Connecting to Hardware

SUMMARY: *In this lab, you'll learn how to setup and connect to your ChipWhisperer hardware. We'll also cover how to build firmware for your target microcontroller, how to capture power traces, and how to communicate with target devices.*

LEARNING OUTCOMES:

- Setting up your ChipWhisperer Hardware
- Using the ChipWhisperer Python API to connect to your hardware
- Communication with the target
- Capturing a power trace

Prerequisites

Hold up - before continuing, ensure you have done the following:

- ☑ Validated your setup using the ChipWhisperer Setup Test notebook.
- ☑ Run through the Jupyter introduction.

Physical Setup

ChipWhisperer-Nano

The ChipWhisperer-Nano is a single-board device. It includes the capture hardware, along with a built in STM32F0 target. To use this target, simply plug in the device, then go get a drink to reward yourself:


CW-Nano Plugged In

For the tutorials, you should use `PLATFORM='CWNANO'`.

Connecting to ChipWhisperer NANO

Now that your hardware is all setup, we can now learn how to connect to it.

In this board, we have two programable chips, one is the 'Scope' to the left of the image and the other is the 'Target', to the right of the image. From Jupyter, the Scope is controlled by the `scope` object, while the Target is controlled by the `target` object.

Conneting to the Scope

We can connect to the ChipWhisperer with:

In [1]:
```
import chipwhisperer as cw
scope = cw.scope()
```

Hopefully, this command does not return any errors. If it does, please review the Common Issues section of the Assignment.

By default, ChipWhisperer will try to autodetect the type of device your're running. There are many Scope boards by ChipWhisperer, but in our case, we use the the CWNano. See API documentation for manually specifying the scope type. We can verify that the connection actually happened, and see the scope initialized parameters as follows.

In [2]:
```
%whos
scope
```

```
Variable   Type      Data/Info
-----------------------------
cw         module    <module 'chipwhisperer' f<...>pwhisperer\\__init__.py'>
scope      CWNano    ChipWhisperer Nano Device<...>= 0\n    ext_offset = 0\n
```

Out[2]:
```
ChipWhisperer Nano Device
fw_version =
    major = 0
    minor = 65
    debug = 0
io =
    tio1         = None
    tio2         = None
    tio3         = None
    tio4         = high_z
    pdid         = True
    pdic         = False
    nrst         = True
    clkout       = 7500000.0
    cdc_settings = bytearray(b'\x01\x00\x00\x00')
adc =
    clk_src  = int
    clk_freq = 7500000.0
    samples  = 5000
glitch =
    repeat     = 0
    ext_offset = 0
```

The first command `%whos` display all the variables in the current notebook. In our case, it should display:

Variable Type Data/Info

cw module <module 'chipwhisperer' f<...>pwhisperer\init.py'>

scope CWNano ChipWhisperer Nano Device<...>= 0\n ext_offset = 0\n

The second command will display the parameters of the `scope` object. Here we should see the firmware version, the `io` pins of the Scope chip that control the Target chip, the ADC settings and the Glitch settings.

Some sane default settings can be set using:

In [3]:
```
scope.default_setup()
scope
```

Out[3]:
```
ChipWhisperer Nano Device
fw_version =
    major = 0
    minor = 65
    debug = 0
io =
    tio1         = None
    tio2         = None
    tio3         = None
    tio4         = high_z
    pdid         = True
    pdic         = False
    nrst         = True
    clkout       = 7500000.0
    cdc_settings = bytearray(b'\x01\x00\x00\x00')
adc =
    clk_src  = int
    clk_freq = 7500000.0
    samples  = 5000
glitch =
    repeat     = 0
    ext_offset = 0
```

Sets up sane capture defaults for this scope. You can check these settings by putting the cursor anywhere in the default_setup() and hitting shift+tab, then the + sign at the top right corner of the pop up window.

- 7.5MHz ADC clock
- 7.5MHz output clock, which means that both the ADC and target will be running at the same clock frequency, allowing us to get one sample per clock.
- 5000 capture samples
- tio1 = serial rx
- tio2 = serial tx
- glitch module off

File: c:\users\mtaha\chipwhisperer5_64\cw\home\portable\chipwhisperer\software\chipwhisperer\capture\scopes\cwnano.py

You can change any of these settings by directly assigning a value to the parameter. Here, we change the number of samples to the maximum of 50,000 samples. The specifications can be found in https://rtfm.newae.com/Capture/ChipWhisperer-Nano/

In [4]:
```
scope.adc.samples=50000
scope
```

Out[4]:
```
ChipWhisperer Nano Device
fw_version =
    major = 0
    minor = 65
    debug = 0
io =
    tio1         = None
    tio2         = None
    tio3         = None
    tio4         = high_z
    pdid         = True
    pdic         = False
    nrst         = True
    clkout       = 7500000.0
    cdc_settings = bytearray(b'\x01\x00\x00\x00')
adc =
    clk_src  = int
    clk_freq = 7500000.0
    samples  = 50000
glitch =
    repeat     = 0
    ext_offset = 0
```

Connecting to the Target

Connecting to the target device to the previously initialized scope:

In [5]:
```
target = cw.target(scope)
```

Similarly, we can inspect the `target` object as follows:

In [6]:
```
target
```

Out[6]:
```
SimpleSerial Settings =
        output_len             = 16
        baud                   = 38400
        simpleserial_last_read =
        simpleserial_last_sent =
```

It should show the output_len, the connection baud rate, and the connection protocol. The default connection protocol is the Simple Serial protocol.

