Cynthion

Great Scott Gadgets

May 22, 2026

USER DOCUMENTATION

1 Cynthion Project Description **3**

2 Getting Started with Cynthion **5**
2.1 Prerequisites . . . 5
2.2 Cynthion Host Software Installation . . . 5
2.3 Test Installation . . . 6
2.4 Updating Cynthion Host Software . . . 7
2.5 Updating Cynthion Microcontroller Firmware and FPGA configuration flash . . . 7

3 Using Cynthion with Packetry **9**
3.1 Prerequisites . . . 9
3.2 USB Analyzer Bitstream . . . 9
3.3 Connect Hardware . . . 10

4 Using Cynthion with Facedancer **11**
4.1 Install the Facedancer library . . . 11
4.2 Load Facedancer Bitstream and Firmware . . . 11
4.3 Connect Hardware . . . 12
4.4 Run a Facedancer example . . . 12
4.5 More Information . . . 13

5 Using Cynthion with USB Proxy **15**
5.1 Connect Hardware . . . 15
5.2 Run a USB Proxy example . . . 15
5.3 More Information . . . 16

6 Using Cynthion USER I/O with Facedancer **17**
6.1 Requirements . . . 17
6.2 Using Cynthion APIs . . . 17
6.3 Using USER Button and Leds with Facedancer . . . 18
6.4 USER Pmod inputs and outputs . . . 20

7 The `cynthion` command line interface **25**
7.1 Command Documentation . . . 25

8 Protocol analysis of a USB keyboard **29**
8.1 Prerequisites . . . 29
8.2 Determine device speed . . . 29
8.3 Connect . . . 30
8.4 Capture . . . 30
8.5 Troubleshooting . . . 33

9 Emulation of a USB Device **35**
9.1 Prerequisites . . . 35
9.2 Try to Detect a HackRF One . . . 35
9.3 Connect . . . 36
9.4 Emulate the Vendor ID and Product ID . . . 36
9.5 Try the Suggested Code . . . 37
9.6 Handle the Version String Request . . . 39
9.7 Handle the Part ID Request . . . 40
9.8 Handle the Close Request . . . 41
9.9 Put It All Together . . . 41

10 Gateware Blinky **43**
10.1 Prerequisites . . . 43
10.2 Create a new Amaranth module . . . 43
10.3 Obtain a platform resource . . . 44
10.4 Timer State . . . 44
10.5 Timer Logic . . . 45
10.6 Put It All Together . . . 45
10.7 Build and Upload FPGA Bitstream . . . 46
10.8 Exercises . . . 46
10.9 More information: . . . 46

11 USB Gateware: Part 1 - Enumeration **47**
11.1 Prerequisites . . . 47
11.2 Define a USB Device . . . 47
11.3 Testing the Device . . . 50
11.4 Conclusion . . . 53
11.5 Exercises . . . 54
11.6 More information . . . 54
11.7 Source Code . . . 54

12 USB Gateware: Part 2 - WCID Descriptors **57**
12.1 Prerequisites . . . 57
12.2 WCID Devices . . . 57
12.3 Testing the Device . . . 62
12.4 Conclusion . . . 64
12.5 Exercises . . . 64
12.6 More information . . . 64
12.7 Source Code . . . 64

13 USB Gateware: Part 3 - Control Transfers **69**
13.1 Prerequisites . . . 69
13.2 Data Transfer between a Host and Device . . . 69
13.3 Test Control Endpoints . . . 73
13.4 Exercises . . . 76
13.5 More information . . . 76
13.6 Source Code . . . 76

14 USB Gateware: Part 4 - Bulk Transfers **83**
14.1 Prerequisites . . . 83
14.2 Add Bulk Endpoints . . . 83
14.3 Test Bulk Endpoints . . . 88
14.4 Exercises . . . 91
14.5 More information . . . 91
14.6 Source Code . . . 91

15 Introduction **99**
15.1 Cynthion Hardware . . . 100

16 Device Overview **101**
16.1 Top View . . . 101
16.2 Left View . . . 101
16.3 Right View . . . 101
16.4 Front View . . . 102
16.5 Bottom View . . . 102

17 Self-made Hardware Bringup **103**
17.1 Prerequisites . . . 103
17.2 Bring-up Process . . . 103
17.3 Build/upload Saturn-V . . . 103
17.4 Build/upload Apollo . . . 105
17.5 Running Self-Tests . . . 106
17.6 Troubleshooting . . . 106

18 The `apollo` command line utility **107**

19 Getting Help **109**

20 Cynthion Projects and Mentions **111**

21 Safety Information **113**
21.1 Warnings . . . 113
21.2 Instructions For Safe Use . . . 113

22 Introduction **115**
22.1 Setting up a Development Environment . . . 115
22.2 Prerequisites . . . 115
22.3 Installation . . . 115

23 Bitstream Generation **117**
23.1 Cynthion Gateware . . . 117

24 Facedancer SoC Firmware Compilation **119**
24.1 Prerequisites . . . 119
24.2 Install Rust Dependencies . . . 119
24.3 Building and Running . . . 119
24.4 Running Firmware Unit Tests . . . 120
24.5 Firmware Examples . . . 120

Cynthion™
Powered by LUNA
A B C D E
5 4 3 2 1 0
GREAT SCOTT GADGETS™
RESET
TARGET A
TARGET C
A
V G 4 3 2 1
V G 10 9 8 7
V G 4 3 2 1
V G 10 9 8 7
B

CHAPTER

ONE

CYNTHION PROJECT DESCRIPTION

Cynthion is an all-in-one tool for building, testing, monitoring, and experimenting with USB devices. Built around a unique FPGA-based architecture, Cynthion's digital hardware can be fully customized to suit the application at hand. As a result, it can act as a no-compromise High-Speed USB protocol analyzer, a USB-hacking multi-tool, or a USB development platform.

Out-of-the-box, Cynthion acts as a USB protocol analyzer capable of capturing and analyzing traffic between a host and any Low-, Full-, or High-Speed ("USB 2.0") USB device. It works seamlessly with our Packetry open-source analysis software.

Combined with our LUNA gateware and Facedancer libraries, Cynthion becomes a versatile USB-hacking and development tool. Facedancer makes it quick and easy to create or tamper with real USB devices—not just emulations—even if you don't have experience with digital hardware design, HDL, or FPGA architecture!

CHAPTER

TWO

GETTING STARTED WITH CYNTHION

2.1 Prerequisites

To use Cynthion you will need to ensure the following software is installed:

- Python v3.9, or later.

2.2 Cynthion Host Software Installation

The Cynthion host software distribution can be obtained from the Python Package Index (PyPI) or *directly from source*.

> **Note:** For more information on installing Python packages from PyPI please refer to the "Installing Packages" section of the Python Packaging User Guide.

Use pip to install the Cynthion host software:

```
pip install cynthion
```

Install udev Rules

Configure your system to allow access to Cynthion for logged in users:

```
cynthion setup
```

If you'd prefer to perform this step manually, you can download and install the rules as follows:

```
# download udev rules
curl -O https://raw.githubusercontent.com/greatscottgadgets/cynthion/main/
↪cynthion/python/assets/54-cynthion.rules

# install udev rules
sudo cp 54-cynthion.rules /etc/udev/rules.d

# reload udev rules
sudo udevadm control --reload

# apply udev rules to any devices that are already plugged in
sudo udevadm trigger
```

You can check that the rules are installed correctly with:

```
cynthion setup --check
```

Use Homebrew to install Python and libusb:

```
brew install python libusb
```

Use pip to install the Cynthion host software:

```
pip install cynthion
```

Note: The Cynthion host software uses the `libusb1` Python package to communicate with the hardware. On macOS, the package does not install the native dynamic library with it, so it's necessary to install the `libusb` native library through Homebrew, MacPorts or some other route.

If you are not using a Python distribution from Homebrew you may be able to direct Cynthion to the correct location by explicitly setting `DYLD_FALLBACK_LIBRARY_PATH` to the location of the `libusb` native library.

For example:

```
DYLD_FALLBACK_LIBRARY_PATH="/opt/homebrew/lib" cynthion info
```

Use pip to install the Cynthion host software:

```
pip install cynthion
```

2.3 Test Installation

2.3.1 Connect Hardware

- Connect the Host computer to the Cynthion **CONTROL** port.
- Check that the *LED A power-on indicator* lights up.

2.3.2 Test Hardware Connectivity

Open a terminal and confirm that everything is working by running:

```
cynthion info --force-offline
```

If everything is working you will see the following output:

```
Found Cynthion device!
    Hardware: Cynthion r1.4
    Manufacturer: Great Scott Gadgets
    Product: Cynthion Apollo Debugger
    Serial number: xxxxxxxxxxxxxxxxxxxxxxxx
    Vendor ID: 1d50
    Product ID: 615c
    bcdDevice: 0104
```

(continues on next page)

(continued from previous page)

```
    Firmware version: v1.0.6
    USB API version: 1.1
    Flash UID: xxxxxxxxxxxxxxxx
```

2.4 Updating Cynthion Host Software

To update the Cynthion host software to the latest version run:

```
pip install --upgrade cynthion
```

2.5 Updating Cynthion Microcontroller Firmware and FPGA configuration flash

To upgrade the Cynthion Microcontroller firmware and FPGA configuration flash to the latest versions run:

```
cynthion update
```

You can update the Microcontroller firmware separately with:

```
cynthion update --mcu-firmware
```

You can update the FPGA configuration flash separately with:

```
cynthion update --bitstream
```

CHAPTER

THREE

USING CYNTHION WITH PACKETRY

Together with Packetry, Cynthion can be used as a USB 2.0 protocol analyzer capable of capturing and analyzing traffic between a host and any Low, Full, and High Speed USB device.

Before proceeding, please ensure you have completed all steps in the *Getting Started with Cynthion* section.

3.1 Prerequisites

To use Cynthion's USB Analyzer you will need to ensure the following software is installed:

- Packetry

3.2 USB Analyzer Bitstream

Cynthion ships from the factory with the USB Analyzer as the default bitstream for the FPGA.

If you have previously flashed a different default bitstream you can run the USB Analyzer bitstream with:

```
cynthion run analyzer
```

If you want to configure USB Analyzer as the default bitstream for the FPGA:

```
cynthion flash analyzer
```

You can verify that everything is working by running:

```
cynthion info
```

You should see output like:

```
Detected a Cynthion device!
    Bitstream: USB Analyzer (Cynthion Project)
    Hardware: Cynthion r1.4
    Flash UID: xxxxxxxxxxxxxxxx
```

3.3 Connect Hardware

Next, see the Packetry documentation for more detail, or the tutorial *Protocol analysis of a USB keyboard* for a worked example.

CHAPTER

FOUR

USING CYNTHION WITH FACEDANCER

Together with Facedancer, Cynthion can be used to quickly and easily emulate USB devices controlled from Python running on the host computer.

Before proceeding, please ensure you have completed all steps in the *Getting Started with Cynthion* section.

4.1 Install the Facedancer library

You can install the Facedancer library from the Python Package Index (PyPI), a release archive or directly from source.

4.1.1 Install From PyPI

You can use the pip tool to install the Facedancer library from PyPI using the following command:

```
pip install facedancer
```

For more information on installing Python packages from PyPI please refer to the "Installing Packages" section of the Python Packaging User Guide.

4.1.2 Install From Source

```
git clone https://github.com/greatscottgadgets/facedancer.git
cd facedancer/
```

Once you have the source code downloaded you can install the Facedancer library with:

```
pip install .
```

4.2 Load Facedancer Bitstream and Firmware

You can run the Facedancer Bitstream and Firmware by running:

```
cynthion run facedancer
```

You can verify that everything is working by running:

```
cynthion info
```

You should see output like:

```
Detected a Cynthion device!
    Bitstream: Facedancer (Cynthion Project)
    Hardware: Cynthion r1.4
    Flash UID: xxxxxxxxxxxxxxxx
```

4.3 Connect Hardware

Make sure that the target host is running a program that can receive keyboard input such as a terminal or text editor and that it has focus.

4.4 Run a Facedancer example

Create a new Python file called `rubber-ducky.py` with the following content:

```python
import asyncio

from facedancer                    import main
from facedancer.devices.keyboard   import USBKeyboardDevice

device = USBKeyboardDevice()

async def type_letters():
    # Wait for device to connect
    await asyncio.sleep(2)

    # Type a string with the device
    await device.type_string("echo hello, facedancer\n")

main(device, type_letters())
```

Open a terminal and run:

```
python ./rubber-ducky.py
```

If all goes well, you should see the string `hello, facedancer` typed into the target host.

4.5 More Information

For further information, see the Facedancer documentation.

CHAPTER

FIVE

USING CYNTHION WITH USB PROXY

Together with USB Proxy, Cynthion can proxy packets between a target host and a target device attached to the control computer.

Before proceeding, please ensure you have completed all steps in the *Getting Started with Cynthion* and *Using Cynthion with Facedancer* sections.

5.1 Connect Hardware

5.2 Run a USB Proxy example

Create a new Python file called `usbproxy.py` with the following content:

```
#!/usr/bin/env python3
#
# This file is part of Facedancer.
#
""" USB Proxy example; forwards all USB transactions and logs them to the console. """

from facedancer          import main

from facedancer.proxy    import USBProxyDevice
from facedancer.filters  import USBProxySetupFilters, USBProxyPrettyPrintFilter

# replace with the proxied device's information
ID_VENDOR  = 0x1050
ID_PRODUCT = 0x0407


if __name__ == "__main__":
    # create a USB Proxy Device
    proxy = USBProxyDevice(idVendor=ID_VENDOR, idProduct=ID_PRODUCT)

    # add a filter to forward control transfers between the target host and
    # proxied device
    proxy.add_filter(USBProxySetupFilters(proxy, verbose=0))

    # add a filter to log USB transactions to the console
```

(continues on next page)

(continued from previous page)

```
    proxy.add_filter(USBProxyPrettyPrintFilter(verbose=5))

    main(proxy)
```

Open a terminal and run:

```
python ./usbproxy.py
```

Note: USBProxy requires root privileges on macOS in order to claim the device being proxied from the operating system.

```
sudo python ./usbproxy.py
```

```
python ./usbproxy.py
```

If all goes well you should see the output from device enumeration in your terminal and the proxied USB device should be detected by the target computer.

5.3 More Information

For further information, see the Facedancer USB Proxy documentation.

CHAPTER

SIX

USING CYNTHION USER I/O WITH FACEDANCER

In addition to the four USB ports Cynthion also includes the following user i/o ports:

- *USER Button*
- *USER PMOD A & B*
- *USER LEDs*

Apart from PMOD B, which is used for the Facedancer SoC UART and JTAG interface, all of these can be accessed from within Python Facedancer devices.

Before proceeding, please ensure you have completed all steps in the *Getting Started with Cynthion* and *Using Cynthion with Facedancer* sections.

6.1 Requirements

- A Cynthion running the Facedancer bitstream.
- Two USB Cables.

6.2 Using Cynthion APIs

To access Cynthion USER i/o using Python you first need an instance of the `Cynthion` object, which can be created as follows:

```
from cynthion import Cynthion
c = Cynthion()
```

Once you have a `Cynthion` instance you will be able to access USER i/o using the `leds` and `gpio` APIs.

For example, open a new Python shell:

```
$ python

Python 3.11.11 (main, Feb 12 2025, 14:40:14) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

First obtain a `Cynthion` instance:

```
>>> from cynthion import Cynthion
>>> c = Cynthion()
```

Turn on USER Led5:

```
>>> c.leds[5].on()
```

Turn off USER Led5:

```
>>> c.leds[5].off()
```

Toggle USER Led4:

```
>>> c.leds[4].toggle()
```

Get the USER Button:

```
>>> user_button = c.gpio.get_pin("USER")
```

Wait for the USER Button to be pressed: (hit enter twice to start)

```
>>> while user_button.read() == False: pass
...
```

6.3 Using USER Button and Leds with Facedancer

Lets modify the Facedancer rubber-ducky example to give us a bit more information and control using Cynthion USER i/o. We'll subclass a Facedancer device and add some calls to the USER i/o APIs in response to host requests and device responses.

Create a new Python file called `facedancer-user-io.py` and add the following content:

Listing 1: facedancer-user-io.py

```
import asyncio
import logging

from facedancer                   import main, errors
from facedancer.devices.keyboard  import USBKeyboardDevice

from cynthion                     import Cynthion

# Subclass USBKeyboardDevice
class MyKeyboardDevice(USBKeyboardDevice):
    def __post_init__(self):
        super().__post_init__()

        # Get a Cynthion instance.
        cynthion = Cynthion()

        # Get USER Leds
        self.leds = cynthion.leds
```

(continues on next page)

(continued from previous page)

```

        # Make sure all USER Leds are off
        [led.off() for led in self.leds.values()]

        # Get USER Button
        self.user_button = cynthion.gpio.get_pin("USER")

    def handle_bus_reset(self):
        # Strobe USER Led0 every time we see a bus reset
        self.leds[0].strobe(duration=0.1)
        super().handle_bus_reset()

    def handle_request(self, request):
        # Strobe USER Led1 every time the host makes a control request
        self.leds[1].strobe(duration=0.1)
        super().handle_request(request)

    def control_send(self, endpoint_number, in_request, data, *, blocking = False):
        # Strobe USER Led2 every time the device responds to a control request
        self.leds[2].strobe(duration=0.1)
        super().control_send(endpoint_number, in_request, data, blocking=blocking)

    def handle_data_requested(self, endpoint):
        report = self._generate_hid_report()
        endpoint.send(report)

        # Strobe USER Led3 every time the host requested a HID report descriptor from␣
↪the host
        if report[2] == 0:
            self.leds[3].strobe(duration=0.1)
        # Strobe USER Led4 if the report descriptor contained a scancode for the host
        else:
            self.leds[4].strobe(duration=0.1)

# Rubber-ducky control script
async def type_letters():
    # Wait for device to connect
    await asyncio.sleep(2)

    logging.info("Press the USER button to proceed.")

    # Wait until Cynthion's USER button is pressed
    while device.user_button.read() == False:
        await asyncio.sleep(0.01)

    logging.info("Typing string into target device.")

    # Type a string with the device
    await device.type_string("echo hello, facedancer\n")

    logging.info("Finished. Press the USER button again to quit.")

```

(continues on next page)

(continued from previous page)

```
    # Done
    while device.user_button.read() == False:
        await asyncio.sleep(0.01)

    raise errors.EndEmulation("User quit the emulation.")

