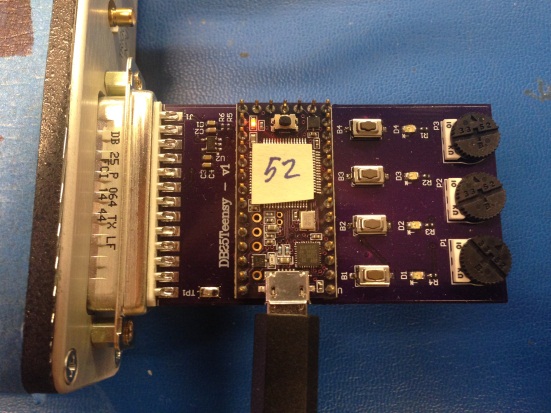
The Generator hooks up to an SSIM and outputs a signal simulating various sensor signals. It allows testing of SSIM and UI software without having to mess around with lasers and actual sensors.



*Waveform Generator connected to SSIM*

The minimum configuration is just the Generator plugged into the SSIM, as shown above. The Generator draws power from the SSIM and generates a standard test pattern. The standard pattern is a sequence of a sine wave, a square wave, and a pulse train, changing approximately every two seconds. This is adequate for a wide range of basic testing of the SSIM and UI software.

However, the Generator also may be connected to a PC via USB. Any terminal emulator program may be used to communicate with it. In the default, “demo mode” it also prints a description of the waveform being generated to the terminal.

# Operating Controls

## Power

The generator normally accepts power from the SSIM via the DB25 connector. As soon as the device is plugged in, it begins running. The normal startup sequence blinks each of the 4 colored LEDs once; then it begins blinking the LED on the Teensy twice per second. If this light stops blinking it means the firmware crashed.

The generator also can draw power from the USB cable. This is useful for firmware development, but otherwise of limited utility since the whole point is to debug software talking to the SSIM.

## Buttons

The unit has 5 buttons.

* ***Reset button***. There’s a little black button on the Teensy board itself, right next to the blinking LED. Pressing it resets the device, and in some circumstances uploads new firmware.
* The button labeled ***B4*** is the ***Zeroing Button***. Pressing the zeroing button toggles the device into and out of zeroing mode. When in Zeroing mode, the generator outputs a 0 level on the DAC, instead of the erstwhile requested waveform. This allows the user to press the Zero button on the software app to zero the meter. Clicking the Zeroing Button again toggles out of zeroing mode, and the generator resumes sending the same waveform as before.
* The ***Red LED (D4)*** blinks, as a clear signal that the device is in zero mode rather than sending normal signals.
* The button labeled ***B3*** is the ***One Shot Button***. The One Shot command (below) causes the generator to wait until this button is pressed before sending a waveform. This may be used to test triggering events in the meter.
* The ***Orange LED (D3)*** is illuminated when the device is in One Shot Mode, waiting for user input. Each click of the One Shot Button will cause a waveform to be sent to the meter, after which it again waits for a button click.
* The ***Green LED (D1)*** is illuminated when the device is in *variable gain mode*. In this mode, the pot P1 controls the overall gain of the system. Turning the pot will change the gain in real time. Gain ranges from 0 to 2X normal.
* The remaining buttons and LEDs (B1, B2, and D2) are reserved for future use.

# Analog Output

The primary objective of the generator is to output waveforms on the DAC port. In the software DAC levels are represented by 16 bit values. The DAC is 12 bits wide, and thus can accept values in the range 0 through 4095 (hex 0xFFF). Theoretically, this corresponds to an output range of 0 to 3.3 VDC. However, I believe some clipping occurs in the circuit before it attains the high rail. Furthermore, the SSIM clips at a much lower level, around to 2.44 volts.

The following definitions are used by the firmware:

#define DAC\_PIN A14 // The Teensey DAC is on A14, and has 12 bits

#define DAC\_BITS 12

#define DAC\_MAX 4095 // (( 1 << DAC\_BITS ) – 1 )

#define DAC\_MAX\_VOLTS 3.3 // supply rail = max output for DAC\_MAX

#define VOLTS\_TO\_BITS( VOLTS ) ((uint)( DAC\_MAX \* VOLTS / DAC\_MAX\_VOLTS ))

#define BITS\_TO\_VOLTS( BITS ) ((uint)( DAC\_MAX\_VOLTS \* BITS / float( DAC\_MAX )))

#define METER\_MAX\_VOLTS 2.44 // ~200 Watts on Heisenberg

// (there's a different max for Meter-less)

#define METER\_MAX\_BITS VOLTS\_TO\_BITS( METER\_MAX\_VOLTS ) // 2574.62 for SSIM

That the firmware has a separate Gain setting, which if enabled, is multiplied against each data sample before writing to the DAC. Gain settings can range from 0 to 2.0. Gain settings can moderate or exacerbate clipping of the resulting signal.

For example, the Demo command generates sine waves with the following command:

**Sine 1158,1158,200,10**

This produces a sine wave of magnitude 1158 DAC counts, but also offsets it by the same amount, so the signal ranges from 0-2036 instead of -1158 thru +1158 (for which the negative values would be clipped).

But the demo command applies a ***gain of 0.1*** to all signals, so the actual signal is attenuated by 90%. So the actual peak voltage is

3.5 \* 1158 / 4095 \* 0.1

which is around 0.0989 volts,

which shows as around 12-13 Watts on the SSIM.

# Generator Command Syntax

The generator accepts commands from and prints any responses to the USB port.

## Lexical Issues

* Commands are ASCII text, terminated by carriage return.
* Upper/lower case doesn’t matter.
* Newline characters are ignored.
* Multiple arguments are separated by commas. Additional spaces may appear before or after arguments or commas.
* Individual numbers must not contain spaces.

## Overall Structure

* Commands start with a name, which must be one word, containing no spaces or punctuation.
* At least one space must separate command name from arguments, if any
* Arguments generally have fixed meaning, based on position
* In a few cases, specific arguments are optional and default values are supplied if that argument is omitted

## Argument Types

* Generator command arguments generally are all numbers. Any c-style floating point literal may be entered. Examples: *0, 1, 10000, 1.5*
* In some cases, the numbers are interpreted as integers, in which case fractional amounts if any are truncated.
* In a few cases, the numbers are interpreted as Boolean values, in which case “0” means false or “off” and 1 or any non-zero value is true or “on”.
* Many arguments are time intervals, for pulse widths or periods. They are interpreted as integer microseconds. Thus “100” means 100 microseconds or 100e-6 or 1e-4 or 0.0001 seconds.
* Many other arguments are DAC output levels. This is an integer number to be presented to the Digital to Analog Converter, which ultimately becomes an output voltage. The DAC is 12 bit, so levels are convert to values in the range 0 through 4095. This was discussed at length, above, in the section on *Analog Output*.
* For the gain command, the argument is a floating point number representing the gain, and may range as follows: 0 < gain <= 2. The gain is applied to each sample level value before being written to the DAC. Default gain is 1.0 which means no change.
* The generator truncates negative values to 0 by, and values > 4095 are truncated to 4095.
* Note that the SSIM meter cannot necessarily read all the values that can be produced by the generator. Too large values will be clipped by the meter.

# USB Commands

Commands are of two types: commands and queries. Queries always include a question mark in the name. Commands further maybe categorized into several categories, each having its own section below.

## Utility Commands

*Utility commands* affect the overall operation of the device, generally without changing the type of waveform being generated.

***\*idn?*** – The Identify query. The device prints some identifying information, including the firmware version number and date.

***?*** – The Help query. Prints a list of all commands and their arguments.

***Gain*** – set gain factor command. Each data sample of any waveform is multiplied by that scale factor before being written to the DAC.

*Gain* arguments:

* *Factor* = optional gain factor to apply to all subsequent output. If the optional argument is omitted, gain is set to 1.0 which means that the DAC levels in the waveform definitions are applied unmodified to the DAC.

***gain?*** – query gain factor

***startup*** *[command]* -- save “*[command]*” in EEPROM and use it for the startup command each time the device is restarted. There must be at least one space between ***startup*** and *[command]*.

E.g., the factory default is “*startup demo*”.

***Startup?*** – query startup settings

***Factory*** – restore EEPROM to factory defaults. This reverses the effect of the ***startup*** command.

## Mode Commands

Mode commands change the overall operational behavior of the device.

***Stop*** command causes the unit to zero the DAC and exit *Demo Mode* if it is active. There is no *Start* command. You have to specify a new waveform command to start the device running again.

***Demo*** command starts the generator cycling the familiar demo waveforms. As it runs it prints to the terminal the equivalent commands necessary to form the waveform.

There is one, optional argument:

* *OnOff* = whether demo mode is on or off. This argument is optional.
  + ***0*** turns demo mode off, which is same as *Stop*.
  + ***1*** (or no argument) to turn it on.

***Zero*** command causes the device to enter *zeroing mode*, same as if the zeroing button is toggled. Unlike the zeroing button, the zero command does not toggle the mode. The command accepts a single, optional argument. ***Zero*** or ***Zero 1*** turns zeroing on, and ***zero 0*** turns it off. However, like the zeroing button, zeroing mode remembers the waveform currently being sent to the DAC, and that waveform resumes generation when zeroing mode exits. Also, the zeroing button still does toggle zeroing mode. So the mode may be started or stopped by either the button or a command.

***VariableGain*** *command and V****ariableGain?*** *query* controls variable gain mode. When enabled, the gain of the output signal for all waveforms is controlled by pot ***P1***. The gain varies from 0 to 2.0, depending on the position of the pot P1.

*VariableGain a*rguments:

* *OnOff* = set variable gain mode; ***1*** turns it on, ***0*** turns it off.

***OneShot*** command starts one-shot mode. A pulse of the specified magnitude and duration will be sent each time the B3 button is pressed.

*OneShot* arguments:

* *MaxLevel* = the magnitude of the pulse
* *MinLevel* [optional] = the value to output before and after the pulse; 0 if omitted
* *Duration* = the duration of the pulse

## Waveform Commands

*Waveform commands* are the most common ones. They cause the selected waveform to be repeatedly sent to the DAC. Waveform generation continues indefinitely until the device receives another command or is stopped or restarted.

***Fixed*** command causes the DAC to continuously output the stated *level*.

*Fixed* arguments:

* *Level* = DAC level to output continuously.

For example, “***fixed 0***” will flat-line the DAC (a lot like Zeroing mode, except no blinking light, or button toggle to exit), and “***fixed 1000***” will set it to the middle.

***Pulse*** command indefinitely alternates two output levels. It can generate square waves of any duty cycle.

*Pulse* Arguments:

* *firstLevel* = DAC level for first portion of waveform
* *secondLevel [optional]* = DAC level for second portion of waveform; this argument defaults to 0 if only 3 arguments are supplied.
* *firstWidth* = width of the first portion of waveform, the pulse
* *secondWidth* = width of the second portion of waveform

For example, in demo mode *Pulse* is used both for the pulse train and for the square wave signal:

* square wave (50% duty cycle): ***Pulse 2316, 0, 400, 400***
* pulse train (10% duty cycle): ***Pulse 2316, 0, 100, 900***

***Sine*** command outputs sine waves.

*Sine* arguments:

* *amplitude* = amplitude of the sine wave
* *offset* = offset from zero (usually >= *amplitude*, so waveform doesn’t clip on the bottom)
* *count* = number of samples
* *width* = duration of each sample

The arguments define a single full wave sine cycle, which then plays out repeatedly. The frequency of the waveform depends on the number of samples and the duration of each one. More samples in your sine wave will make the waveform smoother. Fewer samples will distort the shape. For a given frequency, more samples will require a higher sample rate, and beyond a point, the meter won’t be able to keep up. Just like analog electronics, if you push the limits you’ll get distortion.

E.g., for a sine wave of 100 Hz, each cycle lasts 0.01 seconds or 10 milliseconds. If you want 100 samples per cycle, then the duration of each sample must be 0.01/100, 0.0001 seconds, or 100 microseconds. 10 microseconds per sample seems to be close to the practical limit of the generator, so that would limit you to 1 kHz using 100 samples.

A simple rule of thumb, the frequency of your sine wave in Hz will be the number of samples times the duration of each one in microseconds. Following are three different 1 kHz waveforms, , and:

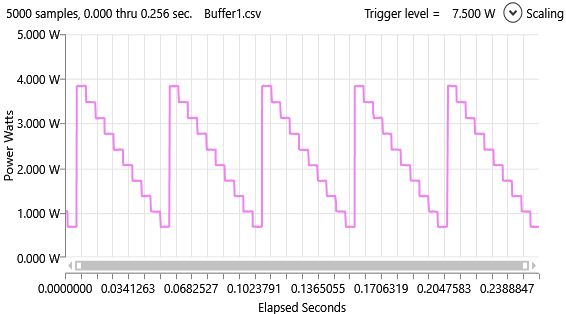
|  |  |  |
| --- | --- | --- |
| *50 samples at 200 uS* | *100 samples at 100 uS* | *1000 samples at 10 uS* |
| **sine 1200,1200,50,200** | **sine 1200,1200,100,100** | **sine 1200,1200,1000,10** |
| C:\Users\jb\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Trend.png | C:\Users\jb\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Trend.png | C:\Users\jb\Desktop\Trend.png |

***Step*** command repeatedly sends a stair-step sequence.

*Step* arguments:

* *Count* – the number of steps
* *Start* – the starting level
* *Step* – the amount of increase (or decrease) for each step
* *Width* – the width of each step

E.g.: ***Step 10, 1000, -100, 5000*** will produce a waveform that starts at 1000, decrease by 100, every 5 milliseconds, for a total of 10 steps.



***VarSine*** isour newest command. It outputs a sine wave, but key settings are adjustable via the analog Pots, as follows:

* *P1* controls amplitude
* *P2* controls offset
* *P3* controls frequency

***varsine*** arguments define the ranges of these pots:

* *ampMin* – minimum amplitude (at P1 full ccw position)
* *ampMax* – maximum amplitude (at P1 full cw position)
* *offMin* – minimum offset (at P2 full ccw position)
* *offMax* – maximum offset (at P2 full cw position)
* *freqMin* – minimum frequency (at P3 full ccw position)
* *freqMax* – maximum frequency (at P3 full cw position)
* *count* – the number of samples in the waveform [optional, 100 if omitted]

The number of samples in the waveform (memory allocation) is fixed for the duration of the command. It affects the smoothness of the curve. 100 seems like plenty of samples.

# Downloadable Waveform Commands

The 2.0.2 release includes for the first time the ability to download custom waveforms. The way this works is that the system can keep track of a single, custom waveform. You download it, then you can play it, and stop it. You can run some other non-custom waveforms, and later return to play the custom one.

There are two types of custom waveforms: *uniform* and *arbitrary*. *Uniform* waveforms are a sequence of DAC levels that will be generated, each having a uniform duration. *Arbitrary* waveforms consist of samples, each of which has its own duration.

*Downloadable Waveforms* is the newest feature. Please let me know if you find any bugs or see ways to improve it. I have tested it, and it should work if you precisely follow the specs. But if you push the envelope you may crash the system. It was tricky getting this to work and the current commands are essentially just a first draft. So I’m also interested in suggestions how to make the commands more usable before anybody becomes overly dependent on this implementation.

***Def*** command – defines a new custom waveform. This allocates space for the waveform and prepares for downloading of the data.

*Def* arguments:

* *Keyword*: “arb” or “uni” – to select the type of waveform being defined.
* *Count* – the number of samples in the waveform
* *Duration* – the duration of each sample, in microseconds, if the waveform is uniform. Argument is optional for arbitrary waveforms, and will be ignored in that case if present.

E.g: ***def uni, 20, 100*** specifies a uniform waveform with 20 samples, each with a 100 uSec duration.

***Def?*** query – displays definition of current custom waveform. Mainly for my own debugging, but the question may come up for you too.

***Load*** command – loads data for a custom waveform that is being constructed. Each load command loads one data sample, and you must send as many load commands as specified by the count in the *Def* command.

*Load* Arguments:

* *Duration* of this sample, in microseconds. This argument is optional and may be omitted for Uniform waveforms. Or if present, it will be ignored for non-arbitrary waveforms.
* Value – the DAC level for this sample. The value is required for both types of waveforms, and syntactically, it will always be the last/right-most column of numbers in a sequence of *Load* commands.

***Play*** command – plays the current custom waveform, if any. Not meaningful to Play an incompletely loaded waveform, or if there isn’t one defined. *Play* command doesn’t have any arguments.

***Undef*** command – deletes the current custom waveform, if any, and reclaims the memory.

## Usage Notes

* The durations in arbitrary waveform definitions are *durations*, and *not timestamps*. Most waveform data comes with timestamps, not durations, and will need to be converted. Thus 10, 20, 30, 40, etc. becomes, 10, 10, 10, 10…
* In both kinds of waveforms, the levels are DAC values, though they’ll be subject to scaling by the gain mechanism.
* Defining a waveform causes any current waveform generation to be stopped.
* After defining a waveform you have to explicitly ***Play*** it to get it to run.
* Defined waveforms persist until explicitly undefined or overwritten. That is, you can define a waveform, play some other standard waveform, then return and play the custom waveform again.
* Since there are multiple load commands, it is problematic starting the waveform playing when the loading is completed. Not impossible, but easy enough to simply include a play command at the end of your download file.

## Examples

I have successfully loaded and run the following waveforms:

|  |  |
| --- | --- |
| ***TestArb1.txt:***  def arb, 10  load 1000, 100  load 1000, 2000  load 100,1000  load 100,2000  load 100,1000  load 100,2000  load 100,1000  load 1000, 2000  load 1000, 500  load 1000, 100  play |  |

|  |  |
| --- | --- |
| ***TestUni1.txt:***  def uni, 10, 250  load 100  load 200  load 400  load 800  load 2000  load 2000  load 1000  load 800  load 600  load 500  play |  |

# Memory Consumption

The *Teensy Arduino* processor has 64K of static ram available for program use. About 10% of that is consumed by the app itself. The remainder is available for waveforms, which must not overflow the available space

Most example waveforms consume only a small fraction of this space. Furthermore, the program frees up old waveforms before creating new ones. After each new waveform is created, the program prints out the amount of “Free ram” remaining in the system.

Unfortunately, it is a known bug in the Arduino runtime system that it does not return a proper error indication when the program requests too much memory. Unfortunately, if you request too much memory, the app crashes. If not for this bug I would print a message, as you might expect.

I have successfully repeatedly create waveforms with 5000 data points, without a problem. IMHO, 5K samples is WAY beyond any practical use of this device. Several hundred samples should be more than enough for most applications. SO long as you keep your requirements modest, you should be not have any problems. Memory requirements for uniform waveforms are approximately 2 bytes per sample. For arbitrary waveforms, it is 6 bytes per sample.