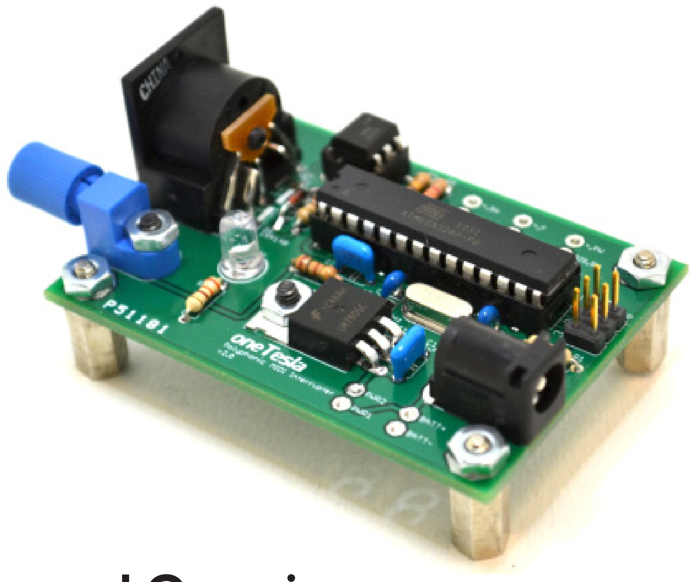
oneTesla
Interrupter Firmware Guide

Contents:



General Overview

The oneTesla MIDI controller is based on the ATMega 328P microcontroller, manufactured by Atmel. At its heart, the hardware on the board is extremely simple - a 5V regulator provides power to the board, a DIN5 MIDI jack connects a MIDI source to the microcontroller through an optocoupler (this is necessary to prevent ground loops between the MIDI source and MIDI receiver), and an optical transmitter (which is simply an LED in a fancy package) sends the output pulses to the Tesla coil.

The interrupter's firmware is written in C++, and is provided as an Arduino project (compatible with the Arduino IDE). Upon reception of a MIDI message, the code calls an appropriate handler function (*callback function*) to process the message.

The code is organized in several files. In general, each chunk of code is split across two files - a `.h` *header file*, which defines constants and provides function definitions to other files, and a `.cpp` file, which contains the actual implementations of the functions.

shared.h and constants.h

shared.h and constants.h contain definitions for global variables and defined constants, respectively. A *global variable* is one that is defined across all files, and is accessible by all functions. They need to be declared in a header file - this is necessary in order to indicate to all program files accessing this variable that it exists. However, they can only be defined once, in a .cpp file.

```
#ifndef __SHARED_H

extern volatile int on_time_1, on_time_2, current_
pitch_1, current_pitch_2, global_scaler_1, global_
scaler_2;
extern volatile long ticks_1, ticks_2;
extern volatile long note_1_period, note_2_period;
extern volatile boolean note_playing_1, note_play-
ing_2;
extern volatile long sustain_time_1, sustain_time_2;
extern volatile boolean sustain, sustaining_1, sus-
taining_2;

#define __SHARED_H
#endif
```

on_time_1 and on_time_2 are the computed ON times of the two currently running notes. current_pitch_1 and current_pitch_2 are the current pitches of the two notes (in MIDI byte format). global_scaler_1 and global_scaler_2 are values from 0-127 which are used to scale the volumes of the notes.
ticks_1 and ticks_2 are two values used internally by the sustain feature. note_1_period and note_2_period are the periods (in microseconds) of notes 1 and 2. note_playing_1 and note_playing_2 are flags which indicate whether notes 1 and 2 are present. sustain_time_1 and sustain_time_2 are internal variables used by the sustain feature. sustain indicates whether the sustain pedal is depressed, and sustaining_1 and sustaining_2 indicate whether notes 1 and 2 are in the "tail" part of a sustain, respectively. Moving on to constants.h:

```
#define LOOKUP_TABLE_SCALE 3
```

This line defines a constant which is used to scale all of the values in the pulse width table. When modifying the code, it is important to remember this value exists! It makes modifying the table values for different sizes of Tesla coil easy.

```
#define MIN_ON_TIME 10
```

This line defines a minimum ON time value (in microseconds). This insures that the pulse widths are never too low (which would make the coil stop producing output). The next few lines should not be messed with, so moving along...

```
#define map_velocity(v) (vel_map[v])
#define map_sustain(t) ((float) t / (float) SUSTAIN_
TIME)
```

These two change the mapping curves for velocity and the shape of the sustain envelope, respectively. v is a byte from 0-127 (the higher the velocity, the larget the value). t is the time into the sustain curve, in microseconds.

