VICOR

Introducing The Benefits Of Software To Your Hardware Design Workbench

IEEE Long Island Power Electronics Conference
November 7th 2024

C. R. Swartz; Senior Principal Engineer/Manager

Joshua Lapierre, Engineering Aid

Advanced Systems Design Engineering

Forward

- Has anyone ever designed a power supply that requires communication just to turn on?
- How about programming the power supply and trimming compensation, output etc.?
- Ever try to buy a turn-key test solution that works with your existing test equipment?
- These solutions also have to communicate with your power supply too!
- Often you have to buy a rack along with new test equipment to get automation.
- You would still have to write customized code and spend a lot of money (and time)

This seminar

It became necessary to dig in to developing software to help me in the product development. I don't profess to be an expert coder, but thanks to the great (free) and low cost tools available today, you don't have to be one to get some very meaningful things accomplished. There's always time to learn to be more proficient.

Today I hope to show you how to:

- Become familiar with Python® from a non-expert, practical point of view
- Utilize your existing test equipment and make measurements at low cost!
- Create a simple test code that can be used to control PMBus® power supplies
- Log and save your data in a common, simple format.

Benefits

- Ultra-low cost since most of the development tools are free
- Low development time as there are examples to learn from
- Your test programs are like building blocks, easy to reuse or modify
- Learn to be a better programmer as you go
- No internet connection required once you install all of the software
- Can use virtually any PC or old laptop as even old laptops are fast enough
- Final result is a very stable platform that is easily modifiable.

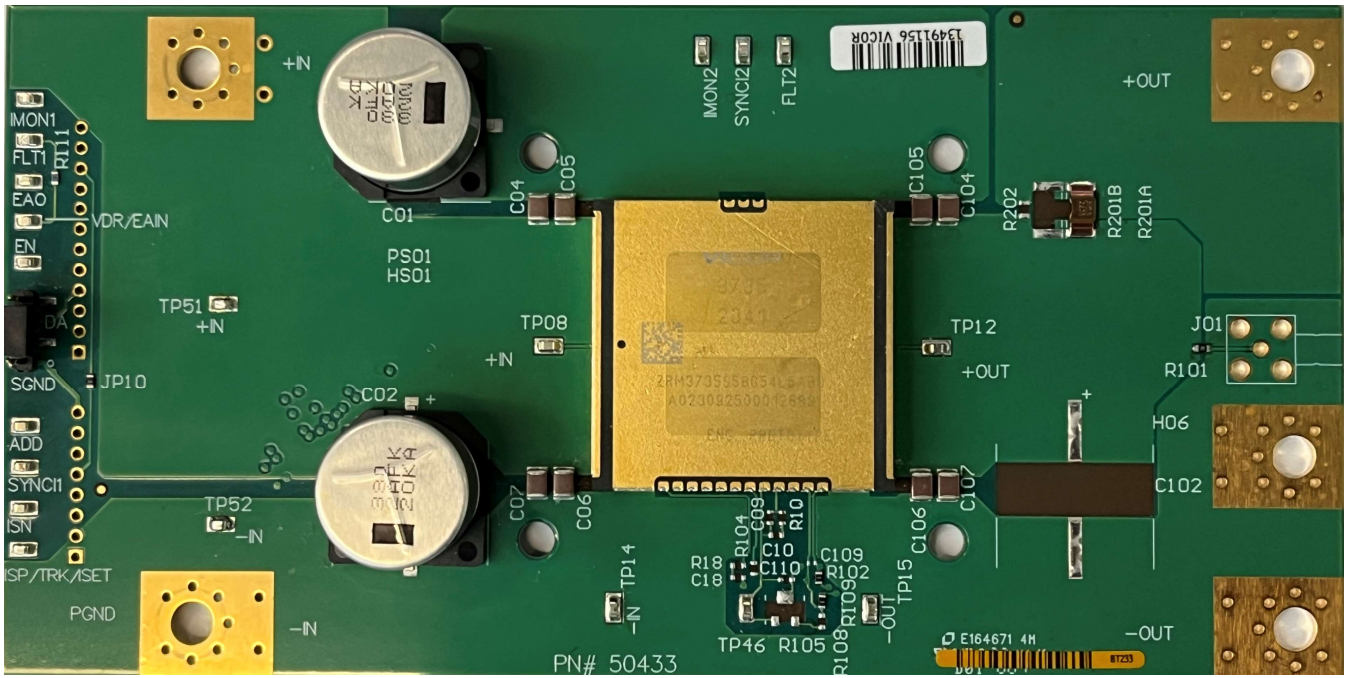
- With some C experience in my background, I had a working test station in (4) four days

Our examples use our new industrial PRM3735

- 36.6 x 35.4 x 7.4mm SM-ChiP™
- Weight: 40g
- Wide input range 31 – 58V_{DC}
- Wide output range 36 – 54V_{DC}
- 99.2% peak efficiency (48V input, 48V output)
- Up to 2.5kW continuous operation
- 4.3kW/in^3 power density
- Thermally-adept SM-ChiP package
- PMBus®-compatible telemetry

Product Ratings	
V_{IN} = 31.0 – 58.0V	P_{OUT} = 2500W
V_{OUT} = 48V (36.0 – 54.0V Trim)	I_{OUT} = 52.1A

Evaluation board for the PRM3735



Comparing languages

Python®

- High-level language – python code is highly readable, and easy to learn. Allows dynamic typing
- Extensive libraries – provides diverse libraries for different applications
- Interpreted language – interprets code directly without converting to machine code first. Results in slower execution of code.

C:

- Mid-level language – C code has more complex syntax and lower level constructs such as pointers. Is more difficult to learn but offers fine grain control over memory
- Smaller libraries – provides a smaller amount of libraries but offers powerful low-level system functions
- Compiled language – source code is translated into machine code by the compiler before the computer can execute it. Results in efficient and fast code.

Downloading……

- Visit https://www.python.org/downloads/
- Python® is compatible with Windows, macOS®, and Linux®
- Download and execute an installer for the latest version of that is compatible with your OS

VICOR

Installing……..

- Click on install now or customize installation if you wish to choose your file location, here we are using version 3.8 on this machine

Additional software requirements

- NI-VISA is a national instruments driver that uses the VISA I/O standard – download from National Instruments Inc.

- PyVISA is a Python package that enables you to control various measurement equipment remotely via the python program using the NI-VISA program installed on your system

Equipment drivers

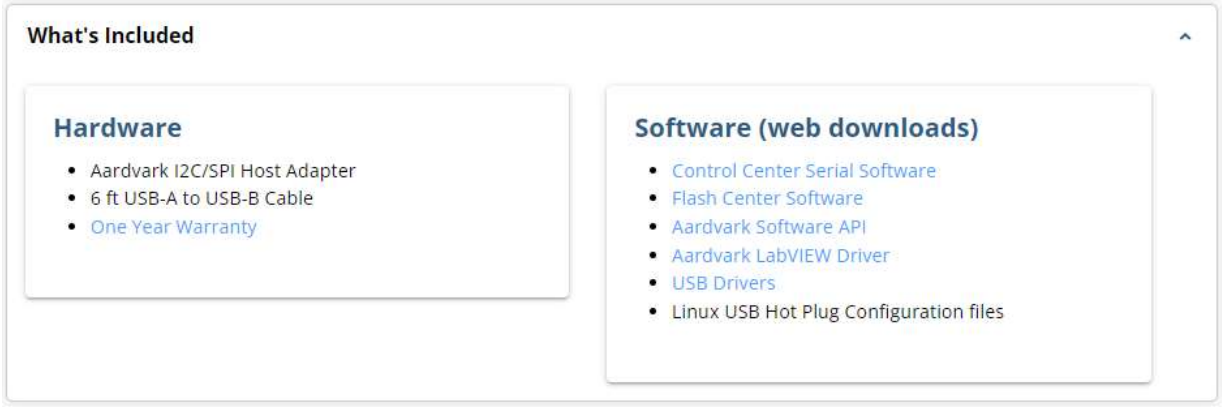
- PythonEquipmentDrivers are a collection of files that use pyVISA to communicate with ATE instruments
- Is available on GitHub at https://github.com/AnnaGiasson/PythonEquipmentDrivers

Serial host adapter

- The Aardvark™ is a host adapter that allows the communication between Windows, Linux® , or macOS® through USB to an external device using I2C or SPI protocols
- Total Phase™ offers drivers, API and Control Center Serial Software to use with the Aardvark unit

Downloading……USB driver and GUI

- Visit https://www.totalphase.com/products/aardvark-i2cspi/
- Scroll down the page and click the drop down labeled “What’s Included”
- Download Control Center Serial Software, Aardvark™ Software API, USB Drivers



Serial host adapter software description

- Control Center Serial Software – provides an interface to communicate with an Aardvark™ I2C/SPI Host Adapter. Useful for testing connections and debugging, but not suitable to ATE applications as it can't be synched to your test station code
- Aardvark™ Software API – provides Application Programming Interface and Python® libraries as well as other programming languages
- USB Drivers – software that enables the computer to communicate to the serial device through a USB port

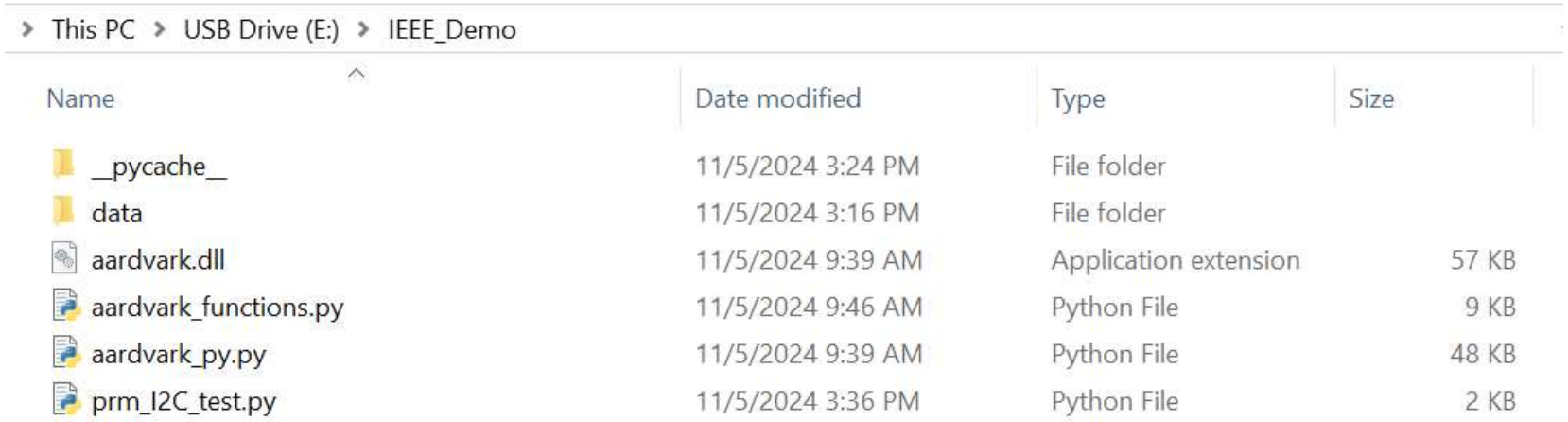
Debuggers

- PyScripter is an open-source Integrated Development Environment (IDE) for Python®
- PyScripter includes an integrated debugger, which features remote debugging, variable and watch windows, conditional breakpoints, post-mortem analysis, and the ability to run or debug files without saving them first
- PyScripter was used in the development and testing of this project and will be used going forward in this presentation
- There are many other options available, browse and choose one you like best

Set-up files

- Open the downloaded file “aardvark-api-windows-x86_64-v6.00” from Aardvark™ Software API
- The python folder will have the API named “aardvark_py.py”, and the DLL file named “aardvark.dll”
- Copy “aardvark_py.py”, and “aardvark.dll” and place them into your project file
- The python folder has script examples using “aardvark_py.py”
  - This folder provides multiple useful examples that can help you to understand how the module is used

Set-up files



Our telemetry module

- The goal was to create a module named “aardvark_functions.py” that can be called to perform PMBus® commands from a main test program, using the API supplied by Total Phase
- These functions will constitute complete commands in a simple and compact format
- “aardvark_functions.py” will then be used in conjunction with “PythonEquipmentDrivers” in order to create tests that evaluate the telemetry and control capabilities of a PRM3735 unit

Creating needed functions

- Two types of sub-functions need to be created
  - Command function
  - Conversion functions

- Command function – writes a command to an external PMBus® compatible device, then reads the device response
- Conversion functions – takes the external device response and converts it to the desired output format

Defining constants

- Constants will be used to store the address and data to be sent to the external device
- “aardvark_py.py” uses the Python® module array
- Command constants need to be defined using arrays of data type “B” (int, 1 byte min)
- Example: OPERATION_ON = array('B', [**0x01**, **0x84**])
- Pre-defined constants will be used as arguments in the command function to send commands to the unit

PRM3735 commands and data format

Supported Command List and Supported Commands Transaction Type

Command Name	Command Code	Function	Default Data Content	SMBus Write Transaction	SMBus Read Transaction	Number of Data Bytes	Data Format	PEC
OPERATION	01h	PMBus enable/disable	84h	Send Byte	Read Byte	1	bit	Supported
CLEAR_FAULTS	03h	Clear fault status register	n/a	Send Byte	n/a	0	bit	Unsupported
STORE_USER_CODE	17h	Writes variable parameter to non-volatile memory	n/a	Write Byte	n/a	1	bit	Unsupported
CAPABILITY	19h	PRM key capabilities set by factory	A0h	n/a	Read Byte	1	bit	Supported
VOUT_MODE	20h	Format for VOUT_COMMAND	17h	n/a	Read Byte	1	bit	Supported
VOUT_COMMAND	21h	Set PRM output voltage	6000h	Write Word	Read Word	2	ULINEAR16	Supported
VOUT_TRANSITION_RATE	27h	Set PRM output voltage slew rate in operation	1900h	Write Word	Read Word	2	LINEAR11	Supported
IOUT_OC_FAULT_LIMIT	46h	Set PRM constant current limit	E3F0h	Write Word	Read Word	2	LINEAR11	Supported
STATUS_BYTE	78h	Fault Readback	n/a	n/a	Read Byte	1	bit	Supported
STATUS_WORD	79h	Generic Fault Readback	n/a	n/a	Read Word	2	bit	Supported
READ_VIN	88h	PRM Input Voltage	n/a	n/a	Read Word	2	LINEAR11	Supported
READ_VOUT	8Bh	PRM Output Voltage	n/a	n/a	Read Word	2	ULINEAR16	Supported
READ_IOUT	8Ch	PRM Output Current	n/a	n/a	Read Word	2	LINEAR11	Supported
READ_TEMPERATURE_1	8Dh	PRM Temperature at Regulator Controller	n/a	n/a	Read Word	2	LINEAR11	Supported
MFR_ID	99h	Manufacturer ID	"VI"	n/a	Block Read	2	ASCII	Unsupported
IC_DEVICE_ID	ADh	Device identification	"4210008"	n/a	Block Read	7	ASCII	Unsupported
MFR_STATUS_FAULTS	F0h	PRM Specific Faults	n/a	n/a	Read Word	4	bit	Supported

Opening and closing the port

- In order to communicate with the Aardvark™ a port needs to be opened with the aa_open() function
- The function needs to be placed in your main program before any commands are sent
- The function takes the desired port number as an argument, in this case zero, and returns the handle value
- The handle variable should be saved to the aardvark_functions module, to be used in the command function which will facilitate communication to the Aardvark™ unit

```
port = 0
af.handle = aa_open(port)
```

- At the end of the program it is good practice to close the port using the aa_close() function, which uses the handle as an argument

```
aa_close(af.handle)
```

VICOR

Address

- The command function needs to know the child address of the unit it is controlling
- Find the address of the unit to be communicated with and save it to a variable for the command function at the top of the aardvark_functions module for the command function to use

```
child_addr = int('0x69', 0)
```

Command function

- The command function will be constructed with the aa_i2c_read function and aa_i2c_write function
- It will always write a command to the PMBus line and read its response.
- If there is no anticipated response, then run the command without saving the returned value

Command function

```
def command(data_out, byte_num):
    """
    convert_bit(array, bytes)
    sends PMBus signal from aardvark to unit and reads units response
    Returns:
        hex value
    """
    count = aa_i2c_write(handle, child_addr, AA_I2C_NO_FLAGS, data_out)
    if count < 0:
        print("error: %s" % aa_status_string(count))

    count, data_in = aa_i2c_read(handle, child_addr, AA_I2C_NO_FLAGS, byte_num)
    if count < 0:
        print("error: %s" % aa_status_string(count))
    return data_in
```

Conversion functions created

- Takes the output of the command function and converts it to a useful value
  - convert_linear11 - converts LINEAR11 binary value to real-world value
  - convert_ulinear16 - converts ULINEAR16 binary value to real-world value
  - convert_hex - converts bytes to hex
  - convert_bit - converts byte from bit data format to hex
  - convert_ascii - converts byte from ASCII data format to hex

Conversion functions

```
def convert_ulinear16(array_value):
    """
    convert_ulinear16(array)
    converts ulinear16 binary code to real world value.
    Returns:
        ulinear16 decoded value
    """
    mantissa = array_value[1]*256 + array_value[0]
    return mantissa * 2**-9
```

Convert from LINEAR11

- The convert_linear11(array) function takes the output of a command and converts the array from LINEAR11 to a real-world value
- The array received will have the high byte and low byte stored separately as decimal numbers

Convert from LINEAR11

```
def convert_linear11(array_value):
    """
    convert_linear11(array)
    converts linear11 binary value to real world value.
    Returns:
        floating-point real world value
    """
    binary_value = format(array_value[1]*256 + array_value[0], '#018b')
    mantissa = int(binary_value[7:8])*-1024 + int(binary_value[8:18], 2)
    exponent = int(binary_value[2:3])*-16 + int(binary_value[3:7], 2)
    return mantissa * 2**exponent
```

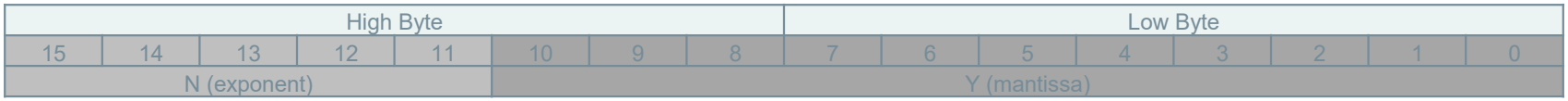
Binary conversion

- The high byte of the array is multiplied by 256, so that when converted to binary the high byte will be shifted 8 digits left.
- The high and low byte are then added together to produce a single value that is then converted to a string using the format function.
- The format function is specified with the code ‘#018b’.
- The ‘b’ specifies that the output should be in binary.
- The ‘#’ adds the prefix ‘0b’ to the output value.
- The ‘018’ directs the output length to be 18 characters long.

```
binary_value = format(array_value[1]*256 + array_value[0], '#018b')
```

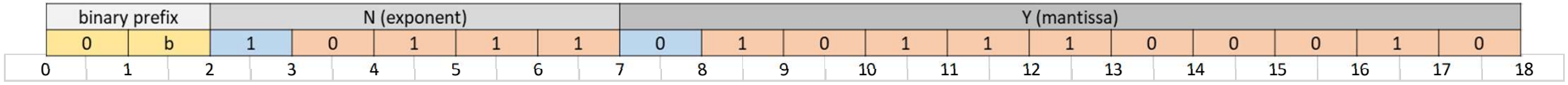
VICOR

Linear11

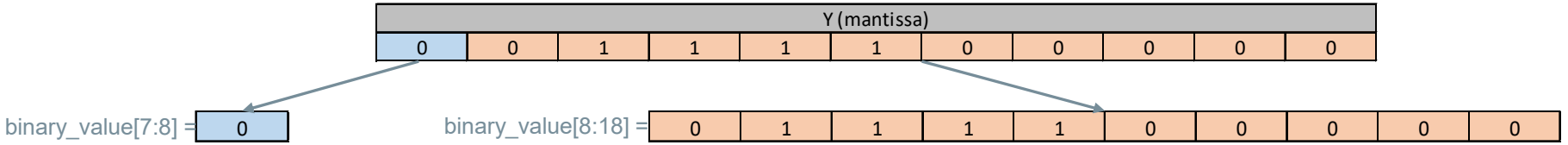


- The received array is two bytes long
- The exponent value is encoded into the 5 most significant bits, while the mantissa is in the 11 least significant bits
- The mantissa and exponent are signed numbers in twos compliment form

Finding the mantissa and exponent



- The mantissa and exponent value need to be extracted from their two's compliment form.
- The first value of each number will be selected separately from the rest, then multiplied by its negative place value, and finally added to the rest of the number



$$mantissa = binary\ value[7{:}8] * -1024 + binary\ value[8{:}18]$$

```
mantissa = int(binary_value[7:8])*-1024 + int(binary_value[8:18], 2)
exponent = int(binary_value[2:3])*-16 + int(binary_value[3:7], 2)
```

VICOR

Solving for the real-world value

$$X = Y * 2^{N}$$

X is the real-world value

Y is the mantissa

N is the exponent

- The mantissa value and the exponent value are now plugged into the LINEAR11 equation to find the real-world value
- The result is then returned as the functions output value

```
return mantissa * 2**exponent
```

Pairing functions……..

- In order to create PMBus® commands; we will combine constants, the command function and conversion functions
- The constant will determine the command code address byte and any additional data bytes sent
- The command function will send the message to the unit then return the response
- The conversion function will convert the response to the desired format

```
def run_vout_command(v_out_set):
    """
    run_vout_command()
    sends PMBus VOUT_COMMAND command to unit then converts response from
    ulinear16 data format to hex
    Returns:
        hex value
    """
    return convert_ulinear16(command(v_out_set, 2))
```

VICOR

Available command functions

- run_operation()
- run_operation_on()
- run_operation_off()
- run_vout_mode()
- run_read_temperature()
- run_vout_command()
- run_slew_read()
- run_read_vin()
- run_read_vout()
- run_read_iout()
- run_capability()
- run_status_byte()
- run_status_word()
- run_mfr_id()
- run_mfr_status_faults()
- run_ic_device_id()

VICOR

Our example program

- Tests the PRM3735 unit at different V_{IN}, I_{OUT} and V_{OUT} conditions
- Use developed commands to set V_{OUT} and to measure V_{IN}, V_{OUT}, I_{OUT}, and internal temperature in degrees C
- Saves data to datum as a list

Main program

Create the directory in advance

Define the port

Assign the port

Main loop

1 second delay per pass

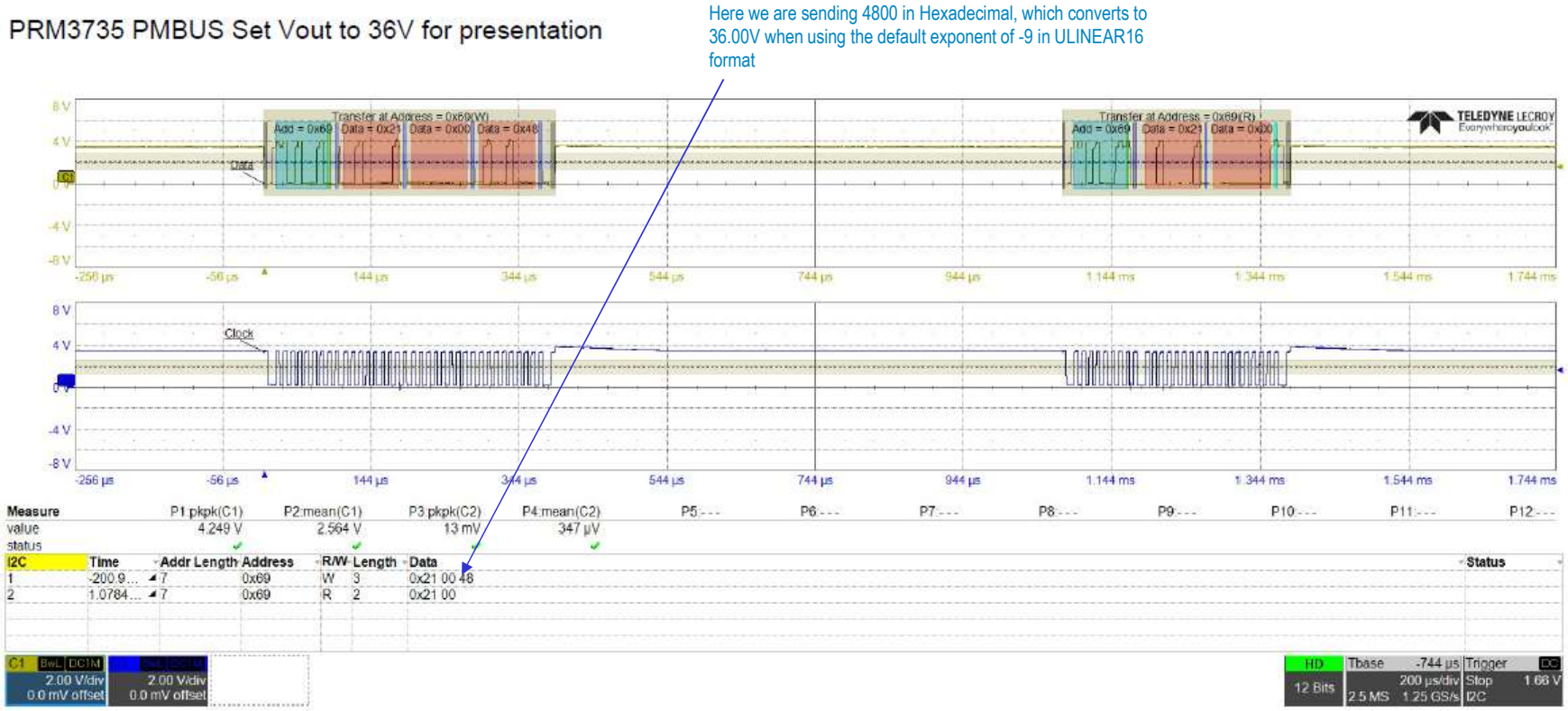
Write the file when done

Close the port

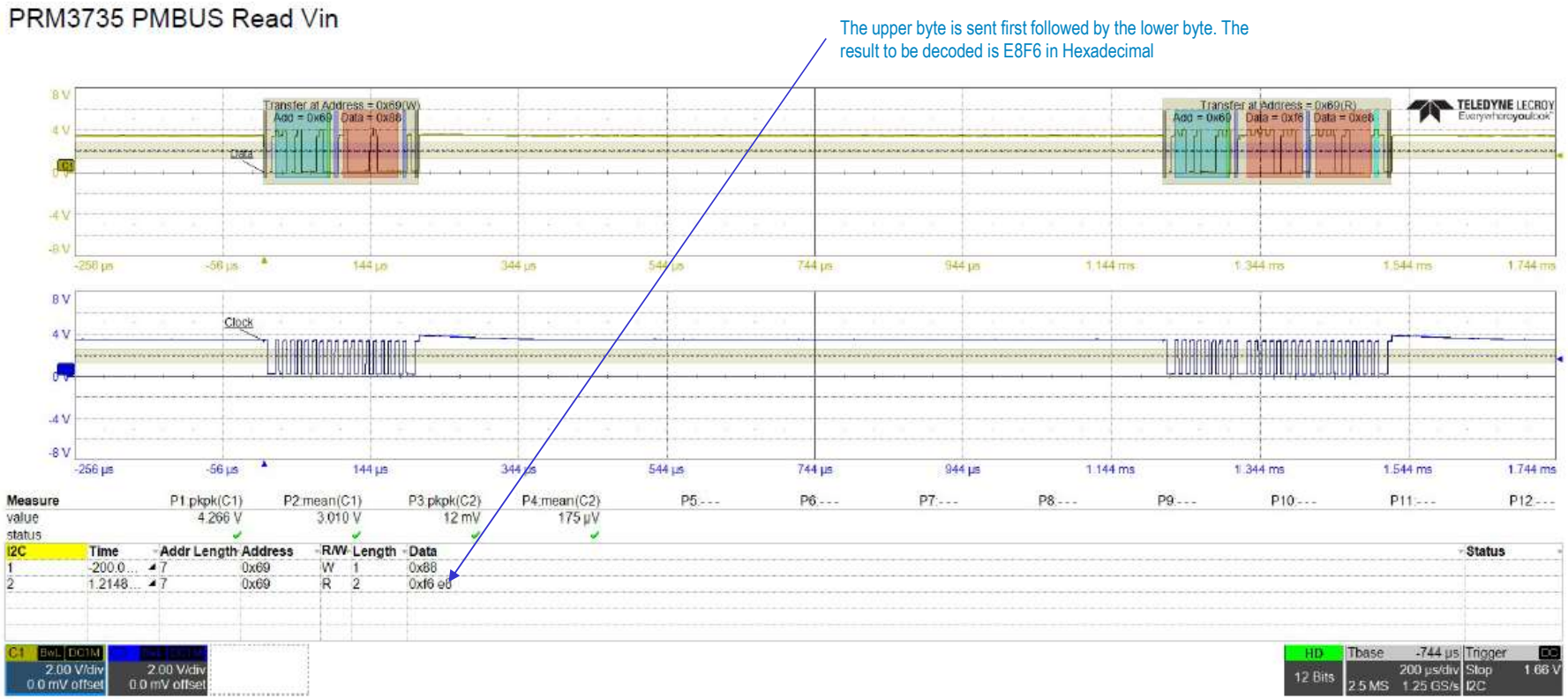
```python
# import modules
import pythonequipmentdrivers as ped
from aardvark_py import *
import aardvark_functions as af
from time import sleep
import csv

# file save location
data_file_name = "C:\\Users\\JLapierre\\Documents\\temp\\test_file.csv"

# connect to meter using pyvisa
v_out_meter = ped.multimeter.Keysight_34461A('USB0::0x2A8D::0x1601::MY57102231::INSTR')

# open port 0 and get handle
port = 0
af.handle = aa_open(port)

# test loop
v_out_conditions = [af.VOUT_COMMAND_36V, af.VOUT_COMMAND_48V, af.VOUT_COMMAND_54V]
data = [['v_in', 'v_out(PMBus)', 'v_out(meter)', 'i_out', 'temp']]
for v_out_set in v_out_conditions:
    af.run_vout_command(v_out_set)
    sleep(0.2)
    datum = [af.run_read_vin(),
             af.run_read_vout(),
             v_out_meter.measure_voltage(),
             af.run_read_iout(),
             af.run_read_temperature(),]
    data.append(datum)
    sleep(1)

# log data
with open(data_file_name, "w", newline='') as file:
    writer = csv.writer(file)
    for row in data:
        writer.writerow(row)
print(f'data saved to: {data_file_name}')

# close port
aa_close(af.handle)
```

VICOR

Set the output voltage



Read input voltage telemetry



Read input voltage telemetry - checking the math

- E8F6 in Hex is “1110 1000 1111 0110” in Binary

The first 5 bits are a signed integer. This represents N.
So we get -16 + 13 = -3

The remaining 11 bits convert to 246 Decimal, which is our data to be converted

- Our measured value in our program should be 30.75 Volts
- We can check this in our test results .CSV file

$$Low\_Bits\_Value := 246$$

$$Low\_Bits\_Value = 0f6h$$

$$N_{data} := -3$$

$$N_{data} = -3h$$

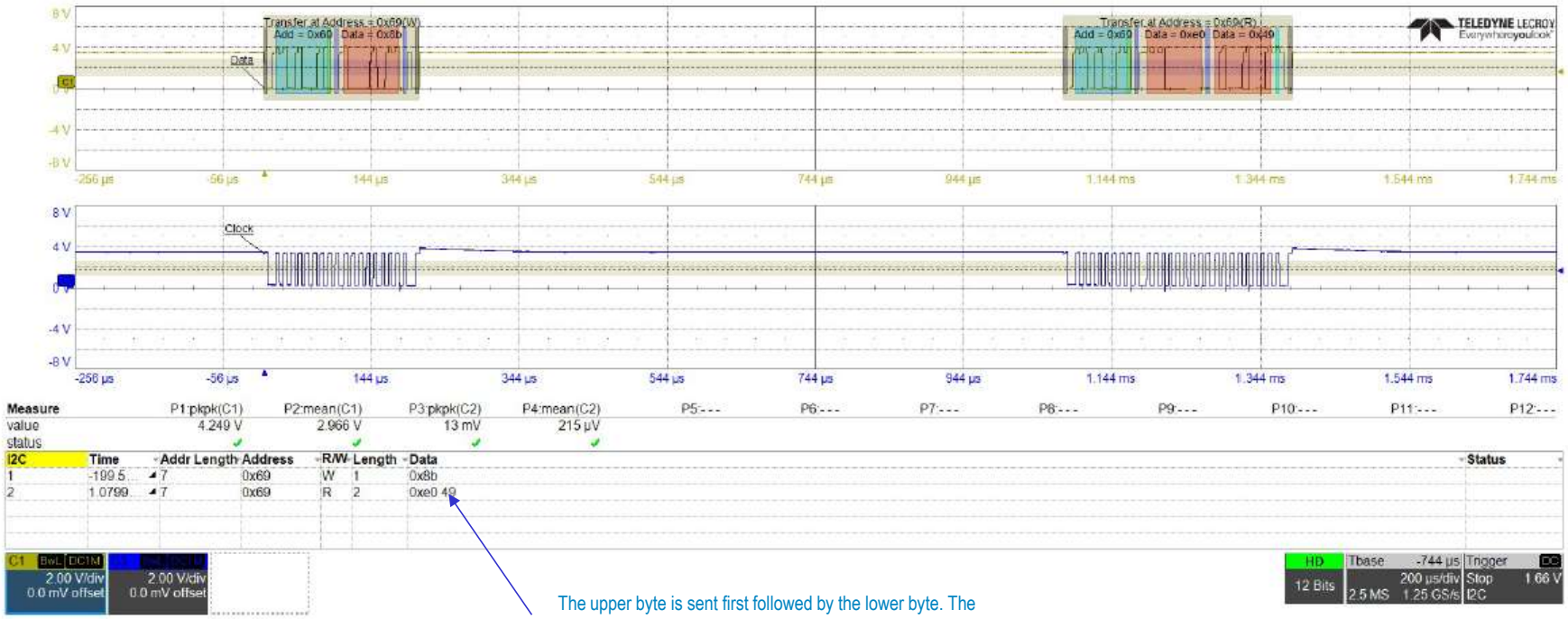
$$Real_X := Low\_Bits\_Value \cdot 2^{N_{data}}$$

$$Real_X = 30.75$$

VICOR

Read output voltage telemetry

PRM3735 PMBUS Read Vout



The upper byte is sent first followed by the lower byte. The result to be decoded is 49E0 in Hexadecimal

Read output telemetry- ULINEAR16 math check

- 49E0 in Hex is “0100 1001 1110 0000” in Binary

All of the read 16 bits are used for higher resolution and convert to 18912 Decimal, which is our data to be converted

- Exponent is fixed in this case at -9 by design

- Our measured value in our program should be 36.938 Volts
- We can check this in our test results .CSV file

$Read\_Data\_Value := 18912$

$Read\_Data\_Value = 49e0h$

$N_{exp} := -9$

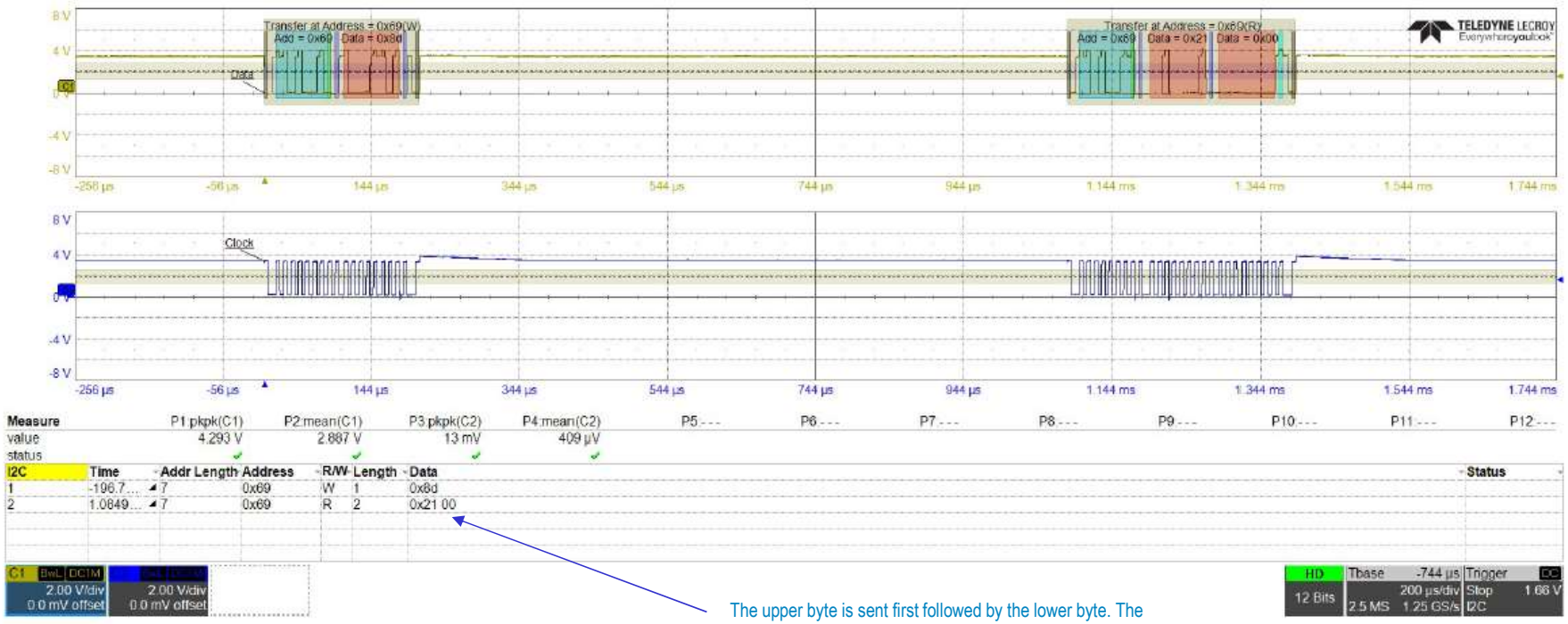
$N_{exp} = -9h$

$Real_{ulinear16} := Read\_Data\_Value \cdot 2^{N_{exp}}$

$Real_{ulinear16} = 36.938$

Read temperature command LINEAR11

PRM3735 PMBUS Read Temp



The upper byte is sent first followed by the lower byte. The result to be decoded is 0021 in Hexadecimal

Read temperature command- LINEAR11 math

- 0021 in Hex is “0000 0000 0010 0001” in Binary

The first 5 bits are a signed integer. This represents N. So N in this case is zero

The remaining 11 bits convert to 33 Decimal, which is our data to be converted

- Our measured value in our program should be 33 C
- We can check this in our test results .CSV file

$$Read\_Temp\_Value := 33$$

$$Read\_Temp\_Value = 21h$$

$$N_{expt} := 0$$

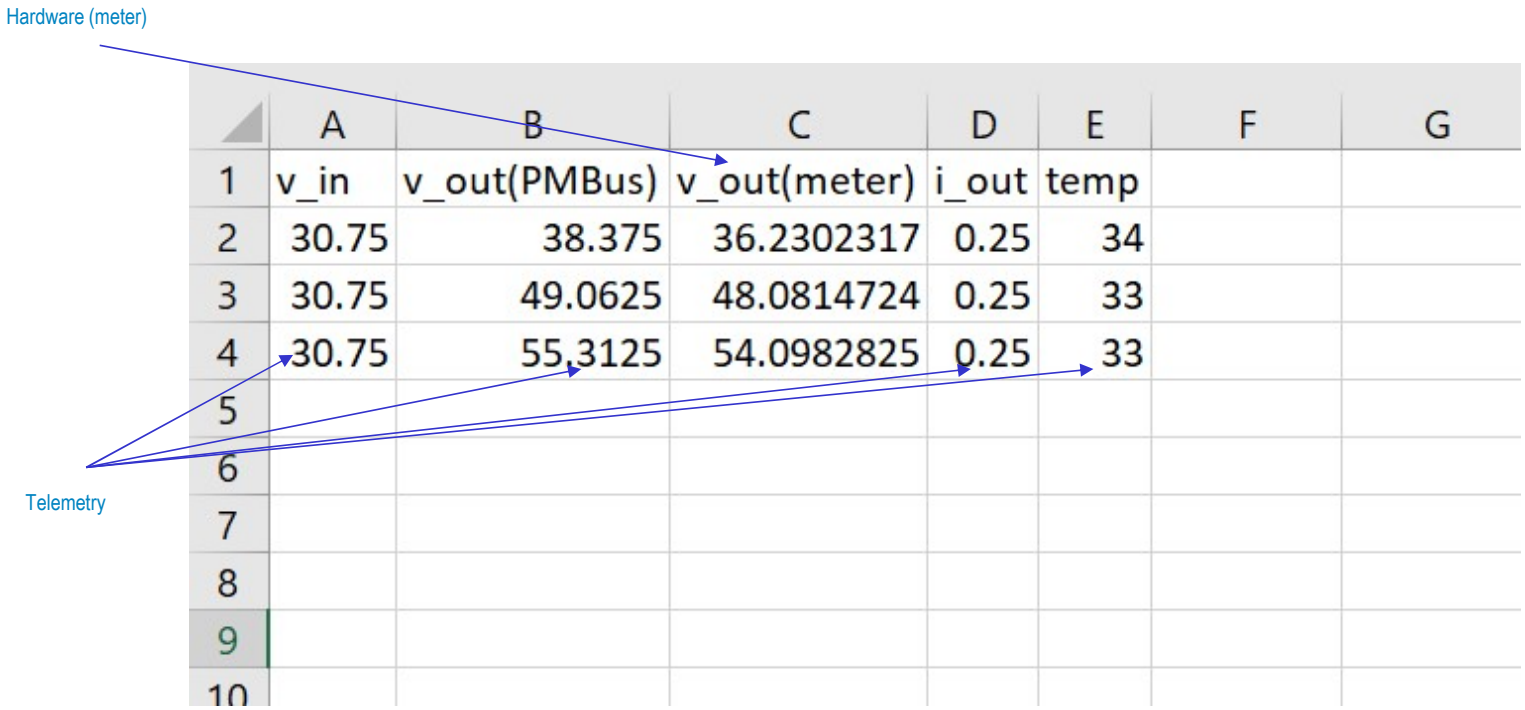
$$N_{expt} = 0h$$

$$Real_{Temp} := Read\_Temp\_Value \cdot 2^{N_{expt}}$$

$$Real_{Temp} = 33$$

VICOR

Test file produced by our program (testfile.csv)



	A	B	C	D	E	F	G
1	v_in	v_out(PMBus)	v_out(meter)	i_out	temp		
2	30.75	38.375	36.2302317	0.25	34		
3	30.75	49.0625	48.0814724	0.25	33		
4	30.75	55.3125	54.0982825	0.25	33		
5							
6							
7							
8							
9							
10							

Closing comments

- We have shown how to create a simple test program in Python®
- This program can control a power supply and measure telemetry using PMBus®
- It can also measure external voltages using a bench DMM through USB
- Both measurements can be combined in a .CSV data file which can then be imported
- This simple program can and has been extended to include scopes, oven control, chiller control, network analyzers, current monitors, loads, sources etc. to make a complete, very low cost, customizable and stable automated test system
- It is my hope that you will be able to embark on creating your own automated test bench!

Acknowledgements

- Windows is a trademark of Microsoft Corportation
- NI-VISA is a registered trademark of National Instruments Corporation
- macOS® is a registered trademark of Apple Inc.
- Python and Python® are registered trademarks of the Python Software Foundation
- PMBus® and PMBus are registered trademarks of the System Management Interface Forum Inc. (SMIF Inc.)
- Linux® is a registered trademark of Linus Torvalds in the U.S. and other countries
- Aardvark™, Aardvark I2C/SPI, Control Center and Total Phase are all trademarks of Total Phase Inc.

- Other company names and product names may be service marks, trademarks or registered trademarks of their respective companies

Questions?