
scaffold Documentation

Release 0.3

Olivier Hériveaux

Sep 11, 2019

Contents:

1	Getting started	3
1.1	Board tour	3
1.2	Connecting the board	4
1.3	Using the Python API	4
2	Platform and DUT sockets	7
3	Kits	9
3.1	Breadboard kit	9
3.2	STM32F2 kits	9
3.3	Smartcard kit	12
4	Python API	15
4.1	Main Scaffold API	15
4.2	STM32 API	21
4.3	ISO7816 API	23
5	Architecture	27
5.1	Communication protocol	27
5.2	FPGA bus	28
5.3	FPGA bus bridge	29
5.4	Output module	33
6	Building and flashing the FPGA bitstream	35
6.1	Prerequisites	35
6.2	Building the bitstream	35
6.3	Flashing the FPGA	35
7	Indices and tables	39
	Python Module Index	41
	Index	43

Scaffold is a FPGA board made for security research on hardware embedded devices. It is capable of communicating with many circuits using standard protocols. It can be easily extended with new protocols by writing new transceivers in VHDL or Verilog language, and integrating it in the proposed scaffold global architecture. When exchanging messages with the devices under tests, Scaffold can generate triggers for chosen commands, spy data on a bus or intercept and traffic.

Scaffold board also embeds special electronics tailored for hardware attacks:

- Sense resistor and analog amplifier for real-time current measurement
- FPGA safety protections against voltage glitch attacks
- Controllable power switches
- Fast power tearing capability, for emergency shutdown of the device under test
- Delay and pulse generators, allowing firing glitches or laser pulses.

An easy-to-use python API is provided to control the board.

1.1 Board tour

- **A:** USB2 link with host computer. Also used to power the board.
- **B:** Main board power switch.
- **C:** Switch the jumper to the right to power the board from an external 5 V power supply and not from the USB.
- **D:** External power supply for the platform socket.
- **E:** External power supply for the DUT socket.
- **F:** Adjustable voltage regulator for the platform socket. When the jumper is set on top position, this power source is not used and the platform socket is powered from the external power supply.

- **G:** Adjustable voltage regulator for the DUT socket. When the jumper is set on top position, this power source is not used and the DUT socket is powered from the external power supply.
- **H:** Adjustable voltage regulator for the I/O bank of the FPGA connected to the platform and DUT sockets. This allows setting the correct voltage depending on the connected device. Supported voltage range goes from 1.5 V up to 3.3 V.
- **I:** FPGA active serial connector for bitstream programming. An USB blaster can be used to update the bit-stream.
- **J:** FPGA reset button. Push if the board enters error state.
- **K:** Switches to select Spy or Intercept mode for each I/O of the FPGA. In intercept mode, the signals of the platform and DUT sockets are not connected anymore and the FPGA can act as a man in the middle circuit.
- **L:** Switches to enable 1 KOhm series protection resistors on the I/Os (at the cost of slew-rate). Shall be enabled when there is a risk to damage the FPGA, with high-voltage glitches for instance.
- **M:** Tearing input. Any positive edge will power-off the DUT immediately and shunt all I/Os to ground.
- **N:** A0, A1, A2, A3 voltage standard selection between 3.3 V or 5 V. Move the jumper to change the voltage of the corresponding I/O.
- **O:** I/Os with voltage translators. 5 V is suited to drive Alphanov TTL 50 Ohm laser sources (other 3.3 V I/Os can't).
- **P:** FPGA I/Os. Maximum voltage is 3.3 V. Those I/Os are connected to the platform and DUT sockets and are usually used to communicate with the target device and generate triggers. SMA connectors are provided for D0 to D6.
- **Q:** Platform socket and power state LED.
- **R:** DUT socket and power state LED.
- **S:** Adjustable shunt resistor for power trace measurement.
- **T:** Output of the analog 11 X amplifier for power trace measurement.

1.2 Connecting the board

Connect the board with a micro-USB cable to your computer. Power-on the board using the power switch. The operating system shall detect the board as a USB-To-Serial device (COMx on Windows, /dev/ttyUSBx on linux). It may be necessary to install FTDI driver for some Windows versions.

1.3 Using the Python API

The file `scaffold.py` is the library which can be used to interact with Scaffold board.

```
from scaffold import Scaffold

# Connect to the board.
# This will open the serial device and check for hardware version
scaffold = Scaffold('/dev/ttyUSB0')

# Configure UART0 for operation
uart = scaffold.uart0
uart.baudrate = 115200
```

(continues on next page)

(continued from previous page)

```
# Connect UART0 signals to board pins
# << operator connects signals. Left operand is the destination, right operand
# is the source (the order is important)
# In this example, D0 and D1 are configured as outputs, D2 is configured as an
# input.
scaffold.d0 << uart.tx
scaffold.d1 << uart.trigger
uart.rx << scaffold.d2

# UART is now ready to use
uart.send('Hello world !')
```


CHAPTER 2

Platform and DUT sockets

The following figure is a simplified drawing for the Scaffold v1.1 socket footprints. It can be used to design custom daughter-boards.

Female HE10 connectors can be used on the daughter-boards. All holes are aligned on a 2.54 mm (100 mil) grid, which is compatible with most of the prototyping breadboards for very cheap daughter-board making.

Some kits for specific applications with Scaffold are available.

3.1 Breadboard kit

The Scaffold breadboard provides solderable test points for all the DUT socket pins. It can be used to mount a setup with any target.

3.2 STM32F2 kits

STM32 kits allows communicating with the embedded ST bootloader of STM32F205 and STM32F427 devices. It is possible to write the Flash memory to load code, and then execute it after reset.

Those QFP64 and QFP100 may suit other STM32 devices. Some breadboard space allows customizing the daughter-board for special needs, such as clock or voltage glitch electronics.

3.2.1 Python API example

The class `scaffold.stm32.STM32` of the Python API provides methods to communicate with the circuit and setup tests very quickly.

```
from scaffold import Scaffold
from scaffold.stm32 import STM32

stm = STM32(Scaffold('/dev/ttyUSB0'))
# Load some code into Flash memory
stm.startup_bootloader()
stm.extended_erase()
stm.write_memory(0x08000000, open('program.bin', 'rb').read())
```

(continues on next page)





(continued from previous page)

```
# Run the program
stm.startup_flash()
```

3.2.2 Example script

An example file in *examples/stm32.py* can be used to load and execute code onto a STM32F2 device.

```
$ python3 stm32.py -d /dev/ttyUSB0 --load program.bin --run
Communication initiated
Product ID: 0x0411
Possible device match: stm32f2xxxx
Get: 310001021121314463738292
Bootloader version: 3.1
Option bytes: ffaa0055ffaa0055ffff0000ffff0000
RDP: no protection
Erasing Flash memory...
Programming...
Verifying...
Flash memory written successfully!
Rebooting from Flash memory...
```