`shared.h` and `constants.h`

```
#define SUSTAIN_TIME 750000
#define ON_TIME_ARRAY_LENGTH 20
static int on_times[] = ...
```

`SUSTAIN_TIME` defines the length of the sustain period in microseconds. `ON_TIME_ARRAY_LENGTH` defines the length of the on-times array (necessary since C++ arrays don't store their own lengths). `on_times[]` is an array that stores the on-times of the coil for different frequencies of note, measured in micro-seconds.

```
static int vel_map[] = ...
```

Finally, this line defines a lookup table for velocity mapping.

interrupter.ino

`interrupter.ino` contains the main program code. It does two things: set up the microntroller configuration registers and enter an infinite main loop which handles MIDI messages.

```
void setup() {
  DDRD  |=  (1 << 2);
  PORTD &= ~(1 << 2);
  DDRD &= ~(1 << 3);
```

The first statement sets the transmitter pin to output mode. The second sets its value to zero. The third sets the mode switch pin to input mode.

```
setupADC();
setupTimers();
```

These two lines call functions (defined in `util.cpp`) that set up the ADC and timers.

```
MIDI.begin(MIDI_CHANNEL_OMNI);
MIDI.setHandleNoteOn(HandleNoteOn);
MIDI.setHandleNoteOff(HandleNoteOff);
MIDI.setHandleStop(HandleStop);
MIDI.setHandleContinue(HandleContinue);
MIDI.setHandleControlChange(HandleControlChange);
MIDI.setHandleSystemExclusive(HandleSystemExclusive);
MIDI.setHandlePitchBend(HandlePitchBend);
MIDI.setHandleSystemReset(HandleSystemReset);
```

These statements set up the MIDI library. The first line initiates MIDI communications. The next few lines set the handlers for each type of MIDI message. To add a new handler, declare a new function somewhere that takes the appropriate arguments (look in `MIDI.h` for the argument types), and then attach it to the MIDI class with:

```
MIDI.setHandleMessageType(messageHandlerName);
```

```
if (!(PIND & (1 << 3))) fixedLoop();
```

Finally, this line reads the pin the toggle switch is connected to, and enters the fixed-frequency loop if that pin is low. `interrupter.ino` also contains two loops - a MIDI loop and a fixed-frequency loop. The MIDI loop is very simple:

```
void loop() {
  MIDI.read();
}
```

The loop simply repeatedly calls `MIDI.read()`. Nothing else is needed - whenever a MIDI message is received, the MIDI library automatically calls the appropriate callback function we attached in `setup()`.

The fixed-frequency loop is a little more complex. Briefly, it reads the values set by the two potentiometers every iteration, then turns the optical transmitter ON and OFF for appropriate amounts of time using delays.

```
void fixedLoop() {
  while (1) {
    ADMUX &= ~(1 << MUX0);
    _delay_us(300);
    uint8_t val_2 = ADCH;
```

This first sets the value of the `ADMUX` register. This selects which ADC pin to read from. Next, it waits for 300 microseconds to make sure the ADC is selected, and finally reads the ADC.

interrupter.ino

```
int period_delay = 3500 + (long) 1500 * val_2 / 256;
```

This line computes the cycle delay time based on the value read from the ADC. The minimum frequency is 200Hz, the maximum frequency is approximately 300Hz.

```
ADMUX |= (1 << MUX0);
_delay_us(300);
uint8_t val_1 = ADCH;
```

