

---

# Cynthion

**Great Scott Gadgets**

**May 22, 2026**



# USER DOCUMENTATION

<b>1</b>	<b>Cynthion Project Description</b>	<b>3</b>
<b>2</b>	<b>Getting Started with Cynthion</b>	<b>5</b>
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
<b>3</b>	<b>Using Cynthion with Packetry</b>	<b>9</b>
3.1	Prerequisites . . . . .	9
3.2	USB Analyzer Bitstream . . . . .	9
3.3	Connect Hardware . . . . .	10
<b>4</b>	<b>Using Cynthion with Facedancer</b>	<b>11</b>
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
<b>5</b>	<b>Using Cynthion with USB Proxy</b>	<b>15</b>
5.1	Connect Hardware . . . . .	15
5.2	Run a USB Proxy example . . . . .	15
5.3	More Information . . . . .	16
<b>6</b>	<b>Using Cynthion USER I/O with Facedancer</b>	<b>17</b>
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
<b>7</b>	<b>The cynthion command line interface</b>	<b>25</b>
7.1	Command Documentation . . . . .	25
<b>8</b>	<b>Protocol analysis of a USB keyboard</b>	<b>29</b>
8.1	Prerequisites . . . . .	29
8.2	Determine device speed . . . . .	29
8.3	Connect . . . . .	30
8.4	Capture . . . . .	30
8.5	Troubleshooting . . . . .	33

<b>9</b>	<b>Emulation of a USB Device</b>	<b>35</b>
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
<b>10</b>	<b>Gateway Blinky</b>	<b>43</b>
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
<b>11</b>	<b>USB Gateway: Part 1 - Enumeration</b>	<b>47</b>
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
<b>12</b>	<b>USB Gateway: Part 2 - WCID Descriptors</b>	<b>57</b>
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
<b>13</b>	<b>USB Gateway: Part 3 - Control Transfers</b>	<b>69</b>
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
<b>14</b>	<b>USB Gateway: Part 4 - Bulk Transfers</b>	<b>83</b>
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

<b>15 Introduction</b>	<b>99</b>
15.1 Cynthion Hardware . . . . .	100
<b>16 Device Overview</b>	<b>101</b>
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
<b>17 Self-made Hardware Bringup</b>	<b>103</b>
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
<b>18 The apollo command line utility</b>	<b>107</b>
<b>19 Getting Help</b>	<b>109</b>
<b>20 Cynthion Projects and Mentions</b>	<b>111</b>
<b>21 Safety Information</b>	<b>113</b>
21.1 Warnings . . . . .	113
21.2 Instructions For Safe Use . . . . .	113
<b>22 Introduction</b>	<b>115</b>
22.1 Setting up a Development Environment . . . . .	115
22.2 Prerequisites . . . . .	115
22.3 Installation . . . . .	115
<b>23 Bitstream Generation</b>	<b>117</b>
23.1 Cynthion Gateware . . . . .	117
<b>24 Facedancer SoC Firmware Compilation</b>	<b>119</b>
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 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](#) gateway 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!



## 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: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
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: xxxxxxxxxxxxxxxxx
```

## 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
```



## 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: xxxxxxxxxxxxxxxxx
```

### 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.

## 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

---

```
cynthion info
```

You should see output like:

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

### 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:

```
1 import asyncio  
2  
3 from facedancer import main  
4 from facedancer.devices.keyboard import USBKeyboardDevice  
5  
6 device = USBKeyboardDevice()  
7  
8 async def type_letters():  
9     # Wait for device to connect  
10    await asyncio.sleep(2)  
11  
12    # Type a string with the device  
13    await device.type_string("echo hello, facedancer\n")  
14  
15 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](#).



## 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:

```
1  #!/usr/bin/env python3
2  #
3  # This file is part of Facedancer.
4  #
5  """ USB Proxy example; forwards all USB transactions and logs them to the console. """
6
7  from facedancer      import main
8
9  from facedancer.proxy import USBProxyDevice
10 from facedancer.filters import USBProxySetupFilters, USBProxyPrettyPrintFilter
11
12 # replace with the proxied device's information
13 ID_VENDOR = 0x1050
14 ID_PRODUCT = 0x0407
15
16
17 if __name__ == "__main__":
18     # create a USB Proxy Device
19     proxy = USBProxyDevice(idVendor=ID_VENDOR, idProduct=ID_PRODUCT)
20
21     # add a filter to forward control transfers between the target host and
22     # proxied device
23     proxy.add_filter(USBProxySetupFilters(proxy, verbose=0))
24
25     # add a filter to log USB transactions to the console
```

(continues on next page)

(continued from previous page)

```
26 proxy.add_filter(USBProxyPrettyPrintFilter(verbose=5))
27
28 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](#).

## 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-duddy 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`

```
1 import asyncio
2 import logging
3
4 from facedancer import import main, errors
5 from facedancer.devices.keyboard import USBKeyboardDevice
6
7 from cynthion import import Cynthion
8
9 # Subclass USBKeyboardDevice
10 class MyKeyboardDevice(USBKeyboardDevice):
11     def __post_init__(self):
12         super().__post_init__()
13
14         # Get a Cynthion instance.
15         cynthion = Cynthion()
16
17         # Get USER Leds
18         self.leds = cynthion.leds
```

(continues on next page)

(continued from previous page)

```

19
20     # Make sure all USER Leds are off
21     [led.off() for led in self.leds.values()]
22
23     # Get USER Button
24     self.user_button = cynthion.gpio.get_pin("USER")
25
26     def handle_bus_reset(self):
27         # Strobe USER Led0 every time we see a bus reset
28         self.leds[0].strobe(duration=0.1)
29         super().handle_bus_reset()
30
31     def handle_request(self, request):
32         # Strobe USER Led1 every time the host makes a control request
33         self.leds[1].strobe(duration=0.1)
34         super().handle_request(request)
35
36     def control_send(self, endpoint_number, in_request, data, *, blocking = False):
37         # Strobe USER Led2 every time the device responds to a control request
38         self.leds[2].strobe(duration=0.1)
39         super().control_send(endpoint_number, in_request, data, blocking=blocking)
40
41     def handle_data_requested(self, endpoint):
42         report = self._generate_hid_report()
43         endpoint.send(report)
44
45         # Strobe USER Led3 every time the host requested a HID report descriptor from
46         ↪ the host
47         if report[2] == 0:
48             self.leds[3].strobe(duration=0.1)
49         # Strobe USER Led4 if the report descriptor contained a scancode for the host
50         else:
51             self.leds[4].strobe(duration=0.1)
52
53     # Rubber-ducky control script
54     async def type_letters():
55         # Wait for device to connect
56         await asyncio.sleep(2)
57
58         logging.info("Press the USER button to proceed.")
59
60         # Wait until Cynthion's USER button is pressed
61         while device.user_button.read() == False:
62             await asyncio.sleep(0.01)
63
64         logging.info("Typing string into target device.")
65
66         # Type a string with the device
67         await device.type_string("echo hello, facedancer\n")
68
69         logging.info("Finished. Press the USER button again to quit.")

```

(continues on next page)

(continued from previous page)

```
70 # Done
71 while device.user_button.read() == False:
72     await asyncio.sleep(0.01)
73
74     raise errors.EndEmulation("User quit the emulation.")
75
76 # Start emulation
77 device = MyKeyboardDevice()
78 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 flashing in response to device events.

## 6.4 USER Pmod inputs and outputs

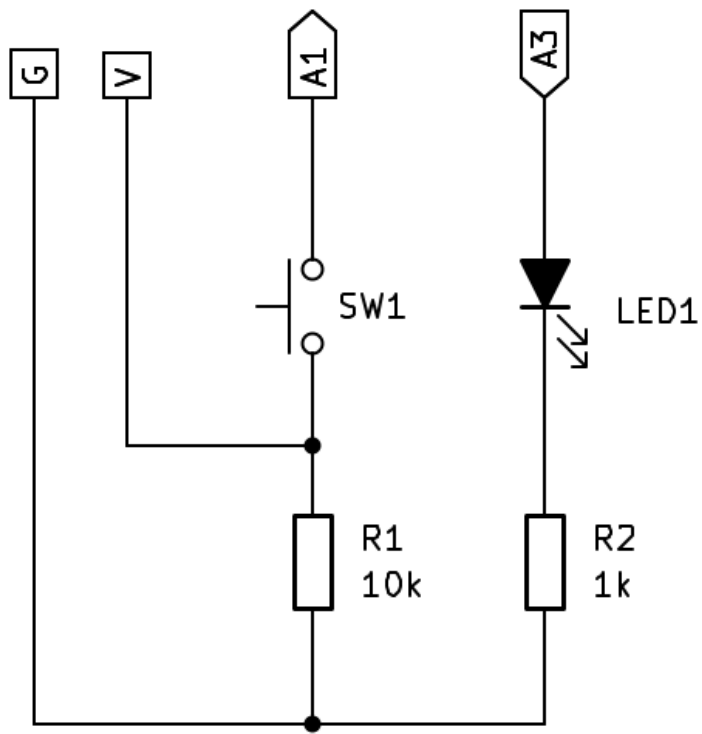
In addition to the USER Button and Leds, Facedancer can also make use of Cynthion USER Pmod A (USER Pmod B is used for JTAG and UART duties) to trigger or respond to external hardware. They use the same gpio APIs as the USER Button but individual pins can also be configured as inputs or outputs.

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:

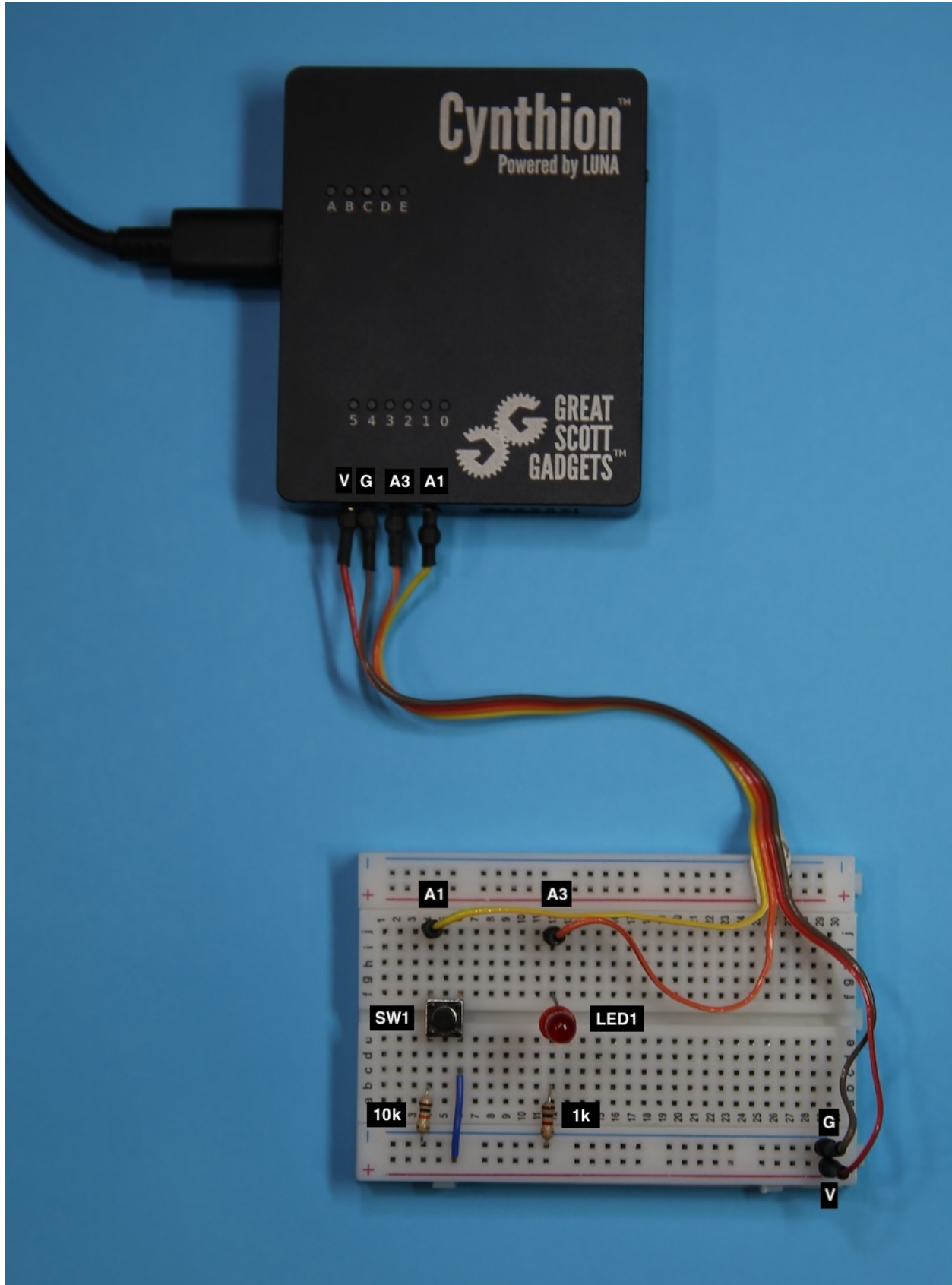
- 1x SPST Switch
- 1x 1 kOhm resistor
- 1x 10 kOhm resistor
- 1x LED
- 1x Breadboard

### 6.4.2 Circuit Diagram





### 6.4.3 Breadboard Layout



6.4. USER Pmod inputs and outputs

## 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`

```
1 import time
2
3 from cynthion import Cynthion
4 from cynthion.interfaces.gpio import PinDirection
5
6 # Get Cynthion instance
7 c = Cynthion()
8
9 # Get USER Pmod Pin A1 and configure it as an input
10 a1 = c.gpio.get_pin("A1")
11 a1.set_direction(PinDirection.Input)
12
13 # Get USER Pmod Pin A3 and configure it as an output
14 a3 = c.gpio.get_pin("A3")
15 a3.set_direction(PinDirection.Output)
16
17 # Continuously read the input value of Pin A1 and output it to Pin A3.
18 while True:
19     value = a1.read()
20     a3.write(value)
21     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.

## 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>
```



## 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 gateway 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 determine 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

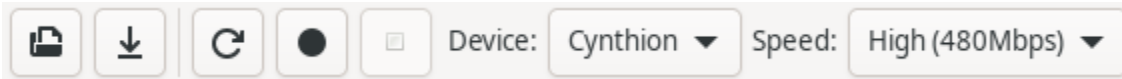
Next, we'll connect everything up for capture. Within the Cynthion hardware the **TARGET C** and **TARGET A** ports are connected together, and the analyzer gateway will capture any packets that are seen going between those ports. These packets are then sent out through the **CONTROL** port.

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

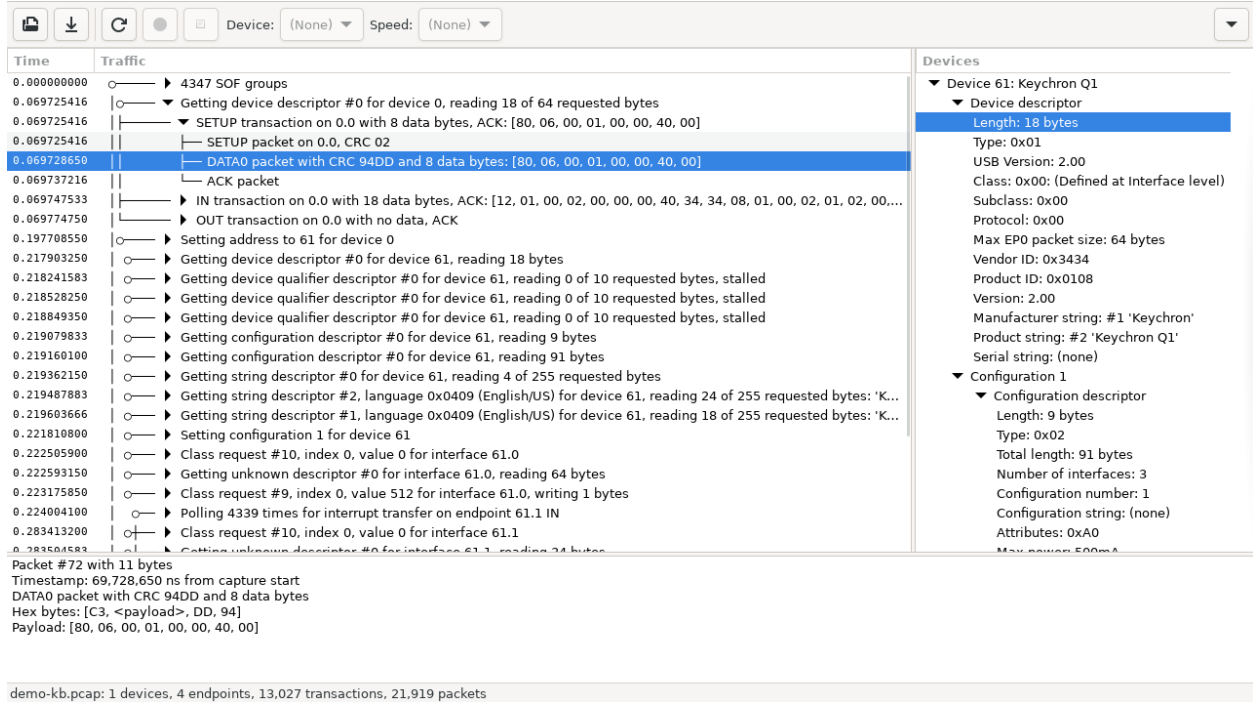
Open Packetry. At the top of the window, you should see the [action bar](#):



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).

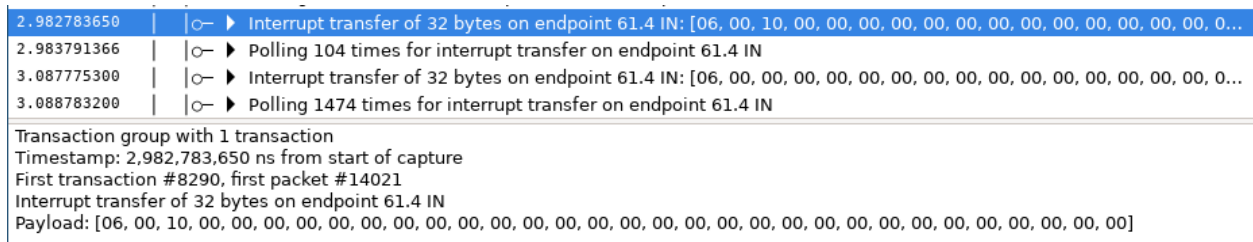


The screenshot shows a Wireshark interface with a traffic capture pane on the left and a device details pane on the right. The traffic pane shows a list of packets with time and description. The device details pane shows information for 'Device 61: Keychron Q1', including device descriptor details, configuration 1, and configuration descriptor details. A packet #72 is expanded to show its details: 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:



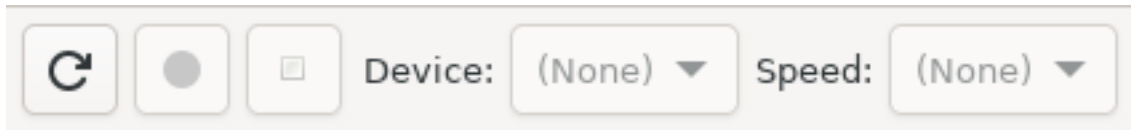
The screenshot shows a Wireshark interface with a traffic capture pane on the left and a transaction details pane on the right. The traffic pane shows a list of packets with time and description. The transaction details pane shows information for a transaction group with 1 transaction: Timestamp: 2,982,783,650 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].

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 gateway 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"

 A screenshot of the Packetry interface showing a traffic capture. The top bar has the same controls as in 8.5.1. The main window is divided into two panes: 'Traffic' on the left and 'Devices' on the right. The 'Traffic' pane shows a list of captured packets. The first three packets are highlighted in blue:
 

Time	Traffic
0.000000000	1 invalid groups
0.000000000	1 malformed packet
0.000000000	Malformed 0-byte packet

 The rest of the list consists of many '1 invalid groups' entries. Below the list, there is a summary: 'Packet #1 with 0 bytes', 'Timestamp: 0 ns from capture start', and 'Malformed 0-byte packet'. At the bottom of the interface, a status bar reads: 'demo-wrong-speed-invalid-groups.pcap: 0 devices, 0 endpoints, 1,806 transactions, 1,806 packets'.

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.



## 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)

```
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)
```

## 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 `gateway-blinky.py` and add the following code to it:

```
1 from amaranth import *
2
3 class Top(Elaboratable):
4     def elaborate(self, platform):
5         m = Module()
6
7         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 `gateway-blinky.py` and add the highlighted line:

```
1 from amaranth import *
2
3 class Top(Elaboratable):
4     def elaborate(self, platform):
5         m = Module()
6
7         leds: Signal(6) = Cat(platform.request("led", n).o for n in range(0, 6))
8
9         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:

```
1 from amaranth import *
2
3 class Top(Elaboratable):
4     def elaborate(self, platform):
5         m = Module()
6
7         leds: Signal(6) = Cat(platform.request("led", n).o for n in range(0, 6))
8
9         half_freq: int = int(60e6 // 2)
10        timer: Signal(25) = Signal(range(half_freq))
11
12        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:

```

1 from amaranth import *
2
3 class Top(Elaboratable):
4     def elaborate(self, platform):
5         m = Module()
6
7         leds: Signal(6) = Cat(platform.request("led", n).o for n in range(0, 6))
8
9         half_freq: int = int(60e6 // 2)
10        timer: Signal(25) = Signal(range(half_freq))
11
12        with m.If(timer == half_freq - 1):
13            m.d.sync += leds.eq(~leds)
14            m.d.sync += timer.eq(0)
15
16        with m.Else():
17            m.d.sync += timer.eq(timer + 1)
18
19        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 `gateway-blinky.py` should now look like this:

```

1 #!/usr/bin/env python3
2 #
3 # This file is part of Cynthion.
4 #
5 # Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
6 # SPDX-License-Identifier: BSD-3-Clause
7
8 from amaranth import *
9
10 class Top(Elaboratable):
11     def elaborate(self, platform):
12         m = Module()

```

(continues on next page)

```
13
14     leds: Signal(6) = Cat(platform.request("led", n).o for n in range(0, 6))
15
16     half_freq: int    = int(60e6 // 2)
17     timer: Signal(25) = Signal(range(half_freq))
18
19     with m.If(timer == half_freq - 1):
20         m.d.sync += leds.eq(~leds)
21         m.d.sync += timer.eq(0)
22
23     with m.Else():
24         m.d.sync += timer.eq(timer + 1)
25
26     return m
27
28 if __name__ == "__main__":
29     from luna import top_level_cli
30     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 gateway-blinky.py
```

The blinky gateway 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.](#)

## 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 gateway 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 *Gateway 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 `gateway-usb-device.py` and add the following code to it:

Listing 1: `gateway-usb-device.py`

```

1 from amaranth import *
2 from usb_protocol.emitters import DeviceDescriptorCollection
3
4 VENDOR_ID = 0x1209 # https://pid.codes/1209/
5 PRODUCT_ID = 0x0001
6
7 class GatewayUSBDevice(Elaboratable):
8     def create_descriptors(self):
9         descriptors = DeviceDescriptorCollection()
10
11         with descriptors.DeviceDescriptor() as d:
12             d.idVendor = VENDOR_ID
13             d.idProduct = PRODUCT_ID
14             d.iManufacturer = "Cynthion Project"
15             d.iProduct = "Gateway USB Device"
16             d.bNumConfigurations = 1
17
18         return descriptors
19
20     def elaborate(self, platform):
21         m = Module()
22         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 situation. 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: `gateway-usb-device.py`

```

1 from amaranth import *
2 from usb_protocol.emitters import DeviceDescriptorCollection
3
4 VENDOR_ID = 0x1209 # https://pid.codes/1209/
5 PRODUCT_ID = 0x0001
6
7 class GatewayUSBDevice(Elaboratable):
8     def create_descriptors(self):
9         descriptors = DeviceDescriptorCollection()
10
11         with descriptors.DeviceDescriptor() as d:
12             d.idVendor = VENDOR_ID

```

(continues on next page)

(continued from previous page)

```

13     d.idProduct          = PRODUCT_ID
14     d.iManufacturer      = "Cynthion Project"
15     d.iProduct           = "Gateware USB Device"
16     d.bNumConfigurations = 1
17
18     with descriptors.ConfigurationDescriptor() as c:
19         with c.InterfaceDescriptor() as i:
20             i.bInterfaceNumber = 0
21
22     return descriptors
23
24     def elaborate(self, platform):
25         m = Module()
26         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

```

1  from amaranth          import *
2  from luna.usb2         import USBDevice
3  from usb_protocol.emitters import DeviceDescriptorCollection
4
5  VENDOR_ID = 0x1209 # https://pid.codes/1209/
6  PRODUCT_ID = 0x0001
7
8  class GatewareUSBDevice(Elaboratable):
9      def create_descriptors(self):
10         descriptors = DeviceDescriptorCollection()
11
12         with descriptors.DeviceDescriptor() as d:
13             d.idVendor          = VENDOR_ID
14             d.idProduct         = PRODUCT_ID
15             d.iManufacturer     = "Cynthion Project"
16             d.iProduct          = "Gateware USB Device"
17             d.bNumConfigurations = 1
18
19         with descriptors.ConfigurationDescriptor() as c:

```

(continues on next page)

```
20         with c.InterfaceDescriptor() as i:
21             i.bInterfaceNumber = 0
22
23         return descriptors
24
25     def elaborate(self, platform):
26         m = Module()
27
28         # configure cynthion's clocks and reset signals
29         m.submodules.car = platform.clock_domain_generator()
30
31         # request the physical interface for cynthion's TARGET C port
32         ulpi = platform.request("target_phy")
33         m.submodules.usb = usb = USBDevice(bus=ulpi)
34
35         # create our descriptors and add them to the device's control endpoint
36         descriptors = self.create_descriptors()
37         control_ep = usb.add_standard_control_endpoint(descriptors)
38
39         # configure the device to connect by default when plugged into a host
40         m.d.comb += usb.connect.eq(1)
41
42         return m
43
44 if __name__ == "__main__":
45     from luna import top_level_cli
46     top_level_cli(GatewayUSBDevice)
```

## 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 gateway and upload it to your Cynthion by typing the following into your terminal shell:

```
python ./gateway-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 successfully 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-gateway-usb-device.py` and add the following code to it:

Listing 4: `test-gateway-usb-device.py`

```

1 import usb1
2
3 def list_available_usb_devices(context):
4     for device in context.getDeviceList():
5         try:
6             manufacturer = device.getManufacturer()
7             product = device.getProduct()
8             print(f"{device}: {manufacturer} - {product}")
9         except Exception as e:
10            print(f"{device}: {e}")
11
12 if __name__ == "__main__":
13     with usb1.USBContext() as context:
14         list_available_usb_devices(context)

```

Run the file with:

```
python ./test-gateway-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 - Gateway 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 similar 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 similar to the highlighted line:

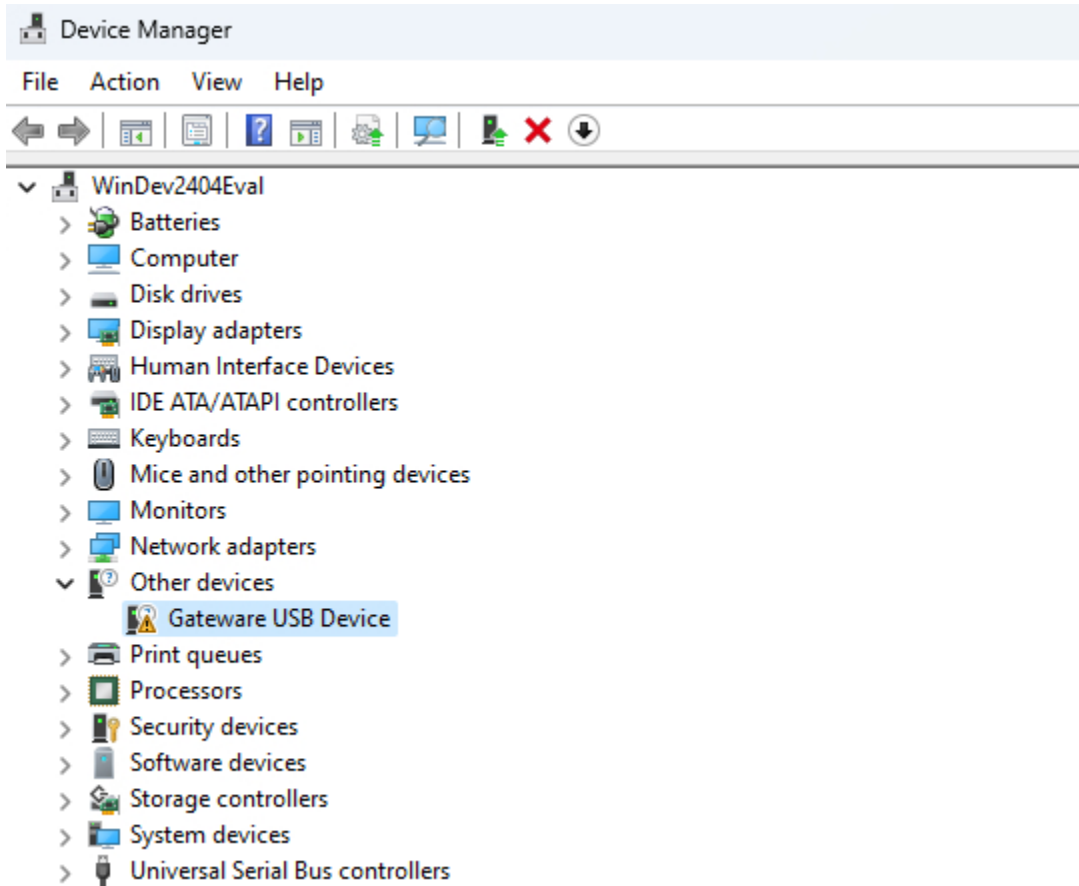
```
% ioreg -b -p IOUSB

+-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 Gateway 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 "Gateway 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 necessity 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 Gateway: 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: gateway-usb-device-01.py

```
1  #!/usr/bin/env python3
2  #
3  # This file is part of Cynthion.
4  #
5  # Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
6  # SPDX-License-Identifier: BSD-3-Clause
7
8  from amaranth import *
9  from luna.usb2 import USBDevice
10 from usb_protocol.emitters import DeviceDescriptorCollection
11
12 VENDOR_ID = 0x1209 # https://pid.codes/1209/
13 PRODUCT_ID = 0x0001
14
15 class GatewayUSBDevice(Elaboratable):
16     """ A simple USB device that can only enumerate. """
17
18     def create_standard_descriptors(self):
19         """ Create the USB descriptors for the device. """
20
21         descriptors = DeviceDescriptorCollection()
22
23         # all USB devices have a single device descriptor
24         with descriptors.DeviceDescriptor() as d:
25             d.idVendor = VENDOR_ID
26             d.idProduct = PRODUCT_ID
27             d.iManufacturer = "Cynthion Project"
28             d.iProduct = "Gateway USB Device"
29
30             d.bNumConfigurations = 1
31
32         # and at least one configuration descriptor
33         with descriptors.ConfigurationDescriptor() as c:
34
```

(continues on next page)

(continued from previous page)

```

35     # with at least one interface descriptor
36     with c.InterfaceDescriptor() as i:
37         i.bInterfaceNumber = 0
38
39         # interfaces also need endpoints to do anything useful
40         # but we'll add those later!
41
42     return descriptors
43
44
45 def elaborate(self, platform):
46     m = Module()
47
48     # configure cynthion's clocks and reset signals
49     m.submodules.car = platform.clock_domain_generator()
50
51     # request the physical interface for cynthion's TARGET C port
52     ulpi = platform.request("target_phy")
53
54     # create the USB device
55     m.submodules.usb = usb = USBDevice(bus=ulpi)
56
57     # create our standard descriptors and add them to the device's control endpoint
58     descriptors = self.create_standard_descriptors()
59     control_endpoint = usb.add_standard_control_endpoint(descriptors)
60
61     # configure the device to connect by default when plugged into a host
62     m.d.comb += usb.connect.eq(1)
63
64     return m
65
66
67 if __name__ == "__main__":
68     from luna import top_level_cli
69     top_level_cli(GatewayUSBDevice)

```

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

```

1  import usb1
2
3  # - list available usb devices -----
4
5  def list_available_usb_devices(context):
6      for device in context.getDeviceList():
7          try:
8              manufacturer = device.getManufacturer()
9              product = device.getProduct()
10             print(f"{device}: {manufacturer} - {product}")
11         except Exception as e:
12             print(f"{device}: {e}")
13
14

```

(continues on next page)

(continued from previous page)

```
15 # - main -----
16
17 if __name__ == "__main__":
18     with usb1.USBContext() as context:
19         list_available_usb_devices(context)
```

## 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 `gateway-usb-device.py` file from the previous tutorial and add/modify the highlighted lines:

Listing 1: `gateway-usb-device.py`

```

1  from amaranth                               import *
2  from luna.usb2                               import USBDevice
3  from usb_protocol.emitters                  import DeviceDescriptorCollection
4
5  from usb_protocol.emitters.descriptors.standard import get_string_descriptor
6
7  ...
8
9  class GatewayUSBDevice(Elaboratable):
10     ...
11
12     def elaborate(self, platform):
13         m = Module()
14
15         # configure cynthion's clocks and reset signals
16         m.submodules.car = platform.clock_domain_generator()
17
18         # request the physical interface for cynthion's TARGET C port
19         ulpi = platform.request("target_phy")
20
21         # create the USB device
22         m.submodules.usb = usb = USBDevice(bus=ulpi)
23
24         # create our standard descriptors and add them to the device's control endpoint
25         descriptors = self.create_standard_descriptors()
26         control_endpoint = usb.add_standard_control_endpoint(descriptors)
27
28         # add the microsoft os string descriptor
29         descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
30
31         # configure the device to connect by default when plugged into a host
32         m.d.comb += usb.connect.eq(1)
33
34     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: gateway-usb-device.py

```

1 from amaranth import *
2 from luna.usb2 import USBDevice
3 from usb_protocol.emitters import DeviceDescriptorCollection
4
5 from luna.gateway.usb.request.windows import (
6     MicrosoftOS10DescriptorCollection,
7 )
8 from usb_protocol.emitters.descriptors.standard import get_string_descriptor
9
10
11 VENDOR_ID = 0x1209 # https://pid.codes/1209/
12 PRODUCT_ID = 0x0001
13
14 class GatewayUSBDevice(Elaboratable):
15     ...
16
17     def elaborate(self, platform):
18         ...
19
20         # add the microsoft os string descriptor
21         descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
22
23         # add a microsoft descriptor collection for our other two microsoft descriptors
24         msft_descriptors = MicrosoftOS10DescriptorCollection()
25
26         # add the microsoft compatible id feature descriptor
27         with msft_descriptors.ExtendedCompatIDDescriptor() as c:
28             with c.Function() as f:
29                 f.bFirstInterfaceNumber = 0
30                 f.compatibleID = 'WINUSB'
31
32         # configure the device to connect by default when plugged into a host
33         m.d.comb += usb.connect.eq(1)
34
35     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 extend 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: gateway-usb-device.py

```
1 from amaranth import *
2 from luna.usb2 import USBDevice
3 from usb_protocol.emitters import DeviceDescriptorCollection
4
5 from luna.gateway.usb.request.windows import (
6     MicrosoftOS10DescriptorCollection,
7 )
8 from usb_protocol.emitters.descriptors.standard import get_string_descriptor
9 from usb_protocol.types.descriptors.microsoft10 import RegistryTypes
10
11 ..
12
13 class GatewayUSBDevice(Elaboratable):
14     ...
15
16     def elaborate(self, platform):
17         ...
18
19         # add the microsoft os string descriptor
20         descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
21
22         # add a microsoft descriptor collection for our other two microsoft descriptors
23         msft_descriptors = MicrosoftOS10DescriptorCollection()
24
25         # add the microsoft compatible id feature descriptor
26         with msft_descriptors.ExtendedCompatIDDescriptor() as c:
27             with c.Function() as f:
28                 f.bFirstInterfaceNumber = 0
29                 f.compatibleID          = 'WINUSB'
```

(continues on next page)

(continued from previous page)

```

30
31     # add microsoft extended properties feature descriptor
32     with msft_descriptors.ExtendedPropertiesDescriptor() as d:
33         with d.Property() as p:
34             p.dwPropertyDataType = RegistryTypes.REG_SZ
35             p.PropertyName       = "DeviceInterfaceGUID"
36             p.PropertyData       = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"
37
38     # configure the device to connect by default when plugged into a host
39     m.d.comb += usb.connect.eq(1)
40
41     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: gateway-usb-device.py

```

1  from amaranth                               import *
2  from luna.usb2                               import USBDevice
3  from usb_protocol.emitters                  import DeviceDescriptorCollection
4
5  from luna.gateway.usb.request.windows      import (
6      MicrosoftOS10DescriptorCollection,
7      MicrosoftOS10RequestHandler,
8  )
9  from usb_protocol.emitters.descriptors.standard import get_string_descriptor
10 from usb_protocol.types.descriptors.microsoft10 import RegistryTypes
11
12 ..
13
14 class GatewayUSBDevice(Elaboratable):
15     ...
16
17     def elaborate(self, platform):
18         ...
19
20         # add the microsoft os string descriptor
21         descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
22
23         # add a microsoft descriptor collection for our other two microsoft descriptors
24         msft_descriptors = MicrosoftOS10DescriptorCollection()
25

```

(continues on next page)

(continued from previous page)

```

26  # add the microsoft compatible id feature descriptor
27  with msft_descriptors.ExtendedCompatIDDescriptor() as c:
28      with c.Function() as f:
29          f.bFirstInterfaceNumber = 0
30          f.compatibleID          = 'WINUSB'
31
32  # add microsoft extended properties feature descriptor
33  with msft_descriptors.ExtendedPropertiesDescriptor() as d:
34      with d.Property() as p:
35          p.dwPropertyDataType = RegistryTypes.REG_SZ
36          p.PropertyName      = "DeviceInterfaceGUID"
37          p.PropertyData      = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"
38
39  # add the request handler for Microsoft descriptors
40  msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
41  control_endpoint.add_request_handler(msft_handler)
42
43  # configure the device to connect by default when plugged into a host
44  m.d.comb += usb.connect.eq(1)
45
46  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

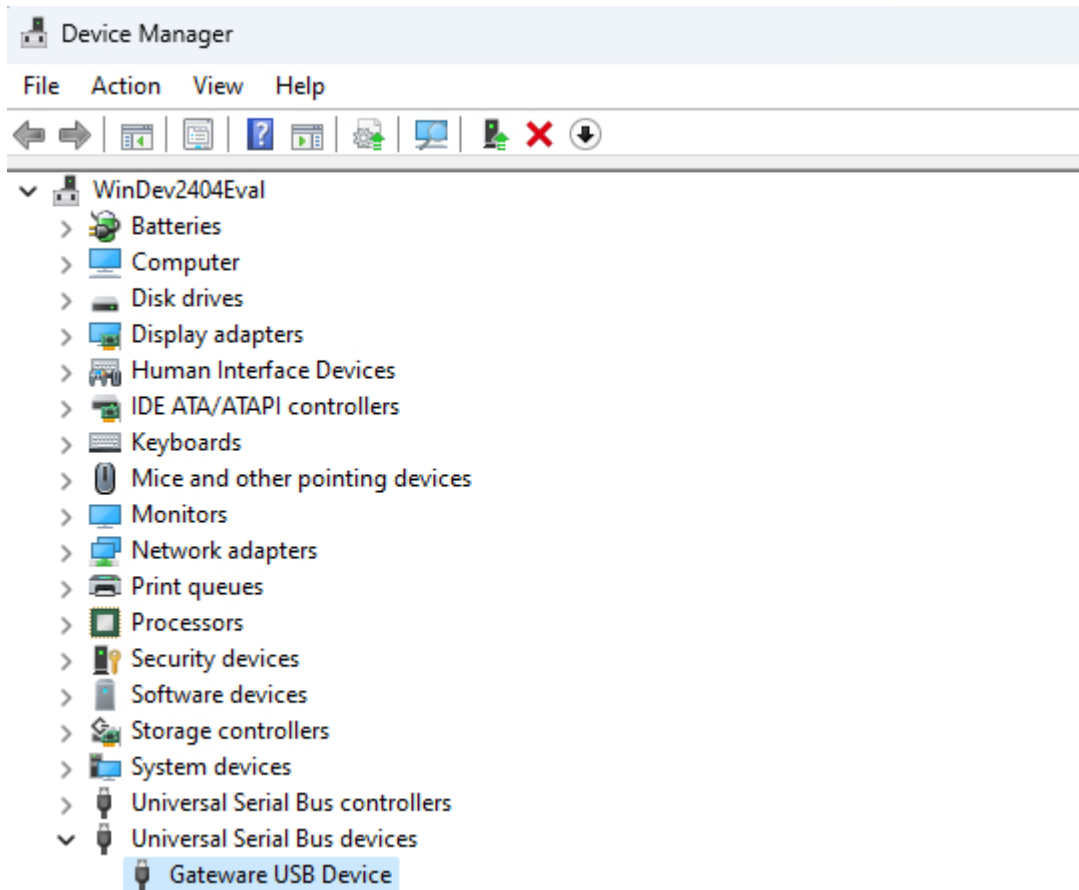
Build the device gateway and upload it to your Cynthion by typing the following into your terminal shell:

```
python ./gateway-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 Gateway: Part 1 - Enumeration* now works as expected:

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

```

1 import usb1
2
3 def list_devices(context):
4     for device in context.getDeviceList():
5         try:
6             manufacturer = device.getManufacturer()
7             product = device.getProduct()
8             print(f"{device}: {manufacturer} - {product}")
9         except Exception as e:
10            print(f"{device}: {e}")
11
12 if __name__ == "__main__":
13     with usb1.USBContext() as context:
14         list_devices(context)

```

Run the file with:

```
python ./test-gateway-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 - Gateway 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 Gateway: 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: gateway-usb-device-02.py

```
1 #!/usr/bin/env python3
2 #
3 # This file is part of Cynthion.
4 #
5 # Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
6 # SPDX-License-Identifier: BSD-3-Clause
7
8 from amaranth import *
```

(continues on next page)

(continued from previous page)

```

9  from luna.usb2                                import USBDevice
10 from usb_protocol.emitters                    import DeviceDescriptorCollection
11
12 from luna.gateware.usb.request.windows        import (
13     MicrosoftOS10DescriptorCollection,
14     MicrosoftOS10RequestHandler,
15 )
16 from usb_protocol.emitters.descriptors.standard import get_string_descriptor
17 from usb_protocol.types.descriptors.microsoft10 import RegistryTypes
18
19 VENDOR_ID = 0x1209 # https://pid.codes/1209/
20 PRODUCT_ID = 0x0001
21
22 class GatewareUSBDevice(Elaboratable):
23     """ A simple USB device that can also enumerate on Windows. """
24
25     def create_standard_descriptors(self):
26         """ Create the USB descriptors for the device. """
27
28         descriptors = DeviceDescriptorCollection()
29
30         # all USB devices have a single device descriptor
31         with descriptors.DeviceDescriptor() as d:
32             d.idVendor      = VENDOR_ID
33             d.idProduct     = PRODUCT_ID
34             d.iManufacturer = "Cynthion Project"
35             d.iProduct      = "Gateware USB Device"
36
37             d.bNumConfigurations = 1
38
39         # and at least one configuration descriptor
40         with descriptors.ConfigurationDescriptor() as c:
41
42             # with at least one interface descriptor
43             with c.InterfaceDescriptor() as i:
44                 i.bInterfaceNumber = 0
45
46                 # interfaces also need endpoints to do anything useful
47                 # but we'll add those later!
48
49         return descriptors
50
51
52     def elaborate(self, platform):
53         m = Module()
54
55         # configure cynthion's clocks and reset signals
56         m.submodules.car = platform.clock_domain_generator()
57
58         # request the physical interface for cynthion's TARGET C port
59         ulpi = platform.request("target_phy")
60

```

(continues on next page)

(continued from previous page)

```

61     # create the USB device
62     m.submodules.usb = usb = USBDevice(bus=ulpi)
63
64     # create our standard descriptors and add them to the device's control endpoint
65     descriptors = self.create_standard_descriptors()
66     control_endpoint = usb.add_standard_control_endpoint(descriptors)
67
68     # add the microsoft os string descriptor
69     descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
70
71     # add a microsoft descriptor collection for our other two microsoft descriptors
72     msft_descriptors = MicrosoftOS10DescriptorCollection()
73
74     # add the microsoft compatible id feature descriptor
75     with msft_descriptors.ExtendedCompatIDDescriptor() as c:
76         with c.Function() as f:
77             f.bFirstInterfaceNumber = 0
78             f.compatibleID           = 'WINUSB'
79
80     # add microsoft extended properties feature descriptor
81     with msft_descriptors.ExtendedPropertiesDescriptor() as d:
82         with d.Property() as p:
83             p.dwPropertyDataType = RegistryTypes.REG_SZ
84             p.PropertyName       = "DeviceInterfaceGUID"
85             p.PropertyData       = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"
86
87     # add the request handler for Microsoft descriptors
88     msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
89     control_endpoint.add_request_handler(msft_handler)
90
91     # configure the device to connect by default when plugged into a host
92     m.d.comb += usb.connect.eq(1)
93
94     return m
95
96
97 if __name__ == "__main__":
98     from luna import top_level_cli
99     top_level_cli(GatewayUSBDevice)

```

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

```

1  import usb1
2
3  # - list available usb devices -----
4
5  def list_available_usb_devices(context):
6      for device in context.getDeviceList():
7          try:
8              manufacturer = device.getManufacturer()
9              product = device.getProduct()
10             print(f"{device}: {manufacturer} - {product}")

```

(continues on next page)

(continued from previous page)

```
11     except Exception as e:
12         print(f"{device}: {e}")
13
14
15 # - main -----
16
17 if __name__ == "__main__":
18     with usb1.USBContext() as context:
19         list_available_usb_devices(context)
```



## 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 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: `gateway-usb-device.py`

```

1  from amaranth                                import *
2  from luna.usb2                                import USBDevice
3  from usb_protocol.emitters                    import DeviceDescriptorCollection
4
5  from luna.gateway.usb.request.windows         import (
6      MicrosoftOS10DescriptorCollection,
7      MicrosoftOS10RequestHandler,
8  )
9  from usb_protocol.emitters.descriptors.standard import get_string_descriptor
10 from usb_protocol.types.descriptors.microsoft10 import RegistryTypes
11
12 from luna.gateway.stream.generator            import StreamSerializer
13 from luna.gateway.usb.request.control         import ControlRequestHandler
14 from luna.gateway.usb.usb2.transfer           import USBInStreamInterface
15
16 VENDOR_ID = 0x1209 # https://pid.codes/1209/
17 PRODUCT_ID = 0x0001
18
19 class VendorRequestHandler(ControlRequestHandler):
20     VENDOR_SET_FPGA_LEDS = 0x01
21     VENDOR_GET_USER_BUTTON = 0x02
22
23     def elaborate(self, platform):
24         m = Module()
25
26         # shortcuts
27         interface: RequestHandlerInterface = self.interface
28         setup: SetupPacket = self.interface.setup
29
30         # get a reference to the FPGA LEDs and USER button
31         fpga_leds = Cat(platform.request("led", i).o for i in range(6))
32         user_button = platform.request("button_user").i
33
34         # create a streamserializer for transmitting IN data back to the host
35         serializer = StreamSerializer(
36             domain = "usb",
37             stream_type = USBInStreamInterface,
38             data_length = 1,
39             max_length_width = 1,
40         )

```

(continues on next page)

(continued from previous page)

```

41     m.submodules += serializer
42
43     return m
44
45 class GatewayUSBDevice(Elaboratable):
46
47     ...
48
49     def elaborate(self, platform):
50         m = Module()
51
52         # configure cynthion's clocks and reset signals
53         m.submodules.car = platform.clock_domain_generator()
54
55         # request the physical interface for cynthion's TARGET C port
56         ulpi = platform.request("target_phy")
57
58         # create the USB device
59         m.submodules.usb = usb = USBDevice(bus=ulpi)
60
61         # create our standard descriptors and add them to the device's control endpoint
62         descriptors = self.create_standard_descriptors()
63         control_endpoint = usb.add_standard_control_endpoint(descriptors)
64
65         # add microsoft os 1.0 descriptors and request handler
66         descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
67         msft_descriptors = MicrosoftOS10DescriptorCollection()
68         with msft_descriptors.ExtendedCompatIDDescriptor() as c:
69             with c.Function() as f:
70                 f.bFirstInterfaceNumber = 0
71                 f.compatibleID = 'WINUSB'
72         with msft_descriptors.ExtendedPropertiesDescriptor() as d:
73             with d.Property() as p:
74                 p.dwPropertyDataType = RegistryTypes.REG_SZ
75                 p.PropertyName = "DeviceInterfaceGUID"
76                 p.PropertyData = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"
77         msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
78         control_endpoint.add_request_handler(msft_handler)
79
80         # add our vendor request handler
81         control_endpoint.add_request_handler(VendorRequestHandler())
82
83         # configure the device to connect by default when plugged into a host
84         m.d.comb += usb.connect.eq(1)
85
86         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: gateway-usb-device.py

```

1 class VendorRequestHandler(ControlRequestHandler):
2     VENDOR_SET_FPGA_LEDS = 0x01
3     VENDOR_GET_USER_BUTTON = 0x02
4
5     def elaborate(self, platform):
6         m = Module()
7
8         # Shortcuts.
9         interface: RequestHandlerInterface = self.interface
10        setup: SetupPacket = self.interface.setup
11
12        # Grab a reference to the FPGA LEDs and USER button.
13        fpga_leds = Cat(platform.request("led", i).o for i in range(6))
14        user_button = platform.request("button_user").i
15
16        # Create a StreamSerializer for sending IN data back to the host
17        serializer = StreamSerializer(
18            domain = "usb",
19            stream_type = USBInStreamInterface,
20            data_length = 1,
21            max_length_width = 1,
22        )
23        m.submodules += serializer
24
25        # we've received a setup packet containing a vendor request.
26        with m.If(setup.type == USBRequestType.VENDOR):
27            # use a state machine to sequence our request handling
28            with m.FSM(domain="usb"):
29                with m.State("IDLE"):
30                    with m.If(setup.received):
31                        with m.Switch(setup.request):
32                            with m.Case(self.VENDOR_SET_FPGA_LEDS):
33                                m.next = "HANDLE_SET_FPGA_LEDS"
34                            with m.Case(self.VENDOR_GET_USER_BUTTON):
35                                m.next = "HANDLE_GET_USER_BUTTON"
36
37                with m.State("HANDLE_SET_FPGA_LEDS"):
38                    # take ownership of the interface
39                    m.d.comb += interface.claim.eq(1)
40
41                    # if we have an active data byte, set the FPGA LEDs to the payload
42                    with m.If(interface.rx.valid & interface.rx.next):
43                        m.d.usb += fpga_leds.eq(interface.rx.payload[0:6])
44
45                    # once the receive is complete, respond with an ACK
46                    with m.If(interface.rx_ready_for_response):
47                        m.d.comb += interface.handshakes_out.ack.eq(1)

```

(continues on next page)

(continued from previous page)

```

48
49     # finally, once we reach the status stage, send a ZLP
50     with m.If(interface.status_requested):
51         m.d.comb += self.send_zlp()
52         m.next = "IDLE"
53
54     with m.State("HANDLE_GET_USER_BUTTON"):
55         # take ownership of the interface
56         m.d.comb += interface.claim.eq(1)
57
58         # write the state of the user button into a local data register
59         data = Signal(8)
60         m.d.comb += data[0].eq(user_button)
61
62         # transmit our data using a built-in handler function that
63         # automatically advances the FSM back to the 'IDLE' state on
64         # completion
65         self.handle_simple_data_request(m, serializer, data)
66
67     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 interested 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 gateway to your Cynthion with:

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

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

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

```

1 import usb1
2 import time
3

```

(continues on next page)

```
4  VENDOR_ID = 0x1209 # https://pid.codes/1209/
5  PRODUCT_ID = 0x0001
6
7  VENDOR_SET_FPGA_LEDS = 0x01
8  VENDOR_GET_USER_BUTTON = 0x02
9
10 # - list available usb devices -----
11
12 def list_available_usb_devices(context):
13     for device in context.getDeviceList():
14         try:
15             manufacturer = device.getManufacturer()
16             product = device.getProduct()
17             print(f"{device}: {manufacturer} - {product}")
18         except Exception as e:
19             print(f"{device}: {e}")
20
21
22 # - wrappers for control requests -----
23
24 def set_fpga_leds(device_handle, led_state):
25     response = device_handle.controlWrite(
26         request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
27         request      = VENDOR_SET_FPGA_LEDS,
28         index        = 0,
29         value         = 0,
30         data          = [led_state],
31         timeout       = 1000,
32     )
33
34 def get_user_button(device_handle):
35     response = device_handle.controlRead(
36         request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE | usb1.ENDPOINT_OUT,
37         request      = VENDOR_GET_USER_BUTTON,
38         index        = 0,
39         value         = 0,
40         length       = 1,
41         timeout       = 1000,
42     )
43     return response[0]
44
45
46 # - test control endpoints -----
47
48 def test_control_endpoints(device_handle):
49     led_counter = 0
50     last_button_state = False
51
52     while True:
53         # led counter
54         set_fpga_leds(device_handle, led_counter)
55         led_counter = (led_counter + 1) % 256
```

(continues on next page)

(continued from previous page)

```

56     # reset led counter when the USER button is pressed
57     button_state = get_user_button(device_handle)
58     if button_state:
59         led_counter = 0
60
61
62     # print button state when it changes
63     if button_state != last_button_state:
64         print(f"USER button is: {'ON' if button_state else 'OFF' }")
65         last_button_state = button_state
66
67     # slow the loop down so we can see the counter change
68     time.sleep(0.1)
69
70
71 # - main -----
72
73 if __name__ == "__main__":
74     with usb1.USBContext() as context:
75         # list available devices
76         list_available_usb_devices(context)
77
78         # get a device handle to our simple usb device
79         device_handle = context.openByVendorIDAndProductID(VENDOR_ID, PRODUCT_ID)
80         if device_handle is None:
81             raise Exception("Device not found.")
82
83         # claim the device's interface
84         device_handle.claimInterface(0)
85
86         # pass the device handle to our control endpoint test
87         test_control_endpoints(device_handle)

```

Run the file with:

```
python ./test-gateway-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: gateway-usb-device-03.py

```

1  #!/usr/bin/env python3
2  #
3  # This file is part of Cynthion.
4  #
5  # Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
6  # SPDX-License-Identifier: BSD-3-Clause
7
8  from amaranth                                import *
9  from luna.usb2                                import USBDevice
10 from usb_protocol.emitters                    import DeviceDescriptorCollection
11
12 from luna.gateway.usb.request.windows         import (
13     MicrosoftOS10DescriptorCollection,
14     MicrosoftOS10RequestHandler,
15 )
16 from usb_protocol.emitters.descriptors.standard import get_string_descriptor
17 from usb_protocol.types.descriptors.microsoft10 import RegistryTypes
18
19 from luna.gateway.stream.generator           import StreamSerializer
20 from luna.gateway.usb.request.control        import ControlRequestHandler
21 from luna.gateway.usb.request.interface     import SetupPacket
22 from luna.gateway.usb.usb2.request          import RequestHandlerInterface
23 from luna.gateway.usb.usb2.transfer         import USBInStreamInterface
24 from usb_protocol.types                      import USBRequestType
25
26 VENDOR_ID = 0x1209 # https://pid.codes/1209/
27 PRODUCT_ID = 0x0001
28
29 class VendorRequestHandler(ControlRequestHandler):
30     VENDOR_SET_FPGA_LEDS = 0x01
31     VENDOR_GET_USER_BUTTON = 0x02
32
33     def elaborate(self, platform):
34         m = Module()

```

(continues on next page)

(continued from previous page)

```

35
36 # shortcuts
37 interface: RequestHandlerInterface = self.interface
38 setup: SetupPacket = self.interface.setup
39
40 # get a reference to the FPGA LEDs and USER button
41 fpga_leds = Cat(platform.request("led", i).o for i in range(6))
42 user_button = platform.request("button_user").i
43
44 # create a streamserializer for transmitting IN data back to the host
45 serializer = StreamSerializer(
46     domain          = "usb",
47     stream_type     = USBInStreamInterface,
48     data_length     = 1,
49     max_length_width = 1,
50 )
51 m.submodules += serializer
52
53 # we've received a setup packet containing a vendor request.
54 with m.If(setup.type == USBRequestType.VENDOR):
55     # use a state machine to sequence our request handling
56     with m.FSM(domain="usb"):
57         with m.State("IDLE"):
58             with m.If(setup.received):
59                 with m.Switch(setup.request):
60                     with m.Case(self.VENDOR_SET_FPGA_LEDS):
61                         m.next = "HANDLE_SET_FPGA_LEDS"
62                     with m.Case(self.VENDOR_GET_USER_BUTTON):
63                         m.next = "HANDLE_GET_USER_BUTTON"
64
65         with m.State("HANDLE_SET_FPGA_LEDS"):
66             # take ownership of the interface
67             m.d.comb += interface.claim.eq(1)
68
69             # if we have an active data byte, set the FPGA LEDs to the payload
70             with m.If(interface.rx.valid & interface.rx.next):
71                 m.d.usb += fpga_leds.eq(interface.rx.payload[0:6])
72
73             # once the receive is complete, respond with an ACK
74             with m.If(interface.rx_ready_for_response):
75                 m.d.comb += interface.handshakes_out.ack.eq(1)
76
77             # finally, once we reach the status stage, send a ZLP
78             with m.If(interface.status_requested):
79                 m.d.comb += self.send_zlp()
80                 m.next = "IDLE"
81
82         with m.State("HANDLE_GET_USER_BUTTON"):
83             # take ownership of the interface
84             m.d.comb += interface.claim.eq(1)
85
86             # write the state of the user button into a local data register

```

(continues on next page)

(continued from previous page)

```
87         data = Signal(8)
88         m.d.comb += data[0].eq(user_button)
89
90         # transmit our data using a built-in handler function that
91         # automatically advances the FSM back to the 'IDLE' state on
92         # completion
93         self.handle_simple_data_request(m, serializer, data)
94
95     return m
96
97
98 class GatewareUSBDevice(Elaboratable):
99     """ A simple USB device that can communicate with the host via vendor requests. """
100
101     def create_standard_descriptors(self):
102         """ Create the USB descriptors for the device. """
103
104         descriptors = DeviceDescriptorCollection()
105
106         # all USB devices have a single device descriptor
107         with descriptors.DeviceDescriptor() as d:
108             d.idVendor          = VENDOR_ID
109             d.idProduct         = PRODUCT_ID
110             d.iManufacturer     = "Cynthion Project"
111             d.iProduct          = "Gateware USB Device"
112
113             d.bNumConfigurations = 1
114
115         # and at least one configuration descriptor
116         with descriptors.ConfigurationDescriptor() as c:
117
118             # with at least one interface descriptor
119             with c.InterfaceDescriptor() as i:
120                 i.bInterfaceNumber = 0
121
122                 # interfaces also need endpoints to do anything useful
123                 # but we'll add those later!
124
125         return descriptors
126
127     def elaborate(self, platform):
128         m = Module()
129
130         # configure cynthion's clocks and reset signals
131         m.submodules.car = platform.clock_domain_generator()
132
133         # request the physical interface for cynthion's TARGET C port
134         ulpi = platform.request("target_phy")
135
136         # create the USB device
137         m.submodules.usb = usb = USBDevice(bus=ulpi)
138
```

(continues on next page)

(continued from previous page)

```

139     # create our standard descriptors and add them to the device's control endpoint
140     descriptors = self.create_standard_descriptors()
141     control_endpoint = usb.add_standard_control_endpoint(descriptors)
142
143     # add microsoft os 1.0 descriptors and request handler
144     descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
145     msft_descriptors = MicrosoftOS10DescriptorCollection()
146     with msft_descriptors.ExtendedCompatIDDescriptor() as c:
147         with c.Function() as f:
148             f.bFirstInterfaceNumber = 0
149             f.compatibleID = 'WINUSB'
150     with msft_descriptors.ExtendedPropertiesDescriptor() as d:
151         with d.Property() as p:
152             p.dwPropertyDataType = RegistryTypes.REG_SZ
153             p.PropertyName = "DeviceInterfaceGUID"
154             p.PropertyData = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"
155     msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
156     control_endpoint.add_request_handler(msft_handler)
157
158     # add the vendor request handler
159     control_endpoint.add_request_handler(VendorRequestHandler())
160
161     # configure the device to connect by default when plugged into a host
162     m.d.comb += usb.connect.eq(1)
163
164     return m
165
166
167 if __name__ == "__main__":
168     from luna import top_level_cli
169     top_level_cli(GatewayUSBDevice)

```

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

```

1  import usb1
2  import time
3
4  VENDOR_ID = 0x1209 # https://pid.codes/1209/
5  PRODUCT_ID = 0x0001
6
7  VENDOR_SET_FPGA_LEDS = 0x01
8  VENDOR_GET_USER_BUTTON = 0x02
9
10 # - list available usb devices -----
11
12 def list_available_usb_devices(context):
13     for device in context.getDeviceList():
14         try:
15             manufacturer = device.getManufacturer()
16             product = device.getProduct()
17             print(f"{device}: {manufacturer} - {product}")
18         except Exception as e:

```

(continues on next page)

```
19         print(f"{device}: {e}")
20
21
22 # - wrappers for control requests -----
23
24 def set_fpga_leds(device_handle, led_state):
25     response = device_handle.controlWrite(
26         request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
27         request      = VENDOR_SET_FPGA_LEDS,
28         index       = 0,
29         value       = 0,
30         data        = [led_state],
31         timeout     = 1000,
32     )
33
34 def get_user_button(device_handle):
35     response = device_handle.controlRead(
36         request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE | usb1.ENDPOINT_OUT,
37         request      = VENDOR_GET_USER_BUTTON,
38         index       = 0,
39         value       = 0,
40         length      = 1,
41         timeout     = 1000,
42     )
43     return response[0]
44
45
46 # - test control endpoints -----
47
48 def test_control_endpoints(device_handle):
49     led_counter = 0
50     last_button_state = False
51
52     while True:
53         # led counter
54         set_fpga_leds(device_handle, led_counter)
55         led_counter = (led_counter + 1) % 256
56
57         # reset led counter when the USER button is pressed
58         button_state = get_user_button(device_handle)
59         if button_state:
60             led_counter = 0
61
62         # print button state when it changes
63         if button_state != last_button_state:
64             print(f"USER button is: {'ON' if button_state else 'OFF' }")
65             last_button_state = button_state
66
67         # slow the loop down so we can see the counter change
68         time.sleep(0.1)
69
70
```

(continues on next page)

(continued from previous page)

```
71 # - main -----
72
73 if __name__ == "__main__":
74     with usb1.USBContext() as context:
75         # list available devices
76         list_available_usb_devices(context)
77
78         # get a device handle to our simple usb device
79         device_handle = context.openByVendorIDAndProductID(VENDOR_ID, PRODUCT_ID)
80         if device_handle is None:
81             raise Exception("Device not found.")
82
83         # claim the device's interface
84         device_handle.claimInterface(0)
85
86         # pass the device handle to our control endpoint test
87         test_control_endpoints(device_handle)
```



## 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 `gateway-usb-device.py` and add the highlighted lines:

Listing 1: `gateway-usb-device.py`

```

1 from amaranth import *
2 from luna.usb2 import USBDevice
3 from usb_protocol.emitters import DeviceDescriptorCollection
4
5 from luna.gateway.usb.request.windows import (
6     MicrosoftOS10DescriptorCollection,
7     MicrosoftOS10RequestHandler,
8 )
9 from usb_protocol.emitters.descriptors.standard import get_string_descriptor
10 from usb_protocol.types.descriptors.microsoft10 import RegistryTypes
11
12 from luna.gateway.stream.generator import StreamSerializer
13 from luna.gateway.usb.request.control import ControlRequestHandler
14 from luna.gateway.usb.request.interface import SetupPacket
15 from luna.gateway.usb.usb2.request import RequestHandlerInterface
16 from luna.gateway.usb.usb2.transfer import USBInStreamInterface
17 from usb_protocol.types import USBRequestType
18
19 from luna.usb2 import USBStreamInEndpoint,
20     USBStreamOutEndpoint,
21 )
22 from usb_protocol.types import USBDirection,
23     USBTransferType,
24 )
25
26
27
28 VENDOR_ID = 0x1209 # https://pid.codes/1209/
29 PRODUCT_ID = 0x0001
30
31 MAX_PACKET_SIZE = 512
32
33 class VendorRequestHandler(ControlRequestHandler):
34     ...
35
36 class GatewayUSBDevice(Elaboratable):
37     def create_standard_descriptors(self):
38         descriptors = DeviceDescriptorCollection()
39
40         with descriptors.DeviceDescriptor() as d:
41             d.idVendor = VENDOR_ID
42             d.idProduct = PRODUCT_ID
43             d.iManufacturer = "Cynthion Project"
44             d.iProduct = "Gateway USB Device"
45             d.bNumConfigurations = 1
46
47         with descriptors.ConfigurationDescriptor() as c:

```

(continues on next page)

(continued from previous page)

```

48     with c.InterfaceDescriptor() as i:
49         i.bInterfaceNumber = 0
50         # EP 0x01 OUT - receives bulk data from the host
51         with i.EndpointDescriptor() as e:
52             e.bEndpointAddress = USBDirection.OUT.to_endpoint_address(0x01)
53             e.bmAttributes      = USBTransferType.BULK
54             e.wMaxPacketSize    = MAX_PACKET_SIZE
55         # EP 0x82 IN - transmits bulk data to the host
56         with i.EndpointDescriptor() as e:
57             e.bEndpointAddress = USBDirection.IN.to_endpoint_address(0x02)
58             e.bmAttributes      = USBTransferType.BULK
59             e.wMaxPacketSize    = MAX_PACKET_SIZE
60
61     return descriptors
62
63     def elaborate(self, platform):
64         ...

```

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 gateway 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 gateway components.

Listing 2: gateway-usb-device.py

```

1     ...
2
3     class GatewayUSBDevice(Elaboratable):
4         def create_standard_descriptors(self):
5             ...
6
7         def elaborate(self, platform):
8             ...
9
10            # add the vendor request handler
11            control_endpoint.add_request_handler(VendorRequestHandler())
12
13            # create and add stream endpoints for our device's Bulk IN & OUT endpoints
14            ep_out = USBStreamOutEndpoint(
15                endpoint_number=0x01, # (EP 0x01)

```

(continues on next page)

(continued from previous page)

```

16         max_packet_size=MAX_PACKET_SIZE,
17     )
18     usb.add_endpoint(ep_out)
19     ep_in = USBStreamInEndpoint(
20         endpoint_number=0x02, # (EP 0x82)
21         max_packet_size=MAX_PACKET_SIZE
22     )
23     usb.add_endpoint(ep_in)
24
25     # configure the device to connect by default when plugged into a host
26     m.d.comb += usb.connect.eq(1)
27
28     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: gateway-usb-device.py

```

1  from amaranth                               import *
2  from amaranth.lib.fifo                       import SyncFIFO
3  from luna.usb2                               import USBDevice
4  from usb_protocol.emitters                  import DeviceDescriptorCollection
5  ...
6
7  class VendorRequestHandler(ControlRequestHandler):
8      ...
9
10 class GatewayUSBDevice(Elaboratable):
11     ...
12
13     def elaborate(self, platform):
14         ...
15

```

(continues on next page)

(continued from previous page)

```

16  # create and add stream endpoints for our device's Bulk IN & OUT endpoints
17  ep_out = USBStreamOutEndpoint(
18      endpoint_number=0x01, # (EP 0x01)
19      max_packet_size=MAX_PACKET_SIZE,
20  )
21  usb.add_endpoint(ep_out)
22  ep_in = USBStreamInEndpoint(
23      endpoint_number=0x02, # (EP 0x82)
24      max_packet_size=MAX_PACKET_SIZE
25  )
26  usb.add_endpoint(ep_in)
27
28  # create a FIFO queue we'll connect to the stream interfaces of our
29  # IN & OUT endpoints
30  m.submodules.fifo = fifo = DomainRenamer("usb")(
31      SyncFIFO(width=8, depth=MAX_PACKET_SIZE)
32  )
33
34  # connect our Bulk OUT endpoint's stream interface to the FIFO's write port
35  stream_out = ep_out.stream
36  m.d.comb += fifo.w_data.eq(stream_out.payload)
37  m.d.comb += fifo.w_en.eq(stream_out.valid)
38  m.d.comb += stream_out.ready.eq(fifo.w_rdy)
39
40  # connect our Bulk IN endpoint's stream interface to the FIFO's read port
41  stream_in = ep_in.stream
42  m.d.comb += stream_in.payload.eq(fifo.r_data)
43  m.d.comb += stream_in.valid.eq(fifo.r_rdy)
44  m.d.comb += fifo.r_en.eq(stream_in.ready)
45
46  # configure the device to connect by default when plugged into a host
47  m.d.comb += usb.connect.eq(1)
48
49  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 & gateway 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 gateway, 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-gateway-usb-device.py` and add the following code to it:

Listing 4: `test-gateway-usb-device.py`

```
1 import usb1
2 import time
3 import random
4
5 VENDOR_ID = 0x1209 # https://pid.codes/1209/
6 PRODUCT_ID = 0x0001
7
8 VENDOR_SET_FPGA_LEDS = 0x01
9 VENDOR_GET_USER_BUTTON = 0x02
10
11 MAX_PACKET_SIZE = 512
12
13 # - list available usb devices -----
14
15 def list_available_usb_devices(context):
16     for device in context.getDeviceList():
17         try:
18             manufacturer = device.getManufacturer()
19             product = device.getProduct()
20             print(f"{device}: {manufacturer} - {product}")
21         except Exception as e:
22             print(f"{device}: {e}")
23
24
25 # - wrappers for control requests -----
26
27 def set_fpga_leds(device_handle, led_state):
28     response = device_handle.controlWrite(
29         request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
30         request      = VENDOR_SET_FPGA_LEDS,
31         index        = 0,
32         value        = 0,
33         data         = [led_state],
34         timeout      = 1000,
35     )
36
37 def get_user_button(device_handle):
38     response = device_handle.controlRead(
39         request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE | usb1.ENDPOINT_OUT,
40         request      = VENDOR_GET_USER_BUTTON,
41         index        = 0,
```

(continues on next page)

(continued from previous page)

```

42     value         = 0,
43     length        = 1,
44     timeout       = 1000,
45 )
46 return response[0]
47
48
49 # - test control endpoints -----
50
51 def test_control_endpoints(device_handle):
52     led_counter = 0
53     last_button_state = False
54
55     while True:
56         # led counter
57         set_fpga_leds(device_handle, led_counter)
58         led_counter = (led_counter + 1) % 256
59
60         # reset led counter when the USER button is pressed
61         button_state = get_user_button(device_handle)
62         if button_state:
63             led_counter = 0
64
65         # print button state when it changes
66         if button_state != last_button_state:
67             print(f"USER button is: {'ON' if button_state else 'OFF' }")
68             last_button_state = button_state
69
70         # slow the loop down so we can see the counter change
71         time.sleep(0.1)
72
73
74 # - wrappers for bulk requests -----
75
76 def bulk_out_transfer(device_handle, data):
77     response = device_handle.bulkWrite(
78         endpoint = 0x01,
79         data      = data,
80         timeout   = 1000,
81     )
82     return response
83
84 def bulk_in_transfer(device_handle, length):
85     response = device_handle.bulkRead(
86         endpoint = 0x02,
87         length   = length,
88         timeout  = 1000,
89     )
90     return response
91
92
93 # - test bulk endpoints -----

```

(continues on next page)

(continued from previous page)

```

94
95 def test_bulk_endpoints(device_handle):
96     # bulk_out - write a list of random numbers to memory
97     data = list([random.randint(0, 255) for _ in range(MAX_PACKET_SIZE)])
98     response = bulk_out_transfer(device_handle, data)
99     print(f"OUT endpoint transmitted {response} bytes: {data[0:4]} ... {data[-4:]}")
100
101     # bulk_in - retrieve the contents of our memory
102     response = list(bulk_in_transfer(device_handle, MAX_PACKET_SIZE))
103     print(f"IN endpoint received {len(response)} bytes: {response[0:4]} ...
↪ {response[-4:]}")
104
105     # check that the stored data matches the sent data
106     assert(data == list(response))
107
108
109 # - main -----
110
111 if __name__ == "__main__":
112     with usb1.USBContext() as context:
113         # list available devices
114         list_available_usb_devices(context)
115
116         # get a device handle to our simple usb device
117         device_handle = context.openByVendorIDAndProductID(VENDOR_ID, PRODUCT_ID)
118         if device_handle is None:
119             raise Exception("Device not found.")
120
121         # claim the device's interface
122         device_handle.claimInterface(0)
123
124         # pass the device handle to our bulk endpoint test
125         test_bulk_endpoints(device_handle)
126
127         # pass the device handle to our control endpoint test
128         test_control_endpoints(device_handle)

```

Run the file with:

```
python ./test-gateway-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 Gateway 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: `gateway-usb-device-04.py`

```

1  #!/usr/bin/env python3
2  #
3  # This file is part of Cynthion.
4  #
5  # Copyright (c) 2024 Great Scott Gadgets <info@greatscottgadgets.com>
6  # SPDX-License-Identifier: BSD-3-Clause
7
8  from amaranth                               import *
9  from amaranth.lib.fifo                       import SyncFIFO
10 from luna.usb2                               import USBDevice
11 from usb_protocol.emitters                  import DeviceDescriptorCollection
12
13 from luna.gateway.usb.request.windows       import (
14     MicrosoftOS10DescriptorCollection,
15     MicrosoftOS10RequestHandler,
16 )
17 from usb_protocol.emitters.descriptors.standard import get_string_descriptor
18 from usb_protocol.types.descriptors.microsoft10 import RegistryTypes
19
20 from luna.gateway.stream.generator          import StreamSerializer
21 from luna.gateway.usb.request.control      import ControlRequestHandler
22 from luna.gateway.usb.request.interface    import SetupPacket
23 from luna.gateway.usb.usb2.request        import RequestHandlerInterface
24 from luna.gateway.usb.usb2.transfer       import USBInStreamInterface
25 from usb_protocol.types                    import USBRequestType
26
27 from luna.usb2                             import USBStreamInEndpoint,
28     ↳USBStreamOutEndpoint
29 from usb_protocol.types                      import USBDirection, USBTransferType
30
31 VENDOR_ID = 0x1209 # https://pid.codes/1209/
32 PRODUCT_ID = 0x0001

```

(continues on next page)

```

33 MAX_PACKET_SIZE = 512
34
35 class VendorRequestHandler(ControlRequestHandler):
36     VENDOR_SET_FPGA_LEDS = 0x01
37     VENDOR_GET_USER_BUTTON = 0x02
38
39     def elaborate(self, platform):
40         m = Module()
41
42         # shortcuts
43         interface: RequestHandlerInterface = self.interface
44         setup: SetupPacket = self.interface.setup
45
46         # get a reference to the FPGA LEDs and USER button
47         fpga_leds = Cat(platform.request("led", i).o for i in range(6))
48         user_button = platform.request("button_user").i
49
50         # create a streamserializer for transmitting IN data back to the host
51         serializer = StreamSerializer(
52             domain = "usb",
53             stream_type = USBInStreamInterface,
54             data_length = 1,
55             max_length_width = 1,
56         )
57         m.submodules += serializer
58
59         # we've received a setup packet containing a vendor request.
60         with m.If(setup.type == USBRequestType.VENDOR):
61             # use a state machine to sequence our request handling
62             with m.FSM(domain="usb"):
63                 with m.State("IDLE"):
64                     with m.If(setup.received):
65                         with m.Switch(setup.request):
66                             with m.Case(self.VENDOR_SET_FPGA_LEDS):
67                                 m.next = "HANDLE_SET_FPGA_LEDS"
68                             with m.Case(self.VENDOR_GET_USER_BUTTON):
69                                 m.next = "HANDLE_GET_USER_BUTTON"
70
71                 with m.State("HANDLE_SET_FPGA_LEDS"):
72                     # take ownership of the interface
73                     m.d.comb += interface.claim.eq(1)
74
75                     # if we have an active data byte, set the FPGA LEDs to the payload
76                     with m.If(interface.rx.valid & interface.rx.next):
77                         m.d.usb += fpga_leds.eq(interface.rx.payload[0:6])
78
79                     # once the receive is complete, respond with an ACK
80                     with m.If(interface.rx_ready_for_response):
81                         m.d.comb += interface.handshakes_out.ack.eq(1)
82
83                     # finally, once we reach the status stage, send a ZLP
84                     with m.If(interface.status_requested):

```

(continues on next page)

(continued from previous page)

```

85         m.d.comb += self.send_zlp()
86         m.next = "IDLE"
87
88     with m.State("HANDLE_GET_USER_BUTTON"):
89         # take ownership of the interface
90         m.d.comb += interface.claim.eq(1)
91
92         # write the state of the user button into a local data register
93         data = Signal(8)
94         m.d.comb += data[0].eq(user_button)
95
96         # transmit our data using a built-in handler function that
97         # automatically advances the FSM back to the 'IDLE' state on
98         # completion
99         self.handle_simple_data_request(m, serializer, data)
100
101     return m
102
103
104 class GatewareUSBDevice(Elaboratable):
105     """ A simple USB device that can communicate with the host via vendor and bulk
106     ↪ requests. """
107
108     def create_standard_descriptors(self):
109         """ Create the USB descriptors for the device. """
110
111         descriptors = DeviceDescriptorCollection()
112
113         # all USB devices have a single device descriptor
114         with descriptors.DeviceDescriptor() as d:
115             d.idVendor      = VENDOR_ID
116             d.idProduct     = PRODUCT_ID
117             d.iManufacturer = "Cynthion Project"
118             d.iProduct      = "Gateware USB Device"
119
120             d.bNumConfigurations = 1
121
122         # and at least one configuration descriptor
123         with descriptors.ConfigurationDescriptor() as c:
124
125             # with at least one interface descriptor
126             with c.InterfaceDescriptor() as i:
127                 i.bInterfaceNumber = 0
128
129             # an endpoint for receiving bulk data from the host
130             with i.EndpointDescriptor() as e:
131                 e.bEndpointAddress = USBDirection.OUT.to_endpoint_address(0x01) # EP_
132                 ↪ 0x01 OUT
133
134                 e.bmAttributes      = USBTransferType.BULK
135                 e.wMaxPacketSize    = MAX_PACKET_SIZE
136
137             # and an endpoint for transmitting bulk data to the host

```

(continues on next page)

(continued from previous page)

```

135         with i.EndpointDescriptor() as e:
136             e.bEndpointAddress = USBDirection.IN.to_endpoint_address(0x02) # EP_
↳ 0x82 IN
137             e.bmAttributes      = USBTransferType.BULK
138             e.wMaxPacketSize    = MAX_PACKET_SIZE
139
140         return descriptors
141
142     def elaborate(self, platform):
143         m = Module()
144
145         # configure cynthion's clocks and reset signals
146         m.submodules.car = platform.clock_domain_generator()
147
148         # request the physical interface for cynthion's TARGET C port
149         ulpi = platform.request("target_phy")
150
151         # create the USB device
152         m.submodules.usb = usb = USBDevice(bus=ulpi)
153
154         # create our standard descriptors and add them to the device's control endpoint
155         descriptors = self.create_standard_descriptors()
156         control_endpoint = usb.add_standard_control_endpoint(descriptors)
157
158         # add microsoft os 1.0 descriptors and request handler
159         descriptors.add_descriptor(get_string_descriptor("MSFT100\xee"), index=0xee)
160         msft_descriptors = MicrosoftOS10DescriptorCollection()
161         with msft_descriptors.ExtendedCompatIDDescriptor() as c:
162             with c.Function() as f:
163                 f.bFirstInterfaceNumber = 0
164                 f.compatibleID          = 'WINUSB'
165         with msft_descriptors.ExtendedPropertiesDescriptor() as d:
166             with d.Property() as p:
167                 p.dwPropertyDataType = RegistryTypes.REG_SZ
168                 p.PropertyName       = "DeviceInterfaceGUID"
169                 p.PropertyData       = "{88bae032-5a81-49f0-bc3d-a4ff138216d6}"
170         msft_handler = MicrosoftOS10RequestHandler(msft_descriptors, request_code=0xee)
171         control_endpoint.add_request_handler(msft_handler)
172
173         # add the vendor request handler
174         control_endpoint.add_request_handler(VendorRequestHandler())
175
176         # create and add stream endpoints for our device's Bulk IN & OUT endpoints
177         ep_out = USBStreamOutEndpoint(
178             endpoint_number=0x01, # (EP 0x01)
179             max_packet_size=MAX_PACKET_SIZE,
180         )
181         usb.add_endpoint(ep_out)
182         ep_in = USBStreamInEndpoint(
183             endpoint_number=0x02, # (EP 0x82)
184             max_packet_size=MAX_PACKET_SIZE
185         )

```

(continues on next page)

(continued from previous page)

```

186     usb.add_endpoint(ep_in)
187
188     # create a FIFO queue we'll connect to the stream interfaces of our
189     # IN & OUT endpoints
190     m.submodules.fifo = fifo = DomainRenamer("usb")(
191         SyncFIFO(width=8, depth=MAX_PACKET_SIZE)
192     )
193
194     # connect our Bulk OUT endpoint's stream interface to the FIFO's write port
195     stream_out = ep_out.stream
196     m.d.comb += fifo.w_data.eq(stream_out.payload)
197     m.d.comb += fifo.w_en.eq(stream_out.valid)
198     m.d.comb += stream_out.ready.eq(fifo.w_rdy)
199
200     # connect our Bulk IN endpoint's stream interface to the FIFO's read port
201     stream_in = ep_in.stream
202     m.d.comb += stream_in.payload.eq(fifo.r_data)
203     m.d.comb += stream_in.valid.eq(fifo.r_rdy)
204     m.d.comb += fifo.r_en.eq(stream_in.ready)
205
206     # configure the device to connect by default when plugged into a host
207     m.d.comb += usb.connect.eq(1)
208
209     return m
210
211
212 if __name__ == "__main__":
213     from luna import top_level_cli
214     top_level_cli(GatewayUSBDevice)

```

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

```

1  import usb1
2  import time
3  import random
4
5  VENDOR_ID = 0x1209 # https://pid.codes/1209/
6  PRODUCT_ID = 0x0001
7
8  VENDOR_SET_FPGA_LEDS = 0x01
9  VENDOR_GET_USER_BUTTON = 0x02
10
11 MAX_PACKET_SIZE = 512
12
13 # - list available usb devices -----
14
15 def list_available_usb_devices(context):
16     for device in context.getDeviceList():
17         try:
18             manufacturer = device.getManufacturer()
19             product = device.getProduct()
20             print(f"{device}: {manufacturer} - {product}")

```

(continues on next page)

```
21     except Exception as e:
22         print(f"{device}: {e}")
23
24
25 # - wrappers for control requests -----
26
27 def set_fpga_leds(device_handle, led_state):
28     response = device_handle.controlWrite(
29         request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE,
30         request      = VENDOR_SET_FPGA_LEDS,
31         index        = 0,
32         value        = 0,
33         data         = [led_state],
34         timeout      = 1000,
35     )
36
37 def get_user_button(device_handle):
38     response = device_handle.controlRead(
39         request_type = usb1.TYPE_VENDOR | usb1.RECIPIENT_DEVICE | usb1.ENDPOINT_OUT,
40         request      = VENDOR_GET_USER_BUTTON,
41         index        = 0,
42         value        = 0,
43         length       = 1,
44         timeout      = 1000,
45     )
46     return response[0]
47
48
49 # - test control endpoints -----
50
51 def test_control_endpoints(device_handle):
52     led_counter = 0
53     last_button_state = False
54
55     while True:
56         # led counter
57         set_fpga_leds(device_handle, led_counter)
58         led_counter = (led_counter + 1) % 256
59
60         # reset led counter when the USER button is pressed
61         button_state = get_user_button(device_handle)
62         if button_state:
63             led_counter = 0
64
65         # print button state when it changes
66         if button_state != last_button_state:
67             print(f"USER button is: {'ON' if button_state else 'OFF' }")
68             last_button_state = button_state
69
70         # slow the loop down so we can see the counter change
71         time.sleep(0.1)
72
```

(continues on next page)

(continued from previous page)

```

73
74 # - wrappers for bulk requests -----
75
76 def bulk_out_transfer(device_handle, data):
77     response = device_handle.bulkWrite(
78         endpoint = 0x01,
79         data      = data,
80         timeout   = 1000,
81     )
82     return response
83
84 def bulk_in_transfer(device_handle, length):
85     response = device_handle.bulkRead(
86         endpoint = 0x02,
87         length   = length,
88         timeout  = 1000,
89     )
90     return response
91
92
93 # - test bulk endpoints -----
94
95 def test_bulk_endpoints(device_handle):
96     # bulk_out - write a list of random numbers to memory
97     data = list([random.randint(0, 255) for _ in range(MAX_PACKET_SIZE)])
98     response = bulk_out_transfer(device_handle, data)
99     print(f"OUT endpoint transmitted {response} bytes: {data[0:4]} ... {data[-4:]}")
100
101     # bulk_in - retrieve the contents of our memory
102     response = list(bulk_in_transfer(device_handle, MAX_PACKET_SIZE))
103     print(f"IN endpoint received {len(response)} bytes: {response[0:4]} ...
104     ↪ {response[-4:]}")
105
106     # check that the stored data matches the sent data
107     assert(data == list(response))
108
109 # - main -----
110
111 if __name__ == "__main__":
112     with usb1.USBContext() as context:
113         # list available devices
114         list_available_usb_devices(context)
115
116         # get a device handle to our simple usb device
117         device_handle = context.openByVendorIDAndProductID(VENDOR_ID, PRODUCT_ID)
118         if device_handle is None:
119             raise Exception("Device not found.")
120
121         # claim the device's interface
122         device_handle.claimInterface(0)
123

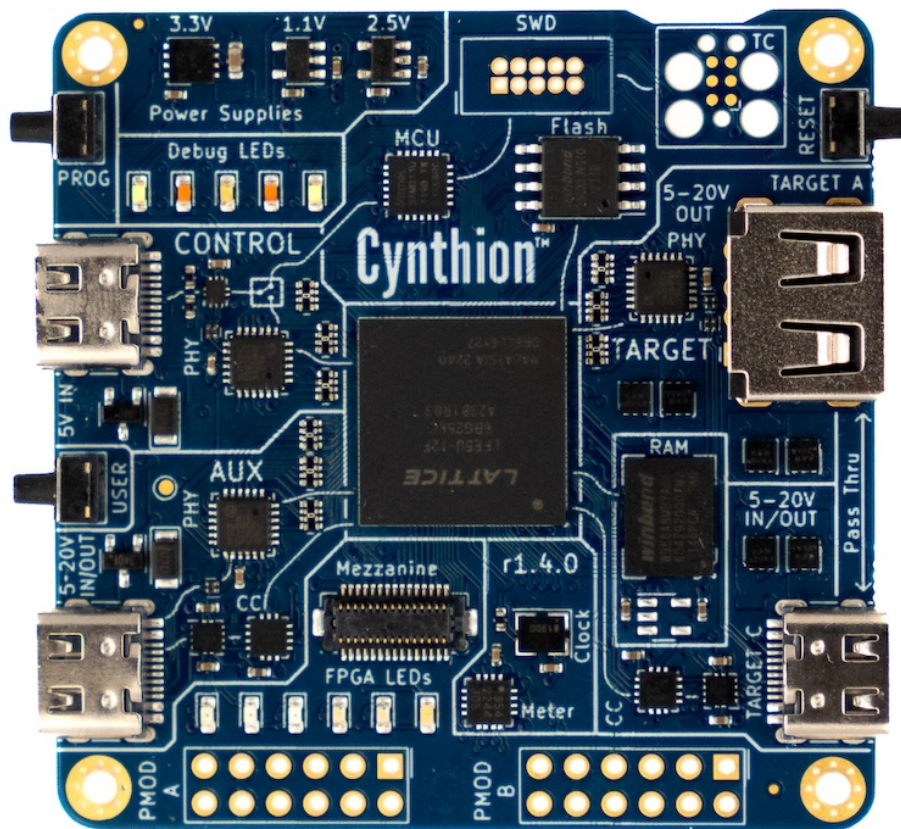
```

(continues on next page)

(continued from previous page)

```
124     # pass the device handle to our bulk endpoint test  
125     test_bulk_endpoints(device_handle)  
126  
127     # pass the device handle to our control endpoint test  
128     test_control_endpoints(device_handle)
```

INTRODUCTION



## 15.1 Cynthion Hardware

## 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

## 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: 0xffffffff msp: 0xffffffff
** 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 gateway 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 gateway 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.

## 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
```



## 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.



## 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](mailto:info@greatscottgadgets.com) with a link to what you've done and we might post it here!



## 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.



## 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
```

## Cynthion

---

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 gateway_  
→development  
pip install --editable .
```

## BITSTREAM GENERATION

Before proceeding, please ensure you have followed the prerequisites in the *Setting up a Development Environment* section.

### 23.1 Cynthion Gateway

The Cynthion repository contains gateway 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 Gateway

```
# change to the 'cynthion' Python package directory
cd cynthion/python/

# generate bitstream
python3 -m cynthion.gateway.analyzer.top
```

#### 23.1.2 Facedancer SoC Gateway

```
# change to the 'cynthion' Python package directory
cd cynthion/python/

# generate bitstream
python3 -m cynthion.gateway.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.
```

## 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
```

## Cynthion

---

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>
```