Mini-Project:
Keyboard Synthesizer

Maazin Zubair - mzuba002
Zerong Cai - zcai047

Demo Link: https://www.youtube.com/watch?v=XiGN0LQf3ps

Project Description:

Summary

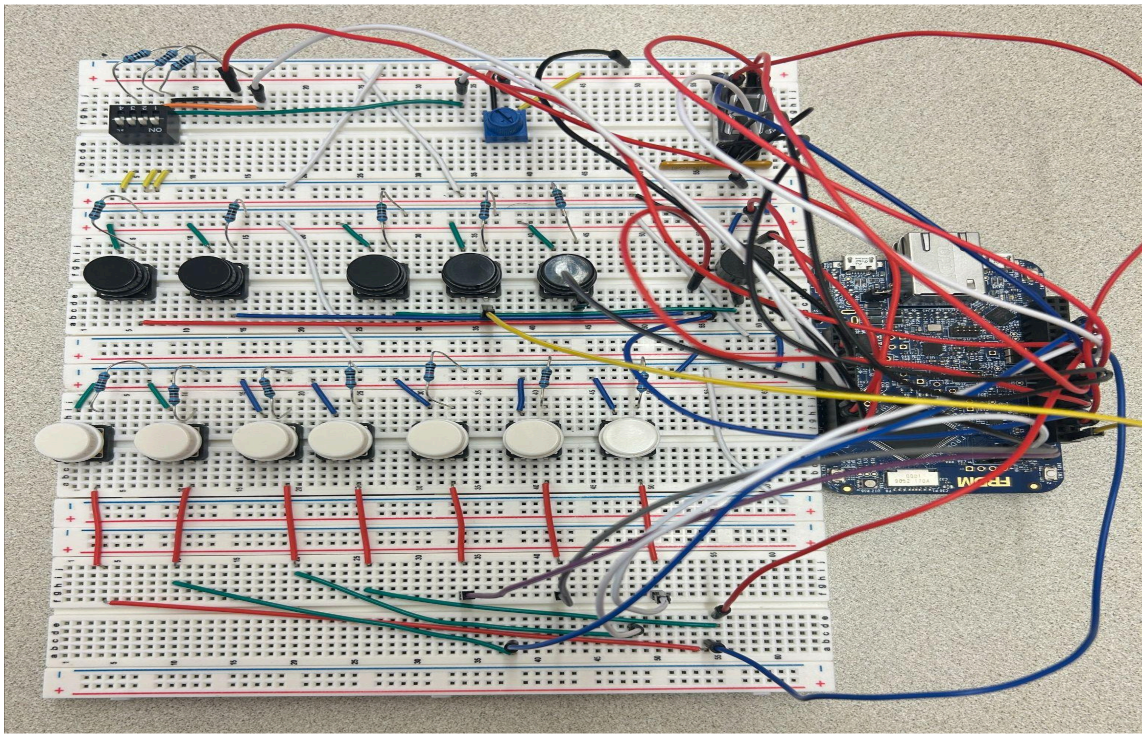
This project brought together multiple core concepts from class—GPIO, ADC, PWM, and timers—to create a working piano keyboard synthesizer on the K64F microcontroller. From wiring the circuit to writing and testing the code, we integrated different features like real-time sound output, octave switching, and basic audio recording/playback. While we faced hardware limitations and bugs along the way, especially with the dip switches and buzzer control, we were able to troubleshoot and adjust our design to meet the project's main goals. Overall, this project helped us apply what we've learned in a hands-on way and challenged us to think critically about both hardware and software design.

Requirements/goals

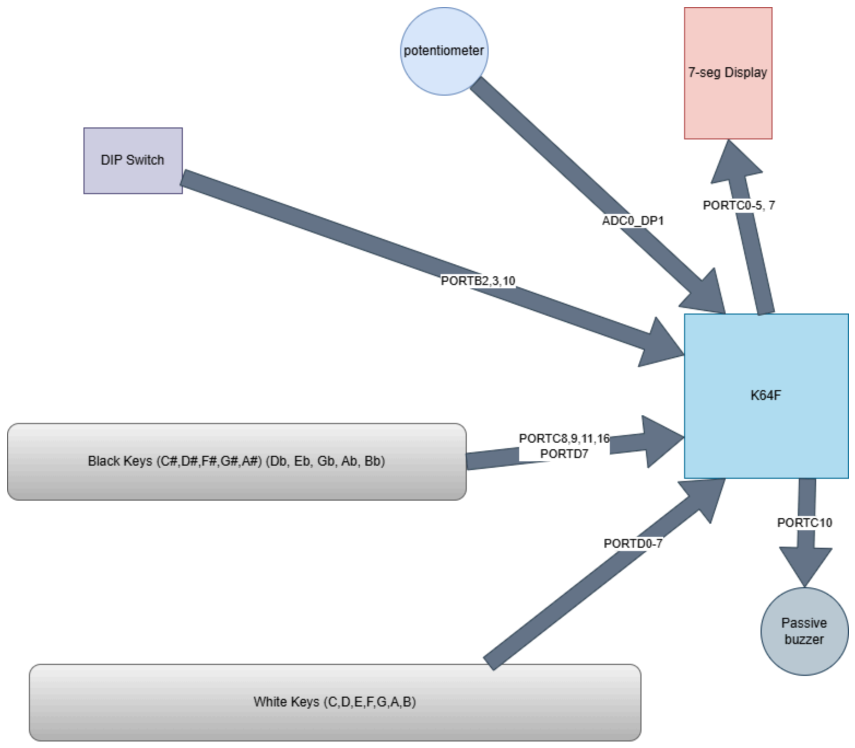
Our project is a piano keyboard synthesizer, where the goal is to ultimately replicate the features of an actual keyboard, including recording and playback, volume adjustments, and octave changes. Topics used include GPIO, ADC, and PWM. PWM was used to play different notes on the passive buzzer, and ADC was used for volume and octave change. Requirements for components include buttons for keys, a 7-segment display for displaying the notes, a potentiometer for ADC, and a dip switch to enable different features. For this project, it is required to be able to play and display the correct notes in each octave, and record and replay every note based on its frequency and duration.

