Project 1 - Function Generator

Cal Poly, San Luis Obispo

CPE 316-01-2234

Paul Hummel

Noah Masten

Spring Quarter

April 30th, 2023

Device Behavior

This project represents a simple function generator that displays a variety of waveforms at a constant peak-to-peak voltage of 3.0 V with a DC-bias of 1.5 V. The project utilizes a STM32L476RG microcontroller board, a MCP4922 Digital-to-Analog Converter (DAC), and a 4x4 keypad.

Upon startup, the device produces a 100 Hz square wave with a 50% duty cycle. The user can choose to press one of 12 keys on the keypad, each with a different function:

- **Keys 1-5:**
    - Adjusts the frequency to:
        - **1:** 100 Hz
        - **2:** 200 Hz
        - **3:** 300 Hz
        - **4:** 400 Hz
        - **5:** 500 Hz
- **Keys 6-9**:
    - Adjusts the wave form to:
        - **6:** sine wave
        - **7:** triangle wave
        - **8:** sawtooth/ramp wave
        - **9:** square wave
- **Keys STAR, 0, or HASHTAG (for square waves only):**
    - Changes the current duty cycle of the square wave:
        - **STAR:** By -10%
        - **0:** To 50%
        - **HASHTAG:** By +10%
    - Note: the duty cycle of the square wave cannot go below 10% or exceed 90%.

System Specifications

Table 1. System Specifications Table

Function Generator	
Output Voltage	3.0 Vpp
DC bias	1.5 V
Waveform types	Sine, Triangle, Sawtooth, Square
Frequency range	100 Hz – 500 Hz
Duty cycle range	10% – 90% (square wave only)
STM32L476RG Board	
Input voltage	-0.3–4.0 V (min. – max.)
Total current draw	150 mA (max.)
Power consumption	0.6 W
Power connection	USB to Mini-B cable
MCP4922 DAC	
Operating Voltage	2.7-5.5 V (min. – max.)
Operating Current	700 μA (max.)
Power consumption	3.85 mW
4x4 Keypad	
Usable keys/key function	**1:** 100 Hz **2:** 200 Hz **3:** 300 Hz **4:** 400 Hz **5:** 500 Hz **6:** Sine wave **7:** Triangle wave **8:** Sawtooth wave **9:** Square wave **STAR:** -10% duty cycle (square wave) **0:** 50% duty cycle (square wave) **HASHTAG:** +10% duty cycle (square wave)

System Schematic

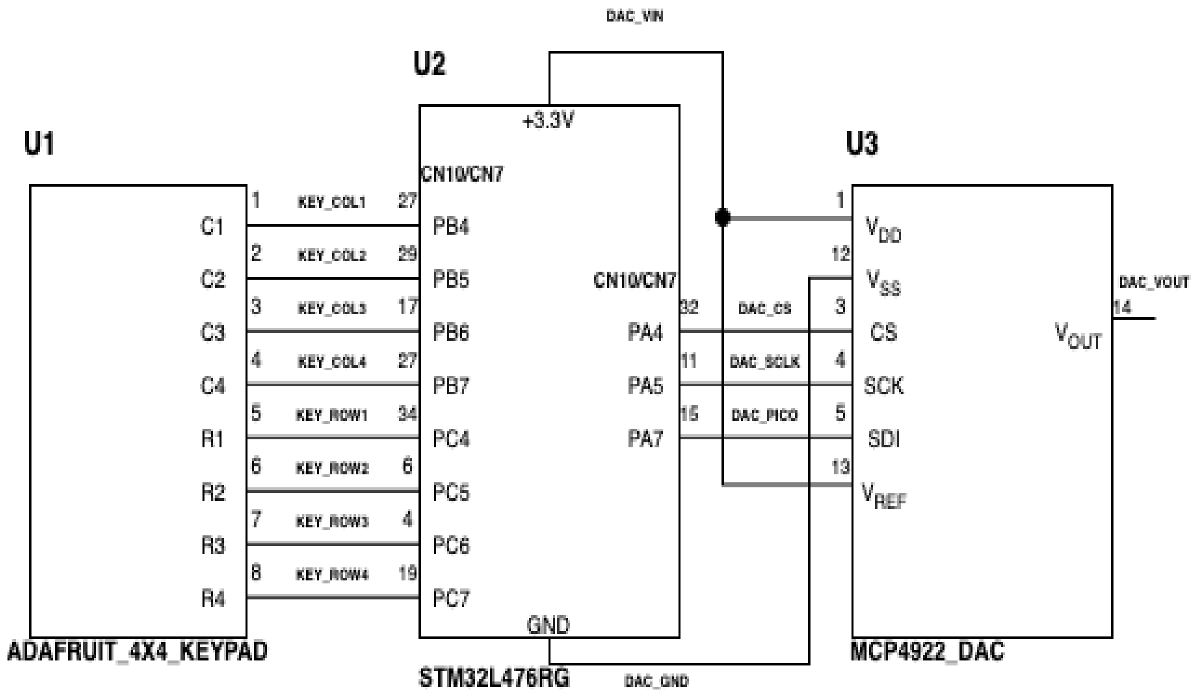


Figure 1. System Schematic

Software Architecture

1. Overview

The program starts by initializing all global variables, **`#define`** variables, and Finite State Machine states. It then configures the keypad pins for GPIO, and the DAC pins for alternate function GPIO (Serial Clock, Chip Select, and SDI), as well as configuring the timer (Auto Reload and Compare/Capture registers). The timer interrupt flags are then enabled, and the timer is turned on. Before the program enters the while loop, it initializes the default 100 Hz, 50% duty cycle square wave.

The heart of the program is a Finite State Machine (FSM), used to represent every possible behavior of the function generator: display waveform, change frequency, change waveform, or change duty cycle. Depending on the key the user presses, the FSM will switch to the respective state. Once the function generator has completed the task of the key the user pressed, it will switch back to the display waveform state until another key is pressed.

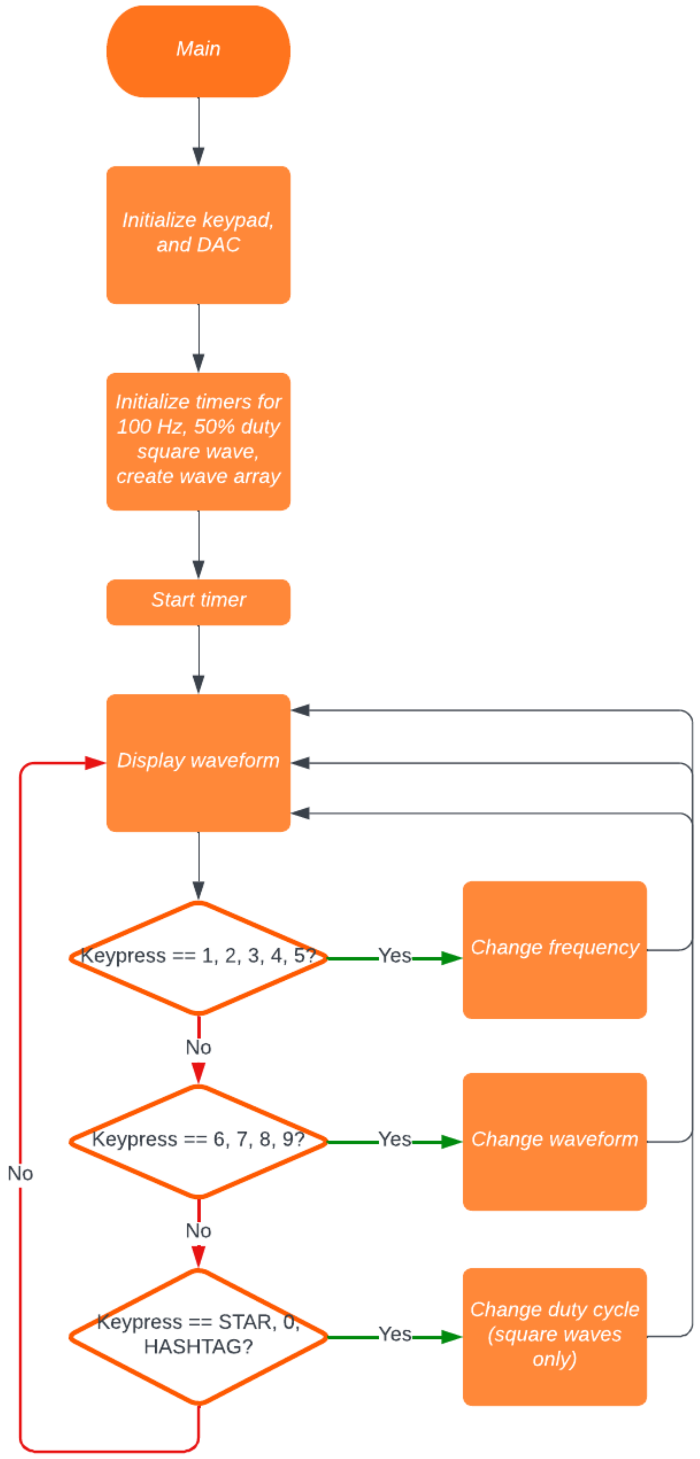

Main
Initialize keypad, and DAC
Initialize timers for 100 Hz, 50% duty square wave, create wave array
Start timer
Display waveform
Keypress == 1, 2, 3, 4, 5?
Yes
Change frequency
No
Keypress == 6, 7, 8, 9?
Yes
Change waveform
No
Keypress == STAR, 0, HASHTAG?
Yes
Change duty cycle (square waves only)
No

Figure 2. `main()` flowchart

2. Variables

Globals

These variables are used in multiple functions throughout the program.