And that's it! Your ChipWhisperer is now setup and ready to attack a target.

Note that, the Scope chip is already programmed by the firmware. However the Target chip is still empty. Yes, we did specify the communication protocol between the Scope and the Target chip, but the Target is not yet running any code. Next, we will program the Target chip.

Building and Uploading a Simple Target

The next step in attacking a target is to get some code built and uploaded onto it.

Target codes must be built on the system's command line. Luckily, thanks to Jupyter, we can run a command within a notebook as follows. %%bash will run a command on the bash on behalf of Jupyter

In [7]:
```
%%sh
cd ../../hardware/victims/firmware/simpleserial-base/
make PLATFORM=CWNANO CRYPTO_TARGET=NONE
```

```
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
make[1]: '.dep' is up to date.
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
.
Welcome to another exciting ChipWhisperer target build!!
arm-none-eabi-gcc (GNU Arm Embedded Toolchain 10-2020-q4-major) 10.2.1 20201103 (release)
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

.
Compiling:
    simpleserial-base.c ...Done!
.
Compiling:
    .././simpleserial/simpleserial.c ...Done!
.
Compiling:
    .././hal/stm32f0_nano/stm32f0_hal_nano.c ...Done!
.
Compiling:
    .././hal/stm32f0/stm32f0_hal_lowlevel.c ...Done!
.
Assembling: .././hal/stm32f0/stm32f0_startup.S
arm-none-eabi-gcc -c -mcpu=cortex-m0 -I. -x assembler-with-cpp -mthumb -mfloat-abi=soft -ffunction-sections -DF_CPU=7372800 -Wa,-gsta
bs,-adhlns=objdir-CWNANO/stm32f0_startup.lst -I.././simpleserial/ -I.././simpleserial/ -I.././hal -I.././hal/stm32f0 -I.././hal/stm32
f0/CMSIS -I.././hal/stm32f0/CMSIS/core -I.././hal/stm32f0/CMSIS/device -I.././hal/stm32f0/Legacy -I.././crypto/ .././hal/stm32f0/stm3
2f0_startup.S -o objdir-CWNANO/stm32f0_startup.o
.
LINKING:
    simpleserial-base-CWNANO.elf ...Done!
.
Creating load file for Flash: simpleserial-base-CWNANO.hex
arm-none-eabi-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.hex
.
Creating load file for Flash: simpleserial-base-CWNANO.bin
arm-none-eabi-objcopy -O binary -R .eeprom -R .fuse -R .lock -R .signature simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.bin
.
Creating load file for EEPROM: simpleserial-base-CWNANO.eep
arm-none-eabi-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" \
--change-section-lma .eeprom=0 --no-change-warnings -O ihex simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.eep || exit 0
.
Creating Extended Listing: simpleserial-base-CWNANO.lss
arm-none-eabi-objdump -h -S -z simpleserial-base-CWNANO.elf > simpleserial-base-CWNANO.lss
.
Creating Symbol Table: simpleserial-base-CWNANO.sym
arm-none-eabi-nm -n simpleserial-base-CWNANO.elf > simpleserial-base-CWNANO.sym
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
Size after:
   text    data     bss     dec     hex filename
   4644      12    1428    6084    17c4 simpleserial-base-CWNANO.elf
+--------------------------------------------------------
+ Default target does full rebuild each time.
+ Specify buildtarget == allquick == to avoid full rebuild
+--------------------------------------------------------
+--------------------------------------------------------
+ Built for platform CWNANO Built-in Target (STM32F030) with:
+ CRYPTO_TARGET = NONE
+ CRYPTO_OPTIONS = AES128C
+--------------------------------------------------------
```

Hopefully, you have the setup complete and the ARM compiler is correctly installed as part of the ChipWhesperer installation. If that is that case, the `make` command will build the code avaible in the directly specified to it. In our case, the Simple Serial Base folder in: cd ../hardware/victims/firmware/simpleserial-base/ It should report:

- Built for platform CWNANO Built-in Target (STM32F030) with:
- CRYPTO_TARGET = NONE
- CRYPTO_OPTIONS = AES128C

---

and generate a hex file in the same folder above. The output above should also specify the generated hex file as Creating load file for Flash: simpleserial-base-CWNANO.hex"

Then, in order to upload the generated hex file, we use the cw.program_target while specifying the Scope which connects us to the Target, the programmer which in this case the STM32Programmer, and the path to the hex file.

