ironman Documentation

Release 0.5.3.dev9

Giordon Stark

Feb 11, 2022

CONTENTS

This documentation is for at least 0.5.3.

Contents:

CHAPTER

ONE

IRONMAN

1.1 What is Ironman?

Ironman is a general purpose software toolbox to be run on L1Calo hardware with embedded processors (SoCs).

Look how easy it is to use

```
>>> import ironman
>>> # Get your stuff done
>>> ironman.engage()
```

1.2 Features

- Be awesome
- Make things faster

1.3 Getting Started

1.3.1 Installing

Install ironman by running

```
pip install ironman
```

1.3.2 Developing

If it is your first time...

```
git clone git@github.com:kratsg/ironman
cd ironman && mkvirtualenv ironman
pip install -r requirements.txt
```

and then afterwards...

```
workon ironman
python setup.py develop
... do work here ...
pip uninstall ironman
```

Testing

```
tox
```

or with

```
py.test
```

1.3.3 Contributing

- Issue Tracker
- Source Code

1.3.4 Support

If you are having issues, let us know.

1.3.5 Releasing

1. Do some work on your package (i.e. fix bugs, add features, etc)
2. Make sure the tests pass. Run `pytest`.
3. Update the `__version__` number via `bump2version`.
4. Push to the default branch.

1.4 Tutorial

Since we will be predominantly using Twisted within the Zynq to manage the Reactor workflow ("callbacks"), I suggest reading through this tutorial on your own time to get up to speed on how it works and some details of sockets.

I'm following the guide based on `sandman` here

1.5 To Do

- split udp and tcp into different, separate protocols: http://stackoverflow.com/questions/33224142/twisted-protocol-that-simultaneously-handles-tcp-and-udp-at-once

1.6 Ideas

- make it like twisted.web - we build Request objects which need to find Resource objects that provide actions (maybe too complicated, try and simplify?) link

CHAPTER

TWO

BACKGROUND

Ironman is a software infrastructure that will standardize communication protocols on chips with embedded processors. In particular, this was written for use in L1Calo with IPBus, but was written to maintain modularity and flexibility to evolve with changing needs.

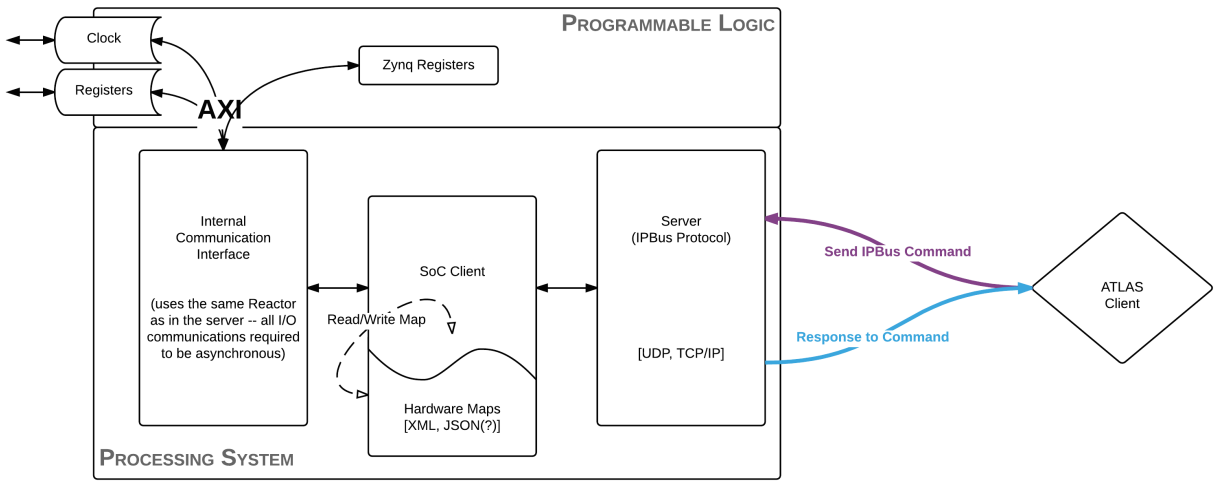
To explain this approach, we first started with our vision for the software with an overview. Below, we explain each part in more detail.

2.1 Goals

We make it as easy as possible for *anyone* to put their pieces in to the general framework while maintaining the overall procedure. This software will

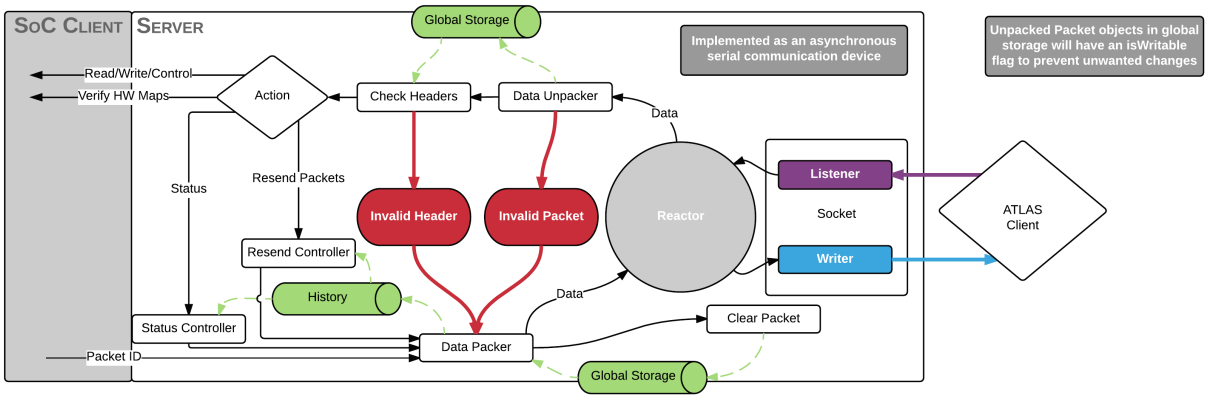
- provide a **wide array of standard networking protocols** for reading and writing packets
- allow for implementation of **custom communication protocols** for reading and writing the various hardware components
- allow for definition of **custom hardware maps** which specify the layout of the entire board
- use a single-threaded **reactor** model, an event-driven model, which is a global loop that fires listeners when certain events have triggered

2.2 Overview



An external client, such as a human or a monkey, is tasked with the job of communicating a transaction request or a status query of a piece of hardware (*ATLAS Client*). This is sent, for example, as an IPBus command to the board running **ironman** software. This request is received by a server actively listening and then dispatches this request to the System-on-Chip client. The System-on-Chip client is made self-aware using hardware definitions and dispatches this request along a callback chain to the Internal Communications Interface. It is at this point that the software handles the customized communication to fulfill the request of the biological being controling the ATLAS Client. After this, the response is formed, transferred back to the server who will transmit the information back to the client.

2.2.1 Server



The server has a few jobs to do upon receipt of a packet. As the server is actively listening, it is going to plug itself into the reactor and kick off the callback chain for us (known as *deferreds* or *promises*).

The server knows the format of the packet entirely and will unpack the data into a Request Packet Object that will be used by the rest of the software. It is at this point that a few checks are done, such as checking that the data *can* be unpacked as well as making sure the headers are valid.

If the basic sanity checks look good, then it must decide what to do with the packet. If the request requires communication with the hardware, then it will pass along the packet to the SoC Client to dispatch the request. If it simply requires information about the history of packets sent (such as a Resend Packet), then it will return the packets it records in history.

This leads to the other part of the server which is to maintain a log of all inbound/request and outbound/response packets. It is at this point which the board communicates with the outside world and makes it a suitable place to implement the history recording.

For those who are more familiar or learn better with some code:

```
>>> # grab pieces of ironman
>>> from ironman.server import ServerFactory
>>> from ironman.packet import IPBusPacket
>>> from ironman.history import History
>>> history = History()
>>>
>>> # we need deferreds and reactor from twisted
>>> from twisted.internet import reactor
>>> from twisted.internet.defer import Deferred
>>>
>>> # build up our callback chain
>>> def callbacks():
...     return Deferred().addCallback(IPBusPacket).addCallback(history.record)
>>>
```

(continues on next page)

(continued from previous page)

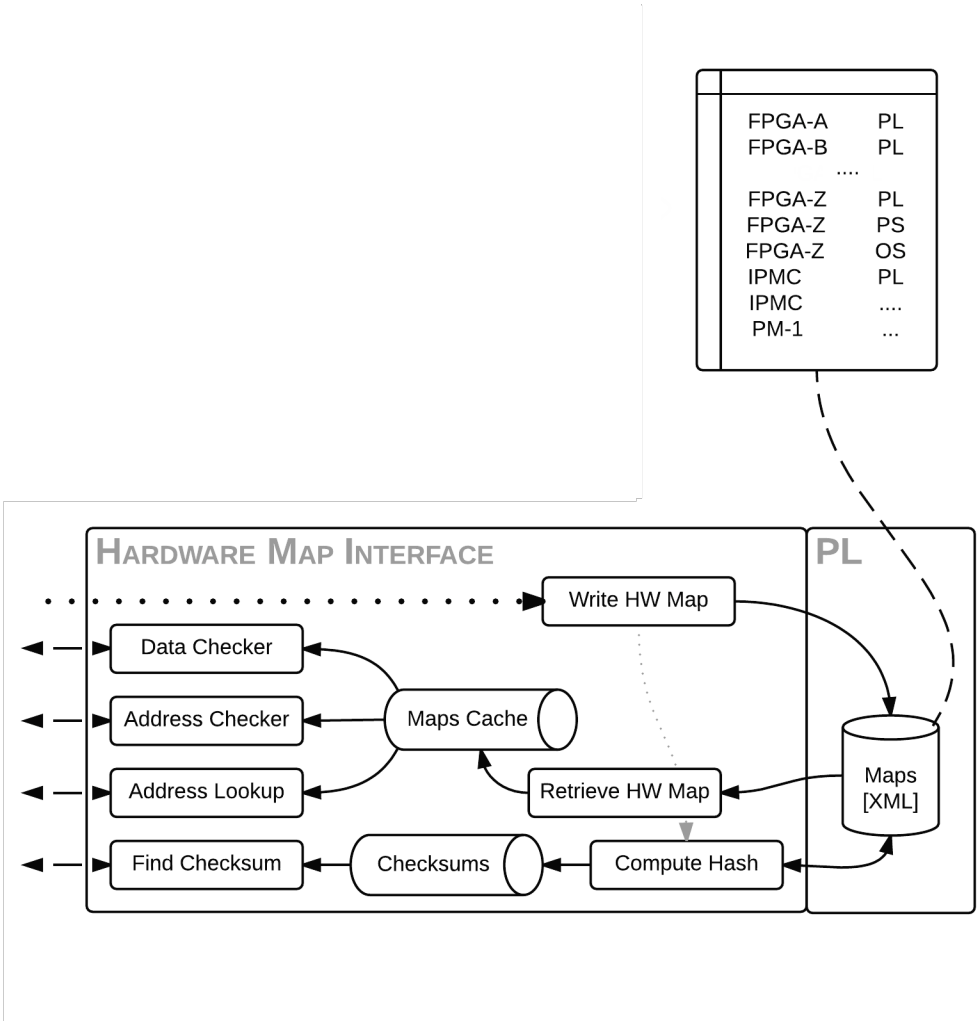
```
>>> # listen for IPBus packets over UDP at port 8888
>>> reactor.listenUDP(8888, ServerFactory('udp', callbacks))
<<class 'ironman.server.UDP'> on 8888>
>>>
>>> # listen for IPBus packets over TCP at port 8889
>>> reactor.listenTCP(8889, ServerFactory('tcp', callbacks))
<<class 'twisted.internet.tcp.Port'> of <class 'ironman.server.TCPFactory'> on 8889>
```

and of course, this is all building up our logic. To actually start up the server, we simply need:

```
# start the global loop
reactor.run()
```

Notice how that independent of which transport is being used to communicate with the server, the same callback chain is being executed correctly.

2.2.2 Hardware Interface



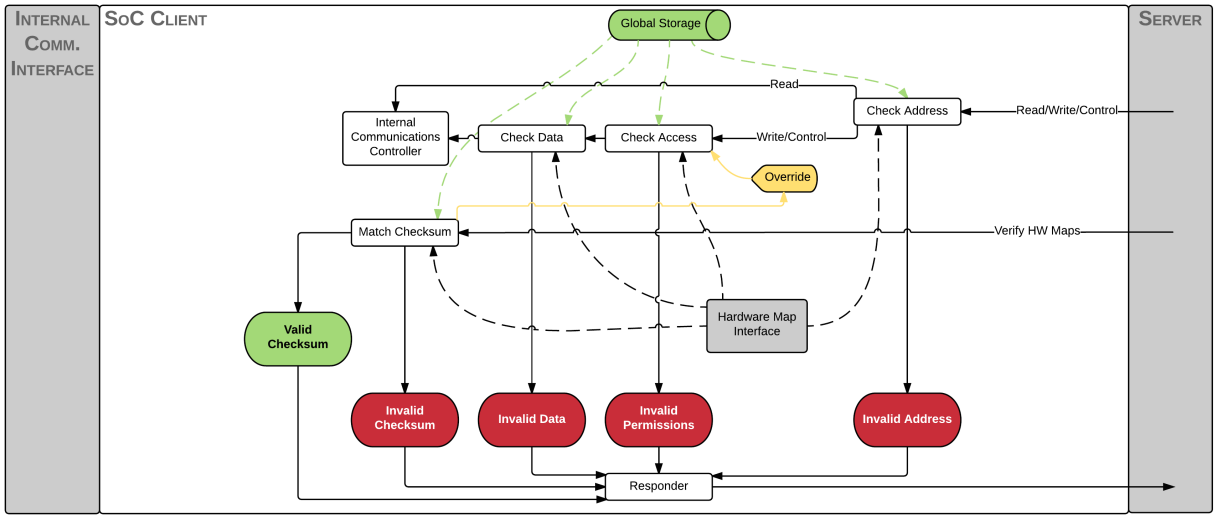
The job of the Hardware Interface is to parse the hardware definitions transferred to the board and build up a cached, global mapping of address to properties about the address. In Python terminology, this is a giant dictionary. It must

assess that a single address is not taken up by two different hardware definitions (no conflicts) and that the hardware map is parseable and valid (the latter has yet to be defined yet). It will also provide a way to compute the checksum of the hardware map files to ensure that the board is running on the same definitions that the monkey has communicated to the board with.

The Hardware Manager is our primary means of interfacing. We can demonstrate via another code example:

```
>>> xadcyml = """ nodes:
...     -
...         id: temperature
...         address: 0x00000000
...         nodes:
...             - &offset {id: offset, address: 0x0, permissions: 1}
...             - &raw {id: raw, address: 0x1, permissions: 1}
...             - &scale {id: scale, address: 0x2, permissions: 1}
...     -
...         id: vccint
...         address: 0x00000010
...         nodes: [*raw, *scale]
...     -
...         id: vccaux
...         address: 0x00000020
...         nodes: [*raw, *scale]
...     -
...         id: vccbram
...         address: 0x00000030
...         nodes: [*raw, *scale]
...     -
...         id: vccpint
...         address: 0x00000040
...         nodes: [*raw, *scale]
...     -
...         id: vccpaux
...         address: 0x00000050
...         nodes: [*raw, *scale]
...     -
...         id: vccoddr
...         address: 0x00000060
...         nodes: [*raw, *scale]
...     -
...         id: vrefp
...         address: 0x00000070
...         nodes: [*raw, *scale]
...     -
...         id: vrefn
...         address: 0x00000080
...         nodes: [*raw, *scale]"""
>>>
>>> # initialize a manager to use for everyone that needs it
>>> from ironman.hardware import HardwareManager, HardwareMap
>>> manager = HardwareManager()
>>> # add a map to the manager
>>> manager.add(HardwareMap(xadcyml, 'xadc'))
```

2.2.3 Client



The job of the client here is to analyze the packet more thoroughly. If the client is handling the packet, then it must be a request packet. It will then communicate with the Hardware Interface to determine whether or not the transaction packet is good: valid address, valid permissions, valid data. If all of these things pass, it then passes the packet along to the Internal Communications which will build up a response.

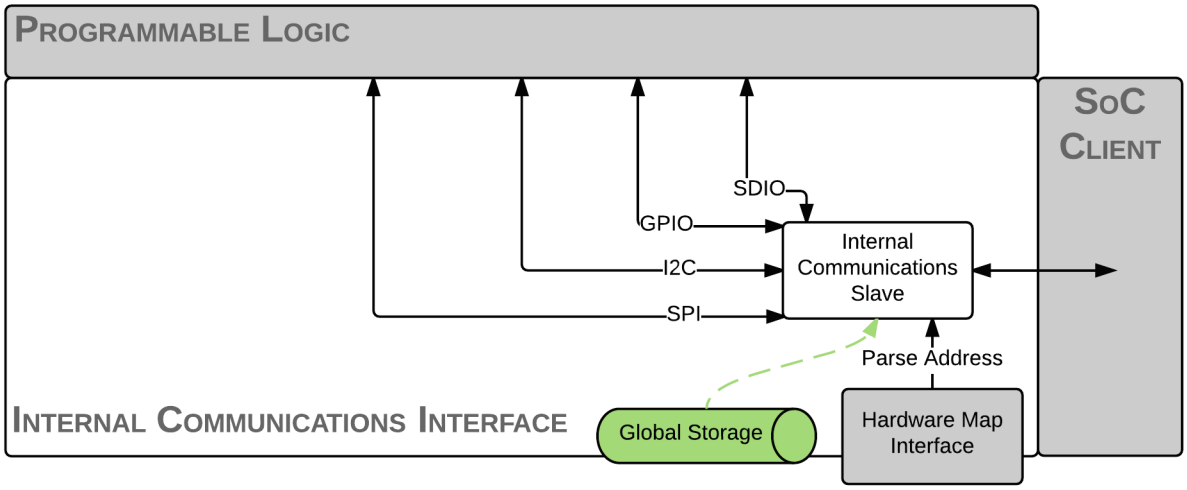
It should be noted that the client is not allowed to modify the response packet at all. Only the Server and the Internal Communications are allowed to do this.

In **ironman**, the client is known as Jarvis (the assistant, get it?). Jarvis is used like so:

```
>>> # now let's make jarvis
>>> from ironman.communicator import Jarvis, ComplexIO
>>> jarvis = Jarvis()
>>> # tell Jarvis about our hardware manager
>>> jarvis.set_hardware_manager(manager)
```

In particular, Jarvis is one of the easiest things to set up since it contains a lot of internal logic to route requests appropriately and execute controllers for you. In this way, Jarvis is a lot like a *router*.

2.2.4 Internal Communications



Lastly, the Internal Communications is primarily custom code written by the developers to do exactly that: communicate with the board. Depending on how the board is set up, there may be a virtual filesystem or raw pointers or custom drivers that the code will need to access. Since this is something that will vary on a board-by-board basis, we leave most of this code up to the user and only provide a few simple cases for file reading and writing.

Continuing on with our code examples as above, you might have your driver create a virtual file system for the temperature. So how would you create a custom communications controller that Jarvis knows about that handles the requests?

```
>>> # register a controller with jarvis
>>> @jarvis.register('xadc')
... class XADCController(ComplexIO):
...     __base__ = "/sys/devices/soc0/amba@0/f8007100.ps7-xadc/iio:device0/"
...     __f__ = {
...                 0:   __base__+"in_temp0_offset",
...                 1:   __base__+"in_temp0_raw",
...                 2:   __base__+"in_temp0_scale",
...                 17:  __base__+"in_voltage0_vccint_raw",
...                 18:  __base__+"in_voltage0_vccint_scale",
...                 33:  __base__+"in_voltage1_vccaux_raw",
...                 34:  __base__+"in_voltage1_vccaux_scale",
...                 49:  __base__+"in_voltage2_vccbram_raw",
...                 50:  __base__+"in_voltage2_vccbram_scale",
...                 65:  __base__+"in_voltage3_vccpint_raw",
...                 66:  __base__+"in_voltage3_vccpint_scale",
...                 81:  __base__+"in_voltage4_vccpaux_raw",
...                 82:  __base__+"in_voltage4_vccpaux_scale",
...                 97:  __base__+"in_voltage5_vccoddr_raw",
...                 98:  __base__+"in_voltage5_vccoddr_scale",
...                 113: __base__+"in_voltage6_vrefp_raw",
...                 114: __base__+"in_voltage6_vrefp_scale",
...                 129: __base__+"in_voltage7_vrefn_raw",
...                 130: __base__+"in_voltage7_vrefn_scale"
...             }
```

And you are done. This will read from `/sys/devices/soc0/amba@0/f8007100.ps7-xadc/iio:device0/in_temp0_offset` if an IPBus read request is recieved for address `0x0`. Similarly, it will read from `in_voltage6_vrefp_raw` if the address is `0x71` (which is `113` in decimal).

In this particular example, it is assumed you had added a hardware definitions for the *xADC* controller which is being registered to Jarvis. Each file path is associated with an address that you would explicitly map out. A future iteration of how hardware gets defined should alleviate the numerous redefinitions of addresses that occur.

CHAPTER

THREE

COOKBOOK

3.1 Handling IPBus packets

It is common to use ironman to parse and build ipbus packets. Expecting this major usage of the software being written, we use the awesome construct package to build an *IPBusConstruct* builder/parser to make it easier for everyone to use.

In the examples that follow, we will use (and assume) a *big-endian* aligned data packet that contains the IPBus commands.

```
data = '\x20\x00\x00\xf0\x20\x00\x01\x0f\x00\x00\x00\x03'
```

which is a single read transaction request from the *base address* `0x3`. In particular, it contains three 32-bit words:

Word	Hex	Meaning
0	0x200000f0	IPBus Packet Header
1	0x200001f0	Read Transaction Header
2	0x00000003	Base Address of Read

3.1.1 Parsing an IPBus Packet

```
>>> from ironman.constructs.ipbus import IPBusConstruct
>>> data = b'\x20\x00\x00\xf0\x20\x00\x01\x0f\x00\x00\x00\x03'
>>> p = IPBusConstruct.parse(data)
>>> print(p)
Container:
    endian = (enum) BIG 240
    header = Container:
        protocol_version = 2
        reserved = 0
        id = 0
        byteorder = 15
        type_id = (enum) CONTROL 0
    transactions = ListContainer:
        Container:
            header = Container:
                protocol_version = 2
                id = 0
                words = 1
                type_id = (enum) READ 0
                info_code = (enum) REQUEST 15
```

(continues on next page)

(continued from previous page)

```
            address = 3
            data = None
    status = None
    resend = None
```

```
>>>
```

3.1.2 Building an IPBus Packet

Because of duck-typing, any object can make do when passing into the construct builder. See the *construct* docs for more information here. In this case, we will take the original packet which has a packet id `0x0` in the header and update it to `0x1`

```
>>> from ironman.constructs.ipbus import IPBusConstruct
>>> data = b'\x20\x00\x00\xf0\x20\x00\x01\x0f\x00\x00\x00\x03'
>>> p = IPBusConstruct.parse(data)
>>> p.header.packet_id = 0x1
>>> new_data = IPBusConstruct.build(p)
>>> print(repr(new_data))
b' \x00\x00\xf0 \x00\x01\x0f\x00\x00\x00\x03'
>>>
```

Note that when building an IPBus Packet, an error would be raised if we cannot build it. For example, if we tried to bump the protocol version to a non-valid one

```
>>> from ironman.constructs.ipbus import IPBusConstruct
>>> data = b'\x20\x00\x00\xf0\x20\x00\x01\x0f\x00\x00\x00\x03'
>>> p = IPBusConstruct.parse(data)
>>> p.header.protocol_version = 0x0
>>> new_data = IPBusConstruct.build(p)
Traceback (most recent call last):
    ...
construct.core.ValidationError: object failed validation: 0
```

which is letting us know (not a very verbose error) that the `0x0` value is wrong.

3.1.3 Creating a Response Packet

As seen from the above examples, we have a read packet. Let's pretend the response is `1234`. How do we build a response packet?

```
>>> from ironman.constructs.ipbus import IPBusConstruct
>>> in_data = b'\x20\x00\x00\xf0\x20\x00\x01\x0f\x00\x00\x00\x03'
>>> in_p = IPBusConstruct.parse(in_data)
>>> out_p = in_p
>>> out_p.transactions[0].data = [int(b"1234".hex(), 16)]
>>> out_data = IPBusConstruct.build(out_p)
Traceback (most recent call last):
    ...
construct.core.CheckError: check failed during building
>>> out_p.transactions[0].header.info_code = 'SUCCESS'
>>> out_data = IPBusConstruct.build(out_p)
>>> print(repr(out_data))
```

(continues on next page)

(continued from previous page)

```
b' \x00\x00\xf0 \x00\x01\x001234'
>>>
```

and our work is done! Notice that it's not just a matter of setting the data field and building the packet.. we must also set the `info_code` field to a `SUCCESS` to signify that we're sending a *successful* response back.

3.2 Random Number Generator

One might like to be able to generate a full test of the `ironman` suite by setting up fake routes for reading/writing as a proof-of-concept. I demonstrate such a concept using a lot of different pieces of code here:

```
>>> from ironman.constructs.ipbus import IPBusConstruct, IPBusWords
>>> from ironman.hardware import HardwareManager, HardwareMap
>>> from ironman.communicator import Jarvis
>>> from ironman.packet import IPBusPacket
>>> from twisted.internet.defer import Deferred
>>> from ironman.constructs.ipbus import IPBusWord
>>> import random, struct
>>> random.seed(1)
>>>
>>> hardware_map = '''
... nodes:
...   -
...     id: random_number_generator
...     address: 0x00000000
...     nodes:
...       - {id: generate, address: 0x0, permissions: 1}
...       - {id: low_val, address: 0x1, permissions: 2}
...       - {id: high_val, address: 0x2, permissions: 2}
... '''
...
>>> j = Jarvis()
>>> manager = HardwareManager()
>>>
>>> manager.add(HardwareMap(hardware_map, 'main'))
>>> j.set_hardware_manager(manager)
>>>
>>> @j.register('main')
... class RandomNumberGeneratorController:
...   __low__  = 0
...   __high__ = 9
...   def generate(self):
...     return IPBusWord.build(random.randint(self.__class__.__low__, self.__class__._
↪_high__))
...
...   def read(self, offset, size):
...     if offset == 0x0: return ''.join(self.generate() for i in range(size))
...     elif offset == 0x1: return IPBusWord.build(self.__class__.__low__)
...     elif offset == 0x2: return IPBusWord.build(self.__class__.__high__)
...
...   def write(self, offset, data):
...     if offset == 0x0: pass
...     elif offset == 0x1: self.__class__.__low__ = IPBusWord.parse(data[0])
...     elif offset == 0x2: self.__class__.__high__ = IPBusWord.parse(data[0])
```

(continues on next page)

(continued from previous page)

```
...     return
...
>>> def buildResponsePacket(packet):
...     packet.response.transactions[0].header.info_code = 'SUCCESS'
...     return IPBusConstruct.build(packet.response)
...
>>> def printPacket(raw):
...    print("raw: {0:s}".format(repr(raw.hex())))
...    packet = IPBusConstruct.parse(raw)
...    print(packet)
...    print("data: {0:d}".format(struct.unpack('=I', IPBusWords.build(packet.
↪transactions[0].data))[0]))
...
>>> d = Deferred().addCallback(IPBusPacket).addCallback(j).
↪addCallback(buildResponsePacket).addCallback(printPacket)
>>> d.callback(bytearray.fromhex('200000f02000010f00000002'))  # read the upper limit
raw: '200000f02000010000000009'
Container:
    endian = (enum) BIG 240
    header = Container:
        protocol_version = 2
        reserved = 0
        id = 0
        byteorder = 15
        type_id = (enum) CONTROL 0
    transactions = ListContainer:
        Container:
            header = Container:
                protocol_version = 2
                id = 0
                words = 1
                type_id = (enum) READ 0
                info_code = (enum) SUCCESS 0
            address = None
            data = ListContainer:
                b'\x00\x00\x00\t'
    status = None
    resend = None
data: 9
```

CHAPTER

FOUR

API REFERENCE

4.1 ironman package

4.1.1 Subpackages

ironman.constructs package

Submodules

ironman.constructs.ipbus module

ironman.constructs.ipbus.**ControlHeaderStruct = <TransformData <Struct>>**
Struct detailing the Control Header logic

ironman.constructs.ipbus.**ControlStruct = <Renamed ControlTransaction <Struct>>**
Struct detailing the Control Action logic

Note:

- RMWBits: Should compute via $X \Leftarrow (X \wedge A) \vee (B \wedge (!A))$
- RMWSum: Should compute via $X \Leftarrow X + A$

---

ironman.constructs.ipbus.**IPBusConstruct = <Renamed IPBusPacket <Struct>>**
Top-level IPBus Construct which is a packet parser/builder

ironman.constructs.ipbus.**IPBusWords_long = <GreedyRange <TransformData <Bytes>>>**
This works _IPBusWord_long = Bytes(8) IPBusWord_long = Bytes(8) IPBusWords_long = GreedyRange(IPBusWord_long)

ironman.constructs.ipbus.**PacketHeaderStruct = <TransformData <Struct>>**
Struct detailing the Packet Header logic

byteorder is *0xf* if big-endian and *0x0* if little-endian

ironman.constructs.ipbus.**ResendStruct = <Renamed ResendTransaction +nonbuild <Struct +nonbu**
Struct detailing the Resend Action logic

ironman.constructs.ipbus.**StatusResponseStruct = <Renamed StatusTransaction <Struct>>**
Struct detailing the Status Action logic

Module contents

4.1.2 Submodules

ironman.communicator module

This file implements all of the various communications one might need to do.

Jarvis provides a callback structure that looks up (in its registry) for an appropriate communication protocol.

class ironman.communicator.**ComplexIO**
Bases: object

read(*offset*, *size*)

write(*offset*, *data*)

class ironman.communicator.**Jarvis**
Bases: object

This is the general communication slave.

Jarvis is what lets us pass around communications to various routes/protocols while keeping the details separated from us. Here's an example of how one might use it:

```
>>> from ironman.communicator import Jarvis, SimpleIO
>>> # create a Jarvis instance to manage what we want to register
>>> j = Jarvis()
>>> # tell Jarvis to register this class for the given route
>>> @j.register('fpgaOne')
... class FileOne(SimpleIO):
...     __f__ = '/path/to/fileOne'
...
>>> # tell Jarvis to register this class for the given route
>>> @j.register('fpgaTwo')
... class FileTwo(SimpleIO):
...     __f__ = '/path/to/fileTwo'
...
>>> # print the available registered classes
>>> import pprint
>>> pprint.pprint(j.registry)
{'fpgaOne': <class 'ironman.communicator.FileOne'>,
 'fpgaTwo': <class 'ironman.communicator.FileTwo'>}
```

Jarvis does the wrapping for *Jarvis.register()* so that a class defined at run-time is automatically inserted.

parse_address(*address*)

register(*route*)

set_hardware_manager(*hwmanager*)

unregister(*route*)

class ironman.communicator.**SimpleIO**
Bases: object

read(*offset*, *size*)

write(*offset*, *data*)

ironman.globals module

ironman.hardware module

class ironman.hardware.**BlockMemHardwareManager**
Bases: *ironman.hardware.HardwareManager*

get_node(*address*)

class ironman.hardware.**HardwareManager**
Bases: dict

add(*new_hw_map*)
Add the HW map only if it doesn't exist for a given key, and no address collisions

check_address(*address*)

check_data(*address*, *data*)

find_address(*address*)

get_checksum(*map_name*)

get_node(*address*)

get_route(*address*)

raw_maps = {}

subtract(*route*)
Remove the route entirely.

class ironman.hardware.**HardwareMap**(*yml*, *route*)
Bases: dict

isOk()

parse(*yml*)

class ironman.hardware.**HardwareNode**(*node*, *hw_map*)
Bases: dict

property allowed

property disallowed

property isOk

isValueValid(*val*)

property permissions

property readable

property writeable

class ironman.hardware.**NullHardwareMap**
Bases: dict

isOk()

parse(*yml*)

route = None

class ironman.hardware.**NullHardwareNode**
Bases: dict

allowed = {}

disallowed = {}

hw_map = {}

isOk = False

isValueValid = False

permissions = {}

readable = False

writeable = False

ironman.history module

class ironman.history.**History**(*maxlen=100*)
Bases: dict

record(*packet*)

ironman.interfaces module

interface ironman.interfaces.**ICommunicationDriver**
The standard driver that is expected for all methods of communication on the board

read(*offset*, *size*)
Read from the given address (offset) for N bytes (size)

write(*offset*, *value*)
Write to the given address (offset) for N bytes (len(value))

interface ironman.interfaces.**ICommunicationSlave**
Manages the communication with the programmable logic for us

__call__(*packet*)
A non-blocking I/O call passing along the packet

Returns the responses

__transaction__(*transaction*)
Handle a single transaction and return the response

parse_address(*address*)
Parses address and returns what function to call

set_hardware_manager(*hwmanager*)
Set the hardware manager that the slave communications with

interface ironman.interfaces.**IHardwareManager**
Our Hardware Maps manager

add(*hw_map*)
Add the Map object to the Hardware

check_address(*address*)
Given an address, checks if it is valid

check_data(*address*, *data*)
Given an address, checks if the data is a valid value to write

get_checksum(*route*)
Look up the checksum for a given map name (route)

get_node(*address*)
Given an address, return the node associated with it

get_route(*address*)
Given an address, return the route for it

raw_maps
A dictionary of the maps added so we can keep track which makes it easier to add and remove.

subtract(*route*)
Remove the route from the hardware manager

interface ironman.interfaces.**IHardwareMap**
Manages information about a single map, should be an overloaded dictionary

__init__(*xml*, *route*)
Initialize a hardware map object by giving it the data to parse and associate it with a route

isOk()
Whether or not the given hardware map is ok. Should just be a loop over *IHardwareNode.isOk()*.

parse(*xml*)
Parse the xml hardware map data to set things up

route
The route associated for this hardware map.

interface ironman.interfaces.**IHardwareNode**
Manages information about a single address. Simply a well-defined dictionary.

__init__(*node*, *hw_map*)
Initialize the node by giving it the parsed xml data as well as the hw_map

allowed
A list of allowed values for the node.

disallowed
A list of disallowed values for the node.

hw_map
The hardware map this is associated with.

isOk
Is the given node ok? EG: can't set allowed and disallowed objects at the same time and cannot block a node from being readable.

isValueValid(*val*)
Whether the given value is a valid value for the node

permissions
Mark the node's read/write capabilities.

readable
Is the given node readable?

writeable
Is the given node writeable?

interface ironman.interfaces.**IHistory**
Enhanced dictionary to store inbound and outbound packet pairs

record(*packet*)
record the packet

interface ironman.interfaces.**IIPBusPacket**
IPBus Packet object

__eq__(*other*)
Define a way to identify two packets as being equivalent. Best way is to compare the underlying structs

__init__(*blob*)
Packet is initialized with a data blob to decode. Determine if it is big or little endian.

__ne__(*other*)
This should just be `return not self.__eq__(other).`

_raw
The raw request packet

byteorder
The byte-order in the header. Should assert == 0xf.

packet_id
The id of the ipbus packet.

packet_type
The type of packet.

Value	Type
0x0	Control
0x1	Status
0x2	Re-send request
0x3-f	Reserved

protocol_version
The packet header protocol version. This does not check that the encapsulated transactions also match.

raw
The raw datagram blob.

request
The parsed request packet

reserved
Reserved. Should be 0x0.

response
The data response to be passed along to another function that builds the response packet. This should be a list [] to append responses to.

ironman.packet module

class ironman.packet.**IPBusPacket**(*blob*)
Bases: object

property byteorder

property packet_id

property packet_type

property protocol_version

property raw

property reserved

ironman.server module

class ironman.server.**FauxCP**(*dgen*)
Bases: twisted.internet.protocol.Protocol

dataReceived(*data*)
After receiving the data, generate the deferreds and add myself to it.

class ironman.server.**FauxFactory**(*dgen*)
Bases: twisted.internet.protocol.ServerFactory

buildProtocol(*addr*)
Create an instance of a subclass of Protocol.

The returned instance will handle input on an incoming server connection, and an attribute "factory" pointing to the creating factory.

Alternatively, L{None} may be returned to immediately close the new connection.

Override this method to alter how Protocol instances get created.

@param addr: an object implementing L{twisted.internet.interfaces.IAddress}

protocol
alias of *ironman.server.FauxCP*

ironman.server.**ServerFactory**(*proto*, *dgen*)

class ironman.server.**TCP**(*dgen*)
Bases: twisted.internet.protocol.Protocol

dataReceived(*data*)
After receiving the data, generate the deferreds and add myself to it.

class ironman.server.**TCPFactory**(*dgen*)
Bases: twisted.internet.protocol.ServerFactory

buildProtocol(*addr*)
Create an instance of a subclass of Protocol.

The returned instance will handle input on an incoming server connection, and an attribute "factory" pointing to the creating factory.

Alternatively, L{None} may be returned to immediately close the new connection.

Override this method to alter how Protocol instances get created.

@param addr: an object implementing L{twisted.internet.interfaces.IAddress}

protocol
alias of *ironman.server.TCP*

class ironman.server.**UDP**(*dgen*)
Bases: twisted.internet.protocol.DatagramProtocol

datagramReceived(*datagram*, *address*)
After receiving a datagram, generate the deferreds and add myself to it.

ironman.utilities module

class ironman.utilities.**PrintContext**
Bases: construct.core.Construct

ironman.utilities.**chunks**(*mylist*, *chunk_size*)
Yield successive n-sized chunks from a list.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

INDEX

Symbols

A

B

C

D

F

G

H

I

J

M

N

P

R

S

T

U

W