ARL-TN-1132 ● SEP 2022

DEVCOM ARMY RESEARCH LABORATORY

Low Size, Weight, Power, and Cost (SWaP-C) Phasor Calibrator Module

by Brandon Parks and Kevin Claytor

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-1132 ● SEP 2022

Low Size, Weight, Power, and Cost (SWaP-C) Phasor Calibrator Module

Brandon Parks
DEVCOM Army Research Laboratory

Kevin Claytor
Fibertek, Inc.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
September 2022	Technical Note	1 October 2019 – 30 July 2022

4. TITLE AND SUBTITLE
Low Size, Weight, Power, and Cost (SWaP-C) Phasor Calibrator Module

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Brandon Parks and Kevin Claytor

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEVCOM Army Research Laboratory
ATTN: FCDD-RLS-EM
Adelphi, MD 20783

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TN-1132

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Rapid development of low-cost Internet-of-Things (IoT) devices can lead to unforeseen code instability and errors. This can be disastrous if the IoT devices are connected as hardware-in-the-loop for control of the power grid. For measurement devices such as synchrophasor measurement units, an external signal source of known amplitude, frequency, and harmonic distortion can be used as a calibration. This report presents a low-cost, low-power, low-complexity calibration device able to perform a number of key checks for synchrophasor measurement units. Additionally, because it is controllable via USB serial communications, it is able to allow IoT power monitoring devices to self-calibrate using a wide range of preprogrammed waveforms. The hardware and software development considerations of this calibrator are presented here as well as its performance and ability to check a real-world IoT power-monitoring device, the US Army Combat Capabilities Development Command Army Research Laboratory Autonomous Real-Time Electric- and Magnetic-Field Integrated Sensor.

15. SUBJECT TERMS
Electromagnetic Spectrum Sciences, Network and Computational Sciences, electric power, electric field, magnetic field, digital signal processing, embedded systems, data acquisition

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Brandon Parks
Unclassified	Unclassified	Unclassified	UU	53	19b. TELEPHONE NUMBER (Include area code) (301) 394-2347

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

1. Introduction

Low-cost Internet-of-Things (IoT) distributed power monitoring can have beneficial effects on grid stability, reliability, and resilience.[1] However, with the speed of IoT device development, an external, physical calibration device is needed to ensure performance is not adversely affected during an update. This report introduces a low size, weight, power, and cost (SWaP-C) calibration device that can be used to generate a range of test waveforms and ensure proper operation of a voltage- and current-sensing power meter. The calibration device is small enough to be portable for field tests and on-site debugging. Additionally, the calibrator both draws power and has serial communication over USB, allowing the IoT device under test to host it and cycle through the calibration waveforms as needed.

These calibration sequences include power system waveforms that test

- fundamental and harmonics,
- amplitude,
- off-nominal frequencies,
- total harmonic distortion, and
- split- and 3-phase virtual loads.

The device is built around a low-cost ATmega32u4 microcontroller-based development board that generates a pulse-width-modulation waveform on the digital output pins. This digital output can be low-pass filtered (LPFed) or passed directly to the US Army Combat Capabilities Development Command (DEVCOM) Army Research Laboratory (ARL) Autonomous Real-Time Electric- and Magnetic-Field Integrated Sensor (ARTEMIS) unit, and the onboard filtering will low-pass the test signal. Test tones are generated using lookup tables (LUTs), which provide for an easy implementation of the waveforms but does not generate an arbitrary tone. The waveforms included in this system are targeted to 60-Hz power systems but may be extended to other grid frequencies.

In general, the functions and the test cases they describe are given by the following:

- Function 0 – Create test tone:
    - Calibrate frequency and amplitude for LUT generation
- Function 1 – Constant amplitude, mode varies frequency:
    - Tests that root mean square values are being produced

    - Tests that response is flat against frequency
    - Tests that there is no channel permutation
- Function 2 – Test amplitude:
    - Tests linearity of demodulated signal
- Function 3 – Test frequency:
    - Tests off-nominal frequency performance
- Function 4 – Test total harmonic distortion fundamental (THDF)[2]:
    - Tests that THDF is being computed from the even harmonics
    - Tests that THDF is being computing using odd harmonics
    - Tests that THDF is being correctly computed using even and odd harmonics

2. Methods

The calibration box was implemented using only an ATmega32u4 microcontroller and a series of low-pass resistor–capacitor (RC) filters. The ATmega32u4 is capable of analog signal simulation through direct digital synthesis using pulse width modulation (PWM). The waveforms are implemented as a series of 128-point LUTs. This was determined to be a good tradeoff between minimizing LUT size and retaining enough resolution for the off-nominal frequency tests. These LUTs are stored in the larger flash, or PROGMEM, instead of the smaller, faster static random-access memory (SRAM). When a desired waveform is needed, it is transferred from PROGMEM to SRAM, and, moreover, each channel has a phase-offset variable that allows relative phase to be set between channels. Finally, serial communication allows a controller computer to change the waveform on demand and enables hardware-in-the-loop (HIL)-type operation. These capabilities are described in detail in the following sections.

2.1 Direct Digital Synthesis of Waveforms

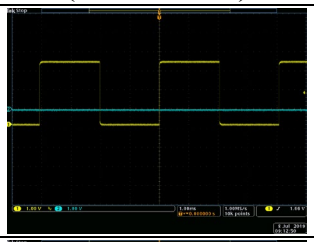
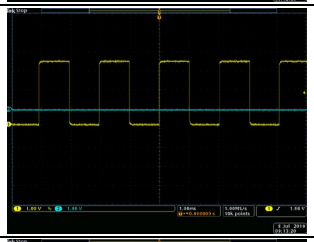
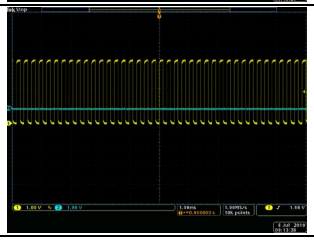
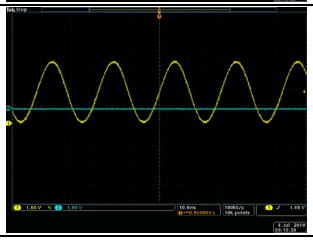
The ATmega32u4 is a pure digital micro controller and has no analog-to-digital converters (ADCs). Instead, it simulates analog output through PWM,[3] which rapidly pulses an output pin and changes the duty cycle of that pulse to simulate a corresponding analog voltage. This allows us to generate our desired waveforms without the added complexity of a separate ADC chip; however, the waveforms are restricted to the 0- to 3.3-V range.

A PWM pulse train with 0% duty cycle would be 0 V; a PWM pulse train with 100% duty cycle would correspond to the system voltage (here, 3.3 V); and a PWM pulse train with 50% duty cycle would be 50% of the system voltage (3.3 V; 2 = 1.65 V). However, the frequency of the PWM signal also matters. When using PWM to dim an LED or drive a motor for a robot a few hundred hertz is fine. In this case (or in audio synthesis), the signal frequency (60 Hz) is a large fraction of the default PWM frequency (500–1000 Hz). This has two implications:

1) A filter is likely to cause large roll-off of our desired frequency component, and
2) even after filtering, the PWM frequency is likely to be present in the output.

Table 1 demonstrates this effect with a 50% duty-cycle PWN signal and a sine wave after filtering with a low-pass RC filter with cutoff frequency of approximately 150 Hz. Note this cutoff frequency is for demonstration only; the final calibration boards implemented an LPF with a cutoff frequency of approximately 1.5 kHz.

Table 1 A visualization of a 50% duty cycle PWM waveform (unfiltered, left column), and a 60-Hz sinusoid waveform (low-pass RC-filtered, right column) for three different PWM frequencies. The 60-Hz sinusoid waveform is severely distorted at low PWM frequencies and does not resemble the correct waveform until a PWM frequency of 8 kHz. Note the different timescales for the PWM and filtered sine plots.

PWM frequency	50% PWM signal (10-ms window)	Sine wave with filtering (100-ms window)
500 Hz		
1000 Hz		
8000 Hz		

The ATmega32u4 has four configurable timers on board that can be used to drive the PWM signals and provide interrupts for updating the waveform. In this application we use Timer 1 and 4 for the fast PWM while Timer 0 runs the interrupts that update the waveform shape. In particular, output pins 9, 10, 11, and 13 are run at a high PWM rate for output as per Table 2.

Table 2 PWM frequency for output channels. Pin 13 is the simulated voltage channel for the device and is run at a higher PWM frequency.

Pin	PWM timer	Timer frequency (kHz)	Output channel label
13	Timer 4	94	0
11	Timer 1	62	1
10	Timer 1	62	2
9	Timer 1	62	3
N/A	Timer 0	62	N/A

Additionally, Timer 0 is repurposed as a timed interrupt to update the outputs from the LUTs. Since Timer 0 is used to implement the `millis()` call, the output of this function will no longer represent elapsed milliseconds.[4]

2.2 Lookup Tables

Waveform generation is performed by LUT instead of dynamic creation. This simplifies the logic on the device itself, thus requiring only a memory copy and pointer update operations. Since the operations are the same, it results in equal-time waveform generation and a one-time calibration applies to all waveforms. This avoids a time calibration that could depend on the type of waveform being produced and the computational complexity to create it; for example, sine versus square wave generation. It also eases waveform creation, which can be generated using higher-level numerical libraries such as numpy[5] and verified graphically before being uploaded to the device. Finally, this makes the logic simple to add additional waveforms; any arbitrary waveform can be precomputed, adjusted for the fixed time lookup delay, and added to the command switch statement.

As described in Section 2.1, Timer 0 controls the interrupt and is set to 62 kHz. Ideally this would indicate that we could create frequencies up to 31 kHz. However, in practice the interrupt logic requires some time to execute, and this reduces the maximum realizable frequency. In fact, for our four-channel LUT scheme the maximum frequencies we can create for a Timer 0 frequency of 62, 8, and 1 kHz are 3600, 1400, and 180 Hz, respectively, as shown in Table 3. From this we can compute the minimum LUT size required for 50- and 60-Hz signals (the waveform will fit into any larger table, but a smaller table could not hold the full waveform).

Table 3 Maximum output frequency, and corresponding LUT size for 50- and 60-Hz waves for a Timer 0 frequency of 62, 8, and 1 kHz

Timer 0 frequency (kHz)	Max frequency (Hz)	Minimum LUT size 50 Hz	Minimum LUT size 60 Hz	Points per Hz at 50 Hz	Points per Hz at 60 Hz
62	3600	144	120	2.7	1.9
8	1400	56	47	1.1	0.74
1	180	7	6	0.13	0.10

First, we want to be able to test at least the 300 Hz (fifth harmonic) signal. According to Table 3 this could be achieved by the 62- and 8-kHz clock rates, where the 300-Hz signal would have a length of approximately 25 and 9 points, respectively, but not with the clock rate of 1 kHz, which has a maximum frequency of 180 Hz.