In [8]:
```
cw.program_target(scope, cw.programmers.STM32FProgrammer, "../../hardware/victims/firmware/simpleserial-base/simpleserial-base-CWNANO
```

```
Detected known STMF32: STM32F04xxx
Extended erase (0x44), this can take ten seconds or more
Attempting to program 4655 bytes at 0x8000000
STM32F Programming flash...
STM32F Reading flash...
Verified flash OK, 4655 bytes
```

This should upload the hex file to the Target.

Please navigate to the folder ../hardware/victims/firmware/simpleserial-base/ and check the simpleserial-base.c file that we just compiled

The main function initializes the platform, the uart, and the trigger setup. Then, it defines three SimpleSerial commands `p`, `k` and `x`.

- `p` accepts 16 bytes then performs the `get_pt` function
- `k` accepts 16 bytes then performs the `get_key` function
- `x` accept no bytes, and directly performs a reset.

The `get_pt`, `get_key` and `reset` functions are also defined in the file, just above the main(). Then, it puts the Target into an infinite loop of `simpleserial_get();` waiting for any comming command.

Please also check the `simpleserial-base-CWNANO.hex` that we generated and check its `Last Modified` time.

Next we'll lear how to communicate with the target and how to capture power traces.

Communication with the Target

Communication with targets, which is done through the `SimpleSerial target` object we got earlier, is grouped into two categories:

1. Raw serial via `target.read()`, `target.write()`, `target.flush()`, etc.
2. SimpleSerial commands via `target.simpleserial_read()`, `target.simpleserial_write()`, `target.simpleserial_wait_ack()`, etc.

The firmware we uploaded uses the simpleserial protocol (https://wiki.newae.com/SimpleSerial), so we'll start off with simpleserial. Later, we'll use the raw serial commands to send the same messages.

If you check the simpleserial-base firmware (`simpleserial-base.c`) you'll find that for the simpleserial `'p'` command, the target will read 16 bytes and execute the `get_pt()` function to perform a quick `trigger_high();` and `trigger_low();` followed by echo back the data by `simpleserial_put('r', 16, pt);`, which sends a `r` letter, followed by the 16 bytes of `pt`. Please recheck the hardware/victims/firmware/simpleserial-base/simpleserial-base.c for confirmation. Let's try that out now:

In [9]:
```
msg = bytearray([0]*16) #simpleserial uses bytearrays
target.simpleserial_write('p', msg)
```

Let's check if we got a response:

In [10]:
```
print(target.simpleserial_read('r', 16))
```

```
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
```

The `simpleserial_read('r', 16)` command performs a read of 17 bytes, and check if the first letter is indeed an `r` followed by 16 bytes of data.

Try sending different values, and check if you are getting the same values back.

In [11]:
```
msg = bytearray([11]*16) #simpleserial uses bytearrays
target.simpleserial_write('p', msg)
print(target.simpleserial_read('r', 16))
```

```
bytearray(b'\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b')
```

Lets also try sending the `'k'` command, which expects 16 bytes but does not perform any action on the trigger pins and returns only the value 0x00. Please recheck the hardware/victims/firmware/simpleserial-base/simpleserial-base.c for confirmation.

In [12]:
```
target.simpleserial_write('k', bytearray([0]*16))
```

If we checked the return string, it will give us a warning (not an error).

In [13]:
```
print(target.simpleserial_read('r', 16))
```

```
(ChipWhisperer Target WARNING|File SimpleSerial.py:410) Unexpected start to command: z
None
```

It is just a warning. The command `k` does return one byte that is `0x00`. However, since this value is not the expected `r` of `simpleserial_read('r', 16)`, it issues a warning. It also says that there are no 16 bytes of data after the first letter, hence the print of `None`. You can read a single byte by the `wait_ack` function as follows:

In [14]:
```
target.simpleserial_write('k', bytearray([0]*16))
print(target.simpleserial_wait_ack()) #should return 0
```

```
0
```

Another slightly lower-level method to send and recieved data is to use `target.write()` and `target.read()`. This method does not understand the nature of commands. Hence, you have to build the commands yourself and understand the return value yourself. One Simpleserial low-level message generally take the form:

`command_character + ascii_encoded_bytes + '\n'`
For our first command, `command_character='p'` and `ascii_encoded_bytes="00"*32` (keep in mind this isn't a binary `0x00`, it's ASCII `"00"`, which has a binary value of `0x3030`). Try resending the `'p'` command from earlier using `target.write()`: Note that, the `p` text must be exactly 16 bytes. Less bytes will keep the target waiting for more incoming bytes before executing the `get_pt()` function. So, you won't get any data back.