# Start emulation
device = MyKeyboardDevice()
main(device, type_letters())
```

Open a terminal and run:

```
python ./facedancer-user-io.py
```

If everything went well you should see prompts at various points to press the USER button to continue execution as well as the USER Leds flashiing in response to device events.

6.4 USER Pmod inputs and outputs

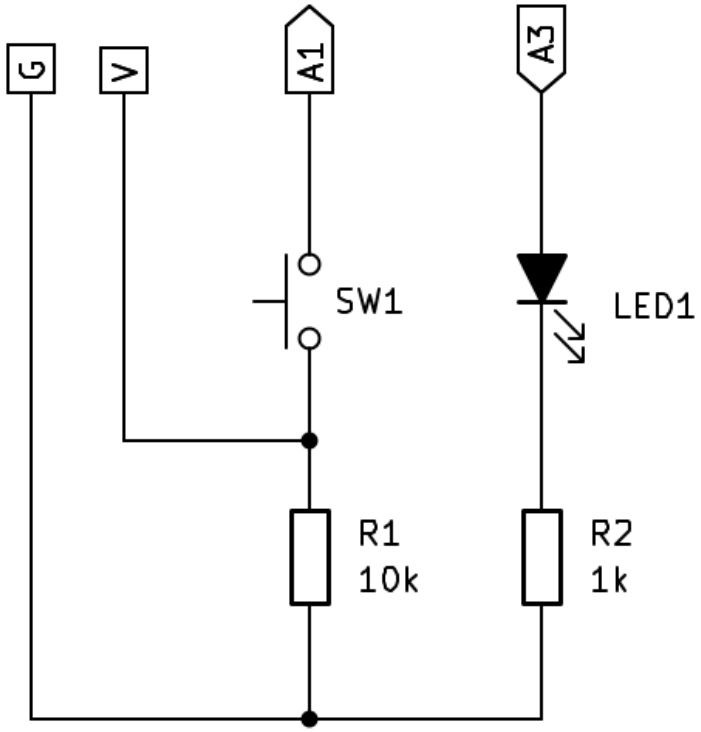
In addition to the USER Button and Leds, Facedancer can also make use of Cynthion USER Pmod A (USER Pmod B is used for JTAG and UART duties) to trigger or respond to external hardware. They use the same `gpio` APIs as the USER Button but individual pins can also be configured as inputs or outputs.

Let's build a simple example that uses two of the USER Pmod A pins to connect a switch and a LED to Cynthion.

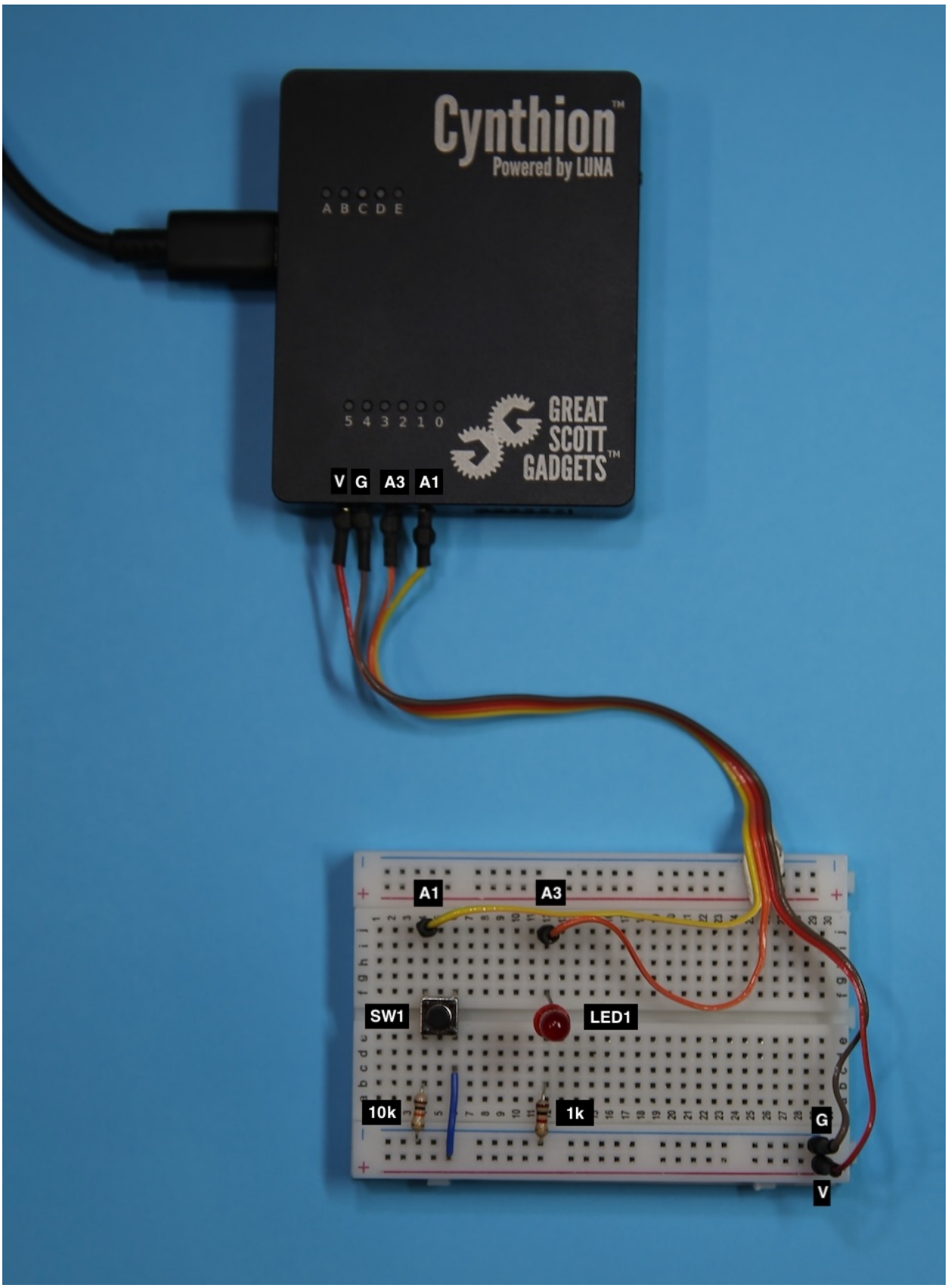
6.4.1 You will need:

- 1x SPST Switch
- 1x 1 kOhm resistor
- 1x 10 kOhm resistor
- 1x LED
- 1x Breadboard

6.4.2 Circuit Diagram



6.4.3 Breadboard Layout



6.4.4 Source Code

Create a new Python file called `cynthion-user-pmod.py` with the following content:

Listing 2: cynthion-user-pmod.py

```python
import time

from cynthion import            Cynthion
from cynthion.interfaces.gpio  import PinDirection

# Get Cynthion instance
c = Cynthion()

# Get USER Pmod Pin A1 and configure it as an input
a1 = c.gpio.get_pin("A1")
a1.set_direction(PinDirection.Input)

# Get USER Pmod Pin A3 and configure it as an output
a3 = c.gpio.get_pin("A3")
a3.set_direction(PinDirection.Output)

# Continuously read the input value of Pin A1 and output it to Pin A3.
while True:
    value = a1.read()
    a3.write(value)
    time.sleep(0.1)
```

Open a terminal and run:

```
python ./cynthion-user-pmod.py
```

If all goes well, the LED should light up when you press the switch and turn off when you release it.

CHAPTER

SEVEN

THE CYNTHION COMMAND LINE INTERFACE

```
$ apollo
usage: cynthion [-h] command ...

Cynthion command line interface

positional arguments:
  command
    run       run a bitstream on the FPGA
    flash     overwrite the FPGA's configuration flash with the target bitstream
    update    update MCU firmware and FPGA configuration flash to the latest
              installed versions
    info      print device information
    setup     install Cynthion support files required for operation (Linux only)

optional arguments:
  -h, --help  show this help message and exit
```

7.1 Command Documentation

7.1.1 Display Cynthion Information

Display Cynthion bitstream information:

```
cynthion info
```

Display Cynthion Microcontroller information:

```
cynthion info --force-offline
```

Note: Once you have switched to the Cynthion Microcontroller by pressing the PROGRAM button or the `--force-offline` option you will need to press the RESET button to return control to the FPGA.

7.1.2 Set up Cynthion

Check that your host environment is set up for Cynthion:

```
cynthion setup --check
```

Set up your host environment for Cynthion:

```
cynthion setup
```

Remove all files installed during set up:

```
cynthion setup --uninstall
```

7.1.3 Update Cynthion

Update both the Cynthion Debug Microcontroller firmware and USB Analyzer bitstream to the latest installed factory versions:

```
cynthion update
```

Update Cynthion Debug Microcontroller firmware to the latest installed factory version:

```
cynthion update --mcu-firmware
```

Update Cynthion USB Analyzer bitstream to the latest installed factory version:

```
cynthion update --bitstream
```

7.1.4 Run bitstream

Runs the given factory bitstream on the FPGA:

```
cynthion run <analyzer|facedancer|selftest>
```

Runs the bitstream specified by `<filename>` on the FPGA.

```
cynthion run --bitstream <filename>
```

7.1.5 Flash firmware and bitstreams

Overwrite the FPGA's default bitstream with the given factory bitstream:

```
cynthion flash <analyzer|facedancer>
```

Overwrite the FPGA's default bitstream with the one specified by `<filename>`:

```
cynthion flash --bitstream <filename>
```

Overwrite the Microcontroller firmware with the one specified by `<filename>`:

```
cynthion flash --mcu-firmware <filename>
```

Overwrite the SoC firmware with the one specified by `<filename>`:

```
cynthion flash --soc-firmware <filename>
```

CHAPTER

EIGHT

PROTOCOL ANALYSIS OF A USB KEYBOARD

This tutorial walks through the whole process of running a USB protocol capture of a target device, in this case a keyboard. Hopefully most people have a USB keyboard available and can follow along, though this process is also applicable to any USB peripheral device.

8.1 Prerequisites

- Install the Cynthion tools by following *Getting Started with Cynthion*
- Install Packetry by following Getting Started with Packetry
- Run the analyzer gateware on Cynthion by following *Using Cynthion with Packetry*

8.2 Determine device speed

USB 2.0 supports three different speeds: Low (1.5 Mbit/s), Full (12 Mbit/s), and High (480 Mbit/s). The analyzer needs to know what speed to expect, so we need to detemine what speed the target device is using.

> **Note:** Soon this step won't be necessary, as the analyzer will be able to determine the speed automatically, but that feature is currently in development.

To determine the speed, we plug the target device into a host check what speed it reports. The way to check depends on the host operating system:

Run the command `sudo dmesg -W` in a terminal window and then plug the target device in, some new lines should show up with information about it:

```
[975321.743878] usb 1-6.1: new full-speed USB device number 12 using xhci_hcd
[975321.821329] usb 1-6.1: New USB device found, idVendor=3434, idProduct=0108,␣
↪bcdDevice= 2.00
[975321.821341] usb 1-6.1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[975321.821345] usb 1-6.1: Product: Keychron Q1
[975321.821348] usb 1-6.1: Manufacturer: Keychron
```

Here, it shows that the target device has enumerated at full-speed.

Go to the Apple menu -> About This Mac -> More Info -> System Report -> Hardware -> USB. Highlight the target device and check the Speed field.

The easiest way to look at USB device information on Windows is to install and run USBView, then find the target device and check its *Device Bus Speed* field.

8.3 Connect

Next, we'll connect everything up for capture. Within the Cynthion hardware the **TARGET C** and **TARGET A** ports are connected together, and the analyzer gateware will capture any packets that are seen going between those ports. These packets are then sent out through the **CONTROL** port.

First, connect the **CONTROL** port to the host that will be running Packetry. Next, connect the **TARGET C** port to a host. This can be a different host from the one running Packetry, or it can be the same host. If it is the same host, the two connections must not be on the same hub.

The **TARGET A** port will connect to the target device, but for now we leave it disconnected.

8.4 Capture

Open Packetry. At the top of the window, you should see the action bar:

Select the correct speed, press the capture button (the filled circle), and plug in the target device. The cable connections should look like this:

Upon plugging in the target device, a collection of entries should show up in the Traffic Pane, these show the requests that a host makes to find out information about a new device (known as USB enumeration).

```
Device: (None)   Speed: (None)

Time          Traffic
0.000000000   4347 SOF groups
0.069725416   Getting device descriptor #0 for device 0, reading 18 of 64 requested bytes
0.069725416   SETUP transaction on 0.0 with 8 data bytes, ACK: [80, 06, 00, 01, 00, 00, 40, 00]
0.069725416   SETUP packet on 0.0, CRC 02
0.069728650   DATA0 packet with CRC 94DD and 8 data bytes: [80, 06, 00, 01, 00, 00, 40, 00]
0.069737216   ACK packet
0.069747533   IN transaction on 0.0 with 18 data bytes, ACK: [12, 01, 00, 02, 00, 00, 00, 40, 34, 34, 08, 01, 00, 02, 01, 02, 00,...
0.069774750   OUT transaction on 0.0 with no data, ACK
0.197708550   Setting address to 61 for device 0
0.217903250   Getting device descriptor #0 for device 61, reading 18 bytes
0.218241583   Getting device qualifier descriptor #0 for device 61, reading 0 of 10 requested bytes, stalled
0.218528250   Getting device qualifier descriptor #0 for device 61, reading 0 of 10 requested bytes, stalled
0.218849350   Getting device qualifier descriptor #0 for device 61, reading 0 of 10 requested bytes, stalled
0.219079833   Getting configuration descriptor #0 for device 61, reading 9 bytes
0.219160100   Getting configuration descriptor #0 for device 61, reading 91 bytes
0.219362150   Getting string descriptor #0 for device 61, reading 4 of 255 requested bytes
0.219487883   Getting string descriptor #2, language 0x0409 (English/US) for device 61, reading 24 of 255 requested bytes: 'K...
0.219603666   Getting string descriptor #1, language 0x0409 (English/US) for device 61, reading 18 of 255 requested bytes: 'K...
0.221810800   Setting configuration 1 for device 61
0.222505900   Class request #10, index 0, value 0 for interface 61.0
0.222593150   Getting unknown descriptor #0 for interface 61.0, reading 64 bytes
0.223175850   Class request #9, index 0, value 512 for interface 61.0, writing 1 bytes
0.224004100   Polling 4339 times for interrupt transfer on endpoint 61.1 IN
0.283413200   Class request #10, index 0, value 0 for interface 61.1

Devices
Device 61: Keychron Q1
  Device descriptor
    Length: 18 bytes
    Type: 0x01
    USB Version: 2.00
    Class: 0x00: (Defined at Interface level)
    Subclass: 0x00
    Protocol: 0x00
    Max EP0 packet size: 64 bytes
    Vendor ID: 0x3434
    Product ID: 0x0108
    Version: 2.00
    Manufacturer string: #1 'Keychron'
    Product string: #2 'Keychron Q1'
    Serial string: (none)
  Configuration 1
    Configuration descriptor
      Length: 9 bytes
      Type: 0x02
      Total length: 91 bytes
      Number of interfaces: 3
      Configuration number: 1
      Configuration string: (none)
      Attributes: 0xA0

Packet #72 with 11 bytes
Timestamp: 69,728,650 ns from capture start
DATA0 packet with CRC 94DD and 8 data bytes
Hex bytes: [C3, <payload>, DD, 94]
Payload: [80, 06, 00, 01, 00, 00, 40, 00]

demo-kb.pcap: 1 devices, 4 endpoints, 13,027 transactions, 21,919 packets
```

The target device should also show up in the *Devices* pane to the right. All of the entries in the Traffic and Device panes can be expanded for more detail, by clicking on the black triangles. Clicking on an entry in the Traffic pane to highlight it will show extra detail in the pane below.

If you press and release a key on the keyboard, some new entries should show up:

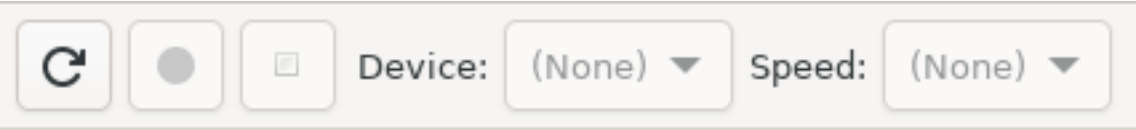
```
2.982783650   Interrupt transfer of 32 bytes on endpoint 61.4 IN: [06, 00, 10, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 0...
2.983791366   Polling 104 times for interrupt transfer on endpoint 61.4 IN
3.087775300   Interrupt transfer of 32 bytes on endpoint 61.4 IN: [06, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 0...
3.088783200   Polling 1474 times for interrupt transfer on endpoint 61.4 IN

Transaction group with 1 transaction
Timestamp: 2,982,783,650 ns from start of capture
First transaction #8290, first packet #14021
Interrupt transfer of 32 bytes on endpoint 61.4 IN
Payload: [06, 00, 10, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00]
```

Here I pressed the a key, causing two interrupt transfers; one for the press and one for the release event. If you want to go further, you could look at the USB human interface device class (HID) specification, to learn about the class-specific descriptors and endpoints, and match up what you see on the bus to keypress events.

8.5 Troubleshooting

Below are some common issues you may run into, with some advice for resolving them. If you run into further issues, please see the *Getting Help* section for more support.

8.5.1 Capture button is grayed out and no capture device shows up in Packetry

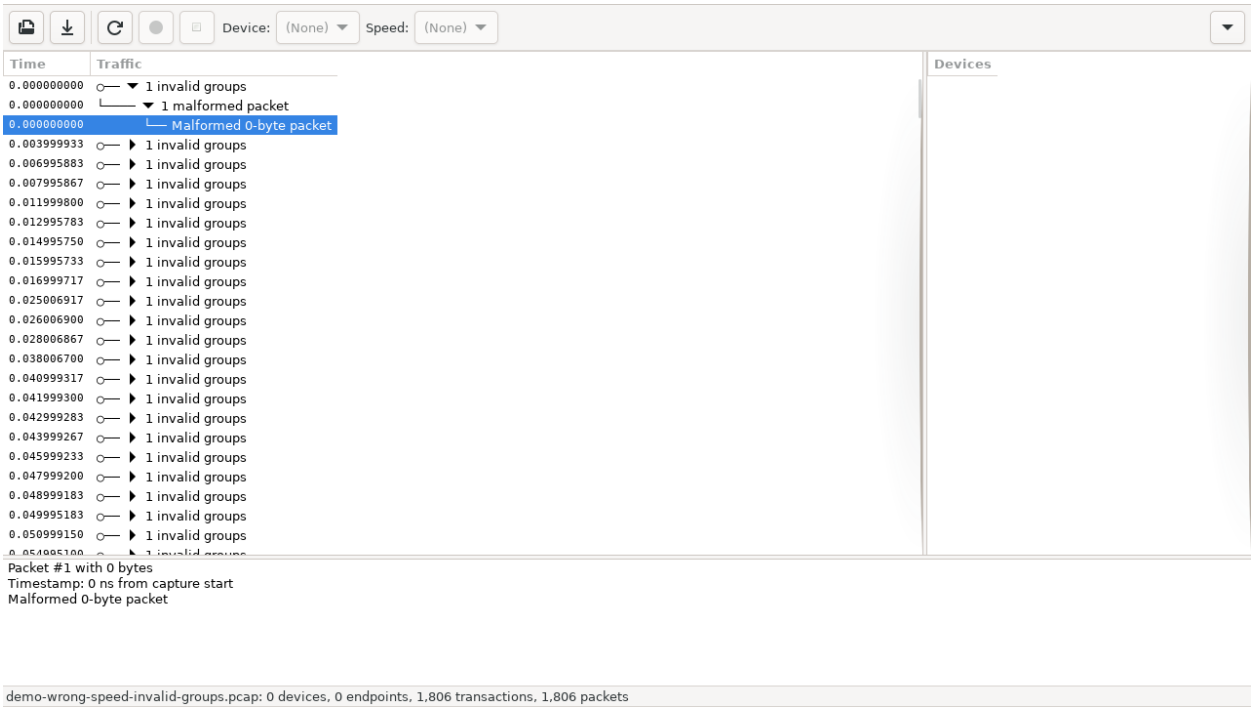


- Double check that the Cynthion **CONTROL** port is connected to the host running Packetry.
- Check that the cable is good, ideally trying it with another USB device that transfers data (not just charging).
- Make sure that the analyzer gateware is running on the Cynthion device by following *Using Cynthion with Packetry*.

8.5.2 No traffic shows up during capture

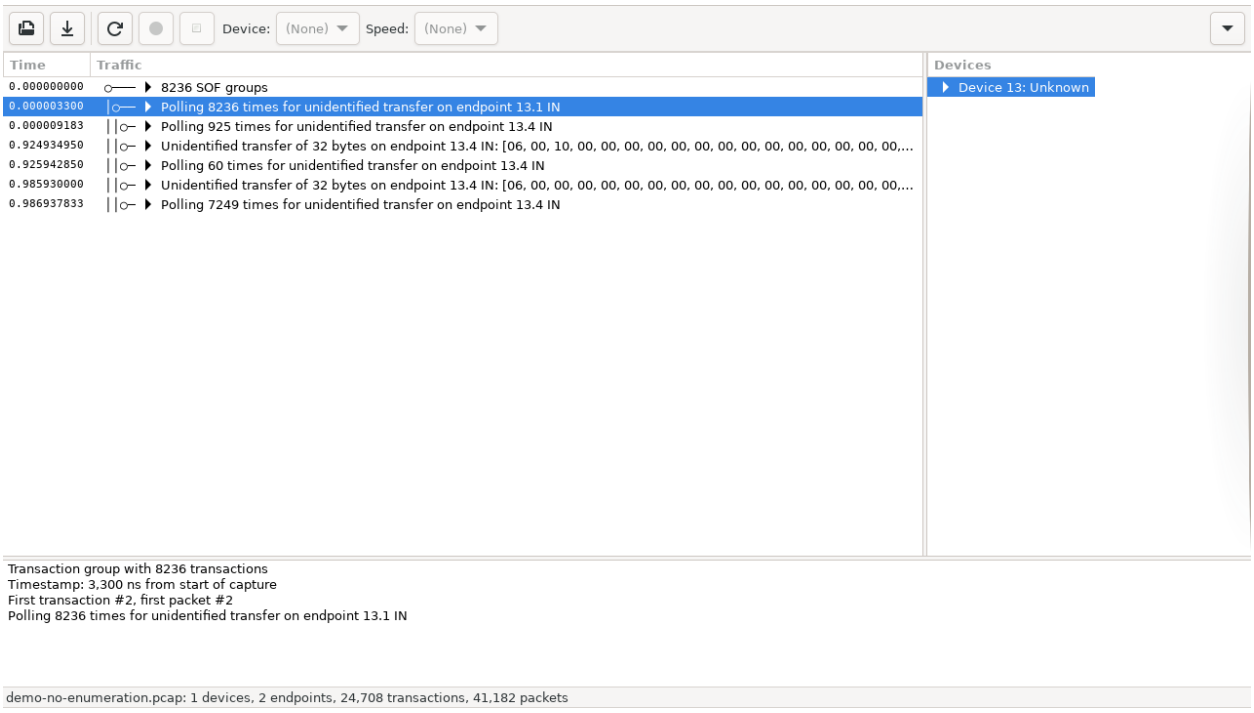
First, make sure the target device is operating correctly. If following along with a keyboard, make sure that any keypresses get through to the target host. If traffic still isn't showing up, this is usually caused by selecting the wrong capture speed, try capturing with each of the other two speed options.

8.5.3 Traffic shows "Invalid Groups"



This means that the analyzer is detecting packets that are invalid. This is usually caused by selecting the wrong capture speed, try capturing with each of the other two speed options.

8.5.4 Traffic shows “unidentified transfer” and Devices shows an “Unknown” device



This means that valid packets have been captured, but Packetry did not see the target device enumeration so it doesn’t have enough information to fully decode the USB transactions/transfers. If you would like to see the full decoding, make sure to start the capture in Packetry before plugging in the target device.

CHAPTER

NINE

EMULATION OF A USB DEVICE

This tutorial walks through the whole process of emulating a USB device with Cynthion and Facedancer. We'll emulate HackRF One, a software-defined radio platform. The goal of our emulation is to fool the `hackrf_info` command into reporting that a HackRF One is connected.

9.1 Prerequisites

- Install the Cynthion tools by following *Getting Started with Cynthion*.
- Install HackRF Tools by following Installing HackRF Software.
- Install the Facedancer library and run the Facedancer bitstream and firmware as described in *Using Cynthion with Facedancer*.

> **Note:** If you would like to configure your Cynthion for Facedancer operation permanently instead of temporarily, use `cynthion flash facedancer` instead of `cynthion run facedancer`.

9.2 Try to Detect a HackRF One

Use the `hackrf_info` command to detect any connected HackRF devices:

```
hackrf_info
```

The command output should indicate that no HackRF devices are found:

```
hackrf_info version: 2023.01.1
libhackrf version: 2023.01.1 (0.8)
No HackRF boards found.
```

9.3 Connect

We need to connect our Cynthion before we can use it to emulate a HackRF One. If you followed the prerequisites above, you should already have connected the Cynthion's **CONTROL** port to your computer.

Now also connect the **TARGET C** port to your computer. Facedancer software uses **CONTROL** to control the Cynthion and **TARGET C** to connect to the target host, the computer which we'll try to fool into thinking that there is a HackRF One connected. The control host and target host can be two separate computers, but in this tutorial we will use the same computer as both the control host and the target host.

9.4 Emulate the Vendor ID and Product ID

Use your favorite text editor to create a new Python program called `hackrf_emulation.py` with the following contents:

```
from facedancer import *
from facedancer import main

@use_inner_classes_automatically
class HackRF(USBDevice):
    product_string       : str = "HackRF One (Emulated)"
    manufacturer_string  : str = "Facedancer"
    serial_number_string : str = "1234"
    vendor_id            : int = 0x1d50
    product_id           : int = 0x6089

    class DefaultConfiguration(USBConfiguration):
        class DefaultInterface(USBInterface):
            pass

main(HackRF)
```

Every USB device identifies itself to its host computer using a 16-bit Vendor ID and 16-bit Product ID. This program uses the Facedancer library to implement a device with the Vendor ID and Product ID associated with HackRF One. It also configures some strings which make our emulated HackRF distinguishable from an actual HackRF One (with tools such as `lsusb`) for convenience.

Execute the program:

```
python hackrf_emulation.py --suggest
```

While the program is running, open another terminal and execute `hackrf_info`. It should display output similar to this:

```
hackrf_info version: 2023.01.1
libhackrf version: 2023.01.1 (0.8)
Found HackRF
Index: 0
Serial number: 1234
hackrf_board_id_read() failed: Pipe error (-1000)
```

We've just convinced `hackrf_info` that it has found a HackRF device! However, `hackrf_info` failed to read the HackRF's board ID which distinguishes between the various hardware platforms supported by HackRF software. The pipe error indicates that the device did not provide the expected response to the host's request for the board ID.

Terminate `hackrf_emulation.py` by typing `ctrl-c`. Because we used the `--suggest` option, it should provide output like this:

```
Automatic Suggestions
These suggestions are based on simple observed behavior;
not all of these suggestions may be useful / desirable.


Request handler code:

    @vendor_request_handler(number=14, direction=USBDirection.IN)
    @to_device
    def handle_control_request_14(self, request):
        # Most recent request was for 1B of data.
        # Replace me with your handler.
        request.stall()
```

9.5 Try the Suggested Code

Add the suggested code to the `HackRF` class in `hackrf_emulation.py`. The program should now look like:

```
from facedancer import *
from facedancer import main

@use_inner_classes_automatically
class HackRF(USBDevice):
    product_string       : str = "HackRF One (Emulated)"
    manufacturer_string  : str = "Facedancer"
    serial_number_string : str = "1234"
    vendor_id            : int = 0x1d50
    product_id           : int = 0x6089

    class DefaultConfiguration(USBConfiguration):
        class DefaultInterface(USBInterface):
            pass

    @vendor_request_handler(number=14, direction=USBDirection.IN)
    @to_device
    def handle_control_request_14(self, request):
        # Most recent request was for 1B of data.
        # Replace me with your handler.
        request.stall()

main(HackRF)
```

Execute the program:

```
python hackrf_emulation.py --suggest
```

While the program is running, execute `hackrf_info` in another terminal:

```
hackrf_info version: 2023.01.1
libhackrf version: 2023.01.1 (0.8)
Found HackRF
Index: 0
Serial number: 1234
hackrf_board_id_read() failed: Pipe error (-1000)
```

It turns out that our emulation still results in a pipe error. This is because we are stalling vendor request number 14 which is meant to return a 1 byte board ID. Terminate `hackrf_emulation.py` and replace the `request_stall()` line with:

```
request.reply([1])
```

Execute the program:

```
python hackrf_emulation.py --suggest
```

While the program is running, execute `hackrf_info` in another terminal:

```
hackrf_info version: 2023.01.1
libhackrf version: 2023.01.1 (0.8)
Found HackRF
Index: 0
Serial number: 1234
Board ID Number: 1 (Jawbreaker)
hackrf_version_string_read() failed: Pipe error (-1000)
```

We've now convinced `hackrf_info` that our Cynthion is a HackRF Jawbreaker which was the beta platform that preceded HackRF One. Let's try a higher board ID number. Replace `request.reply([1])` with:

```
request.reply([2])
```

Execute the program:

```
python hackrf_emulation.py --suggest
```

While the program is running, execute `hackrf_info` in another terminal:

```
hackrf_info version: 2023.01.1
libhackrf version: 2023.01.1 (0.8)
Found HackRF
Index: 0
Serial number: 1234
Board ID Number: 2 (HackRF One)
hackrf_version_string_read() failed: Pipe error (-1000)
```

We did it! Our new board ID represents HackRF One! In this example we guessed low numbers for the board ID byte, but we could have discovered that 2 represents HackRF One by observing the behavior of an actual HackRF One or by reading the libhackrf source code or HackRF firmware source code.

9.6 Handle the Version String Request

Unfortunately, `hackrf_info` still indicates an error, this time with reading a version string. The `--suggest` option on your Facedancer program should give you an idea of how to handle that request:

```
@vendor_request_handler(number=15, direction=USBDirection.IN)
@to_device
def handle_control_request_15(self, request):
    # Most recent request was for 255B of data.
    # Replace me with your handler.
    request.stall()
```

Notice that this time the host has requested `255` bytes instead of just one byte. USB devices often return a smaller number of bytes than the length requested by the host. In this case we can guess that the host is requesting a maximum length string and that we can probably return something shorter. Let's try adding this to the `HackRF` class in `hackrf_emulation.py`:

```
@vendor_request_handler(number=15, direction=USBDirection.IN)
@to_device
def handle_control_request_15(self, request):
    # Most recent request was for 255B of data.
    request.reply(b"tutorial version")
```

The complete program should now look like:

```
from facedancer import *
from facedancer import main

@use_inner_classes_automatically
class HackRF(USBDevice):
    product_string       : str = "HackRF One (Emulated)"
    manufacturer_string  : str = "Facedancer"
    serial_number_string : str = "1234"
    vendor_id            : int = 0x1d50
    product_id           : int = 0x6089

    class DefaultConfiguration(USBConfiguration):
        class DefaultInterface(USBInterface):
            pass

    @vendor_request_handler(number=14, direction=USBDirection.IN)
    @to_device
    def handle_control_request_14(self, request):
        # Most recent request was for 1B of data.
        # Replace me with your handler.
        request.reply([2])

    @vendor_request_handler(number=15, direction=USBDirection.IN)
    @to_device
    def handle_control_request_15(self, request):
        # Most recent request was for 255B of data.
        request.reply(b"tutorial version")
```

(continues on next page)

(continued from previous page)

```
main(HackRF)
```

Execute the program:

```
python hackrf_emulation.py --suggest
```

While the program is running, execute `hackrf_info` in another terminal:

```
hackrf_info version: 2023.01.1
libhackrf version: 2023.01.1 (0.8)
Found HackRF
Index: 0
Serial number: 1234
Board ID Number: 2 (HackRF One)
Firmware Version: tutorial version (API:0.00)
hackrf_board_partid_serialno_read() failed: Pipe error (-1000)
```

9.7 Handle the Part ID Request

Now we can see another unhandled request made by `hackrf_info`. The `--suggest` output tells us that we can handle it with something like:

```
@vendor_request_handler(number=18, direction=USBDirection.IN)
@to_device
def handle_control_request_18(self, request):
    # Most recent request was for 24B of data.
    # Replace me with your handler.
    request.stall()
```

The host is asking for 24 bytes this time, suggesting that it is looking for exactly 24 bytes. Let's try replying with 24 bytes of dummy data:

```
@vendor_request_handler(number=18, direction=USBDirection.IN)
@to_device
def handle_control_request_18(self, request):
    # Most recent request was for 24B of data.
    request.reply(b"A" * 24)
```

Execute the program:

```
python hackrf_emulation.py --suggest
```

While the program is running, execute `hackrf_info` in another terminal:

```
hackrf_info version: 2023.01.1
libhackrf version: 2023.01.1 (0.8)
Found HackRF
Index: 0
Serial number: 1234
Board ID Number: 2 (HackRF One)
Firmware Version: tutorial version (API:0.00)
```

(continues on next page)

(continued from previous page)

```
Part ID Number: 0x41414141 0x41414141
hackrf_close() failed: Pipe error (-1000)
```

It looks like the part ID was interpreted as a valid number, and now `hackrf_info` is trying to close the device! We're almost done!

9.8 Handle the Close Request

Based on the `--suggest` output, add the following to `hackrf_emulation.py`:

```
@vendor_request_handler(number=1, direction=USBDirection.OUT)
@to_device
def handle_control_request_1(self, request):
    request.ack()
```

Notice that this time the direction of the vendor request is `OUT` instead of `IN`. This means that the host is sending data to the device, not asking the device to send data to the host. We acknowledge the request instead of replying with data.

Execute the program:

```
python hackrf_emulation.py --suggest
```

While the program is running, execute `hackrf_info` in another terminal:

```
hackrf_info version: 2023.01.1
libhackrf version: 2023.01.1 (0.8)
Found HackRF
Index: 0
Serial number: 1234
Board ID Number: 2 (HackRF One)
Firmware Version: tutorial version (API:0.00)
Part ID Number: 0x41414141 0x41414141
```

Success! `hackrf_info` now exits without error!

9.9 Put It All Together

With a few edits based on what we've learned, our complete program might look like this:

```
from facedancer import *
from facedancer import main

@use_inner_classes_automatically
class HackRF(USBDevice):
    product_string       : str = "HackRF One (Emulated)"
    manufacturer_string  : str = "Facedancer"
    serial_number_string : str = "1234"
    vendor_id            : int = 0x1d50
    product_id           : int = 0x6089
```

(continues on next page)

(continued from previous page)

```
    class DefaultConfiguration(USBConfiguration):
        class DefaultInterface(USBInterface):
            pass

    @vendor_request_handler(number=14, direction=USBDirection.IN)
    @to_device
    def handle_board_id_request(self, request):
        # return 1-byte board ID
        request.reply([2])

    @vendor_request_handler(number=15, direction=USBDirection.IN)
    @to_device
    def handle_version_string_request(self, request):
        # return up to 255 bytes
        request.reply(b"tutorial version")

    @vendor_request_handler(number=18, direction=USBDirection.IN)
    @to_device
    def handle_part_id_request(self, request):
        # return 24 byte part ID
        request.reply(b"A" * 24)

    @vendor_request_handler(number=1, direction=USBDirection.OUT)
    @to_device
    def handle_close_request(self, request):
        request.reply([])

main(HackRF)
```

CHAPTER

TEN

GATEWARE BLINKY

This tutorial walks through the process of developing a simple "blinky" example for Cynthion's ECP5 FPGA. We'll use the Amaranth Language & toolchain to create the design and generate a FPGA bitstream using the Yosys Open SYnthesis Suite.

10.1 Prerequisites

Before you begin, please make sure you have installed the Cynthion tools by following *Getting Started with Cynthion*.

10.1.1 Install Toolchain

To generate bitstreams for Cynthion you will need a synthesis toolchain that can convert the Verilog produced by Amaranth into a bitstream for Cynthion's ECP5 FPGA.

For these tutorials we recommend YoWASP which provides unofficial WebAssembly-based packages for Yosys and NextPNR. It runs a little slower than the official OSS CAD Suite distribution but it's platform-independent and much easier to get started with.

Install YoWASP using pip:

```
pip install yowasp-yosys yowasp-nextpnr-ecp5
```

10.2 Create a new Amaranth module

Create a new file called `gateware-blinky.py` and add the following code to it:

```
from amaranth import *

class Top(Elaboratable):
    def elaborate(self, platform):
        m = Module()

        return m
```

Amaranth designs are built from a hierarchy of smaller modules, which are called *elaboratables*. The `Top` class expresses that this will be the top-level or entry-point of your design.

Right now the `Top` module does not do anything except to create an Amaranth `Module()` and return it from the `elaborate(...)` method.

The `elaborate(...)` method also takes an argument called `platform` that contains resources specific to the board or platform the module is compiled for.

In this case, the argument will be an instance of the Cynthion Board Description and contain a map of Cynthion resources such as LEDs, USB PHY's, USER PMOD connectors and board constraints.

10.3 Obtain a platform resource

Edit `gateware-blinky.py` and add the highlighted line:

```
from amaranth import *

class Top(Elaboratable):
    def elaborate(self, platform):
        m = Module()

        leds: Signal(6) = Cat(platform.request("led", n).o for n in range(0, 6))

        return m
```

Amaranth platform resources can be obtained via the `platform.request(name, number=0)` method where `name` is the name of the resource and `number` is the index of the resource if there are more than one defined.

In this case we use a Python list comprehension to obtain all six FPGA led's and concatenate them into a six-bit addressable Amaranth `Signal` using the `Cat` operation.

10.4 Timer State

To make the LED blink at predictable intervals we'll use a simple timer.

To start with, let's define the timer state by adding the highlighted lines:

```
from amaranth import *

class Top(Elaboratable):
    def elaborate(self, platform):
        m = Module()

        leds: Signal(6) = Cat(platform.request("led", n).o for n in range(0, 6))

        half_freq: int    = int(60e6 // 2)
        timer: Signal(25) = Signal(range(half_freq))

        return m
```

First we'll declare a variable `half_freq` which is exactly half of Cynthion FPGA's default clock frequency in Hz, next we'll declare `timer` to be an Amaranth `Signal` which is wide enough to contain a value equal to `half_freq - 1`.

If we increment the `timer` by one for each clock cycle until it reaches `half_freq - 1` we get a timer with a 500ms period.

10.5 Timer Logic

Now that we have a state definition for our timer we can move forward to the implementation logic, edit your file and add the highlighted lines:

```
from amaranth import *

class Top(Elaboratable):
    def elaborate(self, platform):
        m = Module()

        leds: Signal(6) = Cat(platform.request("led", n).o for n in range(0, 6))

        half_freq: int    = int(60e6 // 2)
        timer: Signal(25) = Signal(range(half_freq))

        with m.If(timer == half_freq - 1):
            m.d.sync += leds.eq(~leds)
            m.d.sync += timer.eq(0)

        with m.Else():
            m.d.sync += timer.eq(timer + 1)

        return m
```

Amaranth combines normal Python expressions with Amaranth in order to describe a design. Whenever you see the prefix `m.` you are making a call to the `Module` object you created at the beginning of the `elaborate(...)` method. These calls are what build the logic which makes up a design.

The `with m.If(...):` and `with m.Else():` blocks operate much like you'd expect where, every clock-cycle, the expression `timer == half_freq - 1` will be evaluated and trigger the corresponding branch.

The first block represents the point at which the timer has expired and we'd like to change the state of the LEDs and then reset the timer back to zero.

In the second block the timer is still active so we simply increment `timer` by one.

10.6 Put It All Together

The contents of `gateware-blinky.py` should now look like this:

```
#!/usr/bin/env python3
#
# This file is part of Cynthion.
#
# Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
# SPDX-License-Identifier: BSD-3-Clause

from amaranth import *

class Top(Elaboratable):
    def elaborate(self, platform):
        m = Module()
```

(continues on next page)

(continued from previous page)

```
        leds: Signal(6) = Cat(platform.request("led", n).o for n in range(0, 6))

        half_freq: int    = int(60e6 // 2)
        timer: Signal(25) = Signal(range(half_freq))

        with m.If(timer == half_freq - 1):
            m.d.sync += leds.eq(~leds)
            m.d.sync += timer.eq(0)

        with m.Else():
            m.d.sync += timer.eq(timer + 1)

        return m

if __name__ == "__main__":
    from luna import top_level_cli
    top_level_cli(Top)
```

10.7 Build and Upload FPGA Bitstream

Make sure your Cynthion CONTROL port is plugged into the host, open a terminal and then run:

```
python gateware-blinky.py
```

The blinky gateware will now be synthesized, placed, routed and automatically uploaded to the Cynthion's FPGA.

Once this process has completed successfully all six of Cynthion's FPGA LEDs should be flashing on and off.

10.8 Exercises

1. Modify the tutorial to turn the FPGA LEDs into a binary counter that increments by one every 250ms.
2. Connect the USER PMOD A port to the output of your counter and use a logic analyzer (e.g. GreatFET One) to view the values as they increment.

10.9 More information:

- Amaranth Language & tool change documentation.

CHAPTER

ELEVEN

USB GATEWARE: PART 1 - ENUMERATION

This series of tutorial walks through the process of implementing a complete USB device with Cynthion and LUNA:

- *USB Gateware: Part 1 - Enumeration* *(This tutorial)*
- *USB Gateware: Part 2 - WCID Descriptors*
- *USB Gateware: Part 3 - Control Transfers*
- *USB Gateware: Part 4 - Bulk Transfers*

The goal of this tutorial is to create a gateware design for the simplest USB Device that can still be enumerated by a host.

11.1 Prerequisites

- Install the Cynthion tools by following *Getting Started with Cynthion*.
- Complete the *Gateware Blinky* tutorial.

11.2 Define a USB Device

USB devices are defined using a hierarchy of descriptors that contain information such as:

- The product name and serial number.
- The vendor who made it.
- The class of device it is.
- The ways in which it can be configured.
- The number and types of endpoints it has.

At the root of this hierarchy lies the *Device Descriptor* and a device can only have one.

11.2.1 Create the Device Descriptor

Create a new file called `gateware-usb-device.py` and add the following code to it:

Listing 1: gateware-usb-device.py