The program uses the **`#define`** keyword to represent constant integers. These values are universal constants, and never change during program execution. The **`#define`** variables used are as follows:

`SINE` – the keypress that changes the waveform to a sine wave, also used in the switch-case block that changes the waveform. Value is 6.

`TRI` – the keypress that changes the waveform to a triangle wave, also used in the switch-case block that changes the waveform. Value is 7.

`SAW` – the keypress that changes the waveform to a sawtooth wave, also used in the switch-case block that changes the waveform. Value is 8.

`SQUARE` – the keypress that changes the waveform to a square wave, also used in the switch-case block that changes the waveform. Value is 9.

`CLK_SPEED` – clock speed of the processor, 20 MHz ($20*10^6$)

`V_PEAK` – Peak to peak voltage of the waveform, 3.0 V. This value is set to 3000 (the voltage in mV) to avoid excessive floating point calculations.

`V_LOW` – minimum voltage of the waveform, 0 V.

`DEBOUNCE` – arbitrary number used for a button debounce delay. This value was found through trial and error. Value is 4000.

`NUM_WRITES` – the integer value representing the number of DAC writes in one period of a waveform. Calculated by taking 1 divided by the total execution time of one DAC write (found to be 86 microseconds), which is the frequency of DAC write. That value is then divided by the 100 Hz frequency (a 100 Hz wave will have the largest number of voltage writes, as it has the longest period of waveform. All other waveforms will have $\frac{1}{frequency}$ times the amount of writes as the 100 Hz waveform).

- $NUM\_WRITES\ =\ \frac{(\frac{1}{(86*10^{-6}\ seconds)})}{100\ Hz} = 116$

`V_CHANGE` – calculated by taking the maximum voltage **`V_PEAK`**, and dividing it by **`NUM_WRITES`**. This represents the integer value in which the current voltage of the waveform will increment by.

- $V\_CHANGE\ =\ \frac{3000}{116} = 26$

`PI` – the number pi, used to calculate the voltage values of the sine waveform. Value is 3.141592.

This program also uses global variables in which the values can change during program execution:

`int8_t wave` – the current waveform, which can change to any of the waveform keypresses (6, 7, 8, 9) during program execution. Initialized to **`SQUARE`**, the waveform that appears on startup.

`int8_t freq` – the current frequency, which can change to any of the frequency keypresses (1, 2, 3, 4, 5) during program execution. Instead of being initialized to 100, it is initialized to 1, which matches the keypress associated with 100 Hz. Any calculations involving this value will multiply the value by 100 to get the respective frequency associated with the keypress.

`int8_t duty` – represents the percentage of the current duty cycle, which can be incremented or decremented by 10 during program execution. This value can never be less than 10 or greater than 90. Initialized to 50, the startup waveform duty cycle.

`uint16_t *wave_array` – array that contains all possible voltage values of one period of the current waveform (not including the square wave). This array is continuously being referenced in the TIM2 interrupt handler, which constantly displays the current voltage value. The size and values of this array change each time the waveform or frequency is changed. For that reason, this array utilizes dynamic memory allocation.

`uint32_t array_size` – represents the size of **`wave_array`**. This value is calculated by dividing **`NUM_WRITES`** by the keypress value of the current frequency (1-5). A higher frequency will make **`array_size`** smaller, as the voltage values will have to be cycled through and displayed *n* times as fast, where *n* is the keypress value of the current frequency. This can also be interpreted as the "resolution" of the wave. Initialized to 116, which is equal to **`NUM_WRITES`** / 1.

`uint16_t count` – represents the current index of **`wave_array`**, and is used to reference the values in **`wave_array`**. This value is incremented in the TIM2 interrupt handler.

`uint32_t curr_volt` – the current voltage value, which is written by the DAC. In the TIM2 interrupt handler, this value is set to **`wave_array[count]`**. Initialized to 0, the beginning voltage value before it increments.

Enum

The **`typedef enum`** keyword is used to declare the **`state_var_type`** variable, which consists of the four FSM states: ***`DISPLAY, CHANGE_WAVE, CHANGE_FREQ,`*** and ***`CHANGE_DUTY`***.

c. Variables declared in `main()`

These variables were intentionally declared in **`main()`** because they are exclusively used in the **`main()`** function, and nowhere else in the program.

`int8_t keyPress` – the current value of the keypress, inputted by the user. Initialized to -1, which represents no key press.

`state_var_type state` – the current state of the FSM. This value changes based on the value of **`keyPress`**, which changes the state of the FSM.

2. Finite State Machine

The states of the FSM were chosen based on the different functions of the device. Since the device can either display the waveform, change the waveform, change the frequency, or change the duty cycle, the states were chosen to model this.

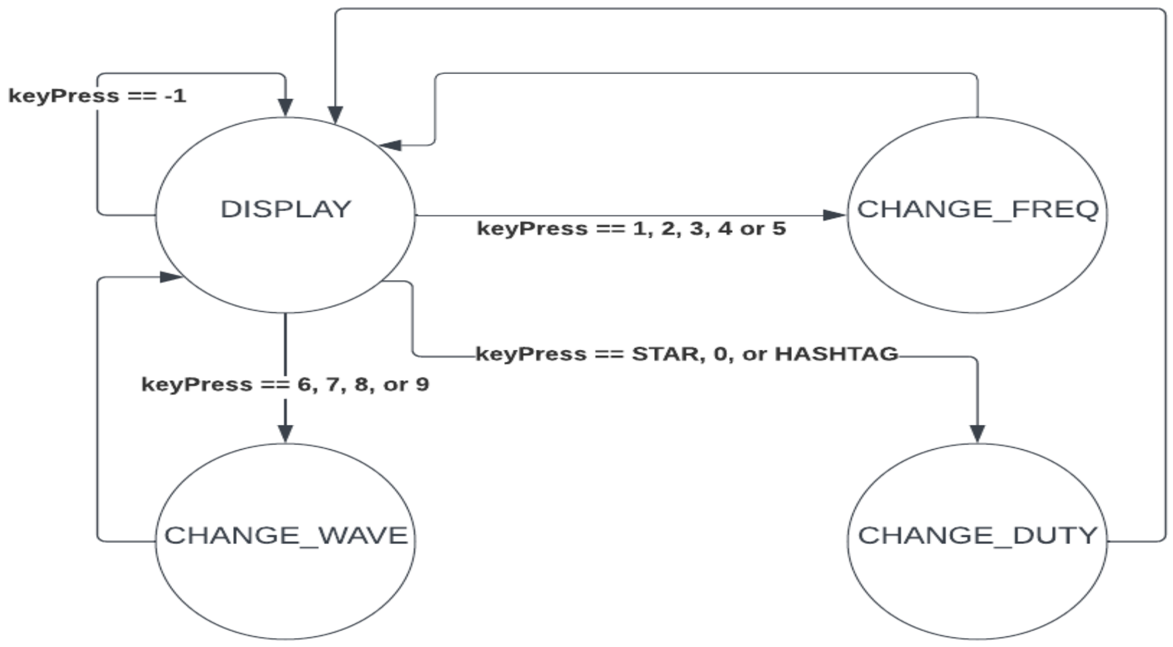

keyPress == -1
DISPLAY
keyPress == 1, 2, 3, 4 or 5
CHANGE_FREQ
keyPress == STAR, 0, or HASHTAG
keyPress == 6, 7, 8, or 9
CHANGE_WAVE
CHANGE_DUTY

Figure 3. State Diagram for Finite State Machine

DISPLAY

This state was chosen as both the initial and default state, because no matter where in the program execution (including the very beginning), the function generator should always be displaying a waveform.