In [15]:
```
target.write('p' + "00112233445566778899AABBCCDDEEFF" + '\n') #fill in the rest here
```

A simple `target.read()` will return all the characters that have been sent back from the target so far. Let's see what the device returned to us:

In [16]:
```
recv_msg = ""
recv_msg += target.read() #you might have to run this block a few times to get the full message
print(recv_msg)
```

```
r00112233445566778899AABBCCDDEEFF
z00
```

Note that the first letter is `r` that was previously recognized and ignored by the `target.simpleserial_read()`. Also, the function displays the end of message indicator of `0x00`.

The simpleserial commands are usually sufficient for taking to simpleserial firmware, but you'll need the raw serial commands for some of the advanced functions.

Building and Uploading a customized Target

Now, lets do something that is more interesting. We will change the Target code to modify the `get_pt` response, recompile the code, upload the code to the target and test the new code.

- Open the file `hardware/victims/firmware/simpleserial-base/simpleserial-base.c` by Jupyter or any other text editor.
- Add the following code around line 40, between `trigger_high()` and `trigger_low()`

```
for (uint8_t i = 0; i < 16; i++ ){
        pt[i] = ( pt[i] + 1 ) % 256;
    }
```

- Save

Then, build

In [17]:
```
%%sh
cd ../../hardware/victims/firmware/simpleserial-base/
make PLATFORM=CWNANO CRYPTO_TARGET=NONE
```

```
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
make[1]: '.dep' is up to date.
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
.
Welcome to another exciting ChipWhisperer target build!!
arm-none-eabi-gcc (GNU Arm Embedded Toolchain 10-2020-q4-major) 10.2.1 20201103 (release)
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

.
Compiling:
    simpleserial-base.c ...Done!
.
Compiling:
    .././simpleserial/simpleserial.c ...Done!
.
Compiling:
    .././hal/stm32f0_nano/stm32f0_hal_nano.c ...Done!
.
Compiling:
    .././hal/stm32f0/stm32f0_hal_lowlevel.c ...Done!
.
Assembling: .././hal/stm32f0/stm32f0_startup.S
arm-none-eabi-gcc -c -mcpu=cortex-m0 -I. -x assembler-with-cpp -mthumb -mfloat-abi=soft -ffunction-sections -DF_CPU=7372800 -Wa,-gsta
bs,-adhlns=objdir-CWNANO/stm32f0_startup.lst -I.././simpleserial/ -I.././simpleserial/ -I.././hal -I.././hal/stm32f0 -I.././hal/stm32
f0/CMSIS -I.././hal/stm32f0/CMSIS/core -I.././hal/stm32f0/CMSIS/device -I.././hal/stm32f0/Legacy -I.././crypto/ .././hal/stm32f0/stm3
2f0_startup.S -o objdir-CWNANO/stm32f0_startup.o
.
LINKING:
    simpleserial-base-CWNANO.elf ...Done!
.
Creating load file for Flash: simpleserial-base-CWNANO.hex
arm-none-eabi-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.hex
.
Creating load file for Flash: simpleserial-base-CWNANO.bin
arm-none-eabi-objcopy -O binary -R .eeprom -R .fuse -R .lock -R .signature simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.bin
.
Creating load file for EEPROM: simpleserial-base-CWNANO.eep
arm-none-eabi-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" \
--change-section-lma .eeprom=0 --no-change-warnings -O ihex simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.eep || exit 0
.
Creating Extended Listing: simpleserial-base-CWNANO.lss
arm-none-eabi-objdump -h -S -z simpleserial-base-CWNANO.elf > simpleserial-base-CWNANO.lss
.
Creating Symbol Table: simpleserial-base-CWNANO.sym
arm-none-eabi-nm -n simpleserial-base-CWNANO.elf > simpleserial-base-CWNANO.sym
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
Size after:
   text    data     bss     dec     hex filename
   4644      12    1428    6084    17c4 simpleserial-base-CWNANO.elf
+--------------------------------------------------------
+ Default target does full rebuild each time.
+ Specify buildtarget == allquick == to avoid full rebuild
+--------------------------------------------------------
+--------------------------------------------------------
+ Built for platform CWNANO Built-in Target (STM32F030) with:
+ CRYPTO_TARGET = NONE
+ CRYPTO_OPTIONS = AES128C
+--------------------------------------------------------
```

Upload

In [18]:
```
cw.program_target(scope, cw.programmers.STM32FProgrammer, "../../hardware/victims/firmware/simpleserial-base/simpleserial-base-CWNANO
```

```
Detected known STMF32: STM32F04xxx
Extended erase (0x44), this can take ten seconds or more
Attempting to program 4655 bytes at 0x8000000
STM32F Programming flash...
STM32F Reading flash...
Verified flash OK, 4655 bytes
```

Test

In [19]:
```
msg = bytearray([0]*16) #simpleserial uses bytearrays
target.simpleserial_write('p', msg)
print(target.simpleserial_read('r', 16))
```

```
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
```

The target should reply with the same value that you sent, plus 1. You can reedit, rebuild and retest the code using the came cells above in-place, without copying them. This should demonstrate the easyness and effectiveness of modifying the target.