Additionally, the off-nominal frequency test involves a step of 1 Hz, producing frequencies of 58, 59, 61, and 62 Hz. To produce these signals relatively cleanly (without a large discontinuity at the wraparound edges), we desire a near integer number of points per hertz change. This is achieved with the 62-kHz Timer 0 frequency for 60-Hz signals and 8 kHz for 50-Hz signals.

At the maximum Timer 0 clock rate of 62 kHz, we can only produce total harmonic distortion up to the approximately 60th harmonic (2 points/cycle), or 3600 Hz. This frequency is substantially higher than the calibrator filter cutoff frequency of 1.5 kHz. However, the calibrator filter can be bypassed, and the raw PWM signal can be LPFed by the ARTEMIS anti-aliasing filter, which has an approximately 3.2-kHz cutoff frequency.

2.2.1 Frequency and Amplitude Calibration

With the interrupt clock frequency set, we can produce a "calibration waveform" that can be used to scale the time base and magnitude of the LUTs to produce the correct frequency and amplitude signals. This calibration waveform fills the 128-point LUT with a single sinusoid with an expected amplitude of 1 V_{RMS}. The expected amplitude in the PWM levels is given by

$$a = \lfloor 1 * 2 * np.sqrt(2) * \lfloor n_{PWM}/2 \rfloor / V \rfloor \quad (1)$$

For a V = 3.3-V chip logic and n_PWM = 255 PWM levels, this maps to an integer amplitude of 108. After measuring the frequency and amplitude of this signal as f_{cal} and a_{cal}, the desired frequency and amplitude can be scaled as

$$f_{\text{ccorrected}} = (1/128) * (f/f_{\text{cal}}) \quad (2)$$

$$a_{\text{corrected}} = \lfloor 108 * a/a_{\text{cal}} \rfloor \quad (3)$$

where the value of 128 is the length of the LUT and the value 108 is defined by Eq.1. The resulting calibrated, 60-Hz 1 V_{RMS} waveform is shown in Fig. 1.

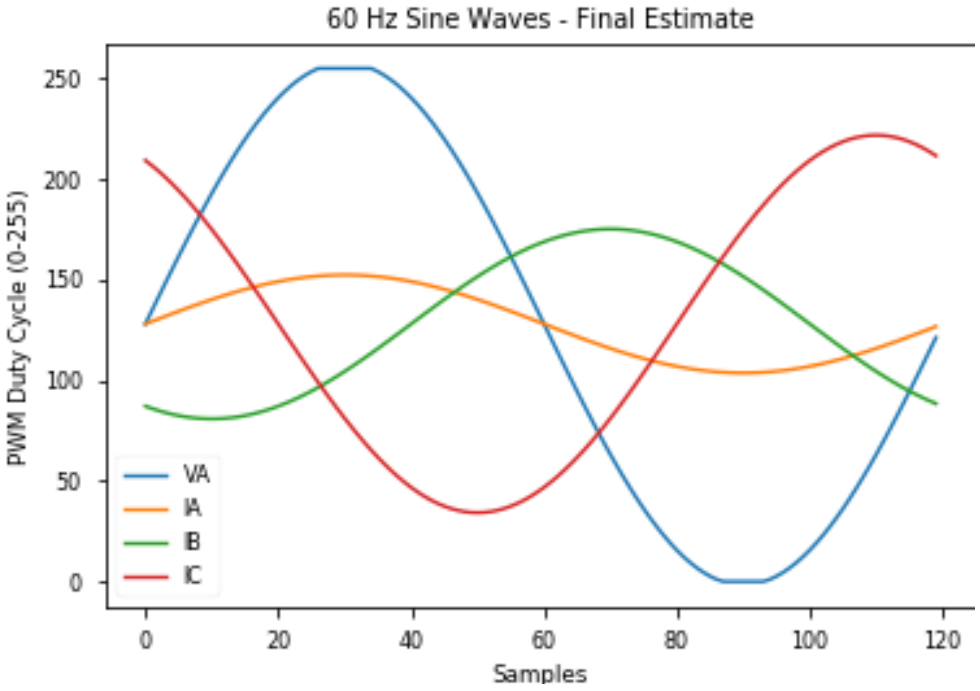


Fig. 1 Final Python waveform. Although it appears to clip, because of the LPF averaging of the PWM waveform, the resulting waveform does not clip.

2.2.2 Memory Considerations

With the LUT logic and accurate timing incorporated, all the waveforms must be loaded into the device memory.

According to the product datasheet,[3] the ATmega32u4 has 32 kb of flash and 2 kb of SRAM. Initially, the LUTs were written as global variables to the more restrictive SRAM. With two LUTs, this method worked, but *always* failed when extended to a third, with the Arduino compiler yielding the following message:

```
Sketch uses 9060 bytes (31%) of program storage space. Maximum is
28672 bytes.
Global variables use **2798** bytes of dynamic memory.
```

Although not particularly clear, “dynamic memory” is synonymous with SRAM, and the approximately 2.8 kb required for the three LUTs exceeds the 2 kb available. There are two modifications we can make; first, we can reduce the size of the array by specifying only the largest data type required to hold the data, and second, we can move the data to the flash memory.

Instead of using the 16-bit (2-byte) signed integer, `int`, datatype, the PWM duty cycle ranges from 0 to 255, allowing a single byte to be used.[6] This reduces the space to approximately 1.6 kb of SRAM well within the overhead. However, this is still insufficient to store the required number of tables for the wide variety of desired waveforms.

Since the values within the LUTs do not change and they are specified as constant, `const` arrays, we can move them to the flash memory with the `PROGMEM` utility.[7,8] This is as simple as including the utility header `pgmspace.h` and specifying the variables as `PROGMEM`:

```
#include <avr/pgmspace.h>

const byte lut_size = 128;

const byte table_cal_128_size = 128;
const byte table_cal_128[128] PROGMEM = {...};
```

However, the pointer used to reference the location of the array exists in SRAM and can only point to variables in SRAM. Consequently, we use a 4 × 128 array that stores the current waveform on all channels. When a command to switch waveforms is issued, the corresponding array location in `PROGMEM` is memory copied into a row of this array. This also improves performance over fetching the individual elements from flash memory.

Finally, strings are notorious for using up SRAM, so we move these to flash with the `F()` macro. This frees up enough memory in SRAM for the 4 × 128 current waveform array.

With these changes, the compiler reports the following:

```
Sketch uses 7688 bytes (26%) of program storage space. Maximum is
28672 bytes.
Global variables use **665** bytes of dynamic memory.
```

Not only are we using only approximately 30% of the SRAM, we also have plenty of overhead in flash for additional waveforms should they be desired.

2.3 Communication

Serial communication on the Arduino is fairly straightforward: Characters can be read with `Serial.read()` and written with `Serial.print()` or `Serial.println()`. However, sending commands strings that can be parsed for specific values is somewhat more detailed.

Since the waveform generation from the LUTs is performed completely in the timer interrupts, this leaves the main loop available for serial communications: There is a nonblocking call to `recvWithStartEndMarkers()`, which continues to scan the serial port and gather characters until both the start ($) and end (*) markers have been captured. Once a message has been received, a flag is set and the COMMAND, FUNCTION, MODE, CHANNEL, and PHASE are extracted via comma delimiters from the string.[9] If the command is successfully parsed and executed, an acknowledgement string and newline character `<ACK...>\n` are returned.

To change the output waveform, connect to the device over USB serial communication and issue an ARLTX command. An example implementation in Python is shown in the following:

```
import serial
from serial.tools import list_ports

def arltx_1(ser):
    msg = "$ARLTX,1*00"
    print(f"Sending:  {msg}")
    ser.write(msg.encode())
    resp = ser.readline().decode()
    print(f"Received: {resp}")

if __name__ == "__main__":
    # Discover all serial ports
    devices = [ser.name for ser in list_ports.comports()]
    logger.info(f"Found devices: {devices}")
    # Assume port 0 is the calibrator device
    ser = serial.Serial(f"/dev/{devices[0]}", 115200, timeout=2)
    arltx_1(ser)
```

The full range of ARLTX commands can be found in Appendix A.

2.4 Hardware

The microcontroller chosen was a low-cost (under $10) Adafruit Itsybitsy 8-Mhz 3.3 V.[10,11] This is based on the ATmega32u4 microcontroller[3] and has an integrated USB serial connection for programming and communication. This board was chosen over the Feather series,[12] as no integrated battery would be required; the device would be powered through the USB port while communicating with the host ARTEMIS platform that was being calibrated.

The hardware design and layout were done with the open-source Fritzing[13] software, which can be downloaded from their GitHub releases page.[14] This was done to provide LPFing of the PWM signal and connectors for the BNC bulkheads.

As described in Section 2.1, the waveforms are synthesized by PWM. It is difficult to visualize the low-frequency content with the high-frequency PWM signal. The ARTEMIS system contains a front-end filter that rolls off at approximately 3.2 kHz and will filter out the PWM signal, so no additional filter is needed. However, for calibration of the reference waveform and human visualization, a low-pass RC filter with a cutoff frequency of approximately 1.5 kHz was designed into the hardware board. The schematic and printed circuit board (PCB) layout are shown in Fig. 2

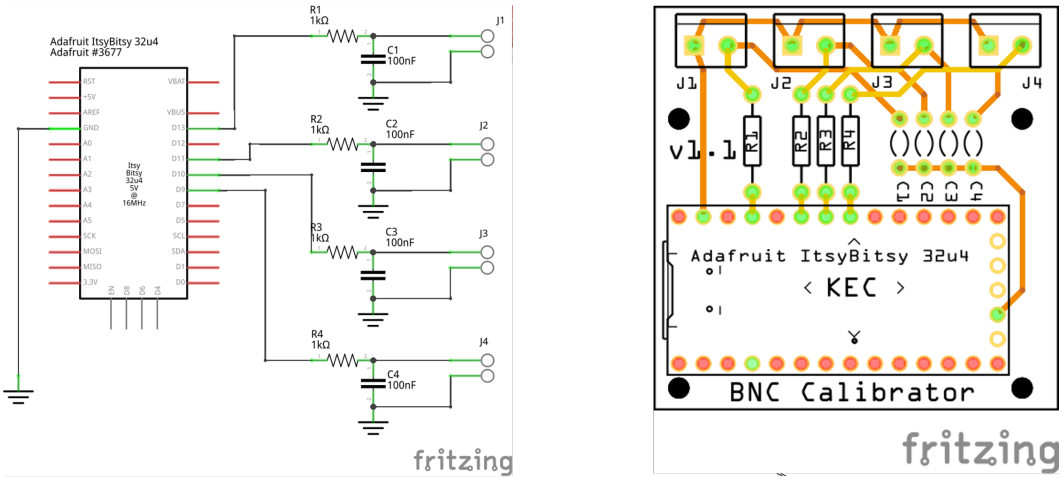


Fig. 2 (left) Calibrator schematic and (right) PCB layout

The first system built is shown in Fig. 3 left. It successfully demonstrated the validity of the schematic and software. However, the bulkhead connectors added substantial volume and cost and required additional cables to connect to the ARL-ARTEMIS. The second board spin, v1.1, shown in Fig. 3 right, did away with these in favor of male BNC to pigtail connectors that can connect directly to ARTEMIS. The calibrator can also be encased in a 3-D printed enclosure for weatherproofing.