```
from amaranth                import  *
from usb_protocol.emitters   import  DeviceDescriptorCollection

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

class GatewareUSBDevice(Elaboratable):
    def create_descriptors(self):
        descriptors = DeviceDescriptorCollection()

        with descriptors.DeviceDescriptor() as d:
            d.idVendor           = VENDOR_ID
            d.idProduct          = PRODUCT_ID
            d.iManufacturer      = "Cynthion Project"
            d.iProduct           = "Gateware USB Device"
            d.bNumConfigurations = 1

        return descriptors

    def elaborate(self, platform):
        m = Module()
        return m
```

We have now created a minimal device descriptor with a vendor id, product id, a manufacturer, a product description and one *Configuration Descriptor*.

USB devices can have multiple configurations but only one can be active at a time. This allows a USB device to be configured differently depending on the situtation. For example, a device might be configured differently if it's bus-powered vs self-powered.

11.2.2 Create the Configuration Descriptor

Next, add a configuration descriptor for our device by adding the highlighted lines:

Listing 2: gateware-usb-device.py

```
from amaranth                import *
from usb_protocol.emitters   import DeviceDescriptorCollection

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

class GatewareUSBDevice(Elaboratable):
    def create_descriptors(self):
        descriptors = DeviceDescriptorCollection()

        with descriptors.DeviceDescriptor() as d:
            d.idVendor           = VENDOR_ID
```

(continues on next page)

(continued from previous page)

```
            d.idProduct          = PRODUCT_ID
            d.iManufacturer      = "Cynthion Project"
            d.iProduct           = "Gateware USB Device"
            d.bNumConfigurations = 1

        with descriptors.ConfigurationDescriptor() as c:
            with c.InterfaceDescriptor() as i:
                i.bInterfaceNumber = 0

        return descriptors

    def elaborate(self, platform):
        m = Module()
        return m
```

We have now created the descriptors for a device with a single configuration descriptor and one interface descriptor with no endpoints. (We'll add some endpoints later!)

Note: Each USB Configuration can have multiple interface descriptors and they can all be active at the same time. This allows a USB device to create functional groups that are each responsible for a single function of the device. For example, a USB Audio Interface may have one interface descriptor with two endpoints for audio input/output and another interface descriptor with one endpoint for MIDI input.

11.2.3 Create Device Gateware

Now that we have defined our device's descriptors we need to create the interface between our device's physical USB port and the gateware that implements the device's function(s). Fortunately the LUNA library takes care of all the hard work for us and we only need to add the following lines:

Listing 3: gateware-usb-device.py

```
from amaranth               import *
from luna.usb2              import USBDevice
from usb_protocol.emitters  import DeviceDescriptorCollection

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

class GatewareUSBDevice(Elaboratable):
    def create_descriptors(self):
        descriptors = DeviceDescriptorCollection()

        with descriptors.DeviceDescriptor() as d:
            d.idVendor           = VENDOR_ID
            d.idProduct          = PRODUCT_ID
            d.iManufacturer      = "Cynthion Project"
            d.iProduct           = "Gateware USB Device"
            d.bNumConfigurations = 1

        with descriptors.ConfigurationDescriptor() as c:
```

(continues on next page)

(continued from previous page)

```
            with c.InterfaceDescriptor() as i:
                i.bInterfaceNumber = 0

        return descriptors

    def elaborate(self, platform):
        m = Module()

        # configure cynthion's clocks and reset signals
        m.submodules.car = platform.clock_domain_generator()

        # request the physical interface for cynthion's TARGET C port
        ulpi = platform.request("target_phy")
        m.submodules.usb = usb = USBDevice(bus=ulpi)

        # create our descriptors and add them to the device's control endpoint
        descriptors = self.create_descriptors()
        control_ep = usb.add_standard_control_endpoint(descriptors)

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m

if __name__ == "__main__":
    from luna import top_level_cli
    top_level_cli(GatewareUSBDevice)
```

11.3 Testing the Device

11.3.1 Connect

We need to connect our Cynthion before we can test our new USB device. If you followed the prerequisites above, you should already have connected the Cynthion's **CONTROL** port to your computer.

Now also connect the **TARGET C** port to your computer as this is the port we requested our USB Device to run on. The control host and target host can be two separate computers, but in this tutorial we will use the same computer as both the control host and the target host.

11.3.2 Build

Build the device gateware and upload it to your Cynthion by typing the following into your terminal shell:

```
python ./gateware-usb-device.py
```

If everything went well and Cynthion's **TARGET C** port is connected we should now be able to check if the target host managed to succesfully enumerate our device.

11.3.3 Test

To check if the device was recognized by the target host's operating system follow the corresponding instructions:

Create a new file called `test-gateware-usb-device.py` and add the following code to it:

Listing 4: test-gateware-usb-device.py

```
import usb1

def list_available_usb_devices(context):
    for device in context.getDeviceList():
        try:
            manufacturer = device.getManufacturer()
            product = device.getProduct()
            print(f"{device}:  {manufacturer} - {product}")
        except Exception as e:
            print(f"{device}: {e}")

if __name__ == "__main__":
    with usb1.USBContext() as context:
        list_available_usb_devices(context)
```

Run the file with:

```
python ./test-gateware-usb-device.py
```

And, if the device is recognized, you should see a line like:

```
Bus 000 Device 001: ID 1d5c:5010:  Fresco Logic, Inc. - USB2.0 Hub
Bus 000 Device 002: ID 1d5c:5000:  Fresco Logic, Inc. - USB3.0 Hub
Bus 000 Device 003: ID 1d50:615c:  Great Scott Gadgets - Cynthion Apollo Debugger
Bus 000 Device 007: ID 1209:0001:  Cynthion Project - Gateware USB Device
```

If you're running on Windows you may instead see something like:

```
Bus 000 Device 001: ID 1d5c:5010:  Fresco Logic, Inc. - USB2.0 Hub
Bus 000 Device 002: ID 1d5c:5000:  Fresco Logic, Inc. - USB3.0 Hub
Bus 000 Device 003: ID 1d50:615c:  Great Scott Gadgets - Cynthion Apollo Debugger
Bus 000 Device 007: ID 1209:0001:  LIBUSB_ERROR_NOT_SUPPORTED [-12]
```

The devices Product and Vendor ID's are correct (`1209:0001`) but Windows could not obtain the product or manufacturer strings. This behaviour is expected and we'll be taking a closer look at it in the next part of the tutorial.

Run the following command in a terminal window:

```
lsusb
```

If the device enumerated successfully you should see an entry similiar to the highlighted line:

```
% lsusb

Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 2109:2822 VIA Labs, Inc. USB2.0 Hub
Bus 001 Device 045: ID 1d50:615c OpenMoko, Inc. Cynthion Apollo Debugger
```

(continues on next page)

(continued from previous page)

```
Bus 001 Device 046: ID 1209:0001 Generic pid.codes Test PID
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
```

To view the device's descriptors, pass the product and vendor id's by running:

```
lsusb -d 1209:0001 -v
```

Run the following command in a terminal window:

```
ioreg -b -p IOUSB
```

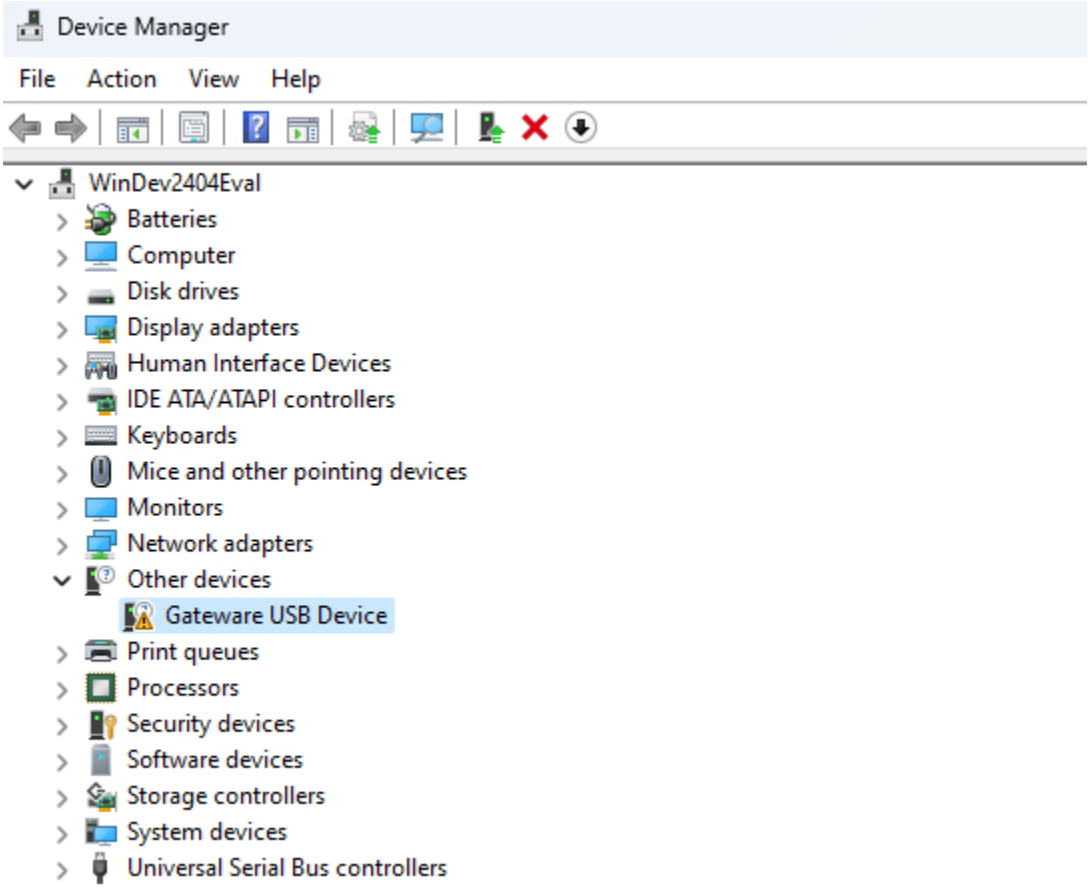
If the device enumerated successfully you should see an entry similiar to the highlighted line:

```
 % ioreg -b -p IOUSB

 +-o Root  <class IORegistryEntry, id 0x100000100, retain 30>
   +-o AppleT8103USBXHCI@00000000  <class AppleT8103USBXHCI, id 0x100000331, registered,␣
↪matched, ac$
     +-o USB2.0 Hub@00100000  <class IOUSBHostDevice, id 0x1000ee65d, registered,␣
↪matched, active, b$
     | +-o USB2.0 Hub@00140000  <class IOUSBHostDevice, id 0x1000ee6b0, registered,␣
↪matched, active,$
     | | +-o Cynthion Apollo Debugger@00144000  <class IOUSBHostDevice, id 0x100180243,␣
↪registered, $
     | +-o Gateware USB Device@00130000  <class IOUSBHostDevice, id 0x100181cb3,␣
↪registered, matched$
     +-o USB3.0 Hub@00200000  <class IOUSBHostDevice, id 0x100181add, registered,␣
↪matched, active, b$
       +-o USB3.0 Hub@00240000  <class IOUSBHostDevice, id 0x100181aef, registered,␣
↪matched, active,$
```

To view more information, pass the device name:

```
ioreg -b -p IOUSB -n "Gateware USB Device"
```

The easiest way to check a USB device is to open the Windows Device Manager. However, if you try this with our device you will notice there's a small problem:



We can see our device, but it has a warning icon indicating that it does not have an installed device driver. Unlike macOS or Linux, Windows does not support a generic USB driver for non-class devices with custom vendor interfaces. In the next part of the tutorial we'll look at how to solve this.

11.4 Conclusion

Our device can now be enumerated by a host but, if you're running Microsoft Windows, you will have noticed that the device still requires a device driver to function.

The next part of the tutorial is optional and will cover WCID Descriptors which is a mechanism introduced by Microsoft to allow Windows applications to communicate directly with USB devices without the neccessity of writing device drivers.

If you don't need to target Windows please feel free to skip the next part and jump straight to *USB Gateware: Part 3 - Control Transfers* to learn how to add the Control Request Handlers to our device that allow it to receive and respond to control requests from the host.

11.5 Exercises

1. Try changing the device descriptor information to match an existing hardware USB device. What happens?

11.6 More information

- Amaranth Language & tool change documentation.
- Beyond Logic's USB in a NutShell.
- LUNA Documentation

11.7 Source Code

Listing 5: gateware-usb-device-01.py

```
#!/usr/bin/env python3
#
# This file is part of Cynthion.
#
# Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
# SPDX-License-Identifier: BSD-3-Clause

from amaranth               import *
from luna.usb2              import USBDevice
from usb_protocol.emitters  import DeviceDescriptorCollection

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

class GatewareUSBDevice(Elaboratable):
    """ A simple USB device that can only enumerate. """

    def create_standard_descriptors(self):
        """ Create the USB descriptors for the device. """

        descriptors = DeviceDescriptorCollection()

        # all USB devices have a single device descriptor
        with descriptors.DeviceDescriptor() as d:
            d.idVendor           = VENDOR_ID
            d.idProduct          = PRODUCT_ID
            d.iManufacturer      = "Cynthion Project"
            d.iProduct           = "Gateware USB Device"

            d.bNumConfigurations = 1

        # and at least one configuration descriptor
        with descriptors.ConfigurationDescriptor() as c:

```

(continues on next page)

(continued from previous page)

```
            # with at least one interface descriptor
            with c.InterfaceDescriptor() as i:
                i.bInterfaceNumber = 0

                # interfaces also need endpoints to do anything useful
                # but we'll add those later!

        return descriptors


    def elaborate(self, platform):
        m = Module()

        # configure cynthion's clocks and reset signals
        m.submodules.car = platform.clock_domain_generator()

        # request the physical interface for cynthion's TARGET C port
        ulpi = platform.request("target_phy")

        # create the USB device
        m.submodules.usb = usb = USBDevice(bus=ulpi)

        # create our standard descriptors and add them to the device's control endpoint
        descriptors = self.create_standard_descriptors()
        control_endpoint = usb.add_standard_control_endpoint(descriptors)

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m


if __name__ == "__main__":
    from luna import top_level_cli
    top_level_cli(GatewareUSBDevice)
```

Listing 6: test-gateware-usb-device-01.py

```
import usb1

# - list available usb devices ------------------------------------------------

def list_available_usb_devices(context):
    for device in context.getDeviceList():
        try:
            manufacturer = device.getManufacturer()
            product = device.getProduct()
            print(f"{device}:  {manufacturer} - {product}")
        except Exception as e:
            print(f"{device}: {e}")


```

(continues on next page)

(continued from previous page)

```
# - main ------------------------------------------------------------------

if __name__ == "__main__":
    with usb1.USBContext() as context:
        list_available_usb_devices(context)
```

CHAPTER

TWELVE

USB GATEWARE: PART 2 - WCID DESCRIPTORS

This series of tutorial walks through the process of implementing a complete USB device with Cynthion and LUNA:

- *USB Gateware: Part 1 - Enumeration*
- *USB Gateware: Part 2 - WCID Descriptors (This tutorial)*
- *USB Gateware: Part 3 - Control Transfers*
- *USB Gateware: Part 4 - Bulk Transfers*

The goal of this tutorial is to define the descriptors that will tell Microsoft Windows to use the built-in generic WinUSB driver to communicate with our device.

This tutorial is optional and only required if you would like to use your device on Windows.

12.1 Prerequisites

- Complete the *USB Gateware: Part 1 - Enumeration* tutorial.

12.2 WCID Devices

WCID devices or "Windows Compatible ID devices", are USB devices that provide extra information to Windows in order to facilitate automatic driver installation or, more frequently, allow programs to obtain direct access to the device.

Historically, Windows required manual installation of drivers for non-class devices with custom vendor interfaces. Contrasted with Linux or macOS which will automatically assign a generic USB driver that allows for direct interaction with the device's endpoints via a cross-platform library such as libusb or operating system API's.

Microsoft eventually relented and now provide a Windows-specific mechanism for a device to advertise that it requires a generic WinUSB driver.

The full details are documented in the Microsoft OS 1.0 and Microsoft OS 2.0 specifications but the basic mechanism consists of a set of Windows-specific descriptor requests made by the host whenever a new device is plugged in.

For Microsoft OS 1.0, this boils down to three descriptor requests we need to be able to handle:

1. Microsoft OS String Descriptor
2. Microsoft Compatible ID Feature Descriptor
3. Microsoft Extended Properties Feature Descriptor

12.2.1 Microsoft OS String Descriptor

To start with, edit your `gateware-usb-device.py` file from the previous tutorial and add/modify the highlighted lines:

Listing 1: gateware-usb-device.py

```
from amaranth                                       import *
from luna.usb2                                      import USBDevice
from usb_protocol.emitters                          import DeviceDescriptorCollection

from usb_protocol.emitters.descriptors.standard  import get_string_descriptor

...

class GatewareUSBDevice(Elaboratable):
    ...

    def elaborate(self, platform):
        m = Module()

        # configure cynthion's clocks and reset signals
        m.submodules.car = platform.clock_domain_generator()

        # request the physical interface for cynthion's TARGET C port
        ulpi = platform.request("target_phy")

        # create the USB device
        m.submodules.usb = usb = USBDevice(bus=ulpi)

        # create our standard descriptors and add them to the device's control endpoint
        descriptors = self.create_standard_descriptors()
        control_endpoint = usb.add_standard_control_endpoint(descriptors)

        # add the microsoft os string descriptor
        descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m
```

The Microsoft OS String Descriptor responds to a standard String Descriptor request with an index of `0xee`. It encodes two values:

```
0x12,         # Descriptor Length: 18 bytes
0x03,         # Descriptor Type: 3 = String
0x4d, 0x00,   # M
0x53, 0x00,   # S
0x46, 0x00,   # F
0x54, 0x00,   # T
0x31, 0x00,   # 1
0x30, 0x00,   # 0
0x30, 0x00,   # 0
```

(continues on next page)

(continued from previous page)

```
0xee, 0x00,   # Vendor Code: 0xee
```

The first 14 bytes correspond to the little-endian encoded Unicode string `MSFT100`, with the remaining two bytes corresponding to the Vendor Code Windows should use when requesting the other descriptors. This is often set to the same value as the Microsoft OS String Descriptor index of `0xee`, but you can use another value if it conflicts with an existing Vendor Code used by your device.

12.2.2 Microsoft Compatible ID Feature Descriptor

Next, add the Microsoft Compatible ID Feature Descriptor:

Listing 2: gateware-usb-device.py

```
from amaranth                                       import *
from luna.usb2                                      import USBDevice
from usb_protocol.emitters                          import DeviceDescriptorCollection

from luna.gateware.usb.request.windows              import (
    MicrosoftOS10DescriptorCollection,
)
from usb_protocol.emitters.descriptors.standard     import get_string_descriptor


VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

class GatewareUSBDevice(Elaboratable):
    ...

    def elaborate(self, platform):
        ...

        # add the microsoft os string descriptor
        descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)

        # add a microsoft descriptor collection for our other two microsoft descriptors
        msft_descriptors = MicrosoftOS10DescriptorCollection()

        # add the microsoft compatible id feature descriptor
        with msft_descriptors.ExtendedCompatIDDescriptor() as c:
            with c.Function() as f:
                f.bFirstInterfaceNumber = 0
                f.compatibleID          = 'WINUSB'

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m
```

Our remaining descriptors are not returned via Standard Requests, instead they are implemented as Vendor Requests with Microsoft-defined Vendor Indices and the Vendor Code supplied in the Microsoft OS String Descriptor. We will implement the actual vendor request handler in the final step of the tutorial but for now we are just defining the Microsoft

OS 1.0 Descriptor Collection that will contain these descriptors.

Our example is defining the simplest possible Compatible ID Feature descriptor, specifying a Function with a device interface number of `0` and a compatible ID of `WINUSB`. This is how we tell Windows to use the generic WinUSB driver for the interface.

If our device had multiple interfaces we could simply extended this by adding additional functions for each interface like so:

```
with msft_descriptors.ExtendedCompatIDDescriptor() as c:
    with c.Function() as f:
        f.bFirstInterfaceNumber = 0
        f.compatibleID          = 'WINUSB'
    with c.Function() as f:
        f.bFirstInterfaceNumber = 1
        f.compatibleID          = 'WINUSB'
    ...
```

12.2.3 Microsoft Extended Properties Feature Descriptor

We now come to our third descriptor, the Microsoft Extended Properties Feature Descriptor:

Listing 3: gateware-usb-device.py

```
from amaranth                                       import *
from luna.usb2                                      import USBDevice
from usb_protocol.emitters                          import DeviceDescriptorCollection

from luna.gateware.usb.request.windows              import (
    MicrosoftOS10DescriptorCollection,
)
from usb_protocol.emitters.descriptors.standard     import get_string_descriptor
from usb_protocol.types.descriptors.microsoft10     import RegistryTypes

..

class GatewareUSBDevice(Elaboratable):
    ...

    def elaborate(self, platform):
        ...