System Design:

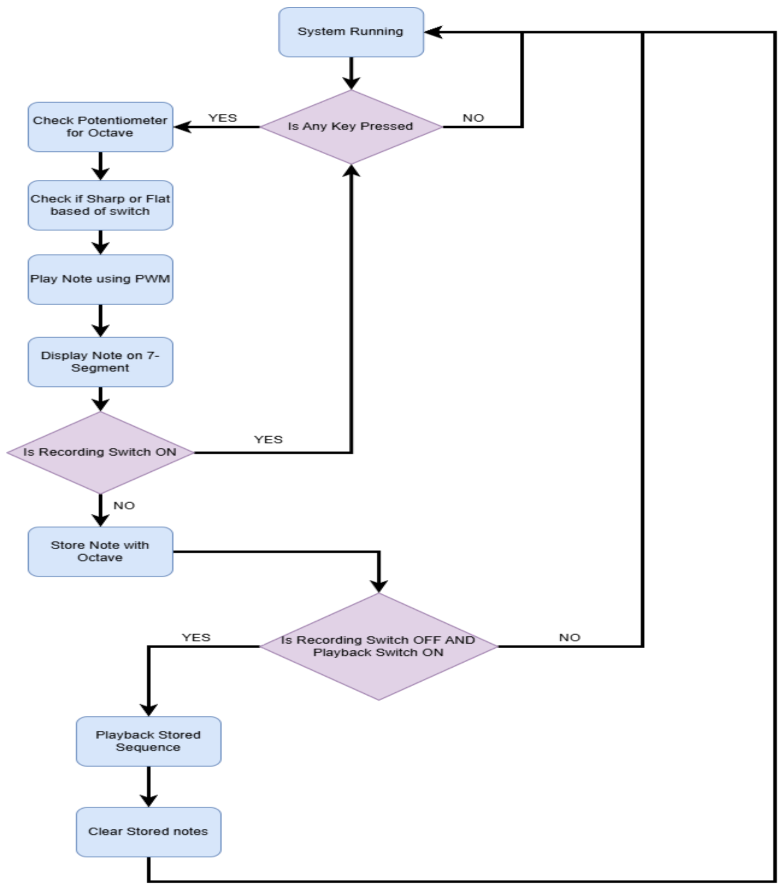
Picture of Circuit



Block Diagram

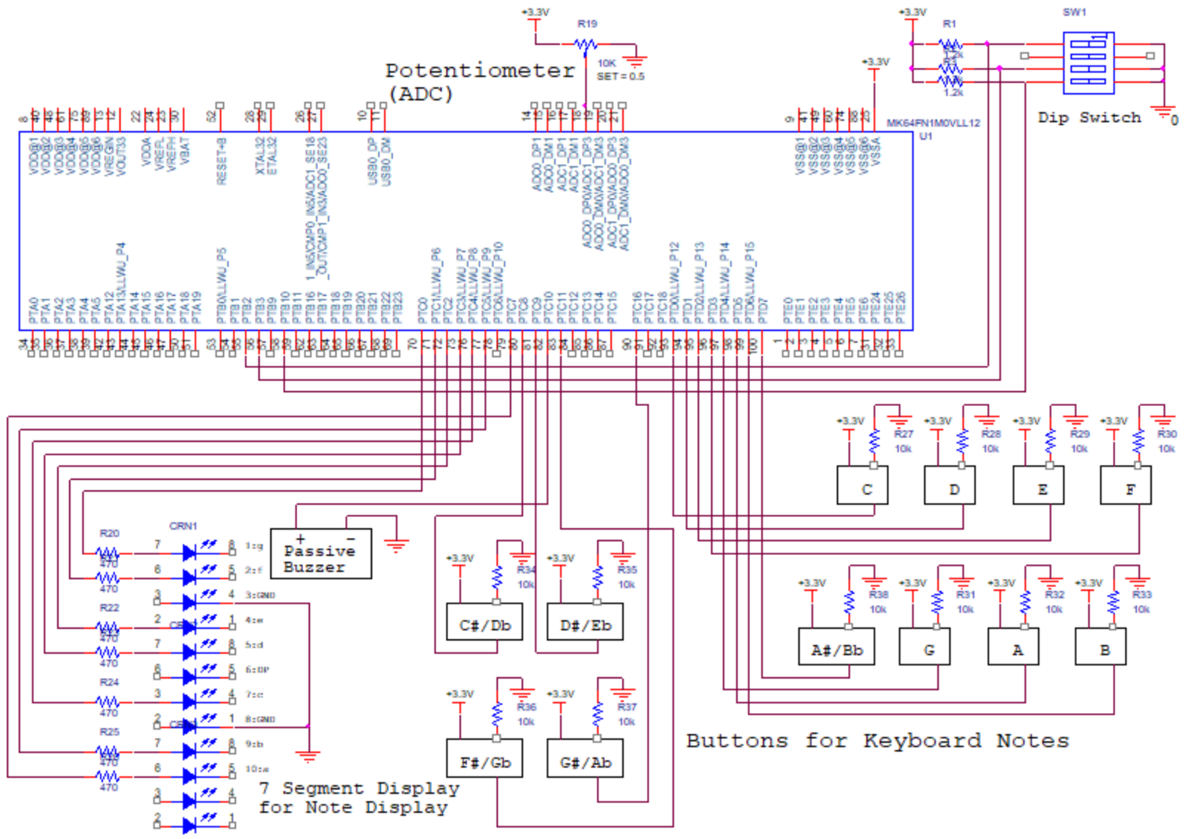


Flow chart



Implementation Details:

Schematics



* Power Symbol +3.3V indicates that 3.3V is coming from the K64F, which is used to save space due to a lot of components.

The image above is our schematic diagram of our circuit for the project, which accurately maps all of the connections for the components with the K64F. As shown, 12 buttons in total were utilized, where 7 buttons were used for the white keys (notes C, D, E, F, G, A, B) and 6 buttons were used for black keys (sharps/ flats notes C#/Db, D#/Eb, F#/Gb, G#/Ab, A#/Bb). In addition, a dip switch is used where switches 1,2, and 4 were utilized for three different features. A segment display was used to display the notes. A potentiometer was used for ADC such as volume and octave change. Finally, the circuit utilizes a passive buzzer to play the notes, where it is controlled by PWM created by the K64F.