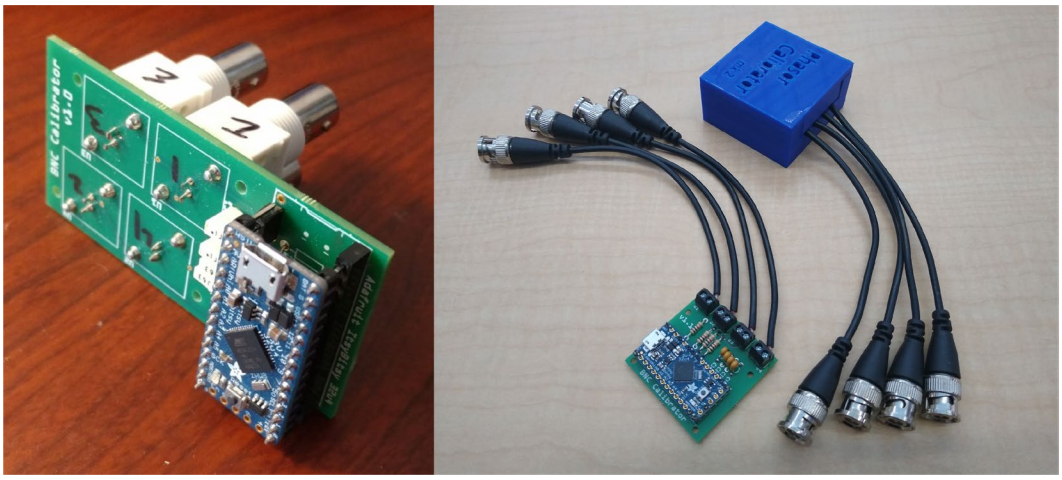


Fig. 3 Hardware builds: (left) v1.0 device and (right) v1 board and v1.1 board

3. Results

In this section we show the example output waveforms and demonstrate how they can be used as a calibrator for a power-sensing system.

3.1 Example Waveforms

Figure 4 shows the pure tone fundamental (60 Hz) and harmonic waveforms. While the waveform does not appear very clean, this is an artifact of the LPF; they all have $1V_{RMS}$ of energy at the corresponding frequency bin. It can be seen, however, that the highest frequency (300 Hz) is somewhat off. This is due to the calibration being computed at the lower frequency; it drifts somewhat as the frequency increases. One could apply the frequency calibration in Eq. 2 to the higher frequencies to obtain a more accurate frequency.

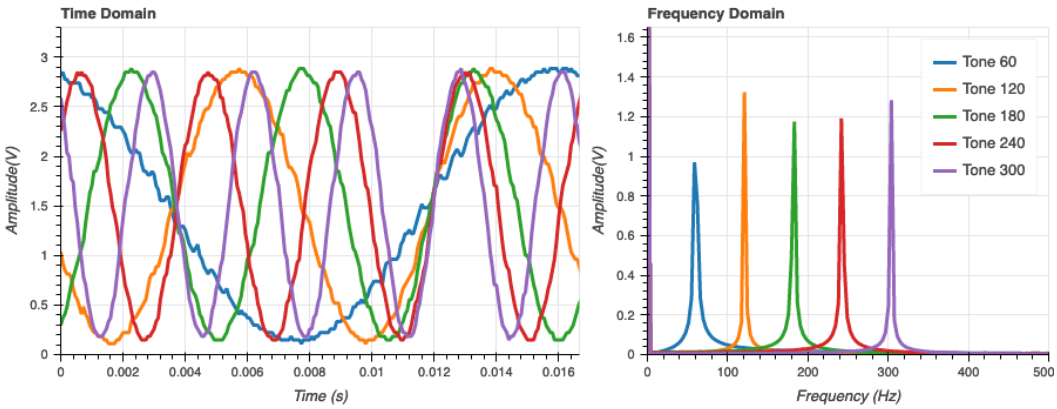


Fig. 4 Oscilloscope time domain measurement of the pure tone test waveforms, as well as their frequency domain representation

Figure 5 shows the amplitude sweep from $1V_{RMS}$ down to $1/16V_{RMS}$. Again, while the $1/16^{th}$ amplitude waveform appears very jagged, due to the LPF and the average duty cycle being near 50%, it has the appropriate energy at 60 Hz.

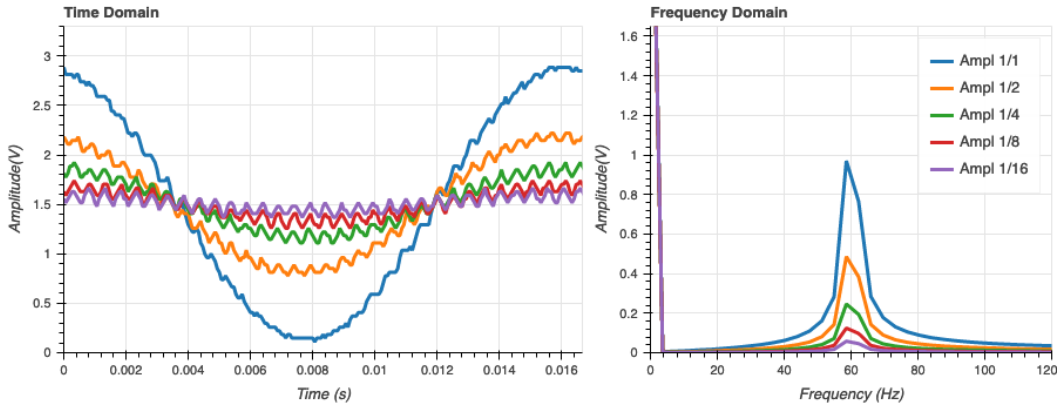


Fig. 5 Oscilloscope time domain measurement of the amplitude test waveforms as well as their frequency domain representation

Figure 6 shows the off-nominal frequency waveforms. Unfortunately, the oscilloscope did not capture enough points to yield a very accurate frequency estimation, yet it can be seen that the 60-Hz signal has energy spread between nearby bins while the lower and higher frequencies have more energy in the lower and higher bins, respectively. The frequency variation is perhaps clearer in the time domain waveform.

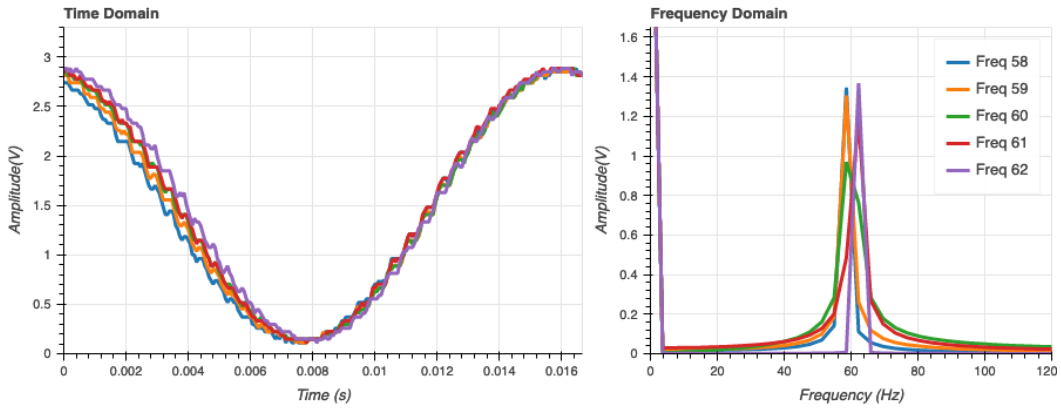


Fig. 6 Oscilloscope time domain measurement of the frequency test waveforms as well as their frequency domain representation

Figure 7 shows the THDF test waveforms. The even and odd harmonic tests have more energy in the fundamental and then equal values in the harmonics.

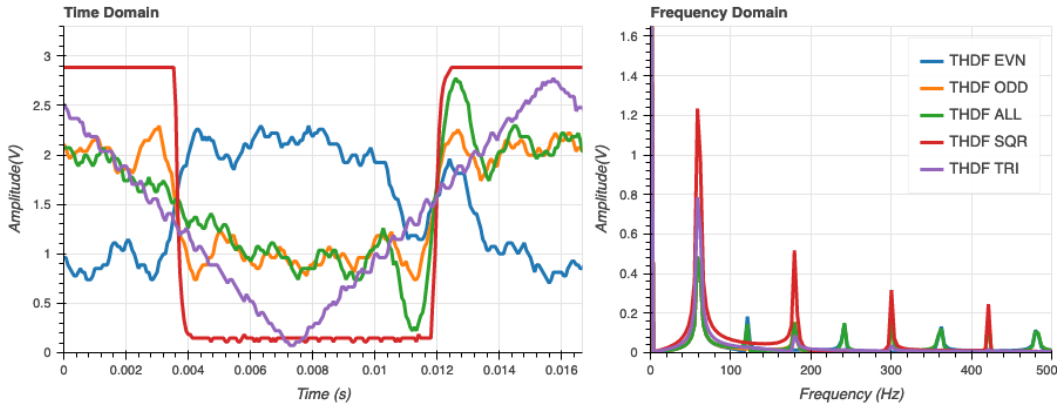


Fig. 7 Oscilloscope time domain measurement of the total harmonic distortion test waveforms as well as their frequency domain representation

A full description of output waveforms can be found in Appendix B, which also contains higher-resolution oscilloscope captured data as well as data for all the channels as opposed to the Channel 0 waveforms shown here.

3.2 ARTEMIS Self-Test

The Python USB-serial communication code outlined in Section 2.3 can be extended to loop over all waveforms. Since the ARTEMIS has Python bindings for receiving data as well, a single Python control function can iterate over calibration waveforms and ensure the correct responses are being measured—a full HIL test.

Although each calibrator module can only generate signals on four channels, it is possible to identify multiple, uniquely addressable calibrator modules and extend this calibration to 8, 16, or more channels. This is exactly what is done in the Python script in Appendix C, which uses the powerful pytest testing framework[15] to parameterize a range of tests over all available calibrators and available banks (sets of four inputs). Additionally, the tolerance of test results can be independently set and even marked as expected to fail (for example, the default parameters are to monitor only harmonics 3 and 5 in addition to the fundamental, so the even harmonic test will fail).

4. Conclusions

This project demonstrates that a low-cost, small form-factor calibrator has been developed for the calibration of power system monitoring equipment using nothing more than a microcontroller development board. Signals are generated via PWM that mimic the output voltage from voltage and current clamps. These signals are LPFed and passed to the measurement device. Because the four output channels

can be individually controlled to produce 24 unique waveforms for testing, as well as a split- and three-phase emulated circuit, and multiple devices can be multiplexed together and controlled via USB serial, allows for an exceptionally flexible calibration device.

There are, of course, other interesting avenues that could be improved upon. First, the device currently only has LUTs for 60-Hz waveforms. The LUTs for 50-Hz waveforms could also be created, and the Timer0 controlled to provide accurate frequency steps for the frequency test.