In this state, the program waits for the user to press a key. If the user presses keys 1-5, the state will change to ***CHANGE_FREQ***. If the user presses keys 6-9, the state will change to ***CHANGE_WAVE***. If the current waveform is a square wave and the user presses keys STAR, 0, or HASHTAG the state will change to ***CHANGE_DUTY***. The program then waits for the user to release the keypress. An empty loop is included at the end, functioning as a debounce delay.

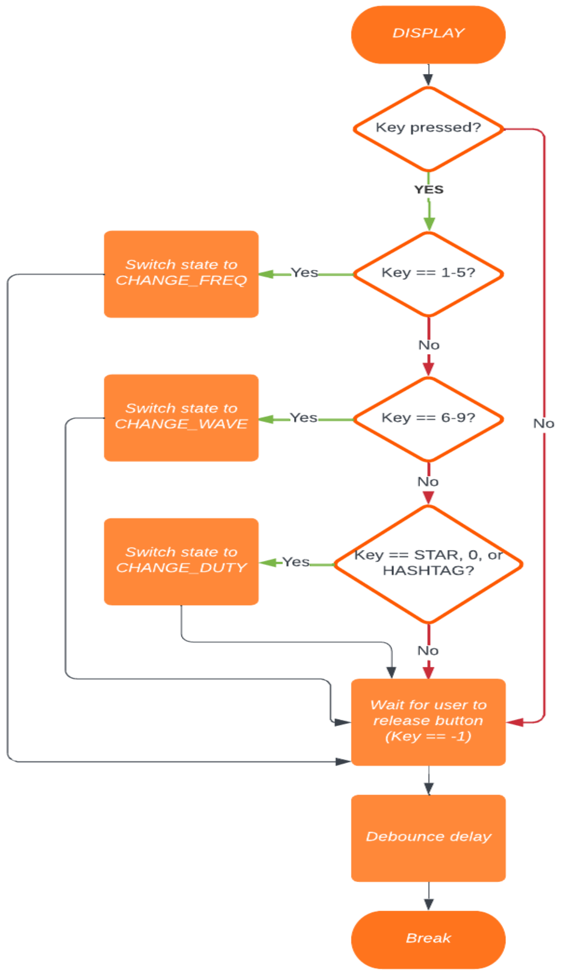

DISPLAY
Key pressed?
YES
Key == 1-5?
Yes
Switch state to CHANGE_FREQ
No
Key == 6-9?
Yes
Switch state to CHANGE_WAVE
No
Key == STAR, 0, or HASHTAG?
Yes
Switch state to CHANGE_DUTY
No
No
Wait for user to release button (Key == -1)
Debounce delay
Break

Figure 4. *DISPLAY* state flowchart

CHANGE_FREQ

In this state, **`freq`**, the current frequency, is set equal to the keypress. The values of the TIM2 registers are then recalculated using this new frequency. This happens in the **`change_timers()`** function (see section 3). Additionally, the values of **`wave_array`** are then recalculated based on the new frequency and the current waveform. This is done in the **`construct_waveform()`** function (see section 3). The current state then reverts back to ***DISPLAY***, where the program will check for another keypress.

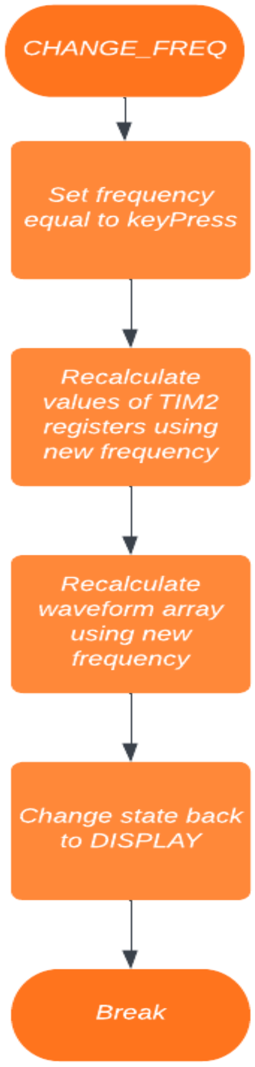

CHANGE_FREQ
Set frequency equal to keyPress
Recalculate values of TIM2 registers using new frequency
Recalculate waveform array using new frequency
Change state back to DISPLAY
Break

Figure 5. *CHANGE_FREQ* state flowchart

CHANGE_WAVE

In this state, **`wave`**, the current waveform, is set equal to the keypress. The values of **`wave_array`** are then recalculated based on the current frequency and the new waveform. The current state then reverts back to ***`DISPLAY`***, where the program will check for another keypress.

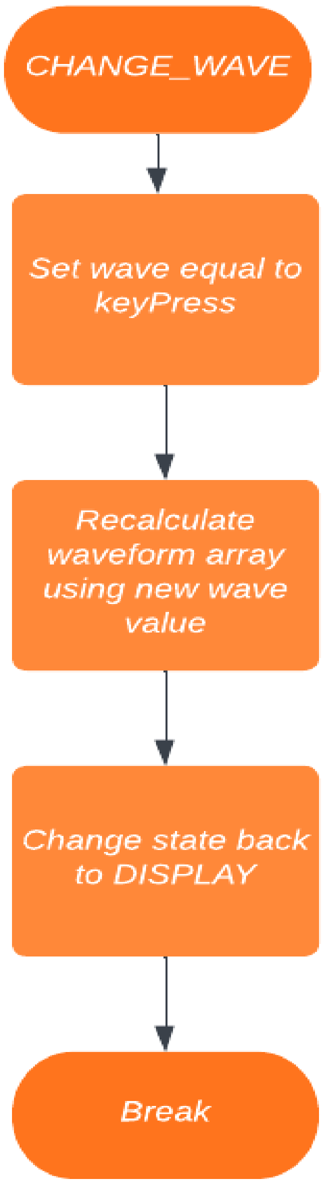

CHANGE_WAVE
Set wave equal to keyPress
Recalculate waveform array using new wave value
Change state back to DISPLAY
Break

Figure 6. *CHANGE_WAVE* state flowchart

CHANGE_DUTY

In this state, **`duty`**, the current duty cycle, is either decremented by 10, reset to 50, or incremented by 10, based on if the keypress equals STAR, 0, or HASHTAG. The values of the TIM2 registers are then recalculated using this new duty cycle, which happens in the **`change_timers()`** function (see section 3). The current state then reverts back to ***DISPLAY***, where the program will check for another keypress.