Code

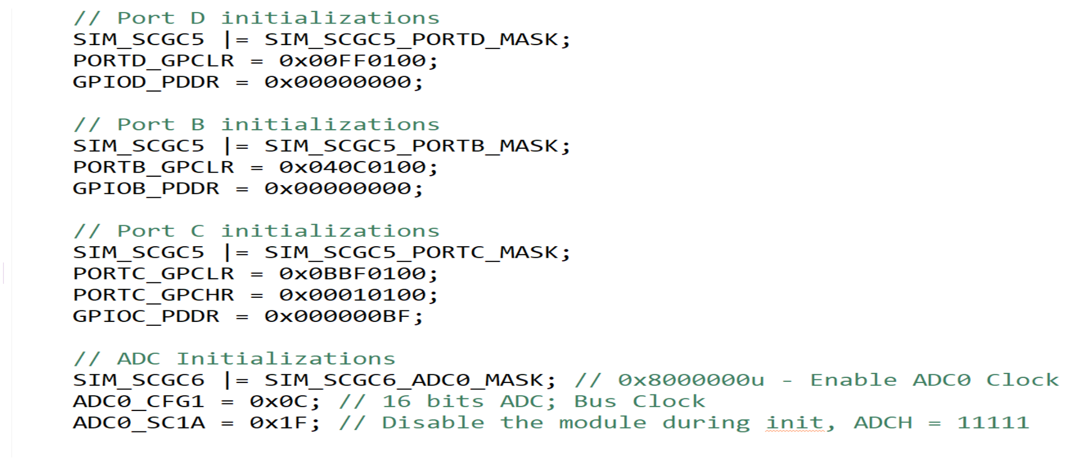
K64F Pins Initializations and Purposes

Using the K64F, various ports and pins were utilized in order to build the circuit for the keyboard. PORTD pins 0-6 were configured to GPIO and initialized as inputs, where the pins 0-6 correspond respectively to the notes of the white keys C-B. PORTC pins 0-5 and 7 were configured to GPIO and initialized as output for turning on the 7 segment display that displays the notes. Additionally, pins from PORTC and PORTD were configured to be used for the black keys. In PORTC, pins 8, 9, 11, and 16 represent the notes C#/Db, D#/Eb, F#/Gb, and G#/Ab and

PORTD pin 7 represented the note A#/Bb. In addition to PORTC, PORTC pin 10 was configured as FTM3_CH6 and enabled to utilize FTM3 channel 6 for PWM.

