```c
/* ========================================
 *
 * Copyright 6.115, 2024
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include <project.h>
#include "GUI.h"
#include "tft.h"
#include <stdlib.h> // for random

// Note Detection Counter
/* These variables are accessed in the ISRs
 * Accessed in ISR_Compare to update the counter value */
extern uint8 compare_occured;

/* The clock frequency for the PWM_Window. The PWM_Window clock frequency must be in KHz


/* Define for 1 second in terms of millisecond */
#define NO_OF_MSEC 1000
#define PWM_FREQ 100 //stay at 100kHz
uint32 input_freq = 0;

/* Variables to store the Period of the PWM_Window */
static uint16 PWM_windowPeriod = 0;

/* Variable to store the count value after capture */
static uint32 counter_countVal;

/* Set up the main project states */
enum projectState {
    WELCOME,        //S0
    LEARN,          //S1
    WORKSTATION,    //S2
    MELODIES,       //S3
    CHORDS,         //S7
    RIPTIDE,        //S4
    PERFECT,        //S5
    IRIS,           //S6
    CChord, DChord, EChord, AChord, GChord, DMChord, EMChord, AMChord, RANDOMCHORD, //S8 - S16
    RECORD,         //S17

            PLAY            //S19
};
```

```c
enum projectState holdStatePar = 0;
enum projectState currentState = 0;
enum projectState backStatePar = 0;

//Defining a dictionary to hold guitarString
struct KeyValuePair{
    const char *key;
    int value;
};

//Defining a dictionary to hold guitarString
struct KeyChordPair{
    const char *key;
    int value[3];
};

// Define all functions
void InitProject(void);
int Navigation(int menuBottom, enum projectState holdState, enum projectState backState);
void WelcomeMode(void);
void LearnMode(void);
void WorkstationMode(void);
void MelodiesMode(void);
void ChordsMode(void);
void RiptideMode();
void PerfectMode();
void IrisMode();
void GuitarTab(void);
enum projectState LoadMelodyNotes(int arr[][3], int melodySize);
int RecordMode();
void PlayMode();
int NoteDetectionMode(int actualFreq, int isNote);
int findValueByKey(const struct KeyValuePair dict[], int size, const char *key);
enum projectState LoadChordNotes(int numChords, int randomState, int chordChoice);
void CChordMode();
void DChordMode();
void EChordMode();
void AChordMode();
void GChordMode();
void DMChordMode();
void EMChordMode();
void AMChordMode();
int adcReading(int muxSelect);
int finalAdcResult;
int pickRandomInt(int min, int max);

// Global Variables
// ADC
```

```
uint16 adcResult = 0;
const char * adcFreqBuff[100];
const char * adcBuff[100];
const char * finalAdcResultBuff[100];

//DAW
unsigned char songPosition = 0;            // start LCD position and incremtn ↩
later
int songList[10000];   // initialize an empty display array

//Note Counter
uint16 notesLeft = 0;
const char * notesLeftBuff[20];
uint16 currentNote = 0;

// Location References
int backCol = 180;
int backRow = 20;
int pointerCol = 0;
int menuTop = 80;
int menuBottomPar = 0;
int tftCenterCol = 100;
int tftCenterRow = 130;

//Guitar Tabs String locations
int guitarStringStartCol = 10;
int guitarStringEndCol = 240;
int displayNoteOffset = 8;

//Int rows
int guitarStringOneRow = 80;
int guitarStringTwoRow = 120;
int guitarStringThreeRow = 160;
int guitarStringFourRow = 200;
int guitarStringFiveRow = 240;
int guitarStringSixRow = 280;

//Loading Melody and Chords vars
int currentFret;
int currentFreq;
int currentGuitarStringRow;
int currentGuitarStringCol;
const char * currentFretBuff[5];
int melodyNoteCounter;
int melodyGroupCounter;
int melodyGroupSize = 5; //Shows how many notes show up at a time on teh ↩
display
int currentMelodySizeIndex;

// Set up which guitar string ROW currently on
enum guitarStringCols {
    guitarStringColOne,
    guitarStringColTwo,
    guitarStringColThree,
    guitarStringColFour,
    guitarStringColFive,
};
enum guitarStringCols guitarStringCol = 0;

static const char * const guitarStringColsSTR[] = {
    [guitarStringColOne] = "guitarStringColOne",
    [guitarStringColTwo] = "guitarStringColTwo",
    [guitarStringColThree] = "guitarStringColThree",
    [guitarStringColFour] = "guitarStringColFour",
    [guitarStringColFive] = "guitarStringColFive"
};

// Set up which guitar string ROW currently on
enum guitarStringRows {
    IgnoreOffSetRow, // adding offset to correlated #/string
    guitarStringRowOne,
    guitarStringRowTwo,
    guitarStringRowThree,
    guitarStringRowFour,
    guitarStringRowFive,
    guitarStringRowSix,
};
enum guitarStringRows guitarStringRow = 0;
// Set up ENU   M TO STR

static const char * const guitarStringRowsSTR[] = {
    [guitarStringRowOne] = "guitarStringRowOne",
    [guitarStringRowTwo] = "guitarStringRowTwo",
    [guitarStringRowThree] = "guitarStringRowThree",
    [guitarStringRowFour] = "guitarStringRowFour",
    [guitarStringRowFive] = "guitarStringRowFive",
    [guitarStringRowSix] = "guitarStringRowSix",
};
struct KeyValuePair guitarStringDict[] = {