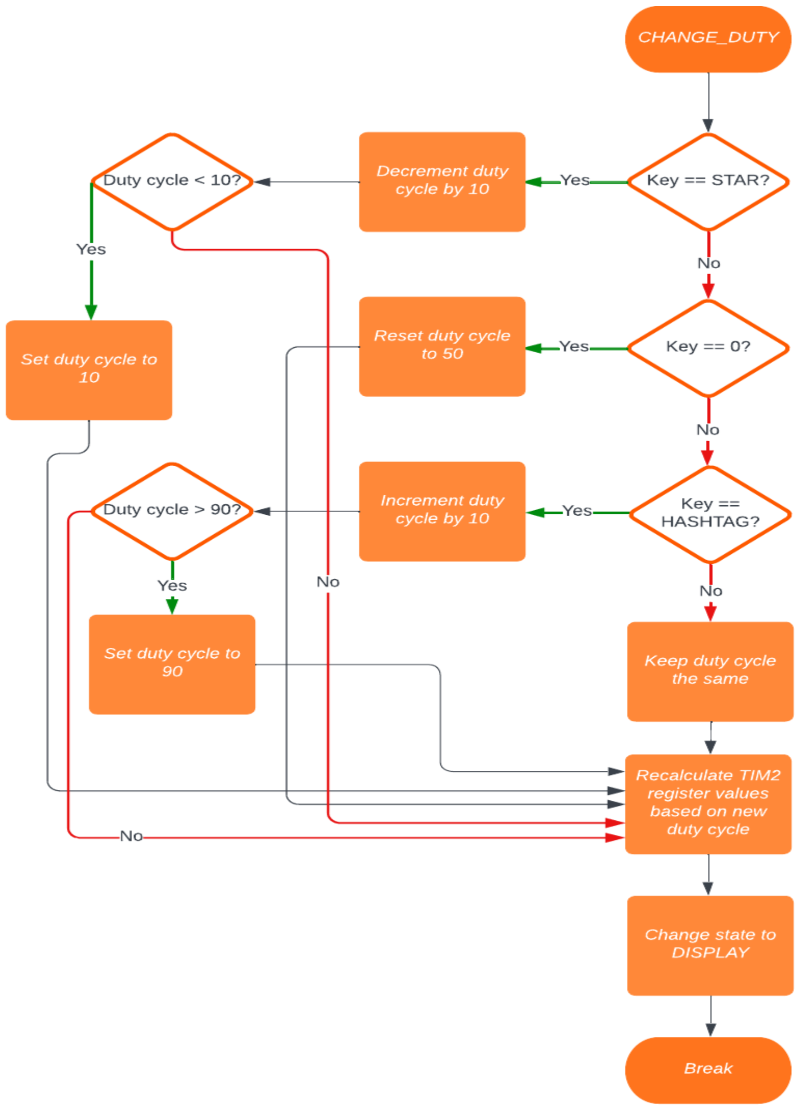


Figure 7. *CHANGE_DUTY* state flowchart

3. Other Functions

`change_timers`

This function modifies the TIM2 register values based on an input frequency and duty cycle. It has two inputs, **`new_freq`** and **`new_duty`**. The registers that are modified are:

> **`TIM2->ARR`** – Auto Reload Register. This functions as the period of the waveform, which can be calculated by dividing the clock speed by the frequency. In the equation, the total period ends up being one less than the actual value, because the timer counts from 0 up to **`TIM2->ARR`**.
>
> - **`TIM2->ARR = (CLK_SPEED / (new_freq * 100)) - 1;`**
>
> **`TIM2->CCR1`** – Capture/Compare Register. Once the timer count reaches this value, an update event is triggered. For square waves, this functions as the amount of time the voltage stays high, determining the duty cycle of the wave. This can be calculated by multiplying the period of the wave (**`TIM2->ARR`**) by the new duty cycle percentage (**`new_duty / 100`**). Once again, one is subtracted from the final value.
>
> - **`TIM2->CCR1 = ((TIM2->ARR * new_duty) / 100) - 1;`**
>   - Note: this calculation is only done for square waves

Additionally, once the values of these registers have been changed, the timers must be reinitialized:

- **`TIM2->EGR |= (TIM_EGR_UG);`**

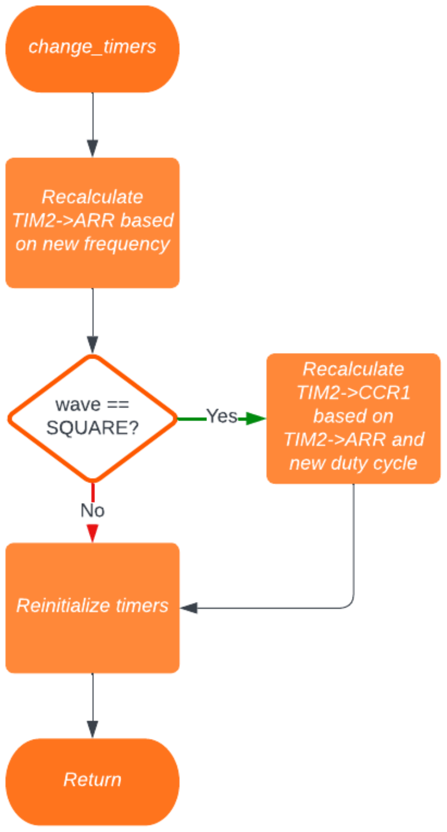

change_timers
Recalculate TIM2->ARR based on new frequency
wave == SQUARE?
Yes
Recalculate TIM2->CCR1 based on TIM2->ARR and new duty cycle
No
Reinitialize timers
Return

Figure 8. `change_timers()` flowchart

`construct_waveform`

This function re-creates **`wave_array`** based on the current waveform and frequency. First, the memory for **`wave_array`** is freed, so the array can be re-initialized. Then, **`array_size`** is recalculated using the same formula mentioned above (**`NUM_WRITES / freq`**), representing the number of DAC writes for the given frequency. The memory for **`wave_array`** is then reallocated using the new **`array_size`**, and the array is populated with new voltage values (using a for-loop). The voltage values for each of the waveforms are calculated as follows:

`SINE:` A sine wave can be modeled by the standard equation $y = A\sin(\omega t) + C$, where A is amplitude, ω is frequency, t is time, and C is the offset. In this case:

- y is the output voltage, **`wave_array[i]`**.

- A is **`1500`**. A peak-to-peak voltage of 3.0V gives a 1.5 V amplitude.
- t is **`i/array_size`**. Similar to how time increases, the **`i`** initialized in the for-loop also increases, allowing the output voltage to change with respect to **`i`**. The variable **`i`** is divided by **`array_size`** to scale the value based on the current frequency.
- C is **`1500`**, representing the 1.5 V (or 1500 mV) DC bias.
- The frequency ω does not need to be accounted for here, as the **`array_size`** was already recalculated based on the current frequency. However, the value inside the sine function must be multiplied by 2π to convert Hertz into radians per second (ω = 2πf).
- The final equation is:
    - **`wave_array[i] = 1500*sin(2*PI*((float) i / array_size))+1500;`**

`SAWTOOTH:` A sawtooth wave is simply a linear slope that ascends to the peak voltage, then drops back to zero. To model this, a variable **`val`** is initialized to 0 before the for-loop, representing the starting voltage of a sawtooth and triangle wave. Within the loop, **`wave_array[i]`** is set to **`val`**, then **`val`** is incremented by **`V_CHANGE*freq`**. In this case, **`freq`** represents the slope of the sawtooth wave.

`TRIANGLE:` Calculating the triangle wave is similar to the sawtooth wave, except that it has to ascend to the peak voltage and descend back to zero volts in the same period. This means that the slope of the triangle wave is twice that of a saw wave of the same frequency. For this reason, **`val`** is incremented by **`2*V_CHANGE*freq`**. Additionally, the voltage must begin to descend once half the period is reached, so once **`i`** equals **`array_size/2`**, **`val`** is decremented by **`2*V_CHANGE*freq`**.

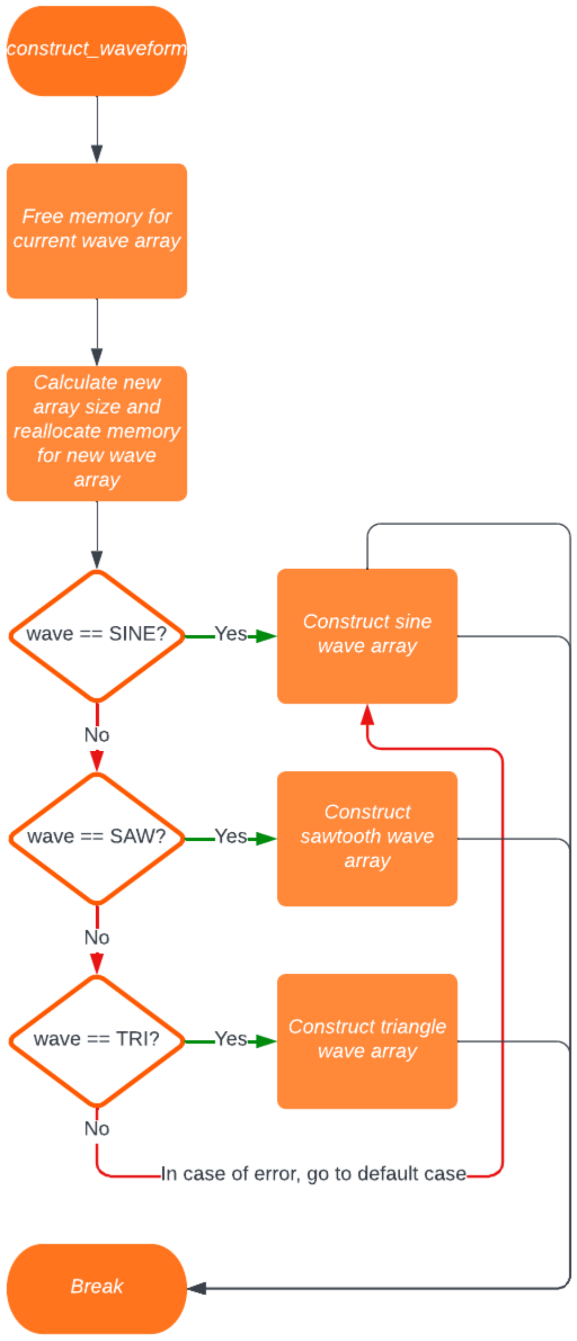

construct_waveform
Free memory for current wave array
Calculate new array size and reallocate memory for new wave array
wave == SINE?
Yes
Construct sine wave array
No
wave == SAW?
Yes
Construct sawtooth wave array
No
wave == TRI?
Yes
Construct triangle wave array
No
In case of error, go to default case
Break