These three lines read in the value of the pulsewidth pot.

```
double on_time_scale = (double) val_1 / 256;
int on_time = getOnTime(1 / period_delay);
on_time *= on_time_scale;
if(on_time < MIN_ON_TIME) {on_time = MIN_ON_TIME;}
```

Next, we compute the scale factor for the on time, call `getOn-Time()` to compute the on time, and scale it by the computed scale factor. Finally, if the on time is too small, we clamp it from below to `MIN_ON_TIME`.

```
PORTD |= (1 << 2);
delayMicroseconds(on_time);
PORTD &= ~(1 << 2);
delayMicroseconds(period_delay);
```

Finally, we turn the optocoupler on, wait for `on_time` microsec-onds, turn it off, and wait for `period_delay` microcseconds.

handlers.cpp

`handlers.cpp` contains the MIDI message handlers that are called upon reception of MIDI messages. These, along with the timer setup and ISR's found in `timers.cpp`, form the core functionality of the interrupter.

```
void HandleNoteOn(byte channel, byte pitch,
                  byte velocity) {
  if (channel != 1) return;
  PORTD &= ~(1 << 2);
```

`HandleNoteOn` takes the channel the message occured on, the MIDI pitch byte, and the MIDI velocity byte of the note as arguments. First, if the message is not on channel 1, it is ignored. Next, we turn off the optical transmitter, just to be safe.

```
if (velocity == 0) {
  if (pitch == current_pitch_1) {
    HandleNoteOff(channel, pitch, velocity);
  }
  if (pitch == current_pitch_2) {
    HandleNoteOff(channel, pitch, velocity);
  }
  return;
}
```

Next, we check whether the velocity is 0. A zero-velocity note-on message turns off a note - we check which note to turn off by comparing it to the stored note pitches (note that a file or instrument should never turn on a note twice in a row, with no note-off message in between).

```
if (pitch == current_pitch_1 && note_playing_1 &&
sustaining_1) {
  stopTimer1();
  sustaining_1 = false;
  global_scaler_1 = map_velocity(velocity);
  startTimer1(pitch);
  return;
}
if (pitch == current_pitch_2 && note_playing_2 &&
sustaining_2) {
  stopTimer2();
  sustaining_2 = false;
  global_scaler_2 = map_velocity(velocity);
  startTimer2(pitch);
  return;
}
```

We next handle the special case where a key is released, but immediately afterwards, during the sustain period, the key is depressed again. In this case, we clear the "sustaining" status of the previous note, and restart the timers, taking into account the new keypress.

```
if (note_playing_1) {
  if (note_playing_2) {
    stopTimer1();
    stopTimer2();
```

Next, we try to start playing the note. In the first case, notes 1 and 2 are both playing, so we proceed by stopping both timers.

handlers.cpp

```
if (pitch >= TIMER_2_MIN) {
  global_scaler_1 = global_scaler_2;
  sustaining_1 = sustaining_2;
  sustain_time_1 = sustain_time_2;
  global_scaler_2 = map_velocity(velocity);
  sustaining_2 = false;
  startTimer1(current_pitch_2);
  startTimer2(pitch);
}
```

The next bit of code tries to find a timer to play the new note. The default behavior of the interrupter tries to forget the oldest note. In other words, when a new note comes in, the current note 2 replaces note 1, and the new note becomes note 2. However, because of timer prescaler restrictions, note 2 cannot be below a certain pitch (defined as `TIMER_2_MIN`). If the new note statisfies this restriction, the code proceeds as described.

```
else if (current_pitch_2 >= TIMER_2_MIN) {
  global_scaler_1 = map_velocity(velocity);
  sustaining_1 = false;
  startTimer1(pitch);
  startTimer2(current_pitch_2);
}
```

Otherwise, the code leaves note 2 as is, and sets note 1 to the current note.

```
else {
  startTimer1(current_pitch_1);
  startTimer2(current_pitch_2);
}
```

Finally, if all else fails, the code ignores the new note and continues playing the old notes.