PORTB pins 2,3, and 10 were also configured to GPIO and initialized for the dip switch. Pin 2 represents the mode to switch between displaying sharps or flats when the black keys are pressed. Pin 3 is used for recording and pin 10 is used for playback.

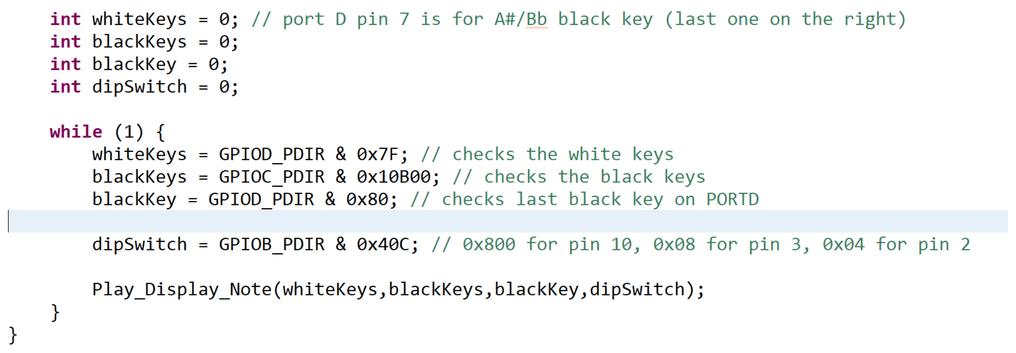
Finally, the ADC0 clock is enabled for the potentiometer and the bus clock for ADC was enabled for ADC0_DP0, where the conversion resolution was set to 16 bits. In addition, we are using the ADC channel 0 for single-ended DADP0 input.



```
// Port D initializations
SIM_SCGC5 |= SIM_SCGC5_PORTD_MASK;
PORTD_GPCLR = 0x00FF0100;
GPIOD_PDDR = 0x00000000;

// Port B initializations
SIM_SCGC5 |= SIM_SCGC5_PORTB_MASK;
PORTB_GPCLR = 0x040C0100;
GPIOB_PDDR = 0x00000000;

// Port C initializations
SIM_SCGC5 |= SIM_SCGC5_PORTC_MASK;
PORTC_GPCLR = 0x0BBF0100;
PORTC_GPCHR = 0x00010100;
GPIOC_PDDR = 0x000000BF;

// ADC Initializations
SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK; // 0x8000000u - Enable ADC0 Clock
ADC0_CFG1 = 0x0C; // 16 bits ADC; Bus Clock
ADC0_SC1A = 0x1F; // Disable the module during init, ADCH = 11111
```

How the system works

Polling is utilized for this project, where each activity is checked continuously. The functions for playing the notes have been put into header files for better organization. The function for playing the keyboard takes in inputs from the buttons and dip switch to make decisions such as what kind of note to play and whether recording or playback is on:

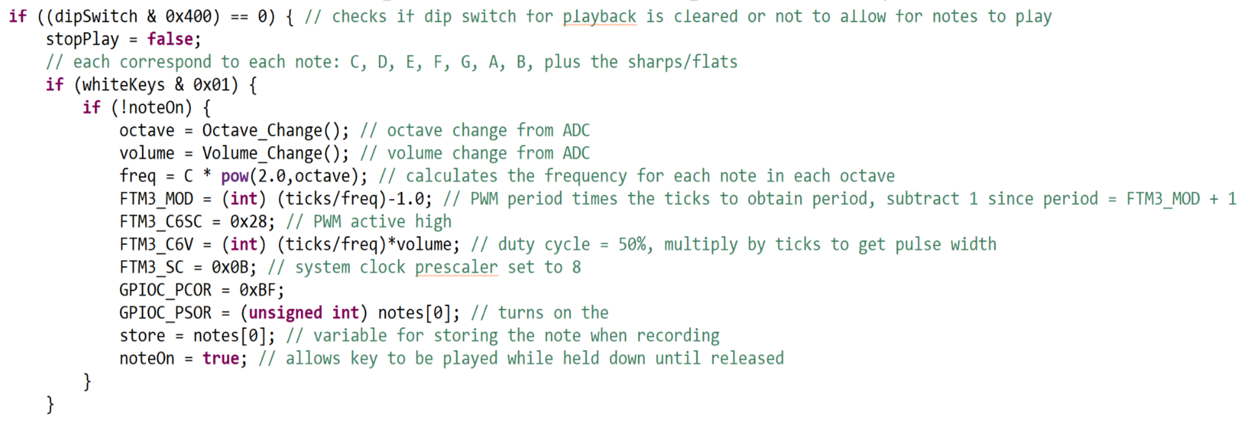


```
    int whiteKeys = 0; // port D pin 7 is for A#/Bb black key (last one on the right)
    int blackKeys = 0;
    int blackKey = 0;
    int dipSwitch = 0;

    while (1) {
        whiteKeys = GPIOD_PDIR & 0x7F; // checks the white keys
        blackKeys = GPIOC_PDIR & 0x10B00; // checks the black keys
        blackKey = GPIOD_PDIR & 0x80; // checks last black key on PORTD

        dipSwitch = GPIOB_PDIR & 0x40C; // 0x800 for pin 10, 0x08 for pin 3, 0x04 for pin 2

        Play_Display_Note(whiteKeys,blackKeys,blackKey,dipSwitch);
    }
}
```