Figure 9. **`construct_waveform()`** **flowchart**

`TIM2_IRQHandler`

The IRQHandler for TIM2 runs continuously during program execution. This function checks if the timer count reaches the period of the wave,

`TIM2->ARR`, or the time in which the output voltage should be incremented to the next value in the waveform array, **`TIM2->CCR1`**. This can be done by and-ing the status register with the respective interrupt flags:

```
if (TIM2->SR & TIM_SR_UIF)
```

The first if-block represents the beginning of a new waveform period. Once the count reaches the period of the wave, the global variable **`curr_volt`** is reset back to zero (**`V_LOW`**), as well as the global variable **`count`** that references the waveform array. If the wave is not a square wave, **`TIM2->CCR1`** is set to **`TIM2->ARR / array_size`**. This calculation represents the time it takes for one DAC write, given the frequency. This way, the DAC will write exactly **`array_size`** amount of times, ensuring we output each value of the waveform array. If the wave *is* a square wave, **`TIM2->CCR1`** remains unchanged, as it directly determines the duty cycle of the square wave. Finally, the ARR interrupt flag is set back to zero in the status register so it can be re-set later in the program execution.

The else if-block represents the instance in which the output voltage changes. Once the count reaches **`TIM2->CCR1`**, **`count`** is incremented so the next voltage value can be outputted. If the wave is not a square wave, **`curr_volt`** is set to **`wave_array[count]`**. Additionally, **`TIM2->CCR1`** increments by the value of itself, so the timer count can reach **`TIM2->CCR1`** again and re-increment the output voltage. If the wave *is* a square wave **`curr_volt`** is set to **`V_PEAK`**, as there is only one possible non-zero value of a square wave. The value of **`TIM2->CCR1`** once again remains unchanged. Finally, the CCR interrupt flag is set back to zero in the status register so it can be re-set later in the program execution.

After both if-blocks, the output voltage is converted then written to the DAC.

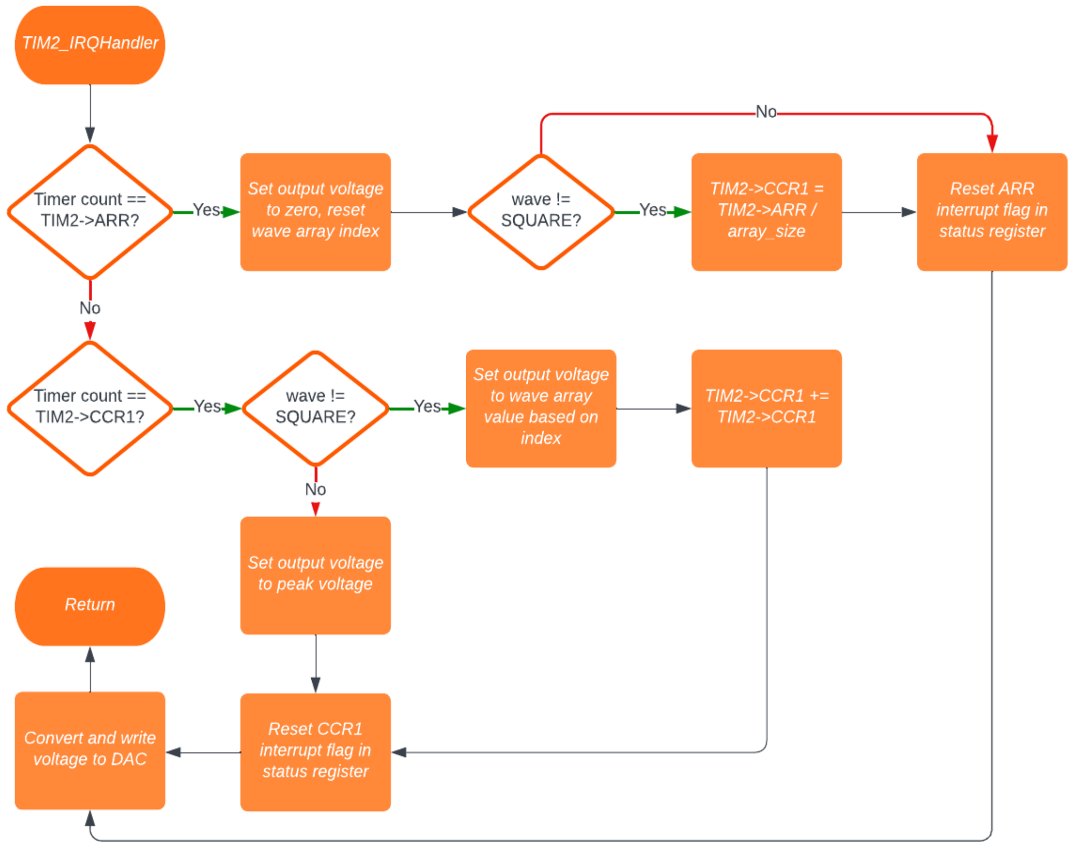

TIM2_IRQHandler
Timer count == TIM2->ARR?
Yes
No
Set output voltage to zero, reset wave array index
wave != SQUARE?
Yes
No
TIM2->CCR1 = TIM2->ARR / array_size
Reset ARR interrupt flag in status register
Timer count == TIM2->CCR1?
Yes
wave != SQUARE?
Yes
No
Set output voltage to wave array value based on index
TIM2->CCR1 += TIM2->CCR1
Set output voltage to peak voltage
Return
Convert and write voltage to DAC
Reset CCR1 interrupt flag in status register

Figure 10. `TIM2_IRQHandler()` flowchart

4. Keypad Functions

`Keypad_Init`

This function configures the pins on the 4x4 keypad for General Purpose Input/Output (GPIO). On the board, pins PB4-PB7 are used for the column pins, and pins PC4-PC7 are used for the row pins. The rows are configured as follows:

> Input mode: Allows the rows to turn high once pressed, so an input key can be detected.

Pull down resistor: Initializes the row pins as low, so the respective row turns high once a key is pressed.

The column pins are configured as follows:

Output mode: Allows the column pins to constantly output high, which makes the logic for detecting a keypress using columns and rows more straightforward.

The column pins are then turned on, to be utilized in the **`getKey()`** function.

getKey

This function will return the integer value of the key pressed on the keypad. If no key is pressed, the function returns -1. The STAR key corresponds to 14, and the HASHTAG key corresponds to 15 (So keys A, B, C, and D could correspond to their hex to decimal conversions of 10, 11, 12, and 13). These integer values are stored in a two-dimensional array, with their index corresponding to their physical location on the keypad.

The function begins by checking if the rows input register is zero. If it is, then -1 is returned, indicating no key press. Otherwise, both a row counter and a column counter are initialized to zero, and the columns are switched off. Each column pin is iterated over, switching the respective column on and checking if any of the rows in that column are switched on. If a row is switched on, the function returns **`keypad_matrix[row_num][col_num]`**, which represents the keypress associated with the current row count and column count. If the row is not switched on, the row count is incremented. If none of the rows associated with that column are on, then the column counter is incremented, indicating that none of the keys in that column were pressed. It is important to note that before the function can return a value, the columns must be switched back on so that the logic of this function can be reused.