        # add the microsoft os string descriptor
        descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)

        # add a microsoft descriptor collection for our other two microsoft descriptors
        msft_descriptors = MicrosoftOS10DescriptorCollection()

        # add the microsoft compatible id feature descriptor
        with msft_descriptors.ExtendedCompatIDDescriptor() as c:
            with c.Function() as f:
                f.bFirstInterfaceNumber = 0
                f.compatibleID          = 'WINUSB'
```

(continues on next page)

(continued from previous page)

```

        # add microsoft extended properties feature descriptor
        with msft_descriptors.ExtendedPropertiesDescriptor() as d:
            with d.Property() as p:
                p.dwPropertyDataType = RegistryTypes.REG_SZ
                p.PropertyName       = "DeviceInterfaceGUID"
                p.PropertyData       = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m
```

The Extended Properties Feature Descriptor can be used to define additional device registry settings but, in our example, we only define the Device Interface GUID we'd like our device to be accessed with.

In this case it's the Microsoft-defined GUID of `{88bae032-5a81-49f0-bc3d-a4ff138216d6}` which is defined as *"all USB devices that don't belong to another class"*. If, for example, our device were a Keyboard or Mouse we'd need to use the appropriate value here.

12.2.4 Microsoft Descriptor Request Handler

Finally, now that all our descriptors are defined we need to add the actual Vendor Request Handler that will be responsible for responding to descriptor requests from a Windows Host:

Listing 4: gateware-usb-device.py

```
from amaranth                                       import *
from luna.usb2                                      import USBDevice
from usb_protocol.emitters                          import DeviceDescriptorCollection

from luna.gateware.usb.request.windows              import (
    MicrosoftOS10DescriptorCollection,
    MicrosoftOS10RequestHandler,
)
from usb_protocol.emitters.descriptors.standard     import get_string_descriptor
from usb_protocol.types.descriptors.microsoft10     import RegistryTypes

..

class GatewareUSBDevice(Elaboratable):
    ...

    def elaborate(self, platform):
        ...

        # add the microsoft os string descriptor
        descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)

        # add a microsoft descriptor collection for our other two microsoft descriptors
        msft_descriptors = MicrosoftOS10DescriptorCollection()

```

(continues on next page)

(continued from previous page)

```
        # add the microsoft compatible id feature descriptor
        with msft_descriptors.ExtendedCompatIDDescriptor() as c:
            with c.Function() as f:
                f.bFirstInterfaceNumber = 0
                f.compatibleID          = 'WINUSB'

        # add microsoft extended properties feature descriptor
        with msft_descriptors.ExtendedPropertiesDescriptor() as d:
            with d.Property() as p:
                p.dwPropertyDataType = RegistryTypes.REG_SZ
                p.PropertyName       = "DeviceInterfaceGUID"
                p.PropertyData       = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"

        # add the request handler for Microsoft descriptors
        msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
        control_endpoint.add_request_handler(msft_handler)

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m
```

LUNA provides a pre-defined implementation for handling Microsoft OS10 Descriptor Requests and only requires the descriptor collection and the vendor request code we defined in the Microsoft OS10 String Descriptor.

12.3 Testing the Device

12.3.1 Connect

- For this tutorial you will need to connect the Cynthion **TARGET C** port to a Windows computer for testing.
- Plug the **CONTROL** port into the computer you've been using to control Cynthion. If this is the same machine as the Windows computer you're using to test, plug it in there.

12.3.2 Build

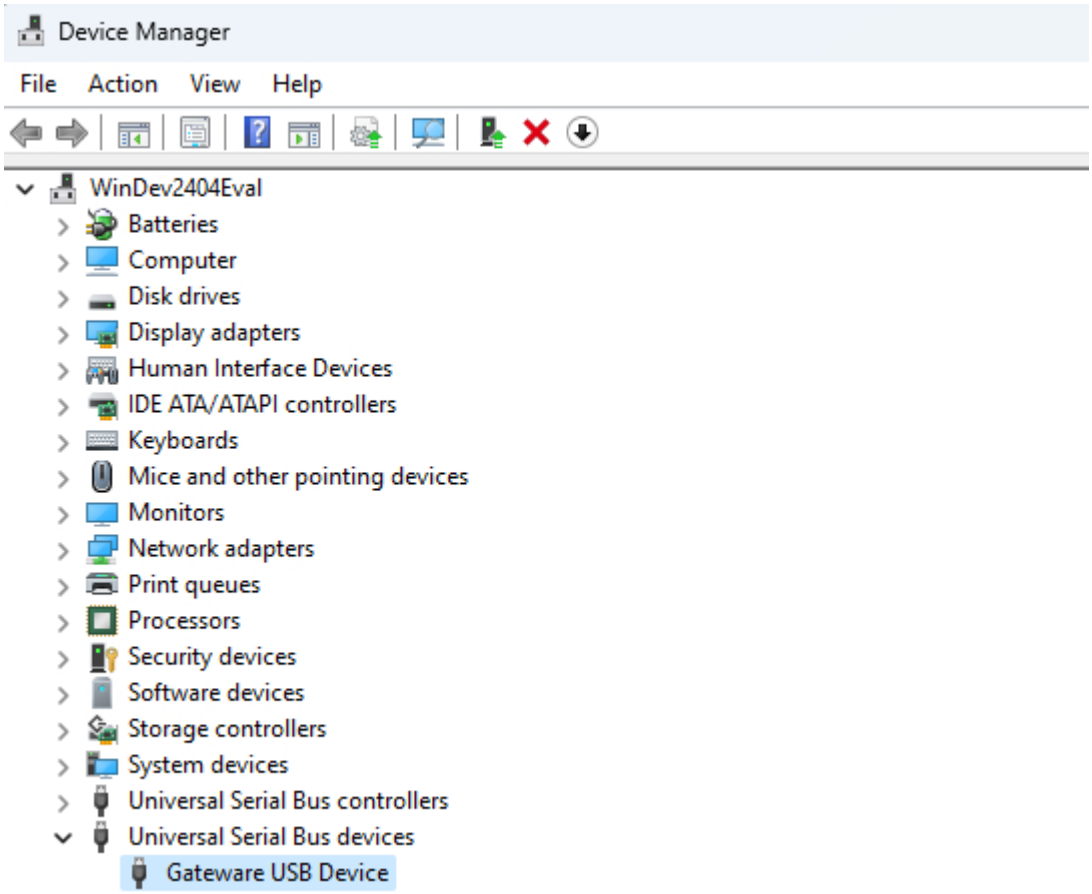
Build the device gateware and upload it to your Cynthion by typing the following into your terminal shell:

```
python ./gateware-usb-device.py
```

If everything went well we should now be able to check if Windows can recognize the device.

12.3.3 Test

To test whether the WCID descriptors have been recognized, open the Windows Device Manager and look for the device under the *"Universal Serial Bus devices"* section:



You should find that the Python test program from *USB Gateware: Part 1 - Enumeration* now works as expected:

Listing 5: test-gateware-usb-device.py

```
import usb1

def list_devices(context):
    for device in context.getDeviceList():
        try:
            manufacturer = device.getManufacturer()
            product = device.getProduct()
            print(f"{device}:  {manufacturer} - {product}")
        except Exception as e:
            print(f"{device}: {e}")

if __name__ == "__main__":
    with usb1.USBContext() as context:
        list_devices(context)
```

Run the file with:

```
python ./test-gateware-usb-device.py
```

And, if the device is recognized, you should see a line like:

```
Bus 000 Device 001: ID 1d5c:5010:  Fresco Logic, Inc. - USB2.0 Hub
Bus 000 Device 002: ID 1d5c:5000:  Fresco Logic, Inc. - USB3.0 Hub
Bus 000 Device 003: ID 1d50:615c:  Great Scott Gadgets - Cynthion Apollo Debugger
Bus 000 Device 007: ID 1209:0001:  Cynthion Project - Gateware USB Device
```

12.4 Conclusion

Our device can now be enumerated by Microsoft Windows but it can't actually do anything yet. In the next part we'll learn how to add Vendor Request Handlers to our device that allow it to receive and respond to control requests from the host: *USB Gateware: Part 3 - Control Transfers*

12.5 Exercises

- Modify the example to use a different request code, does it still work?
- Could you use the information you learnt in this tutorial modify the LUNA ACM Serial example example to support Windows?
- Modify the `PropertyData` field of the extended properties descriptor to one of the Microsoft-provided USB device class drivers. What happens?

12.6 More information

- Pete Batard's excellent introduction to WCID Devices.
- Microsoft OS 1.0 Descriptors Specification.
- Microsoft OS 2.0 Descriptors Specification.
- Microsoft USB device class drivers included in Windows.
- Microsoft System-defined device setup classes available to vendors.

12.7 Source Code

Listing 6: gateware-usb-device-02.py

```
#!/usr/bin/env python3
#
# This file is part of Cynthion.
#
# Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
# SPDX-License-Identifier: BSD-3-Clause

from amaranth                                       import *
```

(continues on next page)

(continued from previous page)

```
from luna.usb2                                      import USBDevice
from usb_protocol.emitters                          import DeviceDescriptorCollection

from luna.gateware.usb.request.windows              import (
    MicrosoftOS10DescriptorCollection,
    MicrosoftOS10RequestHandler,
)
from usb_protocol.emitters.descriptors.standard     import get_string_descriptor
from usb_protocol.types.descriptors.microsoft10     import RegistryTypes

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

class GatewareUSBDevice(Elaboratable):
    """ A simple USB device that can also enumerate on Windows. """

    def create_standard_descriptors(self):
        """ Create the USB descriptors for the device. """

        descriptors = DeviceDescriptorCollection()

        # all USB devices have a single device descriptor
        with descriptors.DeviceDescriptor() as d:
            d.idVendor           = VENDOR_ID
            d.idProduct          = PRODUCT_ID
            d.iManufacturer      = "Cynthion Project"
            d.iProduct           = "Gateware USB Device"

            d.bNumConfigurations = 1

        # and at least one configuration descriptor
        with descriptors.ConfigurationDescriptor() as c:

            # with at least one interface descriptor
            with c.InterfaceDescriptor() as i:
                i.bInterfaceNumber = 0

                # interfaces also need endpoints to do anything useful
                # but we'll add those later!

        return descriptors


    def elaborate(self, platform):
        m = Module()

        # configure cynthion's clocks and reset signals
        m.submodules.car = platform.clock_domain_generator()

        # request the physical interface for cynthion's TARGET C port
        ulpi = platform.request("target_phy")

```

(continues on next page)

(continued from previous page)

```
        # create the USB device
        m.submodules.usb = usb = USBDevice(bus=ulpi)

        # create our standard descriptors and add them to the device's control endpoint
        descriptors = self.create_standard_descriptors()
        control_endpoint = usb.add_standard_control_endpoint(descriptors)

        # add the microsoft os string descriptor
        descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)

        # add a microsoft descriptor collection for our other two microsoft descriptors
        msft_descriptors = MicrosoftOS10DescriptorCollection()

        # add the microsoft compatible id feature descriptor
        with msft_descriptors.ExtendedCompatIDDescriptor() as c:
            with c.Function() as f:
                f.bFirstInterfaceNumber = 0
                f.compatibleID          = 'WINUSB'

        # add microsoft extended properties feature descriptor
        with msft_descriptors.ExtendedPropertiesDescriptor() as d:
            with d.Property() as p:
                p.dwPropertyDataType = RegistryTypes.REG_SZ
                p.PropertyName       = "DeviceInterfaceGUID"
                p.PropertyData       = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"

        # add the request handler for Microsoft descriptors
        msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
        control_endpoint.add_request_handler(msft_handler)

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m


if __name__ == "__main__":
    from luna import top_level_cli
    top_level_cli(GatewareUSBDevice)
```

Listing 7: test-gateware-usb-device-02.py

```
import usb1

# - list available usb devices ------------------------------------------------

def list_available_usb_devices(context):
    for device in context.getDeviceList():
        try:
            manufacturer = device.getManufacturer()
            product = device.getProduct()
            print(f"{device}:  {manufacturer} - {product}")
```

(continues on next page)

(continued from previous page)

```
        except Exception as e:
            print(f"{device}: {e}")


# - main ---------------------------------------------------------------

if __name__ == "__main__":
    with usb1.USBContext() as context:
        list_available_usb_devices(context)
```

CHAPTER

THIRTEEN

USB GATEWARE: PART 3 - CONTROL TRANSFERS

This series of tutorial walks through the process of implementing a complete USB device with Cynthion and LUNA:

- *USB Gateware: Part 1 - Enumeration*
- *USB Gateware: Part 2 - WCID Descriptors*
- *USB Gateware: Part 3 - Control Transfers* *(This tutorial)*
- *USB Gateware: Part 4 - Bulk Transfers*

The goal of this tutorial is to define a control interface for the device we created in Part 1 that will allow it to receive and respond to control requests from a host.

13.1 Prerequisites

- Complete the *USB Gateware: Part 1 - Enumeration* tutorial.
- Complete the *USB Gateware: Part 2 - WCID Descriptors* tutorial. *(Optional, required for Windows support)*

13.2 Data Transfer between a Host and Device

USB is a *host-centric bus*, what this means is that all transfers are initiated by the host irrespective of the direction of data transfer.

For data transfers to the device, the host issues an OUT token to notify the device of an incoming data transfer. When data has to be transferred from the device, the host issues an IN token to notify the device that it should send some data to the host.

The USB 2.0 specification defines four endpoint or transfer types:

- **Control Transfers:** Typically used for command and status operations, control transfers are the only transfer type with a defined USB format.
- **Bulk Transfers:** Bulk transfers are best suited for large amounts of data delivered in bursts such as file transfers to/from a storage device or the captured packet data from Cynthion to the control host.
- **Interrupt Transfers:** Interrupt transfers are a bit of a misnomer as the host needs to continuously poll the device to check if an interrupt has occurred but the principle is the same. Commonly used for peripherals that generate input events such as a keyboard or mouse.
- **Isochronous Transfers:** Finally, isochronous transfers occur continuously with a fixed periodicity. Suited for time-sensitive information such as video or audio streams they do not offer any guarantees on delivery. If a packet or frame is dropped it's is up to the host driver to decide on how to best handle it.

By default all LUNA devices have a default implementation for two endpoints: An OUT Control Endpoint and an IN Control endpoint. These endpoints are used by the host to enumerate the device but they can also be extended to support various other class or custom vendor requests.

We'll start by extending our control endpoints to support two vendor requests: One to set the state of the Cynthion **FPGA LEDs** and another to get the state of the Cynthion **USER BUTTON**.

13.2.1 Extend Default Control Endpoints

To implement vendor requests, begin by adding a `VendorRequestHandler` to our device's control endpoint:

Listing 1: gateware-usb-device.py

```
from amaranth                                       import *
from luna.usb2                                      import USBDevice
from usb_protocol.emitters                          import DeviceDescriptorCollection

from luna.gateware.usb.request.windows              import (
    MicrosoftOS10DescriptorCollection,
    MicrosoftOS10RequestHandler,
)
from usb_protocol.emitters.descriptors.standard     import get_string_descriptor
from usb_protocol.types.descriptors.microsoft10     import RegistryTypes

from luna.gateware.stream.generator                 import StreamSerializer
from luna.gateware.usb.request.control              import ControlRequestHandler
from luna.gateware.usb.usb2.transfer                import USBInStreamInterface

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

class VendorRequestHandler(ControlRequestHandler):
    VENDOR_SET_FPGA_LEDS   = 0x01
    VENDOR_GET_USER_BUTTON = 0x02

    def elaborate(self, platform):
        m = Module()

        # shortcuts
        interface: RequestHandlerInterface = self.interface
        setup: SetupPacket = self.interface.setup

        # get a reference to the FPGA LEDs and USER button
        fpga_leds   = Cat(platform.request("led", i).o for i in range(6))
        user_button = platform.request("button_user").i

        # create a streamserializer for transmitting IN data back to the host
        serializer = StreamSerializer(
            domain           = "usb",
            stream_type      = USBInStreamInterface,
            data_length      = 1,
            max_length_width = 1,
        )
```

(continues on next page)

(continued from previous page)

```
        m.submodules += serializer

        return m

class GatewareUSBDevice(Elaboratable):

    ...

    def elaborate(self, platform):
        m = Module()

        # configure cynthion's clocks and reset signals
        m.submodules.car = platform.clock_domain_generator()

        # request the physical interface for cynthion's TARGET C port
        ulpi = platform.request("target_phy")

        # create the USB device
        m.submodules.usb = usb = USBDevice(bus=ulpi)

        # create our standard descriptors and add them to the device's control endpoint
        descriptors = self.create_standard_descriptors()
        control_endpoint = usb.add_standard_control_endpoint(descriptors)

        # add microsoft os 1.0 descriptors and request handler
        descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
        msft_descriptors = MicrosoftOS10DescriptorCollection()
        with msft_descriptors.ExtendedCompatIDDescriptor() as c:
            with c.Function() as f:
                f.bFirstInterfaceNumber = 0
                f.compatibleID          = 'WINUSB'
        with msft_descriptors.ExtendedPropertiesDescriptor() as d:
            with d.Property() as p:
                p.dwPropertyDataType = RegistryTypes.REG_SZ
                p.PropertyName       = "DeviceInterfaceGUID"
                p.PropertyData       = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"
        msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
        control_endpoint.add_request_handler(msft_handler)

        # add our vendor request handler
        control_endpoint.add_request_handler(VendorRequestHandler())

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m
```

Vendor requests are unique to a device and are identified by the 8-bit `bRequest` field of the control transfer setup packet. Here we've defined two id's corresponding to setting the led states and getting the button state.

So far our `VendorRequestHandler` contains references to Cynthion's **FPGA LEDs** and **USER BUTTON**, as well as a **StreamSerializer** we'll be using to send data back to the host when it asks for the **USER BUTTON** status.

13.2.2 Implement Vendor Request Handlers

Let's implement that functionality below:

Listing 2: gateware-usb-device.py

```
class VendorRequestHandler(ControlRequestHandler):
    VENDOR_SET_FPGA_LEDS   = 0x01
    VENDOR_GET_USER_BUTTON = 0x02

    def elaborate(self, platform):
        m = Module()

        # Shortcuts.
        interface: RequestHandlerInterface = self.interface
        setup: SetupPacket = self.interface.setup

        # Grab a reference to the FPGA LEDs and USER button.
        fpga_leds   = Cat(platform.request("led", i).o for i in range(6))
        user_button = platform.request("button_user").i

        # Create a StreamSerializer for sending IN data back to the host
        serializer = StreamSerializer(
            domain           = "usb",
            stream_type      = USBInStreamInterface,
            data_length      = 1,
            max_length_width = 1,
        )
        m.submodules += serializer

        # we've received a setup packet containing a vendor request.
        with m.If(setup.type == USBRequestType.VENDOR):
            # use a state machine to sequence our request handling
            with m.FSM(domain="usb"):
                with m.State("IDLE"):
                    with m.If(setup.received):
                        with m.Switch(setup.request):
                            with m.Case(self.VENDOR_SET_FPGA_LEDS):
                                m.next = "HANDLE_SET_FPGA_LEDS"
                            with m.Case(self.VENDOR_GET_USER_BUTTON):
                                m.next = "HANDLE_GET_USER_BUTTON"

                with m.State("HANDLE_SET_FPGA_LEDS"):
                    # take ownership of the interface
                    m.d.comb += interface.claim.eq(1)

                    # if we have an active data byte, set the FPGA LEDs to the payload
                    with m.If(interface.rx.valid & interface.rx.next):
                        m.d.usb += fpga_leds.eq(interface.rx.payload[0:6])

                    # once the receive is complete, respond with an ACK
                    with m.If(interface.rx_ready_for_response):
                        m.d.comb += interface.handshakes_out.ack.eq(1)
```

(continues on next page)

(continued from previous page)