The LUTs could be abandoned entirely for a more flexible arbitrary waveform generation. However, the waveform (e.g., sinusoid) must be generated fast enough that the interrupts can update and produce the correct frequency.

Since the output waveforms are generated by digital PWM, they currently span 0–3.3 V. While some DC offset is to be expected, this could certainly be reduced and the waveforms centered around 0 V.

Finally, these devices were designed to calibrate the ARTEMIS system. These devices have a secondary input of LEMO bulkheads in addition to the BNC bulkheads. These are designed to pull calibration information from a transducer electronic datasheet (TEDS)-compliant smart sensor through a one-wire interface. There exists one-wire emulation for Arduino[16] that could be used to emulate a sensor, and thus calibrate the LEMO inputs, without any additional hardware.

5. References

1. Parks B, Claytor K, Hull D. Distributed Live Animated Multi Phasor (dLAMP). Presented at the Military Sensing Symposium (MSS) Specialty Groups on Battlespace Acoustic, Seismic, Magnetic, and Electric-Field sensing and Signatures (BAMS); 2017.
2. Shmilovitz D. On the definition of total harmonic distortion and its effect on measurement interpretation. IEEE Trans Power Delivery. 2005 Jan;20(1):526–528. doi: 10.1109/TPWRD.2004.839744.
3. Microchip. PWM Sine Wave Generation.1. ATmega328 8-bit AVR microcontrollers. https://www.microchip.com/wwwproducts/en/ATmega328.
4. Hoang K. Khoih-prog/TimerInterrupt; 2021 Jan. https://github.com/khoih-prog/TimerInterrupt/blob/master/src/TimerInterrupt.h.
5. NumPy. https://numpy.org/.
6. learn.sparkfun.com. Data types in Arduino. https://learn.sparkfun.com /tutorials/data-types-in-arduino/all.
7. nongnu.org. Avr-libc: <avr/pgmspace.h>: program space utilities. http://www.nongnu.org/avr-libc/user-manual/group__avr__pgrmspac.html.
8. Arduino.cc. PROGMEM Arduino reference. https://www.arduino.cc /reference/en/language/variables/utilities/progmem/.
9. Forum Arduino.cc. Serial input basics - updated; 2016. https://forum .arduino.cc/index.php?topic=396450.
10. Adafruit Industries. Adafruit ItsyBitsy 32u4 - 3V 8MHz. https://www.adafruit .com/product/3675.
11. Adafruit Learning System. Introducing ItsyBitsy 32u4. https://learn.adafruit .com/introducting-itsy-bitsy-32u4/overview.
12. Adafruit Industries. Adafruit Feather 32u4 Basic Proto. https://www.adafruit.com/product/2771.
13. Fritzing. http://fritzing.org/.
14. Fritzing. fritzing/fritzing-app. github/fritzing/fritzing-app/releases.
15. Pytest. pytest: helps you write better programs. https://docs.pytest.org/en /latest/.

16. Splitt II. Orgua/OneWireHub; 2021 Jan. https://www.arduino.cc/reference/en/libraries/onewirehub/.

Appendix A. Command Tables

A.1 Main Command Table

There are currently five supported commands: 1, 10, 20, 30, 120, and 130. These are detailed in Table A-1. Command 1 is an identification query; 10, 20, and 30 are set waveforms on a specific channel; and 120 and 130 produce simulated split- and three-phase circuits, respectively. Only integers should be sent as the `FUNCTION`, `MODE`, `CHANNEL`, and `PHASE` variables. The `FUNCTION NAME` and `MODE NAME` variables are used to provide human-readable feedback about the current state of the calibrator. As acknowledgement that the mode was set, the calibrator returns an `ACK` string with human-readable `FUNCTION NAME` and `MODE NAME`. The identification acknowledgement contains the device ID, hardware version, software version, and build date.

Table A-1 The serial commands parsable by the calibration box. These include an identification command (Command 1), specific channel control commands (Commands 10, 20, and 30), and quick circuit simulation commands (Commands 120 and 130).

Command	Syntax	Response	Meaning	Channel output
1	`$ARLTX,1*00`	`<ACK,[ID], [HW VER],[SW VER], [BUILD_DATE]>`	Query device ID and version info	Waveforms left unchanged
10	`$ARLTX,10,[FUNCTION],[MODE],[CHANNEL],[PHASE]*00`	`<ACK,10, [FUNCTION NAME`[d]`], [MODE NAME`[d]`], [CHANNEL], [PHASE]>`	Set device operation to produce `FUNCTION` and `MODE`[a] waveform on CHANNEL[b] and with PHASEB[c]	Set `FUNCTION` and `MODE` waveform on `CHANNEL`; other channels unchanged
20	`$ARLTX,20,[FUNCTION],[MODE],[CHANNEL],[PHASE]*00`	`As above`	Quick check `CHANNEL`	Set `FUNCTION` and `MODE` waveform on `CHANNEL` 0 output on other channels
30	`$ARLTX,30,[FUNCTION],[MODE],[IGN],[PHASE]*00`	`As above`	Quick check all `CHANNELs`	Set `FUNCTION` and `MODE` waveform on all channels

[a] `FUNCTION` and `MODE` are integers in the range [0, 4]. Their function is described in Section A.2.
[b] `CHANNEL` is an output channel (integer in the range [0, 4]).
[c] `PHASE` can vary in the range [0, 128], which maps to [0, 2*pi].
[d] `FUNCTION NAME` and `MODE NAME` are human-friendly names for the corresponding function and mode and are returned in the response string.

Table A-1 The serial commands parsable by the calibration box. These include an identification command (Command 1), specific channel control commands (Commands 10, 20, and 30), and quick circuit simulation commands (Commands 120 and 130) (continued).

Command	Syntax	Response	Meaning	Channel output
120	`$ARLTX,120*00`	`<ACK,120>`	Quick check split-phase circuit	Channel 0 has 1.1 V_{rms} Phase A; Channel 1 has 100 mV_{rms} Phase A; Channel 2 has 100 mV_{rms} Phase B (–180°); Channel 3 has 0 mV_{rms} Phase N (0°)
130	`$ARLTX,130*00`	`<ACK,130>`	Quick check three-phase circuit	Channel 0 has 1.1 V_{rms} Phase A; Channel 1 has 100 mV_{rms} Phase A; Channel 2 has 100 mV_{rms} Phase B (–120°); Channel 3 has 100 mV_{rms} Phase C (+120°)

[a] `FUNCTION` and `MODE` are integers in the range [0, 4]. Their function is described in Section A.1.

[b] `CHANNEL` is an output channel (integer in the range [0, 4]).

[c] `PHASE` can vary in the range [0, 128], which maps to [0, 2*pi].

[d] `FUNCTION NAME` and `MODE NAME` are human-friendly names for the corresponding function and mode and are returned in the response string.

A.2 Function and Mode Table

Table A-2 describes all available functions, modes, and a description of their output waveform.

Table A-2 An enumeration of the available `FUNCTION` and `MODE` integers with the corresponding `FUNCTION NAME` and `MODE NAME` strings that are returned on a successful set, and their physical interpretation

Function	Function name	Mode	Mode name	Output / meaning
0	CALI	0	CAL	128 point, full amplitude sine wave
		1	x00	128 point, all points 0% duty cycle
		2	x7f	128 point, all points 50% duty cycle
		3	xff	128 point, all points 100% duty cycle
		4	ERR	UNUSED/RESERVED FOR FUTURE USE
1	TONE	0	60	1 V_rms at 60-Hz signal.
		1	120	1 V_rms at 120-Hz signal
		2	180	1 V_rms at 180-Hz signal
		3	240	1 V_rms at 240-Hz signal
		4	300	1 V_rms at 300-Hz signal
2	AMPL	0	1/1	1 V_rms at 60-Hz signal
		1	1/2	1/2 V_rms at 60-Hz signal
		2	1/4	1/4 V_rms at 60-Hz signal
		3	1/8	1/8 V_rms at 60-Hz signal
		4	1/16	1/16 V_rms at 60-Hz signal
3	FREQ	0	58	1 V_rms at 58-Hz signal
		1	59	1 V_rms at 59-Hz signal
		2	60	1 V_rms at 60-Hz signal
		3	61	1 V_rms at 61-Hz signal
		4	62	1 V_rms at 62-Hz signal
4	THDF	0	EVN	Fundamental + Even Harmonics (THDF ~= 25%)
		1	ODD	Fundamental + Odd Harmonics (THDF ~= 25%)
		2	ALL	Fundamental + All Harmonics (THDF ~= 35%)
		3	SQR	Square wave (THDF ~= 48.3%.
		4	TRI	Triangle wave (THDF ~= 12.1%)

Note: THDF = total harmonic distortion fundamental

Appendix B. Waveforms

This appendix shows oscilloscope plots of the waveforms for all supported command, function, and modes.

B.1 Command 10

Command 10 sets the mode of a specific channel without changing the modes of the other channels. In the following table, only oscilloscope Channel 1 (calibrator Channel 0) is shown.

Table B-1 Command 10

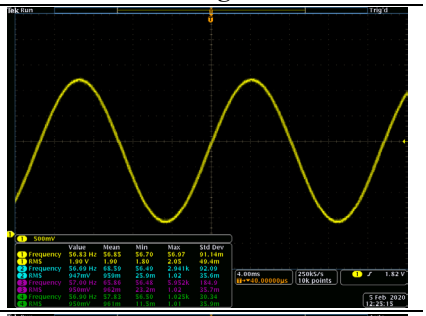
Command	Function name	Image
$ARLTX,10,0,0,0,0*00	CALI 128	
$ARLTX,10,0,1,0,0*00	CALI x00	
$ARLTX,10,0,2,0,0*00	CALI x7f	
$ARLTX,10,0,3,0,0*00	CALI xff	

Table B-1 Command 10 (continued)