Capturing Traces

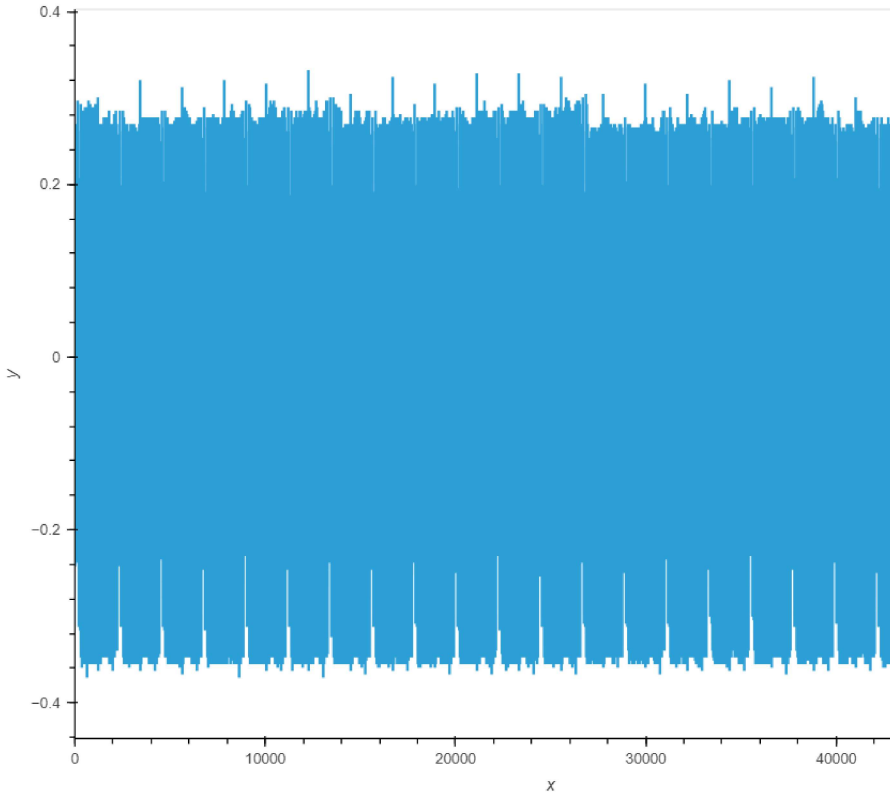
The last step in this tutorial is to capture the power trace of the target chip while operating some code. To capture a trace:

1. Arm the ChipWhisperer with `scope.arm()`. It will begin capturing as soon as it is triggered by the `trigger_high();` and stops at the indicatednumber of samples in the Scope object.
2. `scope.capture()` will read back the captured power trace, blocking until either ChipWhisperer is done recording, or the scope times out. Note that the error return will tell you whether or not the scope timed out. It does not return the captured scope data.
3. You can read back the captured power trace with `scope.get_last_trace()`.

`simpleserial_base` will trigger the ChipWhisperer when we send the `'p'` command. Try capturing a trace now:

In [20]:
```
msg = bytearray([0]*16)
scope.arm()
target.simpleserial_write('p', msg)
scope.capture()
cw.plot(scope.get_last_trace())
```

Out[20]:



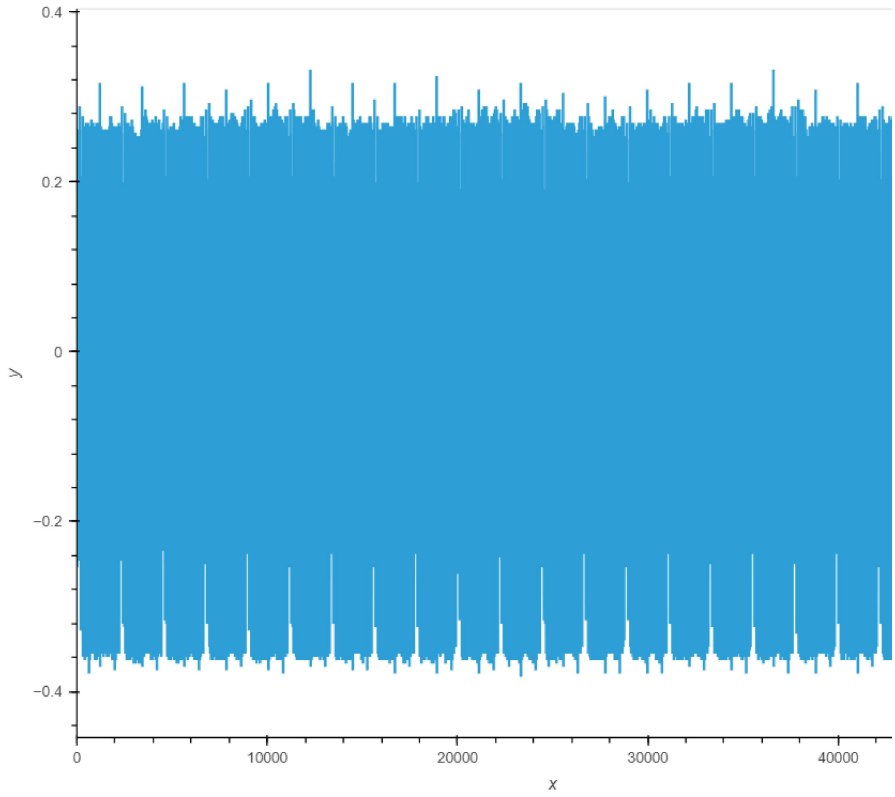
ChipWhisperer also has a `capture_trace()` convience function that:

1. Optionally sends the `'k'` command
2. Arms the scope
3. Sends the `'p'` command
4. Captures the trace
5. Reads the return `'r'` message
6. Returns a `Trace` class that groups the trace data, `'p'` message, the `'r'` message, and the `'k'` message

It isn't always the best option to use, but it's usually sufficient for most simpleserial applications

In [21]:
```
msg = bytearray([0]*16)
trace = cw.capture_trace(scope, target, msg)
cw.plot(trace.wave)
```

Out[21]:



Match the code to its power trace

Now, lets build two different codes on the target and compare their power traces to make sure that we are actually measuring activities on the target board.

- Open the file `hardware/victims/firmware/simpleserial-base/simpleserial-base.c` by Jupyter or any other text editor.
- Add the following code around line 40, between `trigger_high()` and `trigger_low()`

```
volatile long int A = 0x2BAA;
for (uint8_t i = 0; i < 10; i++ ){
        A = (A*A) % 65535;
    }
```

- Note that the variable A must be volatile to ensure that the compiler does not optimize the code and removes the actual computation. Save
- Build, upload and capture one trace.

You should notice a plot with 10 spikes representing the operation of the 10 loops in the code, followed by almost uniform signal representing the waiting for next command.

In [22]:
```
%%sh
cd ../../hardware/victims/firmware/simpleserial-base/
make PLATFORM=CWNANO CRYPTO_TARGET=NONE
```

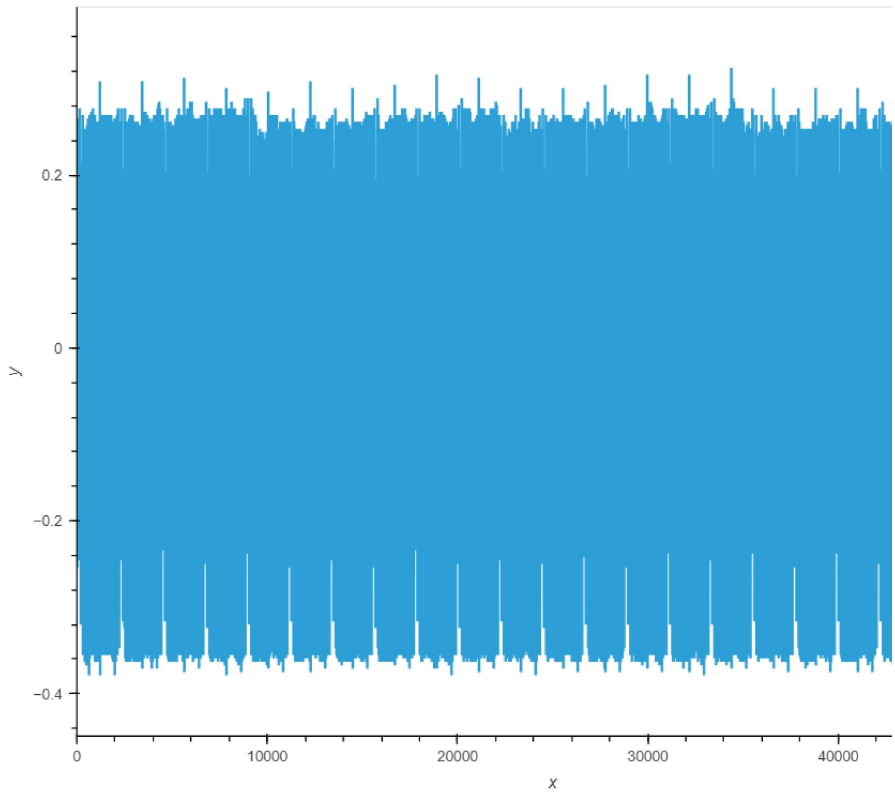
```
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
make[1]: '.dep' is up to date.
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
.
Welcome to another exciting ChipWhisperer target build!!
arm-none-eabi-gcc (GNU Arm Embedded Toolchain 10-2020-q4-major) 10.2.1 20201103 (release)
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

.
Compiling:
    simpleserial-base.c ...Done!
.
Compiling:
    .././simpleserial/simpleserial.c ...Done!
.
Compiling:
    .././hal/stm32f0_nano/stm32f0_hal_nano.c ...Done!
.
Compiling:
    .././hal/stm32f0/stm32f0_hal_lowlevel.c ...Done!
.
Assembling: .././hal/stm32f0/stm32f0_startup.S
arm-none-eabi-gcc -c -mcpu=cortex-m0 -I. -x assembler-with-cpp -mthumb -mfloat-abi=soft -ffunction-sections -DF_CPU=7372800 -Wa,-gsta
bs,-adhlns=objdir-CWNANO/stm32f0_startup.lst -I.././simpleserial/ -I.././simpleserial/ -I.././hal -I.././hal/stm32f0 -I.././hal/stm32
f0/CMSIS -I.././hal/stm32f0/CMSIS/core -I.././hal/stm32f0/CMSIS/device -I.././hal/stm32f0/Legacy -I.././crypto/ .././hal/stm32f0/stm3
2f0_startup.S -o objdir-CWNANO/stm32f0_startup.o
.
LINKING:
    simpleserial-base-CWNANO.elf ...Done!
.
Creating load file for Flash: simpleserial-base-CWNANO.hex
arm-none-eabi-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.hex
.
Creating load file for Flash: simpleserial-base-CWNANO.bin
arm-none-eabi-objcopy -O binary -R .eeprom -R .fuse -R .lock -R .signature simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.bin
.
Creating load file for EEPROM: simpleserial-base-CWNANO.eep
arm-none-eabi-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" \
--change-section-lma .eeprom=0 --no-change-warnings -O ihex simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.eep || exit 0
.
Creating Extended Listing: simpleserial-base-CWNANO.lss
arm-none-eabi-objdump -h -S -z simpleserial-base-CWNANO.elf > simpleserial-base-CWNANO.lss
.
Creating Symbol Table: simpleserial-base-CWNANO.sym
arm-none-eabi-nm -n simpleserial-base-CWNANO.elf > simpleserial-base-CWNANO.sym
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
Size after:
   text    data     bss     dec     hex filename
   4644      12    1428    6084    17c4 simpleserial-base-CWNANO.elf
+--------------------------------------------------------
+ Default target does full rebuild each time.
+ Specify buildtarget == allquick == to avoid full rebuild
+--------------------------------------------------------
+--------------------------------------------------------
+ Built for platform CWNANO Built-in Target (STM32F030) with:
+ CRYPTO_TARGET = NONE
+ CRYPTO_OPTIONS = AES128C
+--------------------------------------------------------
```