    {"guitarStringColOne", 34},
    {"guitarStringColTwo", 74},
    {"guitarStringColThree",120},
    {"guitarStringColFour", 170},
    {"guitarStringColFive", 210},
    {"guitarStringRowOne", 80},
    {"guitarStringRowTwo", 120},
    {"guitarStringRowThree", 160},
    {"guitarStringRowFour", 200},
    {"guitarStringRowFive", 240},
    {"guitarStringRowSix", 280}
};

int guitarStringDictSize = sizeof(guitarStringDict) / sizeof(guitarStringDict[↩
0]);
```

```c
// Notes in Riptide, {String#, Fret#, Freq Hz}
int RiptideNotes[155][3] = {
    {3, 2, 220},{2, 0, 247},{2, 1, 262},{2, 3, 294},{1, 0, 330},{1, 5, 440},{1, 3, 392},{2, 3, 294},{1, 0, 330}, // Chorus 9 0:20
    {3, 2, 220},{2, 0, 247},{2, 1, 262},{2, 3, 294},{1, 0, 330},{1, 5, 440},{1, 3, 392},{2, 3, 294},{1, 0, 330},
    {1, 3, 392},{1, 0, 330},{1, 3, 392},{1, 0, 330},{1, 3, 392},{2, 3, 294},{1, 0, 330},{1, 0, 330},{1, 0, 330},
    {1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 0, 330},{2, 3, 294},{1, 0, 330},
    {1, 0, 330},{2, 3, 294},{1, 0, 330},{2, 3, 294},{1, 0, 330},{2, 3, 294},{1, 0, 330},{2, 3, 294},{1, 0, 330},{2, 3, 294},
    {3, 2, 220},{2, 0, 247},{2, 0, 247},{2, 0, 247},{2, 0, 247},{2, 0, 247},{2, 0, 247},{2, 0, 247},{2, 0, 247},{2, 0, 247},{2, 1, 262},
    {3, 0, 196},{1, 5, 440},{1, 3, 392},{1, 0, 330},{3, 0, 196},{1, 5, 440},{1, 3, 392},{1, 0, 330},
    {1, 3, 392},{1, 0, 330},{1, 3, 392},{1, 3, 392},{1, 5, 440},{1, 3, 392},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 3, 392},
    {2, 3, 294},{2, 3, 294},{2, 3, 294},{2, 1, 262},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 3, 392},{1, 0, 330},{2, 3, 294},{1, 0, 330},
    {1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 3, 392},{2, 3, 294},{2, 3, 294},{2, 1, 262},{2, 1, 262},{2, 1, 262},{2, 1, 262},
    {2, 1, 262},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 3, 392},{2, 3, 294},{2, 3, 294},{2, 3, 294},{2, 1, 262},{1, 0, 330},
    {1, 0, 330},{1, 0, 330},{1, 3, 392},{1, 0, 330},{2, 3, 294},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 3, 392},
    {2, 3, 294},{2, 3, 294},{2, 1, 262}, {3, 2, 220},{2, 0, 247},{2, 1, 262},{2, 3, 294},{1, 0, 330},{1, 5, 440},{1, 3, 392},{2, 3, 294},//0:59
    {1, 0, 330},{3, 2, 220},{2, 0, 247},{2, 1, 262},{2, 3, 294},{1, 0, 330},{1, 5, 440},{1, 3, 392},{2, 3, 294},{1, 0, 330},
    {2, 3, 294},{1, 0, 330},{2, 3, 294},{1, 0, 330},{1, 3, 392},{2, 3, 294},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 0, 330},{1, 0, 330},
    {2, 3, 294},{1, 0, 330},{1, 0, 330},{2, 3, 294},{2, 3, 294} //1:23
};

// PERFECT {String#, Fret#, Freq Hz}
int PerfectNotes[140][3] = {
    //0:20 to 1:42
    {2, 1, 262},{3, 3, 233},{3, 1, 208},{2, 1, 262},{2, 4, 311},{2, 1, 262},{3, 3, 233},{3, 1, 208},{3, 1, 208},
    {3, 1, 208},{3, 3, 233},{2, 1, 262},{2, 2, 278},{2, 2, 278},{2, 1, 262},{3, 3, 233},{3, 3, 233},{3, 1, 208},{3, 1, 208},{3, 3, 233},
    {2, 1, 262},{3, 3, 233},{2, 4, 311},{2, 4, 311},{2, 4, 311},{1, 1, 349},{2, 1, 262},{3, 3, 233},{2, 1, 262},{2, 1, 262},{2, 1, 262},
    {2, 1, 262},{3, 3, 233},{3, 1, 208},{2, 1, 262}, //0:42
    {2, 1, 262},{2, 1, 262},{2, 1, 262},{3, 3, 233},{3, 1, 208},{2, 2, 278},{2, 1, 262},{3, 1, 208},{4, 1, 156},{2, 1, 262},{2, 2, 278},
    {2, 1, 262},{3, 3, 233},{2, 1, 262},{3, 3, 233},{3, 1, 208},{2, 1, 262},{2, 1, 262},{2, 1, 262},{2, 1, 262},{3, 3, 233},{3, 1, 208},
    {2, 1, 262},{2, 1, 262},{2, 1, 262},{2, 1, 262},{3, 3, 233},{3, 1, 208},{2, 2, 278},{2, 1, 262},{3, 1, 208},{4, 1, 156},
    {3, 3, 233},{3, 3, 233},{2, 4, 311},{2, 1, 262},{3, 1, 208},{1, 4, 415},{1, 3, 392},{1, 1, 349},{1, 3, 392},{2, 1, 262},
    {2, 4, 311},{2, 2, 278},{2, 1, 262},{3, 3, 233},{3, 1, 208},{1, 4, 415},{1, 3, 392},{1, 1, 349},{3, 1, 208},{2, 1, 262},
    {2, 4, 311},{2, 4, 311},{2, 4, 311},{1, 1, 349},{2, 1, 262},{3, 3, 233},{2, 1, 262},{2, 1, 262},{2, 4, 311},{1, 4, 415},{1, 3, 392},
    {1, 1, 349},{1, 3, 392},{2, 1, 262},{3, 1, 208},{3, 3, 233},{2, 1, 262},{2, 4, 311},{2, 2, 278},{2, 1, 262},{2, 2, 278},{2, 1, 262},
    {3, 3, 233},{2, 2, 278},{2, 1, 262},{3, 1, 208},{3, 3, 233},{2, 1, 262},{3, 3, 233},{3, 3, 233},{3, 1, 208},{3, 1, 208},
};

// IRIS {String#, Fret#, Freq Hz}
int IrisNotes[115][3] = {
    {4, 0, 147},{4, 0, 147},{3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 0, 196},{4, 4, 185},{4, 0, 147},{4, 2, 165},{4, 0, 147},
    {4, 0, 147},{3, 2, 220},{3, 0, 196},{4, 4, 185},{4, 4, 185},{4, 2, 165},{4, 0, 147},{4, 0, 147},
    {4, 0, 147},{4, 0, 147},{3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 0, 196},{4, 2, 165},{3, 2, 220},{3, 0, 196},{3, 0, 196},
    {4, 4, 185},{4, 4, 185},{4, 2, 165},{4, 0, 147},{4, 2, 165},{4, 4, 185},{4, 0, 147},{4, 2, 165},{4, 4, 185},{4, 0, 147},{4, 0, 147},
    {4, 0, 147},{4, 0, 147},{3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 0, 196},{4, 4, 185},{4, 0, 147},{4, 0, 147},{4, 0, 147},
    {3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 0, 196},{4, 4, 185},
    {4, 0, 147},{4, 0, 147},{3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 2, 220},{3, 0, 196},{4, 4, 185},{4, 0, 147},{4, 2, 165},
    {4, 0, 147},{4, 2, 165},{4, 2, 165},{4, 4, 185},{4, 0, 147},{4, 2, 165},{4, 4, 185},{4, 0, 147},{4, 0, 147},
    {2, 3, 294},{2, 3, 294},{1, 5, 440},{1, 5, 440},{1, 5, 440},{1, 5, 440},{1, 3, 392},{1, 2, 370},{2, 3, 294},{1, 0, 330},
    {2, 3, 294},{1, 0, 330},{1, 2, 370},{1, 0, 330},{2, 3, 294},{1, 0, 330},{1, 2, 370},{2, 3, 294},{2, 3, 294},{2, 3, 294},
    {1, 5, 440},{1, 5, 440},{1, 5, 440},{1, 5, 440},{1, 3, 392},{1, 2, 370},{2, 3, 294},{1, 0, 330},{2, 3, 294},{1, 0, 330},
    {1, 2, 370},{1, 0, 330},{2, 3, 294},{1, 2, 370},{2, 3, 294},{2, 3, 294},
};

// Define a structure for chord notes
struct Chord {
    int notes[6][2];  // Array to hold up to 6 strings (1 to 6) and their respective frets
    int numStrings;   // Number of strings used in the chord (e.g., 5 for standard guitar)
    int freqs[3]; // shows the frequencies that make up the notes
    char * name;
};

// Define chords using the Chord structure
struct Chord chordsDict[] = {
    // Chord C
    {
```

```
    .notes = {
        {1, 0},  // String 1 (highest E) - Open
        {2, 1},  // String 2 (B) - 1st fret
        {3, 0},  // String 3 (G) - Open
        {4, 2},  // String 4 (D) - 2nd fret
        {5, 3}   // String 5 (A) - 3rd fret
    },
    .numStrings = 5,
    .freqs = {261, 329, 196},
    .name = "C Chord"
},
// Chord D
{
    .notes = {
        {3, 2},  // String 3 (G) - 2nd fret
        {4, 3},  // String 4 (D) - 3rd fret
        {5, 2}   // String 5 (A) - 2nd fret
    },
    .numStrings = 3,
    .freqs = {293, 369, 440},
    .name = "D Chord"
},

// Chord E
{
    .notes = {
        {1, 0},  // String 1 (highest E) - Open
        {2, 0},  // String 2 (B) - Open
        {3, 1},  // String 3 (G) - 1st fret
        {4, 2},  // String 4 (D) - 2nd fret
        {5, 2},  // String 5 (A) - 2nd fret
        {6, 0}   // String 6 (lowest E) - Open
    },
    .numStrings = 6,
    .freqs = {164, 207, 246},
    .name = "E Chord"
},
// Chord A
{
    .notes = {
        {1, 0},  // String 1 (highest E) - Open
        {2, 2},  // String 2 (B) - 2nd fret
        {3, 2},  // String 3 (G) - 2nd fret
        {4, 2},  // String 4 (D) - 2nd fret
        {5, 0}   // String 5 (A) - Open
    },
    .numStrings = 5,
    .freqs = {220, 277, 165},  // Optional frequencies for each note in Hz
    .name = "A Chord"
},
// Chord G
{
        .notes = {
            {1, 3},  // String 1 (highest E) - 3rd fret
            {2, 0},  // String 2 (B) - Open
            {3, 0},  // String 3 (G) - Open
            {4, 0},  // String 4 (D) - Open
            {5, 2},  // String 5 (A) - 2nd fret
            {6, 3}   // String 6 (lowest E) - 3rd fret
        },
        .numStrings = 6,
        .freqs = {392, 494, 293},  // Optional frequencies for each note in Hz
        .name = "G Chord"
    },
    // Chord Dm
    {
        .notes = {
            {2, 3},  // String 2 (B) - 3rd fret
            {3, 2},  // String 3 (G) - 2nd fret
            {4, 0}   // String 4 (D) - Open
        },
        .numStrings = 3,
        .freqs = {293, 369, 220},  // Example frequencies
        .name = "DM Chord"
    },
    // Chord Em
    {
        .notes = {
            {1, 0},  // String 1 (highest E) - Open
            {2, 2},  // String 2 (B) - 2nd fret
            {3, 2},  // String 3 (G) - 2nd fret
            {4, 0},  // String 4 (D) - Open
            {5, 0},  // String 5 (A) - Open
            {6, 0}   // String 6 (lowest E) - Open
        },
        .numStrings = 6,
        .freqs = {329, 415, 247},  // Example frequencies
        .name = "EM Chord"
    },
    // Chord Am
    {
        .notes = {
            {1, 0},  // String 1 (highest E) - Open
            {2, 1},  // String 2 (B) - 2nd fret
            {3, 2},  // String 3 (G) - 2nd fret
            {4, 2},  // String 4 (D) - 2nd fret
            {5, 0}   // String 5 (A) - Open
        },
        .numStrings = 5,
        .freqs = {220, 277, 165},  // Example frequencies
        .name = "AM Chord"
    }
};
```

```c
int main(){
    InitProject();                        // Call all Initializations
    /* Calculate the time window during which the counter will count */
    PWM_windowPeriod = PWM_1_ReadPeriod() ;
    /* Update the Time window value according to the clock given to the PWM_Window */
    PWM_windowPeriod = PWM_windowPeriod/ PWM_FREQ;

    for(;;){
        // Reset Variables Before Each State
        switch(currentState) {
            case WELCOME:     // Load Welcome Streen
                WelcomeMode();
                menuBottomPar = 100;
                holdStatePar = LEARN;
                backStatePar = WELCOME;
                currentState = Navigation(menuBottomPar, holdStatePar, backStatePar);
                break;
            case LEARN:                   // Load Learning Screen
                LearnMode();
                menuBottomPar = 100;
                holdStatePar = MELODIES;
                backStatePar = WELCOME;
                currentState = Navigation(menuBottomPar, holdStatePar, backStatePar);
                break;
            case MELODIES:                   // Load Learning Screen
                MelodiesMode();
                menuBottomPar = 120;
                holdStatePar = RIPTIDE;
                backStatePar = LEARN;
                currentState = Navigation(menuBottomPar, holdStatePar, backStatePar);
                break;
            case RIPTIDE:
                RiptideMode();
                currentState = LoadMelodyNotes(RiptideNotes, 155); //calling note generation for song
                break;
            case PERFECT:
                PerfectMode();
                currentState = LoadMelodyNotes(RiptideNotes, 155); //calling note generation for song
                break;
            case IRIS:
                IrisMode();
                currentState = LoadMelodyNotes(RiptideNotes, 155); //calling note generation for song
                break;
            case CHORDS:
                ChordsMode();
                menuBottomPar = 240;
                holdStatePar = CChord;
                backStatePar = LEARN;
                currentState = Navigation(menuBottomPar, holdStatePar, backStatePar);
                break;
            case CChord:
                CChordMode();
                currentState = LoadChordNotes(20, 0, 0);
                break;
            case DChord:
                DChordMode();
                currentState = LoadChordNotes(20, 0, 1);
                break;
            case EChord:
                EChordMode();
                currentState = LoadChordNotes(20,0, 2);
                break;
            case AChord:
                AChordMode();
                currentState = LoadChordNotes(20,0, 3);
                break;
            case GChord:
                GChordMode();
                currentState = LoadChordNotes(20,0, 4);
                break;
            case DMChord:
                DMChordMode();
                currentState = LoadChordNotes(20, 0, 5);
                break;
            case EMChord:
                EMChordMode();
                currentState = LoadChordNotes(20, 0, 6);
                break;
            case AMChord:
                AMChordMode();
                currentState = LoadChordNotes(20, 0, 7);
                break;
            case RANDOMCHORD:
                GuitarTab();
                GUI_SetFont(&GUI_Font20B_ASCII);
                GUI_DispStringAt("Current: ", 20, 40);
                currentState = LoadChordNotes(50, 1, 0); // making chordchoice = 0, doesn't matter
                break;
            case WORKSTATION:
                WorkstationMode();
                menuBottomPar = 100;
                holdStatePar = RECORD;
                backStatePar = WELCOME;
                currentState = Navigation(menuBottomPar, holdStatePar, backStatePar);
                break;
            case RECORD:
                RecordMode();
                menuBottomPar = 100;
                holdStatePar = RECORD;
                backStatePar = WORKSTATION;
                currentState = Navigation(menuBottomPar, holdStatePar, ↩
backStatePar);
                break;
            case PLAY:
                PlayMode();
                menuBottomPar = 100;
                holdStatePar = RECORD;
                backStatePar = WORKSTATION;
                currentState = Navigation(menuBottomPar, holdStatePar, ↩
backStatePar);
                break;
            // run this code if no cases are matched, which should never happen
            default:
                break;
        }
    }
}

void InitProject(){
    CyGlobalIntEnable;                      // Enable global interrupts
    SPIM_1_Start();                         // initialize SPIM component
    GUI_Init();                             // initilize graphics library
    GUI_Clear();
    ADC_SOUND_IN_Start();               // strt the ADC_SOUND_IN
    AMux_1_Start();                     // Start the MUX
    DAC_SOUND_OUT_Start();  // starts DAC and calls enable()
    
    // Note Detection Counter
    ISR_Compare_Start();
    PWM_1_Start();
    Counter_1_Start();
    Clock_PWM_Start();
    Clock_1_Start();
    
}

int Navigation(int menuBottom, enum projectState holdState, enum projectState ↩
backState){
    //make the navigation less repeitioned
    int currentPointerRow = 80; // For determining where menu items should be
    int prevPointerRow = 80;
    while(BTN_SEL_Read() != 1){
        // STATE CHANGES
        CyDelay(100);
        if((BTN_UP_Read() == 1) && (currentPointerRow != menuTop)) ↩
{           // Set State to learn
```

```c
            holdState = holdState - 1;
            currentPointerRow -= 20;
            GUI_DispStringAt("  ", pointerCol, prevPointerRow);
