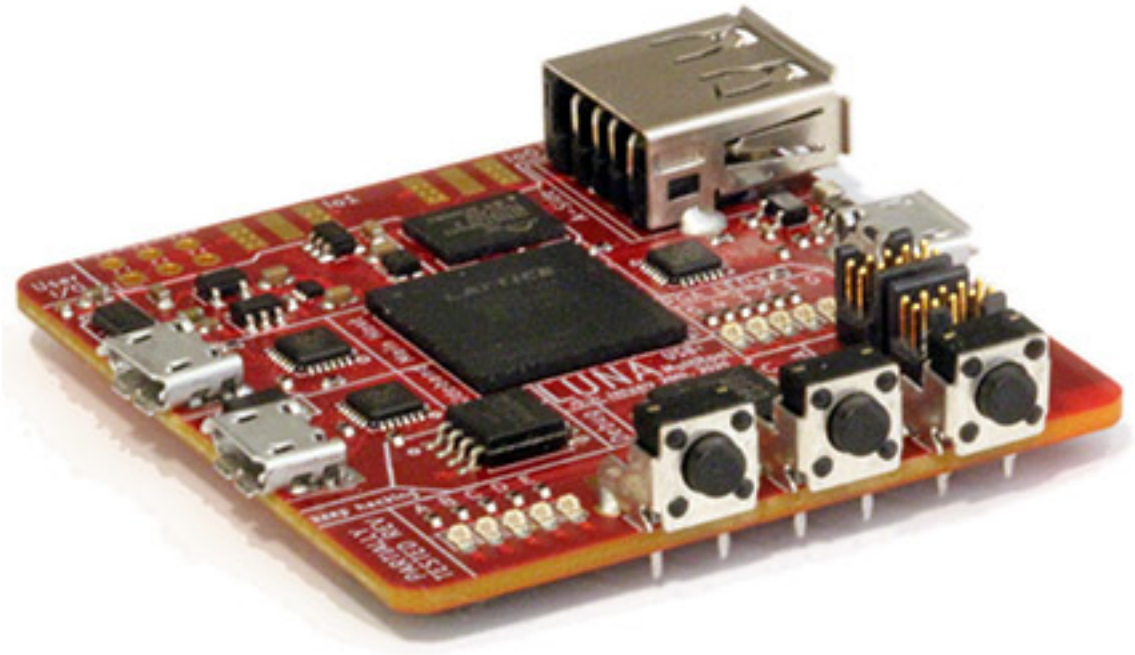
LUNA

Mar 02, 2021

Contents:

This is the documentation for the LUNA gateware library; and the developer document for the LUNA USB multitool hardware and software.



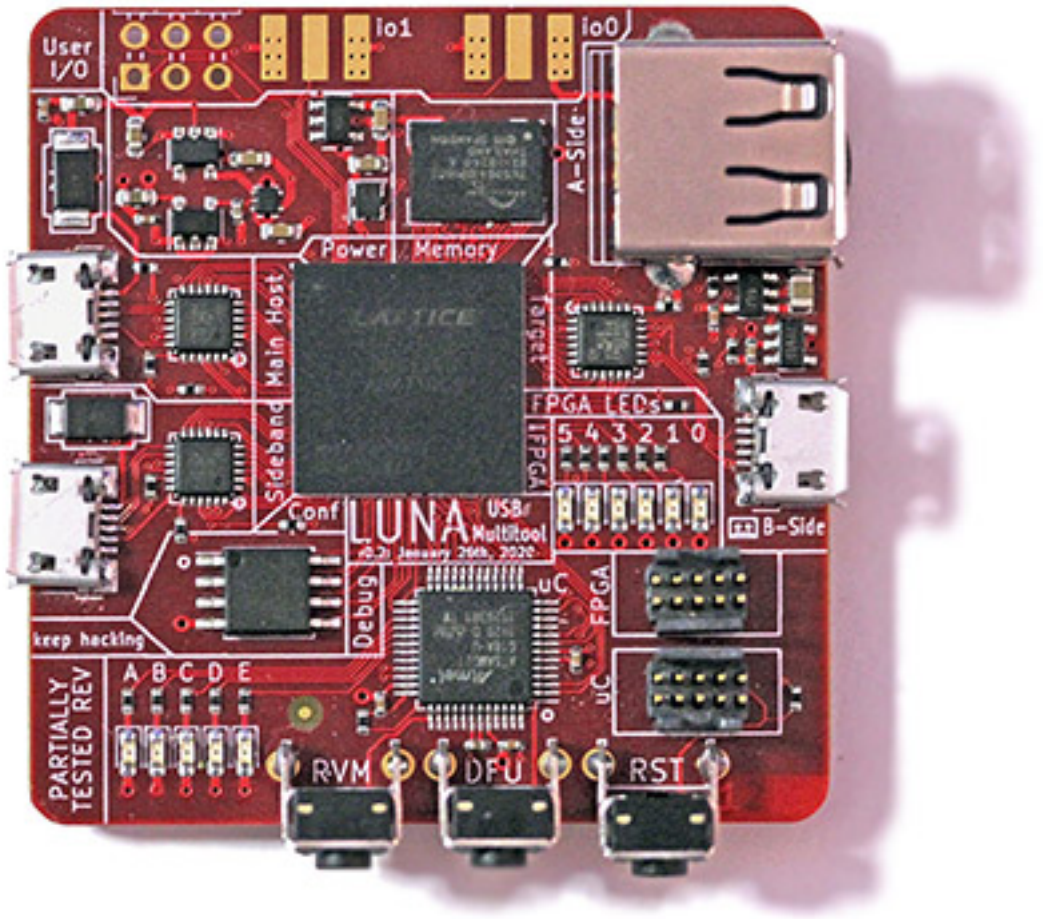
Much like the LUNA hardware, gateware, and software, this documentation is a work in progress. Contributions are always appreciated.

CHAPTER 1

Introduction

Note: LUNA is still a work in progress; and while much of the technology is in a usable state, much of its feature-set is still being built. Consider LUNA an 'unstable' library, for the time being.

Welcome to the LUNA project! LUNA is a full toolkit for working with USB using FPGA technology; and provides hardware, gateware, and software to enable USB applications.



Some things you can use LUNA for, currently:

- **Protocol analysis for Low, Full or High speed USB.** LUNA provides both hardware designs and gateware that allow passive USB monitoring. When combined with the ViewSB USB analyzer toolkit, LUNA hardware+gateware can be used as a full-featured USB analyzer.
- **Creating your own Low, Full or High speed USB device.** LUNA provides a collection of nMigen gateware that allows you to easily create USB devices in gateware, software, or a combination of the two.
- **Building USB functionality into a new or existing System-on-a-Chip (SoC).** LUNA is capable of generating custom peripherals targeting the common Wishbone bus; allowing it to easily be integrated into SoC designs; and the library provides simple automation for developing simple SoC designs.

Some things you'll be able to use LUNA for in the future:

- **Man-in-the-middle'ing USB communications.** The LUNA toolkit will be able to act as a *USB proxy*, transparently modifying USB data as it flows between a host and a device.
- **USB reverse engineering and security research.** The LUNA toolkit will serve as an ideal backend for tools like FaceDancer; allowing easily emulation and rapid prototyping of compliant and non-compliant USB devices.

More detail on these features is covered in the source, and in the remainder of this documentation.

CHAPTER 2

Status & Support

The LUNA library is a work in progress; but many of its features are usable enough for inclusion in your own designs. More testing of our work – and more feedback – is always appreciated!

2.1 Support for Device Mode

Feature		Status
USB Communications	high-/full-speed with `UTMI` PHY	complete, needs testing
	high-/full-speed with `ULPI` PHY	feature complete
	full-speed using raw gpio / pull resistors	feature complete
	super-speed using PIPE PHY	basic support complete; still experimental
	super-speed using SerDes PHY	in progress
	low speed, via ULPI/UTMI PHY	untested
	low speed, using raw gpio / pull resistors	unsupported, currently
Control Transfers / Endpoints	user-defined	feature complete
	fully-gateware-implemented, with user vendor request handler support	complete, could use improvements
	CPU interface	working; needs more interfaces & examples
Bulk Transfers / Endpoints	user-defined	feature complete
	`IN` stream helpers	feature complete
	`OUT` stream helpers	feature complete
	CPU interface	working; needs more interfaces & examples
Interrupt Transfers / Endpoints	user-defined	feature complete
	status-to-host helper	complete, needs testing
	status-from-host helper	planned
	CPU interface	working; needs more interfaces & examples
Isochronous Transfers / Endpoints	user-defined	planned
	`IN` transfer helpers	complete; needs examples and testing
	`OUT` transfer helpers	planned
	CPU interface	planned
USB Analysis	basic analysis	basic analysis working, in progress
	full analysis support	planned

2.2 Support for Host Mode

The LUNA library currently does not provide any support for operating as a USB host; though the low-level USB communications interfaces have been designed to allow for eventual host support. Host support is not currently a priority, but contributions are welcome.

2.3 “Reference” Boards

The LUNA library is intended to work on any FPGA with sufficient fabric performance and resources; but testing is only performed on a collection of reference boards.

Board	FPGA Family	PHY	Status
LUNA Hardware	ECP5	ULPI x3 (USB3343)	Fully Supported
OpenVizsla USB Analyzer	Spartan 6	ULPI (USB3343)	Fully Supported
LambdaConcept ECPIX-5	ECP5	ULPI (USB3300), SerDes PHY	High-Speed Fully Supported / Super-Speed In Progress
TinyFPGA Ex	ECP5	SerDes PHY	Planned Super-Speed Device Mode
Logicbone	ECP5	SerDes PHY	Full-Speed Fully Supported / Super-Speed In Progress
Daisho	Cyclone IV	PIPE (TUSB1310A)	Planned Super-Speed Device Mode
PHYWhisperer-USB	Spartan 7	UTMI	Planned Device Mode Support
LambdaConcept USB2Sniffer	Artix 7	ULPI x2 (USB3300)	Fully Supported
OrangeCrab	ECP5	no hardware PHY	Full-Speed/Device Mode Support
ULX3S	ECP5	no hardware PHY	Full-Speed/Device Mode Support
Fomu PVT/Hacker	iCE40 UP	no hardware PHY	Full-Speed/Device Mode Support
Fomu EVT3	iCE40 UP	no hardware PHY	Full-Speed/Device Mode Support
iCEBreaker Bitsy	iCE40 UP	no hardware PHY	Full-Speed/Device Mode Support
Glasgow	iCE40 HX	no hardware PHY	Planned Full-Speed Support
TinyFPGA Bx	iCE40 LP	no hardware PHY	Full-Speed/Device Mode Support
Digilent Nexys Video (SS with add-on board)	Artix 7	FMC for PIPE (TUSB1310A) add-on boards	Super-Speed Fully Supported
Digilent Genesys2 (SS with add-on board)	Kintex 7	ULPI (TUSB1210), FMC for PIPE (TUSB1310A) add-on boards	High/Super-Speed Fully Supported

CHAPTER 3

Getting Started

3.1 Setting up a Build Environment

This guide highlights the installation / setup process for the `luna` gateware library. It focuses on getting the Python module (and prerequisites) up and running.

3.1.1 Prerequisites

- Python 3.7, 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. All of these tools must be built from `master`.
- A working installation of nMigen. Note that only the official toolchain from *@nmigen <https://github.com/nmigen>* is supported; the *@m-labs <https://github.com/m-labs>* derivative is not.

3.1.2 Installation

Currently, the LUNA library is considered a "work-in-progress"; and thus it's assumed you'll want to use a local copy of LUNA for development.

The easiest way to set this up is to install the distribution in a virtual environment. From the root of the repository:

```
# Pull down poetry, our build system.
pip3 install poetry --user

# Install a copy of our local tools into our virtualenv.
poetry install
```

If you want to install LUNA to your machine globally (not recommended), you can do so using the following single command:

```
# Create a LUNA package, and install it.
pip3 install . --user
```

3.1.3 Testing

The easiest way to test your installation is to build one of the test applets. These applets are just Python scripts that construct and program gateware using nMigen; so they can be run like any other script:

```
# With GSG or self-built LUNA hardware connected; we can run the full test,
# and test both our installation and the attached hardware.
poetry run applets/interactive-test.py

# Without LUNA hardware connected, we'll only build the applet, to exercise
# our toolchain.
poetry run applets/interactive-test.py --dry-run
```

3.1.4 The `apollo` utility.

The `luna` distribution depends on `apollo`, which includes a 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 comms.

```
$ apollo
usage: apollo [-h] command: [[argument]] [[value]]

Utility for LUNA development via an onboard Debug Controller.

positional arguments:
  command:    info       -- Prints information about any connected LUNA-compatible␣
↪boards
              configure  -- Uploads a bitstream to the device's FPGA over JTAG.
              erase      -- Clears the attached board's configuration flash.
              program    -- Programs the target bitstream onto the attached FPGA.
              jtag-scan  -- Prints information about devices on the onboard JTAG␣
↪chain.
              flash-scan -- Attempts to detect any attached configuration flashes.
              svf        -- Plays a given SVF file over JTAG.
              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.
  [argument]  the argument to the given command; often a filename
  [value]     the value to a register write command
```

To have easy access to the `apollo` command, you'll need to ensure that your python binary directory is in your `PATH`. For macOS/Linux, this often means adding `~/.local/bin` to your `PATH`.

CHAPTER 4

LUNA On Your Own Hardware

The LUNA stack is designed to be easy to use on your own FPGA hardware – if you can already run nMigen designs on your board, all you'll need is to set up some I/O definitions and some clock domains.

The exact platform requirements depend on how you'll perform USB interfacing, and are detailed below.

4.1 High-Speed via a ULPI PHY

Using a ULPI PHY is relatively straightforward; and typically requires no hardware beyond the ULPI PHY. LUNA works with both designs that receive their `usb`-domain clocks from the PHY (typical) and designs that provide a 60MHz clock to their PHY.

The following clock domains are required:

Domain Name	Frequency	Description
`usb`	60 MHz	Core clock for the PHY's clock domain. *Can be provided to the FPGA by the PHY, or provided to the PHY by the FPGA. See below.*

An I/O resource with the following subsignals is required:

Subsignal Name	Width	Direction	Description
`clk`	1	input *or* output	The ULPI bus clock. Should be configured as an input if the PHY is providing our clock (typical), or as an output if the FPGA will provide the clock to the PHY.
`data`	8	bidirectional	The bidirectional data bus.
`dir`	1	input	The ULPI *direction* signal.
`nxt`	1	input	The ULPI *next* signal.
`stp`	1	output	The ULPI *stop* signal.
`rst`	1	output	The ULPI *reset* signal. The gateware asserts this signal when the PHY should be reset; if the PHY requires an active-low reset, this can be inverted with `PinsN`.

An example resource might look like:

```
# Targeting the USB3300 PHY, which provides our clock.
Resource("ulpi", 0,
        Subsignal("data",  Pins(data_sites,  dir="io")),
        Subsignal("clk",   Pins(clk_site,    dir="i" )),
        Subsignal("dir",   Pins(dir_site,    dir="i" )),
        Subsignal("nxt",   Pins(nxt_site,    dir="i" )),
        Subsignal("stp",   Pins(stp_site,    dir="o" )),
        Subsignal("rst",   Pins(reset_site,  dir="o" )),
        Attrs(IO_TYPE="LVCMOS33")
    )
```

4.2 Full-Speed using FPGA I/O

LUNA provides a *gateware PHY* that enables an FPGA to communicate at Full Speed using only FPGA 3V3 I/O and a pull-up resistor. The FPGA must be able to provide stable 48 MHz and 12 MHz clocks.

The following clock domains are required:

Domain Name	Frequency	Description
`usb`	12 MHz	Core clock for USB data. Ticks at the USB bitrate of 12MHz, and drives most of the USB logic.
`usb_io`	48 MHz	Edge clock for the USB I/O. Used at the I/O boundary for clock recovery and NRZI encoding.

An I/O resource with the following subsignals is required:

Subsignal Name	Width	Direction	Description
`d_p`	1	bidirectional	The raw USB D+ line; must be on a 3.3V logic bank.
`d_m`	1	bidirectional	The raw USB D- line; must be on a 3.3V logic bank.
`pullup`	1	output	Control for the USB pull-up resistor; should be connected to D+ via a 1.5k resistor.
`vbus_valid`	1	input	*Optional*. If provided, this signal will be used for VBUS detection logic; should be asserted whenever VBUS is present. Many devices are "bus-powered" (receive their power from USB), and thus have no need for VBUS detection, in which case this signal can be omitted.

An example resource might look like:

```
Resource("usb", 0,
    Subsignal("d_p",    Pins("A4")),
    Subsignal("d_n",    Pins("A2")),
    Subsignal("pullup", Pins("D5", dir="o")),
    Attrs(IO_STANDARD="SB_LVCMOS"),
),
```

CHAPTER 5

Core USB 2.0 Device Gateware

The *LUNA* gateware library provides a flexible base *USB Device* model, which is designed to provide the basis for creating both application-specific and general-purpose USB hardware.

USB devices are created using two core components:

- A `USBDevice` instance, which provides hardware that handles low-level USB communications, and which is designed to be applicable to all devices; and
- One or more *endpoint interfaces*, which handle high-level USB communications – and provide the logic the tailors the device to its intended application.

The `USBDevice` communicates with low-level transciever hardware via the FPGA-friendly *USB Transceiver Macrocell Interface* (UTMI). Translators can be used to transparently adapt the FPGA interface to other common bus formats; including the common ULPI low-pin-count variant of UTMI.

Fig. 1: The overall architecture of a LUNA USB 2.0 device, highlighting the `USBDevice` components, their connections to the *endpoint interfaces*, and optional *bus translator*.

5.1 Conceptual Components

The `USBDevice` class contains the low-level communications hardware necessary to implement a USB device; including hardware for maintaining device state, detecting events, reading data from the host, and generating responses.

5.1.1 Token Detector

The *Token Detector* detects *token packets* from the host; and is responsible for:

- Detecting *start of frame* packets, which are used to maintain consistent timing across USB devices.
- Detecting the start of USB *transactions*.
- Identifying the *device* and *endpoint* to which each transaction is addressed.

As each USB transaction starts with a token packet; it is implicitly the Token Detector's responsiblity to notify endpoint interfaces of imminent incoming data (`OUT` transactions) and requests for data (`IN` transactions).

5.1.2 Handshake Detector

The *Handshake Detector* detects *handshake packets* from the host; and is responsible for identifying the host's response to packets from the device – indicating whether the host successfully received a packet sent from the device.

5.1.3 Data Packet Receiver

The *Data Packet Receiver* is responsible for receiving data packets from the device – including the payloads of both `OUT` and `SETUP` transactions – and translating them to a simple data stream.

The Data Receiver handles error detection; and thus validates the checksums of each packet using the Data CRC Unit.

5.1.4 Device State Manager

The *Device State Manager* is responsible for storing global device state – primarily, the device's current *address* and *configuration*. The device state manager accepts changes to the device's address/configuration from each endpoint interface; and automatically resets the relevant parameters when a USB reset is received.

5.1.5 Handshake Generator

The *Handshake Generator* provides a simple, strobe-based interface that allows endpoints to easily emit handshake packets – allowing the device to acknowledge packets (ACK), issue stalls (STALL) , and to rate limit communications (NAK/NYET).

5.1.6 Data Packet Transmitter

The *Data Packet Generator* is responsible for generating outgoing USB packets from simple data streams; including emitting data packet IDs, sending data, and appending data CRCs. This class automatically appends the required data CRC-16s.

5.1.7 Data CRC Unit

The *Data CRC Unit* is shared among the packet receiver and packet generator; and handles computing the CRC-16 for USB data streams.

5.1.8 Interpacket Timer

The *Interpacket Timer* is responsible for maintaining maximum and minimum interpacket delays; ensuring that the device can correctly provide bus turnover times; and knows the window in which handshake packets are expected to arrive.

5.1.9 Reset/Suspend Sequencer

The *Reset/Suspend Sequencer* is responsible for detecing USB reset and suspend events; and where applicable, participating in the USB reset protocol's high-speed detection handshake.

The sequencer:

- Detects USB resets; and communicates to the Device State Manager that it should return the device to an unaddressed, un-configured state.
- Performs the *high speed detection handshake*, which allows the device to switch to High Speed operation; and thus is necessary for the device to operate at high speed.
- Manages the high-speed terminations; as part of the reset-handshake and suspend protocols.
- Detects the periods of inactivity that indicate the device is being suspended; and automatically disengages high-speed terminations while the device is in suspend.

5.2 `usb2.device` Components

5.3 `usb2.packet` Components

5.4 `usb2.reset` Components

CHAPTER 6

Gateware Endpoint Interfaces

The LUNA architecture separates gateware into two distinct groups: the *core device*, responsible for the low-level communications common to all devices, and *endpoint interfaces*, which perform high-level communications, and which are often responsible for tailoring each device for its intended application:

Every useful LUNA device features at least one endpoint interface capable of at least handling enumeration. Many devices will provide multiple endpoint interfaces – often one for each endpoint – but this is not a requirement. Incoming token, data, and handshake packets are routed to all endpoint interfaces; it is up to each endpoint interface to decide which packets to respond to.

> *Note: terms like "interface" are overloaded: the single term "interface" can refer both to hardware interfaces and to the USB concept of an Interface. The "interface" in "endpoint interface" is an instance of the former; they are conceptually distinct from USB interfaces. To reduce conflation, we'll use the full phrase "endpoint interface" in this document.*

As a single endpoint interface may handle packets for multiple endpoints; it is entirely possible to have a device that talks on multiple endpoints, but which uses only one endpoint interface.

6.1 Exclusivity

A LUNA `USBDevice` performs no arbitration – if two endpoint interfaces attempt to transmit at the same time, the result is undefined; and often will result in undesirable output. Accordingly, it's important to ensure a "clear delineation of responsibility" across endpoint interfaces. This is often accomplished by ensuring only one endpoint interface handles a given endpoint or request type.

6.2 `usb2.endpoint` Components

6.3 Provided Endpoint Interfaces

The LUNA library ships with a few provided endpoint interfaces. These include:

- The `USBControlEndpoint`, which provides gateware that facilitates handling USB control requests. To handle requests via this endpoint, the user attaches one or more *request handlers interfaces*; which are documented in their own section.
- The `FIFOInterface` classes, which implement simple, FIFO-based software interfaces. These lightweight interfaces are meant to allow simple CPU control over one or more endpoints. These are based off of the ValentyUSB `eptri` interface; and will eventually be binary-compatible with existing `eptri` code.

6.4 `usb2.control` Components

6.5 `usb2.interfaces.eptri` Components

6.6 Bulk Endpoint Helpers / `usb2.endpoints.stream` Components

6.7 Interrupt Endpoint Helpers / `usb2.endpoints.status` Components

CHAPTER 7

Self-made Hardware Bringup

This guide is intended to help you bring up a LUNA 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.

7.1 Prerequisites

- A LUNA board with a populated *Debug Controller* microprocessor. This is the SAMD microcontroller located in the Debug section at the bottom of the board.
- 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.

7.2 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 `luna` tools, and run through the self-test procedures to validate that your board is working.

7.3 Build/upload Saturn-V

The "recovery mode (RVM)" bootloader for LUNA boards is named *Saturn-V*; as it's the first stage in "getting to LUNA". The bootloader is located in [in its own repository](https://github.com/greatscottgadgets/saturn-v).

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 LUNA 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. 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 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`:

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

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

7.3.1 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 `4KiB` 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 LUNA.

7.4 Build/upload Apollo

The next bringup step is to upload the *Apollo* Debug Controller firmware, which will provide an easy way to interface with the board's FPGA and any gateware running on it. The Apollo source is located [in its own repository](https://github.com/greatscottgadgets/apollo).

You can clone the bootloader using *git*:

```
$ git clone 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 the hardware revision using the `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 `r0.2` would have a `BOARD_REVISION_MAJOR=0` and a `BOARD_REVISION_MINOR=2`. If your board's revision ends in a `+`, do not include it in the revision number.

An example invocation for a `r0.2` board might be:

```
$ make BOARD_REVISION_MAJOR=0 BOARD_REVISION_MINOR=2 dfu
```

Once programming is complete, only LED `E` should be blinking; indicating that the Apollo firmware is idle.

7.5 Running Self-Tests

The final step of bringup is to validate the functionality of your hardware. This is most easily accomplished by running LUNA's interactive self-test applet.

Before you can run the applet, you'll need to have a working `luna` development environment. See [[Setting up the development environment]] to get your environment set up.

Next, we can check to make sure your LUNA board is recognized by the LUNA toolchain. Running the `luna-dev info` command will list any detected devices:

```
$ luna-dev info
Detected a LUNA device!
    Hardware: LUNA r0.2
    Serial number: <snip>
```

Once you've validated connectivity, you're ready to try running the `interactive-test` applet. From the root of the repository:

```
$ python3 applets/interactive-test.py
```

7.6 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 LUNA is was pulled down without its submodules. You can pull down the relevant submodules using `git`:

```
$ git submodule update --init --recursive
```

Issue: the luna-dev info command doesn't see a connected board.

On Linux, this can be caused by a permissions issue. Check first for the presence of your device using `lsusb`; if you see a device with the VID/PID `1d50:615c`, your board is present – and you likely have a permissions issue. You'll likely need to install permission-granting udev rules.

CHAPTER 8

Generated indices

- genindex
- modindex
- search