If only one note is playing, the code tries to find a place for the new note:

```
else if (pitch >= TIMER_2_MIN) {
  stopTimer2();
  global_scaler_2 = map_velocity(velocity);
  sustaining_2 = false;
  startTimer2(pitch);
}
```

If the new note meets the requirements of Timer 2, note 2 is set to the new note.

```
else if (current_pitch_1 >= TIMER_2_MIN) {
  stopTimer1();
  global_scaler_2 = global_scaler_1;
  sustaining_2 = sustaining_1;
  sustain_time_2 = sustain_time_1;
  note_playing_2 = true;
  current_pitch_2 = current_pitch_1;
  global_scaler_1 = map_velocity(velocity);
  sustaining_1 = false;
  note_playing_1 = true;
  current_pitch_1 = pitch;
  startTimer1(current_pitch_1);
  startTimer2(current_pitch_2);
}
```

Otherwise, the code tries to swap note 1 into note 2 if possible, and then set note 1 to the new note. If all fails, the code ignores the new note.

```
else {
  stopTimer1();
  global_scaler_1 = map_velocity(velocity);
  sustaining_1 = false;
  startTimer1(pitch);
}
```

Finally, if no notes are already playing, the new note becomes note 1.

```
void HandleNoteOff(byte channel, byte pitch,
                   byte velocity) {
  if (channel != 1) return;
```

The next function is the note off handler. We begin by ignoring all messages not from channel 1.

```
if (!sustain) {
  if (pitch == current_pitch_1) {
    stopTimer1();
    note_playing_1 = false;
  } else if (pitch == current_pitch_2) {
    stopTimer2();
    note_playing_2 = false;
  }
}
```

Next, if sustain is not enabled, check what note corresponds to the note off message, and stop that note.

```
else {
  if (pitch == current_pitch_1) {
    sustaining_1 = true;
    sustain_time_1 = SUSTAIN_TIME;
  } else if (pitch == current_pitch_2) {
    sustaining_2 = true;
    sustain_time_2 = SUSTAIN_TIME;
  }
}
```

If sustain is enabled, instead of stopping the timer, we set the sustain flag for the appropriate note. `sustain_time_x` counts how many microseconds of the sustain "tail" remain. Note that the handler doesn't actually do any of the fading; instead, the sustain flag tells the timer that it should start fading the note away every cycle.

```
void HandleStop() {
  stopTimer1();
  stopTimer2();
  sustaining_1 = false;
  sustaining_2 = false;
  note_playing_1 = false;
  note_playing_2 = false;
}
```

The next function handles the MIDI stop message. It simply turns off both notes, taking care to reset the `sustaining` states to `false`.

```
void HandleContinue() {
  if (note_playing_1) startTimer1(current_pitch_1);
  if (note_playing_2) startTimer2(current_pitch_2);
}
```

Continue messages are handled by resuming all notes that were playing when MIDI was paused.

handlers.cpp

```
void HandleControlChange(byte channel, byte number,
                         byte value) {
  if (channel != 1) return;
```

Control change messages contain a command number (which specifies the type of command) and a value (the argument of the command). First, as usual, we ignore all messages that are not in channel 1.

```
if (number == 0x78 || number == 0x79 ||
    number == 0x7B || number == 0x7C) {
  stopTimer1();
  stopTimer2();
  sustaining_1 = false;
  sustaining_2 = false;
  note_playing_1 = false;
  note_playing_2 = false;
}
```

Commands `0x78`, `0x79`, `0x7B`, and `0x7C` are all STOP messages; in this case, we turn off the timers, and flag the notes as not playing.

```
if (number == 0x40) {
  if (value < 64) {
    sustain = false;
    if (sustaining_1) {
      sustaining_1 = false;
      note_playing_1 = false;
      stopTimer1();
    }
    if (sustaining_2) {
      sustaining_2 = false;
      note_playing_2 = false;
      stopTimer2();
    }
  } else {
    sustain = true;
  }
}
```

Command `0x40` toggles sustain; data values less than `64` turn sustain off (and halts all sustaining notes); otherwise, sustain is turned on. This message is sent whenever the sustain pedal is depressed or released.

```
void HandlePitchBend(byte channel, int value) {
  if (channel != 1) return;
  float fbend_amount = ((float) value)/((float)
(0x2000));
```

