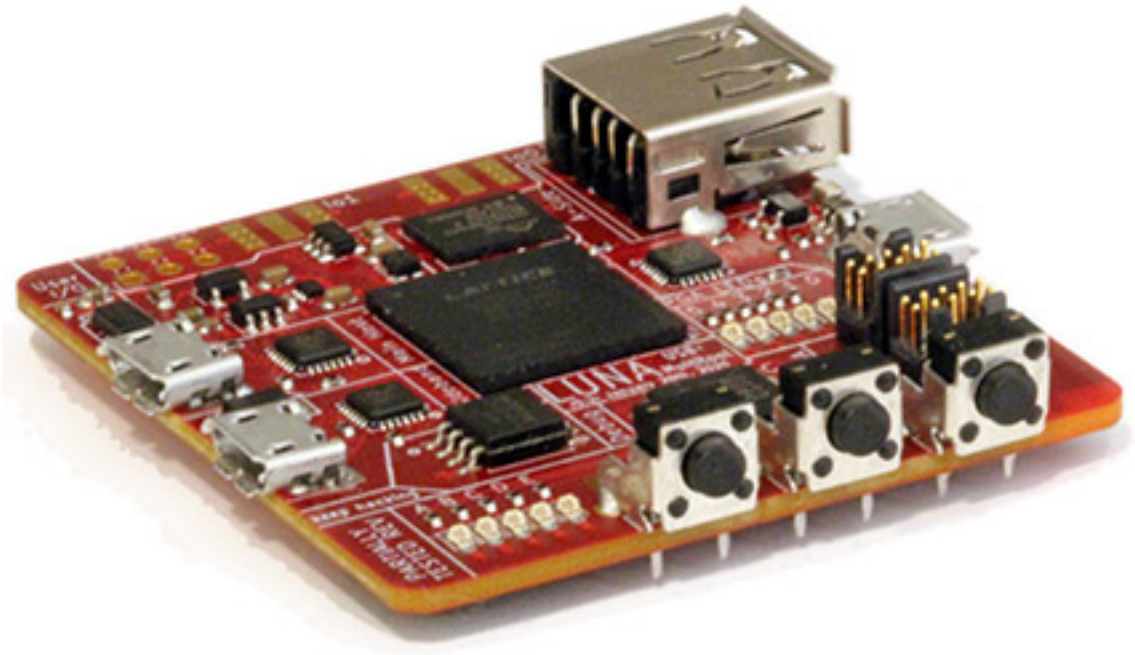
LUNA

Katherine J. Temkin

Apr 29, 2024

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.