How Notes are Played

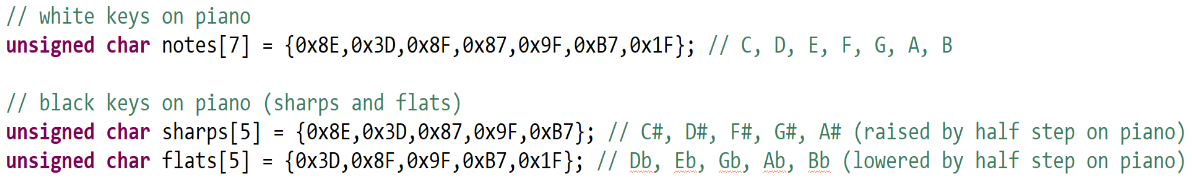
When a note is played, switch 1 is checked to make sure the keyboard is not in playback mode. If it is not, then any white key or black key can be played, where it first takes the octave and volume based on ADC. A boolean value “noteOn” checks if a note was being played or not, which allows the note to be played while being held down and turned off when released.

To play the note, PWM is generated from the flextimer channel 6 of the K64F using the CLK system frequency or 21 MHz, the prescaler which was set to 8, duty cycle, and frequency of the notes. For this project, octaves 1-7 are used, where the frequency of each octave is given by the the frequency of octave 1 (default) times the octave to the power of 2, since the frequency for each note in each octave is twice the frequency of the previous octave. The image below checks if the C note button is pressed, where this is repeated for every note.



```
if ((dipSwitch & 0x400) == 0) { // checks if dip switch for playback is cleared or not to allow for notes to play
    stopPlay = false;
    // each correspond to each note: C, D, E, F, G, A, B, plus the sharps/flats
    if (whiteKeys & 0x01) {
        if (!noteOn) {
            octave = Octave_Change(); // octave change from ADC
            volume = Volume_Change(); // volume change from ADC
            freq = C * pow(2.0,octave); // calculates the frequency for each note in each octave
            FTM3_MOD = (int) (ticks/freq)-1.0; // PWM period times the ticks to obtain period, subtract 1 since period = FTM3_MOD + 1
            FTM3_C6SC = 0x28; // PWM active high
            FTM3_C6V = (int) (ticks/freq)*volume; // duty cycle = 50%, multiply by ticks to get pulse width
            FTM3_SC = 0x0B; // system clock prescaler set to 8
            GPIOC_PCOR = 0xBF;
            GPIOC_PSOR = (unsigned int) notes[0]; // turns on the
            store = notes[0]; // variable for storing the note when recording
            noteOn = true; // allows key to be played while held down until released
        }
    }
```

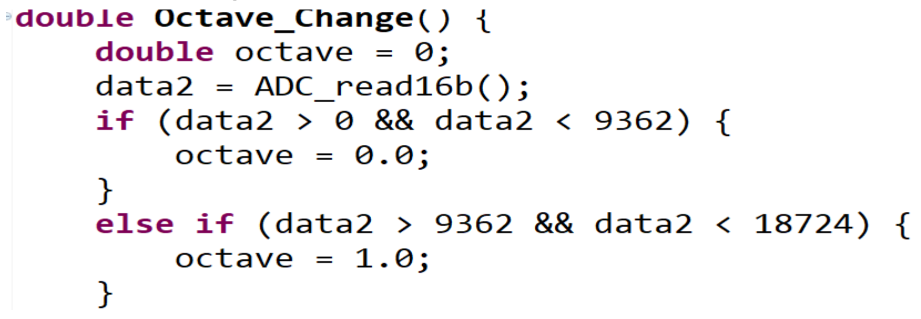
To obtain the ticks, the CLK system frequency is divided by the prescaler and divided by the frequency to get the value for FTM3_MOD, where you have to subtract 1.0 afterwards. To play the note, the number of ticks for the PWM is multiplied by the volume or the duty cycle determined by ADC. This plays the note until the button is released and displays the note on the 7 segment display using GPIOC_PCOR and GPIOC_PSOR and the characters in the arrays for the notes, sharps, and flats as shown below.



```
// white keys on piano
unsigned char notes[7] = {0x8E,0x3D,0x8F,0x87,0x9F,0xB7,0x1F}; // C, D, E, F, G, A, B

// black keys on piano (sharps and flats)
unsigned char sharps[5] = {0x8E,0x3D,0x87,0x9F,0xB7}; // C#, D#, F#, G#, A# (raised by half step on piano)
unsigned char flats[5] = {0x3D,0x8F,0x9F,0xB7,0x1F}; // Db, Eb, Gb, Ab, Bb (lowered by half step on piano)
```

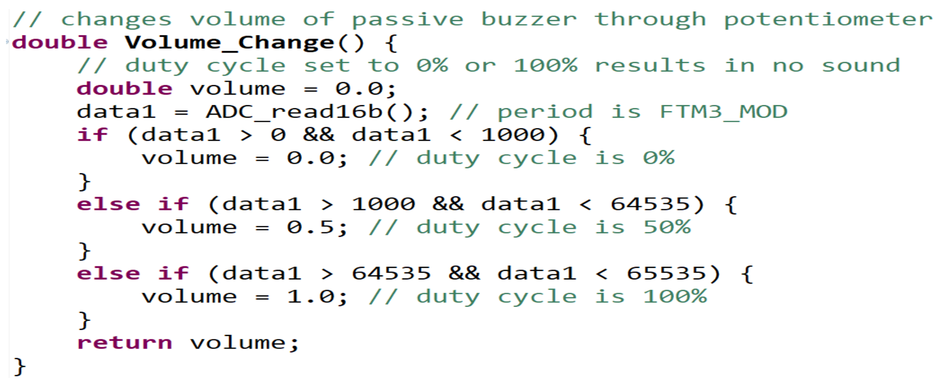
ADC Change for Volume and Octave

The data from the potentiometer is read as an ADC through the ADC_read16b() function that converts the data into a digital value between 0 to 65535. Due to this, the range 0 to 65535 is utilized in ADC to create ranges for each octave and volume.



```
double Octave_Change() {
    double octave = 0;
    data2 = ADC_read16b();
    if (data2 > 0 && data2 < 9362) {
        octave = 0.0;
    }
    else if (data2 > 9362 && data2 < 18724) {
        octave = 1.0;
    }
```

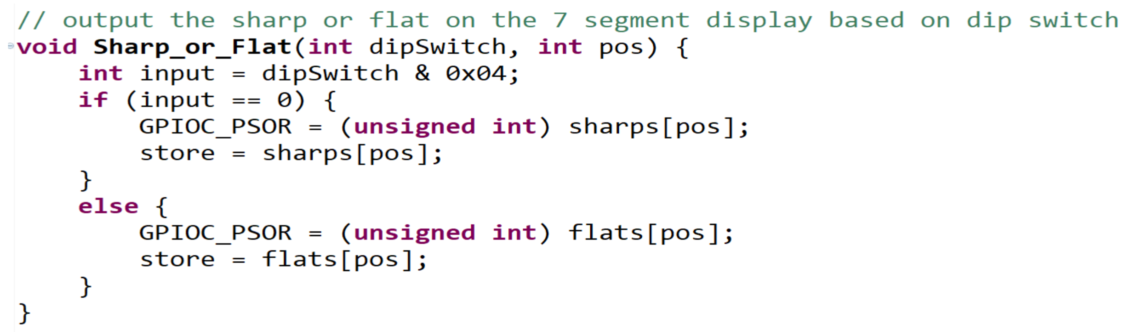
Volume of the passive buzzer is changed by changing the duty cycle. Duty cycles 10% - 90% all sound the same, and at 0% or 100% is when there is no volume. Thus, the ADC data is utilized to create a small range for both 0% and 100% as the beginning and end of the potentiometer range, where ranges in between would create a duty cycle of 50% to keep the passive buzzer sound on.



```
// changes volume of passive buzzer through potentiometer
double Volume_Change() {
    // duty cycle set to 0% or 100% results in no sound
    double volume = 0.0;
    data1 = ADC_read16b(); // period is FTM3_MOD
    if (data1 > 0 && data1 < 1000) {
        volume = 0.0; // duty cycle is 0%
    }
    else if (data1 > 1000 && data1 < 64535) {
        volume = 0.5; // duty cycle is 50%
    }
    else if (data1 > 64535 && data1 < 65535) {
        volume = 1.0; // duty cycle is 100%
    }
    return volume;
}
```

Sharps or Flats Feature

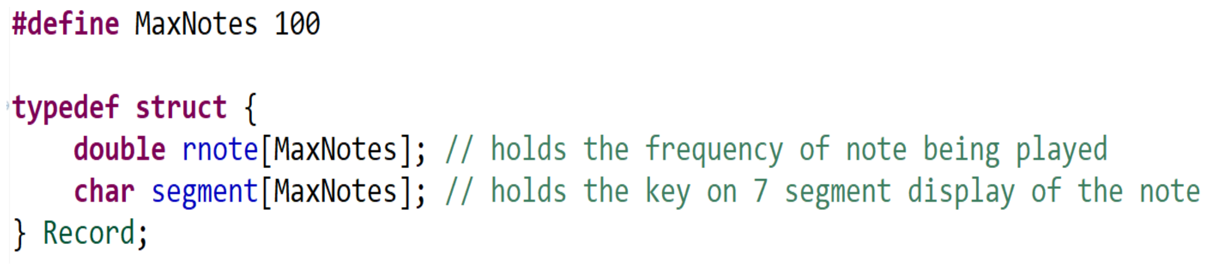
Dip switch 4 is checked for switching between either to display sharps or flats. A variable name “store” is created to store each character that represents each note being displayed on the 7 segment display. The function below determines if sharps or flats are played depending on the dip switch input.



```
// output the sharp or flat on the 7 segment display based on dip switch
void Sharp_or_Flat(int dipSwitch, int pos) {
    int input = dipSwitch & 0x04;
    if (input == 0) {
        GPIOC_PSOR = (unsigned int) sharps[pos];
        store = sharps[pos];
    }
    else {
        GPIOC_PSOR = (unsigned int) flats[pos];
        store = flats[pos];
    }
}
```

Recording Feature

Dip switch 2 is checked for recording. As explained before, the boolean variable “noteOn” checks if the note is being played or not and then when released, the polling checks that the boolean is now set to true and goes on to turn off the key on the release. It also checks if the dip switch 2 is set to 1 as well, which ultimately stores the frequency of the note being played as well as the note for the 7 segment display. During recording, information about the notes are stored in a struct that contains two arrays, one for the frequency of the note and another for the character. The max size of each recording is set to 100. Every recording would increment the “cnt” variable to traverse the struct arrays and update the “size” of the arrays. After all of this, the boolean “noteOn” is set back to false indicating no note is being played.



```
#define MaxNotes 100

typedef struct {
    double rnote[MaxNotes]; // holds the frequency of note being played
    char segment[MaxNotes]; // holds the key on 7 segment display of the note
} Record;
```

```
else if (noteOn) { // on release, turn off note and segment display; check dip switch for recording
    // clear the note and segment display
    FTM3_C6SC = 0x00;
    FTM3_C6V = 0x00;
    FTM3_SC = 0x00;
    GPIOC_PCOR = 0xBF;
    // check dip switch for recording; if yes, record current frequency and segment display in struct arrays
    if (dipSwitch & 0x08) {
        play.rnote[cnt] = freq;
        play.segment[cnt] = store;
        cnt++;
        size++;
    }
    noteOn = false; // flag set back to false for note playing detection
}
```

Playback Feature

Playback only happens when dip switch 1 is checked and set to 1. The boolean variable "stopPlay" is used for checking when playback is on and transition to clearing the notes when the replay is finished. If dip switch 1 is 1 and then stopPlay is false, then playback is performed through the for loop that replays and displays all the notes stored in the struct arrays. The notes being played are given a delay of 1000000 ticks to act as the length of the notes and a delay of 200000 ticks to act as the duration between each note, where it turns off after playing each note. After this, stopPlay is set to true which indicates that the playback is finished.

```
else if ((dipSwitch & 0x400)) { // play back the recording when dip switch is 1
    if (stopPlay == false) {
        for (int i = 0; i < size; i++) {
            if ((dipSwitch & 0x400)) {
                freq = play.rnote[i];
                FTM3_MOD = (int) (ticks/freq) - 1.0;
                FTM3_C6SC = 0x28;
                FTM3_C6V = (int) (ticks/freq) * volume;
                FTM3_SC = 0x0B;

                GPIOC_PCOR = 0xBF;
                GPIOC_PSOR = (unsigned int) play.segment[i];

                software_delay(1000000); // software delay added for length of the note played

                FTM3_C6SC = 0x00;
                FTM3_C6V = 0x00;
                FTM3_SC = 0x00;
                GPIOC_PCOR = 0xBF;

                software_delay(200000); // software delay added for duration between each note played
            }
        }
        stopPlay = true; // sets stopPlay to transition to clearing the notes
    }
```

When playback finishes, stopPlay is checked to be true and then it clears every element inside the struct arrays and initializes "size' and "cnt" to 0, where it also turns off the final note being played. The program remains in the playback mode with the keys disabled until dip switch 1 is set to 0, which initiates note playing again and resets stopPlay to false.

```
else if (stopPlay == true) { // if true, turn off final note and clear the struct arrays as well as counters
    FTM3_C6SC = 0x00;
    FTM3_C6V = 0x00;
    FTM3_SC = 0x00;
    for (int i = 0; i < size; i++) {
        play.rnote[i] = 0.0; // clearing the doubles inside rnote[]
        play.segment[i] = '\0'; // clearing the chars inside segment[]
    }
    size = 0;
    cnt = 0;
}
```

Testing/Evaluation:

Test environment and Required equipment

For testing, no specific equipment or environment was needed, as only a computer and a K64F microcontroller were required. The voltage going into the passive buzzer is around 3.3V, which is very low, so quiet conditions are needed to hear all the notes.

Test scenarios

In terms of test scenarios, different test cases were required for each functionality. For the white and black keys, there are 7 octaves that are implemented, so seven distinct test cases will be needed to check if each note is played in the correct octave. Two additional test cases will also be required to verify the ADC for 0% and 100% duty cycles, which deactivates the passive buzzer. Other test scenarios include the features. The first feature that displays the sharps or flats needs to be checked. Additionally, recording and playback should consist of test cases to verify that they can accurately record the notes and play them back. This includes if it can record notes being pressed for a longer duration, if it can record notes with long durations in between, and if note playing is disabled during playback phase.

Evaluation

Overall, our testing showed that the synthesizer met the main requirements. Each button played the correct note through PWM, and the pitch matched reference piano tones across all seven octaves. The potentiometer successfully adjusted both volume and octave. However, we noticed that the passive buzzer only produced an audible difference at 0% and 100% duty cycles—values in between sounded mostly the same. Due to this, volume change was adjusted to have the passive buzzer turned off only at 0% and 100% duty cycles. The 7-segment display also correctly showed the note being played, including sharps and flats depending on the dip switch setting. Recording and playback worked efficiently as well, where notes were recorded with the right frequency and played back with a slight delay between each played note, which we were very satisfied with. In terms of the recording, there was an issue where it would often record notes more than once. As we continued testing, this was discovered to be a hardware issue, where the program might detect notes more than once when the buttons are pressed and held for longer periods of time. Due to this being only a small human or hardware error, it was kept as a result. In general, the system handled everyday use well and even managed edge cases, such as pressing multiple buttons at once, without crashing. Ultimately, the system performed reliably and achieved what we set out to build. Testing helped us fine-tune the features and revealed areas where future improvements could further enhance the design.

Discussions:

Challenges

Challenges for this project include the dip switch, which was very buggy. There was an issue with the switches, where adjacent switches would not function properly. This issue was resolved by using switches 1, 2, and 4 instead of 1, 2, and 3. Another challenge was recording and playback. It was proven to be very difficult to have the switch as a determining factor for recording because this would create an interrupt service fault and crash the program. We were able to bypass this issue using boolean variables for checking.

Limitations

There were also a few limitations for this project. One limitation is that volume change can only be implemented by changing the duty cycle to either 0% or 100%, since volumes from the passive buzzer with duty cycles between 0% and 100% sound the same. Due to this, volume change was combined into one potentiometer for ADC along with octave change. Another limitation is the lack of GPIO pins, which require more configuration, and as a result, certain functionalities cannot be added. For example, with more pins, another display could be used to show the symbols for the sharps and flats. The last of our limitations was that our recording system did not take into account the durations that the notes were held down and the duration between when you pressed the notes in your desired recording. Implementing these may have been doable, but we decided that it was best to optimize the functionality and other features of our synthesizer more. From this, another limitation would also be to better understand all of the features on the K64F, such as how to use its other timers and ADC, which may help us improve features such as recording and playback and add more features as well.

Possible improvements

Possible improvements we can make include separating the volume change and octave change by using two different ADCs. Another possibility is to also adjust the volume by adding an additional working potentiometer to limit the voltage supply of the passive buzzer. Additionally, it would be beneficial to enhance the project by utilizing UART to display the notes and their corresponding frequencies; however, due to time constraints and compatibility issues with the ADC and GPIO, this was not implemented at this time.

In terms of recording and playback, additional improvements could be made by adding in the duration checking, which may be done by being able to utilize the other flex timer channels. By doing this, input capture could be utilized to capture the length of the PWM signal when the note is being played and capture that length of time as ticks. Another way could be to synchronize the program with a certain period such as using tasks and state machines.

Finally, one improvement would also be to utilize interrupts, which can interrupt the program and perform something in the interrupt service routine such as storing the notes into arrays. In addition, it would also be nice in the future to improve the project by implementing how to play chords or different notes played at the same time. This was not emphasized during the beginning of the project as we were both not musicians, but for the future, this would be a great addition to the project that will allow us to replicate a piano more accurately.

Roles and Responsibilities of Group Members

Group Member	Roles and Responsibilities
Maazin Zubair	1. Helped build the circuit 2. Made the flowchart and block diagram for

	the report 3. Helped find the frequencies for the notes and calculating values for PWM 4. Implemented the recording and playback features of the music 5. Helped write the report
Zerong Cai	1. Helped build the circuit 2. Initialized the GPIO pins, ADC, and timer 3. Built the schematic diagram for the circuit in PSpice 3. Implemented octave change by combining it with volume change for all the notes 4. Helped in calculating PWM and created the formula to play each note 5. Helped write the report

Brief Conclusion

Overall, this project gave us practical experience applying embedded systems concepts to build a real-world application. We were able to implement a functional keyboard synthesizer with features like note playback, octave and volume control, and recording. Despite some limitations, we adapted our design and successfully met the core objectives. The project helped strengthen our understanding of microcontroller programming and circuit integration, and highlighted the importance of testing, debugging, and flexibility in embedded system design.