The pitch bend handler first computes the fractional pitch bend. The MIDI library automatically turns the MIDI pitch bend `unsigned int` into an `int` centered around zero. We scale it by `0x2000` (the maximum pitch bend value) to get the fractional bend value.

```
float bender = 1.0f - 0.1*fbend_amount;
```

Next, it computes a "bend factor"; this is used to scale the frequency.

```
long temp = (long) (bender * ticks_1);
if (temp >= BITS_16) temp = BITS_16 - 1;
OCR1A = (int) temp;
temp = (long) (bender * ticks_2);
if (temp >= BITS_8) temp = BITS_8 - 1;
OCR2A = (int) temp;
```

Finally, it scales the pitch values appropriately, taking care not to overflow the timer compare registers. More precisely, it scales the ticks that are stored in the timer compare registers (`OCR1A` and `OCR2A`), which are proportional to the period lengths of the notes.

```
void HandleSystemReset() {
  stopTimer1();
  stopTimer2();
  note_playing_1 = false;
  note_playing_2 = false;
  on_time_1 = 0;
  on_time_2 = 0;
  current_pitch_1 = 0;
  current_pitch_2 = 0;
  global_scaler_1 = 127;
  global_scaler_2 = 127;
  sustain = sustaining_1 = sustaining_2 = false;
}
```

Finally, the firmware handles system reset messages by halting both notes and restoring all global variables to their default values.

timers.cpp

`timers.cpp` contains the setup functions and the ISR's for the timers, which actually play a note. A *timer* is a special piece of hardware on the microcontroller which counts up every tick. When the counter reaches a preset value, it triggers an interrupt, which tells the microcontroller to stop everything and service the interrupt (by calling the *interrupt service routine*, or ISR).

```
void startTimer1(byte pitch) {
  long frequency = getFrequency(pitch);
  note_1_period = 1000000 / frequency;
  current_pitch_1 = pitch;
  on_time_1 = getOnTime(frequency);
```

The timer 1 setup routine begins by converting the MIDI note to a frequency (in hertz). It then computes the period in microseconds, records the current pitch, and looks up the ON time for the frequency specified.

```
long ticks = 2 * note_1_period;
```

Next, it computes the number of clock ticks per period (since the microcontroller is running at 16 MHz, the default value of 8 cycles per tick translates into 0.5 μS per tick).

```
if (ticks <= BITS_16) {
  TCCR1B = (1 << CS11) | (1 << WGM12);
}
```

If the tick count doesn't make timer 1 overflow, the timer 1 scaler is set to 0.5 μS per tick.

```
else if ((ticks /= 4) <= BITS_16) {
  TCCR1B = (1 << CS11) | (1 << CS10) | (1 << WGM12);
}
```

Otherwise, we set the timer to 2 μS per tick, and scale the value of `ticks` accordingly.

```
ticks--;
ticks_1 = ticks;
OCR1A = ticks;
if (on_time_1 != 0) {TIMSK1 |= (1 << OCIE1A);}
note_playing_1 = true;
```

Finally, it decrements `ticks` (necessary for correct operation; e.g. 1 tick runs for 1 μS and not 0.5, since the timer stops if its compare register is *less than* zero), records the tick value (necessary for pitch bend), writes the tick value to the timer 1 compare register `OCR1A`, starts the timer, and flags note 1 as "playing".

Timer 2 operates nearly identically to timer 1; only the prescaler code is different:

```
long ticks = 16 * note_2_period;
if (ticks <= BITS_8) {TCCR2B = (1 << CS20);}
else if ((ticks /= 8) <= BITS_8) {TCCR2B = (1 <<
CS21);}
else if ((ticks /= 4) <= BITS_8) {TCCR2B = (1 <<
CS21) | (1 << CS20);}
else if ((ticks /= 2) <= BITS_8) {TCCR2B = (1 <<
CS22);}
else if ((ticks /= 2) <= BITS_8) {TCCR2B = (1 <<
CS22) | (1 << CS20);}
else if ((ticks /= 2) <= BITS_8) {TCCR2B = (1 <<
CS22) | (1 << CS21);}
else if ((ticks /= 4) <= BITS_8) {TCCR2B = (1 <<
CS22) | (1 << CS21) | (1 << CS20);}
```