In [23]:

```
cw.program_target(scope, cw.programmers.STM32FProgrammer, "../../hardware/victims/firmware/simpleserial-base/simpleserial-base-CWNANO
trace = cw.capture_trace(scope, target, bytearray([0]*16))
cw.plot(trace.wave)
```

```
Detected known STMF32: STM32F04xxx
Extended erase (0x44), this can take ten seconds or more
Attempting to program 4655 bytes at 0x8000000
STM32F Programming flash...
STM32F Reading flash...
Verified flash OK, 4655 bytes
```

Out[23]:



- reopen the file `hardware/victims/firmware/simpleserial-base/simpleserial-base.c` by Jupyter or any other text editor.
- Modify the previous code to go for 20 rounds

```
volatile long int A = 0x2BAA;
for (uint8_t i = 0; i < 20; i++ ){
        A = (A*A) % 65535;
    }
```

- Save
- Build, upload and capture one trace.

Compare the two figures

In [24]:
```
%%sh
cd ../../hardware/victims/firmware/simpleserial-base/
make PLATFORM=CWNANO CRYPTO_TARGET=NONE
```

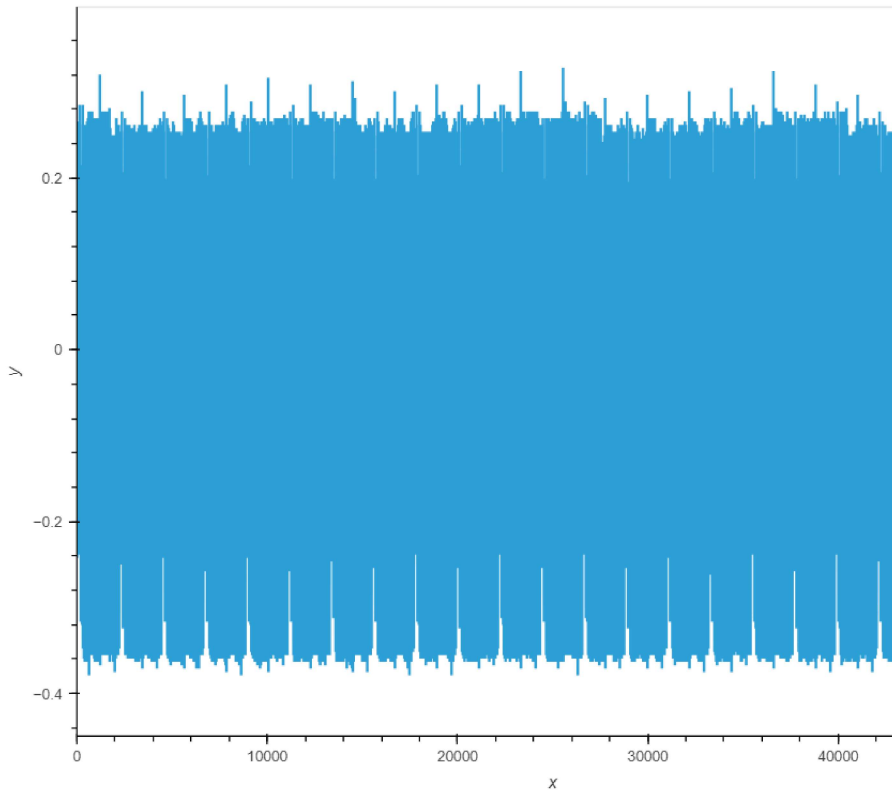
```
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
make[1]: '.dep' is up to date.
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
.
Welcome to another exciting ChipWhisperer target build!!
arm-none-eabi-gcc (GNU Arm Embedded Toolchain 10-2020-q4-major) 10.2.1 20201103 (release)
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

.
Compiling:
    simpleserial-base.c ...Done!
.
Compiling:
    .././simpleserial/simpleserial.c ...Done!
.
Compiling:
    .././hal/stm32f0_nano/stm32f0_hal_nano.c ...Done!
.
Compiling:
    .././hal/stm32f0/stm32f0_hal_lowlevel.c ...Done!
.
Assembling: .././hal/stm32f0/stm32f0_startup.S
arm-none-eabi-gcc -c -mcpu=cortex-m0 -I. -x assembler-with-cpp -mthumb -mfloat-abi=soft -ffunction-sections -DF_CPU=7372800 -Wa,-gsta
bs,-adhlns=objdir-CWNANO/stm32f0_startup.lst -I.././simpleserial/ -I.././simpleserial/ -I.././hal -I.././hal/stm32f0 -I.././hal/stm32
f0/CMSIS -I.././hal/stm32f0/CMSIS/core -I.././hal/stm32f0/CMSIS/device -I.././hal/stm32f0/Legacy -I.././crypto/ .././hal/stm32f0/stm3
2f0_startup.S -o objdir-CWNANO/stm32f0_startup.o
.
LINKING:
    simpleserial-base-CWNANO.elf ...Done!
.
Creating load file for Flash: simpleserial-base-CWNANO.hex