```
                    # finally, once we reach the status stage, send a ZLP
                    with m.If(interface.status_requested):
                        m.d.comb += self.send_zlp()
                        m.next = "IDLE"

                with m.State("HANDLE_GET_USER_BUTTON"):
                    # take ownership of the interface
                    m.d.comb += interface.claim.eq(1)

                    # write the state of the user button into a local data register
                    data = Signal(8)
                    m.d.comb += data[0].eq(user_button)

                    # transmit our data using a built-in handler function that
                    # automatically advances the FSM back to the 'IDLE' state on
                    # completion
                    self.handle_simple_data_request(m, serializer, data)

        return m
```

When handling a control request in LUNA the first thing we look at is the `setup.type` field of the setup packet interface. We could check for other types such as `USBRequestType.CLASS` or `USBRequestType.DEVICE` if we wanted to implement handlers for them but, in this case, we're only interested in vendor requests.

Next, we take ownership of the interface, in order to avoid conflicting with the standard, or other registered request handlers. Then we sequence the actual request handling with an Amaranth Finite State Machine, starting in the `IDLE` state.

While in `IDLE` we wait for the `setup.received` signal to go high and signal the arrival of a new control request. We then parse the `setup.request` field to identify the next state to advance our FSM to. (We could also use the other setup packet fields such as `wValue` and `wIndex` for dispatch or as arguments but for now we're just intered in `bRequest`.)

We then implement two handlers, the first is `HANDLE_SET_FPGA_LEDS`, which needs to read the data sent with our OUT control request in order to set the fpga leds state.

Then the second, in `HANDLE_GET_USER_BUTTON` we will use one of the built-in LUNA helper function to respond to our IN control request with the data containing the state of the user button.

13.3 Test Control Endpoints

First, remember to build and upload the device gateware to your Cynthion with:

```
python ./gateware-usb-device.py
```

Then, open your `test-gateware-usb-device.py` script from the previous tutorials and add the following code to it:

Listing 3: test-gateware-usb-device.py

```
import usb1
import time

```

(continues on next page)

(continued from previous page)

```
VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

VENDOR_SET_FPGA_LEDS   = 0x01
VENDOR_GET_USER_BUTTON = 0x02

# - list available usb devices ------------------------------------------------

def list_available_usb_devices(context):
    for device in context.getDeviceList():
        try:
            manufacturer = device.getManufacturer()
            product = device.getProduct()
            print(f"{device}:  {manufacturer} - {product}")
        except Exception as e:
            print(f"{device}: {e}")


# - wrappers for control requests ---------------------------------------------

def set_fpga_leds(device_handle, led_state):
    response = device_handle.controlWrite(
        request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
        request      = VENDOR_SET_FPGA_LEDS,
        index        = 0,
        value        = 0,
        data         = [led_state],
        timeout      = 1000,
    )

def get_user_button(device_handle):
    response = device_handle.controlRead(
        request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE | usb1.ENDPOINT_OUT,
        request      = VENDOR_GET_USER_BUTTON,
        index        = 0,
        value        = 0,
        length       = 1,
        timeout      = 1000,
    )
    return response[0]


# - test control endpoints ----------------------------------------------------

def test_control_endpoints(device_handle):
    led_counter = 0
    last_button_state = False

    while True:
        # led counter
        set_fpga_leds(device_handle, led_counter)
        led_counter = (led_counter + 1) % 256
```

(continues on next page)

(continued from previous page)

```
        # reset led counter when the USER button is pressed
        button_state = get_user_button(device_handle)
        if button_state:
            led_counter = 0

        # print button state when it changes
        if button_state != last_button_state:
            print(f"USER button is: {'ON' if button_state else 'OFF' }")
            last_button_state = button_state

        # slow the loop down so we can see the counter change
        time.sleep(0.1)


# - main ---------------------------------------------------------------------

if __name__ == "__main__":
    with usb1.USBContext() as context:
        # list available devices
        list_available_usb_devices(context)

        # get a device handle to our simple usb device
        device_handle = context.openByVendorIDAndProductID(VENDOR_ID, PRODUCT_ID)
        if device_handle is None:
            raise Exception("Device not found.")

        # claim the device's interface
        device_handle.claimInterface(0)

        # pass the device handle to our control endpoint test
        test_control_endpoints(device_handle)
```

Run the file with:

```
python ./test-gateware-usb-device.py
```

And, if all goes well you should see the **FPGA LEDs** on Cynthion counting in binary. If you press and release the **USER** button you should see the count reset back to zero and the following text in the terminal.

```
USER button is: ON
USER button is: OFF
```

Job done!

In the next part of the tutorial we'll finish up by adding IN and OUT Bulk endpoints to our device.

13.4 Exercises

1. Add a vendor request to retrieve the current state of the FPGA LEDs.
2. Add a vendor request that will disconnect and then re-connect your device to the USB bus.

13.5 More information

- Beyond Logic's USB in a NutShell.
- LUNA Documentation

13.6 Source Code

Listing 4: gateware-usb-device-03.py

```
#!/usr/bin/env python3
#
# This file is part of Cynthion.
#
# Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
# SPDX-License-Identifier: BSD-3-Clause

from amaranth                                     import *
from luna.usb2                                    import USBDevice
from usb_protocol.emitters                        import DeviceDescriptorCollection

from luna.gateware.usb.request.windows            import (
    MicrosoftOS10DescriptorCollection,
    MicrosoftOS10RequestHandler,
)
from usb_protocol.emitters.descriptors.standard   import get_string_descriptor
from usb_protocol.types.descriptors.microsoft10   import RegistryTypes

from luna.gateware.stream.generator               import StreamSerializer
from luna.gateware.usb.request.control            import ControlRequestHandler
from luna.gateware.usb.request.interface          import SetupPacket
from luna.gateware.usb.usb2.request               import RequestHandlerInterface
from luna.gateware.usb.usb2.transfer              import USBInStreamInterface
from usb_protocol.types                           import USBRequestType

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

class VendorRequestHandler(ControlRequestHandler):
    VENDOR_SET_FPGA_LEDS   = 0x01
    VENDOR_GET_USER_BUTTON = 0x02

    def elaborate(self, platform):
        m = Module()
```

(continues on next page)

(continued from previous page)

```
        # shortcuts
        interface: RequestHandlerInterface = self.interface
        setup: SetupPacket = self.interface.setup

        # get a reference to the FPGA LEDs and USER button
        fpga_leds   = Cat(platform.request("led", i).o for i in range(6))
        user_button = platform.request("button_user").i

        # create a streamserializer for transmitting IN data back to the host
        serializer = StreamSerializer(
            domain           = "usb",
            stream_type      = USBInStreamInterface,
            data_length      = 1,
            max_length_width = 1,
        )
        m.submodules += serializer

        # we've received a setup packet containing a vendor request.
        with m.If(setup.type == USBRequestType.VENDOR):
            # use a state machine to sequence our request handling
            with m.FSM(domain="usb"):
                with m.State("IDLE"):
                    with m.If(setup.received):
                        with m.Switch(setup.request):
                            with m.Case(self.VENDOR_SET_FPGA_LEDS):
                                m.next = "HANDLE_SET_FPGA_LEDS"
                            with m.Case(self.VENDOR_GET_USER_BUTTON):
                                m.next = "HANDLE_GET_USER_BUTTON"

                with m.State("HANDLE_SET_FPGA_LEDS"):
                    # take ownership of the interface
                    m.d.comb += interface.claim.eq(1)

                    # if we have an active data byte, set the FPGA LEDs to the payload
                    with m.If(interface.rx.valid & interface.rx.next):
                        m.d.usb += fpga_leds.eq(interface.rx.payload[0:6])

                    # once the receive is complete, respond with an ACK
                    with m.If(interface.rx_ready_for_response):
                       m.d.comb += interface.handshakes_out.ack.eq(1)

                    # finally, once we reach the status stage, send a ZLP
                    with m.If(interface.status_requested):
                        m.d.comb += self.send_zlp()
                        m.next = "IDLE"

                with m.State("HANDLE_GET_USER_BUTTON"):
                    # take ownership of the interface
                    m.d.comb += interface.claim.eq(1)

                    # write the state of the user button into a local data register
```

(continues on next page)

(continued from previous page)

```
                    data = Signal(8)
                    m.d.comb += data[0].eq(user_button)

                    # transmit our data using a built-in handler function that
                    # automatically advances the FSM back to the 'IDLE' state on
                    # completion
                    self.handle_simple_data_request(m, serializer, data)

        return m


class GatewareUSBDevice(Elaboratable):
    """ A simple USB device that can communicate with the host via vendor requests. """

    def create_standard_descriptors(self):
        """ Create the USB descriptors for the device. """

        descriptors = DeviceDescriptorCollection()

        # all USB devices have a single device descriptor
        with descriptors.DeviceDescriptor() as d:
            d.idVendor           = VENDOR_ID
            d.idProduct          = PRODUCT_ID
            d.iManufacturer      = "Cynthion Project"
            d.iProduct           = "Gateware USB Device"

            d.bNumConfigurations = 1

        # and at least one configuration descriptor
        with descriptors.ConfigurationDescriptor() as c:

            # with at least one interface descriptor
            with c.InterfaceDescriptor() as i:
                i.bInterfaceNumber = 0

                # interfaces also need endpoints to do anything useful
                # but we'll add those later!

        return descriptors

    def elaborate(self, platform):
        m = Module()

        # configure cynthion's clocks and reset signals
        m.submodules.car = platform.clock_domain_generator()

        # request the physical interface for cynthion's TARGET C port
        ulpi = platform.request("target_phy")

        # create the USB device
        m.submodules.usb = usb = USBDevice(bus=ulpi)

```

(continues on next page)

(continued from previous page)

```
        # create our standard descriptors and add them to the device's control endpoint
        descriptors = self.create_standard_descriptors()
        control_endpoint = usb.add_standard_control_endpoint(descriptors)

        # add microsoft os 1.0 descriptors and request handler
        descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
        msft_descriptors = MicrosoftOS10DescriptorCollection()
        with msft_descriptors.ExtendedCompatIDDescriptor() as c:
            with c.Function() as f:
                f.bFirstInterfaceNumber = 0
                f.compatibleID          = 'WINUSB'
        with msft_descriptors.ExtendedPropertiesDescriptor() as d:
            with d.Property() as p:
                p.dwPropertyDataType = RegistryTypes.REG_SZ
                p.PropertyName       = "DeviceInterfaceGUID"
                p.PropertyData       = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"
        msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
        control_endpoint.add_request_handler(msft_handler)

        # add the vendor request handler
        control_endpoint.add_request_handler(VendorRequestHandler())

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m


if __name__ == "__main__":
    from luna import top_level_cli
    top_level_cli(GatewareUSBDevice)
```

Listing 5: test-gateware-usb-device-03.py

```
import usb1
import time

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

VENDOR_SET_FPGA_LEDS   = 0x01
VENDOR_GET_USER_BUTTON = 0x02

# - list available usb devices ------------------------------------------------

def list_available_usb_devices(context):
    for device in context.getDeviceList():
        try:
            manufacturer = device.getManufacturer()
            product = device.getProduct()
            print(f"{device}:  {manufacturer} - {product}")
        except Exception as e:
```

(continues on next page)

(continued from previous page)

```
            print(f"{device}: {e}")


# - wrappers for control requests -------------------------------------------

def set_fpga_leds(device_handle, led_state):
    response = device_handle.controlWrite(
        request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
        request      = VENDOR_SET_FPGA_LEDS,
        index        = 0,
        value        = 0,
        data         = [led_state],
        timeout      = 1000,
    )

def get_user_button(device_handle):
    response = device_handle.controlRead(
        request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE | usb1.ENDPOINT_OUT,
        request      = VENDOR_GET_USER_BUTTON,
        index        = 0,
        value        = 0,
        length       = 1,
        timeout      = 1000,
    )
    return response[0]


# - test control endpoints --------------------------------------------------

def test_control_endpoints(device_handle):
    led_counter = 0
    last_button_state = False

    while True:
        # led counter
        set_fpga_leds(device_handle, led_counter)
        led_counter = (led_counter + 1) % 256

        # reset led counter when the USER button is pressed
        button_state = get_user_button(device_handle)
        if button_state:
            led_counter = 0

        # print button state when it changes
        if button_state != last_button_state:
            print(f"USER button is: {'ON' if button_state else 'OFF' }")
            last_button_state = button_state

        # slow the loop down so we can see the counter change
        time.sleep(0.1)


```

(continues on next page)

(continued from previous page)

```
# - main ---------------------------------------------------------------------

if __name__ == "__main__":
    with usb1.USBContext() as context:
        # list available devices
        list_available_usb_devices(context)

        # get a device handle to our simple usb device
        device_handle = context.openByVendorIDAndProductID(VENDOR_ID, PRODUCT_ID)
        if device_handle is None:
            raise Exception("Device not found.")

        # claim the device's interface
        device_handle.claimInterface(0)

        # pass the device handle to our control endpoint test
        test_control_endpoints(device_handle)
```

CHAPTER

FOURTEEN

USB GATEWARE: PART 4 - BULK TRANSFERS

This series of tutorial walks through the process of implementing a complete USB device with Cynthion and LUNA:

- *USB Gateware: Part 1 - Enumeration*
- *USB Gateware: Part 2 - WCID Descriptors*
- *USB Gateware: Part 3 - Control Transfers*
- *USB Gateware: Part 4 - Bulk Transfers (This tutorial)*

The goal of this tutorial is to define Bulk Endpoints for the device we created in Part 3 that will allow us to efficiently perform larger data transfers than those allowed by Control Transfers.

14.1 Prerequisites

- Complete the *USB Gateware: Part 1 - Enumeration* tutorial.
- Complete the *USB Gateware: Part 2 - WCID Descriptors* tutorial. *(Optional, required for Windows support)*
- Complete the *USB Gateware: Part 3 - Control Transfers* tutorial.

14.2 Add Bulk Endpoints

While Control transfers are well suited for command and status operations they are not the best way to exchange large quantities of data. Control transfers have high per-packet protocol overhead and can only transfer packets of 8 bytes on low speed (1.5Mbps) devices and 64 bytes on full (12Mbps) and high (512Mbps) speed devices.

On the other hand, Bulk transfers support a packet size of up to 512 bytes on high speed devices and do not require any protocol overhead.

In the first section we'll begin by updating our device's descriptors so it can inform the host that it has bulk endpoints available.

14.2.1 Update Device Descriptors

Open `gateware-usb-device.py` and add the highlighted lines:

Listing 1: gateware-usb-device.py

```
from amaranth                                    import *
from luna.usb2                                   import USBDevice
from usb_protocol.emitters                       import DeviceDescriptorCollection

from luna.gateware.usb.request.windows           import (
    MicrosoftOS10DescriptorCollection,
    MicrosoftOS10RequestHandler,
)
from usb_protocol.emitters.descriptors.standard  import get_string_descriptor
from usb_protocol.types.descriptors.microsoft10  import RegistryTypes

from luna.gateware.stream.generator              import StreamSerializer
from luna.gateware.usb.request.control           import ControlRequestHandler
from luna.gateware.usb.request.interface         import SetupPacket
from luna.gateware.usb.usb2.request              import RequestHandlerInterface
from luna.gateware.usb.usb2.transfer             import USBInStreamInterface
from usb_protocol.types                          import USBRequestType

from luna.usb2                                   import (
    USBStreamInEndpoint,
    USBStreamOutEndpoint,
)
from usb_protocol.types                          import (
    USBDirection,
    USBTransferType,
)

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

MAX_PACKET_SIZE = 512

class VendorRequestHandler(ControlRequestHandler):
    ...

class GatewareUSBDevice(Elaboratable):
    def create_standard_descriptors(self):
        descriptors = DeviceDescriptorCollection()

        with descriptors.DeviceDescriptor() as d:
            d.idVendor           = VENDOR_ID
            d.idProduct          = PRODUCT_ID
            d.iManufacturer      = "Cynthion Project"
            d.iProduct           = "Gateware USB Device"
            d.bNumConfigurations = 1

        with descriptors.ConfigurationDescriptor() as c:
```

(continues on next page)

(continued from previous page)

```
            with c.InterfaceDescriptor() as i:
                i.bInterfaceNumber = 0
                # EP 0x01 OUT - receives bulk data from the host
                with i.EndpointDescriptor() as e:
                    e.bEndpointAddress = USBDirection.OUT.to_endpoint_address(0x01)
                    e.bmAttributes     = USBTransferType.BULK
                    e.wMaxPacketSize   = MAX_PACKET_SIZE
                # EP 0x82 IN  - transmits bulk data to the host
                with i.EndpointDescriptor() as e:
                    e.bEndpointAddress = USBDirection.IN.to_endpoint_address(0x02)
                    e.bmAttributes     = USBTransferType.BULK
                    e.wMaxPacketSize   = MAX_PACKET_SIZE

        return descriptors

    def elaborate(self, platform):
        ...
```

This adds two endpoint descriptors to our default interface, each of type `USBTransferType.BULK` and with a `MAX_PACKET_SIZE` of 512. Where the endpoints differ is in their endpoint address. USB endpoint descriptors encode their direction in an 8 bit endpoint address. The first four bits encode the endpoint number, the next three bits are reserved and set to zero and the final bit encodes the direction; 0 for OUT and 1 for IN.

This means that an OUT endpoint number of 0x01 encodes to an endpoint address of 0x01 while an IN endpoint number of `0x02` encodes to the address `0x82`. (`0b0000_0010 + 0b1000_0000 = 0b1000_0010 = 0x82`)

14.2.2 Add USB Stream Endpoints

Once our endpoint descriptors have been added to our device configuration we will need some gateware that will be able to respond to USB requests from the host and allow us to receive and transmit data.

LUNA provides the `USBStreamOutEndpoint` and `USBStreamInEndpoint` components which conform to the Amaranth Data streams interface. Simply put, streams provide a uniform mechanism for unidirectional exchange of arbitrary data between gateware components.

Listing 2: gateware-usb-device.py

```
...

class GatewareUSBDevice(Elaboratable):
    def create_standard_descriptors(self):
        ...

    def elaborate(self, platform):
        ...

        # add the vendor request handler
        control_endpoint.add_request_handler(VendorRequestHandler())

        # create and add stream endpoints for our device's Bulk IN & OUT endpoints
        ep_out = USBStreamOutEndpoint(
            endpoint_number=0x01,  # (EP 0x01)
```

(continues on next page)

(continued from previous page)

```
            max_packet_size=MAX_PACKET_SIZE,
        )
        usb.add_endpoint(ep_out)
        ep_in = USBStreamInEndpoint(
            endpoint_number=0x02,  # (EP 0x82)
            max_packet_size=MAX_PACKET_SIZE
        )
        usb.add_endpoint(ep_in)

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m
```

We now have two streaming endpoints that are able to receive and transmit data between any other module that supports the Amaranth Data streams interface.

However, before we can stream any data across these endpoints we first need to come up with a *USB Function* for each of our endpoints. In other words, what does our device actually _do_?

This could be any data source and/or sink but for the purposes of this tutorial let's create a simple loopback function that will accept a bulk OUT request from the host and then return the request payload when the host makes a bulk IN request.

14.2.3 Define Endpoint Functions

A simple implementation for our device's endpoint functions could be a simple FIFO (First In First Out) queue with enough space to hold the 512 bytes of a bulk transfer.

Using the OUT endpoint we could then transmit a stream of data from the host to Cynthion and write it into the FIFO. Then, when we transmit a request from the host to the IN endpoint we can stream the previously queued data back to the host.

We're only working in a single clock-domain so we can use a SyncFIFO from the Amaranth standard library for our queue:

Listing 3: gateware-usb-device.py

```
from amaranth                                import *
from amaranth.lib.fifo                       import SyncFIFO
from luna.usb2                               import USBDevice
from usb_protocol.emitters                   import DeviceDescriptorCollection
...

class VendorRequestHandler(ControlRequestHandler):
    ...

class GatewareUSBDevice(Elaboratable):
    ...

    def elaborate(self, platform):
        ...

```

(continues on next page)

(continued from previous page)

```
        # create and add stream endpoints for our device's Bulk IN & OUT endpoints
        ep_out = USBStreamOutEndpoint(
            endpoint_number=0x01,  # (EP 0x01)
            max_packet_size=MAX_PACKET_SIZE,
        )
        usb.add_endpoint(ep_out)
        ep_in = USBStreamInEndpoint(
            endpoint_number=0x02,  # (EP 0x82)
            max_packet_size=MAX_PACKET_SIZE
        )
        usb.add_endpoint(ep_in)

        # create a FIFO queue we'll connect to the stream interfaces of our
        # IN & OUT endpoints
        m.submodules.fifo = fifo = DomainRenamer("usb")(
            SyncFIFO(width=8, depth=MAX_PACKET_SIZE)
        )

        # connect our Bulk OUT endpoint's stream interface to the FIFO's write port
        stream_out = ep_out.stream
        m.d.comb += fifo.w_data.eq(stream_out.payload)
        m.d.comb += fifo.w_en.eq(stream_out.valid)
        m.d.comb += stream_out.ready.eq(fifo.w_rdy)

        # connect our Bulk IN endpoint's stream interface to the FIFO's read port
        stream_in  = ep_in.stream
        m.d.comb += stream_in.payload.eq(fifo.r_data)
        m.d.comb += stream_in.valid.eq(fifo.r_rdy)
        m.d.comb += fifo.r_en.eq(stream_in.ready)

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m
```

Note: Something to take note off is the use of an Amaranth DomainRenamer component to wrap `SyncFIFO` in the following lines:

```
m.submodules.fifo = fifo = DomainRenamer("usb")(
    fifo.SyncFIFO(width=8, depth=MAX_PACKET_SIZE)
)
```

Any moderately complex FPGA hardware & gateware design will usually consist of multiple clock-domains running at different frequencies. Cynthion, for example, has three clock domains:

- `sync` - the default clock domain, running at 120 MHz.
- `usb` - the clock domain for USB components and gateware, running at 60 MHz.
- `fast` - a fast clock domain used for the HyperRAM, running at 240 MHz.

Because our designs so far have all been interfacing with Cynthion's USB components we've only needed to use the `usb` clock domain. However, reusable Amaranth components such as `SyncFIFO` are usually implemented using the default `sync` domain. We therefore need to be able to rename its clock domain to match the domain used in our design.

This is what `DomainRenamer` does.

And that's it, we've defined our endpoint functions! Let's try it out.

14.3 Test Bulk Endpoints

Open up `test-gateware-usb-device.py` and add the following code to it:

Listing 4: test-gateware-usb-device.py

```
import usb1
import time
import random

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

VENDOR_SET_FPGA_LEDS   = 0x01
VENDOR_GET_USER_BUTTON = 0x02

MAX_PACKET_SIZE = 512

# - list available usb devices ------------------------------------------------

def list_available_usb_devices(context):
    for device in context.getDeviceList():
        try:
            manufacturer = device.getManufacturer()
            product = device.getProduct()
            print(f"{device}:  {manufacturer} - {product}")
        except Exception as e:
            print(f"{device}: {e}")


# - wrappers for control requests ---------------------------------------------

def set_fpga_leds(device_handle, led_state):
    response = device_handle.controlWrite(
        request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
        request      = VENDOR_SET_FPGA_LEDS,
        index        = 0,
        value        = 0,
        data         = [led_state],
        timeout      = 1000,
    )

def get_user_button(device_handle):
    response = device_handle.controlRead(
        request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE | usb1.ENDPOINT_OUT,
        request      = VENDOR_GET_USER_BUTTON,
        index        = 0,
```

(continues on next page)

(continued from previous page)

```
        value        = 0,
        length       = 1,
        timeout      = 1000,
    )
    return response[0]


# - test control endpoints ----------------------------------------------------

def test_control_endpoints(device_handle):
    led_counter = 0
    last_button_state = False

    while True:
        # led counter
        set_fpga_leds(device_handle, led_counter)
        led_counter = (led_counter + 1) % 256

        # reset led counter when the USER button is pressed
        button_state = get_user_button(device_handle)
        if button_state:
            led_counter = 0

        # print button state when it changes
        if button_state != last_button_state:
            print(f"USER button is: {'ON' if button_state else 'OFF' }")
            last_button_state = button_state

        # slow the loop down so we can see the counter change
        time.sleep(0.1)


# - wrappers for bulk requests ------------------------------------------------

def bulk_out_transfer(device_handle, data):
    response = device_handle.bulkWrite(
        endpoint = 0x01,
        data     = data,
        timeout  = 1000,
    )
    return response

def bulk_in_transfer(device_handle, length):
    response = device_handle.bulkRead(
        endpoint = 0x02,
        length   = length,
        timeout  = 1000,
    )
    return response


# - test bulk endpoints -------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```python

def test_bulk_endpoints(device_handle):
    # bulk_out - write a list of random numbers to memory
    data = list([random.randint(0, 255) for _ in range(MAX_PACKET_SIZE)])
    response = bulk_out_transfer(device_handle, data)
    print(f"OUT endpoint transmitted {response} bytes: {data[0:4]} ... {data[-4:]}")

    # bulk_in - retrieve the contents of our memory
    response = list(bulk_in_transfer(device_handle, MAX_PACKET_SIZE))
    print(f"IN  endpoint received {len(response)} bytes:    {response[0:4]} ...
↪{response[-4:]}")

    # check that the stored data matches the sent data
    assert(data == list(response))


# - main ---------------------------------------------------------------

if __name__ == "__main__":
    with usb1.USBContext() as context:
        # list available devices
        list_available_usb_devices(context)

        # get a device handle to our simple usb device
        device_handle = context.openByVendorIDAndProductID(VENDOR_ID, PRODUCT_ID)
        if device_handle is None:
            raise Exception("Device not found.")

        # claim the device's interface
        device_handle.claimInterface(0)

        # pass the device handle to our bulk endpoint test
        test_bulk_endpoints(device_handle)

        # pass the device handle to our control endpoint test
        test_control_endpoints(device_handle)
```

Run the file with:

```
python ./test-gateware-usb-device.py
```

Assuming everything is going to plan you should see two matching sets of random numbers:

```
OUT endpoint transmitted 512 bytes: [252, 107, 106, 56] ... [109, 175, 112, 126]
IN  endpoint received 512 bytes:    [252, 107, 106, 56] ... [109, 175, 112, 126]
```

Congratulations, if you made it this far then you've just finished building your first complete USB Gateware Device with custom vendor request control and bulk data transfer!

14.4 Exercises

1. Create a benchmark to test the speed of your device when doing Bulk IN and OUT transfers.
2. Move the device endpoint to `aux_phy` and attempt to capture the packets exchanged between a device plugged into a host via the `target_phy` port.

14.5 More information

- Beyond Logic's USB in a NutShell.
- LUNA Documentation

14.6 Source Code

Listing 5: gateware-usb-device-04.py

```
#!/usr/bin/env python3
#
# This file is part of Cynthion.
#
# Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
# SPDX-License-Identifier: BSD-3-Clause

from amaranth                                       import *
from amaranth.lib.fifo                              import SyncFIFO
from luna.usb2                                      import USBDevice
from usb_protocol.emitters                          import DeviceDescriptorCollection

from luna.gateware.usb.request.windows              import (
    MicrosoftOS10DescriptorCollection,
    MicrosoftOS10RequestHandler,
)
from usb_protocol.emitters.descriptors.standard     import get_string_descriptor
from usb_protocol.types.descriptors.microsoft10     import RegistryTypes

from luna.gateware.stream.generator                 import StreamSerializer
from luna.gateware.usb.request.control              import ControlRequestHandler
from luna.gateware.usb.request.interface            import SetupPacket
from luna.gateware.usb.usb2.request                 import RequestHandlerInterface
from luna.gateware.usb.usb2.transfer                import USBInStreamInterface
from usb_protocol.types                             import USBRequestType

from luna.usb2                                      import USBStreamInEndpoint,␣
↪USBStreamOutEndpoint
from usb_protocol.types                             import USBDirection, USBTransferType

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

```

(continues on next page)

(continued from previous page)

```
MAX_PACKET_SIZE = 512

class VendorRequestHandler(ControlRequestHandler):
    VENDOR_SET_FPGA_LEDS   = 0x01
    VENDOR_GET_USER_BUTTON = 0x02

    def elaborate(self, platform):
        m = Module()

        # shortcuts
        interface: RequestHandlerInterface = self.interface
        setup: SetupPacket = self.interface.setup

        # get a reference to the FPGA LEDs and USER button
        fpga_leds   = Cat(platform.request("led", i).o for i in range(6))
        user_button = platform.request("button_user").i

        # create a streamserializer for transmitting IN data back to the host
        serializer = StreamSerializer(
            domain           = "usb",
            stream_type      = USBInStreamInterface,
            data_length      = 1,
            max_length_width = 1,
        )
        m.submodules += serializer

        # we've received a setup packet containing a vendor request.
        with m.If(setup.type == USBRequestType.VENDOR):
            # use a state machine to sequence our request handling
            with m.FSM(domain="usb"):
                with m.State("IDLE"):
                    with m.If(setup.received):
                        with m.Switch(setup.request):
                            with m.Case(self.VENDOR_SET_FPGA_LEDS):
                                m.next = "HANDLE_SET_FPGA_LEDS"
                            with m.Case(self.VENDOR_GET_USER_BUTTON):
                                m.next = "HANDLE_GET_USER_BUTTON"

                with m.State("HANDLE_SET_FPGA_LEDS"):
                    # take ownership of the interface
                    m.d.comb += interface.claim.eq(1)

                    # if we have an active data byte, set the FPGA LEDs to the payload
                    with m.If(interface.rx.valid & interface.rx.next):
                        m.d.usb += fpga_leds.eq(interface.rx.payload[0:6])

                    # once the receive is complete, respond with an ACK
                    with m.If(interface.rx_ready_for_response):
                        m.d.comb += interface.handshakes_out.ack.eq(1)

                    # finally, once we reach the status stage, send a ZLP
                    with m.If(interface.status_requested):
```

(continues on next page)

(continued from previous page)

```
                        m.d.comb += self.send_zlp()
                        m.next = "IDLE"

                with m.State("HANDLE_GET_USER_BUTTON"):
                    # take ownership of the interface
                    m.d.comb += interface.claim.eq(1)

                    # write the state of the user button into a local data register
                    data = Signal(8)
                    m.d.comb += data[0].eq(user_button)

                    # transmit our data using a built-in handler function that
                    # automatically advances the FSM back to the 'IDLE' state on
                    # completion
                    self.handle_simple_data_request(m, serializer, data)

        return m


class GatewareUSBDevice(Elaboratable):
    """ A simple USB device that can communicate with the host via vendor and bulk␣
↪requests. """

    def create_standard_descriptors(self):
        """ Create the USB descriptors for the device. """

        descriptors = DeviceDescriptorCollection()

        # all USB devices have a single device descriptor
        with descriptors.DeviceDescriptor() as d:
            d.idVendor           = VENDOR_ID
            d.idProduct          = PRODUCT_ID
            d.iManufacturer      = "Cynthion Project"
            d.iProduct           = "Gateware USB Device"

            d.bNumConfigurations = 1

        # and at least one configuration descriptor
        with descriptors.ConfigurationDescriptor() as c:

            # with at least one interface descriptor
            with c.InterfaceDescriptor() as i:
                i.bInterfaceNumber = 0

                # an endpoint for receiving bulk data from the host
                with i.EndpointDescriptor() as e:
                    e.bEndpointAddress = USBDirection.OUT.to_endpoint_address(0x01) # EP␣
↪0x01 OUT
                    e.bmAttributes     = USBTransferType.BULK
                    e.wMaxPacketSize   = MAX_PACKET_SIZE

                # and an endpoint for transmitting bulk data to the host
```

(continues on next page)

(continued from previous page)

```
                with i.EndpointDescriptor() as e:
                    e.bEndpointAddress = USBDirection.IN.to_endpoint_address(0x02)  # EP␣
↪0x82 IN
                    e.bmAttributes     = USBTransferType.BULK
                    e.wMaxPacketSize   = MAX_PACKET_SIZE

        return descriptors

    def elaborate(self, platform):
        m = Module()

        # configure cynthion's clocks and reset signals
        m.submodules.car = platform.clock_domain_generator()

        # request the physical interface for cynthion's TARGET C port
        ulpi = platform.request("target_phy")

        # create the USB device
        m.submodules.usb = usb = USBDevice(bus=ulpi)

        # create our standard descriptors and add them to the device's control endpoint
        descriptors = self.create_standard_descriptors()
        control_endpoint = usb.add_standard_control_endpoint(descriptors)

        # add microsoft os 1.0 descriptors and request handler
        descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
        msft_descriptors = MicrosoftOS10DescriptorCollection()
        with msft_descriptors.ExtendedCompatIDDescriptor() as c:
            with c.Function() as f:
                f.bFirstInterfaceNumber = 0
                f.compatibleID          = 'WINUSB'
        with msft_descriptors.ExtendedPropertiesDescriptor() as d:
            with d.Property() as p:
                p.dwPropertyDataType = RegistryTypes.REG_SZ
                p.PropertyName       = "DeviceInterfaceGUID"
                p.PropertyData       = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"
        msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
        control_endpoint.add_request_handler(msft_handler)

        # add the vendor request handler
        control_endpoint.add_request_handler(VendorRequestHandler())

        # create and add stream endpoints for our device's Bulk IN & OUT endpoints
        ep_out = USBStreamOutEndpoint(
            endpoint_number=0x01,  # (EP 0x01)
            max_packet_size=MAX_PACKET_SIZE,
        )
        usb.add_endpoint(ep_out)
        ep_in = USBStreamInEndpoint(
            endpoint_number=0x02,  # (EP 0x82)
            max_packet_size=MAX_PACKET_SIZE
        )
```

(continues on next page)

(continued from previous page)

```
        usb.add_endpoint(ep_in)

        # create a FIFO queue we'll connect to the stream interfaces of our
        # IN & OUT endpoints
        m.submodules.fifo = fifo = DomainRenamer("usb")(
            SyncFIFO(width=8, depth=MAX_PACKET_SIZE)
        )

        # connect our Bulk OUT endpoint's stream interface to the FIFO's write port
        stream_out = ep_out.stream
        m.d.comb += fifo.w_data.eq(stream_out.payload)
        m.d.comb += fifo.w_en.eq(stream_out.valid)
        m.d.comb += stream_out.ready.eq(fifo.w_rdy)

        # connect our Bulk IN endpoint's stream interface to the FIFO's read port
        stream_in  = ep_in.stream
        m.d.comb += stream_in.payload.eq(fifo.r_data)
        m.d.comb += stream_in.valid.eq(fifo.r_rdy)
        m.d.comb += fifo.r_en.eq(stream_in.ready)

        # configure the device to connect by default when plugged into a host
        m.d.comb += usb.connect.eq(1)

        return m


if __name__ == "__main__":
    from luna import top_level_cli
    top_level_cli(GatewareUSBDevice)
```

Listing 6: test-gateware-usb-device-04.py

```
import usb1
import time
import random

VENDOR_ID  = 0x1209 # https://pid.codes/1209/
PRODUCT_ID = 0x0001

VENDOR_SET_FPGA_LEDS   = 0x01
VENDOR_GET_USER_BUTTON = 0x02

MAX_PACKET_SIZE = 512

# - list available usb devices -----------------------------------------------

def list_available_usb_devices(context):
    for device in context.getDeviceList():
        try:
            manufacturer = device.getManufacturer()
            product = device.getProduct()
            print(f"{device}:  {manufacturer} - {product}")
```

(continues on next page)

(continued from previous page)

```
        except Exception as e:
            print(f"{device}: {e}")


# - wrappers for control requests ---------------------------------------------

def set_fpga_leds(device_handle, led_state):
    response = device_handle.controlWrite(
        request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
        request      = VENDOR_SET_FPGA_LEDS,
        index        = 0,
        value        = 0,
        data         = [led_state],
        timeout      = 1000,
    )

def get_user_button(device_handle):
    response = device_handle.controlRead(
        request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE | usb1.ENDPOINT_OUT,
        request      = VENDOR_GET_USER_BUTTON,
        index        = 0,
        value        = 0,
        length       = 1,
        timeout      = 1000,
    )
    return response[0]


# - test control endpoints ----------------------------------------------------

def test_control_endpoints(device_handle):
    led_counter = 0
    last_button_state = False

    while True:
        # led counter
        set_fpga_leds(device_handle, led_counter)
        led_counter = (led_counter + 1) % 256

        # reset led counter when the USER button is pressed
        button_state = get_user_button(device_handle)
        if button_state:
            led_counter = 0

        # print button state when it changes
        if button_state != last_button_state:
            print(f"USER button is: {'ON' if button_state else 'OFF' }")
            last_button_state = button_state

        # slow the loop down so we can see the counter change
        time.sleep(0.1)

```

(continues on next page)

(continued from previous page)

```python

# - wrappers for bulk requests ------------------------------------------------

def bulk_out_transfer(device_handle, data):
    response = device_handle.bulkWrite(
        endpoint = 0x01,
        data     = data,
        timeout  = 1000,
    )
    return response

def bulk_in_transfer(device_handle, length):
    response = device_handle.bulkRead(
        endpoint = 0x02,
        length   = length,
        timeout  = 1000,
    )
    return response


# - test bulk endpoints -------------------------------------------------------

def test_bulk_endpoints(device_handle):
    # bulk_out - write a list of random numbers to memory
    data = list([random.randint(0, 255) for _ in range(MAX_PACKET_SIZE)])
    response = bulk_out_transfer(device_handle, data)
    print(f"OUT endpoint transmitted {response} bytes: {data[0:4]} ... {data[-4:]}")

    # bulk_in - retrieve the contents of our memory
    response = list(bulk_in_transfer(device_handle, MAX_PACKET_SIZE))
    print(f"IN  endpoint received {len(response)} bytes:    {response[0:4]} ...
↪{response[-4:]}")

    # check that the stored data matches the sent data
    assert(data == list(response))


# - main ----------------------------------------------------------------------

if __name__ == "__main__":
    with usb1.USBContext() as context:
        # list available devices
        list_available_usb_devices(context)

        # get a device handle to our simple usb device
        device_handle = context.openByVendorIDAndProductID(VENDOR_ID, PRODUCT_ID)
        if device_handle is None:
            raise Exception("Device not found.")

        # claim the device's interface
        device_handle.claimInterface(0)

```

(continues on next page)

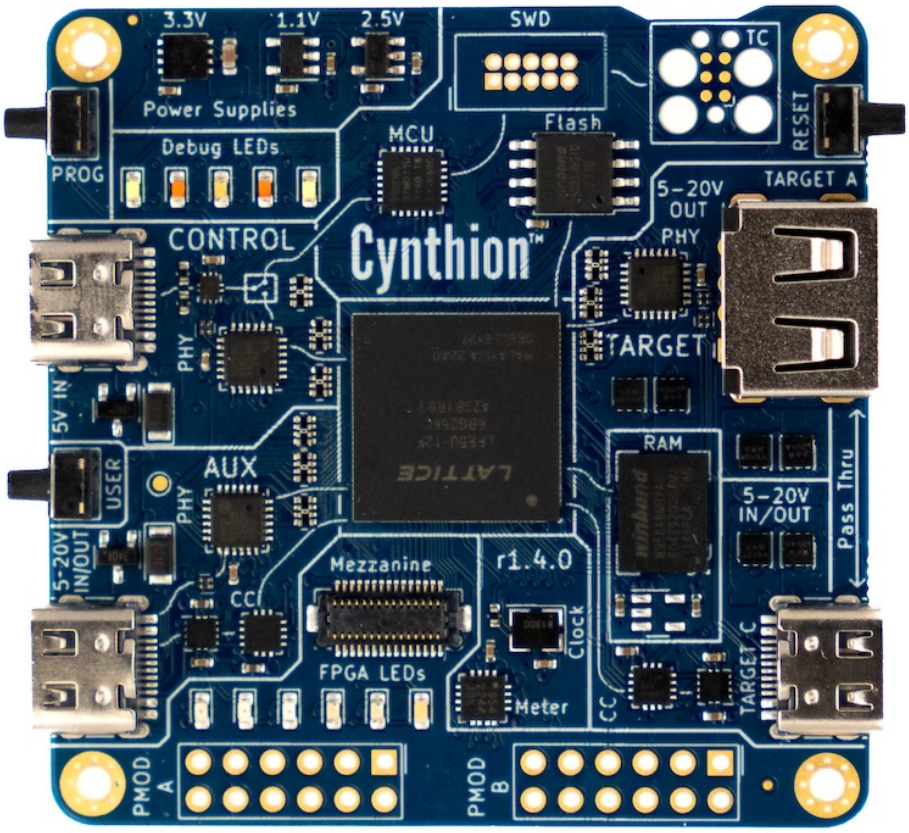
(continued from previous page)

```
        # pass the device handle to our bulk endpoint test
        test_bulk_endpoints(device_handle)

        # pass the device handle to our control endpoint test
        test_control_endpoints(device_handle)
```

CHAPTER

FIFTEEN

INTRODUCTION



15.1 Cynthion Hardware

CHAPTER

SIXTEEN

DEVICE OVERVIEW

16.1 Top View

- **A-E** - Five status LEDs managed by the debug microcontroller.
  - **A** - Power Indicator.
  - **B** - FPGA is online.
  - **C** - FPGA has requested control of the **CONTROL** port.
  - **D** - FPGA has control of the **CONTROL** port.
  - **E** - Reserved for future use.
- **0-5** - Six USER LEDs managed by the FPGA.

16.2 Left View

- **PROGRAM** - Press this button to return control of the **CONTROL** port to the debug microcontroller and hold the FPGA in an unconfigured state.
  - *Recovery mode*: Press this button during power-on to invoke the Saturn-V bootloader on the **CONTROL** port.
- **CONTROL** - Primary USB connector used by the host computer to control Cynthion.
- **USER** - A user-assignable button that can be used in your own designs.
- **AUX** - An auxiliary USB connection that can be used in your own designs.

16.3 Right View

- **TARGET C** - USB Type-C connector for Packetry traffic capture and Facedancer device emulation.
- **TARGET A** - USB Type-A connector shared with the **TARGET C** connector.
- **RESET** - Press this button to reset Cynthion's debug microcontroller and reconfigure the FPGA from flash.

16.4 Front View

- **A & B** - Two Digilent Pmod™ Compatible I/O connectors for a total of 16 high-speed FPGA user IOs.
  - **B** can also be configured to act as a serial port and JTAG connector for debugging SoC designs:
    - **1** - SERIAL RX
    - **2** - SERIAL TX
    - **7** - JTAG TMS
    - **8** - JTAG TDI
    - **9** - JTAG TDO
    - **10** - JTAG TCK

16.5 Bottom View

CHAPTER

SEVENTEEN

SELF-MADE HARDWARE BRINGUP

This guide is intended to help you bring up a Cynthion board you've built yourself. If you've received your board from Great Scott Gadgets, it should already be set up, and you shouldn't need to follow these steps.

17.1 Prerequisites

- A Cynthion board with a populated *Debug Controller* microprocessor. This is the SAMD microcontroller located in the Debug section at the bottom of the board. When powering the board, the test points should have the marked voltages. The FPGA LEDs might be dimly lit.
- A programmer capable of uploading firmware via SWD. Examples include the Black Magic Probe; the Segger J-Link, and many OpenOCD compatible boards.
- A toolchain capable of building binaries for Cortex-M0 processors, such as the GNU Arm Embedded toolchain. If you're using Linux or macOS, you'll likely want to fetch this using a package manager; a suitable toolchain may be called something like `arm-none-eabi-gcc`.
- A DFU programming utility, such as dfu-util.

17.2 Bring-up Process

The high-level process for bringing up your board is as follows:

1. Compile and upload the Saturn-V bootloader, which allows Debug Controller to program itself.
2. Compile and upload the Apollo Debug Controller firmware, which allows FPGA configuration & flashing; and provides debug interfaces for working with the FPGA.
3. Install the `cynthion` tools, and run through the self-test procedures to validate that your board is working.

17.3 Build/upload Saturn-V

The "recovery mode (RVM)" bootloader for Cynthion boards is named *Saturn-V*; as it's the first stage in "getting to Cynthion". The bootloader is located in in its own repository

You can clone the bootloader using `git`:

```
$ git clone https://github.com/greatscottgadgets/saturn-v
```

Build the DFU bootloader by invoking `make`. An example invocation for modern Cynthion hardware might look like:

```
$ cd saturn-v
$ make
```

If you're building a board that predates r0.3 hardware, you'll need to specify the board you're building for:

```
$ cd saturn-v
$ make BOARD=luna_d21
```

The build should yield two useful build products: `bootloader.elf` and `bootloader.bin`; your SWD programmer will likely consume one of these two files.

Next, connect your SWD programmer to the header labeled `uC`, and upload bootloader image. You can use both the ports labelled `Sideband` and `Main Host` to power the board in this process. If you're using the Black Magic Probe, this might look like:

```
$ arm-none-eabi-gdb -nx --batch \
    -ex 'target extended-remote /dev/ttyACM0' \
    -ex 'monitor swdp_scan' \
    -ex 'attach 1' \
    -ex 'load' \
    -ex 'kill' \
    bootloader.elf
```

If you are using openocd, the process might look similar to the following (add the configuration file for your SWD adapter:

```
$ openocd -f openocd/scripts/target/at91samdXX.cfg
Open On-Chip Debugger 0.11.0-rc2
Licensed under GNU GPL v2
Info : Listening on port 4444 for telnet connections
Info : clock speed 400 kHz
Info : SWD DPIDR 0x0bc11477
Info : at91samd.cpu: hardware has 4 breakpoints, 2 watchpoints
Info : at91samd.cpu: external reset detected
```

```
$ nc localhost 4444
Open On-Chip Debugger
> targets
    TargetName         Type       Endian TapName            State
--  ------------------ ---------- ------ ------------------ ------------
0* at91samd.cpu       cortex_m   little at91samd.cpu       reset

> at91samd chip-erase
chip erase started

> program Luna/saturn-v/bootloader.bin verify reset
target halted due to debug-request, current mode: Thread
xPSR: 0xf1000000 pc: 0xfffffffe msp: 0xfffffffc
** Programming Started **
SAMD MCU: SAMD21G18A (256KB Flash, 32KB RAM)
** Programming Finished **
** Verify Started **
** Verified OK **
** Resetting Target **
```

If your programmer works best with `.bin` files, be sure to upload the `bootloader.bin` to the start of flash (address `0x00000000`).

Once the bootloader is installed, you should see LED `A` blinking rapidly. This is the indication that your board is in Recovery Mode (RVM), and can be programmed via DFU.

You can verify that the board is DFU-programmable by running `dfu-util` while connected to the USB port labelled `Sideband`:

```
$ dfu-util --list
dfu-util 0.9

Copyright 2005-2009 Weston Schmidt, Harald Welte and OpenMoko Inc.
Copyright 2010-2016 Tormod Volden and Stefan Schmidt
This program is Free Software and has ABSOLUTELY NO WARRANTY
Please report bugs to http://sourceforge.net/p/dfu-util/tickets/

Found DFU: [1d50:615c] ver=0000, devnum=22, cfg=1, intf=0, path="2-3.3.1.2", alt=1, name=
↪"SRAM"
Found DFU: [1d50:615c] ver=0000, devnum=22, cfg=1, intf=0, path="2-3.3.1.2", alt=0, name=
↪"Flash"
```

If your device shows up as a Cynthion board, congratulations! You're ready to move on to the next step.

17.3.1 Optional: Bootloader Locking

Optionally, you can reversibly lock the bootloader region of the Debug Controller, preventing you from accidentally overwriting the bootloader. This is most useful for users developing code for the Debug Controller.

If you choose to lock the bootloader, you should lock the first `2KiB` of flash. Note that currently, the bootloader lock feature of *Black Magic Probe* devices always locks `8KiB` of flash; and thus cannot be used for Cynthion.

17.4 Build/upload Apollo

The next bringup step is to upload the *Apollo* Debug Controller firmware, which will provide an easy way to interface with the board's FPGA and any gateware running on it. The Apollo source is located in its own repository.

You can clone the bootloader using *git*:

```
$ git clone --recurse-submodules https://github.com/greatscottgadgets/apollo
```

You can build and run the firmware in one step by invoking `make`. In order to ensure your firmware matches the hardware it's running on, you'll need to provide board type and hardware revision using the `APOLLO_BOARD`, `BOARD_REVISION_MAJOR` and `BOARD_REVISION_MINOR` make variables.

The board's hardware revision is printed on its silkscreen in a `r(MAJOR).(MINOR)` format. Board `r1.4` would have a `BOARD_REVISION_MAJOR=1` and a `BOARD_REVISION_MINOR=4`. If your board's revision ends in a +, do not include it in the revision number.

An example invocation for a `r1.4` board might be:

```
$ make APOLLO_BOARD=cynthion BOARD_REVISION_MAJOR=1 BOARD_REVISION_MINOR=4 dfu
```

Once programming is complete, LED's `A`, `B`, `C`, `D` and `E` should all be on; indicating that the Apollo firmware is idle.

You can also upload a firmware binary using *dfu-util* with:

```
$ dfu-util -d 1d50:615c -D firmware.bin
```

17.5 Running Self-Tests

The final step of bringup is to validate the functionality of your hardware. This is most easily accomplished by running Cynthion's interactive self-test applet.

Before you can run the applet, you'll need to have a working `cynthion` development environment. See *Introduction* to get your environment set up.

Next, we can check to make sure your Cynthion board is recognized by the Cynthion toolchain. Running the `apollo info` command will list any detected devices:

```
$ apollo info
Detected a Cynthion device!
    Hardware: Cynthion r1.4
    Serial number: <snip>
```

Once you've validated connectivity, you're ready to try running the `cynthion selftest` command.

```
$ cynthion selftest

 INFO    | __init__    | Building and uploading gateware to attached Cynthion r1.4...
 INFO    | __init__    | Upload complete.
 INFO    | selftest    | Connected to onboard debugger; hardware revision r1.4 (s/n:
↪<snip>).
 INFO    | selftest    | Running tests...

 Automated tests:
         Debug module:   ✓ PASSED
         AUX PHY:        ✓ PASSED
         HyperRAM:       ✓ PASSED
         CONTROL PHY:    ✓ PASSED
         TARGET PHY:     ✓ PASSED
```

17.6 Troubleshooting

Issue: some of the build files weren't found; `make` produces a message like " `no rule to make target` ".

Chances are, your clone of Apollo was pulled down without its submodules. You can pull down the relevant submodules using `git`:

```
$ git submodule update --init --recursive
```

Issue: the ``apollo info`` command doesn't see a connected board.

On Linux, this can be caused by a permissions issue. Check first for the presence of your device using `lsusb`; if you see a device with the VID/PID `1d50:615c`, your board is present – and you likely have a permissions issue. You'll likely need to install permission-granting udev rules.

CHAPTER

EIGHTEEN

THE APOLLO COMMAND LINE UTILITY

The cynthion distribution provides the apollo command-line utility, that can be used to perform various simple functions useful in development; including simple JTAG operations, SVF playback, manipulating the board's flash, and debug communications.

```
$ apollo
usage: apollo [-h] command ...

Apollo FPGA Configuration / Debug tool

positional arguments:
  command
    info                Print device info.
    jtag-scan           Prints information about devices on the onboard JTAG chain.
    flash-info          Prints information about the FPGA's attached configuration
                        flash.
    flash-erase         Erases the contents of the FPGA's flash memory.
    flash-program       Programs the target bitstream onto the FPGA's configuration
                        flash.
    flash-fast          Programs a bitstream onto the FPGA's configuration flash using
                        a SPI bridge.
    flash-read          Reads the contents of the attached FPGA's configuration flash.
    svf                 Plays a given SVF file over JTAG.
    configure           Uploads a bitstream to the device's FPGA over JTAG.
    reconfigure         Requests the attached ECP5 reconfigure itself from its SPI flash.
    force-offline       Forces the board's FPGA offline.
    spi                 Sends the given list of bytes over debug-SPI, and returns the
                        response.
    spi-inv             Sends the given list of bytes over SPI with inverted CS.
    spi-reg             Reads or writes to a provided register over the debug-SPI.
    jtag-spi            Sends the given list of bytes over SPI-over-JTAG, and returns the
                        response.
    jtag-reg            Reads or writes to a provided register of JTAG-tunneled debug␣
↪SPI.
    leds                Sets the specified pattern for the Debug LEDs.

optional arguments:
  -h, --help            show this help message and exit
```

CHAPTER

NINETEEN

GETTING HELP

Before asking for help with Cynthion, check to see if your question is answered in this documentation, or addressed in the Cynthion GitHub repository issues.

For assistance with Cynthion general use or development, please look at the issues on the GitHub project. This is the preferred place to ask questions so that others may locate the answer to your question in the future.

We invite you to join our community discussions on Discord. Note that while technical support requests are welcome here, we do not have support staff on duty at all times. Be sure to also submit an issue on GitHub if you've found a bug or if you want to ensure that your request will be tracked and not overlooked.

CHAPTER

TWENTY

CYNTHION PROJECTS AND MENTIONS

Have you done something cool with Cynthion or mentioned Cynthion in one of your papers or presentations? Email us at info@greatscottgadgets.com with a link to what you've done and we might post it here!

CHAPTER

TWENTYONE

SAFETY INFORMATION

21.1 Warnings

- This product shall only be connected to an external power supply rated at 5 V DC. Any power supply used with this product shall comply with relevant regulations and standards applicable in the country of intended use.
- This product should be used in a shielded enclosure. Removing this product from its enclosure or using this product without a shielded enclosure may expose it to additional risks of damage. Extra care should be taken to prevent damage from electrostatic discharge and impact damage when this product is removed from its enclosure or used without a shielded enclosure.
- This product should only be connected to another device that is rated to be used at the power level this device is configured to provide. Attaching this product to another device that is not rated to be used at the power level that this device is configured to provide may result in damage.

21.2 Instructions For Safe Use

- Do not expose this product to water or moisture.
- Do not expose this product to excessive heat while in use.
- Do not place undue stress on any connector port.
- Any equipment connected to Cynthion should comply with relevant standards for the country of use and be marked accordingly to ensure that safety and performance requirements are met.
- Do not allow equipment connected to Cynthion to make contact with any internal component of Cynthion other than a connector.
- A pass-through power supply connected to Cynthion must supply at least 5 V DC and no more than 20 V DC. The pass-through power supply must be rated for the current drawn by the pass-through device connected to Cynthion.
- A device connected to Cynthion must be rated for 5 V DC or the pass-through power supply voltage and must draw no more than 3 A current from the pass-through power supply or 500 mA current from Cynthion's power supply.
- Use USB cables of no more than 2 m length with Cynthion.
- Do not expose Pmod or mezzanine connectors to voltages greater than 3.3 V.

CHAPTER

TWENTYTWO

INTRODUCTION

22.1 Setting up a Development Environment

This guide highlights the installation and setup process for setting up a local copy of the Cynthion source code for development.

22.2 Prerequisites

- Python v3.9, or later.
- A working FPGA toolchain. We only officially support a toolchain composed of the Project Trellis ECP5 tools, the yosys synthesis suite, and the NextPNR place-and-route tool. You can obtain the latest binary distribution of this software from the oss-cad-suite-build project.
- A working Rust development environment if you want to develop firmware for Cynthion's SoC bitstream.
- A RISC-V GNU Compiler Toolchain if you want to use `gdb` for SoC firmware debugging over JTAG.

22.3 Installation

For development you'll need a local copy of the Cynthion repository:

Use git to clone the repository:

```
git clone https://github.com/greatscottgadgets/cynthion.git
```

Please perform the following steps to enable support for symlinks before attempting to clone the repository on Windows:

1. Open the *"For developers"* page in *System settings* and enable Developer Mode.
2. Restart your computer.
3. Open the Group Policy editor: *gpedit.msc*
4. Navigate to *Computer Configuration* → *Windows Settings* → *Security Settings* → *Local Policies* → *User Rights Assignment* → *Create symbolic links* and check that you have user permission to create symbolic links.
5. Restart your computer.
6. Configure git to enable symbolic links on Windows:

```
git config --global core.symlinks true
```

Use git to clone the repository:

```
git clone https://github.com/greatscottgadgets/cynthion.git
```

Note: To install the `cynthion` Python package and allow for in-place editing of the sources you can use the `pip --editable` command:

```
# change to the 'cynthion' Python package directory
cd cynthion/python/

# install the 'cynthion' Python package, including dependencies required for gateware␣
↪development
pip install --editable .
```

CHAPTER

TWENTYTHREE

BITSTREAM GENERATION

Before proceeding, please ensure you have followed the prerequisites in the *Setting up a Development Environment* section.

23.1 Cynthion Gateware

The Cynthion repository contains gateware for two designs:

- `analyzer` – USB analyzer for using Cynthion with Packetry.
- `facedancer` – System-on-Chip for using Cynthion with Facedancer.

Bitstreams can be generated from the `cynthion` Python package sub-directory as follows:

23.1.1 Analyzer Gateware

```
# change to the 'cynthion' Python package directory
cd cynthion/python/

# generate bitstream
python3 -m cynthion.gateware.analyzer.top
```

23.1.2 Facedancer SoC Gateware

```
# change to the 'cynthion' Python package directory
cd cynthion/python/

# generate bitstream
python3 -m cynthion.gateware.facedancer.top
```

23.1.3 Additional Options

Additional options for bitstream generation can be listed by appending `--help` to the command:

```
$ python3 -m cynthion.gateware.analyzer.top --help

usage: top.py [-h] [--output filename] [--erase] [--upload] [--flash]
              [--dry-run] [--keep-files] [--fpga part_number] [--console port]

Gateware generation/upload script for 'USBAnalyzerApplet' gateware.

optional arguments:
  -h, --help            show this help message and exit
  --output filename, -o filename
                        Build and output a bitstream to the given file.
  --erase, -E           Clears the relevant FPGA's flash before performing
                        other options.
  --upload, -U          Uploads the relevant design to the target hardware.
                        Default if no options are provided.
  --flash, -F           Flashes the relevant design to the target hardware's
                        configuration flash.
  --dry-run, -D         When provided as the only option; builds the relevant
                        bitstream without uploading or flashing it.
  --keep-files          Keeps the local files in the default `build` folder.
  --fpga part_number    Overrides build configuration to build for a given
                        FPGA. Useful if no FPGA is connected during build.
  --console port        Attempts to open a convenience 115200 8N1 UART console
                        on the specified port immediately after uploading.
```

CHAPTER

TWENTYFOUR

FACEDANCER SOC FIRMWARE COMPILATION

24.1 Prerequisites

Before proceeding, please ensure you have followed the prerequisites in the *Setting up a Development Environment* section.

24.2 Install Rust Dependencies

You will need to install RISC-V embedded target support to compile the firmware:

```
rustup target add riscv32imac-unknown-none-elf
rustup component add llvm-tools-preview
cargo install cargo-binutils
```

Optionally, to use `gdb` for firmware debugging over JTAG, you will need a RISC-V GNU tool chain:

```
# debian
apt install gcc-riscv64-unknown-elf

# arch
pacman -S riscv-gnu-toolchain-bin

# macos brew - https://github.com/riscv-software-src/homebrew-riscv
brew tap riscv-software-src/riscv
brew install riscv-gnu-toolchain
```

24.3 Building and Running

Firmware for the Cynthion SoC can be found in the `firmware/moondancer/` sub-directory.

You can rebuild the firmware using `cargo` as follows:

```
# change to the Cynthion firmware directory
cd firmware/moondancer/

# rebuild the firmware
cargo build --release
```

To upload the firmware binary to Cynthion and flash the SoC bitstream you can run:

```
# change to the Cynthion firmware directory
cd firmware/moondancer/

# upload firmware and run it
cargo run --release
```

Note: By default the firmware's flash script will look for the SoC UART on `/dev/ttyACM0`, if this is not the case on your machine you will need to specify the correct path using the `UART` environment variable, for example:

```
UART=/dev/cu.usbmodem22401 cargo run --release
```

By default the SoC bitstream is obtained from the latest build in `cynthion/python/build/top.bit` but you can override it with:

```
BITSTREAM=path/to/bitstream.bit cargo run --release
```

24.4 Running Firmware Unit Tests

Once the firmware is running on the SoC you can execute some unit tests to exercise the firmware.

In order to do this you will need to connect both the **CONTROL** and **AUX** ports of the Cynthion to the host and then run:

```
# change to the Cynthion firmware directory
cd firmware/moondancer/

# run firmware unit tests
python -m unittest
```

24.5 Firmware Examples

There are a number of firmware examples in the `firmware/moondancer/examples/` sub-directory.

```
# change to the Cynthion firmware directory
cd firmware/moondancer/

# run example
cargo run --release --example <example name>
```