timers.cpp

This code repeatedly increases the prescaler and decreases `ticks`, until the value of `ticks` is an 8-bit integer (as required by timer 2). The remainder of the timer 2 setup code is identical to the timer 1 setup code.

```
void stopTimer1() {
  TIMSK1 &= ~(1 << OCIE1A);
}
void stopTimer2() {
  TIMSK2 &= ~(1 << OCIE2A);
}
```

The timer stop functions *only* stop the ISR's from being called; it is up the calling function to properly set the various global flags associated with halting a note.

Finally, we have the ISR handlers. Timer 1 and timer 2 call virtually identical ISR's, so we will only describe ISR 1.

```
ISR (TIMER1_COMPA_vect, ISR_BLOCK) {
```

Note the rather cryptic declaration of the ISR. This is because, `ISR`, `TIMER1_COMPA_vect`, and `ISR_BLOCK` are all defined constants; the C++ preprocessor expands this internally into a more conventional function declaration. It is not necessary to know what it expands to for the end-user's purposes.

```
uint8_t val = ADCH;
float scaler = (float) (val) / 256.0f;
int on_time_1_adjusted = (int) (on_time_1 * scaler);
```

First, we read the value of the ADC (`ADCH` is a defined value that expands to a memory location). Next, we scale it to a `float` between `0.0` and `1.0`, and then scale the ON time by it.

```
scaler = (float) (global_scaler_1) / 256.0f + 0.5f;
on_time_1_adjusted = (int) (on_time_1_adjusted *
scaler);
```

Next, we shift and scale the velocity scaler so it is a `float` between `0.5` and `1.0`, and scale the ON time by it.

```
if (sustaining_1) {
  scaler = map_sustain(sustain_time_1);
  on_time_1_adjusted = (int) (on_time_1_adjusted *
scaler);
```

Next, we scale the ON time to match the time spent during the sustain period. First, the code maps the sustain time via the `map_sustain()` function, which takes the time (in microseconds) spent sustaining, and returns a `float` between `0.0` and `1.0` which describes the scale factor. Next, it multiplies the ON time by that factor.

```
sustain_time_1 -= note_1_period;
```

Next, we update the time remaining in the sustain period. Since we lack an extra hardware timer to do this, we decrement the sustain time by the note period (since we know the timer is called once every cycle).

```
if (on_time_1_adjusted < 3) {
  sustaining_1 = false;
  note_playing_1 = false;
  stopTimer1();
  PORTD &= ~(1 << 2);
}
```

timers.cpp

If the computed pulsewidth is less than 3 microseconds, the ISR stops the timer, and unflags the note as sustaining and playing. This is necessary since `delay_us()` does not function reliably for times under 3 microseconds.

```
if (on_time_1_adjusted < MIN_ON_TIME && !sustain-
ing_1) {on_time_1_adjusted = MIN_ON_TIME;}
```

If the note is not sustaining, we clamp the pulsewidths from below to `MIN_ON_TIME`, which ensures that notes do not disappear at low power settings or velocity settings (we need a few cycles before any sparks are produced).

```
PORTD |= (1 << 2);
delayMicroseconds(on_time_1_adjusted);
PORTD &= ~(1 << 2);
delayMicroseconds(on_time_1_adjusted);
TIFR2 &= ~(1 << OCF2A);
```

Finally, it turns the transmitter on for an appropriate amount of time. After turning off the transmitter, it waits for a period of time, to insure that the timers are not fired too close to each other.

util.cpp

util.cpp contains assorted short setup and utility functions, which will be documented here.

```
void setupADC() {
  ADMUX =
    (1 << REFS0) |  // Use AVCC reference
    (1 << ADLAR) |  // Left-adjust the result
    (1 << MUX0);    // Select ADC1 (on-time pot)

  ADCSRA =
    (1 << ADEN)  |  // Enable the ADC
    (1 << ADSC)  |  // Start free-running conversion
    (1 << ADATE) |  // Enable auto-triggering
    (1 << ADPS0) |  // Set prescaler to 128
    (1 << ADPS1) |
    (1 << ADPS2);

  // Ensure that the auto-triggering source is in
  // free-running mode
  ADCSRB = 0x00;
}
```

setupADC() sets various ADC configuration registers. More precisely, it sets the individual bits of the registers, which are each flags that configure the behavior of the microcontroller. To do this, it bitwise OR's numbers which correspond to singular 1's in the correct position. ADEN, MUX0, etc are integers which determine the position.

```
void setupTimers() {

  // Set up 16-bit Timer1
  // CTC mode, hardware pins disconnected
  TCCR1A = 0x00;
  // CTC mode, don't set prescaler yet
  TCCR1B = (1 << WGM12);
  TIMSK1 = 0x00;

  // Set up 16-bit Timer2
  // CTC mode, hardware pins disconnected, don't set
  // prescaler yet
  TCCR2A = (1 << WGM21);
  TCCR2B = 0x00;
  TIMSK2 = 0x00;
}
```

CTC mode sets the timer to **C**lear **T**imer on **C**ompare; i.e. the timer will automatically count to the timer compare value, then reset its count to zero,.

```
long getFrequency(byte pitch) {
  return (long) (220.0 * pow(pow(2.0, 1.0/12.0),
                 pitch - 57) + 0.5);
}
```

getFrequency() returns the frequency corresponding to a pitch byte, as per MIDI specs.

```
int getOnTime(long frequency) {
  int index = (int) floor(frequency / 100.0);
  if (index > ON_TIME_ARRAY_LENGTH - 1) {return 0;}
  else {return LOOKUP_TABLE_SCALE * on_times[index];}
}
```

getOnTime() looks up the ON time; the lookup table uses divisions of 100Hz, and 0 is returned if the frequency is too high.

Programming the Interrupter

Follow these instructions to update the firmware on the interrupter's ATmega microcontroller. You will need an AVR ISP programmer.

1. Download and install the latest version of Atmel Studio here.
2. Start Atmel Studio and select *Tools > AVR Programming* to open the programming window.
3. Plug in your AVR ISP Mk. II programmer and select it under *Tool > AVRISP mkII*.
4. Select the correct microcontroller under *Device > ATmega328P.*
5. Power your interrupter board and plug in the AVR programming cable with the wire direction facing towards the outside of the board. A green light should appear on the programmer once it is connected correctly. Inserting the cable the wrong way will not damage the interrupter or the programmer.
6. Under *Device ID,* hit "*Read*" to establish a connection with the microcontroller. The Target Voltage should read near 5V.
7. Under *Interface Settings* (the default tab), set the ISP clock slider to 125kHz.
8. Select the Fuses tab, hit "*Read*" to get the fuse settings in the microcontroller and cofirm the following:
   - `CKDIV8` is **not** selected
   - `SUT_CKSEL` is set to `EXTFSXTAL_16KCK_14CK_4MS1`
9. If these fuses are set differently, change them to the above settings and hit "Program" to reprogram the fuses.
10. Select the *Memories* tab and locate the .hex file from the download above under "Flash". Select "*Erase device before programming*" and "*Verify flash after programming*", and hit "*Program*". You're done!

Compiling the Firmware

Compiling the source is best done in the Arduino IDE. While it may be possible to compile it with a vanilla copy of AVR-gcc, the Arduino IDE has preconfigured include paths that make compiling an essentially seamless process.

1. Download and install the Arduino IDE; version 1.0.3 is recommended.
2. Unzip the downloaded source file. You should get a folder called "`interrupter`"; it is important to leave the name of this folder identical to the `.ino` file inside - this is required by the Arduino IDE.
3. Under *File* > *Preferences,* check "*show verbose output when compiling*".
4. Hit "*Verify*" (the button with the check mark in the top left). This will compile the code into a `.hex` file that you can write to the microcontroller. Once the code has been compiled, the location of the temporary folder with the output files will be printed. You're done!