arm-none-eabi-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.hex
.
Creating load file for Flash: simpleserial-base-CWNANO.bin
arm-none-eabi-objcopy -O binary -R .eeprom -R .fuse -R .lock -R .signature simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.bin
.
Creating load file for EEPROM: simpleserial-base-CWNANO.eep
arm-none-eabi-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load" \
--change-section-lma .eeprom=0 --no-change-warnings -O ihex simpleserial-base-CWNANO.elf simpleserial-base-CWNANO.eep || exit 0
.
Creating Extended Listing: simpleserial-base-CWNANO.lss
arm-none-eabi-objdump -h -S -z simpleserial-base-CWNANO.elf > simpleserial-base-CWNANO.lss
.
Creating Symbol Table: simpleserial-base-CWNANO.sym
arm-none-eabi-nm -n simpleserial-base-CWNANO.elf > simpleserial-base-CWNANO.sym
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
Size after:
   text    data     bss     dec     hex filename
   4644      12    1428    6084    17c4 simpleserial-base-CWNANO.elf
+--------------------------------------------------------
+ Default target does full rebuild each time.
+ Specify buildtarget == allquick == to avoid full rebuild
+--------------------------------------------------------
+--------------------------------------------------------
+ Built for platform CWNANO Built-in Target (STM32F030) with:
+ CRYPTO_TARGET = NONE
+ CRYPTO_OPTIONS = AES128C
+--------------------------------------------------------
```

In [25]:

```
cw.program_target(scope, cw.programmers.STM32FProgrammer, "../../hardware/victims/firmware/simpleserial-base/simpleserial-base-CWNANO
trace = cw.capture_trace(scope, target, bytearray([0]*16))
cw.plot(trace.wave)
```

```
Detected known STMF32: STM32F04xxx
Extended erase (0x44), this can take ten seconds or more
Attempting to program 4655 bytes at 0x8000000
STM32F Programming flash...
STM32F Reading flash...
Verified flash OK, 4655 bytes
```

Out[25]:



Conclusion

And that's it! You should be all ready to continue on to the next task of the Assignment.

As a final step, we should disconnect from the hardware so it doesn't stay "in use" by this notebook.

In [26]:
```
scope.dis()
target.dis()
```

In [ ]: