TuneStudio2560

8-Bit Song Creation & Playback Device

v1.0.0

By Jacob Luvisi
September 2021

https://github.com/devjluvisi/TuneStudio2560

Table of Contents

Subsections described with "." (ex. 9.1).
Children of Subsections have a 13pt font compared to 14pt. (ex. 7.3.0)

Project Introduction

First and foremost, TuneStudio2560 is an 8-Bit song playback and creation device created for the Arduino Mega 2560 microcontroller. Designed from the ground up, the device was made to execute the flawless creation of simple 8-bit linear songs via user interaction. The project was first idealized around June of 2021 but the first working version would not be released until August 19, 2021. With the use of external libraries, months of work, and careful planning TuneStudio2560 was able to make it to its current state without any major bugs or issues.

Along with the creation of songs, TuneStudio2560 has also been designed to playback such songs to the user. Songs which the user creates on their device are saved to a microSD card where they can be accessed and edited for future use. The usage of a microSD card allows much more flexibility in the storage of songs compared to on board storage such as an EEPROM (1-4KB in size). When a user wishes to play a song back they can use TuneStudio2560's custom inbuilt media player which allows the pausing, rewinding and fast forwarding of songs. The media player also displays helpful information about the song on various time intervals to the user as well as a progress bar to show the user how far they have progressed into the song.

The creation of songs on TuneStudio2560 is simple and easy to understand for new users or people familiar with musical concepts. The device uses the standard chromatic scale for applying notes to a song. Yes, that means pitches such as GS2 *(G#2)* or F1 are used when creating songs, making the process much easier to understand then a complex proprietary method. When a song is played back, the application converts the pitches (which have been saved in persistent storage such as a microSD) to frequencies which can be played by the on board passive buzzer.

In order to display information to the user, three methods of communication are used. First, a Liquid Crystal Display is used for visually showing the user textual information related to the current context of the program. Examples of usage for the LCD include: creating a song, where the LCD displays the individual pitches that the user has added to the song, as well as when playing back a song, where the LCD shows the user information such as the progress bar. The LCD also provides the user with direct instructions and controls about how to create and playback songs. Second, TuneStudio2560 uses a 4 digit wide 7-Segment display for communication. The segment display is used when the user is creating a song, where it displays the current pitch the user has selected. This allows the user to know what pitch they are about to add to their song. Third, the device utilizes an RGB LED. The variable brightness controlled RGB LED lights up different colors depending on a variety of factors. The RGB LED provides the simplest way for the user to get information about something important without even having to read a display.

When a user has saved their song to the SD card, they can edit the song later on their personal computer or on MakerStudio2560. MakerStudio2560 is an interactive JavaScript client-side song creation and playback tool similar to TuneStudio2560. MakerStudio2560 is completely backwards compatible with the file format of TuneStudio2560, which means that any songs you create in TuneStudio2560 can be edited in MakerStudio2560 and vice versa. If MakerStudio2560 is not used, then the user can open the individual song files of the SD card directly on their computer thanks to the usage of cross-platform FAT16/FAT32 file formatting. Each song is reserved to its own file and the individual song files use the **.TXT** file extension meaning they can be easily opened on modern computers.

TuneStudio2560 is a device with a diverse set of utilities to allow for the creation of simple but catchy songs. All of this, on a mere microcontroller.

Crash Course on Arduino Mega 2560

(and compatible)

As you have probably read at this point, TuneStudio2560 utilizes the Arduino Mega 2560 as its primary source of function. **Why?** The Arduino Mega 2560 provides a great balance between performance, low power consumption, and a large flexible program space. Due to the use of the Mega, along with a few external and publicly available libraries, TuneStudio2560 is accessible to many hobbyists and enthusiasts. TuneStudio2560 utilizes all *open-source code only libraries*, meaning that you can create the full featured TuneStudio2560 for free right now (as long as you have the hardware that is). If it was not for the Arduino ecosystem, then such a diverse set of libraries would not be available and the project would be less accessible to everyone.

Running on an 8-Bit AVR Microarchitecture, TuneStudio2560 is efficient enough to run on battery for an extended period of time. In case you forgot your facts about the Arduino Mega 2560, here is some information about how this microcontroller powers both the software and external hardware which makes up TuneStudio2560.

Core Specifications:[1]

- CPU: 16 MHz 8-BIT AVR
- 8192 Bytes of SRAM
- 256KB of Flash Memory
- 4096 Bytes of EEPROM
- Power Consumption: 70-200mA

Price: $16-$18 (Clone), $39.95 MSRP

Notable Features:

- I2C Capability
- SPI Capability
- 54 Digital IO Pins (15 PWM)

[1] View Source

- Analog Input/Output Pins

The entirety of TuneStudio2560 runs on a 16MHz processor and (as of v1.2.2-R4) utilizes as little as only 1300 Bytes of RAM[2]. All in all, TuneStudio2560 does not even make use of all of the Arduino Mega 2560's diverse feature set. A possible port to future microcontrollers such as the Arduino Uno is possible due to continued code shrinkage and optimization. Even though TuneStudio2560 is currently only verified working on AVR architecture, ports to ARM microcontrollers[3] are currently in their pre-design process but are not confirmed.

The Arduino Mega 2560 provides one of the most user-friendly experiences of any microcontroller on the market and its diverse range of features is one of the major reasons why TuneStudio2560 exists today.

Check out an Arduino Mega 2560 for yourself.

Some Arduino Mega 2560's come bundled with other electronics in a kit, similar to this one.

[2] TuneStudio2560 is estimated to consume about 1300 Bytes of SRAM at compile time (according to PlatformIO). Maximum RAM consumption has been measured to be about 1.7KB-1.8KB during program run time.
[3] Current ports to ARM microcontrollers are not confirmed. The current microcontroller ports which are being considered are ports to the RP2040 (Raspberry Pi Pico) and the Teensy 4.1.

Libraries & Software Used

TuneStudio2560 takes advantage of the Arduino's wide open source library community. There are two types of libraries used in TuneStudio2560: **internal libraries**, which are directly included in the GitHub and integrated into the code, and **external libraries** which must be downloaded by the end user in order to compile the program. All external libraries used are available on the PlatformIO library marketplace.

The following is a direct list of all internal and external libraries which are used in TuneStudio2560.

Internal

- digitalWriteFast
- NewTone

External

- LiquidCrystal_I2C
- SdFat
- SevSegShift
- Unity[4]

Software

- Visual Studio Code
- PlatformIO Extension

Technologies

- C++11 GNU
- Arduino Framework
- Git
- Doxygen

[4] Unity is a unit testing library for C which can be ingrained in PlatformIO. Unity is not required for compiling the program.

TuneStudio 2560 Design Philosophy

Now that we have been introduced to TuneStudio2560 and its basic functions, let us dive deeper into the mindset and design patterns which were planned when designing this project, as well as the overall functionality of the device.

TuneStudio2560 is both a creation of hardware and software, and such components must be able to interact with each other to achieve the goals of the device and provide a fluent user experience. TuneStudio2560 is constructed on top of two breadboards (see images). In general, there must be a way for the user to receive feedback, so an LCD is connected to a display which provides text feedback. However, the user must also be able to physically communicate with the device. To achieve this, two components are used, standard switch buttons and a 10K potentiometer.

The switch buttons allow the user to directly interact with the device and perform different actions depending on what state the program is in. The switch buttons serve two main purposes. The switch button's first purpose is to help with navigation. Two switch buttons on the device known as the “ADD/SELECT” and “DEL/CANCEL” buttons are used to navigate from the home screen to the various “modes” of the program. Different modes allow the user to perform different tasks *(for example, one mode would be to create a song and another would be to play it back)*. Second, the switch buttons are used to create songs and perform additional actions. The device has eight (8) switch buttons in total. These eight buttons each have different functionality in various modes (see Figure 1). These eight buttons are each sanctioned into two separate groups, “control” and “musical”. The “control” group buttons are used for navigation and user actions while the “musical” group buttons are used for creation of music and sometimes additional actions. Figure 1 displays what buttons have which names and where they are located on the breadboard. A brief description of each of the buttons is written below.

TuneStudio2560 is an interactive application, meaning that users can perform a variety of tasks on it. In the code, we must create a separation of these different tasks or “states” a user can be in. For example, one state could be to create a new song, another could be a state to view instructions for how to playback songs, etc. What is known as a **Program State** defines the separation of different functionality throughout the program. There are 5 program states in TuneStudio2560. Main Menu, Listening Mode Instructions, Creator Mode Instructions, Listening Mode, Creator Mode. **Creator Mode** represents the “state” where the user can create their own songs, play them back, and save them. **Listening Mode** represents a “state” where a user playback songs, pauses them, rewinds, fast forwards, and deletes them. Listening Mode plays songs which were previously made in Creator Mode.

The hardware for playing back notes and songs is simple. A piezo buzzer is used to generate a square wave and play back a specific frequency in hertz. Songs are constructed in TuneStudio2560 as a “list of hertz”. When a song is first created it has no hertz in it, but as notes get added, a linear list of hertz to play grows. In order to make the process of creating songs easier, users do not have to deal with the raw hertz values (ex. 31, 158). Instead, every possible frequency which can be played in TuneStudio2560 is represented in its chromatic form via a human readable pitch. Instead of adding the frequency “**233**” to a song, a user can add the pitch “**AS3**” (A#3) instead. This allows users to remember previously learned standardized pitches when making a song, instead of utilizing a proprietary standard. Because the piezo buzzer cannot read regular human strings and letters the pitch must be converted back to a frequency when the song is played. The program can automatically convert these pitches into frequencies and the frequencies back into pitches using a lookup table of values.

TuneStudio2560 would not be fun if everytime you restart the application all your songs are gone. To circumvent this, TuneStudio2560 takes advantage of its SPI Bus and uses a microSD card module for non-volatile storage. This microSD card stores every song in TuneStudio2560 that the user saves. As mentioned in the previous paragraph, all of the notes are saved in pitches (ex. AS3) instead of frequencies. When the file is selected for playback, the program converts these pitches to frequencies. Each different song is stored as a different document using the ".txt" file extension, making editing easy on a PC.

Control Group[5]

- ADD/SELECT -> Responsible for both navigation as well as the confirmation of prompts. In general, this button can be thought of as the "enter" button on a standard QWERTY keyboard.
- DEL/CANCEL -> Used similar to "ADD/SELECT" button but usually performs the opposite action as it. This button is also used for navigation but is usually used to reject prompts or to remove notes from a song/delete files.

Musical Group[6]

- Tune Button 1
    - Green LED
- Tune Button 2
    - Blue LED
- Tune Button 3
    - Red LED
- Tune Button 4

[5] Check out the wiki for a list of full actions that all of these buttons can do for each program state.
[6] Note that the language of "Tune Button" is also interchangeable with "Tone Button". Sometimes these words are used to define the same object.

    - Yellow LED
- Tune Button 5
    - White LED

The various buttons in the musical group are most commonly used for adding notes to a song when creating one. Another purpose for these buttons is to act as additional control buttons in certain states[7].

[7] Check out user guide for a full list of actions that each button can perform in different states.

Hardware

This section is a "parent section" meaning that it has one or many "child" sub-sections. Each of these subsections is defined by having a decimal number as shown in the Table of Contents.

6.0: Introduction

Now that the design of TuneStudio2560 as well as some basic functionalities have been covered, it is finally time to get into what makes TuneStudio2560 work. Schematics to the construction of TuneStudio2560 are provided in the figures at the end of this section. The following list describes the individual parts of TuneStudio2560.

Parts List:

The parts list below is also provided on GitHub.
For a detailed description of connections please visit the "Build It" section on the Wiki.
Last Updated: 9/11/2021

- Arduino Mega 2560 (or compatible)
- 2x Full Size Breadboards
- 1x Passive Piezo Buzzer
- 5x Colored LEDs (GBYRW)
- 1x RGB Led
- 8x 220Ω Resistors
- 4x 330Ω Resistors
- 1x 10K Potentiometer (or similar)
- 8x Standard Switching Buttons
- 1x 20x4 Liquid Crystal Display w/ I2C Protocol
- 1x microSD Card Module
- 2x SN74HC595N Shift Registers (or compatible)
- 1x 4-Digit 7-Segment Display (Common-Cathode)
- Wires

The above list of parts represents everything that makes TuneStudio2560 function. The following subsections will dive into the individual parts specifications and functionality within the program.

6.1: Arduino Microcontroller

As discussed previously in the “Crash Course on Arduino Mega 2560” the Arduino microcontroller plays the biggest role in TuneStudio2560’s functionality, so much so that it is in the name. The specific microcontroller which was used for the first prototype release for this project is an Elegoo Arduino Mega 2560 Rev. 3. Official versions of the Mega 2560 can be used and bought directly from the Arduino Store but clones can also work as well.

The Mega 2560 provides a large SRAM space to store the various types of data and data structures in TuneStudio2560. Although initially required, continued optimizations throughout releases have allowed TuneStudio2560’s working RAM to be less than 1.8KB at any point during the program cycle. This means that future releases to Arduino Uno boards are possible but are currently not confirmed. The program space of the Mega 2560 was also a helpful utility as 256KB of space was a large area to store both the code itself as well as a handful of PROGMEM variables. This too, has been released with versions and the latest release consumes only 29KB of program space[8].

TuneStudio2560 uses a significant amount of program memory to store PROGMEM constants. Such PROGMEM constants (as will be discussed later in this document) consume a significant amount of space at the benefit of reduced static SRAM usage.

[8] Release v1.2.2-R4 consumes approx. 29974 bytes of program space when the PRGM_MODE macro is set to “0” (small size priority).

6.2: Speaker

TuneStudio2560 utilizes a passive piezo speaker to play frequencies. This piezo speaker takes numerical 16-bit unsigned integers in the code and converts them to a sound by vibrating internal components of the speaker. The piezo speaker generates a square wave and, in TuneStudio2560, can range between 31Hz (B0 note) and 3957Hz (B7 Note). The speaker can be connected to a digital pin or an analog pin, a PWM connection is not required for the speaker to operate normally.

The speaker has two primary limitations. First, the speaker cannot be volume adjusted and will play at the same "volume" all the time. Sometimes the speaker may produce "quieter" sounds but this only occurs due to varying tones, not a volume reduction. Second, only one speaker can be connected to the Arduino at any moment. This makes overlapping sounds or stereo sounds impossible (limiting the musical capability to a linear 8-bit song).

The speaker is operated using the **NewTone** internal library. Although the NewTone library will be discussed in greater detail further down the product brief, it replaces the traditional **tone()** function when playing sounds.

6.3: Liquid Crystal Display

In order to communicate with the user, a Liquid Crystal Display (LCD) display is used to provide the primary form of communication (text). The LCD used in the project has **20 columns and 4 rows**. A taller than usual LCD was picked as it is best for displaying large amounts of text on the screen at once and is also much less cramped when navigating menus. Theoretically, a smaller 16x2 LCD (the most common) could be used by adjusting the LCD_ROWS and LCD_COLS macros in tune_studio.h but there would also need to be some software changes to accommodate this.

The display provided uses an I2C protocol to communicate with the Arduino. The I2C protocol is preferred as it uses less wires then a parallel setup and since the I2C bus is not benign utilized by any other device it is optimal to simplify the wiring to the Arduino as well. The display is controlled through the **LiquidCrystal_I2C** library (at a low level) and has custom program methods written for it to display text and scrolling text. Such methods are available in the software documentation.

6.4: 4 Digit 7 Segment Display

A 4-digit 7-segment common cathode display is connected to the Arduino to provide additional information to the user separate from the lcd, specifically in regards to creating songs. When a song is being created, the LCD becomes cluttered with many of the notes the user has added. In order to show the user what the current note they have selected is without showing it on the LCD, a segment display is used.

The segment display is connected directly to 2x SN74HC595N shift registers and controlled through the SevSegShift library. This is done in order to reduce the amount of pins required to connect to the Arduino (only 3 pins needed for the two shift registers). The two shift registers are connected in parallel and control various parts of the segment display.

6.5: SD Card Module

A microSD card module is used in TuneStudio2560 to interface with the microcontroller and to maintain persistent storage of songs. The microSD card module requires that SD cards be formatted as FAT16/FAT32 in order to function properly. The module is wired to the Arduino via the SPI Bus for fast transfer speeds. The **SdFat** library provides high level control over the SD card and by default is clocked at a *1MHz transfer rate* for maximum compatibility among SD cards and SD modules. Prior to v1.2.1-R3 a standard “SD” library was used which was inefficient and took up a large amount of SRAM and program space.

A microSD card is used on the Arduino in order to provide a large space to store collections of songs. In contrast, usage of the Arduino’s internal EEPROM would result in a very limited space and would also wear out the EEPROM over time. SD cards also provide a convenient way to connect directly to a PC, which they support.

Because the microSD card uses FAT32 formatting, it can be transferred directly from the Arduino onto a PC. The PC can read each of the individual files on the SD card and even edit them. Editing files on a PC will still allow them to playback on the Arduino as long as the proper format of the file was maintained. TuneStudio2560 ignores directories when reading the SD card and only focuses for files on the root “/” directory. Each song the user creates is stored as an individual file on the SD card using a .TXT extension for wider compatibility. The file saved must have a name of no longer than 8 characters and must follow the standard format for TuneStudio2560 songs.
A-Z, 0-9 and underscores only.

If a song file on the SD card is bad, then the Arduino will blink its status LED red and tell the user there has been an error. The Arduino will not crash and will recover from the error and return back to the song selection screen. By default, TuneStudio2560 ignores all files on the SD card with invalid file names or extensions (such as .pdf for example).

When the SD card is first loaded, TuneStudio2560 will create a “README.TXT” file on the root directory which includes directions about how to use the microSD card and how to edit song files. If the “PRGM_MODE” macro of the program is set to “0” (low program size) then only a part of the README will be generated.

6.6: RGB Led

The RGB status LED is the simplest form of user communication on TuneStudio2560. The RGB LED will flash different colors to signal various signs to the user. There are very few colors the RGB led will flash and only three primary signals should be known.

1. The RGB status LED will blink **GREEN** when the program initializes. The LED will blink **GREEN** "X" number of times depending on the current PRGM_MODE.
   a. 1 Blink = PRGM_MODE == 0
   b. 2 Blink = PRGM_MODE == 1
   c. 3 Blink = PRGM_MODE == 2
2. The RGB status LED will blink or stay **RED** when the program encounters an error.
   a. SD Card Initialization Error (SD Card not found at startup)
   b. Attempt to read a bad song file.
   c. Any other potential error.
3. The RGB status LED will blink or stay **BLUE** when the user has selected the OPTION button and the program is awaiting further input. The OPTION button (specifically in creating a song) signals that the user wishes to perform an additional action by pressing the select, cancel, or a tune button. When BLUE the program is telling the user that the next input they press will perform an action with the *option characteristics*[9], if that button has any. This state can be cancelled by pressing the option button again.

[9] As described in the design philosophy section, "optional characteristics" refer to when pressing a button will perform a different action than usual because the OPTION button was clicked. This idea can be thought of as pressing the CAPS LOCK button on a keyboard or the CTRL button.

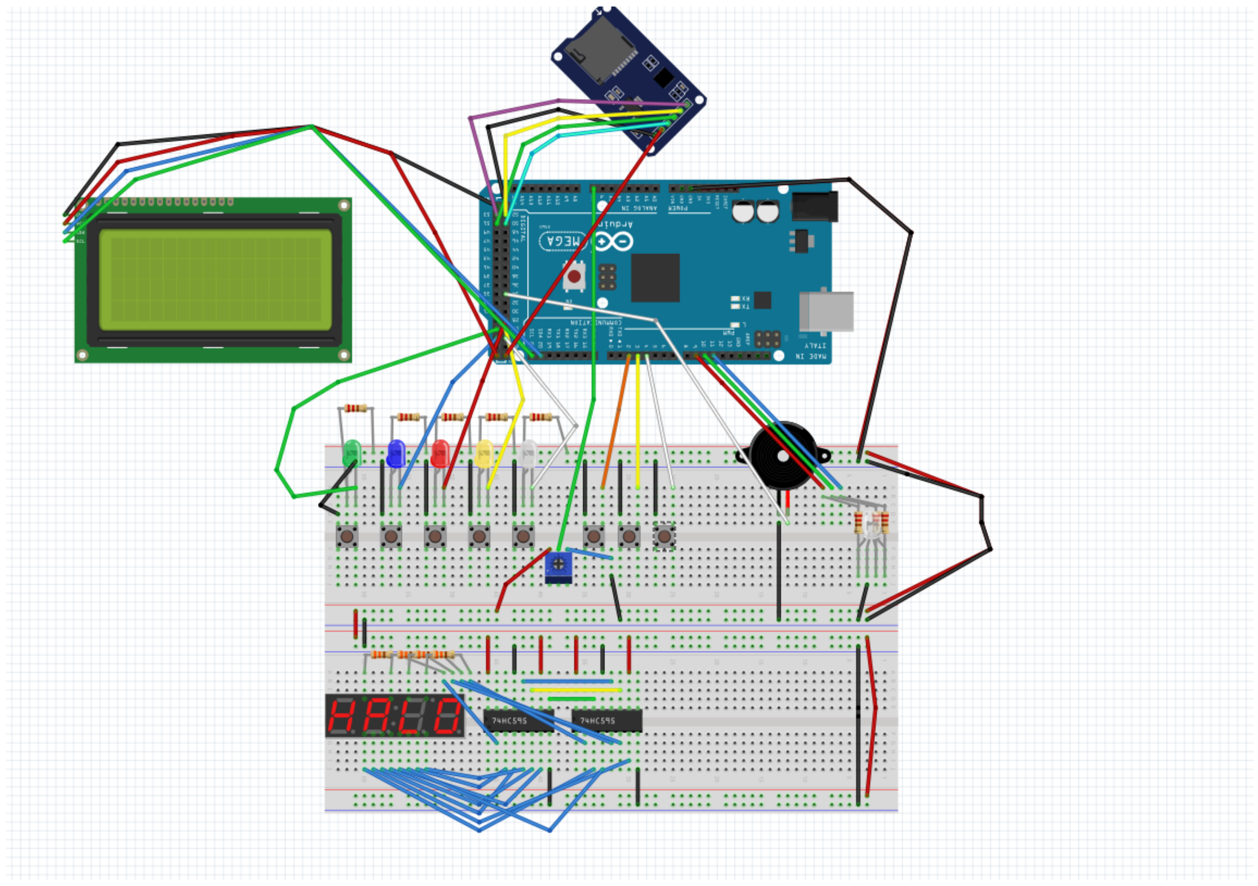
6.7 Extraneous Parts

Some parts of TuneStudio2560 go under “extraneous parts” as they are usually not significant enough to get a dedicated subsection. A notable “extraneous part” includes the standard switch button as previously talked about. The switch buttons are each connected one side to ground and the other side is connected to a pin on the Arduino. This means that whenever a button is pressed it will signal “LOW” to the Arduino instead of “HIGH”. The 5 tune buttons on the Arduino are each electrically wired to various colored LEDs. These LED’s light up whenever the tune button is pressed regardless of the current state or context. These LEDs are not controlled by code, only through the circuit. Each of these LED’s have their anode connected to 5V constantly via a **220Ω** resistor. The cathode is connected to one side of the tune button which is not connected to ground. Whenever the tune button gets pressed, it bridges the row of the breadboard to ground therefore allowing electricity to pass through the LED.

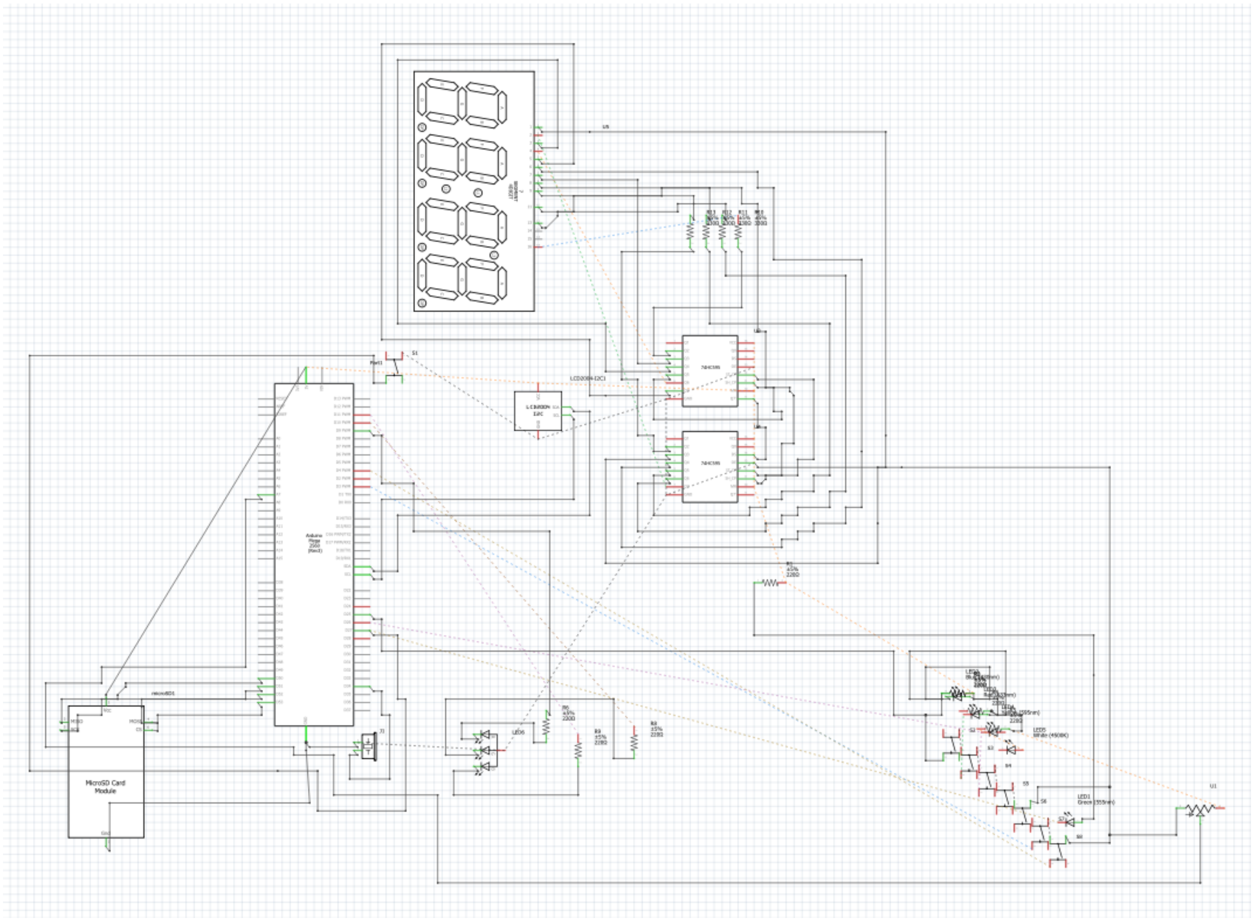
6.8 Diagrams:

There are two provided diagrams (excluding real life operational pictures):

Visual Diagram



Electrical Schematic
WORK IN PROGRESS


S1
LCD2004-I2C1
74HC595
74HC595
Arduino Mega 2560 (Rev3)
J1
LED6
R6 ±5% 220Ω
R9 ±5% 220Ω
R8 ±5% 220Ω
MicroSD Card Module
LED5 White (4500K)
S2
S3
S4
S5
S6
S7
S8
LED1 Green (555nm)
U1

Software

This section is a "parent section" meaning that it has one or many "child" sub-sections. Each of these subsections is defined by having a decimal number as shown in the Table of Contents.

7.0: Introduction

The software portion of TuneStudio2560 is the largest and most significant part of the device and therefore requires the most documentation to understand. The "Pre-Design" subsection will describe the thought process prior to developing software as well as the tools used. The following subsections will be very in depth descriptions of the classes, their functionalities, and how they work together. It should be noted prior to reading that all methods, classes, global variables, and files used in the program are thoroughly documented using Doxygen as well as inline comments. It is highly recommended to view the code comments or doxygen-generated website when reading through this section. The software that works with TuneStudio2560 is always in active development and being changed, it is also important to keep up with current and latest commits on the GitHub for up to date information.

7.1: Pre-Design

When developing TuneStudio2560, the decision was made to use Visual Studio Code and the PlatformIO extension instead of using the standard Arduino IDE. This decision was made because in order to keep the program modular, a variety of files and classes would be constructed. The Arduino IDE, as of now, is very limited in its ability to handle multiple files in one project. The usage of PlatformIO also allows developers to use direct AVR libraries as well as standard C++ functions which are not available to users on the normal Arduino IDE.

TuneStudio2560 is developed using C++ directly, throughout TuneStudio2560's codebase many C++ standard methods are used such as clases, inheritance, templates, and heap allocation[10]. Such methods allow the program to be easier to understand on a higher level by abstracting away some of the harder concepts in exchange for just a class (like a Song).

When developing TuneStudio2560, the idea that the code base should be sectioned off in separate parts proved to be very important. The solution was to divide the program based on different "program states". Each program state acts like its own little program and will act as though the other states do not exist. Each program state handles user input differently, displays differently to the lcd, and handles just about everything different. The main class (main.cpp) is still run and holds many of the utility classes that many of the program states need. When a user navigates to different areas in the program (ex. Going from Main Menu to a song creation) they are really changing a global program state variable. The loop() method on the Arduino constantly runs the current global program state variable's loop method to perform the actions of whatever the program state wants to achieve.

Songs are the most obvious area of TuneStudio2560 which had long thought put into them. In the current release as of this paper, songs are controlled through one global "Song" variable. This song variable can be cleared, have notes added to it, played back, etc. Every time the program has finished using a song, the song variable is simply cleared of its data. Every song variable holds an array (list) of integers which represent various frequencies (in Hz). Whenever a song gets played back, the program iterates through the array and plays each of its frequencies on the speaker. Every song also has its own delays (toneDelay and toneLength) which

[10] Although dynamic heap allocation is used in TuneStudio2560, its usage is very limited. Prior to v1.2.0-R3, Song objects were dynamically created. Although usually fine, it would cause occasional heap fragmentation and on smaller microcontrollers, the SD card would no longer function properly. Currently, dynamic allocation is only used for the ProgramState.

describe how long the delay between each note should be *(toneDelay)* and how long each note should play on the speaker *(toneLength)*.

When a user finishes creating the song we must save it to a more persistent form of data storage such as an SD card. When a user saves a song to an SD card, the program really goes through the current global song variables list of frequencies and converts them to human readable pitches (ex. FS2). These human readable pitches are then stored in the song file and when the user wishes to play the song file again the program converts these human readable pitches into numbers again by using a lookup table.

Similar design patterns are found throughout TuneStudio2560. The general rule is to remember that the main.cpp class contains the first setup() and loop() methods as well as utility methods for each class. Then each ProgramState represents a sectioned off piece of code (only one ProgramState runs at any time) which has its own way of dealing with user input, etc.

One major problem of storing the large pitch-frequency look up table is that it requires a large amount of SRAM to store. As a result, all of the strings in the lookup table are stored in Program Space (**PROGMEM**) which allows the device to act as though the look up table is stored in RAM when it is really stored in Flash. A **pitches.h** file stores the PROGMEM representation of every human readable pitch which is a part of the lookup table.

Brief Description of important files in the program directory:
(There are other files besides the ones listed here, but these are the most important files to take notes of…)

- src/ *All of the main .cpp code files.*
- lib/ *Internal libraries.*
- include/ *Header files.*

- src/studio-libs/states - *Where all of the individual ProgramStates are stored.*
- *"main.cpp" - Where the program starts, includes utility methods for each ProgramState.*
- *"state.cpp" - Includes the main ProgramState parent class that all of the ProgramStates inherit from.*
- *"song.cpp" - A representation of the song object.*
- *"tune_studio.h" - A header file which contains all of the important constant data and PROGMEM data. Shared by every class. Defines methods for the main class.*
- *"pitches.h" - A header file which contains nothing but every string representation of a human readable pitch (in PROGMEM)(85 total).*