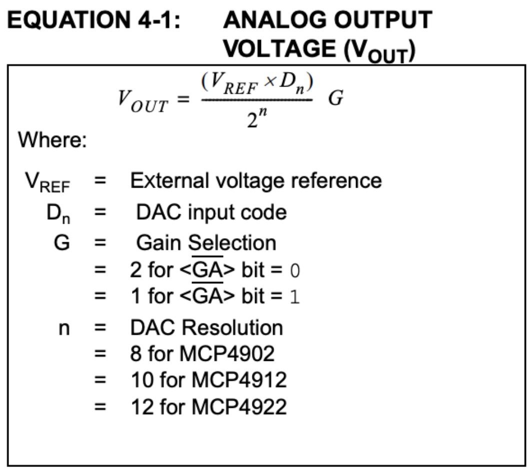
4. DAC Functions

`DAC_Init`

This function configures pins PA4, PA5, and PA7 for alternate function GPIO and SPI. The alternate function for PA4 is Chip Select (CS/NSS), PA5 is Serial Clock (SCK), and PA7 is PICO (Peripheral In, Controller Out). [3]

`DAC_volt_conv`

This function takes in an input voltage (in mV) as a parameter, and returns the converted voltage value, prepended with a **`DAC_HEADER`**. First, the voltage value is multiplied by a conversion factor, which is calculated below:



EQUATION 4-1: ANALOG OUTPUT VOLTAGE (V_{OUT})

$$V_{OUT} = \frac{(V_{REF} \times D_n)}{2^n}\ G$$

Where:

V_{REF}	=	External voltage reference
D_n	=	DAC input code
G	=	Gain Selection
	=	2 for <$\overline{GA}$> bit = 0
	=	1 for <$\overline{GA}$> bit = 1
n	=	DAC Resolution
	=	8 for MCP4902
	=	10 for MCP4912
	=	12 for MCP4922

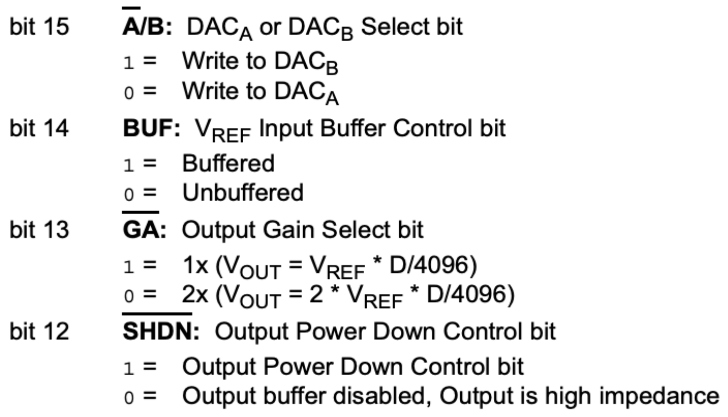
Figure 11. Calculation of V_{OUT} [1]

In this case, D_n is the conversion factor. G = 1 (gain value of 1), V_{REF} is the maximum output voltage of the MCU (3300), and n is 12 (using the MCP4922 DAC). Solving for D_n, the resulting equation is:

$$D_n = \frac{2^{12}}{3300} = 1.24$$

This number was adjusted slightly in testing to yield more accurate voltage values.

Next, the converted voltage is checked to see if it exceeds the maximum output voltage of the MCU. If it does, then the voltage is set equal to 3300. Finally, the 12-bit voltage value is prepended with **`DAC_HEADER`**, which is calculated as follows:



bit 15	**$\overline{\text{A}}$/B:** DAC$_A$ or DAC$_B$ Select bit	
	1 =	Write to DAC$_B$
	0 =	Write to DAC$_A$
bit 14	**BUF:** V$_{REF}$ Input Buffer Control bit	
	1 =	Buffered
	0 =	Unbuffered
bit 13	**$\overline{\text{GA}}$:** Output Gain Select bit	
	1 =	1x (V$_{OUT}$ = V$_{REF}$ * D/4096)
	0 =	2x (V$_{OUT}$ = 2 * V$_{REF}$ * D/4096)
bit 12	**$\overline{\text{SHDN}}$:** Output Power Down Control bit	
	1 =	Output Power Down Control bit
	0 =	Output buffer disabled, Output is high impedance

Figure 12. MSB configuration of DAC data [1]

This this case, DAC$_A$ is being written to (this is arbitrary), the Input Buffer Control bit is unbuffered, the Output Gain Select bit is 1 (because the output should have a gain of 1), and the Output Power Down Control bit is 1 (because a voltage value should be constantly outputted). Altogether, these bits combine to be 0011, which is 0x3 in hex. These bits must be prepended to the 12-bit data transmitted from the MCU, combining to be 16 bits.

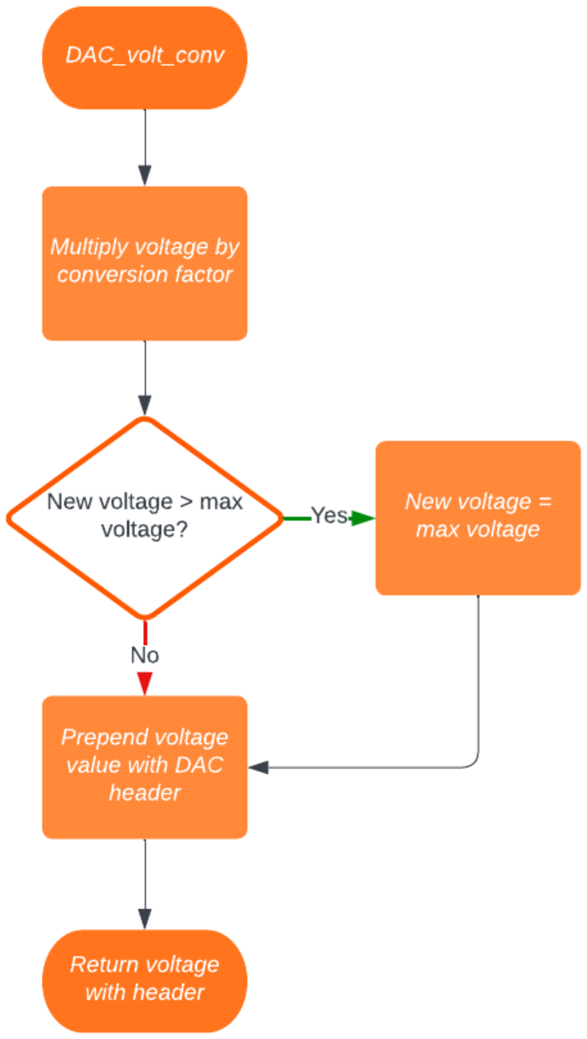

DAC_volt_conv
Multiply voltage by conversion factor
New voltage > max voltage?
Yes
New voltage = max voltage
No
Prepend voltage value with DAC header
Return voltage with header

Figure 13. `DAC_volt_conv()` flowchart

`DAC_write`

This function takes the prepended voltage value above as an input, and writes the value to the SPI data register (**`SPI1->DR`**). This is done in one line of code.

Documents Referenced

[1] “MCP4902/4912/4922 Data Sheet” Microchip Technology Inc., Accessed: Apr. 30, 2023 [Online]. Available: https://ww1.microchip.com/downloads/en/devicedoc/22250a.pdf

[2] P. Hummel and J. Gerfen, “STM32 Lab Manual,” Google Docs, Accessed: Apr. 30, 2023 [Online]. Available: https://docs.google.com/document/d/1Btl--IQGtYRRn8naFpLwn64Av9y7no5OkXmoK1pFN4g/edit#heading=h.z6v22gj9iqm0

[3] “ST32L476xx Data Sheet,” ST.com., Accessed: Apr. 30, 2023 [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l476rg.pdf

[4] “ST32L476 Reference Manual,” ST.com., Accessed: Apr. 30, 2023 [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxxstm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcusstmicroelectronics.pdf.

```c
/* main.c */

/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "keypad.h"
#include "dac.h"
#include <math.h>
#include <stdlib.h>

#define SINE 6 // Sine wave keypress
#define TRI 7 // Tri wave keypress
#define SAW 8 // Saw wave keypress
#define SQUARE 9 // Square wave keypress
#define CLK_SPEED 20000000 // 20 MHz clock
#define V_PEAK 3000 // peak to peak voltage, 3.0 V
#define V_LOW 0 // low voltage, 0 V
#define DEBOUNCE 4000 // debounce delay
#define V_CHANGE 26 // Voltage change per increment
#define NUM_WRITES 116 // number of writes per waveform period
#define PI 3.141592

void SystemClock_Config(void);
void change_timers(int8_t new_freq, int8_t new_duty);
void construct_waveform(void);

int8_t wave = SQUARE, freq = 1, duty = 50; // initial waveform: square wave, 100 Hz, 50% duty cycle
uint32_t curr_volt = 0; // current voltage being written and displayed

typedef enum { // define FSM states
    DISPLAY, CHANGE_WAVE, CHANGE_FREQ, CHANGE_DUTY
} state_var_type;

uint16_t *wave_array; // stores all voltages within one period
uint32_t array_size = 116; // beginning array size, NUM_WRITES / freq
uint16_t count = 0; // current index of wave_array

int main(void) {
    HAL_Init();
    SystemClock_Config();
    Keypad_Init();
    DAC_Init();

    int8_t keyPress = NO_KEY_PRESS; // Default key press value

    RCC->APB1ENR1 |= RCC_APB1ENR1_TIM2EN; // enable TIM2 clock

    TIM2->CR1 = (TIM_CR1_ARPE); // Enable Auto Reload/Preload (ARR)

    change_timers(freq, duty); // Initialize timers with 100 Hz, 50% duty cycle

    // Enable interrupt flags
    TIM2->DIER |= (TIM_DIER_UIE); // ARR flag
    TIM2->DIER |= (TIM_DIER_CC1IE); // CCR1 flag

    // enable interrupts in NVIC
    NVIC->ISER[0] = (1 << (TIM2_IRQn & 0x1F));

    // Turn on timer
    TIM2->CR1 |= TIM_CR1_CEN;

    construct_waveform(); // create 100 Hz, 50% duty square wave
    __enable_irq();

    state_var_type state = DISPLAY; // default FSM state

    while (1) {

        // FSM
        switch (state) {

        // Default state, func gen should always display a waveform and look for next key press
        case DISPLAY:
            if ((keyPress = getKey()) != NO_KEY_PRESS) { // wait for key to be pressed
                // Switch states depending on the function of the key press
                if (keyPress >= 1 && keyPress <= 5) {
                    state = CHANGE_FREQ;
                } else if (keyPress >= 6 && keyPress <= 9) {
                    state = CHANGE_WAVE;
                } else if (keyPress == 0 || keyPress == STAR || keyPress == HASHTAG) {
                    state = CHANGE_DUTY;
                }
            }
            // wait for user to release button
            while (getKey() != NO_KEY_PRESS);
            for (int i = 0; i < DEBOUNCE; i++); // debounce delay
            break;

        case CHANGE_WAVE:
            wave = keyPress;

            // recalc values of wave_array with new wave
            construct_waveform();
            state = DISPLAY; // look for next key press
            break;

        case CHANGE_FREQ:
            freq = keyPress;

            // recalc ARR and CCR1 with new freq
            change_timers(freq, duty);

            // recalc values of wave_array with new freq
            construct_waveform();
            state = DISPLAY; // look for next key press
            break;

        case CHANGE_DUTY:

            // Check if the key pressed is STAR, 0, or HASHTAG
            switch (keyPress) {
            case STAR:
                duty -= 10; // decrement duty cycle by 10%

                // Ensure duty cycle does not go below 10%
                if (duty < 10) {
                    duty = 10;
                }
                break;

            case 0:
                duty = 50; // set duty cycle to 50%
                break;

            case HASHTAG: // increment duty cycle by 10%
                duty += 10;

                // Ensure duty cycle does not exceed 90%
                if (duty > 90) {
                    duty = 90;
                }
                break;

            default: // In case of error, keep the same duty cycle
                break;
            }

            // recalc ARR and CCR1 based on new duty cycle
            change_timers(freq, duty);
            state = DISPLAY; // look for next key press
            break;

        // In case of error, look for next key press
        default:
            state = DISPLAY;
            break;
        }
    }
}

/* TIMER 2 ISR */
void TIM2_IRQHandler(void) {

    // Once count reaches TIM2->ARR
    if (TIM2->SR & TIM_SR_UIF) {
        curr_volt = V_LOW; // reset current voltage to 0
        count = 0; // reset wave_array index

        // If not a square wave, reset CCR1
        TIM2->CCR1 = (wave != SQUARE) ? TIM2->ARR / array_size : TIM2->CCR1;
        TIM2->SR &= ~(TIM_SR_UIF); // reset interrupt flag
    } else if (TIM2->SR & TIM_SR_CC1IF) {
        curr_volt = (wave != SQUARE) ? wave_array[count] : V_PEAK; // set current voltage to wave_array value
        count++; // increment wave_array index

        // If not a square wave, increment CCR1
        TIM2->CCR1 += (wave != SQUARE) ? (TIM2->ARR / array_size) : 0;
        TIM2->SR &= ~(TIM_SR_CC1IF); // reset interrupt flag
    }
    DAC_write(DAC_volt_conv(curr_volt));
}

/* RECALCULATES ARR AND CCR1 BASED ON FREQUENCY AND DUTY CYCLE */
void change_timers(int8_t new_freq, int8_t new_duty) {
    TIM2->ARR = (CLK_SPEED / (new_freq * 100))+100; // Calculate ARR based on Clock Speed and frequency

    // If square wave, set CCR1 to a constant value
    if (wave == SQUARE) {
        TIM2->CCR1 = ((TIM2->ARR * new_duty) / 100) - 1; // Calculate CRR based off of ARR and duty cycle
    }

    // Must be done any time a timer value is changed
    TIM2->EGR |= (TIM_EGR_UG); // Update interrupt event

}

/* CALCULATES THE VALUES OF WAVE_ARRAY BASED ON CURRENT WAVEFORM AND FREQUENCY */
void construct_waveform(void) {
    free(wave_array); // free the memory of current wave_array
    array_size = NUM_WRITES / freq; // calculate new wave_array size
    wave_array = (uint16_t*) malloc(sizeof(uint16_t) * array_size); // realloc wave_array with new size
    uint32_t val = 0; // initial value of tri and saw waves

    // Calculate wave_array based on current waveform
    switch (wave) {
    case SINE:
        for (uint32_t i = 0; i < array_size; i++) {
            // Standard sine wave equation, 1500 mV amplitude and DC bias
            wave_array[i] = 1500 * sin(2 * PI * ((float) i / array_size)) + 1500;
        }
        break;

    case SAW:
        for (uint32_t i = 0; i < array_size; i++) {
            wave_array[i] = val;
            val += V_CHANGE * freq; // increment val by slope of saw wave
        }
        break;

    case TRI:
        for (uint32_t i = 0; i < array_size; i++) {
            wave_array[i] = val;

            if (i < array_size / 2) {
                val += 2 * V_CHANGE * freq; // slope increments twice as fast

            // If half the period has been reached, decrement voltage instead
            } else {
                val -= 2 * V_CHANGE * freq;
            }
        }
        break;

    // In case of error, output a sine wave
    default:
        for (uint32_t i = 0; i < array_size; i++) {
            wave_array[i] = 1500 * sin(2 * PI * ((float) i / array_size)) + 1500; // upon error, create sine wave
        }
        break;

    }

}
/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void) {
    RCC_OscInitTypeDef RCC_OscInitStruct = { 0 };
    RCC_ClkInitTypeDef RCC_ClkInitStruct = { 0 };

    /** Configure the main internal regulator output voltage
     */
    if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1)
            != HAL_OK) {
        Error_Handler();
    }

    /** Initializes the RCC Oscillators according to the specified parameters
     * in the RCC_OscInitTypeDef structure.
     */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_MSI;
    RCC_OscInitStruct.MSIState = RCC_MSI_ON;
    RCC_OscInitStruct.MSICalibrationValue = 0;
    RCC_OscInitStruct.MSIClockRange = RCC_MSIRANGE_6;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_MSI;
    RCC_OscInitStruct.PLL.PLLM = 1;
    RCC_OscInitStruct.PLL.PLLN = 20;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
    RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
    RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
        Error_Handler();
    }

    /** Initializes the CPU, AHB and APB buses clocks
     */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK
            | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV2;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK) {
        Error_Handler();
    }
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
 * @brief  This function is executed in case of error occurrence.
 * @retval None
 */
void Error_Handler(void) {
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    __disable_irq();
    while (1) {
    }
    /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
```

```c
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