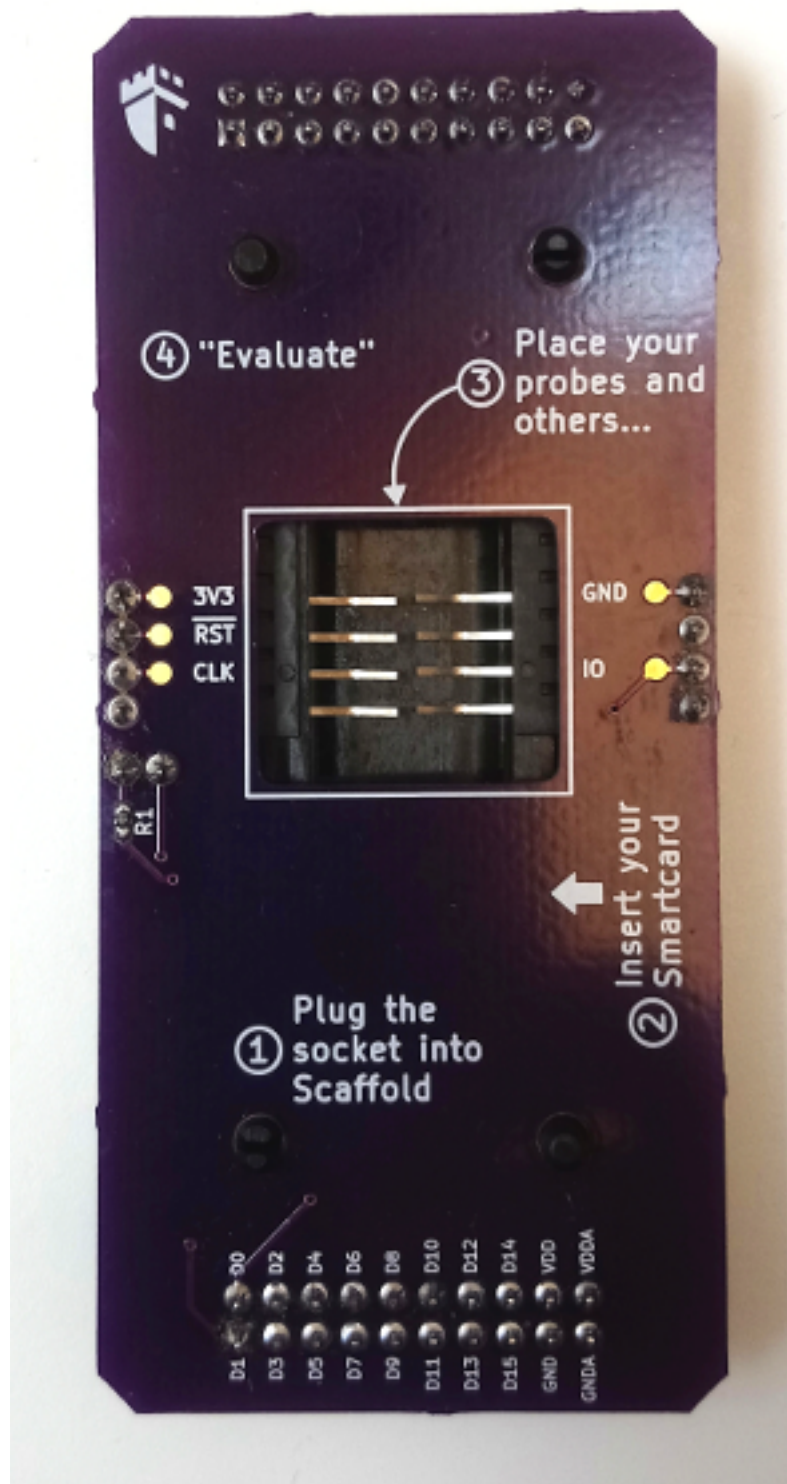
3.3 Smartcard kit

This kit allows communicating with any smartcard using 7816 protocol.

The class `scaffold.iso7816.Smartcard` of the Python API provides methods to communicate with an ISO7816 Smartcard and setup tests very quickly. Currently, only T=0 protocol is supported by the API and it has not been tested extensively yet.

3.3.1 Pinout

D0	I/O. This signal is pulled up with a resistor on the daughter board
D1	nRST
D2	CLK
D3	Socket card presence contactor



4.1 Main Scaffold API

This page documents the main classes and methods of the Scaffold Python API.

Manipulating the attributes of the different modules of a `Scaffold` instance will read or write in the FPGA registers. Some registers may be cached by the Python API, so reading them does not require any communication with the board and thus can be fast.

class `scaffold.Scaffold` (*dev*='/dev/scaffold', *init_ios*=False)

This class connects to a Scaffold board and provides access to all the device parameters and peripherals.

Variables

- **uarts** – list of `scaffold.UART` instance managing UART peripherals.
- **i2cs** – list of `scaffold.I2C` instance managing I2C peripherals.
- **iso7816** – `scaffold.ISO7816` instance managing the ISO7816 peripheral.
- **pgens** – list of four `scaffold.PulseGenerator` instance managing the FPGA pulse generators.
- **power** – `scaffold.Power` instance, enabling control of the power supplies of DUT and platform sockets.
- **leds** – `scaffold.LEDs` instance, managing LEDs brightness and lighting mode.
- **[a0, a1, a2, a3, b0, b1, c0, c1, d0, d1, d2, d3, d4, d5]** – `scaffold.Signal` instances for connecting and controlling the corresponding I/Os of the board.

__init__ (*dev*='/dev/scaffold', *init_ios*=False)

Create Scaffold API instance.

Parameters

- **dev** – If specified, connect to the hardware Scaffold board using the given serial device. If None, call connect method later to establish the communication.

- **init_ios** – True to enable I/Os peripherals initialization. Doing so will set all I/Os to a default state, but it may generate pulses on the I/Os. When set to False, I/Os connections are unchanged during initialization and keep the configuration set by previous sessions.

class `scaffold.Signal` (*parent, path*)

Base class for all connectable signals in Scaffold. Every *Signal* instance has a Scaffold board parent instance which is used to electrically configure the hardware board when two *Signal* are connected together.

`__init__` (*parent, path*)

Parameters

- **parent** – Scaffold instance which the signal belongs to.
- **path** – Signal path string. Uniquely identifies a Scaffold board internal signal. For instance `‘/dev/uart0/tx’`.

`__lshift__` (*other*)

Feed the current signal with another signal.

Parameters **other** – Another *Signal* instance. The other signal must belong to the same *Scaffold* instance.

`__str__` ()

Returns Signal path. For instance `‘/dev/uart0/tx’`.

name

Signal name (last element of the path). For instance `‘tx’`. Read-only.

parent

Parent *Scaffold* board instance. Read-only.

path

Signal path. For instance `‘/dev/uart0/tx’`. Read-only.

class `scaffold.IO` (*parent, path, index, pullable=False*)

Board I/O.

`clear_event` ()

Clear event register.

Warning If an event is received during this call, it may be cleared without being took into account.

event

I/O event register.

Getter Returns 1 if an event has been detected on this input, 0 otherwise.

Setter Writing 0 to clears the event flag. Writing 1 has no effect.

mode

I/O mode. Default is AUTO, but this can be overridden for special applications.

Type *IOMode*

pull

Pull resistor mode. Can only be written if the I/O supports this feature.

Type *Pull*

value

Current IO logical state.

Getter Senses the input pin of the board and return either 0 or 1.

Setter Sets the output to 0, 1 or high-impedance state (None). This will disconnect the I/O from any already connected internal peripheral. Same effect can be achieved using << operator.

class scaffold.IOMode

An enumeration.

AUTO = 0

OPEN_DRAIN = 1

PUSH_ONLY = 2

class scaffold.Pull

An enumeration.

DOWN = 1

NONE = 0

UP = 3

class scaffold.UARTParity

Possible parity modes for UART peripherals.

EVEN = 2

NONE = 0

ODD = 1

class scaffold.UART(*parent, index*)

UART module of Scaffold.

baudrate

Target UART baudrate.

Getter Returns current baudrate, or None if no baudrate has been previously set during current session.

Setter Set target baudrate. If baudrate cannot be reached within 1% accuracy, a RuntimeError is thrown. Reading the baudrate attribute after setting it will return the real effective baudrate.

flush()

Discard all the received bytes in the FIFO.

parity

Parity mode. Disabled by default.

Type *UARTParity*

receive(*n=1*)

Receive *n* bytes from the UART. This function blocks until all bytes have been received or the timeout expires and a TimeoutError is thrown.

reset()

Reset the UART to a default configuration: 9600 bps, no parity, one stop bit, trigger disabled.

transmit(*data, trigger=False*)

Transmit data using the UART.

Parameters

- **data** – Data to be transmitted. bytes or bytearray.
- **trigger** – True or 1 to enable trigger on last byte, False or 0 to disable trigger.

class `scaffold.ISO7816` (*parent*)

ISO7816 peripheral of Scaffold. Does not provide convention or protocol management. See `scaffold.iso7816.Smartcard` for more features.

clock_frequency

Target ISO7816 clock frequency. According to ISO7816-3 specification, minimum frequency is 1 Mhz and maximum frequency is 5 MHz. Scaffold hardware allows going up to 50 Mhz and down to 195312.5 Hz (although this may not work with the smartcard).

Getter Returns current clock frequency, or None if it has not been set previously.

Setter Set clock frequency. If requested frequency cannot be reached within 1% accuracy, a `RuntimeError` is thrown. Reading this attribute after setting it will return the real effective clock frequency.

empty

True if reception FIFO is empty.

etu

ISO7816 ETU parameter. Value must be in range $[1, 2^{11}-1]$. Default ETU is 372.

flush()

Discard all the received bytes in the FIFO.

parity_mode

Parity mode. Standard is Even parity, but it can be changed to odd or forced to a fixed value for testing purposes. :type: `ISO7816ParityMode`

receive (*n=1*)

Receive bytes. This function blocks until all bytes have been received or the timeout expires and a `TimeoutError` is thrown.

Parameters *n* – Number of bytes to be read.

reset_config()

Reset ISO7816 peripheral to its default configuration.

transmit (*data*)

Transmit data.

Parameters *data* (*bytes*) – Data to be transmitted.

trigger_long

Enable or disable long trigger (set on transmission, cleared on reception). When changing this value, wait until transmission buffer is empty.

Type `bool`

trigger_rx

Enable or disable trigger upon reception. :type: `bool`

trigger_tx

Enable or disable trigger upon transmission. :type: `bool`

class `scaffold.I2C` (*parent*, *index*)

I2C module of Scaffold.

clock_stretching

Enable or disable clock stretching support. When clock stretching is enabled, the I2C slave may hold SCL low during a transaction. In this mode, an external pull-up resistor on SCL is required. When clock stretching is disabled, SCL is always controlled by the master and the pull-up resistor is not required.

Type `bool` or `int`.

flush()

Discards all bytes in the transmission/reception FIFO.

frequency

Target I2C clock frequency.

Getter Returns current frequency.

Setter Set target frequency. Effective frequency may be different if target cannot be reached accurately.

raw_transaction (*data*, *read_size*, *trigger=None*)

Executes an I2C transaction. This is a low-level function which does not manage I2C addressing nor read/write mode (those shall already be defined in data parameter).

Parameters

- **data** (*bytes*) – Transmitted bytes. First byte is usually the address of the slave and the R/W bit. If the R/W bit is 0 (write), this parameter shall then contain the bytes to be transmitted, and *read_size* shall be zero.
- **read_size** (*int*) – Number of bytes to be expected from the slave. 0 in case of a write transaction.
- **trigger** (*int or str.*) – Trigger configuration. If *int* and value is 1, trigger is asserted when the transaction starts. If *str*, it may contain the letter ‘a’ and/or ‘b’, where ‘a’ asserts trigger on transaction start and ‘b’ on transaction end.

Raises I2CNackError – If a NACK is received during the transaction.

read (*size*, *address=None*, *trigger=None*)

Perform an I2C read transaction.

Parameters address (*int or None*) – Slave device address. If *None*, *self.address* is used by default. If defined and addressing mode is 7 bits, LSB must be 0 (this is the R/W bit). If defined and addressing mode is 10 bits, bit 8 must be 0.

Returns Bytes from the slave.

Raises I2CNackError – If a NACK is received during the transaction.

reset_config()

Reset the I2C peripheral to a default configuration.

write (*data*, *address=None*, *trigger=None*)

Perform an I2C write transaction.

Parameters address (*int or None*) – Slave device address. If *None*, *self.address* is used by default. If defined and addressing mode is 7 bits, LSB must be 0 (this is the R/W bit). If defined and addressing mode is 10 bits, bit 8 must be 0.

Raises I2CNackError – If a NACK is received during the transaction.

class scaffold.SPI (*parent*, *index*)

SPI peripheral of Scaffold.

frequency

Target SPI clock frequency.

Getter Returns current frequency.

Setter Set target frequency. Effective frequency may be different if target cannot be reached accurately.

Type float

phase

Clock phase. 0 or 1. :type: int

polarity

Clock polarity. 0 or 1. :type: int

transmit (*value*, *size=8*, *trigger=False*, *read=True*)

Performs a SPI transaction to transmit a value and receive data. If a transmission is still pending, this methods waits for the SPI peripheral to be ready.

Parameters

- **value** (*int*) – Value to be transmitted. Less significant bit is transmitted last.
- **size** (*int*) – Number of bits to be transmitted. Minimum is 1, maximum is 32.
- **read** – Set 0 or False to disable received value readout (the method will return None). Default is True, but disabling it will make this command faster if the returned value can be discarded.

Pram trigger 1 or True to enable trigger upon SPI transmission.

Returns Received value.

Return type int

class scaffold.**PulseGenerator** (*parent*, *index*)

Pulse generator module of Scaffold. Usually abbreviated as pgen.

count

Number of pulses to be generated. Minimum value is 1. Maximum value is 2¹⁶.

Type int

delay

Delay before pulse, in seconds.

Type float

fire ()

Manually trigger the pulse generation.

interval

Delay between pulses, in seconds.

Type float

polarity

Pulse polarity. If 0, output is low when idle, and high during pulses. When 1, output is high when idle, and low during pulses.

Type int

width

Pulse width, in seconds.

Type float

class scaffold.**Clock** (*parent*, *index*)

Clock generator module. This peripheral allows generating a clock derived from the FPGA system clock using a clock divisor. A second clock can be generated and enabled during a short period of time to override the first clock, generating clock glitches.

freq_a

Base clock frequency, in Hertz. Only divisors of the system frequency can be set: 50 MHz, 25 MHz, 16.66 MHz, 12.5 MHz...

Type float

freq_b

Glitch clock frequency, in Hertz. Only divisors of the system frequency can be set: 50 MHz, 25 MHz, 16.66 MHz, 12.5 MHz...

Type float

class scaffold.Chain(*parent, index, size*)

Chain trigger module.

rearm()

Reset the chain trigger to initial state.

class scaffold.LEDs(*parent*)

LEDs module of Scaffold.

brightness

LEDs brightness. 0 is the minimum. 1 is the maximum.

Type float

disabled

If set to True, LEDs driver outputs are all disabled.

override

If set to True, LEDs state is the value of the leds_n registers.

reset()

Set module registers to default values.

class scaffold.Power(*parent*)

Controls the platform and DUT sockets power supplies.

all

All power-supplies state. int. Bit 0 corresponds to the DUT power supply. Bit 1 corresponds to the platform power-supply. When a bit is set to 1, the corresponding power supply is enabled. This attribute can be used to control both power supplies simultaneously.

dut

DUT power-supply state. int.

platform

Platform power-supply state. int.

4.2 STM32 API

This API is for STM32 experimental daughter boards.

class scaffold.stm32.STM32(*scaffold*)

Class for instrumenting STM32 devices using Scaffold board and API. The following Scaffold IOs are used:

- D0: STM32 UART MCU RX pin, Scaffold UART TX
- D1: STM32 UART MCU TX pin, Scaffold UART RX
- D2: STM32 NRST pin for reset

- D6: STM32 BOOT0 pin
- D7: STM32 BOOT1 pin

The uart0 peripheral of Scaffold board is used for serial communication with the ST bootloader.

This class can communicate with the ST bootloader via USART1. This allows programming the Flash memory and then execute the loaded code.

`__init__ (scaffold)`

Parameters **scaffold** – An instance of `scaffold.Scaffold` which will be configured and used to communicate with STM32 daughter board.

`assert_device ()`

Raise a RuntimeError is device is unknown (None).

`checksum (data)`

Calculate the checksum of some data, according to the STM32 bootloader protocol.

Parameters **data** – Input bytes.

Returns Checksum byte. This is the XOR of all input bytes.

`command (index)`

Send a command and return the response.

Parameters **index** – Command index.

Returns Response bytes.

`extended_erase ()`

Execute the Extended Erase command to erase all the Flash memory of the device.

`get ()`

Execute the Get command of the bootloader, which returns the version and the supported commands.

`get_id ()`

Execute the Get ID command. The result is interpreted and the class will try to find information if the ID matches a known device.

`go (address, trigger=0)`

Execute the Go command.

Parameters

- **address** – Jump address.
- **trigger** – 1 to enable trigger on command transmission.

`read_memory (address, length, trigger=0)`

Tries to read some memory from the device. If requested size is larger than 256 bytes, many Read Memory commands are sent.

Parameters

- **address** – Memory address to be read.
- **size** – Number of bytes to be read.
- **trigger** – 1 to enable trigger on command transmission.

`read_option_bytes ()`

Read the option bytes of the device. The method `get_id` must have been called previously for device identification.

Returns Memory content of 'option_bytes' section.

readout_protect()

Execute the Readout Unprotect command.

readout_unprotect()

Execute the Readout Unprotect command. If the device is locked, it will perform mass flash erase, which can be very very long.

startup_bootloader()

Power-cycle and reset target device in bootloader mode (boot on System Memory) and initiate serial communication. The byte 0x7f is sent and the response byte 0x79 (ACK) is expected. If the device does not respond, a Timeout exception is thrown by Scaffold. The device will not respond if it is locked in RDP2 state (Readout Protection level 2).

startup_flash()

Power-cycle and reset target device and boot from user Flash memory.

wait_ack(tag=None)

Wait for ACK byte.

Parameters *tag* – Tag which is set when NACKError are thrown. Useful for error diagnostic.

wait_ack_or_nack()

Wait for ACK or NACK byte.

Returns True if ACK has been received, False if NACK has been received.

write_memory(address, data, trigger=0)

Write data to device memory. If target address is Flash memory, this function DOES NOT erase Flash memory prior to writing. If data size is larger than 256 bytes, many Write Memory commands are sent.

Parameters

- **address** – Address.
- **data** – Data to be written. bytes or bytearray.
- **trigger** – 1 to enable trigger on each command transmission.

4.3 ISO7816 API

This API provides support for ISO7816 support with Scaffold.

class `scaffold.iso7816.Smartcard(scaffold)`

Class for smartcard testing with Scaffold board and API. The following IOs are used:

- D0: ISO7816 IO
- D1: ISO7816 nRST
- D2: ISO7816 CLK
- D3: Socket card contactor sense

`scaffold.Scaffold` class has ISO7816 peripheral support, but it is very limited. This class adds full support to ISO7816 by managing ATR, convention conversion, etc.

Variables

- **atr** (*bytes*) – ATR received from card after reset.
- **convention** (`Convention`) – Communication convention between card and terminal. Updated when first byte TS of ATR is received.

- **protocols** (*set*) – Communication protocols found in ATR. This set contains integers, for instance 0 if T=0 is supported, 1 if T=1 is supported...

__init__ (*scaffold*)

Configure a Scaffold board for use with smartcards. :param scaffold: *scaffold.Scaffold* instance.

apdu (*the_apdu*, *trigger=""*)

Send an APDU to the smartcard and retrieve the response.

Parameters

- **the_apdu** (*bytes* or *str*) – APDU to be sent. *str* hexadecimal strings are allowed, but user should consider using the *apdu_str()* method instead.
- **trigger** (*str*) – If 'a' is in this string, trigger is raised after ISO-7816 header is sent, and cleared when the following response byte arrives. If 'b' is in this string, trigger is raised after data field has been transmitted, and cleared when the next response byte is received.

Raises **ValueError** – if APDU data is invalid.

Return bytes Response data, with status word.

apdu_str (*the_apdu*)

Same as *apdu()* function, with *str* argument and return type for convenience.

Parameters **the_apdu** (*str*) – APDU to be sent, as an hexadecimal string.

Return str Response from the card, as a lowercase hexadecimal string without spaces.

card_inserted

True if a card is inserted, False otherwise. Card insertion is detected with a mechanical switch connected to D3 of Scaffold.

find_info ()

Parse the smartcard ATR list database available at http://ludovic.rousseau.free.fr/softwares/pcsc-tools/smartcard_list.txt and try to match the current ATR to retrieve more info about the card.

The database file cannot be embedded in the library because it uses GPL and not LGPL license. On debian systems, this file is provided in the pcsc-tools package.

Returns A list of *str*, where each item is an information line about the card. Return None if the ATR did not match any entry in the database.

inverse_byte (*byte*)

Inverse order and polarity of bits in a byte. Used for ISO7816 inverse convention decoding.

receive (*n*)

Use the ISO7816 peripheral to receive bytes from the smartcard, and apply direct or inverse convention depending on what has been read in the ATR.

Parameters **n** – Number of bytes to be read.

reset ()

Reset the smartcard and retrieve the ATR. If the ATR is retrieved successfully, the attributes *atr* convention and *protocols* are updated.

Returns ATR from the card.

Raises **ProtocolError** – if the ATR is not valid.

class *scaffold.iso7816.Convention*

Possible ISO7816 communication convention. This is given by the first byte of the ATR returned by the card.

DIRECT = 59

```
INVERSE = 63
```

```
class scaffold.iso7816.ProtocolError(message)
```

Exception raised when a protocol error between the terminal and the smartcard occurs.

The following documentation describes in details the architecture of Scaffold. The targeted audience is the developer who wish fixing bugs in the architecture or extend its functionalities. Reading this documentation may help understanding what's under the hood.

5.1 Communication protocol

5.1.1 General architecture

The FPGA has many embedded peripherals. Each peripheral has registers which can be read/written to receive/send data and perform actions. Each register is assigned a unique 16-bits address and is connected to the system data bus.

A bridge controller, inside the FPGA, controls the read and write operations on the system data bus. This controller can be driven by a host computer using the USB link, allowing the host computer to manipulate the registers connected to the system data bus.

5.1.2 Protocol

When connected to a host computer using a USB cable, the board is recognized as a USB-to-Serial device from FTDI manufacturer. Baudrate is 2 Mbits/s, with one stop bit and no parity check.

A very simple protocol is defined to perform read and write operations through the serial device. To perform an operation, the following data must be sent to Scaffold:

Command	1 byte	Mandatory
Address	2 bytes	Mandatory
Polling address	2 bytes	When polling enabled
Polling mask	1 byte	When polling enabled
Polling value	1 byte	When polling enabled
Size	1 byte	When size enabled
Data	N bytes	For write commands

Bit 0 of command byte indicates if this is a read (0) or write (1) command. Bit 1 indicates if size parameter is present (1 to enable size). Bit 2 indicates if polling is requested for this command (1 to enable polling). All other bits must be set to zero, otherwise the command is considered invalid and Scaffold will enter error mode.

For write and read commands, a response is transmitted by the Scaffold board. This response starts by the data bytes (for register read commands) and terminates with a status byte. The status byte shall be equal to the size of the processed data. If a command times out during polling, the returned status byte will be lower than the size of the data.

5.1.3 Register polling

Read or write operations on registers can be conditioned to a given register expected state. When polling is enabled, each read or write operation is performed when the monitored register reaches a given value for some chosen bits. Polling is enabled when bit 2 of command byte is 1. Read or write operation is performed when $(\text{Register and Mask}) = (\text{Value and Mask})$.

Polling can be used for flow control when using a peripheral to process multiple bytes. For instance, when using a SPI peripheral, polling can be used to wait for incoming bytes.

The polled register can be different from the read or written register (two different addresses can be passed in the command parameters: address and polling address).

5.1.4 Polling timeout

Optionally, a timeout can be configured for polling operations. This is particularly useful if receiving bytes from a peripheral is not guaranteed. When the polling times out in a read operation, the remaining expected bytes are sent by the board as zero. When the polling times out in a write operation, the remaining bytes sent to the board are discarded by the bus bridge. The returned acknowledge byte indicates the number of successfully processed bytes and will be lower than the requested size in the command.

The timeout delay can be configured with a special command:

Command 0x08	1 byte	Mandatory
Polling delay value	4 bytes	MSB first

No response is expected after this command. The new delay value will be applied for all the following commands. If the delay is set to zero, then the timeout is disabled (which is the default).

5.2 FPGA bus

The internal global bus connects all the peripherals together. This bus is controlled by the serial bridge which is connected to the host computer with the USB link. The bus can perform only two simple operations:

- Read a byte from a register,
- Write a byte to a register.

The bus works using many signals:

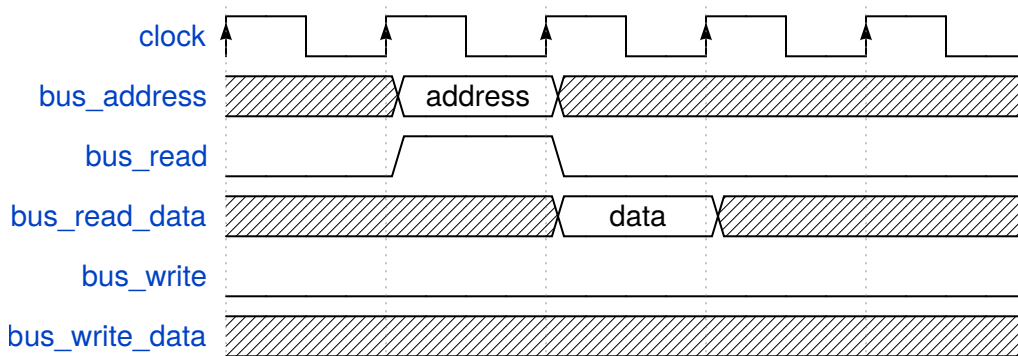
- 16-bits address: bus_address,
- Write assertion signal: bus_write
- 8-bits write data: bus_write_data,
- Read assertion signal: bus_read

- 8-bits read data: `bus_read_data`

Using different data wires for read and write operations makes conflicts between modules impossible. Also, some FPGA devices may not allow internal bidirectional wires.

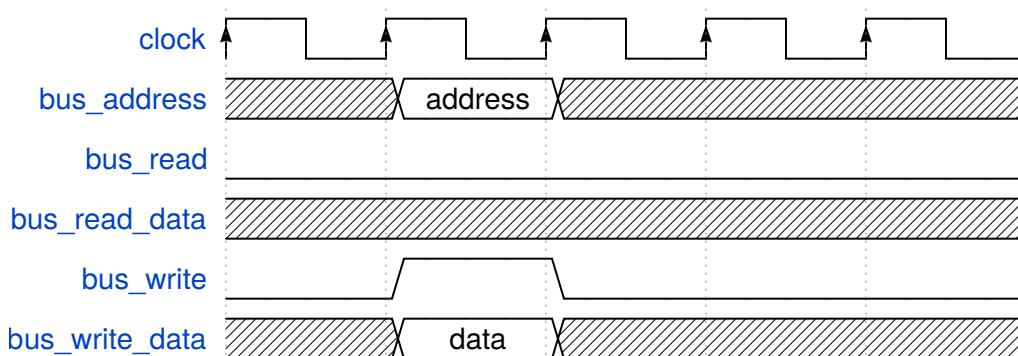
All the signals of the bus are synchronized to the system clock, on rising edges.

5.2.1 Register read cycle



Peripherals connected to the bus must present valid data one clock cycle after the `bus_read` signal is asserted. This allow using FIFO blocks with *read-ahead* option disabled, which is more performant.

5.2.2 Register write cycle



5.3 FPGA bus bridge

The bus bridge allows reading and writing bytes on the internal FPGA bus using commands sent in serial. This bridge is implemented in VHDL using a Finite State Machine. The state machine manages the commands parsing and execution.

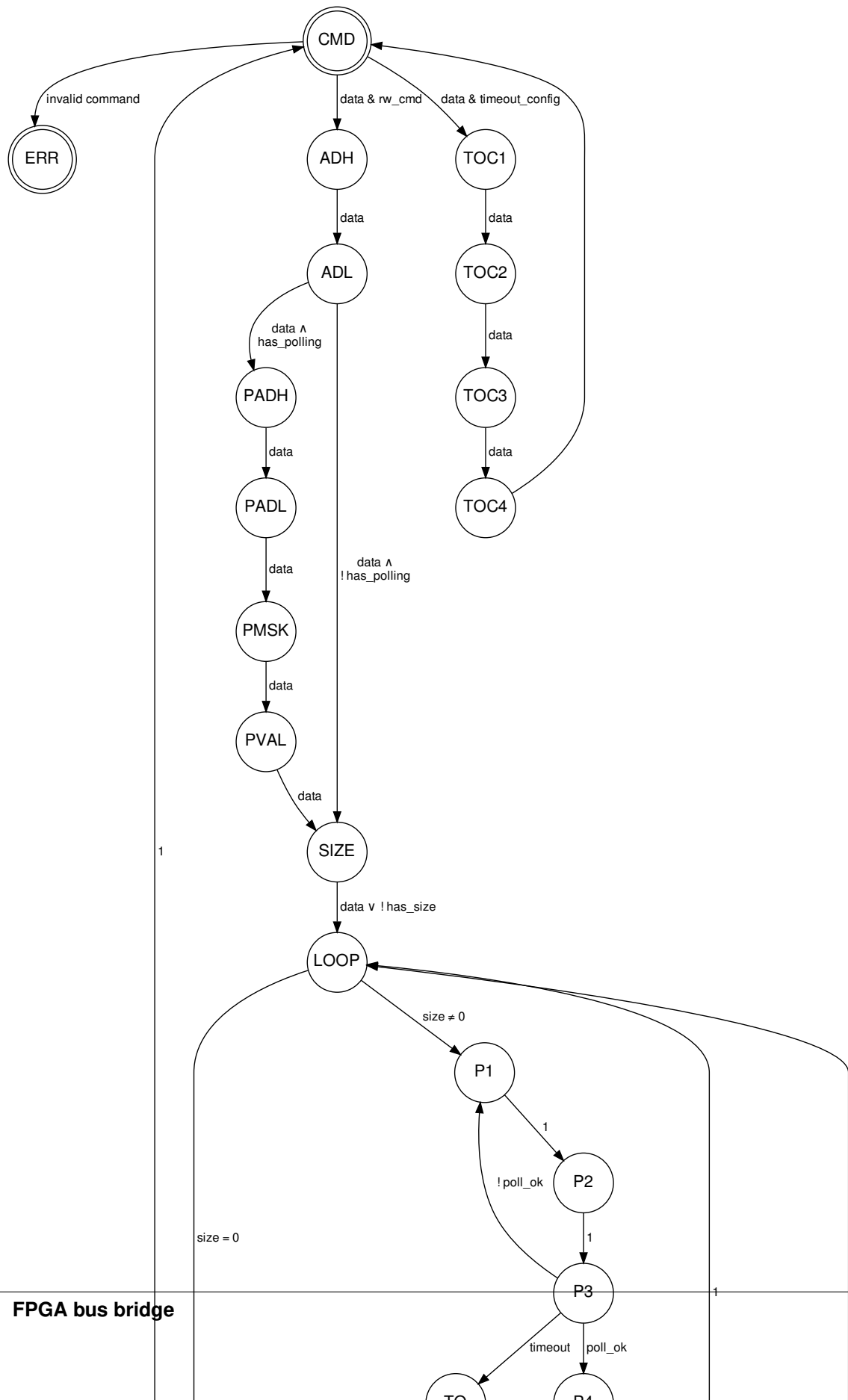
The state machine has a lot of states and is a bit complex. However, the parsing and execution of a command is faster than the time required to receive an incoming new command - making it hard to saturate the commands queue. The

only cases a command queue can be overflowed is when using polling commands - in such cases the application may wait for the response before sending further commands.

Overflowing the commands queue will usually lead to entering the error state as bytes will be discarded, resulting in data bytes being interpreted as invalid command codes. When in error state, the error LED on the board is lit. The only way to recover from the error state is pressing the reset button.

5.3.1 State machine

The following graph is the bus bridge finite state machine implemented in the FPGA. This document is an help for understanding FPGA internals.



Details on states:

- **CMD**: Idle state, awaiting for command byte.
- **ADH**: Awaiting for read/write address high nibble.
- **ADL**: Awaiting for read/write address low nibble.
- **PADH**: Awaiting for polling address high nibble.
- **PADL**: Awaiting for polling address low nibble.
- **PMSK**: Awaiting for polling mask.
- **PVAL**: Awaiting for polling value.
- **SIZE**: Read/write size fetch. If the command does not require size parameter, use 1. Otherwise, wait and store next byte received on UART.
- **LOOP**: Preload polling address on the bus for P1 and P2 states. Return to initial state if all bytes have been read or written.
- **WAIT**: Wait for UART to be ready to transmit a byte. This also acts as a delay cycle required for data bus to be available (due to pipelining).
- **SEND**: Send result byte on UART (command acknowledge or register value).
- **P1**: First polling state. Assert bus read signal. Due to pipelining, the data read from the bus is available to the rest of the logic at P2.
- **P2**: Second polling state. Used for pipelining (registers the data read from the bus).
- **P3**: Polling test. If polling value matches, leave polling by going to P4. Otherwise, return to P1 for a new polling read cycle.
- **P4**: End of polling. Load register address on the bus.
- **RD**: Register read cycle.
- **VAL**: Read from UART the byte to be written in register.
- **WR**: Register write cycle.
- **TO**: Timeout state. Loops until all unprocessed bytes have been discarded (write operation) or returned as zeros (read operation).
- **POP**: Cycle used to discard a byte from the input FIFO during a write operation which has timed out.

Details on transition conditions:

- **1**: Unconditionally pass to the next state at the next clock cycle.
- **data**: True when the UART FIFO has a byte available.
- **size**: Number of remaining bytes to be read or written.
- **ready**: True when the UART of the bus bridge is ready to transmit a byte.
- **read_command**: True when the fetched command byte corresponds to a bus read cycle command.
- **write_command**: True when the fetched command byte corresponds to a bus write cycle command.

5.3.2 Polling timeout configuration

A special command bit allows reconfiguring the polling timeout delay. The delay parameter is stored in an internal 32 bits register and encodes a number of clock cycles to wait during polling state (1 unit equals to 3 clock cycles). When

this register is set to 0, the timeout is disabled. Note that disabling the timeout is not recommended since the FSM may stuck in a polling operation forever or until general reset.

The maximum possible timeout duration is approximately 128 seconds.

5.4 Output module

The available I/Os of the scaffold board can be internally connected to any module output. The “right matrix” controls which module output signals are connected to which I/Os. Each I/O has a register storing its multiplexer selecting the source signal.

5.4.1 Multiplexers selection table

Index	Signal name
0	z
1	0
2	1
3	/power/dut_trigger
4	/power/platform_trigger
5	/uart0/tx
6	/uart0/trigger
7	/uart1/tx
8	/uart1/trigger
9	/iso7816/io_out
10	/iso7816/clk
11	/iso7816/trigger
12	/pgen0/out
13	/pgen1/out
14	/pgen2/out
15	/pgen3/out

Building and flashing the FPGA bitstream

Advanced users may want to modify the architecture of the FPGA of Scaffold to support more peripherals, implement new features or even fix bugs. This section briefly describe how to build the FPGA bitstream with Intel (Altera) tools, and flash the board.

6.1 Prerequisites

Quartus II software from Intel (Altera) must be used to build the FPGA bitstream. The version of Quartus must have support for Cyclone IV devices; version 14.1 can be used. Although *Quartus* is a proprietary tool, the free *Quartus Web Edition* can be used and shall not require any license. Quartus can run on linux and Windows (but we did not give a try on Windows).

For flashing the FPGA, a programmer supporting *Active Serial* mode must be used. *Altera USB Blaster* is a good one.

6.2 Building the bitstream

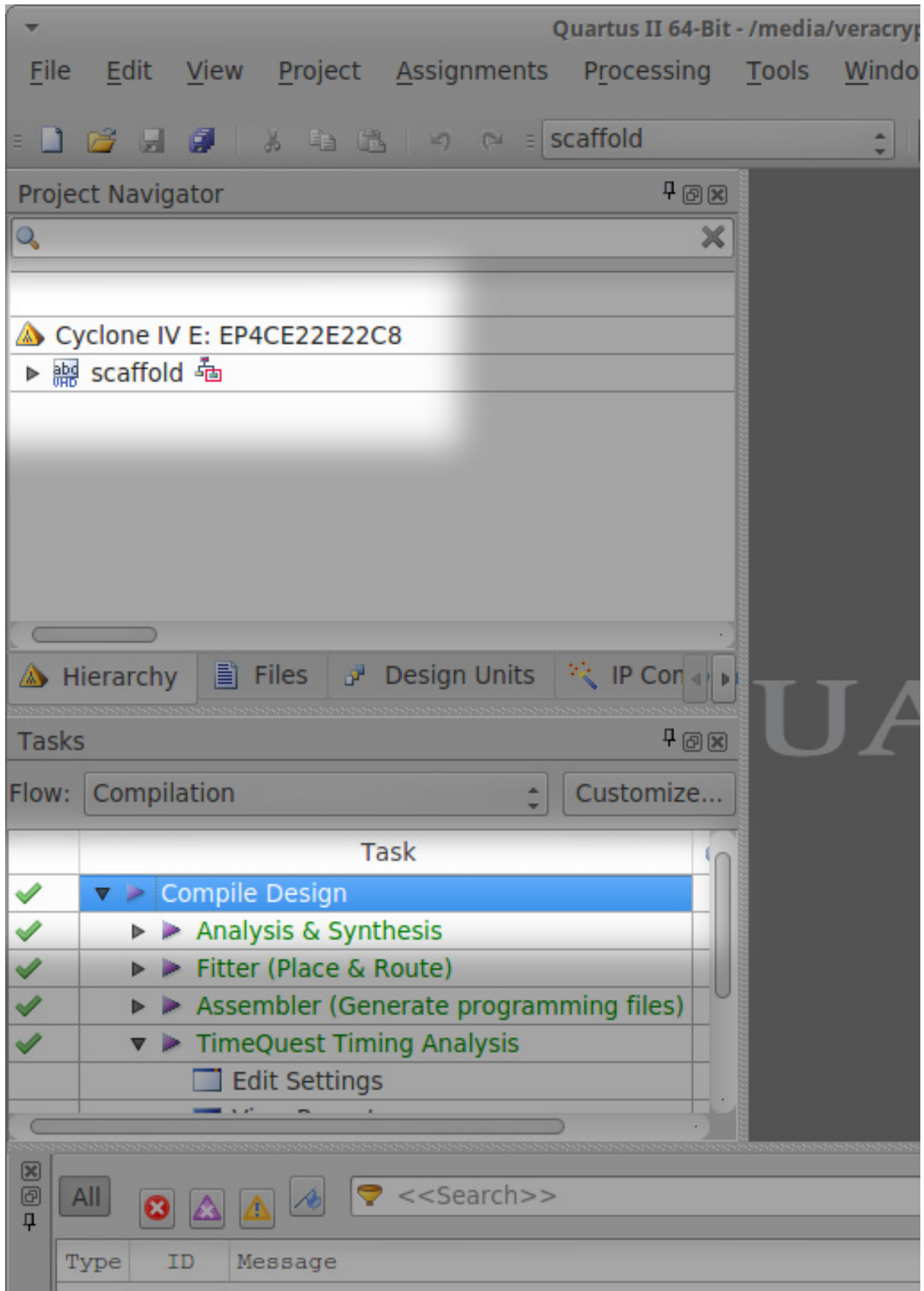
Under *Quartus* software, load the `fpga-arch/scaffold.qpf` project file with the *File > Open Project* menu. Build the design with *Start* context-menu of *Compile Design* task as highlighted below.

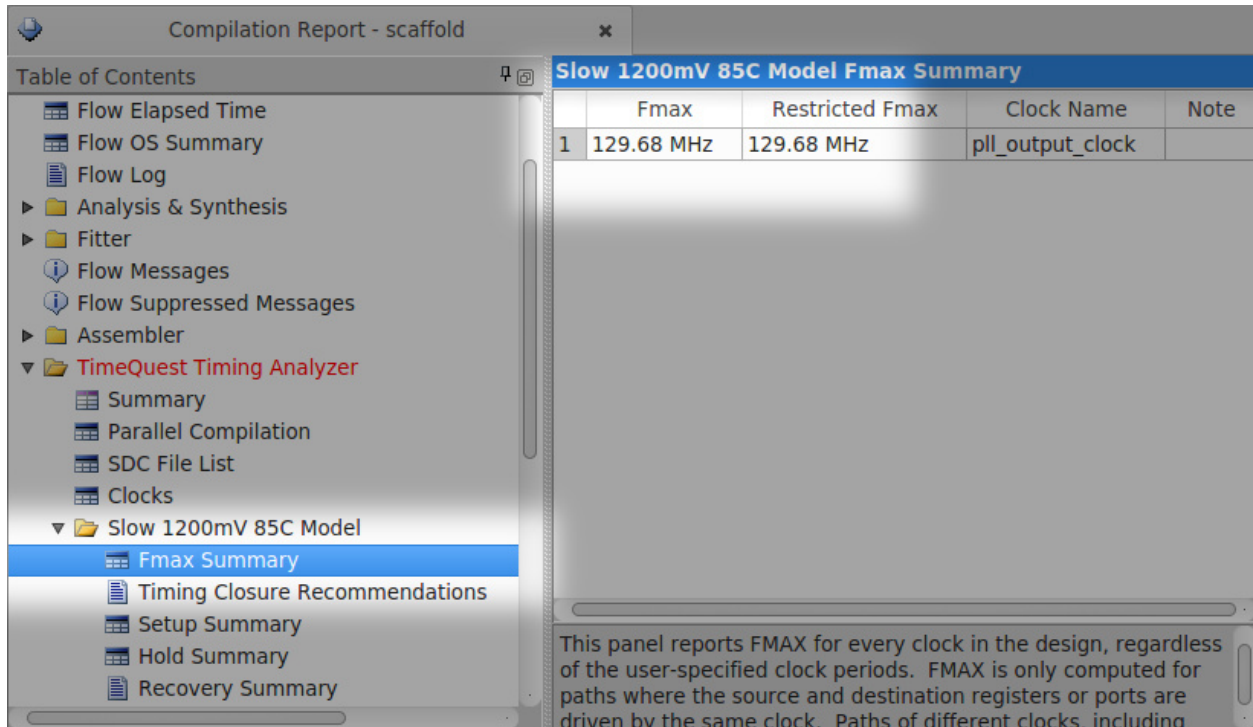
Hopefully the compilation should succeed. If you modified the original design, we recommend you to check in the compilation report that the max frequency of the system clock of the compiled design is at least 110 MHz (100 MHz plus some security margin). If this constraint is not respected, then your design may not work properly and need to be optimized.

6.3 Flashing the FPGA

Power-on the board and connect the programmer as shown below:

In *Quartus*, open the programmer window with the *Tools > Programmer* menu.



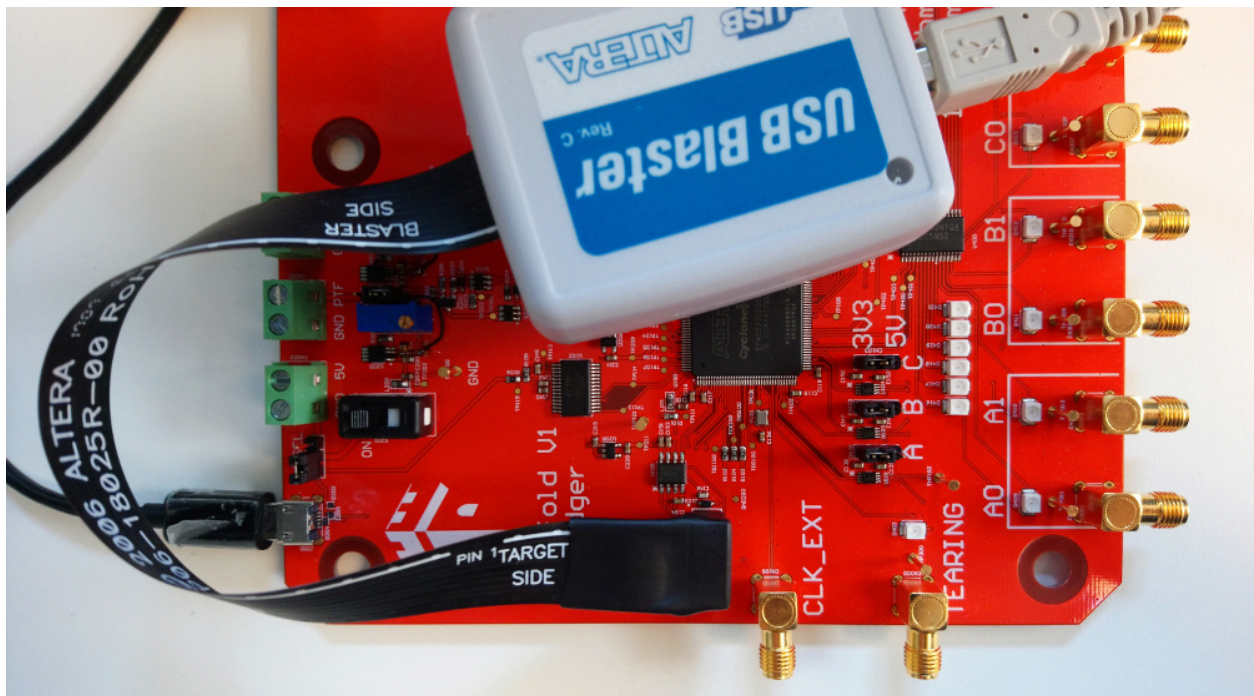


The screenshot shows the 'Compilation Report - scaffold' window. The left sidebar contains a 'Table of Contents' with the following items: Flow Elapsed Time, Flow OS Summary, Flow Log, Analysis & Synthesis, Fitter, Flow Messages, Flow Suppressed Messages, Assembler, TimeQuest Timing Analyzer (expanded), Summary, Parallel Compilation, SDC File List, Clocks, Slow 1200mV 85C Model (expanded), Fmax Summary (selected), Timing Closure Recommendations, Setup Summary, Hold Summary, and Recovery Summary.

The main panel displays the 'Slow 1200mV 85C Model Fmax Summary' table:

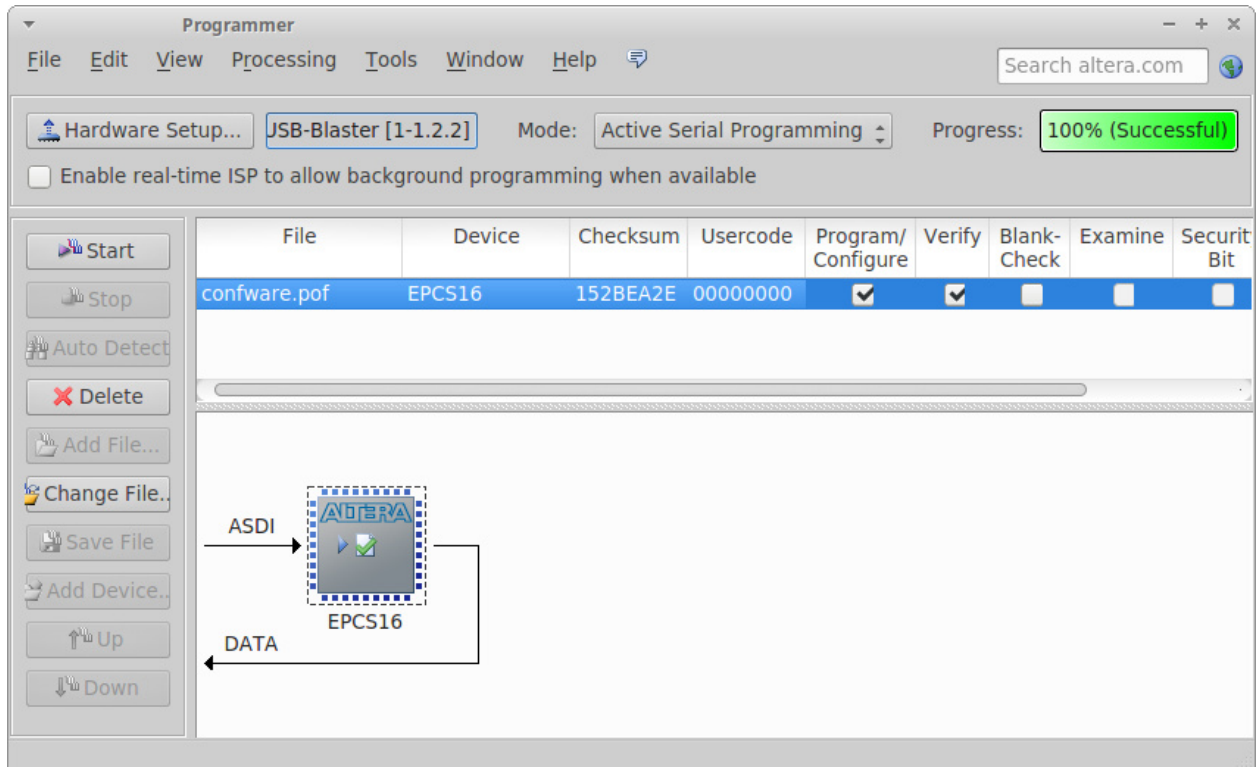
	Fmax	Restricted Fmax	Clock Name	Note
1	129.68 MHz	129.68 MHz	pll_output_clock	

Below the table, a text box states: 'This panel reports FMAX for every clock in the design, regardless of the user-specified clock periods. FMAX is only computed for paths where the source and destination registers or ports are driven by the same clock. Paths of different clocks, including



- Setup your programmer with the *Hardware Setup* button.
- Switch to *Active Serial Programming* mode.
- Click on *Add File...* and select the file `confware.pof`. The setup shall represent an EPCS16 device, which is the on-board Flash memory storing the bitstream and read by the FPGA when powering-up Scaffold.
- Check the *Program/Configure* and *Verify* boxes.
- Click on the *Start* button.