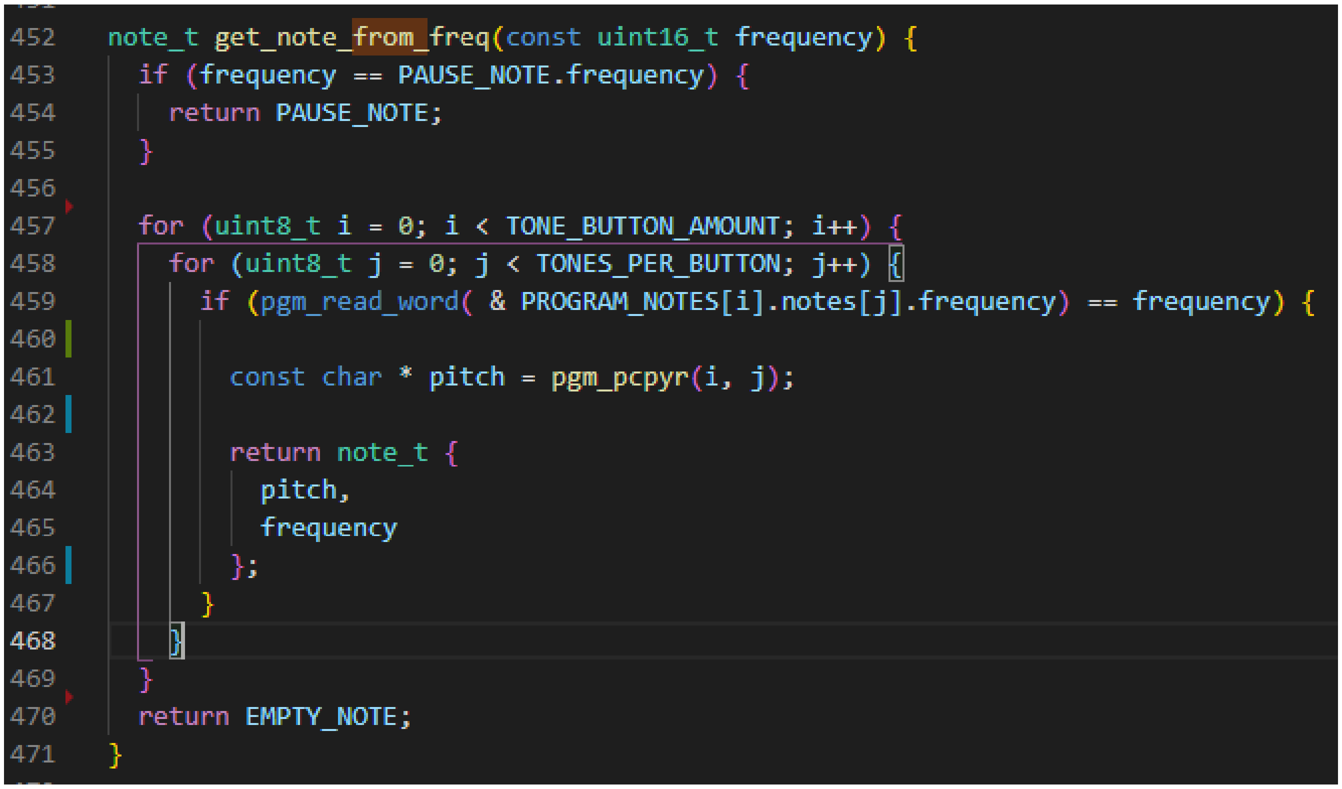
7.2: Performance & Program Size

7.2.0 Prelude

Measuring performance and program size is an important aspect of a project on a microcontroller. Ensuring that code is optimized to the limit of a programmer's knowledge is essential for ensuring that code works fast, efficiently, and fits within the size limitations. Luckily, TuneStudio2560 is a very efficient program and fits even below the Arduino Uno's requirements. This subsection will describe the importance of performance optimizations, program size, how to adjust program sizes, program modes, as well as methods to improve performance.

7.2.1 Optimizations

Many optimizations have been made throughout TuneStudio2560's first release. Although there have been many code size and speed improvements there are still many potential optimizations in TuneStudio2560. For one, there are two methods to improve the speed of the conversion of human readable pitches to 16-bit frequencies and vice versa. The following displays the current code of these methods. (Comments and macro checks removed to simplify)



```
note_t get_note_from_freq(const uint16_t frequency) {
  if (frequency == PAUSE_NOTE.frequency) {
    return PAUSE_NOTE;
  }

  for (uint8_t i = 0; i < TONE_BUTTON_AMOUNT; i++) {
    for (uint8_t j = 0; j < TONES_PER_BUTTON; j++) {
      if (pgm_read_word( & PROGRAM_NOTES[i].notes[j].frequency) == frequency) {

        const char * pitch = pgm_pcpyr(i, j);

        return note_t {
          pitch,
          frequency
        };
      }
    }
  }
  return EMPTY_NOTE;
}
```

The above code block is a current method in v1.2.2-R4 responsible for converting integer frequencies to a human readable pitch (note struct). This code could be improved in two ways.

1. Change the iteration algorithm from linear to a more efficient one.
2. Add a “cache” variable.

The second one in particular is an interesting request. A static “cache” variable could be added in the method and the cache could be whatever the last return value of the method is. Then, every time the method executes, the method will run a conditional branch to check if the frequency matches that of the cached note struct, and if so, return the pitch of that cached note struct. This saves the entire pain of the for-loop. However there are two downsides, one is increased SRAM usage due to the static variable and two is the fact that if the cache “misses” then the program checks an extra branch without needing to, therefore wasting energy. In this case however, adding a cache variable would be beneficial because this method is frequently called in the CreatorMode, and because it is run thousands of times a second to update the segment display, a cache variable would actually be a net positive because a large portion of executions of this method will produce the same result anyways. This is one example of an optimization trade off in TuneStudio2560.

An example of an optimization which has been made late-stage is the decision to convert Song objects in the program from pointers (dynamic) to a single static global reference. The first was used because it provided a cleaner and more "object-orientated" way of making/removing songs. However because the Arduino has no memory management code, this can create problems on the heap over time. The newest method was the use of a global "prgmSong" variable. Now songs are no longer dynamically allocated, rather a single object is addressed whenever a song needs to be changed/played. Whenever the method is done with the song, instead of deleting it from memory, the song instance is simply "cleared".

7.2.2 PROGMEM

PROGMEM[11] is a frequently used keyword in TuneStudio2560. PROGMEM describes the desire to move the data in a variable from RAM to FLASH. When TuneStudio2560 is compiled, global variables are sent to SRAM unless they are requested to remain in PROGMEM. Because PROGMEM is read-only, variables stored in PROGMEM cannot be changed during the program cycle.
The most common use of PROGMEM in the application is in the "pitches.h" file where each individual pitch string is stored in PROGMEM. This was required because in microcontrollers with less than 2KB of ram, the program would crash when trying to read an SD card. The PROGRAM_NOTES[12] global lookup table took up too much SRAM. Although PROGRAM_NOTES was already flagged with PROGMEM, it was storing pointers to strings; the string data was still being saved inside of SRAM. "pitches.h" was made in order to force the microcontroller to store BOTH the string data AND the string pointer in flash memory. PROGMEM is key in TuneStudio2560 as without it, RAM usage would be critical for smaller microcontrollers.

[11] PROGMEM documentation.
[12] A lookup table of every pitch and frequency for every tune button. Stores a struct inside of a struct inside of an array. The table is crucial as otherwise the program would not know how to convert frequencies to pitches and vice versa.

7.2.3 Performance Modes

Performance modes, dictated by the PRGM_MODE macro, are another crucial part of TuneStudio2560. The current performance mode of the program dictates the program on how much SRAM and FLASH it should use. In layman's terms, setting the PRGM_MODE to Zero (low space mode) will remove parts of large strings or text in order to save space. It will also reduce data types where it can and also reduce the maximum allowed size of songs (which saves SRAM). The PRGM_MODE macro is located in the tune_studio.h header file where it can be changed to 0, 1 or 2. The table below describes what happens when changing the program mode.

Updated as of v1.2.2-R4.

PRGM_MODE == 0
```
RAM:   [==        ]  15.9% (used 1300 bytes from 8192 bytes)
Flash: [=         ]  11.8% (used 29974 bytes from 253952 bytes)
```

PRGM_MODE == 1
```
RAM:   [==        ]  20.5% (used 1682 bytes from 8192 bytes)
Flash: [=         ]  12.6% (used 32090 bytes from 253952 bytes)
```

PRGM_MODE == 2
```
RAM:   [===       ]  26.8% (used 2197 bytes from 8192 bytes)
Flash: [=         ]  12.7% (used 32306 bytes from 253952 bytes)
```

Increasing to a larger Program Mode will allow the creation of larger songs as well as additional features and sometimes even faster performance. In general Program Mode 1 is recommended by default for Arduino Megas as it is the most balanced. Program Mode 0 generally reduces flash space by cutting out large strings and parts of instructions. For example, the README.TXT files generated on PRGM_MODE==0 are much smaller then README.TXT files generated on larger Program Modes. *It should be noted that increasing PRGM_MODE from 1 to 2 requires changing the "song_size_t" typedef in song.h from "uint8_t" to "uint16_t" in order to accommodate larger song sizes.*

In Addition to the previously mentioned performance mode, an additional macro should be specified. *This macro is* **"FAST_ADC"** which is automatically enabled in PRGM_MODE==2. When **FAST_ADC** is enabled, the analogRead function in the program executes noticeably faster than usual. This increases the Iterations Per Second (IPS) of CreatorMode and may help prevent flickering of the segment display if the **ProgramState** was running too slowly previously. Enabling **FAST_ADC** takes an extra 30-40 bytes. Enabling the FAST_ADC increases the iterations per second of CreatorMode by around 300-400.

7.3: Low Level Overview

7.3.0 Prelude

This subsection details the "lower level" details of the various components of TuneStudio2560. Here, the code and program execution cycle is explained with detail and C++/Arduino concepts are used liberally. It is recommended to view the source code while going through this section.

7.3.1 Main Class

The program cycle begins in the main class "main.cpp" where the first setup() and loop() functions are executed.
The file is defined as an organized structure.

- Include Tags
- Global Variables
- Global Extern Variable Definitions
- Interrupt/Delay Methods
- Main/Setup Methods
- Utility Methods

A variety of code comments in the file explain and divide the different sections from each other.
First, global variables are defined. These variables include: **immediateInterrupt, lastButtonPress, selectedSong, selectedPage**.

The "**immediateInterrupt**" variable tracks whether or not the user has pressed a control button and we need to navigate to a different program state. When the immediateInterrupt variable is true, many utility methods in the main class will

prematurely finish execution in order to go to the next instruction as fast as possible. This is needed in order to exit blocking delays instantly. The immediateInterrupt variable becomes true when the user is in a program state which makes use of interrupts to skip blocking delays (delay_ms function). After every iteration of the main loop(), the variable is set back to false. The variable can be thought of as almost a "note to the program to skip as fast as possible until the current loop iteration has finished". The immediateInterrupt variable is *ALMOST ALWAYS* used for navigation but there can be other uses for it as well (like skipping instructions without changing the ProgramState).
The "**lastButtonPress**" variable tracks the last time the main class method registered that a button was pressed. The **is_pressed** method runs on a debounce to determine if a switch button's change in value should be recognized. If **is_pressed** is **true**, then the method returns **true** and updates the "lastButtonPress" with the current time in millis(). The debounce is used so the program doesn't register multiple button presses at once. *Note that not all checks to see if a button is pressed in TuneStudio2560 make use of the is_pressed method. When the is_pressed method is not used to check for button presses, then no debounce check is used (unless a different one is specified).*

The "**selectedSong**" and "**selectedPage**" globals are used entirely for the listening mode. These variables represent what position the user is in the SD card. The position the user is in is calculated using a formula depending on the current page and the current selected song. *More info is in the "Program States" section.*

After the globals, extern globals are defined. Extern globals are used in TuneStudio2560 to define global variables (declared extern) which were specified in **tune_studio.h *(Main Header File)***. The three extern variables which are used are all representations of objects in TuneStudio2560 which should be accessible to the entire program. These are:

```
LiquidCrystal_I2C lcd(0x27, LCD_COLS, LCD_ROWS);
SevSegShift segDisplay(SHIFT_PIN_DS, SHIFT_PIN_SHCP, SHIFT_PIN_STCP);
Song<MAX_SONG_LENGTH> prgmSong(SPEAKER_1, DEFAULT_NOTE_LENGTH,
DEFAULT_NOTE_DELAY);
```

```
static SdFat SD;
```

The “**lcd**” object is a globally-accessible object in TuneStudio2560 which represents the LCD attached to the Arduino. Using the lcd object, a variety of tasks using the LiquidCrystal_I2C library can be performed *(such as lcd.print(“Hello”);)*

The “**segDisplay**” object represents the 4-digit 7-segment display connected to the Arduino. The segDisplay is only currently used in CreatorMode but is globally accessible so utility methods can access it and so the display can be cleared/reset from any class.

Finally, the “**prgmSong**” is a new addition as of v1.2.0-R3. Prior to this version, songs were defined as “pointers” and each **ProgramState** would have a different local variable which was a pointer to a song object. Then, when the state was initialized, the pointer would be allocated to memory as a **new** Song object. This could cause some heap fragmentation problems as dynamic memory allocation is usually not recommended for microcontrollers. As a result, a new “global” declaration of a single single object was made for every method in the program to access. Making a global Song object allows the user to have a better understanding of how much SRAM they are using (because the object is counted into static RAM at compile time). Now, whenever a **ProgramState** or method wants to access the current song, they can just call prgmSong.<method>. However, the disadvantage (besides being slightly more complex than new objects) is that the prgmSong variable will retain its value across different program states and methods. This means that every time a user leaves a state, it is important to set prgmSong’s values to their defaults.

The “SD” variable is NOT a extern declaration, rather it is a special case of a global object. Because the SD variable is only ever needed to be accessed in the main class (because of utility methods) it is declared “static” and is inaccessible in other files.

Following the variable declarations, two methods are defined above the setup() and loop() methods. These two methods are "**is_interrupt**" and "**delay_ms**". The reason these two methods are defined above the setup() and loop() is because they are widely used throughout the program and putting them at the top of the file draws significance to the idea that they are widely used in many utility methods. "**is_interrupt**" simply returns if immediateInterrupt is true and "**delay_ms**" is a custom blocking delay function in TuneStudio2560 which is ignored if interrupts are currently being used.

Finally, one more minor exception to the organized structure as defined above, is this declaration.

```
static ProgramState * prgmState;
```

This is the declaration of the official "**ProgramState**" of the application. **ProgramStates** are explained in 7.3.4 subsection but a brief overview will be mentioned here. Every section of the application is sectioned off into "ProgramStates". Each program state manages user input, displays to the lcd, and plays sounds in a different way. In order to be efficient in memory, these different states are not accessible to each other so their global variables, methods, and code does not take up space on the SRAM. Whenever a **ProgramState** is changed, the "**prgmState**" variable is freed from the heap and reallocated as a different **ProgramState** object.

The **setup()** and **loop()** methods are defined next. These methods are only available inside the main.cpp file. The **setup()** method initializes all of the hardware of TuneStudio2560 as well as sets up the Serial monitor (DEBUG only) and initializes the current **ProgramState** to be the **MainMenu**. Code comments are available for the individual actions that **setup()** takes, but a general note is that **setup()** uses direct port manipulation to map inputs and outputs, as well as uploads custom characters to the LCD by converting *PROGMEM* chars to memory and uploading them. The **loop()** function uses a general execute() method from the **ProgramState** (explained in 7.3.4) to run the current **ProgramState.** Then the **loop()** function sets "immediateInterrupt" to "false" in case the interrupt was used. If PERF_METRICS are enabled, then the **loop()**

function also prints a variety of performance monitoring data to the Serial monitor regarding how long the current loop iteration took.

The main class then provides a large set of “utility methods”. These utility methods are accessible throughout all classes in TuneStudio2560. The utility methods in the main class are the only methods which are truly “global”. Each different utility method has its own Doxygen documentation and the utility methods are *somewhat* grouped together if they perform actions on similar functions in the program.

7.3.2 Main Header File

The main header file of TuneStudio2560 is “**tune_studio.h**” located in the include directory. This header file is globally used throughout the program and declares all of the globally-accessible methods (of which are defined in the main class). Along with global methods, this file also declares many constants which are used as well as PROGMEM constants and various macros.

The **DEBUG** and **PERF_METRICS** macros are the most notable in tune_studio.h. The **DEBUG** macro should be set to “true” when the user wants debug messages (Serial.print) to print to the console. If **DEBUG** is “false” then all of the Serial prints are cut out by the preprocessor using #if tags. **PERF_METRICS** macro is used on top of **DEBUG** to provide debugging messages related to performance. **PERF_METRICS** prints out occasional messages to the Serial monitor regarding RAM usage as well as enables the code at the bottom of the loop() which prints out various indications regarding the performance of the **ProgramState**. The **PERF_METRICS** macro is very useful when the user is trying to optimize a **ProgramState** which relies on high iterations per second (IPS) to work well. Enabling **DEBUG** or **PERF_METRICS** (in addition to increasing SRAM and program memory usage) decreases the speed of the program as the application must now use CPU clock cycles to print to the Serial monitor.

When PERF_METRICS is enabled, the loop() method prints the following each iteration of the main loop.
Note that none of the below are impacted by the Serial prints from performance metrics as the total time of completion is calculated before the following messages are printed. Debug messages impact performance though.

- The total time in microseconds (μs the loop took to complete).
- Current RAM usage (in Bytes).
- Current percent of RAM utilization (in percent).
- Clock Cycles the loop took to complete.
- How fast the loop is executing per second (Iterations Per Second [IPS])

7.3.3 LCD & Segment Display

The Liquid Crystal 20x4 I2C Display is used in the program to communicate feedback to the user via text. Every program state in the program utilizes the lcd global object which is declared in **tune_studio.h** and defined in **main.cpp**. The 4-digit 7-segment display, which is also globally defined, is used to communicate the current "pitch" the user has selected in CreatorMode (see 7.3.4 for more info).

The "lcd" object is controlled through a variety of utility methods in the main class. The utility methods to control the lcd in the main class utilize the basic methods provided by the LiquidCrystal_I2C library. Although the regular (library-provided) **lcd.print(F(str));** is frequently used, two utility methods are provided by the main class for special cases of printing text. These are:

```
void print_lcd(const __FlashStringHelper * text, uint8_t charDelay) {...}
void print_scrolling(const __FlashStringHelper * text, uint8_t cursorY, uint8_t
charDelay) {...}
```

As seen by the code block, both of these methods utilize the **__FlashStringHelper*** which is provided by the Arduino main library (Arduino.h) to conveniently store constant strings in PROGMEM. The **print_lcd** method takes in a Flash String and a delay, then it clears the lcd and prints out the entire string character by character with a delay specified by the **charDelay**. The method starts printing characters at *Row 0, Column 0*, and then moves until *Row 3, Column 19*. When the method detects it has reached the end of the lcd, it clears the screen and restarts at *Row 0, Column 0* and continues printing the string. The **print_scrolling** is similar to the **print_lcd**, but only uses ONE row to print text and does not clear the lcd prior to execution. When the text begins, the method prints the first 20 characters of the string and then scrolls to the right by one character each **charDelay** milliseconds until the string has been fully read.

Controlling the segment display is simple. No utility methods are provided for the segment display, and the default library methods from the **SevSegShift** library are used for controlling it. An important reminder is that the segment display needs to be constantly cleared due to the fact that the shift registers may hold garbage data from a previous update. Without clearing the display constantly, extra "garbage" data might stay on the segment display. If the segment display updates too slowly (due to a low IPS) then the segment display flickers.

7.3.4 Program States

The concept of program states are very important to understand in TuneStudio2560. A "**ProgramState**" refers to a class located in **state.cpp** and declared in "**state.h**". A program state represents a parent class in which all program states in the application inherit from. This parent class allows a global "**prgmState**" variable (main.cpp) which can be freed from memory and then reallocated as a new program state. When this variable is reallocated the application begins executing the new program state immediately. The concept of a **ProgramState** is simple.

- Has an enum which represents the type of state the class is (ex. Main Menu, etc)
- Has a public method **get_state** which returns the enum.
- Has a variable which tracks whether or not the **init()** method has been run.
- Has a private method known as **init()** which is executed when the **ProgramState** first runs. When executed, the variable which tracks the init() is changed to "true".

- Has a public method **has_initalized** which returns if the **init()** method has been run.
- Has a private method **loop()** which contains the code that should be run infinitely.
- Has a public method **execute()** which dictates if the **init()** or if the **loop()** method should be run. Also runs a chunk of code which the developer wants to run before every state change (for example, clear the lcd).

Every different program state inherits the **ProgramState** class and defines its own **init()** and **loop()** methods. Similarly, each different program state class also defines its own **StateID**, an enum which allows the fast comparison between program states. The 5 different program states in the application are declared in the “**states.h**” header file and initialized in their respective “.cpp” files in the “src/studio-libs/states” directory.

The “**states.h**” class also contains the global variables that each program state can use as well as the individual methods that each program state can use. This means that any instance of **ProgramState** can indeed have its own class-accessible variables and methods.

Everytime the user changes “**prgmState**” the previous memory is freed and a new instance is allocated. This means that any data in the previous object is irrelevant (such as if the program state ran the **init()** method). The **init()** method should be used similar to **setup()** but instead of initializing data for the whole program, data is only initialized for the specific program state. The **loop()** method is the code which should be run on repeat.

7.3.5 Notes & Pitches

In order to play audio, the piezo buzzer must be interacted with in code. This speaker takes in values as unsigned 16-bit integers and converts them into a square wave to play. Instead of having the user add individual frequencies to a song, we can use human readable “pitches” instead. Using raw frequencies would be far too complicated and anti-user so we convert various frequencies to pitches on the standard chromatic scale. Ex: GS2 -> 104 and 104 -> GS2. This way, an intuitive way of providing sounds can be given to the user and when the sounds need to be played, program methods can convert the strings to integers and vice versa.

The computer does not know how to convert “104” to “GS2”. A lookup table[13] is needed in order to provide a database of various pitches and frequencies. This table is referred to as “**PROGRAM_NOTES**” in the tune_studio.h file and is globally-accessible.

The “**PROGRAM_NOTES**” array contains an array of C++ struct “**buttonFrequencies_t**” which in itself is a pin (8-bit int) and an array of C++ struct “**note_t**”. “**note_t**” is a struct which contains a char pointer “pitch” which refers to a *string in memory* and an *unsigned 16-bit integer value* which refers to the frequency to play.
The **PROGRAM_NOTES** declaration and the two structures code is provided below for reference.

```
typedef struct note {
  const char * const pitch;
  const uint16_t frequency;
} note_t;

typedef struct buttonFrequencies {
  const uint8_t pin;
  const note_t notes[TONES_PER_BUTTON];
} buttonFrequencies_t;

const buttonFrequencies_t PROGMEM PROGRAM_NOTES[TONE_BUTTON_AMOUNT] {...}

const note_t PAUSE_NOTE = { "PS", (const uint16_t)1 };
const note_t EMPTY_NOTE = { "0000", (const uint16_t)0 };
```

Figure: Shows code from tune_studio.h including how the various structures are defined.

[13] View a table of pitches and frequencies.
https://github.com/devjluvisi/TuneStudio2560/wiki/For-Users#supported-pitches

A “**note**” is a structure which contains a PITCH and FREQUENCY.
A “**buttonFrequencies_t**” is a structure which contains the PIN of a tune button and an ARRAY of **notes** that the tune button can play.
The “**PROGRAM_NOTES**” is an ARRAY of **buttonFrequencies_t** of length 5 (5 tune buttons) and defines all of the notes that each tune button can play.

In addition to the “**PROGRAM_NOTES**” array, two additional notes are defined.
PAUSE_NOTE: A note which represents a “block” in the song (delay) for **PAUSE_DELAY** milliseconds (500 default).
EMPTY_NOTE: A note which represents a frequency of “0” and has not been defined. Usually used to indicate the ending of a song.

PROGRAM_NOTES is stored in *PROGMEM* so the values in the array must be retrieved via the various “pgm” methods in Arduino.

The individual pitch strings in **PROGRAM_NOTES** are just pointers to actual string data. Storing those strings directly in the **PROGRAM_NOTES** would take up a lot of SRAM as constant data. In order to store the strings in *PROGMEM*, a “**pitches.h**” file is defined. This file has every possible pitch in it defined in *PROGMEM*. Then the **PROGRAM_NOTES** references these various *PROGMEM* variables. When a note needs to be retrieved from PROGMEM the **pgm_pcpyr** method should be used. This method has a static “buffer” to keep track of its return value. This method converts a X and Y index in the 2D **PROGRAM_NOTES** array and pulls out the specified “pitch” at that index.

```
// Example of a pitch reference in PROGRAM_NOTES
{BTN_TONE_1, {{pitch_b0, 31}, … }

// Examples of three pitches in the "pitches.h" file.
const char pitch_e5[] PROGMEM = "E5";
const char pitch_f5[] PROGMEM = "F5";
const char pitch_fs5[] PROGMEM = "FS5";
```
Figure: Shows how notes are defined in PROGRAM_NOTES as well as how pitch strings are saved in PROGMEM.

In order to convert from pitch to frequency and frequency from pitch, two methods are used.

```
note_t get_note_from_pitch(const char *
  const pitch) {...}
note_t get_note_from_freq(const uint16_t frequency) {...}
```

These two methods perform the opposite actions internally but return the same value: a note structure with the correct "pitch" and "frequency" attributes.

7.3.6 Songs

Songs are represented by the global "**prgmSong**" variable (as of v1.2.0-R3). This global variable is an object from the **Song** class and should be used by any method who wants to use the current song being used by the program. A song object contains the following internal attributes:

Pin: What pin the sounds from the songs should be played on.
Note Delay: The delay (ms) between each note playing.
Note Length: The length (ms) that each note plays for.
Current Size: The current size of the song.
Song Data: An array of unsigned 16-bit integers which represent various frequencies in the song that should be read.

The song class has two important aspects that need to be mentioned.

- **song_size_t** type definition.
- template<> structure.

The "***song_size_t***" typedef represents either a uint8_t or a uint16_t. This variable should be changed depending on the value of the constant **MAX_SONG_SIZE** in "**tune_studio.h**". Increasing the **MAX_SONG_SIZE** above 255 requires changing the **song_size_t** to uint16_t (this takes up more SRAM and program space).
The template<> structure is used to define how large songs are allowed to be in the Song class. This allows the reduction of SRAM usage. The template<> is set to template<**MAX_SONG_SIZE**> by default.

The “**song.cpp**” class contains various methods for messing with songs such as adding notes, removing notes, getting the size, playing it (blocking), setting attributes, etc. It is important that every time a new state is switched to the “**prgmSong**” variable should be cleared using **clear()** method. Some program states (listening mode) have different ways of playing back songs instead of the **play_song()** method in order to prevent blocking with **delay_ms**.

7.3.7 SD Card

Interaction with non-volatile storage is done through main class utility methods. These utility methods make use of the SdFat library for writing and reading. Important data about the SD card is listed below:

- The SD card must be FAT16/FAT32 formatted.
- The names of the song files on the SD card must be 8 characters in length or less (excluding .txt extension).

If the format of the SD card is wrong, TuneStudio2560 will blink a red status led and display an error message on the LCD.

Song files are saved using the following parameters:

- File saved using the “.txt” extension.
- File saved using A-Z, 0-9, and underscores only.
- Files saved using 8 characters MAX in length (1 min).
- File saved using a custom layout so it can be read from and modified easily.
- File has at least 8 notes in it.
- Song notes are saved in their human readable pitch form (Like FS2) and are parsed into integers when the file is loaded from storage.

Song files are read using the following parameters:

- File name is below 8 characters.
- File is a “.txt” file.
- File name using A-Z, 0-9, and underscores only.
- File has the correct internal layout.
- Number of notes in the file is less than the max allowed notes in the program.
- File has at least 8 notes.

- File has proper attributes for TONE_DELAY and TONE_LENGTH.

If any file does not follow the naming and file extension conventions, it is ignored when trying to select songs from the ListeningMode. If a song has correct naming conventions but bad data, then the status LED will blink red and an error message will be displayed. The song will not be played.

In addition to the saving/reading of song files, the SD card also creates a "README.TXT" file which contains important information about editing song files as well as some links. In PRGM_MODE==0, not all of the README is generated in order to save program space.

7.3.8 Additional Information

ProgramStates in the application are designed to be as efficient in both program space and ram usage as possible. Each **ProgramState's** individual methods and variables are not accessible from other **ProgramStates**.

The "***song_size_t***" type definition is automatically changed in most instances when the PRGM_MODE is changed. Each **PRGM_MODE** defines a different **MAX_SONG_SIZE** macro. However when changing **PRGM_MODE**, the "***song_size_t***" must also be updated in the song.h file. This is currently a bug with the preprocessor and is being worked on.

Various string constants in the program (such as README and instructions) are cut out in PRGM_MODE==0. PRGM_MODE==0 is intended for a future port to the Arduino Uno as it is the only program mode which fits into the Arduino's RAM and Program Space requirements.

7.4: Debugging

7.4.0 General Information

Debugging is important to finding bugs and adding features to TuneStudio2560. Most debugging information was mentioned in the "Program Modes" subsection. In general, **DEBUG** macro should be enabled for console prints and any call to Serial.XXX should be within an #if preprocessor check.

7.4.1 Debug Macro

```
#define DEBUG false
```

Enables all of the Serial prints to the console as well as the Serial monitor itself. Enabling the DEBUG macro will slow code execution as the CPU has to print to Serial instead of executing the next instruction.

7.4.2 Performance Metrics Macro

```
#define PERF_METRICS false
```

Prints out performance metrics to the serial monitor (Requires DEBUG macro to be true) about the current RAM utilization as well as various other performance metrics about how fast the current program loop is executing.

When PERF_METRICS is enabled, the loop() method prints the following each iteration of the main loop.
Note that none of the below are impacted by the Serial prints from performance metrics as the total time of completion is calculated before the following messages are printed. Debug messages impact performance though.

- The total time in microseconds (µs the loop took to complete).
- Current RAM usage (in Bytes).
- Current percent of RAM utilization (in percent).
- Clock Cycles the loop took to complete.
- How fast the loop is executing per second (Iterations Per Second [IPS])

7.5: Unit Testing

7.5.0 Prelude

Unit Testing is a separate part of TuneStudio2560 which is used for testing the hardware. The “**test_hardware.cpp**” file contains the code to perform this action. Unit testing is done with the “Unity” library. The “Unity” library is required to perform the Unit test but is not required to upload/run the program.
Important: Currently unit testing has bugs and may not work, the code is still available and does work but some bugs need to be worked out.

7.5.1 How to Use

In order to run the hardware unit test, navigate to the project and then press CTRL+SHIFT+P to enter the command palette. Once there type “PlatformIO: Test” and click the first option. This will run the unit test. For the hardware unit test it is important you follow the on screen instructions to make sure the hardware is working as the computer does not know.

7.5.2 Future Plans

Future plans are to explain Unit Testing to program methods as well instead of just verification of hardware.

Reference Website

TuneStudio2560 has its own reference website! The website is hosted on GitHub pages and is used to provide additional version information, changelogs, downloads, pictures, videos, and Doxygen documentation. The webpage is constructed using basic HTML5 with inline CSS and some JavaScript (JavaScript is not required to load the webpage). The website also has **dark mode**!

Visit it → https://devjluvisi.github.io/TuneStudio2560/

The website also contains the link to MakerStudio2560 (NEW). View in section 9.

MakerStudio2560

9.0: Introduction & Purpose

One of the newest and most transformative features to the TuneStudio2560 project is MakerStudio2560. **MakerStudio2560** is a comprehensive song creation, playback, and editing tool made specifically for TuneStudio2560 by the original programmer (myself). **MakerStudio2560** is a client-side JavaScript (ES6) website where users can develop and edit their own songs without having to use TuneStudio2560. **MakerStudio2560** follows the same basic pattern for creating songs on TuneStudio2560 but provides a variety of additional features due to the fact that **MakerStudio2560** is running on a PC and is not limited by its hardware. **MakerStudio2560** is completely backwards compatible with TuneStudio2560 so any songs you develop on TuneStudio2560 you can playback on **MakerStudio2560** and vice versa. **MakerStudio2560** allows you to download your song file that you made and it is automatically compatible and ready for playback on an SD card.

9.1: How to Use

A full YouTube tutorial on how to use MakerStudio2560 is available on the reference site.

View text instructions on the "View Instructions" details tag at the top of the webpage.

9.2: Code

9.3.0 Prelude

MakerStudio2560 is developed entirely in ES6 JavaScript. The code in MakerStudio2560 is approx. 1000 lines. The code is not minified when it is used.

9.3.1 JavaScript Adoption

In order to "port" TuneStudio2560 to JavaScript some changes were made and some inconsistencies can exist. The speaker on your computer is not the same as the piezo passive buzzer which plays sounds on the Arduino. However a audio library is used to generate square waves and play frequencies (hz) back to you. The JavaScript port of TuneStudio2560 took about 1-2 weeks to develop entirely.

9.3.2 Basic Overview

MakerStudio2560 has all of the features TuneStudio2560 has when it comes to making and playing back songs. Along with these features, MakerStudio2560 has extra features as well:

- Auto-Save to local storage.
- Edit and save an "Author" of a song.
- Save "Creation Date" of a song.
- Save "Last Edit Date" of a song.
- Edit and remove notes at any point in the song.
- *Unlimited* amount of notes.
- Adjust volume when playing the song back.
- View the time passed (in seconds) and max time (in seconds) of a song when playing back a song.
- Adjust Auto-Save Settings.
- Adjust how the website looks (dark mode included).
- Edit previously saved songs.

The website is quite user-friendly to use and a list of instructions is provided on the website as well.
Users can open saved songs on their Arduino using the "Choose File" button or make a new song using the "Create File". Users who have previously been editing a song can click on the "Resume File" button (Auto-Save).

9.3.3 Documentation

A full YouTube tutorial on how to use MakerStudio2560 is available on the reference site.

View text instructions on the "View Instructions" details tag at the top of the webpage.

The JavaScript code contains comments describing methods and functions. The entire code for MakerStudio2560 is available on the GitHub (/docs folder). MakerStudio2560 is not as thoroughly documented as TuneStudio2560 is but if you understand how TuneStudio2560 works, MakerStudio2560 is similar.

Project & Code Documentation

10.0: Prelude

TuneStudio 2560 contains a very large amount of documentation for its code. Besides this product brief, users can view both a User Guide as well as a Developer Guide on the wiki. If users want method by method documentation then they can view the Doxygen documentation.

10.1: Inline Documentation

TuneStudio2560 has comments throughout almost all of its code. Each method in TuneStudio2560 is documented in its respective header file. In addition, all global variables are documented and each file also contains a Doxygen header comment describing what the file does. Every directory outside of src also contains a "README" file describing the contents in the directory.

10.2: GitHub Wiki

View the GitHub for documentation.

- User Guide
- Developer Guide
- Build It
- README

10.3: Public Doxygen Documentation

TuneStudio2560 is commented using the Doxygen standard for comments (similar to JavaDoc). A website is generated from these Doxygen documents and is publicly available for all users to look at. The website is highly recommended for users who want a method-by-method and variable-by-variable documentation.

→ Public Doxygen Documentation ←

10.4: YouTube Tutorials

A variety of YouTube tutorials for TuneStudio2560 are available on the reference website for viewing. The site also contains real life images of the working project that I built.

Future Suggestions

For future suggestions to make TuneStudio2560 better, either request a pull on GitHub or email me directly! I respond to all personal emails from interested developers and users. A list of future planned items is available on the reference website.

Conclusion

I hope you have enjoyed reading the product brief for TuneStudio2560. This document took a long time to make as a full time student! This project was also the biggest project I have ever personally completed in my programming career up to this point. I learned a lot about the Arduino as well as embedded systems. For any future suggestions, comments, or changes to this product brief you want to see, email me or send a private discord message.

jluvisi2021@gmail.com
Interryne#0943

Please note that the following product brief was generated for TuneStudio v1.2.2-R4. Later released versions may have changes or updated information that conflicts with this document.

End of official product brief for TuneStudio2560 and MakerStudio2560.

GitHub: https://github.com/devjluvisi/TuneStudio2560
Website: https://devjluvisi.github.io/TuneStudio2560/
Document assembled 9/11/2021.
Last modification on 9/15/2021.