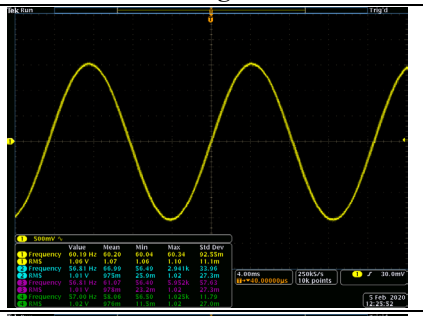
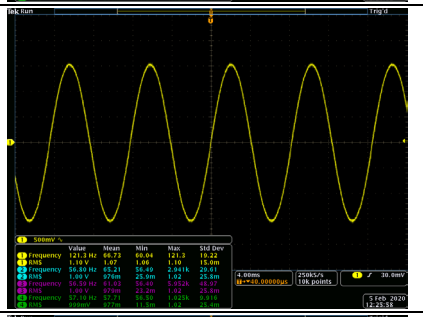
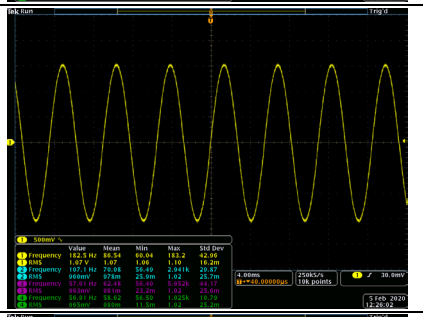
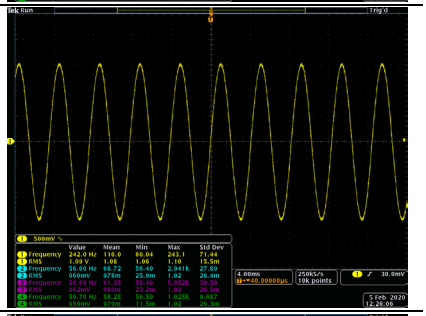
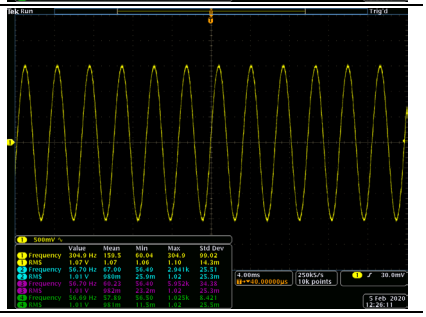
Command	Function name	Image
$ARLTX,10,1,0,0,0*00	TONE 60	
$ARLTX,10,1,1,0,0*00	TONE 120	
$ARLTX,10,1,2,0,0*00	TONE 180	
$ARLTX,10,1,3,0,0*00	TONE 240	
$ARLTX,10,1,4,0,0*00	TONE 300	

Table B-1 Command 10 (continued)

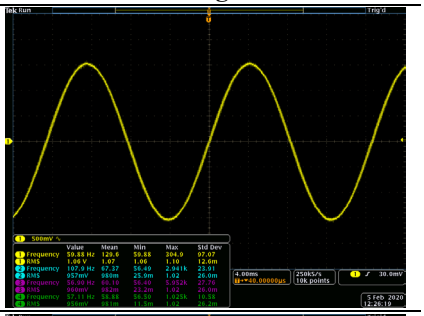
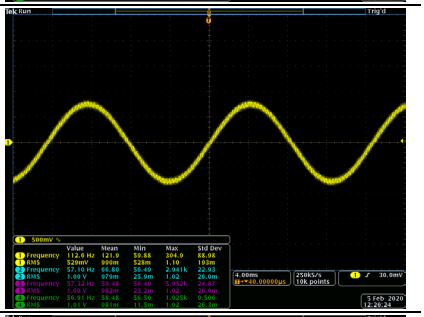
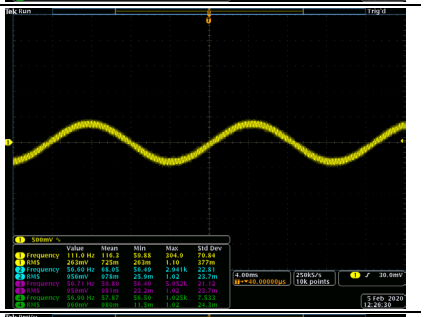
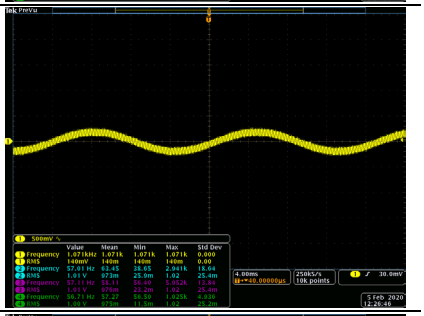
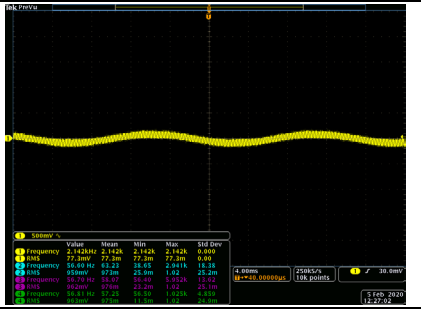
Command	Function name	Image
$ARLTX,10,2,0,0,0*00	AMPL 1	
$ARLTX,10,2,1,0,0*00	AMPL 1/2	
$ARLTX,10,2,2,0,0*00	AMPL 1/4	
$ARLTX,10,2,3,0,0*00	AMPL 1/8	
$ARLTX,10,2,4,0,0*00	AMPL /16	

Table B-1 Command 10 (continued)

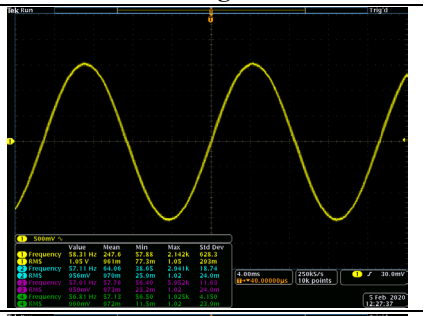
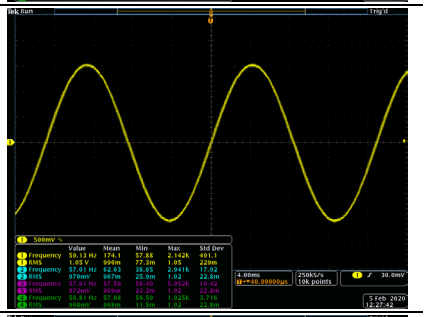
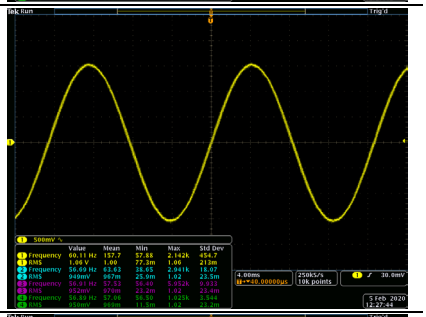
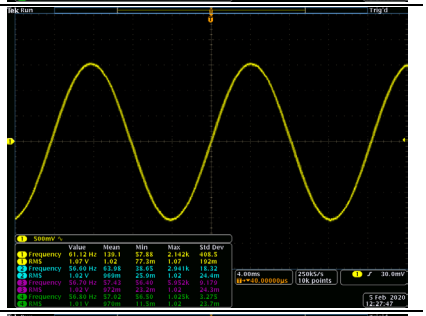
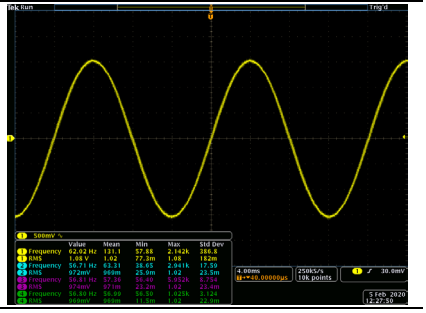
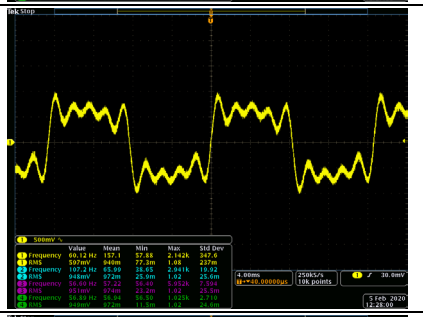
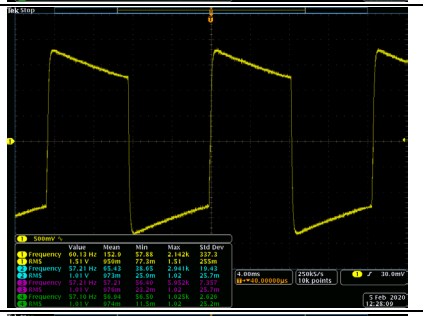
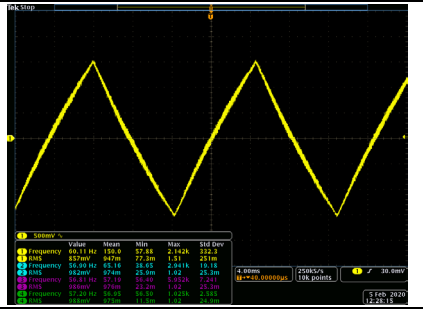
Command	Function name	Image
$ARLTX,10,3,0,0,0*00	FREQ 58	
$ARLTX,10,3,1,0,0*00	FREQ 59	
$ARLTX,10,3,2,0,0*00	FREQ 60	
$ARLTX,10,3,3,0,0*00	FREQ 61	
$ARLTX,10,3,4,0,0*00	FREQ 62	

Table B-1 Command 10 (continued)

Command	Function name	Image
$ARLTX,10,4,0,0,0*00	THDF EVN	
$ARLTX,10,4,1,0,0*00	THDF ODD	
$ARLTX,10,4,2,0,0*00	THDF ALL	
$ARLTX,10,4,3,0,0*00	THDF SQR	
$ARLTX,10,4,4,0,0*00	THDF TRI	

B.2 Command 20

This command sets the active channel to the desired waveform and all other channels to zero. In the following table, oscilloscope Channel 1 (calibrator Channel 0) is the active channel.

Table B-2 Command 20

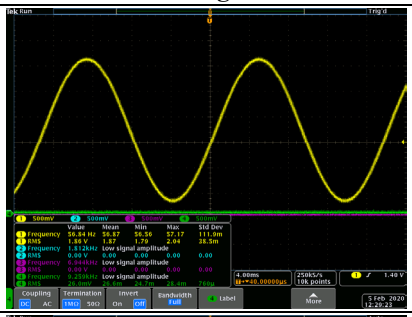
Command	Function name	Image
$ARLTX,20,0,0,0,0*00	CALI 128	
$ARLTX,20,0,1,0,0*00	CALI x00	
$ARLTX,20,0,2,0,0*00	CALI x7f	
$ARLTX,20,0,3,0,0*00	CALI xff	

Table B-2 Command 20 (continued)

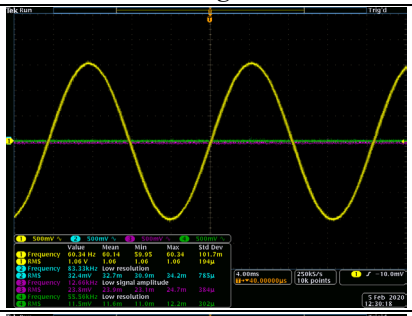
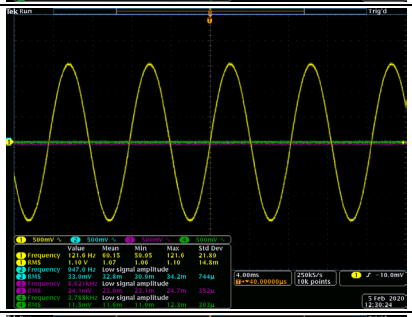
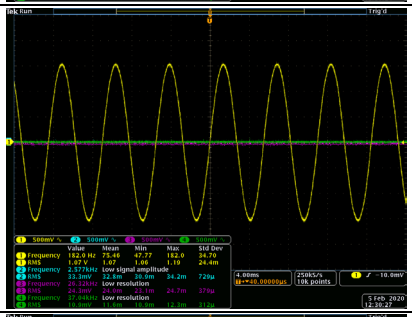
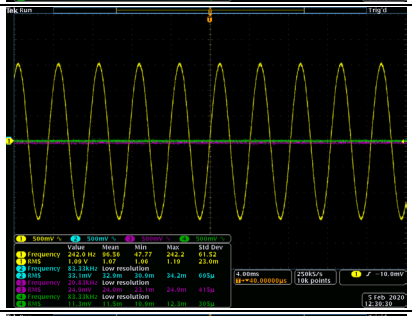
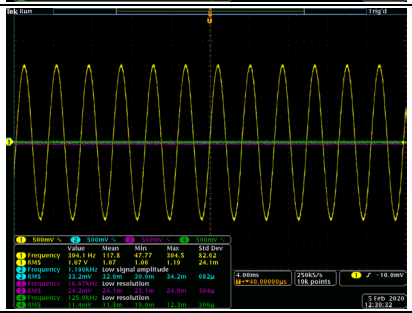
Command	Function name	Image
$ARLTX,20,1,0,0,0*00	TONE 60	
$ARLTX,20,1,1,0,0*00	TONE 120	
$ARLTX,20,1,2,0,0*00	TONE 180	
$ARLTX,20,1,3,0,0*00	TONE 240	
$ARLTX,20,1,4,0,0*00	TONE 300	

Table B-2 Command 20 (continued)

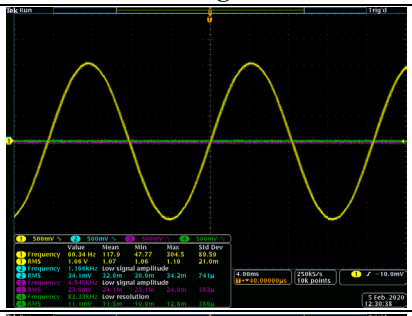
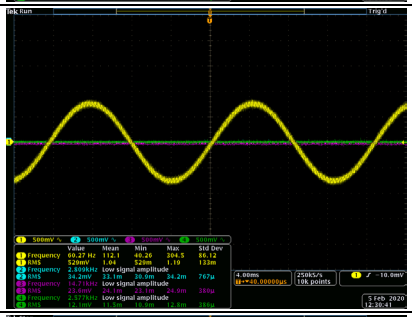
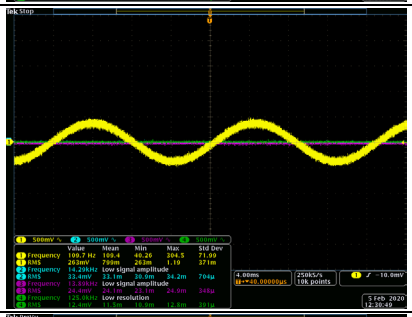
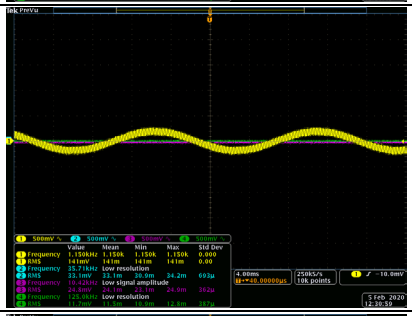
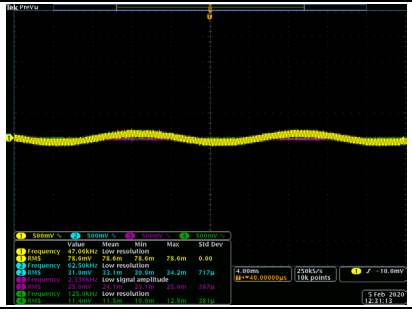
Command	Function name	Image
$ARLTX,20,2,0,0,0*00	AMPL 1	
$ARLTX,20,2,1,0,0*00	AMPL 1/2	
$ARLTX,20,2,2,0,0*00	AMPL 1/4	
$ARLTX,20,2,3,0,0*00	AMPL 1/8	
$ARLTX,20,2,4,0,0*00	AMPL /16	

Table B-2 Command 20 (continued)

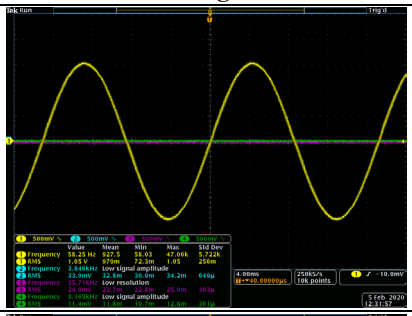
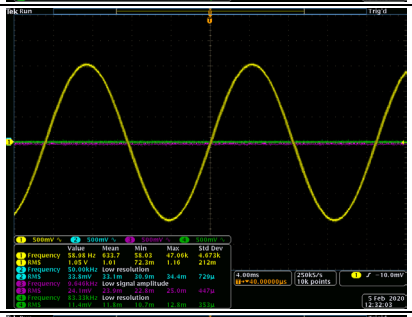
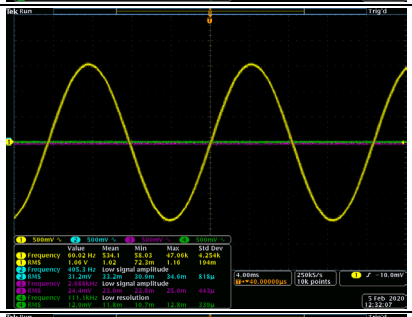
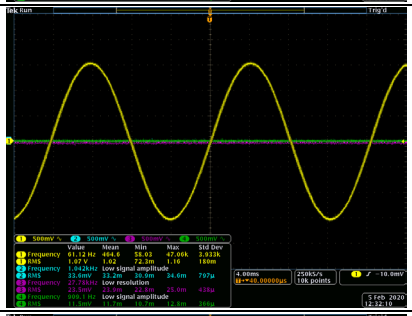
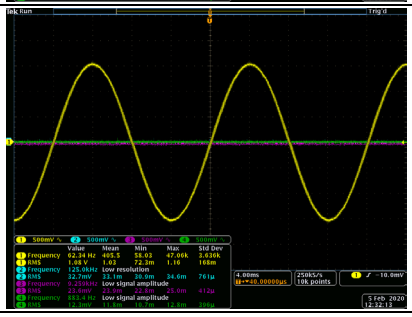
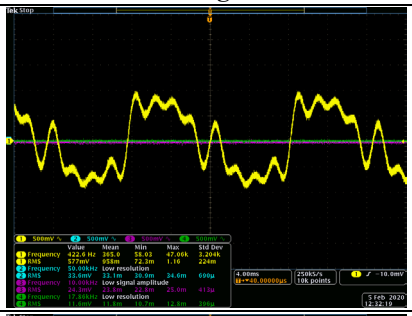
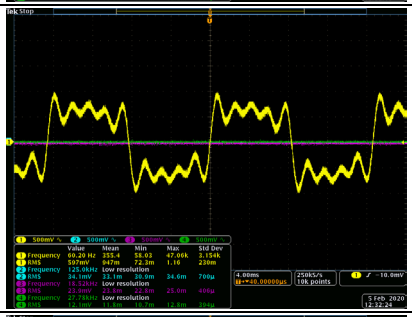
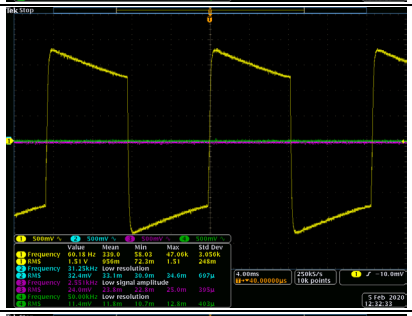
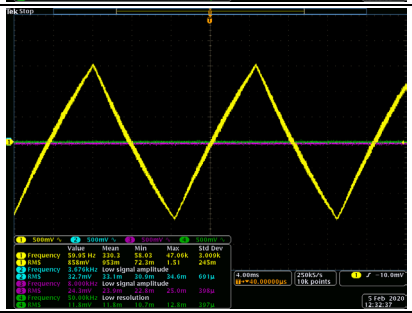
Command	Function name	Image
$ARLTX,20,3,0,0,0*00	FREQ 58	
$ARLTX,20,3,1,0,0*00	FREQ 59	
$ARLTX,20,3,2,0,0*00	FREQ 60	
$ARLTX,20,3,3,0,0*00	FREQ 61	
$ARLTX,20,3,4,0,0*00	FREQ 62	

Table B-2 Command 20 (continued)

Command	Function name	Image
$ARLTX,20,4,0,0,0*00	THDF EVN	
$ARLTX,20,4,1,0,0*00	THDF ODD	
$ARLTX,20,4,2,0,0*00	THDF ALL	
$ARLTX,20,4,3,0,0*00	THDF SQR	
$ARLTX,20,4,4,0,0*00	THDF TRI	

B.3 Command 30

This command sets all channels to the desired waveform. In the following table, oscilloscope Channels 1–4 map to calibrator Channels 0–3.

Table B-3 Command 30

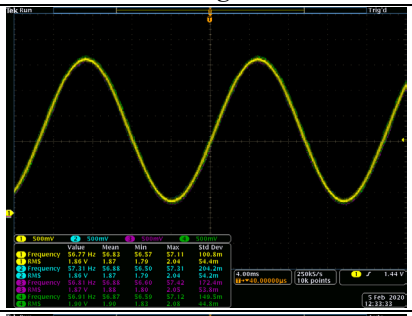
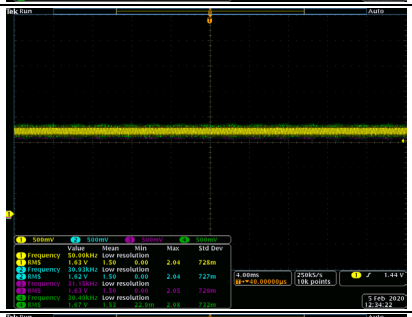
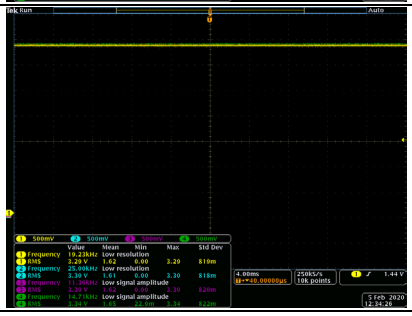
Command	Function name	Image
$ARLTX,30,0,0,0,0*00	CALI 128	
$ARLTX,30,0,1,0,0*00	CALI x00	
$ARLTX,30,0,2,0,0*00	CALI x7f	
$ARLTX,30,0,3,0,0*00	CALI xff	

Table B-3 Command 30 (continued)

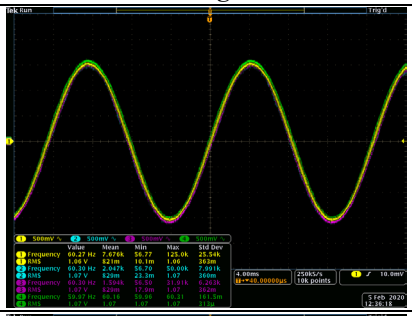
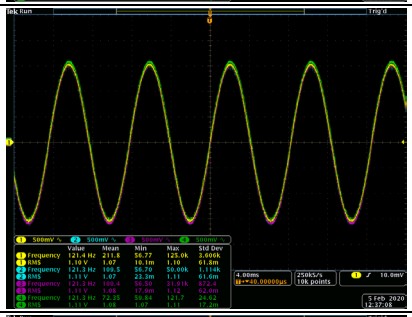
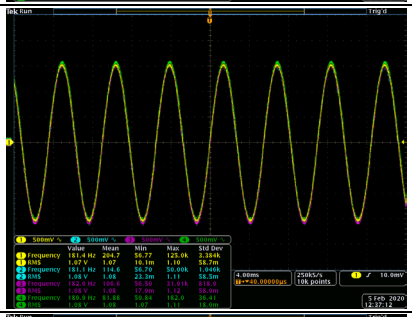
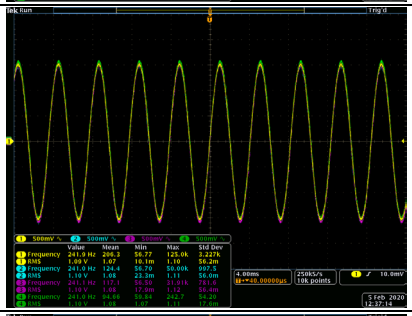
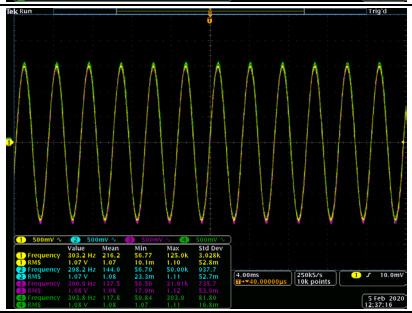
Command	Function name	Image
$ARLTX,30,1,0,0,0*00	TONE 60	
$ARLTX,30,1,1,0,0*00	TONE 120	
$ARLTX,30,1,2,0,0*00	TONE 180	
$ARLTX,30,1,3,0,0*00	TONE 240	
$ARLTX,30,1,4,0,0*00	TONE 300	

Table B-3 Command 30 (continued)

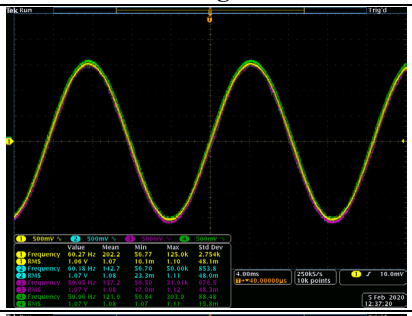
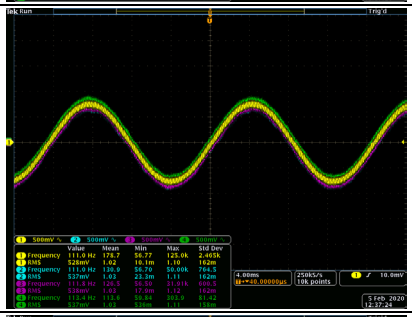
Command	Function name	Image
$ARLTX,30,2,0,0,0*00	AMPL 1	
$ARLTX,30,2,1,0,0*00	AMPL 1/2	
$ARLTX,30,2,2,0,0*00	AMPL 1/4	
$ARLTX,30,2,3,0,0*00	AMPL 1/8	
$ARLTX,30,2,4,0,0*00	AMPL /16	

Table B-3 Command 30 (continued)

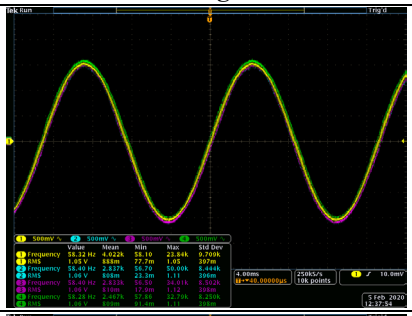
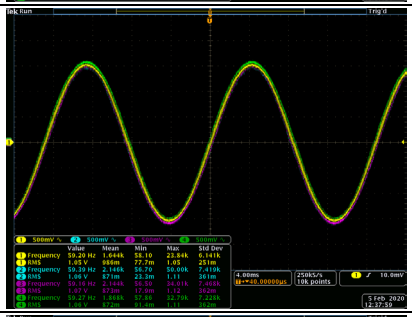
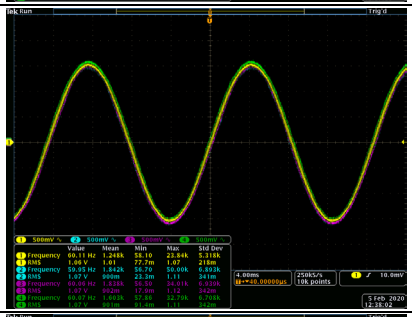
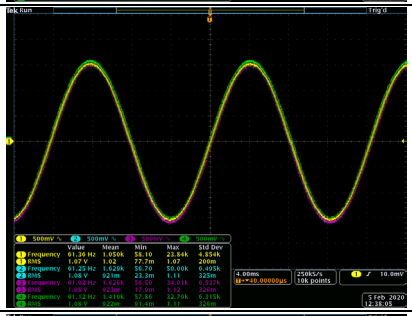
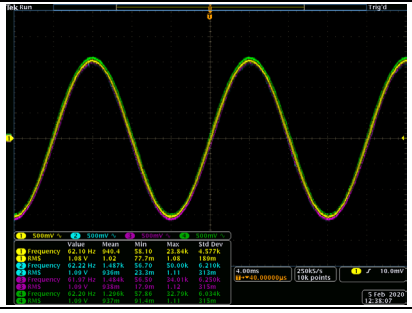
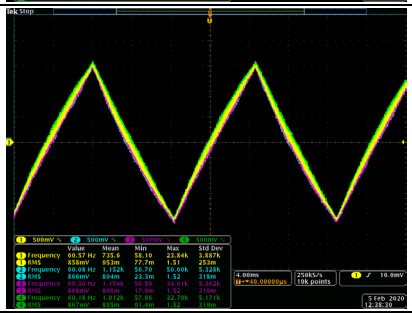
Command	Function name	Image
$ARLTX,30,3,0,0,0*00	FREQ 58	
$ARLTX,30,3,1,0,0*00	FREQ 59	
$ARLTX,30,3,2,0,0*00	FREQ 60	
$ARLTX,30,3,3,0,0*00	FREQ 61	
$ARLTX,30,3,4,0,0*00	FREQ 62	

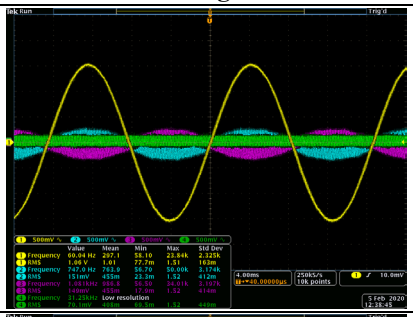
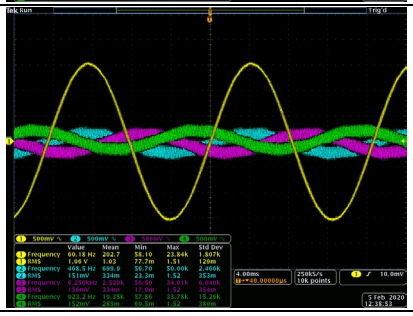
Table B-3 Command 30 (continued)

Command	Function name	Image
$ARLTX,30,4,0,0,0*00	THDF EVN	
$ARLTX,30,4,1,0,0*00	THDF ODD	
$ARLTX,30,4,2,0,0*00	THDF ALL	
$ARLTX,30,4,3,0,0*00	THDF SQR	
$ARLTX,30,4,4,0,0*00	THDF TRI	

B.4 Commands 120 and 130

These commands produce a simulated split-phase and three-phase source.

Table B-4 Commands 120 and 130

Command	Function name	Image
$ARLTX,120,0,0,0,0*00	120	
$ARLTX,130,0,0,0,0*00	130	

Appendix C. Airborne Reconnaissance and Targeting Multi-Mission Intelligence System (ARTEMIS) Python Test Script

This appendix appears in its original form, without editorial change.

This test script is used by ARTEMIS to control all available calibrator modules to perform a self-test of all input banks (sets of four channels).

```
# -*- coding: utf-8 -*-
""" Test behavior of SCB with Hardware Calibrator

NOTE: Since the HW calibrator goes from 0-3.3 V, the signal level
s are too
    large for the bnc1:1 attenuation mode. As a result, we only u
se the bnc10:1.
"""
import logging
import os
import time
import argparse

import numpy as np
import pytest
import core100
import numpy.testing as npt

import dlamp
import serial
from serial.tools import list_ports

logger = logging.getLogger(__name__)

""" Human-readable calibrator function and mode names. """
_FCNS = ["CALI", "TONE", "AMPL", "FREQ", "THDF"]
_MODES = [
    ["128", "x00", "x7f", "xff", "ERR"],
    ["60", "120", "180", "240", "300"],
    ["1/1", "1/2", "1/4", "1/8", "1/16"],
    ["58", "59", "60", "61", "62"],
    ["EVN", "ODD", "ALL", "SQR", "TRI"],
]
_VALUES = [
    [0, 0, 0, 0, 0],
    [60, 120, 180, 240, 300],
    [1, 0.5, 0.25, 0.125, 0.0625],
    [58, 59, 60, 61, 62],
    [0.25, 0.25, 0.35, 0.48, 0.12],
]

""" Parameter sweeps and tolerances. """
_CHAN_PARAM = [0, 1, 2, 3]
_CHAN_TOL = 0.90  # +/-90%
_TONE_PARAM = [
    0,
    1,
```

```
    2,
    pytest.param(3, marks=pytest.mark.xfail),
    pytest.param(4, marks=pytest.mark.xfail),
]
_TONE_TOL = 0.15  # +/-15%
_AMPL_PARAM = [0, 1, 2, 3, 4]
_AMPL_TOL = 0.15  # +/-15%
_FREQ_PARAM = [0, 1, 2, 3, 4]
_FREQ_TOL = 0.5  # +/-0.5 Hz
_THDF_PARAM = [
    pytest.param(0, marks=pytest.mark.xfail),
    1,
    pytest.param(2, marks=pytest.mark.xfail),
    pytest.param(3, marks=pytest.mark.xfail),
    4,
]
_THDF_TOL = 0.10  # +/-10%

try:
    api = core100.Core100()
    nchannels = api.System.GetNumberOfChannels()
    nbanks = nchannels // 4
    print(f"System has {nchannels} channels on {nbanks} banks")
except Exception:
    nchannels = 8
    nbanks = 2
    print(f"Using default channels and banks.")

_BANK_RANGE = list(range(nbanks))


def is_calibrator_device(name, max_reads=30):
    """ Connect to a device and determine if it is a Calibrator.
    """
    # Connect and throw away any welcome message (26 lines)
    logger.info(f"Identifying device: {name}")
    ser = serial.Serial(f"/dev/{name}", 115200, timeout=2)
    reads = 0
    msg = ser.readline()
    while msg and (reads < max_reads):
        msg = ser.readline()
        reads += 1
    # Check that this is a calibration unit
    msg = b"$ARLTX,1*00\r\n"
    ser.write(msg)
    resp = ser.readline()
    logger.debug(f"Sending: {msg}")
    logger.debug(f"Receivd: {resp}")
    try:
        ack, ntx, device, hw, sw, build = resp.decode().split(","
```

```
)
        if (ack == "<ACK") and (device == "1"):
            return True
    except Exception:
        pass
    return False


@pytest.fixture(scope="module")
def api():
    api = core100.Core100()
    yield api


@pytest.fixture(scope="module")
def calibrators():

    devices = [ser.name for ser in list_ports.comports()]
    logger.info(f"Found devices: {devices}")

    calibrators = [
        serial.Serial(f"/dev/{name}", 115200, timeout=2)
        for name in devices
        if is_calibrator_device(name)
    ]
    logger.info(f"Found calibrators: {[c.name for c in calibrator
s]}")

    yield calibrators


@pytest.fixture(scope="module")
def synchro():
    """ Return a ThreadedReceiver for synchrophasors.
    """
    synchro = dlamp.ThreadedReceiver(port=5683)
    synchro.start()
    yield synchro


@pytest.fixture(scope="module")
def relative():
    """ Return a ThreadedReceiver for synchrophasors.
    """
    relative = dlamp.ThreadedReceiver(port=5683)
    relative.start()
    yield relative


def get_synchro_x(s, b, offset):
```

```
    """ Generic get synchrophasor data.
    """
    sample = s.data[-1].data
    sample = sample[offset::5, -1]
    sample = sample[b * 4 : (b * 4 + 4)]
    return sample


def get_synchro_1(s, b):
    return get_synchro_x(s, b, 0)


def get_synchro_2(s, b):
    return get_synchro_x(s, b, 1)


def get_synchro_3(s, b):
    return get_synchro_x(s, b, 2)


def get_synchro_freq(s, b):
    return get_synchro_x(s, b, 3)


def get_synchro_thdf(s, b):
    return get_synchro_x(s, b, 4)


def set_frequencies(api, frequencies):
    """ Set ARTEMIS Filter frequencies.
    """
    logger.debug(f"Set filters to: {frequencies}")
    api.Fir1.SetFrequenciesAsFloats(*frequencies)


def set_calibrator(calibrators, TXN, FCN, MODE, CH, PHS):
    """ Set Calibrator waveform.
    """
    msg = f"$ARLTX,{TXN},{FCN},{MODE},{CH},{PHS}*00"
    want = f"<ACK,{TXN},{_FCNS[FCN]},{_MODES[FCN][MODE]},{CH},{PH
S}>"
    want = want.encode() + b"\r\n"
    for ser in calibrators:
        ser.write(msg.encode())
        resp = ser.readline()
        if want != resp:
            logger.warning(f"ARLTX FAIL ON: {ser.name}")
            logger.debug(f" Sent:   {msg}")
            logger.debug(f" Wanted: {want}")
            logger.debug(f" Recv'd: {resp}")
```

```
        else:
            logger.debug(f"ARLTX,{TXN},{FCN},{MODE} SET ON: {ser.
name}")


@pytest.mark.parametrize("CH", _CHAN_PARAM)
@pytest.mark.parametrize("bank", _BANK_RANGE)
def test_channels(api, calibrators, synchro, bank, CH):
    """ Test that each channel is providing data.
    """
    print()
    scale = 10
    TXN, FCN, MODE, CH, PHS = 20, 1, 0, CH, 0
    frequencies = [60, 60, 60]

    print(f"Testing Bank: {bank} CHAN: {CH}")
    set_frequencies(api, frequencies)
    set_calibrator(calibrators, TXN, FCN, MODE, CH, PHS)
    time.sleep(5.0)

    H1 = get_synchro_1(synchro, bank)
    H2 = get_synchro_2(synchro, bank)
    H3 = get_synchro_3(synchro, bank)
    for H in (H1, H2, H3):
        observed = np.abs(H)
        expected = np.zeros((4,))
        expected[CH] = 1 / scale
        # Loose tolerance, as we just want to ensure there is sig
nal
        tol = _CHAN_TOL * 1 / scale
        # Use absolute tolerance as relative tolerance has issues
with zero.
        npt.assert_allclose(observed, expected, atol=tol)


@pytest.mark.parametrize("MODE", _TONE_PARAM)
@pytest.mark.parametrize("bank", _BANK_RANGE)
def test_tones(api, calibrators, synchro, bank, MODE):
    """ Test that pure tones are registered with the correct ampl
itude.
    """
    print()
    scale = 10
    TXN, FCN, MODE, CH, PHS = 30, 1, MODE, 0, 0
    frequencies = [_VALUES[FCN][MODE]] * 3

    print(f"Testing Bank: {bank} TONE: {_MODES[FCN][MODE]}")
    set_frequencies(api, frequencies)
    set_calibrator(calibrators, TXN, FCN, MODE, CH, PHS)
    time.sleep(5.0)
```

```
    H1 = get_synchro_1(synchro, bank)
    H2 = get_synchro_2(synchro, bank)
    H3 = get_synchro_3(synchro, bank)
    for H in (H1, H2, H3):
        observed = np.abs(H)
        expected = np.ones((4,)) / scale
        tol = _TONE_TOL * 1 / scale
        npt.assert_allclose(observed, expected, atol=tol)


@pytest.mark.parametrize("bank", _BANK_RANGE)
def test_permutation(api, calibrators, synchro, bank):
    """ Test that setting and re-setting frequencies doesn't brea
k permutation.
    """
    print()
    scale = 10
    TXN, FCN, MODE, CH, PHS = 30, 1, 2, 0, 0

    print(f"Testing Bank: {bank} PERMUTATION")
    set_calibrator(calibrators, TXN, FCN, MODE, CH, PHS)

    for ii in range(4):
        set_frequencies(api, [60, 180, 300])
        time.sleep(5.0)

        H1 = get_synchro_1(synchro, bank)
        H2 = get_synchro_2(synchro, bank)
        H3 = get_synchro_3(synchro, bank)

        E1 = np.zeros((4,)) / scale
        E2 = np.ones((4,)) / scale
        E3 = np.zeros((4,)) / scale
        tol = _TONE_TOL * 1 / scale

        npt.assert_allclose(np.abs(H1), E1, atol=tol)
        npt.assert_allclose(np.abs(H2), E2, atol=tol)
        npt.assert_allclose(np.abs(H3), E3, atol=tol)


@pytest.mark.parametrize("MODE", _AMPL_PARAM)
@pytest.mark.parametrize("bank", _BANK_RANGE)
def test_ampl(api, calibrators, synchro, bank, MODE):
    """ Test variation in amplitude.
    """
    print()
    scale = 10
    TXN, FCN, MODE, CH, PHS = 30, 2, MODE, 0, 0
    frequencies = [60, 60, 60]
```

```
    print(f"Testing Bank: {bank} AMPL: {_MODES[FCN][MODE]}")
    set_frequencies(api, frequencies)
    set_calibrator(calibrators, TXN, FCN, MODE, CH, PHS)
    time.sleep(5.0)

    ampl = _VALUES[FCN][MODE]

    H1 = get_synchro_1(synchro, bank)
    H2 = get_synchro_2(synchro, bank)
    H3 = get_synchro_3(synchro, bank)
    for H in (H1, H2, H3):
        observed = np.abs(H)
        expected = ampl * np.ones((4,)) / scale
        tol = _AMPL_TOL * ampl / scale
        npt.assert_allclose(observed, expected, atol=tol)


@pytest.mark.parametrize("MODE", _FREQ_PARAM)
@pytest.mark.parametrize("bank", _BANK_RANGE)
def test_freq(api, calibrators, synchro, bank, MODE):
    """ Test off-nominal frequencies.
    """
    print()
    TXN, FCN, MODE, CH, PHS = 30, 3, MODE, 0, 0
    frequencies = [60, 60, 60]

    print(f"Testing Bank: {bank} FREQ: {_MODES[FCN][MODE]}")
    set_frequencies(api, frequencies)
    set_calibrator(calibrators, TXN, FCN, MODE, CH, PHS)
    time.sleep(5.0)

    delta_freq = _VALUES[FCN][MODE] - 60

    F = get_synchro_freq(synchro, bank)
    observed = np.real(F)
    expected = delta_freq * np.ones((4,))
    tol = _FREQ_TOL
    npt.assert_allclose(observed, expected, atol=tol)


@pytest.mark.parametrize("MODE", _THDF_PARAM)
@pytest.mark.parametrize("bank", _BANK_RANGE)
def test_thdf(api, calibrators, synchro, bank, MODE):
    """ Test off-nominal frequencies.
    """
    print()
    TXN, FCN, MODE, CH, PHS = 30, 4, MODE, 0, 0
    frequencies = [60, 180, 300]
```

```
    print(f"Testing Bank: {bank} THDF: {_MODES[FCN][MODE]}")
    set_frequencies(api, frequencies)
    set_calibrator(calibrators, TXN, FCN, MODE, CH, PHS)
    time.sleep(5.0)

    thdf = _VALUES[FCN][MODE]

    T = get_synchro_thdf(synchro, bank)
    observed = np.real(T)
    expected = thdf * np.ones((4,))
    tol = _THDF_TOL
    npt.assert_allclose(observed, expected, atol=tol)


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="hardware - configure FPGA and Core100 system
"
    )
    parser.add_argument("-d", "--debug", help="Log level DEBUG",
action="store_true")
    args = parser.parse_args()

    if args.debug:
        log_level = "--log-level=DEBUG"
    else:
        log_level = "--log-level=INFO"

    pytest.main(args=[os.path.abspath(__file__), "-s", "-v", log_
level])
```

List of Symbols, Abbreviations, and Acronyms

ADC	analog-to-digital converter
ARL	Army Research Laboratory
ARTEMIS	Airborne Reconnaissance and Targeting Multi-Mission Intelligence System
BNC	Bayonet Neil-Concelman, or British Naval Connector
ID	identification
IoT	Internet of Things
HIL	hardware in the loop
LED	light-emitting diode
LPF	low-pass filter
LUT	lookup table
PCB	printed circuit board
PWM	pulse width modulation
RC	resistor–capacitor
SRAM	static random-access memory
SWaP-C	size, weight, power, and cost
TEDS	transducer electronic datasheet
THDF	total harmonic distortion fundamental
USB	Universal Serial Bus

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLD DCI TECH LIB
1 (PDF)	DEVCOM ARL FCDD RLS EM B PARKS