Stepper Motor Synth

Created by: Hunter Jans and Abe Jaeger Mountain

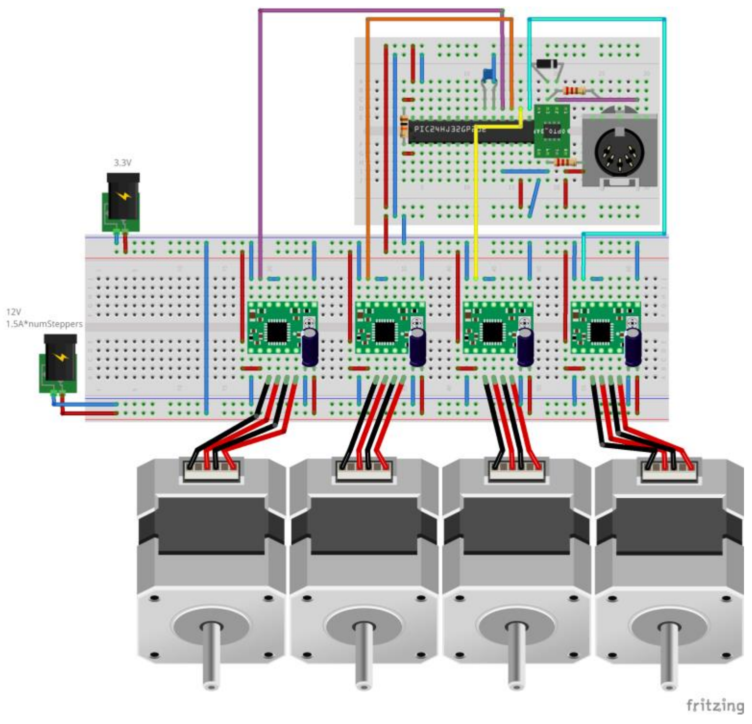
For further documentation and file access, visit the project GitHub Repository

Brief Overview of Project:

The stepper motor synth is a project created with the goal of being able to play music utilizing only stepper motors to generate sound. Under the hardware limitations of having to use a PIC24FJ64GA002 microcontroller, two ‘sister’ libraries were developed in tandem. The first library “MIDI_Device.h” was created to handle the initialization of output channels. This library is capable of initializing up to 4 different channels at a time, each of which is capable of playing tones at different frequencies. The second library “PIC24_MIDI_reciever.h” was created to handle the reception and interpretation of MIDI data. This library allows the device to receive an input via a MIDI cable using UART serial communication and execute the correct play/stop response. When combined with each other these libraries allow the user to turn the PIC24 into a MIDI output device capable of live MIDI playback using either a keyboard or a software such as Anvil Studio to play MIDI audio.

Hardware Requirements:

- 12V / 6A capable power supply
- 4x AN4988 Stepper Motor Drivers
- 4x 100uF capacitors
- 4x Nema 17 Stepper Motors
- 1x PIC24FJ64GA002
- 1x Diode (rec. 1N4148)
- 1x Female MIDI Port
- 1x 6N138 Optocoupler



Library Documentation:

MIDI Device Library:

Classes/Structs/Typedef

- **Timer**
    - volatile uint16_t* TxCON
    - volatile uint16_t* PRx
    - volatile uint16_t* TMRx
    - volatile uint16_t* OCxCON
    - volatile uint16_t* OCxRS

  Timer is a structure containing pointer variables that can be assigned to the addresses of on-board timers. Additional pointers are provided for optional use with timers that can be connected to output compare registers. This library contains predefined static timer objects for timers 1-5 on the PIC24FJ64GA002 named as Tx for which x is the timer number. Additionally, the library has compiled these 4 timers into an array such that the user can utilize an index-based iterative access method.

- **Channel**

  Variables:
    - char avail
        - Internally referenced flag denoting whether or not this channel is currently in use
    - Timer timer
        - reference to a timer object such that this channels functions can modify said timers values
    - uint16_t pin
        - Used to store the pin that this channel outputs to
    - uint16_t ocr
        - Internally referenced to determine whether the channel uses output compare or not
    - uint16_t freq
        - Used to store the frequency currently being played on this channel
    - Channel * this
        - A pointer included to more closely replicate the C++ class structure

  Functions:

- ➢ ChannelFunc_1Arg setPin
- ➢ ChannelFunc_1Arg setFreq
- ➢ ChannelFunc_Null stop
- ➢ ChannelFunc_Null start

The channel functions are declared using predefined types for pointer functions with 1 or no arguments. These pointer functions themselves take an additional argument in a pointer to the channel that the user wishes to modify. This is done so that the channel object variables can be modified while in the scope of the channel functions. These pointer functions can then be assigned to any function so long as that function possesses the same argument types as the pointer function.

A dedicated function is included in the library to automatically handle the definition of channel objects. “initChannel()” is further detailed within its function documentation.

The channel object itself is intended to handle the frequency control of an audio output device connected to a specific pin.

Functions

- ❖ **void initChannel(Channel *channel, uint16_t pin)**
  - ➢ channel
    - ■ A pointer to the channel to be modified
  - ➢ pin
    - ■ An integer argument in range 0-15 (excluding 4)

The initChannel function is capable of dynamically initializing up to 4 different channel objects that are then internally stored and used in this and its sister library. The function first checks the pin argument provided to ensure that it is valid, if not the function leaves the given channel uninitialized. The channel assigns values to all pointer functions and calls the setPin function to configure the necessary registers to use the desired pin.This function also connects the channel to a timer based on the number of already initialized channels. Finally, if necessary, the function enables timer interrupts for the attached timer.

NOTE: If 4 channels have already been initialized the initChannel function will still initialize the provided channel; however the oldest channel stored within the libraries will be overwritten with the newest one, essentially breaking the functionality of the oldest defined channel.

- **void setPin(Channel *this, uint16_t pin)**
    - this
        - A pointer to the channel to be modified
    - pin
        - An integer representing the pin to be configured for the given channel functionality

The setPin function handles any and all reconfigurations of pins during initialization or reconfiguration. The function utilizes the ‘ocr’ variable of the given channel to determine the type of timing system it uses and configures either TRISB or PPS for the desired pin based on this. If a pin had already been set and the user wishes to change the pin for a channel, this function will reset the configuration of the previous pin before configuring the new pin.

- **void start(Channel *this)**
    - this
        - A pointer to the channel to be modified
        -

The start function simply starts the timer connected to the given channel and starts audio output in the process.

- **void stop(Channel *this)**
    - this
        - A pointer to the channel to be modified

The stop function stops the timer connected to the given channel and stops audio output in the process.

- **void setFreq(Channel *this, uint16_t freq)**
    - this
        - A pointer to the channel to be modified
    - freq
        - The frequency to set for a channel

The setFreq function automatically calculates the required prescaler and period needed to oscillate the output of the given channel’s pin at the provided frequency. After a frequency value has been passed, the connected channel will start playing that frequency automatically. The setFreq function has a unique case for zero, if a zero is passed as the frequency argument, the function will instead stop playback. This was done to avoid a divide by zero error.

- **double log2(double x)**
  - x
    - The argument for the base-2 log function

The log2 function is simply a base-2 log function that was created out of a necessity for frequency calculations. The function returns the result as a double.

Changing the Channel Limit

The 4 channel limit was set due to the PIC24 device used having only 4 timers with only 2 being capable of connecting to output compare registers. This limit can be changed via a macro definition in the library header “CHANNEL_LIMIT”. Reducing the channel limit requires only a change to this macro, increasing the limit however is untested and thus not recommended. Hypothetically to increase the limit the user needs first select a device with more timers then define a new timer object for the additional timer to the header then add this timer to the array of predefined timer addresses. Next the user would need to copy the timer interrupt enable condition found in the initChannel function for the next ocr value. Finally the user would have to copy the timer interrupt function shown below and modify the interrupt to apply to the newly utilized timer, additionally the user would have to change the index used to access the channel array.

```
void __attribute__((__interrupt__, __auto_psv__)) _T4Interrupt(void)
{
    _T4IF = 0;
    static int toggle = 1;
    int mask = toggle << _channel[2]->pin;
    LATB ^= mask;
    toggle = (toggle + 1) % 2;
}
```

MIDI Reception Library

Functions

- **MIDI_init()**
  The MIDI_init function initializes a 31,250 baud UART communication on pin RB4 and enables interrupts on every character for UART1. This is done to read the transmission of MIDI data.

- **void attachChannel(Channel *this)**
  - this
    - A pointer to a channel object

  The attachChannel function is used to connect channels defined within the sister library to this one, as such this function is called within the initChannel function definition. This function also sets the availability flag for the given channels to 1, indicating that they can be written to.