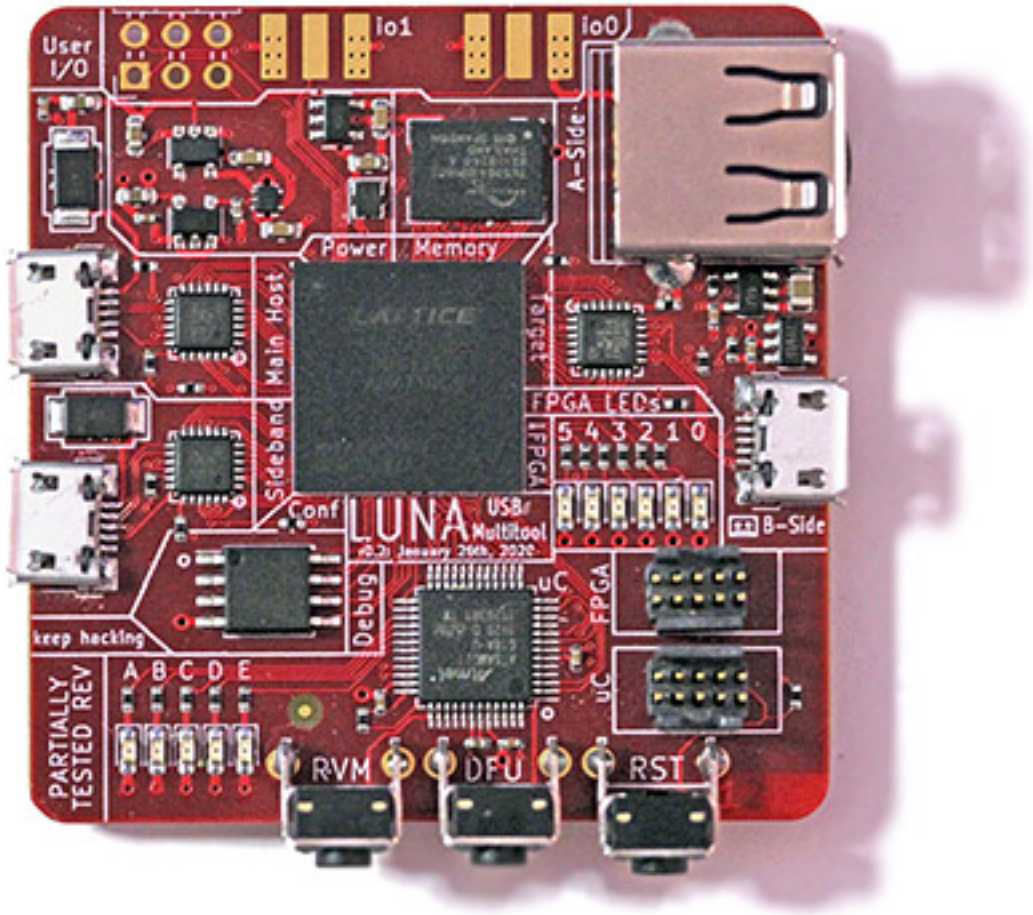
CHAPTER

ONE

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 Amaranth 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 [luna-soc](https://github.com/greatscottgadgets/luna-soc) 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

TWO

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

THREE

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 Amaranth HDL.

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 your working environment. From the root of the repository:

```
# Install a copy of our local tools.
pip install .

# Alternatively: install all dependencies,
# including optional development packages (required for running applets and examples).
pip install .[dev]
```

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.
pip 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 Amaranth HDL; 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.
python applets/interactive-test.py

# Without LUNA hardware connected, we'll only build the applet, to exercise
# our toolchain.
python 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

FOUR

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 Amaranth 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:

Sub-signal 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:

Sub-signal Name	Width	Direction	Description
`d_p`	1	bidirectional	The raw USB D+ line; must be on a 3.3V logic bank.
`d_n`	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_vali`	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

FIVE

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

Contains the organizing hardware used to add USB Device functionality to your own designs; including the core *USBDevice* class.

class luna.gateware.usb.usb2.device.**USBDevice**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: `Elaboratable`

Core gateware common to all LUNA USB2 devices.

The `USBDevice` module 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.

This class can be instantiated directly, and used to build a USB device, or can be subclassed to create custom device types.

To configure a `USBDevice` from a CPU or other wishbone master, see `USBDeviceController`; which can easily be attached using its *attach* method.

Parameters

- **bus** (*[UTMI interface, ULPI Interface]*) – The UTMI or ULPI PHY connection to be used for communications.
- **handle_clocking** (*bool, Optional*) – True iff we should attempt to connect up the *usb* clock domain to the PHY automatically based on the clk signals's I/O direction. This option may not work for non-simple connections; in which case you will need to connect the clock signal yourself.

connect

Held high to keep the current USB device connected; or held low to disconnect.

Type
Signal(), input

low_speed_only

If high, the device will operate at low speed.

Type
Signal(), input

full_speed_only

If high, the device will be prohibited from operating at high speed.

Type
Signal(), input

frame_number

The current USB frame number.

Type
Signal(11), output

microframe_number

The current USB microframe number. Always 0 on non-HS connections.

Type
Signal(3), output

sof_detected

Pulses for one cycle each time a SOF is detected; and thus our frame number has changed.

Type
Signal(), output

new_frame

Strobe that indicates a new frame (not microframe) is detected.

Type
Signal(), output

reset_detected

Asserted when the USB device receives a bus reset.

Type
Signal(), output

 # State signals.

suspended

High when the device is in USB suspend. This can be (and by the spec must be) used to trigger the device to enter lower-power states.

Type
Signal(), output

tx_activity_led

Signal that can be used to drive an activity LED for TX.

Type
Signal(), output

rx_activity_led

Signal that can be used to drive an activity LED for RX.

Type
Signal(), output

add_control_endpoint()

Adds a basic control endpoint to the device.

Does not add any request handlers. If you want standard request handlers; *add_standard_control_endpoint* automatically adds standard request handlers.

Return type
Returns the endpoint object for the control endpoint.

`add_endpoint`(*endpoint*)

Adds an endpoint interface to the device.

Parameters
endpoint (*Elaborateable*) – The endpoint interface to be added. Can be any piece of gateware with a `EndpointInterface` attribute called `interface`.

`add_standard_control_endpoint`(*descriptors: DeviceDescriptorCollection*, ***kwargs*)

Adds a control endpoint with standard request handlers to the device.

Parameters will be passed on to StandardRequestHandler.

5.2.1 Return value

The endpoint object created.

5.3 `usb2.packet` Components

Contains the gatware module necessary to interpret and generate low-level USB packets.

`class luna.gateware.usb.usb2.packet.DataCRCInterface`

Bases: `Record`

Record providing an interface to a USB CRC-16 generator.

start

Strobe that indicates that a new CRC computation should be started.

Type
Signal(), input to CRC generator

crc

The current CRC-16 value; updated with each sent or received byte.

Type
Signal(), output from CRC generator

`class luna.gateware.usb.usb2.packet.HandshakeExchangeInterface`(*, *is_detector*)

Bases: `Record`

Record that carries handshakes detected -or- generated between modules.

ack

When connected to a generator, pulsing this strobe will trigger generating of an ACK. When connected to a detector, this strobe will be pulsed when an ACK is detected from the host.

Type
Signal()

nak

When connected to a generator, pulsing this strobe will trigger generating of an NAK. When connected to a detector, this strobe will be pulsed when an NAK is detected from the host.

Type
Signal()

stall

When connected to a generator, pulsing this strobe will trigger generation of a STALL. Unused in a detector, currently.

Type
Signal()

nyet

When connected to a generator, pulsing this strobe will trigger generation of a NYET. Unused in a detector, currently.

Type
Signal()

Parameters
is_detector (*bool*) – If true, this will be considered an interface to a detector that identifies handshakes. Otherwise, this will be considered an interface to a generator that accepts handshake requests.

class luna.gateware.usb.usb2.packet.**InterpacketTimerInterface**

Bases: Record

Record providing an interface to our interpacket timer.

See [USB2.0: 7.1.18] and the USBInterpacketTimer gateware for more information.

start

Strobe that indicates when the timer should be started. Usually started at the end of an Rx or Tx event.

Type
Signal(), input to timer

tx_allowed

Strobe that goes high when it's safe to transmit after an Rx event.

Type
Signal(), output from timer

tx_timeout

Strobe that goes high when the transmit-after-receive window has passed.

Type
Signal(), output from timer

rx_timeout

Strobe that goes high when the receive-after-transmit window has passed.

Type
Signal(), output from timer

attach(**subordinates*)

Attaches subordinate interfaces to the given timer interface.

Parameters
subordinates ([InterpacketTimerInterface, *Signal*]) – Each *InterpacketTimerInterface* is provided will be fully connected to a given timer interface. Each Signal provided will be interpreted as a timer reset, and added to the list of all resets.

class luna.gateware.usb.usb2.packet.**TokenDetectorInterface**

Bases: Record

Record providing an interface to a USB token detector.

pid

The Packet ID of the most recent token.

Type
Signal(4), detector output

address

The address associated with the relevant token.

Type
Signal(7), detector output

endpoint

The endpoint indicated by the most recent token.

Type
Signal(4), detector output

new_token

Strobe asserted for a single cycle when a new token packet has been received.

Type
Signal(), detector output

ready_for_response

Strobe asserted for a single cycle one inter-packet delay after a token packet is complete. Indicates when the token packet can be responded to.

Type
Signal(), detector output

frame

The current USB frame number.

Type
Signal(11), detector output

new_frame

Strobe asserted for a single cycle when a new SOF has been received.

Type
Signal(), detector output

is_in

High iff the current token is an IN.

Type
Signal(), detector output

is_out

High iff the current token is an OUT.

Type
Signal(), detector output

is_setup

High iff the current token is a SETUP.

Type
Signal(), detector output

is_ping

High iff the current token is a PING.

Type
Signal(), detector output

class luna.gateware.usb.usb2.packet.**USBDataPacketCRC**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: `Elaboratable`

Gateware that computes a running CRC-16.

By default, this module has no connections to the modules that use it.

These are added using *add_interface*; this module supports an arbitrary number of connection interfaces; see *add_interface()* for restrictions.

rx_data

Receive data input; can be carried directly from a UTMI interface.

Type
Signal(8), input

rx_valid

Receive validity signal; can be carried directly from a UTMI interface.

Type
Signal(), input

tx_data

Transmit data input; can be carried directly from a UTMI interface.

Type
Signal(8), input

tx_valid

When high, the *tx_data* input is used to update the CRC.

Type
Signal(), input

Parameters
initial_value (*[int, Const]*) – The initial value of the CRC shift register; the USB default is used if not provided.

add_interface(*interface:* DataCRCInterface)

Adds an interface to the CRC generator module.

Each interface can reset the CRC; and can read the current CRC value. No arbitration is performed; it's assumed that no more than one interface will be computing a running CRC at at time.

Parameters
interface (DataCRCInterface) – The interface to be added; accepts control signals from other modules, and brings CRC output to them. This method can be called multiple times to generate multiplpe CRCs.

class luna.gateware.usb.usb2.packet.**USBDataPacketDeserializer**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Gateware that captures USB data packet contents and parallelizes them.

data_crc

Connection to the CRC generator.

Type
DataCRCInterface

new_packet

Strobe that pulses high for a single cycle when a new packet is delivered.

Type
Signal(), output

packet_id

The packet ID of the captured PID.

Type
Signal(4), output

packet

Packet data for a the most recently received packet.

Type
Signal(max_packet_size), output

length

The length of the packet data presented on the packet[] output.

Type
Signal(range(0, max_packet_length +1)), output

Parameters

- **utmi** (*UTMIInterface, or equivalent*) – The UTMI bus to observe.
- **max_packet_size** (*int*) – The maximum packet (payload) size to be deserialized, in bytes.
- **create_crc_generator** (*bool*) – If True, a submodule CRC generator will be created. Excellent for testing.

class luna.gateware.usb.usb2.packet.**USBDataPacketGenerator**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Module that converts a FIFO-style stream into a USB data packet.

Handles steps such as PID generation and CRC-16 injection.

As a special case, if the stream pulses *last* (with valid=1) without pulsing *first*, we'll send a zero-length packet.

data_pid

The data packet number to use. The potential PIDS are: 0 = DATA0, 1 = DATA1, 2 = DATA2, 3 = MDATA; the interface is designed so that most endpoints can tie the MSb to zero and then perform PID toggling by toggling the LSb.

Type
Signal(2), input

crc

Interface to our data CRC generator.

Type
DataCRCInterface

stream

Stream input for the raw data to be transmitted.

Type
USBInStreamInterface

tx

UTMI-subset transmit interface

Type
UTMITransmitInterface

Parameters
standalone (*bool*) – If True, this unit will include its internal CRC generator. Perfect for unit testing or debugging.

class luna.gateware.usb.usb2.packet.**USBDataPacketReceiver**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Gateware that converts received USB data packets into a data-stream packets.

It's important to note that packet payloads are mostly directly carried over from UTMI. Since USB data is received -prior- to its CRC, one cannot know if a packet is valid until after it has been compeltely received. As a result, this interface will generate data of unknown validity, followed by a strobe on either *packet_complete* or *crc_mismatch*. The receiving interface must be prepared to handle *crc_mismatch* by discarding the received data.

data_crc

Connection to the CRC generator.

Type
DataCRCInterface

timer

Connection to our interpacket timer.

Type
InterpacketTimerInterface

stream

Stream that carries captured packet data.

Type
USBOutDataStream, output

active_pid

The PID of the data currently being received.

Type
Signal(4), output

packet_id

The packet ID of the most recently captured PID. Becomes valid simultaneous to a strobe on *packet_complete* or *crc_mismatch*.

Type
Signal(4), output

packet_complete

Strobe that pulses high when a new packet is delivered with a valid CRC.

Type
Signal(), output

crc_mismatch

Strobe that pulses high when the given packet has a CRC mismatch; and thus the data received this far should be discarded.

Type
Signal(), output

ready_for_response

Strobe that indicates that an inter-packet delay has passed since *packet_complete*, and thus we're now ready to respond with a handshake.

Type
Signal(), output

Parameters

- **utmi** (*UTMIInterface, or equivalent*) – The UTMI bus to observe.
- **max_packet_size** (*int*) – The maximum packet (payload) size to be deserialized, in bytes.
- **standalone** (*bool*) – Debug value. If True, a submodule CRC generator will be created.
- **speed** (*USBSpeed*) – USBSpeed signal or constant that specifies our speed in standalone mode.

class luna.gateware.usb.usb2.packet.**USBHandshakeDetector**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Gateware that detects handshake packets.

detected

Strobes that indicate which handshakes we're detecting.

Type
HandshakeExchangeInterface

Parameters
utmi (*[UTMIInterface, UTMITranslator]*) – The UTMI interface to listen on.

ACK_PID = 2

NAK_PID = 10

NYET_PID = 6

STALL_PID = 14

class luna.gateware.usb.usb2.packet.**USBHandshakeGenerator**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Module that generates handshake packets, on request.

Attributes:

issue_ack: Signal(), input
Pulsed to generate an ACK handshake packet.

issue_nak: Signal(), input
Pulsed to generate a NAK handshake packet.

issue_stall: Signal(), input
Pulsed to generate a STALL handshake.

tx: UTMITransmitInterface
Interface to the relevant UTMI interface.

class luna.gateware.usb.usb2.packet.**USBInterpacketTimer**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Module that tracks inter-packet timings, enforcing spec-mandated packet gaps.

Ports other than *speed* are added dynamically via :method:add_interface`.

speed

The device's current operating speed. Should be a USBSpeed enumeration value – 0 for high, 1 for full, 2 for low.

Type
Signal(2), input

add_interface(*interface:* InterpacketTimerInterface)

Adds a connection to a user of this module.

This module performs no multiplexing; it's assumed only one interface will be active at a time.

Parameters
interface (InterpacketTimerInterface) – The InterPacketTimer interface to add to our module.

class luna.gateware.usb.usb2.packet.**USBTokenDetector**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Gateware that parses token packets and generates relevant events.

interface

The interface that contains token detection events, and information about detected tokens.

Type
TokenDetectorInterface

speed

Carries a USBSpeed constant identifying the device's current operating speed.

Type
Signal(2), input

address

If :parameter:filter_by_address is true, this is an input that filters our event detector so it only reports tokens directed at a given address. If filter_by_address is false, this is an output that contains the address of the most recent token.

Type
Signal(7), input -or- output

Parameters

- **utmi** (*UTMIInterface*) – The UTMI bus to observe.
- **filter_by_address** (*bool*) – If true, this detector will only report events for the address supplied in the address[] field.

`SOF_PID = 5`

`TOKEN_SUFFIX = 1`

5.4 usb2.reset Components

Gateware that handles USB bus resets & speed detection.

class luna.gateware.usb.usb2.reset.**USBResetSequencer**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Gateware that detects reset signaling on the USB bus.

low_speed_only

If set, the device will be forced to operate as a low-speed device.

Type
Signal(), input

prevent_high_speed

If set, the device will be prohibited from entering high-speed states; and will thus act like it's a full speed device (low_speed_only = 0).

Type
Signal(), input

bus_busy

Hold-off signal that indicates that driving the bus should be delayed.

Type
Signal(), input

vbus_connected

Indicates that the device is connected to VBUS. When this is de-asserted, the device will be held in perpetual bus reset, and reset handshaking will be disabled.

Type
Signal(), input

line_state

The UTMI linestate signals; used to read the current state of the USB D+ and D- lines.

Type
Signal(2), input

`bus_reset`

Strobe; pulses high for one cycle when a bus reset is detected. This signal indicates that the device should return to unaddressed, unconfigured, and should not longer be in High Speed mode.

Type
Signal(), output

`suspended`

Held high while the USB device should be in suspend. This technically indicates that the device should drop down to consuming suspend current (<= 2.5mA), but very few devices are compliant with this requirement. Either way, a polite device might reduce its power consumption while in suspend.

Type
Signal(), output

`current_speed`

A USBSpeed value that indicates the current operating speed. Used both to drive our device's knowledge of operating speed and to drive our PHY's speed selection.

Type
Signal(2), output

`operating_mode`

The current UTMI operating mode. Used to select whether we're driving the USB bus directly; or whether we're letting the PHY handle NRZI/bit-stuffing.

Type
Signal(2), output

`termination_select`

Determines the bus termination mode. In LS/FS, this determines the presence of our presence-detect pull-up. In HS mode, this determines whether the USB high-speed termination is present (0), or whether we're in chirp mode (1).

Type
Signal(), output, default=1

`tx`

– Our UTMI transmit interface; used to drive chirp signaling onto the bus.

Type
UTMITransmitInterface, output stream

CHAPTER

SIX

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

Gateware for working with abstract endpoints.

class `luna.gateware.usb.usb2.endpoint.`**EndpointInterface**

Bases: `object`

Interface that connects a USB endpoint module to a USB device.

Many non-control endpoints won't need to use the latter half of this structure; it will be automatically removed by the relevant synthesis tool.

tokenizer

Interface to our TokenDetector; notifies us of USB tokens.

Type
TokenDetectorInterface, to detector

rx

Receive interface for this endpoint.

Type
USBOutStreamInterface, input stream to endpoint

rx_complete

Strobe that indicates that the concluding rx-stream was valid (CRC check passed).

Type
Signal(), input to endpoint

rx_ready_for_response

Strobe that indicates that we're ready to respond to a complete transmission. Indicates that an interpacket delay has passed after an *rx_complete* strobe.

Type
Signal(), input to endpoint

rx_invalid

Strobe that indicates that the concluding rx-stream was invalid (CRC check failed).

Type
Signal(), input to endpoint

rx_pid_toggle

Value for the data PID toggle; 0 indicates we're receiving a DATA0; 1 indicates Data1.

Type
Signal(), input to endpoint

tx

Transmit interface for this endpoint.

Type
USBInStreamInterface, output stream from endpoint

tx_pid_toggle

Value for the data PID toggle; 0 indicates we'll send DATA0; 1 indicates DATA1. 2 indicates we'll send DATA2, while 3 indicates we'll send DATAM.

Type
Signal(2), output from endpoint

handshakes_in

Carries handshakes detected from the host.

Type
HandshakeExchangeInterface, input to endpoint

handshakes_out

Carries handshakes generate by this endpoint.

Type
HandshakeExchangeInterface, output from endpoint

speed

The device's current operating speed. Should be a USBSpeed enumeration value – 0 for high, 1 for full, 2 for low.

Type
Signal(2), input to endpoint

active_address

Contains the device's current address.

Type
Signal(7), input to endpoint

address_changed

Strobe; pulses high when the device's address should be changed.

Type
Signal(), output from endpoint.

new_address

When *address_changed* is high, this field contains the address that should be adopted.

Type
Signal(7), output from endpoint

active_config

The configuration number of the active configuration.

Type
Signal(8), input to endpoint

config_changed

Strobe; pulses high when the device's configuration should be changed.

Type
Signal(), output from endpoint

new_config

When *config_changed* is high, this field contains the configuration that should be applied.

Type
Signal(8)

timer

Interface to our interpacket timer.

Type
InterpacketTimerInterface

data_crc

Control connection for our data-CRC unit.

Type
DataCRCInterface

class luna.gateware.usb.usb2.endpoint.**USBEndpointMultiplexer**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Multiplexes access to the resources shared between multiple endpoint interfaces.

Interfaces are added using *add_interface*.

shared

The post-multiplexer endpoint interface.

Type
EndpointInterface

add_interface(*interface:* EndpointInterface)

Adds a EndpointInterface to the multiplexer.

Arbitration is not performed; it's expected only one endpoint will be driving the transmit lines at a time.

or_join_interface_signals(*m*, *signal_for_interface*)

Joins together a set of signals on each interface by OR'ing the signals together.

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

Low-level USB transciever gateware – control transfer components.

class luna.gateware.usb.usb2.control.**USBControlEndpoint**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: `Elaboratable`

Gateware that manages control request data progression.

This class is used by creating one or more *request handler* modules; which define how requests are handled. These handlers can be bound using *add_request_handler*.

For convenience, this module can also automatically be populated with a `StandardRequestHandler` via the *add_standard_request_handlers*.

interface

The interface from this endpoint to the core device hardware.

Type
EndpointInterface

Parameters

- **utmi** (*UTMI bus, or equivalent translator*) – The UTMI bus we'll monitor for data. We'll consider this read-only.
- **endpoint_number** (*int, optional*) – The endpoint number for this control interface; defaults to (and almost always should be) zero.

- **standalone** (*bool*) – Debug parameter. If true, this module will operate without external components; i.e. without an internal data-CRC generator, or tokenizer. In this case, tokenizer and timer should be set to None; and will be ignored.

add_request_handler(*request_handler*)

Adds a ControlRequestHandler module to this control endpoint.

No arbitration is performed between request handlers; so it's important that request handlers not overlap in the requests they handle.

add_standard_request_handlers(*descriptors: DeviceDescriptorCollection*, ***kwargs*)

Adds a handlers for the standard USB requests.

This will handle all Standard-type requests; so any additional request handlers must not handle Standard requests.

Parameters will be passed on to StandardRequestHandler.

6.5 usb2.interfaces.eptri Components

6.6 Bulk Endpoint Helpers / usb2.endpoints.stream Components

Endpoint interfaces for working with streams.

The endpoint interfaces in this module provide endpoint interfaces suitable for connecting streams to USB endpoints.

class luna.gateware.usb.usb2.endpoints.stream.**USBMultibyteStreamInEndpoint**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Endpoint interface that transmits a simple data stream to a host.

This interface is suitable for a single bulk or interrupt endpoint.

This variant accepts streams with payload sizes that are a multiple of one byte; data is always transmitted to the host in little-endian byte order.

This endpoint interface will automatically generate ZLPs when a stream packet would end without a short data packet. If the stream's last signal is tied to zero, then a continuous stream of maximum-length-packets will be sent with no inserted ZLPs.

This implementation is double buffered; and can store a single packets worth of data while transmitting a second packet.

stream

Full-featured stream interface that carries the data we'll transmit to the host.

Type
StreamInterface, input stream

interface

Communications link to our USB device.