Remove the programmer cable to test your new design!



CHAPTER 7

Indices and tables

- `genindex`
- `modindex`
- `search`

S

scaffold, [15](#)
scaffold.iso7816, [23](#)
scaffold.stm32, [21](#)

Symbols

[__init__\(\)](#) (*scaffold.Scaffold method*), 15
[__init__\(\)](#) (*scaffold.Signal method*), 16
[__init__\(\)](#) (*scaffold.iso7816.Smartcard method*), 24
[__init__\(\)](#) (*scaffold.stm32.STM32 method*), 22
[_lshift__\(\)](#) (*scaffold.Signal method*), 16
[__str__\(\)](#) (*scaffold.Signal method*), 16

A

[all](#) (*scaffold.Power attribute*), 21
[apdu\(\)](#) (*scaffold.iso7816.Smartcard method*), 24
[apdu_str\(\)](#) (*scaffold.iso7816.Smartcard method*), 24
[assert_device\(\)](#) (*scaffold.stm32.STM32 method*), 22
[AUTO](#) (*scaffold.IOMode attribute*), 17

B

[baudrate](#) (*scaffold.UART attribute*), 17
[brightness](#) (*scaffold.LEDs attribute*), 21

C

[card_inserted](#) (*scaffold.iso7816.Smartcard attribute*), 24
[Chain](#) (*class in scaffold*), 21
[checksum\(\)](#) (*scaffold.stm32.STM32 method*), 22
[clear_event\(\)](#) (*scaffold.IO method*), 16
[Clock](#) (*class in scaffold*), 20
[clock_frequency](#) (*scaffold.ISO7816 attribute*), 18
[clock_stretching](#) (*scaffold.I2C attribute*), 18
[command\(\)](#) (*scaffold.stm32.STM32 method*), 22
[Convention](#) (*class in scaffold.iso7816*), 24
[count](#) (*scaffold.PulseGenerator attribute*), 20

D

[delay](#) (*scaffold.PulseGenerator attribute*), 20
[DIRECT](#) (*scaffold.iso7816.Convention attribute*), 24
[disabled](#) (*scaffold.LEDs attribute*), 21
[DOWN](#) (*scaffold.Pull attribute*), 17
[dut](#) (*scaffold.Power attribute*), 21

E

[empty](#) (*scaffold.ISO7816 attribute*), 18
[etu](#) (*scaffold.ISO7816 attribute*), 18
[EVEN](#) (*scaffold.UARTParity attribute*), 17
[event](#) (*scaffold.IO attribute*), 16
[extended_erase\(\)](#) (*scaffold.stm32.STM32 method*), 22

F

[find_info\(\)](#) (*scaffold.iso7816.Smartcard method*), 24
[fire\(\)](#) (*scaffold.PulseGenerator method*), 20
[flush\(\)](#) (*scaffold.I2C method*), 18
[flush\(\)](#) (*scaffold.ISO7816 method*), 18
[flush\(\)](#) (*scaffold.UART method*), 17
[freq_a](#) (*scaffold.Clock attribute*), 20
[freq_b](#) (*scaffold.Clock attribute*), 21
[frequency](#) (*scaffold.I2C attribute*), 19
[frequency](#) (*scaffold.SPI attribute*), 19

G

[get\(\)](#) (*scaffold.stm32.STM32 method*), 22
[get_id\(\)](#) (*scaffold.stm32.STM32 method*), 22
[go\(\)](#) (*scaffold.stm32.STM32 method*), 22

I

[I2C](#) (*class in scaffold*), 18
[interval](#) (*scaffold.PulseGenerator attribute*), 20
[INVERSE](#) (*scaffold.iso7816.Convention attribute*), 24
[inverse_byte\(\)](#) (*scaffold.iso7816.Smartcard method*), 24
[IO](#) (*class in scaffold*), 16
[IOMode](#) (*class in scaffold*), 17
[ISO7816](#) (*class in scaffold*), 17

L

[LEDs](#) (*class in scaffold*), 21

M

[mode](#) (*scaffold.IO attribute*), 16

N

name (*scaffold.Signal attribute*), 16
 NONE (*scaffold.Pull attribute*), 17
 NONE (*scaffold.UARTParity attribute*), 17

O

ODD (*scaffold.UARTParity attribute*), 17
 OPEN_DRAIN (*scaffold.IOMode attribute*), 17
 override (*scaffold.LEDs attribute*), 21

P

parent (*scaffold.Signal attribute*), 16
 parity (*scaffold.UART attribute*), 17
 parity_mode (*scaffold.ISO7816 attribute*), 18
 path (*scaffold.Signal attribute*), 16
 phase (*scaffold.SPI attribute*), 19
 platform (*scaffold.Power attribute*), 21
 polarity (*scaffold.PulseGenerator attribute*), 20
 polarity (*scaffold.SPI attribute*), 20
 Power (*class in scaffold*), 21
 ProtocolError (*class in scaffold.iso7816*), 25
 Pull (*class in scaffold*), 17
 pull (*scaffold.IO attribute*), 16
 PulseGenerator (*class in scaffold*), 20
 PUSH_ONLY (*scaffold.IOMode attribute*), 17

R

raw_transaction() (*scaffold.I2C method*), 19
 read() (*scaffold.I2C method*), 19
 read_memory() (*scaffold.stm32.STM32 method*), 22
 read_option_bytes() (*scaffold.stm32.STM32 method*), 22
 readout_protect() (*scaffold.stm32.STM32 method*), 22
 readout_unprotect() (*scaffold.stm32.STM32 method*), 23
 rearm() (*scaffold.Chain method*), 21
 receive() (*scaffold.ISO7816 method*), 18
 receive() (*scaffold.iso7816.Smartcard method*), 24
 receive() (*scaffold.UART method*), 17
 reset() (*scaffold.iso7816.Smartcard method*), 24
 reset() (*scaffold.LEDs method*), 21
 reset() (*scaffold.UART method*), 17
 reset_config() (*scaffold.I2C method*), 19
 reset_config() (*scaffold.ISO7816 method*), 18

S

Scaffold (*class in scaffold*), 15
 scaffold (*module*), 15
 scaffold.iso7816 (*module*), 23
 scaffold.stm32 (*module*), 21
 Signal (*class in scaffold*), 16
 Smartcard (*class in scaffold.iso7816*), 23

SPI (*class in scaffold*), 19
 startup_bootloader() (*scaffold.stm32.STM32 method*), 23
 startup_flash() (*scaffold.stm32.STM32 method*), 23
 STM32 (*class in scaffold.stm32*), 21

T

transmit() (*scaffold.ISO7816 method*), 18
 transmit() (*scaffold.SPI method*), 20
 transmit() (*scaffold.UART method*), 17
 trigger_long (*scaffold.ISO7816 attribute*), 18
 trigger_rx (*scaffold.ISO7816 attribute*), 18
 trigger_tx (*scaffold.ISO7816 attribute*), 18

U

UART (*class in scaffold*), 17
 UARTParity (*class in scaffold*), 17
 UP (*scaffold.Pull attribute*), 17

V

value (*scaffold.IO attribute*), 16

W

wait_ack() (*scaffold.stm32.STM32 method*), 23
 wait_ack_or_nack() (*scaffold.stm32.STM32 method*), 23
 width (*scaffold.PulseGenerator attribute*), 20
 write() (*scaffold.I2C method*), 19
 write_memory() (*scaffold.stm32.STM32 method*), 23