            GUI_DispStringAt(">", pointerCol, currentPointerRow);
            prevPointerRow = currentPointerRow;
        }
        if((BTN_DOWN_Read() == 1) && (currentPointerRow != menuBottom)){ //Set State to workstation
            holdState = holdState + 1;
            currentPointerRow += 20;
            GUI_DispStringAt("  ", pointerCol, prevPointerRow);
            GUI_DispStringAt(">", pointerCol, currentPointerRow);
            prevPointerRow = currentPointerRow;
        }
        if(BTN_BACK_Read() == 1){ //Set State to workstation
            holdState = backState;
            break;
        }
    }
    return holdState;
}

// Function to find a value in dict
int findValueByKey(const struct KeyValuePair dict[], int size, const char *key) {
    int i;
    for (i = 0; i < size; i++) {
        if (strcmp(dict[i].key, key) == 0) {
            return dict[i].value;
        }
    }
    return -1; // Return -1 if key is not found (assuming all values are positive)
}

// Function to pick a random item from an array
int pickRandomInt(int min, int max){ // pick a random chord(int min, int max)
    // Generate a random index within the bounds of the array
    int randomIndex = rand() % (max - min + 1) + min;
    // Return the randomly selected item
    return randomIndex;
}

void WelcomeMode(){
// CURRENT STATE CHANGES
    GUI_Clear();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("WELCOME :D!", 20, 20);
    GUI_DispStringAt("GUITAR LEARNER", 20, 40);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt("LEARN MODE", 20, 80);
    GUI_DispStringAt("WORKSTATION MODE", 20, 100);
    GUI_DispStringAt(">", pointerCol, menuTop);
}
void LearnMode(){
    // Option between melodies or chords
    GUI_Clear();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("LEARN MODE", 20, 40);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt("BACK", backCol, backRow);
    GUI_DispStringAt("MELODIES", 20, 80);
    GUI_DispStringAt("CHORDS", 20, 100);
    GUI_DispStringAt(">", pointerCol, menuTop);
}

void MelodiesMode(){
    // Option between melodies
    GUI_Clear();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("MELODIES", 20, 40);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt("BACK", backCol, backRow);
    GUI_DispStringAt("RIPTIDE", 20, 80);
    GUI_DispStringAt("PERFECT", 20, 100);
    GUI_DispStringAt("IRIS", 20, 120);
    GUI_DispStringAt(">", pointerCol, menuTop);
}

void GuitarTab(){
    GUI_Clear();
    GUI_SetFont(GUI_FONT_D48);
    // COUNT DOWN FOR USERS
    GUI_DispStringAt("3", tftCenterCol, tftCenterRow);
    CyDelay(500);
    GUI_DispStringAt("2", tftCenterCol, tftCenterRow);
    CyDelay(500);
    GUI_DispStringAt("1", tftCenterCol, tftCenterRow);
    CyDelay(500);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16_ASCII);
    // DRAWING EACH STRING OF THE GUITAR WITH LABELS
    GUI_DispStringAt("BACK", backCol, backRow);
    GUI_DispStringAt("E", 0, guitarStringOneRow - displayNoteOffset);
    GUI_DrawLine(guitarStringStartCol, guitarStringOneRow, guitarStringEndCol↩
, guitarStringOneRow);
    GUI_DispStringAt("B", 0, guitarStringTwoRow- displayNoteOffset);
    GUI_DrawLine(guitarStringStartCol, guitarStringTwoRow, guitarStringEndCol↩
, guitarStringTwoRow);
    GUI_DispStringAt("G", 0, guitarStringThreeRow- displayNoteOffset);
    GUI_DrawLine(guitarStringStartCol, guitarStringThreeRow, ↩
guitarStringEndCol, guitarStringThreeRow);
    GUI_DispStringAt("D", 0, guitarStringFourRow- displayNoteOffset);
    GUI_DrawLine(guitarStringStartCol, guitarStringFourRow, guitarStringEndCol↩
, guitarStringFourRow);
    GUI_DispStringAt("A", 0, guitarStringFiveRow- displayNoteOffset);
    GUI_DrawLine(guitarStringStartCol, guitarStringFiveRow, guitarStringEndCol↩
, guitarStringFiveRow);
    GUI_DispStringAt("E", 0, guitarStringSixRow- displayNoteOffset);
    GUI_DrawLine(guitarStringStartCol, guitarStringSixRow, guitarStringEndCol↩
, guitarStringSixRow);
    LED_YELLOW_Write(1);
}

enum projectState LoadMelodyNotes(int melodyNotes[][3], int melodySize){
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt("Notes Left: ", 20, backRow);
    int numMelodyGroups = melodySize / melodyGroupSize;// How many melody ↩
groups can be created at that size
    // Iterating through each of the groups, and then each note in the group
    for(melodyGroupCounter = 0; melodyGroupCounter < numMelodyGroups; ↩
melodyGroupCounter++){
        // Resetting the col number
        for(melodyNoteCounter = 0; melodyNoteCounter < melodyGroupSize; ↩
melodyNoteCounter++){
            //Making sure we are not out of the bounds of the list
            currentMelodySizeIndex =  (melodyGroupCounter * melodyGroupSize) ↩
+ melodyNoteCounter;
            if(currentMelodySizeIndex < melodySize) {
                guitarStringRow = melodyNotes[currentMelodySizeIndex][0]; // ↩
Current String
                currentFret = melodyNotes[currentMelodySizeIndex][1]; // ↩
Current Fret
                // Finding Row -> take ENUM int -> ENUM str -> Dict -> value ↩
of ROW
                currentGuitarStringRow = findValueByKey(guitarStringDict, ↩
guitarStringDictSize, guitarStringRowsSTR[guitarStringRow])- displayNoteOffset;
                // Get teh current col value, from the noteCounter# ->
                currentGuitarStringCol = findValueByKey(guitarStringDict, ↩
guitarStringDictSize, guitarStringColsSTR[melodyNoteCounter]);
                sprintf(currentFretBuff, "%d", currentFret); // Noties left ↩
calculation
                GUI_DispStringAt(currentFretBuff, currentGuitarStringCol, ↩
currentGuitarStringRow);
            }
        }
        //REPEATING THE LOOP BUT NOW THAT THE NOTES ARE SHOWN, NOW USER PLAYS
        for(melodyNoteCounter = 0; melodyNoteCounter < melodyGroupSize; ↩
melodyNoteCounter++){
            //Making sure we are not out of the bounds of the list
            currentMelodySizeIndex =  (melodyGroupCounter * melodyGroupSize) ↩
+ melodyNoteCounter;
            if(currentMelodySizeIndex < melodySize) {
                guitarStringRow = melodyNotes[currentMelodySizeIndex][0]; // ↩
Current String
```

```
                currentFret = melodyNotes[currentMelodySizeIndex][1]; // Current Fret
                currentFreq = melodyNotes[currentMelodySizeIndex][2]; // Current Freq
                // REDEFINE CURRENT ROW AND COL COORDINATES
                currentGuitarStringRow = findValueByKey(guitarStringDict, guitarStringDictSize, guitarStringRowsSTR[guitarStringRow])- displayNoteOffset;
                currentGuitarStringCol = findValueByKey(guitarStringDict, guitarStringDictSize, guitarStringColsSTR[melodyNoteCounter]);
                // DRAW RECT AROUND ACTIVE NOTE COL
                GUI_DrawRect(currentGuitarStringCol-8, currentGuitarStringRow-8,currentGuitarStringCol+16, currentGuitarStringRow+16);
                // Check if user plays correct note OR hits back key
                while(NoteDetectionMode(currentFreq, 1) == 0){
                    if(BTN_BACK_Read() == 1){ //Set State to Melodies if back is read
                        currentState = MELODIES;
                        return currentState;
                    }
                };
                // RETURN STATUS LEDS
                LED_YELLOW_Write(1); //Turn yellow back on for next note
                LED_GREEN_Write(0);
                LED_RED_Write(0);
                // PUT LINE BACK WHERE THE NOTE WAS PLAYED
                GUI_ClearRect(currentGuitarStringCol-8, currentGuitarStringRow-8,currentGuitarStringCol+16, currentGuitarStringRow+16);
                GUI_DrawLine(currentGuitarStringCol-8, currentGuitarStringRow + displayNoteOffset, currentGuitarStringCol+16, currentGuitarStringRow + displayNoteOffset);
                // // Notes left calculationClear the area from before + 8 offset
                GUI_ClearRect(tftCenterCol+10, backRow, tftCenterCol+48, backRow+16); // delete three 8x16 digits
                notesLeft = (melodySize - 1) - currentMelodySizeIndex; //Substract one for offset
                sprintf(notesLeftBuff, "%d", notesLeft);
                GUI_DispStringAt(notesLeftBuff, tftCenterCol+10, backRow);
                CyDelay(100); // DELETE LATER
            }
        }
    }
    //LOAD CONGRATULATIONS!
    GUI_Clear();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("Well Done! You Rock!", 20, tftCenterRow-20);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt("Click SEL or BACK to return", 20, tftCenterRow+20);
    GUI_DispStringAt("to Melody Selection Screen", 20, tftCenterRow+40);
    currentState = Navigation(100, MELODIES, MELODIES);
    return currentState;
}
```

```
int adcReading(int muxSelect){
    // GET NOTE PLAYED FROM ADC
    AMux_1_Select(muxSelect); // Select the Sound Input
    ADC_SOUND_IN_StartConvert();        // start the ADC_SOUND_IN conversion
    if(ADC_SOUND_IN_IsEndConversion(ADC_SOUND_IN_WAIT_FOR_RESULT)){ // Makes ↩
sure that conversion is done{
        adcResult = ADC_SOUND_IN_GetResult32();     // read the adc and ↩
assign the value adcResult
        if (adcResult & 0x8000)
        {
            adcResult = 0;      // ignore negative ADC results
        }
    }
    return adcResult;
}

int NoteDetectionMode(int actualFreq, int isNote){
    //GUI_ClearRect(tftCenterCol, 300-8,tftCenterCol+48, 300+16);
    //adcResult = adcReading(0);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    sprintf(adcFreqBuff, "%d", actualFreq);
    GUI_DispStringAt(adcFreqBuff, tftCenterCol-90, 300); // Display actual Freq
    GUI_ClearRect(tftCenterCol+50, 300, tftCenterCol+200, 316);
    if(isNote == 1){ // Detecting a Note, not a chord
        // Counter Stuff
        if (compare_occured == 1)
        {
            /* Read the Counter capture register */
            counter_countVal = Counter_1_ReadCapture();
            /* Convert the counts to frequency.
             * Frequency is the number of counts in seconds
             * In this case counts within "PWM time window" (100 millisecond ↩
in this example) is got from Counter.
             * So we need to find for 1000 milliseconds */
            int micResults = adcReading(0); //see what the mic gives
            sprintf(adcBuff, "%d", micResults);
            GUI_DispStringAt(adcBuff, tftCenterCol-40, 300);
            // Counter Results
            adcResult = ((uint32)(NO_OF_MSEC * (uint32)counter_countVal) / (↩
uint32)PWM_windowPeriod);
            // parse final result
            finalAdcResult = LO16(adcResult);
            sprintf(finalAdcResultBuff, "%d", finalAdcResult);
            GUI_DispStringAt(finalAdcResultBuff, tftCenterCol+50, 300);
            CyDelay(100);
            /* Clear the interrupt flag */
            compare_occured = 0;
        }
        // Margin of error
        int margin_error = 5;
        // Calculate the absolute difference between x and y
```

```c
        int diff = (finalAdcResult > actualFreq) ? (finalAdcResult - actualFreq) : (actualFreq - finalAdcResult);
        if (diff <= margin_error) {
            LED_GREEN_Write(1);
            LED_YELLOW_Write(0);
            LED_RED_Write(0);
            CyDelay(10);
            return 1; // take 1 as TRUE
        } else {
            LED_GREEN_Write(0);
            LED_YELLOW_Write(0);
            LED_RED_Write(1);
            return 0; // Take 0 as FALSE
        }

    }
    else{ // If detecting a chord
        if (CHORD_TEST_Read() == 0) {
            LED_GREEN_Write(1);
            LED_YELLOW_Write(0);
            LED_RED_Write(0);
            CyDelay(50);
            return 1; // take 1 as TRUE
        } else {
            LED_GREEN_Write(0);
            LED_YELLOW_Write(0);
            LED_RED_Write(1);
            return 0; // Take 0 as FALSE
        }
    }
}

void RiptideMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("RIPTIDE", 20, 40); // Title
}

void PerfectMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("PERFECT", 20, 40);
}

void IrisMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("IRIS", 20, 40);
}

void ChordsMode(){
    // Option between melodies
    GUI_Clear();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("CHORDS", 20, 40);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt("BACK", backCol, backRow);
    GUI_DispStringAt("C", 20, 80);
    GUI_DispStringAt("D", 20, 100);
    GUI_DispStringAt("E", 20, 120);
    GUI_DispStringAt("A", 20, 140);
    GUI_DispStringAt("G", 20, 160);
    GUI_DispStringAt("DM", 20, 180);
    GUI_DispStringAt("EM", 20, 200);
    GUI_DispStringAt("AM", 20, 220);
    GUI_DispStringAt("RANDOM", 20, 240);
    GUI_DispStringAt(">", pointerCol, menuTop);
}

//Loading Chores
enum projectState LoadChordNotes(int numChords, int randomState, int chordChoice){
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt("Chords Left: ", 20, backRow);
    int numChord;
    int chordNoteIndex;
    int randomChoice;
    int chordNotesSize;
    int chordNote[6][2];
    char * currentChordName;
    if(randomState == 0){
        chordNotesSize = chordsDict[chordChoice].numStrings; // knowing how many notes to iterate through
        memcpy(chordNote, chordsDict[chordChoice].notes, sizeof(chordNote)); // Copy notes into chordNote

    }
    // Iterate through number of chords to play
    for(numChord = 0; numChord < numChords; numChord++){
        // If not random state, assign proper chord
        if(randomState == 1){
            randomChoice = pickRandomInt(0, 7); // pick a random chord
            // Filling in the data with randomchoice int
            chordNotesSize = chordsDict[randomChoice].numStrings; // knowing how many notes to iterate through
            memcpy(chordNote, chordsDict[randomChoice].notes, sizeof(chordNote)); // Copy notes into chordNote
            currentChordName = chordsDict[randomChoice].name;
            GUI_ClearRect(tftCenterCol+10, 40, tftCenterCol+100, 40);
            GUI_SetFont(&GUI_Font20B_ASCII);
            GUI_DispStringAt(currentChordName, tftCenterCol+10, 40);
            GUI_SetFont(&GUI_Font8x16_ASCII);
        }
        // Iterate through each note making up the chord
        for(chordNoteIndex = 0; chordNoteIndex < chordNotesSize; ↩
chordNoteIndex++){
            //DISPLAY: Iterate through the fret and string number for each ↩
note
            guitarStringRow = chordNote[chordNoteIndex][0]; // Current String
            currentFret = chordNote[chordNoteIndex][1]; // Current Fret
            // Finding Row -> take ENUM int -> ENUM str -> Dict -> value of ROW
            currentGuitarStringRow = findValueByKey(guitarStringDict, ↩
guitarStringDictSize, guitarStringRowsSTR[guitarStringRow])- displayNoteOffset;
            // Get teh current col value, from the noteCounter# ->
            currentGuitarStringCol = 120; // center column
            sprintf(currentFretBuff, "%d", currentFret); // Noties left ↩
calculation
            GUI_DispStringAt(currentFretBuff, currentGuitarStringCol, ↩
currentGuitarStringRow);
        }
        // Check if user plays correct note OR hits back key
        while(NoteDetectionMode(currentFreq, 0) == 0){
            if(BTN_BACK_Read() == 1){ //Set State to Melodies if back is read
                currentState = CHORDS;
                return currentState;
            }
        };
        // RETURN STATUS LEDS
        LED_YELLOW_Write(1); //Turn yellow back on for next note
        LED_GREEN_Write(0);
        LED_RED_Write(0);
        // PUT LINE BACK WHERE THE NOTE WAS PLAYED
        for(chordNoteIndex = 0; chordNoteIndex < chordNotesSize; ↩
chordNoteIndex++){
            //DISPLAY: Iterate through the fret and string number for each ↩
note
            guitarStringRow = chordNote[chordNoteIndex][0]; // Current String
            currentFret = chordNote[chordNoteIndex][1]; // Current Fret
            // Finding Row -> take ENUM int -> ENUM str -> Dict -> value of ROW
            currentGuitarStringRow = findValueByKey(guitarStringDict, ↩
guitarStringDictSize, guitarStringRowsSTR[guitarStringRow])- displayNoteOffset;
            // Get teh current col value, from the noteCounter# ->
            currentGuitarStringCol = 120;
            GUI_ClearRect(currentGuitarStringCol-8, currentGuitarStringRow-8,↩
currentGuitarStringCol+16, currentGuitarStringRow+16);
            GUI_DrawLine(currentGuitarStringCol-8, currentGuitarStringRow + ↩
displayNoteOffset, currentGuitarStringCol+16, currentGuitarStringRow + ↩
displayNoteOffset);
        }
        // Notes left calculationClear the area from before + 8 offset
        GUI_ClearRect(tftCenterCol+10, backRow, tftCenterCol+48, backRow+16); ↩
// delete three 8x16 digits
        notesLeft = (numChords - 1) - numChord; //Substract one for offset
        sprintf(notesLeftBuff, "%d", notesLeft);
        GUI_DispStringAt(notesLeftBuff, tftCenterCol+20, backRow);
        CyDelay(100); // DELETE LATER
```

```c
    }
    //LOAD CONGRATULATIONS!
    GUI_Clear();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("Well Done! You Rock!", 20, tftCenterRow-20);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt("Click SEL or BACK to return", 20, tftCenterRow+20);
    GUI_DispStringAt("to Melody Selection Screen", 20, tftCenterRow+40);
    currentState = Navigation(220, CHORDS, CHORDS);
    return currentState;
}

void CChordMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("C CHORD", 20, 40);
}
void DChordMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("D CHORD", 20, 40);
}
void EChordMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("E CHORD", 20, 40);
}
void AChordMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("A CHORD", 20, 40);
}
void GChordMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("G CHORD", 20, 40);
}
void DMChordMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("DM CHORD", 20, 40);
}
void EMChordMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("EM CHORD", 20, 40);
}
void AMChordMode(){
    GuitarTab();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("AM CHORD", 20, 40);
}
```

```
void WorkstationMode(){
    // Option between melodies or chords
    GUI_Clear();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("WORKSTATION MODE", 20, 40);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt("BACK", backCol, backRow);
    GUI_DispStringAt("RECORD", 20, 80);
    GUI_DispStringAt(">", pointerCol, menuTop);
    //GUI_DrawCircle(tftCenterCol, tftCenterRow, 40);
}
int RecordMode(){
    GUI_Clear();
    GUI_SetFont(GUI_FONT_D48);
    GUI_DispStringAt("3", tftCenterCol, tftCenterRow);
    CyDelay(500);
    GUI_DispStringAt("2", tftCenterCol, tftCenterRow);
    CyDelay(500);
    GUI_DispStringAt("1", tftCenterCol, tftCenterRow);
    CyDelay(500);
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_Clear();
    GUI_DispStringAt("RECORDING", 20, 80);
    int songListSize = sizeof(songList) / sizeof(songList[0]);
    int songIndex;
    // Iterate through length of songList to append
    // Iterate through length of songList to append
    for(songIndex = 0; songIndex < songListSize; songIndex++){
        adcResult = adcReading(0); // TAkes readings
        songList[songIndex] = adcResult;
    }
    // Option to Record New or Play
    GUI_SetFont(&GUI_Font8x16_ASCII);
    GUI_DispStringAt(">", pointerCol, menuTop);
    GUI_DispStringAt("BACK", backCol, backRow);
    GUI_DispStringAt("RECORD NEW", 20, 80);
    GUI_DispStringAt("PLAY", 20, 100);
}

void PlayMode(){
    GUI_Clear();
    GUI_SetFont(&GUI_Font20B_ASCII);
    GUI_DispStringAt("PLAYING", 20, 40);
    // Select the PITCH Input
    int pitchResult = adcReading(1);
    // Select the REVERB Input
    int reverbResult = adcReading(2);
    finalAdcResult = adcResult;
    int songListSize = sizeof(songList) / sizeof(songList[0]);
    int songIndex;
    for(songIndex = 0; songIndex < songListSize; songIndex++){
        DAC_SOUND_OUT_SetValue(songList[songIndex]);
        CyDelay(1);
    }
    GUI_Clear();
    GUI_DispStringAt(">", pointerCol, menuTop);
    GUI_DispStringAt("BACK", backCol, backRow);
    GUI_DispStringAt("RECORD NEW", 20, 80);
    GUI_DispStringAt("PLAY", 20, 100);
}

/* [] END OF FILE */
```