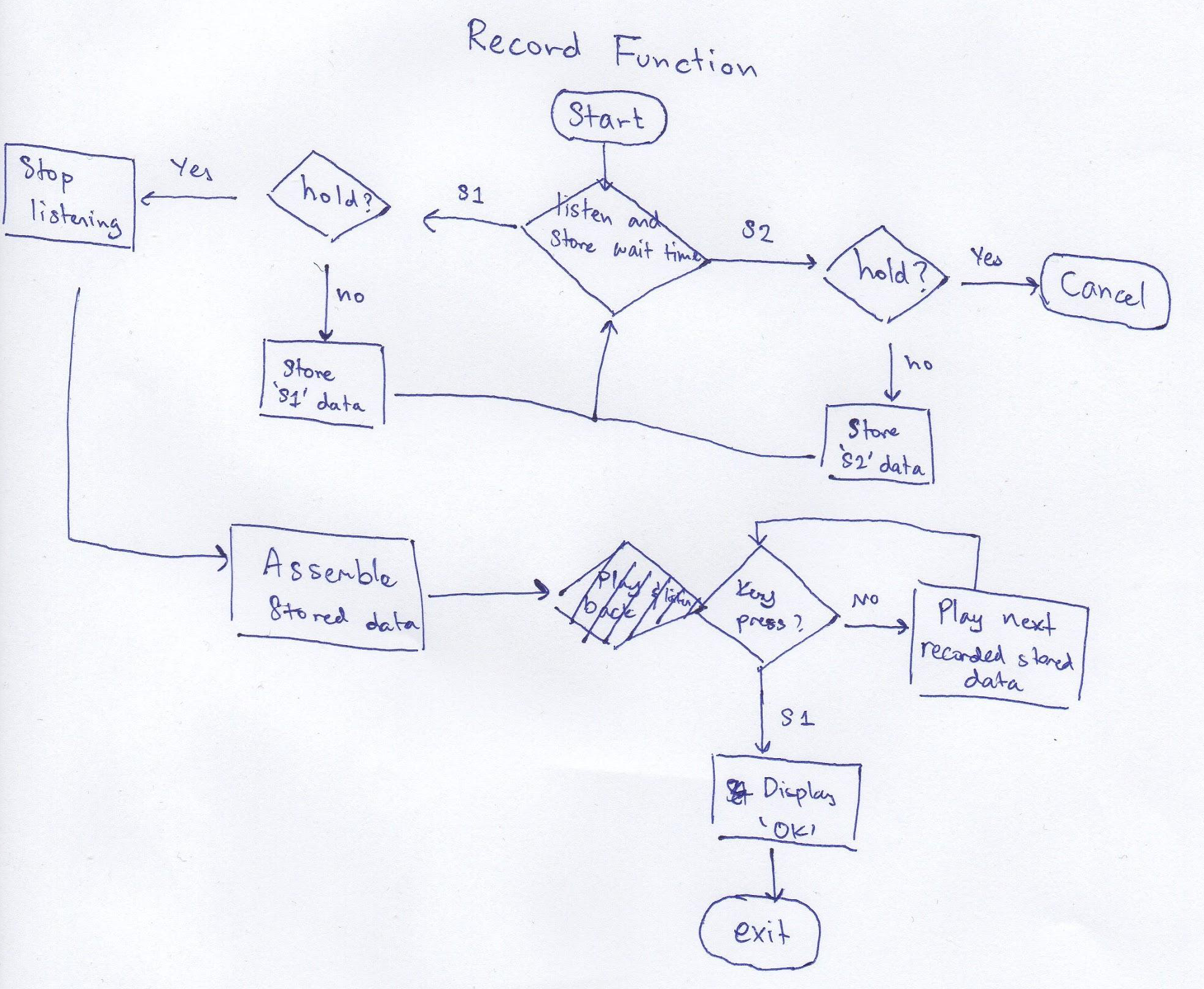


Figure 7

The idea is to *record* the time lapsed between each pressing and replay that in a loop. There are several ways to approach the system. Timer is the first thing to come to mind, but as we want to use the getKey function universally, different approach using delay is considered. The idea behind the *delay* approach is similar to how the getKey function determine whether the key is long press or short press (see figure 2). How the function record the delay is shown in Figure 8 below.

|  |
| --- |
| while(loop){  hold++;  Delay(10);  if(SWITCH\_S1 == 0 || SWITCH\_S2 == 0){  flag = getKey();  if(flag <= S2\_SHORT){  speakerActivate(SPEECH\_ADDR\_SELECT, SPEECH\_SIZE\_SELECT);  time1[key] = hold;  hold = 0;  key++;  }  if(flag >= S1\_LONG){  time1[key] = hold;  loop = 0;  }  } |

Figure 8

As each loop passes, the value of hold will increase. If the button is press shortly, the microcontroller will play a sound, then the value of hold will be stored in *time1[key]* and the value of hold will be reset. If S1 is being hold, then the variable *loop* will be zero, thus ending the loop. Then the microcontroller unit will keep repeating the stored pattern until the long press is registered. The snippet code can be found in Figure 9.

|  |
| --- |
| Display\_Printf("\n\nPLAYING FROM RECORD");  loop = 1;  while(loop){  for(i = 1; i <= key; i++){  speakerActivate(SPEECH\_ADDR\_SELECT, SPEECH\_SIZE\_SELECT);  if(SWITCH\_S1 == 0 || SWITCH\_S2 == 0){  return;  }  Delay((time1[i]\*modifier)/2);  }  for(i=0;i<100;i++){  Delay(5);  if(SWITCH\_S1 == 0 || SWITCH\_S2 == 0){  return;  }  } |

Figure 9

Many pattern were tested. There is slight discrepancies when the time gap between press is large. This is likely due to the run time of the loop and inability of delay to act independently, a feature that timer has. The attempt to fix it is by, when the delay size gets larger, the play delay will be shorter. On the plus side, the program is very stable and show no sign of crashing. Since Delay(10) is only used, there were no problem detecting key, which will return user to the drum kit part 1 function.

The drum kit 2 function also has another variable named *modifier*. It can be change to increase or decrease the gap between press. However, the function occasionally cause crashes in the problem and since been removed. *Modifier* variable still existed in the latest program version, but it has fixed value and has no functionality as of the moment.

**Combining Component**

Combining component were taken into account since the earliest development of the program. Between the two group members, we establish, beforehand, the name of the function that the main.c can calls. We work in pair via GitHub, so any changes to the main.c will be notify to both party. We also discuss about possible global variable (#define S1\_Short 1) beforehand so both party and make use of the variable, knowing what it is.

**Final Testing**

A final testing was done after the every components were put together. Each component was tested for the last time. However, the focus of the testing is on changing from one component to another. The testing result was rather unpredicted due to certain issues that happened. One of the issues was the crash of the program. After running the program and changing to other components, the PIC restarts itself. The other issue was the LED screen doing blank while speaker is still working.

**Result and Conclusion**

The result of the project satisfies the objective of the assignment. However, there are still room for improvements, as the test results suggested. In doing this assignment, not only the skills gained during the first few weeks of this module was used, real life skills, including: parallel coding for speed development and communication within teamwork, were also practiced.

**Reflection on the Work**

If this project were to be done again, time management and plans on how the tasks were to be approached will be done differently. Due to bad time managements, the testing in the tuning utility was not carried out. Additionally, lack of thorough thoughts on how the tasks were to be approached caused massive delays in coding.