```

```c
/*
 * keypad.h
 *
 *  Created on: Apr 12, 2023
 *      Author: Noah Masten, Cole Costa
 */

#include "main.h"
#ifndef SRC_KEYPAD_H_
#define SRC_KEYPAD_H_

#define ROW_PORT GPIOC
#define COL_PORT GPIOB
#define ROW_MASK (GPIO_IDR_ID4 | GPIO_IDR_ID5 | GPIO_IDR_ID6 | GPIO_IDR_ID7) // mask that combines the four row pins
#define LETTER_A 10
#define LETTER_B 11
#define LETTER_C 12
#define LETTER_D 13
#define STAR 14
#define HASHTAG 15
#define NO_KEY_PRESS -1

int32_t const keypad_matrix[4][4] = {{1, 2, 3, LETTER_A},
                                    {4, 5, 6, LETTER_B},
                                    {7, 8, 9, LETTER_C},
                                    {STAR ,0, HASHTAG, LETTER_D}};

void Keypad_Init(void);
int32_t getKey(void);

#endif




```

```c
/*
 * keypad.c
 *
 *  Created on: Apr 12, 2023
 *      Author: Noah Masten, Cole Costa
 */

#include "main.h"
#include "keypad.h"

/*
 * CONFIGURES KEYPAD PINS FOR GPIO
 * PC4 - PC7 FOR ROWS
 * PB4 - PB7 FOR COLS
 */
void Keypad_Init(void) {

    // Enable GPIOB and GPIOC clocks
    RCC->AHB2ENR   |=  (RCC_AHB2ENR_GPIOBEN);
    RCC->AHB2ENR   |=  (RCC_AHB2ENR_GPIOCEN);

    // Configure rows for input
    ROW_PORT->MODER &= ~(GPIO_MODER_MODE4 | GPIO_MODER_MODE5 | GPIO_MODER_MODE6 | GPIO_MODER_MODE7);

    // Configure cols for output
    COL_PORT->MODER &= ~(GPIO_MODER_MODE4 | GPIO_MODER_MODE5 | GPIO_MODER_MODE6 | GPIO_MODER_MODE7);
    COL_PORT->MODER |= (GPIO_MODER_MODE4_0 | GPIO_MODER_MODE5_0 | GPIO_MODER_MODE6_0 | GPIO_MODER_MODE7_0);

    // Pull down for rows
    ROW_PORT->PUPDR &= ~(GPIO_PUPDR_PUPD4 | GPIO_PUPDR_PUPD5 | GPIO_PUPDR_PUPD6 | GPIO_PUPDR_PUPD7);
    ROW_PORT->PUPDR |= (GPIO_PUPDR_PUPD4_1 | GPIO_PUPDR_PUPD5_1 | GPIO_PUPDR_PUPD6_1 | GPIO_PUPDR_PUPD7_1);

    // No PUPD for cols
    COL_PORT->PUPDR &= ~(GPIO_PUPDR_PUPD4 | GPIO_PUPDR_PUPD5 | GPIO_PUPDR_PUPD6 | GPIO_PUPDR_PUPD7);

    // No push-pull for cols
    COL_PORT->OTYPER &= ~(GPIO_OTYPER_OT4 | GPIO_OTYPER_OT5 | GPIO_OTYPER_OT6 | GPIO_OTYPER_OT7);

    // Low speed
    COL_PORT->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED4 | GPIO_OSPEEDR_OSPEED5 | GPIO_OSPEEDR_OSPEED6 | GPIO_OSPEEDR_OSPEED7);

    // Turn on cols
    COL_PORT->BSRR = (GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);

}

/* RETURNS THE INTEGER VALUE OF THE KEY PRESSED BY THE USER */
int32_t getKey(void) {
    int rows = (ROW_PORT->IDR & ROW_MASK); // get current state of row pins
    if (rows == 0) { // No key is pressed
        return NO_KEY_PRESS;
    } else { // Iterate over columns and rows to find the exact key pressed
        int row_num = 0;
        int col_num = 0;
        COL_PORT->BRR = (GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7); // turn cols off
        for (int col = GPIO_PIN_4; col <= GPIO_PIN_7; col <<= 1) { // Start with PIN 1, iterate through PIN 4
            COL_PORT->BSRR = col;
            int checkRows = (ROW_PORT->IDR & ROW_MASK); // See if any of the rows in this column are pressed
            if (checkRows != 0) { // If a row is pressed, figure out which row
                for (int row = GPIO_PIN_4; row <= GPIO_PIN_7; row <<= 1) {
                    if (row==checkRows) { // If current row corresponds to the row pressed, return the keypress
                        COL_PORT->BSRR = (GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7); // reset cols to high
                        return keypad_matrix[row_num][col_num]; // return keypress
                    }
                    row_num++; // increment row
                }
            }
            col_num++; // increment col
            COL_PORT->BRR =  (GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7); // turn cols back off
        }
    }
    COL_PORT->BSRR = (GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7); // reset cols to high
    return NO_KEY_PRESS; // if keypress is missed

}
```

```c
/*
 * dac.h
 *
 *  Created on: Apr 24, 2023
 *      Author: noahmasten
 */

#ifndef SRC_DAC_H_
#include "main.h"
#define SRC_DAC_H_
#define DAC_HEADER 0x3000 // Calculated with MCP4922 data sheet
#define DAC_MASK 0xFFF // ensures voltage won't exceed 3.3 V
#define VDD 3300 // 3.3 V
#define SCALAR 1.245 // Voltage conversion factor

void DAC_Init(void);
void DAC_write(uint16_t data);
uint32_t DAC_volt_conv(uint32_t voltage);

#endif /* SRC_DAC_H_ */

```

```c
/*
 * dac.c
 *
 *  Created on: Apr 24, 2023
 *      Author: noahmasten
 */

#include "dac.h"
#include "main.h"

/* CONFIRURE PA4, PA5, PA7 FOR GPIO AND SPI
 * PA4 FOR CHIP SELECT (CS/NSS)
 * PA5 FOR SERIAL CLOCK (SCK)
 * PA7 FOR PERIPHERAL IN/CONTROLLER OUT
 */
void DAC_Init(void) {

    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN; // Enable GPIOA clock
    RCC->APB2ENR |= RCC_APB2ENR_SPI1EN; // Enable SPI1 clock

    // clear GPIO mode
    GPIOA->MODER &= ~(GPIO_MODER_MODE4|
                      GPIO_MODER_MODE5|
                      GPIO_MODER_MODE7);

    // set GPIO mode to alternate function
    GPIOA->MODER |= (GPIO_MODER_MODE4_1)|
                    (GPIO_MODER_MODE5_1)|
                    (GPIO_MODER_MODE7_1);

    // set alternate function for GPIO to SPI
    GPIOA->AFR[0] &= ~(GPIO_AFRL_AFSEL4 |
                       GPIO_AFRL_AFSEL5 |
                       GPIO_AFRL_AFSEL7);

    GPIOA->AFR[0] |= (5<< GPIO_AFRL_AFSEL4_Pos)|
                     (5<< GPIO_AFRL_AFSEL5_Pos)|
                     (5<< GPIO_AFRL_AFSEL7_Pos);

    // OTYPER -> push pull
    GPIOA->OTYPER &= ~(GPIO_OTYPER_OT4
            | GPIO_OTYPER_OT5
            | GPIO_OTYPER_OT7);

    // OSPEED -> low speed
    GPIOA->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED4
            | GPIO_OSPEEDR_OSPEED5
            | GPIO_OSPEEDR_OSPEED7);

    // PUPDR -> no pull up pull down
    GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPD4
            | GPIO_PUPDR_PUPD5
            | GPIO_PUPDR_PUPD7);

    // initialize SPI
    SPI1->CR1 = (SPI_CR1_MSTR);

    SPI1->CR2 = (SPI_CR2_DS)|
                SPI_CR2_NSSP|
                SPI_CR2_ERRIE;

    // enable SPI
    SPI1->CR1 |= (SPI_CR1_SPE);

}

/* WRITE VOLTAGE TO THE DAC */
void DAC_write(uint16_t data) {
    SPI1->DR = data; // write voltage to SPI data register
}

/* CONVERTS VOLTAGE AND PREPENDS DAC_HEADER SO VOLTAGE CAN BE WRITTEN TO DAC */
uint32_t DAC_volt_conv(uint32_t voltage) {

    uint16_t dacData = 0;
    voltage *= SCALAR; // Adjust voltage using conversion factor

    // check if voltage exceeds max (VDD)
    if (voltage > VDD) {
        dacData = VDD & DAC_MASK; // max allowed voltage is 3.3 V
    } else {
        dacData = voltage & DAC_MASK; // else, keep current voltage
    }

    return dacData | DAC_HEADER; // return value with header of 0x3
}

```