Type
EndpointInterface

Parameters

- **byte_width** (*int*) – The number of bytes to be accepted at once.
- **endpoint_number** (*int*) – The endpoint number (not address) this endpoint should respond to.
- **max_packet_size** (*int*) – The maximum packet size for this endpoint. Should match the wMaxPacketSize provided in the USB endpoint descriptor.

class luna.gateware.usb.usb2.endpoints.stream.**USBStreamInEndpoint**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: `Elaboratable`

Endpoint interface that transmits a simple data stream to a host.

This interface is suitable for a single bulk or interrupt endpoint.

This endpoint interface will automatically generate ZLPs when a stream packet would end without a short data packet. If the stream's `last` signal is tied to zero, then a continuous stream of maximum-length-packets will be sent with no inserted ZLPs.

The `flush` input may be asserted to to cause all pending data to be transmitted as soon as possible. When `flush` is asserted, packets of varying length will be sent as needed, according to the data available.

This implementation is double buffered; and can store a single packets worth of data while transmitting a second packet.

stream

Full-featured stream interface that carries the data we'll transmit to the host.

Type
StreamInterface, input stream

flush

Assert to cause all pending data to be transmitted as soon as possible.

Type
Signal(), input

discard

Assert to cause all pending data to be discarded.

Type
Signal(), input

interface

Communications link to our USB device.

Type
EndpointInterface

Parameters

- **endpoint_number** (*int*) – The endpoint number (not address) this endpoint should respond to.
- **max_packet_size** (*int*) – The maximum packet size for this endpoint. Should match the wMaxPacketSize provided in the USB endpoint descriptor.

class luna.gateware.usb.usb2.endpoints.stream.**USBStreamOutEndpoint**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Endpoint interface that receives data from the host, and produces a simple data stream.

This interface is suitable for a single bulk or interrupt endpoint.

stream

Full-featured stream interface that carries the data we've received from the host.

Type
StreamInterface, output stream

interface

Communications link to our USB device.

Type
EndpointInterface

Parameters

- **endpoint_number** (*int*) – The endpoint number (not address) this endpoint should respond to.
- **max_packet_size** (*int*) – The maximum packet size for this endpoint. If this there isn't *max_packet_size* space in the endpoint buffer, this endpoint will NAK (or participate in the PING protocol.)
- **buffer_size** (*int, optional*) – The total amount of data we'll keep in the buffer; typically two max-packet-sizes or more. Defaults to twice the maximum packet size.

6.7 Interrupt Endpoint Helpers / usb2.endpoints.status Components

Endpoint interfaces for providing status updates to the host.

These are mainly meant for use with interrupt endpoints; and allow a host to e.g. repeatedly poll a device for status.

class luna.gateware.usb.usb2.endpoints.status.**USBSignalInEndpoint**(**args*, *src_loc_at=0*, ***kwargs*)

Bases: Elaboratable

Endpoint that transmits the value of a signal to a host whenever polled.

This is intended to be usable to implement a simple interrupt endpoint that polls for a status signal.

signal

The signal to be relayed to the host. This signal's current value will be relayed each time the host polls our endpoint.

Type
Signal(<variable width>), input

interface

Communications link to our USB device.

Type
EndpointInterface

`status_read_complete`

Strobe that pulses high for a single *usb*-domain cycle each time a status read is complete.

Type
Signal(), output

Parameters

- **width** (*int*) – The width of the signal we'll relay up to the host, in bits.
- **endpoint_number** (*int*) – The endpoint number (not address) this endpoint should respond to.
- **endianness** (*str, "big" or "little", optional*) – The endianness with which to send the data. Defaults to little endian.
- **signal_domain** (*str, optional*) – The name of the domain :attr:`signal` is clocked from. If this value is anything other than "usb", the signal will automatically be synchronized to the USB clock domain.

CHAPTER

SEVEN

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. 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.

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. 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
```

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"
```

(continues on next page)

(continued from previous page)

```
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 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 `apollo info` command will list any detected devices:

```
$ apollo 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 ``apollo info`` command doesn't see a connected board.

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

CHAPTER

EIGHT

GENERATED INDICES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

H

I

L

M

N

O

P

R

S

T

U

V