- **uint16_t getFreq(uint8_t midiNote)**
  - midiNote
    - An 8-bit midi note value that corresponds to a specific frequency

  The getFreq function translates midiNote values to their corresponding frequency by utilizing an array of predefined fundamental frequencies and a power-2 function. This function returns the frequency as a 16-bit unsigned integer.

- **void turnOff(uint8_t note)**
  - note
    - The 8 bit note index received from the MIDI transmission

  The turnOff function is used to stop the playback of a given note. The function first calls the getFreq function to get the frequency translation of the MIDI note. The function then searches the channel array for a channel with a matching frequency. If a matching channel is found, the frequency for that channel is set to 0, stopping playback, and the availability flag for the channel is set to 1.

- **void turnOn(uint8_t note, uint8_t velocity)**
  - note
    - The 8 bit note index received from the MIDI transmission
  - velocity

        - An 8 bit value received from the MIDI transmission that is representative of volume

The turnOn function starts by checking if the velocity argument is zero, if so then the function ends by calling the turnOff function instead. Otherwise the function passes the MIDI note to getFreq to get its frequency translation. Next the function loops through the channel array to check if any of the channels are already playing the frequency (this is done to limit unnecessary use of channels due to the already low count) if a channel is found, a flag value is set to 1. After that, this same flag value is checked, if the value is zero then the function loops through the channel array again to search for the first available channel. If a channel is found, the frequency is passed to that channel and its availability flag is set to 0.

Usage

In order to use the sibling libraries, the user must first add the import statements shown below:

```
#include "MIDI_Device.h"
#include "PIC24_MIDI_receiver.h"
```

Once this is done, the user should declare the desired number of channel objects such that they are accessible from the scope in which the channel functions are called.

Additionally, the user should also include these 3 lines within their setup function:

```
    CLKDIVbits.RCDIV = 0;
    AD1PCFG = 0x9fff;
    TRISB = 0;
```

Basic Usage (only MIDI_Device)

The following is a simple usage case that can be used to verify the user's circuit configuration as it does not require them to have to deal with debugging the MIDI transmission on top of a potentially flawed circuit.

For this case, the user should include the following lines with their import statements:

```
#define FCY 16000000UL
#include <libpic30.h>
```

This will allow the user to use the function __delay_ms(x) to create a blocking delay for x milliseconds.

First, the user should declare any channels they wish to use within their program. Where these channels are declared depends on the location of their respective initChannel functions. If an initChannel function is called within setup(), then the channels must be declared globally and precede the setup function definition. If the initChannel functions are called within main() then the channel declarations need only precede the initChannel calls. An example of each case is shown below along with an example code using the provided declaration method:

Globally Declared Channel

```
Channel channel;
setup() {
    initChannel(&channel, pinNum);
}
```

Channel Declared in Main

```
    Channel channel;
    initChannel(&channel, pinNum);
```

Example Code

```
    int waitFor = 1500
    while(1){
        channel.setFreq(channel.this, 440);
        __delay_ms(waitFor);
        channel.setFreq(channel.this, 880);
        __delay_ms(waitFor);
        channel.setFreq(channel.this, 1320);
        __delay_ms(waitFor);
        channel.setFreq(channel.this, 0);
        __delay_ms(waitFor);
    }
```

The provided example code will cycle through the 3 frequencies 440 Hz, 880 Hz, and 1320 Hz in addition to a stopped playback state. The device will remain in each state for 1.5 seconds.

Advanced Usage (MIDI output device)

The following code is a feature-complete example in which 4 channels are declared and defined before calling the MIDI_Init. This results in the PIC24 interpreting data received via a MIDI connection and automatically playing back up to 4 notes at a time.

NOTE: This code is made with the provided circuit in mind. While the channel pins can easily be changed to the users desired values, the MIDI input pin must remain as RB4

Example Code

```
Channel ch[4];

void setup(void) {
    _RCDIV = 0; //16 MHz frequency
    AD1PCFG = 0xffff; //set all pins to digital
    TRISB = 0;
    initChannel(&ch[0], 6);
    initChannel(&ch[1], 7);
    initChannel(&ch[2], 8);
    initChannel(&ch[3], 9);
}

int main(int argc, char** argv) {
    setup();
    MIDI_init(); //initialize midi receiver and pass the desired
output midiDevice
    while (1) {

    }
    return 0;
}
```