panda-server Documentation

Release 3.0b1-10-g54227ee-dirty

Michael Abbott, Tom Cobb

Sep 22, 2023

Contents

The Panda socket server provides a bridge between the register interface to the FPGA firmware controlling the Panda hardware and users and other software. The interface provided by this server is designed to be simple and robust.

The Panda firmware is structured into numerous functional blocks, with each block configured via a number of fields. This structure is directly reflected in the functional interface provided by this server: most commands read or write specific fields.

The socket server publishes two socket end points, one for configuration control the other for streamed data capture. The configuration control socket accepts simple ASCII commands and returns all data in readable ASCII format. The data capture socket supports no commands and simply streams captured data in a lightly structured binary format.

Source code	https://github.com/PandABlocks/PandABlocks-server
Documentation	https://PandABlocks.github.io/PandABlocks-server
Changelog	https://github.com/PandABlocks/PandABlocks-server/blob/master/CHANGELOG.rst

CHAPTER 1

Configuration Interface

Configuration commands are sent as newline (ASCII character 0x0A) terminated strings and all responses are also newline terminated. Three basic forms of command are accepted:

Query commands. These commands must be terminated by a single `?` character. The three possible responses are: an error message, a single value, or a list of values.

Assignment commands. These commands contain an = character, and are used for assigning values to fields. The two possible responses are an error message or `OK`.

Table assignment. Any command containing a < character (not preceded by `?` or =) is a table assignment command. The initial command may be followed by any number of lines of text, and *must* be terminated by an empty line. The two possible responses are an error message or `OK`.

The four possible responses are:

`ERR` error-message An error response is always sent as `ERR` followed by an error message.

`OK` Successful completion of either form of assignment command generates the `OK` response.

`OK` =value Successful completion of a query command returning a single value returns the value preceded by `OK =`.

Multi-line response Successful completion of a query command returning multiple values returns each value on a line by itself starting with `!` and ends the sequence with a line containing only `.`.

1.1 Example Commands

In the examples below, the command sent is shown preceded by < and the response with >: this is the syntax used by the helper tool `simulation/tcp_client.py`:

Simple server identification command including version information:

```
< *IDN?
> OK =PandA SW: 330bd94-dirty FPGA: 0.1.9 d1275f61 00000000
```

Interrogate list of fields provided by the `TTLIN` block:

```
< TTLIN.*?
> !VAL 0 bit_out
> !TERM 1 param enum
> .
```

Interrogate input termination for `TTLIN1`:

```
< TTLIN1.TERM?
> OK =High-Z
```

Set input termination:

```
< TTLIN1.TERM=50-Ohm
> OK
```

CHAPTER 2

Streaming Capture Interface

All bit and position values received and generated by PandA can be captured on an internally generated trigger and streamed to a data capture port. A variety of data capture, processing, and delivery options can be selected.

CHAPTER 3

Building

Before running `make` first copy the file `CONFIG.example` to `CONFIG` and edit according to your needs. If you wish to build the server or the driver then a Zynq cross-compiler will be needed, and if you wish to build the driver then a pre-build kernel will be needed – for this reference PandABlocks-rootfs and build the kernel target.

The panda-server Makefile supports four main build targets:

`driver` This builds the kernel driver required by the target server to communicate with the FPGA hardware. This target can only be built if the cross-compiler is on the path or `BINUTILS_DIR` has been configured in `CONFIG` to point to the directory containing the cross-compiler, and if `KERNEL_DIR` has been set to the location of a cross-build of the kernel previously generated by `PandABlocks-rootfs`. The built driver is placed in `$(BUILD_DIR)/driver/panda.ko`.

`server` This builds the PandA socket server to run on the target Zynq system. This target can only be built if the cross-compiler is installed as for `driver`, but there is no dependency on the kernel build. The built server is placed in `$(BUILD_DIR)/server/server`.

`sim_server` This builds the simulation version version of the PandA socket server. To run the server run the script `simserver` in the root directory.

`docs` This builds the documentation and places the results in `$(BUILD_DIR)/docs/html`.

3.1 Command Interface

The default server port for the command interface is port 8888. All commands and responses are in ASCII with lines separated with newline characters (ASCII character 0x0A).

All commands can be grouped into three forms (query, assignment, table assignment) and two targets (system and fields). There exactly four possible response formats (ok, ok with value, error, multiple value). This section describes this command interface.

The three basic command forms are:

Name	Format	Description
Query	target?	Interrogates *target* for current value, can return error, single value or a list of multiple values.
Assignment	target=value	Updates *target* with given value, can return error or success.
Table	target<format	Command may be followed by lines of text, *must* be terminated by blank line.

The four basic command responses are:

Name	Format	Description
Success	OK	Returned assignment and table commands to report successful update.
Value	OK =value	Successful return of single value from query command.
Error	ERR error	Error string returned on any command failure
Multi value	! value .	Any number of values can be returned, each preceded by !, and finally . by itself indicates end of input.

Command forms and their possible responses:

Form	Responses
Query	Error, Value, Multi value
Assignment	Error, Success
Table	Error, Success

Each individual query target will either return a single value or multi-value, as documented below.

Finally, there are two basic types of target: configuration commands and system commands.

3.1.1 Configuration Commands

The entire hardware interface to PandA is structured into "blocks" and "fields", and each field may have a number of "attributes" depending on the field type. This structure is reflected in the form of configuration commands which are tabulated below:

Command Syntax	Description
block[number].field?	Return current value of field.
block[number].field=value	Assign value to field.
block[number].field<[<][B]	Write table data to field.
block[number].field.attr?	Return current value of field attribute.
block[number].field.attr=value	Assign value to field attribute.
block[number].*?	Returns list of fields.
block[number].field.*?	Returns list of field attributes.

In all of these commands the number after the block is optional if there is only one instance of that block, and is ignored for the two .*? commands. See the description of the .TABLE fields for an explanation of the optional

format characters in the table write command.

3.1.2 System Commands

All system commands are prefixed with a leading * character. The simplest command is *IDN? which returns a system identification string:

```
< *IDN?
> OK =PandA SW: 330bd94-dirty FPGA: 0.1.9 d1275f61 00000000
```

The available system commands are tabulated here and listed in more detail below:

Command	Description
*IDN?	Device identification.
*ECHO string?	Echo.
*WHO?	List connected clients.
*BLOCKS?	List device blocks.
*DESC. block.field[.attr]? *DESC. block.field[].subfield?	Show description for field, attribute, or table subfield.
*ENUMS. block.field[.attr]? *ENUMS. block.field[].subfield?	List enumerations for field, attribute, or table subfield.
*CHANGES[.group]?	Report changes to values. *group* can be any of CONFIG, BITS, POSN, READ, ATTR, or TABLE.
*CHANGES[.group]=[E|S]	Reset reported changes, *group* as above.
*CAPTURE?	Report fields configured for capture.
CAPTURE.?	List all fields that can be captured.
*CAPTURE.name?	Interrogate capture options, *name* can be OPTIONS or ENUMS.
*CAPTURE=	Reset data capture.
*POSITIONS?	Enumerate possible capture positions.
*BITS?	Enumerate possible bit bus positions.
*VERBOSE=value	Control command logging.
*PCAP.field?	Special position capture status fields. *field* can be any of STATUS, CAPTURED, or COMPLETION.
*PCAP.field=	Position capture actions. *field* can be either ARM, or DISARM.
*SAVESTATE=	Triggers immediate save to file of the persistence file state.
*CLOCK_FREQ?	Returns currently configured system clock frequency

***IDN?** Returns system identification string, for example the following:

```
OK =PandA SW: 1.1 FPGA: 0.1.9 d1275f61 00000000 rootfs: PandA 1.1
```

The first field after "PandA" is the software version, the second field is the FPGA version, the third the firmware build number, and the fourth field identifies the supporting firmware. The final fields (prefixed rootfs:) identify the underlying system on which the server is running.

Note that the `rootfs:` identification is new to version 1.1 of PandA.

`*ECHO string?` Returns string back to caller. Not terribly useful. Note that the echoed string cannot contain any of ?, = or < characters, as this would cause the command to be mistaken for another command format! Example usage:

```
< *ECHO This is a test?
> OK =This is a test
```

`*WHO?` Returns list of client connections, for example:

```
< *WHO?
> !2015-12-04T14:30:40.403Z config 127.0.0.1:34185
> .
```

The first field is the time the connection was made, the second field is either `config` or `data` depending on whether the configuration or data port is connected, and the third field is the remote IP address and socket.

`*BLOCKS?` Returns a list of all the top level blocks in the system. The order in which the blocks is returned is somewhat arbitrary. For example (here the list has been shortened in the middle):

```
< *BLOCKS?
> !TTLIN 6
> !OUTENC 4
...
> !CLOCKS 1
> !BITS 1
> !QDEC 4
> .
```

Block and field commands can be used to interrogate each block. The number after each block records the number of instances of each block.

`*DESC.`block`?`
`*DESC.`block`.`field`?`
`*DESC.`block`.`field`.`attr`?`
`*DESC.`block`.`field`[].`subfield`?`

Returns description string for specified block, field, attribute, or table subfield eg:

```
< *DESC.TTLIN?
> OK =TTL input
< *DESC.TTLIN.TERM?
> OK =Select TTL input termination
< *DESC.TTLIN.TERM.INFO?
> OK =Class information for field
```

`*ENUMS.`block`.`field`?`
`*ENUMS.`block`.`field`.`attr`?`
`*ENUMS.`block`.`field`[].`subfield`?`

Returns list of enumerations for given field, attribute, or table subfield, if appropriate.

```
*CHANGES?
*CHANGES.CONFIG?
*CHANGES.BITS?
*CHANGES.POSN?
*CHANGES.READ?
*CHANGES.ATTR?
*CHANGES.TABLE?
```

Reports changes to the appropriate group of values. Changes are reported since the last request on the connection, and on the first request the current value for every field will be reported. The `*CHANGES?` command reports changes for all groups, otherwise one of the following groups can be selected:

CONFIG	Configuration settings
BITS	Bits on the system bus
POSN	Positions
READ	Polled read values
ATTR	Attributes (included capture enable flags)
TABLE	Table changes

For example:

```
< *CHANGES.CONFIG?
> !TTLIN1.TERM=High-Z
> !TTLIN2.TERM=50-Ohm
> !TTLIN3.TERM=High-Z
...
> !QDEC2.B=TTLIN1.VAL
> !QDEC3.B=TTLIN1.VAL
> !QDEC4.B=TTLIN1.VAL
> .
```

Here 804 (at the time of writing) lines have been deleted from the transcript! Now if we repeat the call we see that no further changes have happened until something is actually changed:

```
< *CHANGES.CONFIG?
> .
< TTLOUT4.VAL=TTLIN3.VAL
> OK
< *CHANGES.CONFIG?
> !TTLOUT4.VAL=TTLIN3.VAL
> .
```

Note that for tables only the fact that the table has changed is shown, no attempt is made to show the current table value:

```
< *CHANGES.TABLE?
> !PCOMP1.TABLE<
> !PCOMP2.TABLE<
> !PCOMP3.TABLE<
> !PCOMP4.TABLE<
> !PGEN1.TABLE<
> !PGEN2.TABLE<
> !SEQ1.TABLE<
> !SEQ2.TABLE<
```

(continues on next page)

(continued from previous page)

```
> !SEQ3.TABLE<
> !SEQ4.TABLE<
> .
```

*CHANGES=[E| S]
*CHANGES.CONFIG=[E| S]
*CHANGES.BITS=[E| S]
*CHANGES.POSN=[E| S]
*CHANGES.READ=[E| S]
*CHANGES.ATTR=[E| S]
*CHANGES.TABLE=[E| S]

These commands reset the change information for the corresponding group of information so that only changes occuring after the reset are reported, or so that all changes are reported. If = or =E (for End) is specified then only new changes are reported, if =S (for Start) then change reporting is reset to the start as for a new connection. For example:

```
< TTLIN1.TERM=50-Ohm
> OK
< *CHANGES=
> OK
< *CHANGES.CONFIG?
> .
```

***CAPTURE?** This returns a list of all positions and bit masks that will be written to the data capture port. This list is controlled by setting the .CAPTURE attribute on the corresponding position fields.

***CAPTURE.*?** This returns a list of all fields that can be configured for capture. This includes all pos_out and ext_out fields.

***CAPTURE.OPTIONS?** Lists the available capture options for pos_out fields. The available options are "Value", "Diff", "Sum", "Mean, "Min", "Max", "StdDev". Availability of the last option "StdDev" depends on the FPGA configuration.

***CAPTURE.ENUMS?** Generates a curated list of capture option selections. This is designed to be used for presenting lists of available capture options as an enumeration. Returns the same as calling *ENUMS.name.field.CAPTURE? on any pos_out field.

***CAPTURE=** This resets all .CAPTURE flags to zero so that no data will be captured.

***POSITIONS?** This command lists all available position capture fields in order.

***BITS?** This command lists all available bit bus positions, but not including the special values ZERO and ONE.

***VERBOSE=value** If *VERBOSE=1 is set then every command will be echoed to the server's log. Set *VERBOSE=0 to restore normal quiet behaviour.

*PCAP.STATUS?
*PCAP.CAPTURED?
*PCAP.COMPLETION?

Interrogates status of position capture:

STATUS	Returns string with three fields: "Busy" or "Idle", followed by the number of connected readers, and the number taking data.
CAPTURED	Returns number of samples captured in the current or most recent data capture.
COMPLETION	Returns completion status from most recent data capture, as listed in the table below.

The completion codes have the following meaning:

Busy	Capture in progress.
Ok	Capture completed without error or intervention.
Disarmed	Capture was manually disarmed by `*PCAP.DISARM=` command.
Framing error	Data capture framing error, probably due to incorrectly configured capture.
DMA data error	Internal data error, should not occur.
Driver data overrun	Data capture too fast, internal buffers overrun. Can also occur if PandA processor overloaded.

`*PCAP.ARM=`
`*PCAP.DISARM=`

Top level capture control:

ARM	Initiates data capture. Will fail if capture already in progress, or no fields configured for capture.
DISARM	Halts ongoing data capture.

`*SAVESTATE=` Updates the persistence state file (as configured on the command line when launched) with the current state. Returns after a file system `sync` call, so it is safe to power-off the system after this command has completed.

`*CLOCK_FREQ?` Returns currently configured FPGA clock frequency as used to convert between times in natural units and times in clock ticks.

3.2 Blocks, Fields and Attributes

The set of hardware blocks can be interrogated with the `*BLOCKS?` command:

```
< *BLOCKS?
> !TTLIN 6
> !OUTENC 4
> !PCAP 1
> !PCOMP 4
> !TTLOUT 10
> !ADC 8
> !DIV 4
> !INENC 4
> !SLOW 1
> !PGEN 2
> !LVDSIN 2
```

(continues on next page)

(continued from previous page)

```
> !POSITIONS 1
> !POSENC 4
> !SEQ 4
> !PULSE 4
> !SRGATE 4
> !LUT 8
> !LVDSOUT 2
> !COUNTER 8
> !ADDER 1
> !CLOCKS 1
> !BITS 1
> !QDEC 4
> .
```

For each block the number after the block tells us how many instances there are of the block. Each block is controlled and interrogated through a number of fields, and the *block*`.*?` command can be used to interrogate the list of fields:

```
< TTLIN.*?
> !VAL 1 bit_out
> !TERM 0 param enum
> .
```

This tells us that block `TTLIN` has two fields, `TTLIN.VAL` and `TTLIN.TERM`. The first field after the field name is a sequence number for user interface display, and the rest of each response describes the "type" of the field. In this case we see that `TTLIN.VAL` is a `bit_out` field, which means can be used for bit data capture and can be connected to any `param bit_mux` field as a data source.

Each field has one or more attributes depending on the field type. The list of attributes can be interrogated with the *block*`.`field`.*?` command:

```
< TTLIN.VAL.*?
> !CAPTURE_WORD
> !OFFSET
> !INFO
> .
< TTLIN.TERM.*?
> !INFO
> .
```

All fields have the `.INFO` attribute, which just repeats the type information already reported, eg `TTLIN1.VAL.INFO?` returns `bit_out` (note that a block number must be specified when interrogating fields and attributes).

3.2.1 Field Types

Each field type determines the set of attributes available for the field. The types and their attributes are documented below.

Field type	Description
`param` subtype	Configurable parameter. The *subtype* determines the precise behaviour and the available attributes.
`read` subtype	A read only hardware field, used for monitoring status. Again, *subtype* determines available attributes.
`write` subtype	A write only field, *subtype* determines possible values and attributes.
`time`	Configurable timer parameter.
`bit_out`	Bit output, can be configured as bit input for `bit_mux` fields.
`pos_out`	Position output, can be configured for data capture and as position input for `pos_mux` fields.
`ext_out` extra	Extended output values, can be configured for data capture, but not available on position bus.
`bit_mux`	Bit input with configurable delay.
`pos_mux`	Position input multiplexer selection.
`table`	Table data with special access methods.

`param` subtype All fields of this type contribute to the `*CHANGES.PARAM` change group and are used to configure the behaviour of the corresponding block. Fields of this type are used for input configuration and other behavioural settings.

`read` subtype All fields of this type contribute to the `*CHANGES.READ` change group, but are only checked when either the field is read or the change group is polled. Fields of this type are used for monitoring the internal status of a block, and they cannot be written to.

`write` subtype Fields of this type can only be written and are used for immediate actions on a block. The `action` subtype is used to support actions without any parameters, for example the following command forces a soft reset on the given pulse block:

```
< PULSE1.FORCE_RESET=
> OK
```

`time` Fields of this type are used for configuring delays. They also contribute to `*CHANGES.PARAM`. The following attributes are supported by fields of this type:

`UNITS` This attribute can be set to any of the strings `min`, `s`, `ms`, or `us`, and is used to interpret how values read and written to the field are interpreted.

`RAW` This attribute can be read or written to report or set the delay in FPGA ticks.

`MIN` This reports the minimum valid value for this field in the currently selected units.

The `UNITS` attribute determines how numbers read or written to the field are interpreted. For example:

```
< PULSE1.DELAY.UNITS=s
> OK
< PULSE1.DELAY=2.5
> OK
< PULSE1.DELAY.RAW?
> OK =312500000
< PULSE1.DELAY.UNITS=ms
> OK
< PULSE1.DELAY?
> OK =2500
```

Note that changing `UNITS` doesn't change the delay, only how it is reported and interpreted.

`bit_out` Fields of this type are used for block outputs which contribute to the internal bit system bus, and they contribute to the `*CHANGES.BITS` change group. They can be captured via the appropriate `PCAP.BITSn`

block as reported by the CAPTURE_WORD attribute.

The following attributes are supported by fields of this type:

CAPTURE_WORD This identifies which pos_out value can be used to capture this bit.

OFFSET This is the bit offset into the captured word of this particular bit.

For example:

```
< TTLIN1.VAL.CAPTURE_WORD?
> OK =PCAP.BITS0
< TTLIN1.VAL.OFFSET?
> OK =2
```

This tells us that if PCAP.BITS0 is captured then TTLIN1.VAL can be read as bit 2 of this word, counting from the least significant bit.

The field itself can be read to return the current value of the bit.

pos_out Fields of this type are used for block outputs which contribute to the internal position bus, and they contribute to the *CHANGES.POSN change group. The following attributes support capture control:

CAPTURE This can be set to manage capture of this field. One of the following enumeration values can be written to this field:

Value	Description
No	Capture is disabled for this field.
Value	The value at the time of trigger will be captured.
Diff	The difference of values is captured.
Sum	The sum of all valid values is captured. This is a 64-bit value, and may be further scaled if PCAP.SHIFT_SUM is set.
Mean	The average of all valid values is captured.
Min	The minimum of all valid values is captured.
Max	The maximum of all valid values is captured.
Min Max	Both minimum and maximum values are captured.
Min Max Mean	All three values, minimum, maximum, average are captured.

The following attributes support formatting of the field when reading it: the current value is returned subject to the formatting rules described below.

OFFSET, SCALE These numbers can be set to configure the conversion from the underlying position to the value captured when scaling is enabled and read from the SCALED attribute.

UNITS This field can be set to any UTF-8 string, and is provided for the convenience of the user interface and is returned as part of the data capture heading.

SCALED This returns the scaled value computed as

value * scale + offset

ext_out extra Fields of this type represent values that can be captured but which are not present on the position bus. These fields also support one capture control field:

CAPTURE As for pos_out, can be set to control capture of this field:

Value	Description
No	This field will not be captured.
Value	This field will be captured.

The *extra* field determines the detailed behaviour of this field, and will be one of the following values:

extra value	Description
`timestamp`	Timestamps in clock ticks with optional scaling to seconds on data capture.
`samples`	Special internal field for counting captured samples.
`bits`	Used to implement bit-bus readout fields. Fields of this sub-type implement an extra `BITS` field.

Fields of type `ext_out bits` implement an extra attribute:

BITS This returns a list of all bit fields associated with this field. Fields of this type can be used to capture a snapshot of the bit bus at the trigger time.

bit_mux Bit input selectors for blocks. Each of these fields can be set to the name of a corresponding `bit_out` field, for example:

```
< TTLOUT1.VAL=TTLIN1.VAL
> OK
```

There are two attributes:

DELAY This can be set to any value between 0 and `MAX_DELAY` to delay the bit input to the block by the specified number of clock ticks.

MAX_DELAY This returns the maximum delay that can be set for this input.

pos_mux Position input selectors for blocks. Each of these fields can be set to the name of a corresponding `pos_out` field, for example:

```
< ADDER1.INPA=ADC2.OUT
> OK
```

table Values of this type are used for long tables of numbers. This server imposes no structure on these values apart from treating them as an array of 32-bit integers.

Tables values are written with the special < syntax:

block number.field<	Normal table write, overwrite table
block number.field<<	Normal table write, append to table
block number.field<B	Base-64 table write, overwrite table
block number.field<<B	Base-64 table write, append to table

For "normal" table writes the data is sent as a sequence of decimal numbers in ASCII, and the whole sequence must be terminate by an empty blank line. For base-64 writes the data is sent in base-64 format, for example:

```
< SEQ3.TABLE<B
< TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1
<
> OK
< SEQ3.TABLE.LENGTH?
> OK =12
```

Note that when data is sent in base-64 format, each individual line must encode a multiple of four bytes, otherwise the write will be rejected.

The following attributes are provided by this field type:

MAX_LENGTH This is the maximum number of 32-bit words which can be stored in the table.

LENGTH This is the current number of words in the table.

B This read-only attribute returns the content of the table in base-64.

FIELDS This returns a list of strings which can be used to interpret the content of the table. Each line returned is of the following format:

```
left:right field-name subtype
```

Here *left* and *right* are bit field indices into a single table row, consisting of a number of 32-bit words concatenated (in little-endian order) with bits numbered from 0 in the least significant position up to 32*ROW_WORDS-1, and *left* >= *right*. The name of the field is given by *field-name*, and *subtype* can be one of `int`, `uint`, or `enum`. If *subtype* is `enum` then the list of enums can be interrogated through the command

```
*ENUMS.block.table[].field?
```

where *block*, *table*, *field* are appropriate names.

ROW_WORDS Returns the number of 32-bit words in a single row of the table. This can be used to help interpret the `FIELDS` result.

3.2.2 Field Sub-Types

The following field sub-types can be used for `param`, `read` and `write` fields.

uint [*max-value*] This is the most basic type: the value read or written is an unsigned 32-bit number. There is one fixed attribute:

MAX This returns the maximum value that can be written to this field.

int Similar to `uint`, but signed, and there is no upper limit on the value.

scalar *scale* [*offset* [*units*]] Floating point values can be read or written, and are converted from and to the underlying signed integer type via the equations below:

value = scale * raw + offset
raw = (value - offset) / scale

The following attributes are supported:

UNITS Returns the configured units string.

RAW Returns the underlying unconverted integer value.

SCALE Returns the configured scaling factor.

OFFSET Returns the configured scaling offset.

bit A value which is 0 or 1, there are no extra attributes.

action A value which cannot be read and always writes as 0. Only useful for `write` fields.

lut This field sub-type is used for the 5-input lookup table function calculation field. This field can be set to any valid logical expression generated from inputs `A` to `E` using the standard operators `&`, `|`, `^`, `~`, `?:` from C together with = for equality and => for implication (`A=>B` abbreviates `~A|B`). All operations have C precedence, = has the same precedence as == in C, and => has precedence between `|` and `?:`.

The following attribute is supported:

RAW This returns the corresponding lookup table assignment as a 32-bit number.

For example:

```
< LUT2.FUNC=A=>B?C:D
> OK
< LUT2.FUNC?
> OK =A=>B?C:D
< LUT2.FUNC.RAW?
> OK =0xF0CCF0F0
```

enum Enumeration fields define a list of valid strings which can be written to the field. To interrogate the list of valid enumeration values use the `*ENUMS` command, for example:

```
< *ENUMS.TTLIN1.TERM?
> !High-Z
> !50-Ohm
> .
```

time Converts between time in specified units and time in FPGA clock ticks. The following attributes are supported:

UNITS This attribute can be set to any of the strings `min`, `s`, `ms`, or `us`, and is used to interpret how values read and written to the field are interpreted.

RAW This attribute can be read or written to report or set the delay in FPGA ticks.

3.2.3 Summary of Sub-Types

Sub-type	Attributes	Description
uint	MAX	Possibly bounded 32-bit unsigned integer value
int		Unbounded 32-bit signed integer value
scalar	RAW, UNITS, SCALE, OFFSET	Scaled signed floating point value
bit		Bit: 0 or 1
action		Write only, no value
lut	RAW	5 input lookup table logical formula
enum	LABELS	Enumeration selection
time	RAW, UNITS	Time intervals converted to FPGA ticks

3.2.4 Summary of Attributes

Field (sub)type	Attribute	Description	R	W	C	M
(all)	INFO	Returns type of field	R			
uint	MAX	Maximum allowed integer value	R			
scalar	RAW	Underlying integer value	R	W		
	UNITS	Configured units for scalar	R			
	SCALE	Configured scaling factor for scalar	R			
	OFFSET	Configured scaling offset for scalar	R			
lut	RAW	Computed Lookup Table 32-bit value	R			
time	UNITS	Units and scaling selection for time	R	W	C	
	RAW	Raw time in FPGA clock cycles	R	W		
	MIN	Minimum valid setting (for type only)	R			
bit_out	CAPTURE_WORD	Capturable word containing this bit	R			
	OFFSET	Offset of this bit in captured word	R			
bit_mux	DELAY	Bit input delay in FPGA ticks	R	W	C	
	MAX_DELAY	Maximum valid delay	R			
pos_out	CAPTURE	Position capture control	R	W	C	
	OFFSET	Position offset	R	W	C	
	SCALE	Position scaling	R	W	C	
	UNITS	Position units	R	W	C	
	SCALED	Position after applying scaling	R			
ext_out bits	BITS	List of bit_out fields	R			M
table	MAX_LENGTH	Maximum table row count	R			
	LENGTH	Current table row count	R			
	B	Table data in base-64	R			M
	FIELDS	Table field descriptions	R			M
	ROW_WORDS	Number of words in a table row	R			

Key:

R Attribute can be read

W Attribute can be written

C Attribute contributes to `*CHANGES.ATTR` change set

M Attribute returns multiple value result.

3.3 Data Capture

3.3.1 Capture Configuration

Both `pos_out` and `ext_out` fields can be configured for data capture through the data capture port by setting the appropriate value in the `CAPTURE` attribute. The possible capture settings depend on the field type as follows:

`pos_out`

Value	Description
No	Capture is disabled for this field.
Value	The value at the time of trigger will be captured.
Diff	The difference of values is captured.
Sum	The sum of all valid values is captured. This is a 64-bit value, and may be further scaled if `PCAP.SHIFT_SUM` is set.
Mean	The average of all valid values is captured.
Min	The minimum of all valid values is captured.
Max	The maximum of all valid values is captured.
Min Max	Both minimum and maximum values are captured.
Min Max Mean	All three values, minimum, maximum, average are captured.

`ext_out`

Value	Description
No	This field will not be captured.
Value	This field will be captured.

3.3.2 Data Capture Port

The default server port for the data interface is port 8889. The initial exchange is in ASCII with newline separated lines, subsequent data communication is as selected in the initial connection.

Data capture proceeds as follows:

1. Connection to the data server port, default 8889.
2. Send capture options string followed by newline. The newline character is mandatory.
3. The server will respond with `OK` unless there was an error parsing the capture options, or if the `NO_STATUS` option was specified. If there was an error then the server responds with `ERR` followed by an error message and the connection is closed.
4. The server will now ignore all further input from the client, and the connection will pause until data capture is started via the `*PCAP.ARM=` command.
5. At the beginning of a round of data capture or "experiment", a header detailing the data to be sent and data format is sent in ASCII followed by an empty line. If `NO_HEADER` was selected then the header and blank line are omitted.
6. Captured data is sent in the requested format until the experiment is complete (either internally disarmed or disarmed via the `*PCAP.DISARM=` command), or there is a communication problem.
7. At the end of the experiment a completion code is sent as a single line in ASCII starting with `END`, unless `NO_STATUS` was specified.
8. Unless `ONE_SHOT` was specified the server will pause until the next experiment (step 4).

Capture Options

A line of capture options *must* be sent after initial connection before any data will be sent. This is a list of any of the following options separated by whitespace ending with a newline character.

ASCII	Specifies that data is to be sent as ASCII numbers.	1	D
BASE64	Binary data will be sent as a stream of base64 strings.	1	
FRAMED	Binary data is sent as a sequence of sized frames.	1	
UNFRAMED	Binary data is sent as a raw stream of bytes.	1	R
SCALED	All scalable data is scaled and sent as doubles.	2	D
RAW	The captured binary data is sent without processing.	2	
NO_HEADER	The data header is omitted.		R
NO_STATUS	The connection and end of experiment status strings are omitted.		R
ONE_SHOT	Only one experiment will be transmitted.		R
XML	The header will be sent in XML format.		
BARE	Selects `UNFRAMED RAW NO_HEADER NO_STATUS ONE_SHOT`		
DEFAULT	Default options.		D

Key:

D Default option if no other option specified.

R Options selected in response to `BARE` option.

1 Data transmission formats, one of these will be selected.

2 Data processing formats, one of these will be selected.

Data Transport Formatting

Note that all binary data is sent with the lowest order byte first.

`ASCII` Each value is formatted as an ASCII number, and transmitted with one line per captured sample.

`BASE64` The stream of binary data is converted to base64 strings and transmitted as a series of lines until the experiment is complete. Each base64 string is preceded by a single space, so the end of the stream is easy to identify.

`FRAMED` In `FRAMED` mode the captured binary data is sent in blocks of unpredictable size. Each block is preceded by 8 bytes. The first four bytes are `BIN` followed by space, the remainind four bytes are the length of the data block in bytes *including* the 8 byte header.

`UNFRAMED` In `UNFRAMED` mode the captured binary data is sent as is. In this mode it is difficult or impossible to reliably detect the end of the data stream, so normally this is best combined with `NO_STATUS` and `ONE_SHOT`.

Data Header

At the beginning of each experiment the following information is sent:

arm_time	UTC Timestamp sampled in correspondence of the ARM command
missed	Number of samples missed by late data port connection.
process	Data processing option: Scaled, Unscaled, or Raw.
format	Data delivery formatting: ASCII, Base64, Framed, or Unframed.
sample_bytes	Number of bytes in one sample unless `format` is `ASCII`.
fields	Information about each captured field.

For each field the following information is sent:

name	Name of captured field.	
type	Data type of transmitted field after data processing.	
capture	Value of `CAPTURE` field used to enable this field.	
scale	Scaling factor if scaled field.	S
offset	Offset if scaled field.	S
units	Units string if scaled field.	S

Key:

S Only present if scaled field

If the `XML` option is selected the header is structured as a single `header` element containing `data` and `fields` elements.

The `type` field can be one of the following strings:

String	Bytes	Description
int32	4	Used for scalable values sent in unscaled modes.
uint32	4	Used for bit masks.
int64	8	Used for raw ADC mean and unscaled 48-bit encoder data.
double	8	Used for all scaled values when `SCALED` selected.

Experiment Completion

At the end of each capture experiment a single line is sent, eg:

```
END 10 Ok
```

This specifies the number of samples sent and gives a completion code, which can be one of the following values:

Ok	Experiment completed without intervention.
Disarmed	Experiment manually completed by `*PCAP.DISARM=` command.
Early disconnect	Client disconnect detected.
Data overrun	Client not taking data quickly or network congestion, internal buffer overflow.
Framing error	Triggers too fast for configured data capture.
Driver data overrun	Probable CPU overload on PandA, should not occur.
DMA data error	Data capture too fast for memory bandwidth.

High performance mode

To get the highest performance, use `FRAMED RAW` mode. This activates a special passthrough mode which avoids copying memory as much as possible. In tests it has been capable of sustaining 60MBytes/s when panda-webcontrol is not installed. The downside to this mode is that if capture fails for any reason, then the last Framed block of data that the server sent should be discarded as it will have been corrupted while being sent.

Examples

Some examples of data capture for different options follow:

Default:

```
arm_time: 2021-05-26T10:34:06.133Z
missed: 0
process: Scaled
format: ASCII
fields:
 PCAP.CAPTURE_TS double Trigger
 COUNTER1.OUT double Triggered scale: 1 offset: 0 units:
 COUNTER2.OUT double Triggered scale: 1 offset: 0 units:
 PGEN1.OUT double Triggered scale: 1 offset: 0 units:

 1e-06 0 0 262143
 3e-06 0 0 262142
 5e-06 0 0 262141
 7e-06 0 0 262140
 9e-06 0 0 262139
END 5 Ok
```

BASE64:

```
arm_time: 2021-05-26T10:34:06.133Z
missed: 0
process: Scaled
format: Base64
sample_bytes: 32
fields:
 PCAP.CAPTURE_TS double Trigger
 COUNTER1.OUT double Triggered scale: 1 offset: 0 units:
 COUNTER2.OUT double Triggered scale: 1 offset: 0 units:
 PGEN1.OUT double Triggered scale: 1 offset: 0 units:

 ju21oPfGsD4AAAAAAAAAAAAAAAAAAAAAAAAPj/D0FU5BBxcyrJPgAAAAAAAAAAAAAAAAAAAAA
 AAAA8P8PQfFo44i1+NQ+AAAAAAAAAAAAAAAAAAAAAAAAADo/w9BuF8+WTFc3T4AAAAAAAAAAAA
 AAAAAAAAAAAAOD/D0E/q8yU1t/iPgAAAAAAAAAAAAAAAAAAAAAAAA2P8PQQ==
END 5 Ok
```

XML:

```
<header>
<data arm_time="2021-05-26T10:35:06.107Z" missed="0" process="Scaled" format="ASCII" /
↪>
<fields>
<field name="PCAP.CAPTURE_TS" type="double" capture="Trigger" />
<field name="COUNTER1.OUT" type="double" capture="Triggered" scale="1"
offset="0" units="" />
<field name="COUNTER2.OUT" type="double" capture="Triggered" scale="1"
offset="0" units="" />
<field name="PGEN1.OUT" type="double" capture="Triggered" scale="1" offset="0"
units="" />
</fields>
</header>

 1e-06 0 0 262143
 3e-06 0 0 262142
 5e-06 0 0 262141
 7e-06 0 0 262140
 9e-06 0 0 262139
END 5 Ok
```

3.4 Building and Configuring Panda Server

Setting up the build for the Panda Socket Server requires configuring a number of dependencies and creating a suitable `CONFIG` file in the root directory so that they can be found.

3.4.1 Dependencies

The following dependencies must be configured before any part of the server can be built.

Zynq cross-compiler toolchain This can be downloaded as part of the Xilinx Vivado build environment for working with Zynq, or probably any ARMv7-A cross-compiler can be used. This is needed to build all applications running on Panda.

PandABlocks-FPGA This part of the Panda project must be available before the server can be built, as it contains a configuration file defining the low level register interface to the Panda firmware.

PandABlocks-rootfs This part of the Panda project is required in order to provide a working kernel build tree, and to provide the zpkg build tool.

3.4.2 Setting up the `CONFIG` file

Start by copying the file `CONFIG.example` to `CONFIG` in the base directory, and edit the file by commenting out lines as appropriate and editing them.

The following symbols must be set to point to the appropriate dependencies:

`BINUTILS_DIR` If the Zynq cross-compiler toolchain is not on the path, this must be set in order to build the kernel module and the target build. This symbol is not required for building the simulation server or the documentation.

`KERNEL_DIR` In order to build the kernel module, this symbol must be pointed to the kernel build tree generated by the PandABlocks-roots build.

`PANDA_ROOTFS` The zpkg build tool is found here.

The following symbols can all be left at their default values:

`BUILD_DIR` This configures where the built files will be placed.

`PYTHON` This configures which Python interpreter will be used for building.

`SPHINX_BUILD` The sphinx-build Python script used for building the documentation.

`DEFAULT_TARGETS` This determines which makefile targets are generated when `make` is run without specifying a particular target, or when `make default` is run.

3.4.3 Build Targets

The following build targets for the top level makefile are useful:

`default` Builds all the targets specified by `$(DEFAULT_TARGETS)`, by default this list is: `driver`, `server`, `sim_server`, `docs`, `zpkg`.

`driver` Builds the kernel driver module.

`server` Builds the server version to run on Panda.

`sim_server` Builds a simulation version of the server to run on the local PC.

`docs` Builds the documentation.

zpkg Builds the final `panda-server` zpgk file.

clean Removes the entire `$(BUILD_DIR)` directory.

3.4.4 Generated Files

In the `$(BUILD_DIR)` directory the following subdirectories and files will be found. In practice the `.zpg` file and `html/` directory will be wanted.

driver/ The kernel module required for hardware access is built here.

`server/`
`sim_server/`

These two directories are used to build the server to run on Panda, and a simulation server to run on the local PC.

html/ The documentation is built in html format in this directory.

`panda-server@`version`.zpg`
`zpkg-panda-server/`

A zpkg for the server is built here.

3.5 Configuration Files

On startup the Panda Server loads its configuration from three files: `config`, `registers`, `description`. These are loaded from the `config_d` directory in build directory when running the basic simulation, and on Panda the configuration files are loaded from `/opt/share/panda/config_d`.

The syntax of each configuration file is documented here. The format of each field definition closely follows the format documented in *Blocks, Fields and Attributes*.

The three files have the following distinct functions.

File	Description
`config`	Configuration: defines list of blocks, fields in each block, and the behaviour of each field.
`registers`	Registers: for each block and field defines the associated register offsets.
`description`	Description: optionally defines a description string for each block and field.

All files have a common structure and indentation is used for structure. Comments begin with `#`, blocks are identified by their name in the first column, and fields are listed at the next level of indentation.

3.5.1 Configuration file `config`

This file defines all of the blocks and fields available to this instance of Panda and is processed first.

The syntax of a block definition is:

```
block-name [ "[" count "]" ]
    [ field-definition ]*
```

This means that a block definition consists of a *block-name*, which can be any word used to name the block, optionally followed by a block *count* enclosed in square brackets, followed by any number of indented *field-definition*s.

The syntax of a field definition is:

```
field-name field-type [ field-type-data ]
```

The *field-name* names the field, the *field-type* is a single word from the list documented below, and the *field-type-data* depended on the field type as documented.

The *field type* determines the basic function of the field, what actions can be performed on the field, and how the field interacts with the hardware. Typically each field corresponds to a single register or function of the block.

Many field type have an associated *field subtype* which is used to convert between the values show to the user of the server and the values written to or read from registers.

Field type

The following field types are defined.

Field	
`param` field-subtype [= value]	
`read` field-subtype	
`write` field-subtype	
`time` [> min_value]	
`bit_out`	
`pos_out` [scale [offset [units]]]	
`ext_out` (`timestamp` | `samples` | `bits` group)	
`bit_mux` [= value]	
`pos_mux`	
`table`	

`param` field-subtype [= value] A `param` field is used to define a single 32-bit value written to a register. The *field-subtype* must be specified. Optionally an initial value (only relevant when no state file has been loaded) can be specified. In this case the initial value is read as a raw unsigned integer which is written directly to the hardware on startup.

`read` field-subtype A `read` field is used for read-only registers. The *field-subtype* must be specified.

`write` field-subtype A `write` field is used for write-only registers which trigger immediate actions on a block. The *field-subtype* must be specified, and the *action* subtype is useful for `write` fields which take no data.

`time` [> min_value] Time fields behave like `param` fields, but need special treatment because the underlying value is 64-bits and so two registers need to be written.

If desired a minimum valid value can be specified as *min_value*. This will prevent the writing of values less than this value and can be read as the `.MIN` attribute.

`bit_out` This identifies an output bit.

`pos_out` [scale [offset [units]]] This identifies a position bus output. Optionally default values for the scale, offset, and units fields can be specified in the config file. Note that these are only effective when there is no persistence file to load.

`ext_out` ext-extra This identifies an entry on the extension bus which needs special treatment. The *ext-extra* field must be one of the following values:

ext-extra	Description
`timestamp`	Capture timestamp as a 64-bit value
`samples`	Captures sample count for data capture.
`bits` group	Defines fields which allow the bit bus to be captured. The group number identifies which 32-bit group of 128 bits is captured.

`bit_mux` [= value]
`pos_mux`

These two are configuration settings for selecting inputs, and behave like `param` fields. As for `param` a default value can be assigned to `bit_mux`, but the only useful value is 129 (`ONE`).

`table` [row-words] Tables are treated specially.

Field subtype

The following field subtype can follow a `param`, `read` or `write` field type:

Type	
`uint` [max-value]	
`int`	
`scalar` scale [offset [units]]	
`bit`	
`action`	
`lut`	
`enum`	
`position`	
`time`	

Note that `enum` must be followed by indented lines each consisting of a number followed by a string: the string is the enumeration value written to the user, the number is the value written to the register.

3.5.2 Register file `registers`

This file defines the register assignments for each block and register. The body of this file should contain a sequencer of block and field definitions repeating the `config` file, except that the field type specification is replaced by a type specific register definition.

So a block definition is:

```
block-name { [ "S" ] block-register | "X" } [ extension-module ]
    [ field-definition ]*
```

If the *block-register* number is prefixed with `S` then the same block register number can be shared with multiple blocks: this allows a single hardware implementation to be presented as multiple software blocks. If `X` is used instead of specifying *block-register* then no fields can use registers, so must be extension fields with no register linkage.

The register number can be followed by an *extension-module* which is used to identify this block to the extension server, and will enable use of the extension register syntax defined below.

A field definition is:

```
field-definition = field-name register-specification
```

where *register-specification* depends on the associated field type as follows:

Class	Register syntax
`param`	register | write-extension
`read`	register | read-extension
`write`	register | write-extension
`time`	low-register high-register
`bit_out`	(bit-index)N
`pos_out`	(pos-index)N
`ext_out timestamp`	ext-index ext-index
`ext_out` other	ext-index
`bit_mux`	register
`pos_mux`	register
`table`	`short` size init-reg fill-reg length-reg
`table`	`long` 2^size base-reg length-reg

where the syntax `(...)N` means that the given register number is repeated N times where N is the number of instances of the block. See below for an explanation of *read-extension* and *write-extension*.

Extension register syntax

If the extension server is enabled (with the `-X` command line option on the server) and if an extension module has been loaded as part of the block specification then `param`, `read`, and `write` subtypes can all be redirected to the extension server using the *read-extension* and *write-extension* syntax:

```
read-extension = [ read-reg ]* "X" field-spec
write-extension = [ read-reg ]* [ "W" [ write-reg ]* ] "X" field-spec
```

In this syntax *field-spec* is passed through to the associated *extension-module* to create the binding between this field and the extension server. The specified *read-reg*s and *write-reg*s will be used when processing this field.

See *Extension Server* for more details on extension fields.

3.5.3 Description file `description`

The entire content of the description file is optional. The basic syntax is:

```
block-name block-description
    [ field ]*
```

where field is:

```
field-name field-description
```

and the description is any newline terminated string in UTF-8 format.

3.6 Extension Server

The extension server is used to implement custom fields. The extension server runs alongsize the Panda socket server and supports the loading of Python modules with a lightweight remote procedure calling interface from the socket server.

As described in *Configuration Files*, individual *read*, *write*, *param* fields can be specified to take their values from and write to the extension register. At the same time, these extension read and write operations can be linked to hardware registers.

The extension server is started alongside the socket server, and on PandA looks in `/opt/share/panda-fpga/extensions` for extension module files. During development any Python module can be specified as a container for extension sub-modules.

Connections from the socket server to the extension server are established while reading the *registers* file. Firstly, if a block has an extension module specified then this module is loaded from the extensions directory and is associated with the block. Next, each extension field is processing by calling the appropriate `parse_read()` or `parse_write()` methods.

Depending on the register configuration any number of hardware block registers can be read or written during processing of an extension field.

3.6.1 Extension Modules

Each extension module must export a class constructor named *Extension*. This takes one argument and must support two methods *parse_read* and *parse_write*

class Extension(*count*)

This class must be defined by each extension module. The class will be instantiated in response to loading a block register definition of the form:

```
block-name [ "S" ] block-register extension-module
```

The parameter *count* is set to the number of instances specified for the block in the *config* file. This class must provide the following methods, as appropriate, to support read and write register fields:

parse_read(*field_spec*)

This is called in response to a *read-extension* line in the register file of the form:

```
[ read-reg ]* "X" field-spec
```

The *field-spec* is passed as a string to *parse_read()*, and this method must return a callable of the following form:

value = read_register(block_num, read_reg1, ..., read_regN)

The first argument *block_num* is the number of the block instance being called (starting from 0), and is guaranteed to be less than *count* as passed to the *Extension* constructor.

The remaining *read_reg1* ... *read_regN* argument must match the number of arguments specified in the *read-reg* block of the register file. These will be populated by reading the corresponding block hardware registers before this function is called.

The value returned must be a single integer, this is the value returned when reading this field.

parse_write(*field_spec*)

This is called in response to a *write-extension* line in the register file of the form:

```
[ read-reg ]* [ "W" [ write-reg ]* ] "X" field-spec
```

As for *parse_read()*, *field-spec* is passed as the argument, and a callable must be returned, of the following form:

(write_reg1, ..., write_regM) = write_register(block_num, value, read_reg1, ...

The first argument *block_num* is the number of the block instance being called (starting from 0), and is guaranteed to be less than *count* as passed to the *Extension* constructor.

The second argument *value* is the value written to this field.

The remaining *read_reg1* ... *read_regN* argument must match the number of arguments specified in the *read-reg* block of the register file. These will be populated by reading the corresponding block hardware registers before this function is called.

The value returned must be a tuple of integers matching the *write-reg* block of the register file. The returned values will be written to the specified hardware registers after processing this function. This defines the action of writing this field.

3.6.2 Injected Values

Every extension module will have two support values injected into the module when the module is loaded into the server. These are available to help with the implementation of extensions.

class ServerError(*Exception*)

Read and write methods should use this exception to report errors. Exceptions of this type are treated specially and are reported as normal read or write errors.

class ExtensionHelper

This can be used inside an extension module to create extension support for individual fields. Pass a block constructor (that must take one argument, the block index) which implements *set_* and *get_* methods as appropriate, and this helper will implement the approprate Extension support.

Use this inside the extension module thus:

```
class MyBlock:
    def __init__(self, n):
        ...

    def get_field(self, *regs):
        ...
        return value

    def set_field(self, value, *regs):
        ...
        return new_regs

def Extension(count):
  return ExtensionHelper(MyBlock, count)
```

3.7 Starting Panda Server

The Panda socket server is normally automatically started at boot time or when the `zpkg-daemon` script is run. The server is started and stopped by the script `etc/panda-server` installed in `/opt/etc/init.d`.

The server can optionally be started from the command line, in which case the following arguments are supported:

-h This option shows the help text for server, listing all the available command line options.

-p port This specifies the socket port to be used for configuration commands. The default value is 8888.

-d port This specifies the socket port to be used for data capture. The default value is 8889.

-R This can be specified to allow socket reuse via the `SO_REUSEADDR` socket option.

-c config-dir This specifies the directory where the `config`, `registers`, and `description` files will be loaded from. This argument must be specified.

-f persistence-file This specifies where the persistence state will be loaded from on startup and saved during operation. See the `-t` option below for notes on how this file is updated. If this is not specified then the persistence state will not be saved.

-t [poll] [":" [holdoff] [":" backoff]] This option sets three parameters (in seconds) controlling the pacing of writes to the persistence file. The behaviour of the system is as follows: every *poll* seconds the internal state of the server is checked for configuration changes. If a configuration change is checked then there is a pause of a further *holdoff* seconds before the updated state is written. Finally, there is a pause of *backoff* seconds before polling for internal changes resumes.

Default values are: *poll* = 2, *holdoff* = 10, *backoff* = 60. The somewhat complex syntax show above allows all or any of these values to be set with a single `-t` option. For example, `-t:20` specifies *holdoff* = 20, other values unchaged.

The intention of this timed behaviour is to reduce file write impact while still keeping on top of changes. With default settings all parameters will be written to the persistence file within 72 seconds.

-D This option requests that the server is run as a daemon. This is the normal mode of operation when running as a server, but is generally omitted for debug.

-p pid-file If specified the given file is written with the process ID of the server, and will be deleted on exit.

-T This mode is used for config file validation only: the server exits immediately after loading configuration files.

-M MAC-list If specified then the given file is used to initialise up to four MAC address registers. The file consists of any number of comment lines (comment lines start with `#` in the first column) together with four MAC address lines, each of which is either blank (newline `\n` only) or is a six octet MAC address written as 2 digit hex numbers separated by colons.

-X port If specified the server will attempt to connect to an extension server running locally and serving on the specified port.

3.8 Supporting Documentation

3.8.1 Useful Tools

There are a number of useful tools in the `python` directory.

`sim_server` This is run as part of the top level `simserver` script to provide emulation of the Panda hardware. The version of this tool provided with the server is *very* basic, for a more functional emulation use the corresponding tool in the PandaFPGA project.

`tcp_client` [server [port]] This tool connects to the Panda server configuration port and helps with sending and receiving configuration commands.

`save-state` server file This saves the entire configuration state for the given Panda to the given file.

`load-state` server file This writes the given configuration file to the Panda.

3.8.2 Panda Status LEDs

Two LEDs on Panda show a rough indication of the current status of Panda. There are two LEDs, STA (status) and DIA (diagnostic); the status LED is green and is used to indicate normal activity, the diagnostic LED is red and is used to indicate various fault conditions.

The table below shows the possible LED indicators and their meaning:

Mnemonic	DIA	STA	Meaning
-	Off	Off	System not running
BOOTING	Off	Blink	Panda booting
SYSTEM_OK	Off	On	Panda running ok
ATTENTION	Blink	Off	User attention required
NW_ERR	Blink	Blink	Network problem detected
-	Blink	On	(not used, should not occur)
ZPKG_ERR	On	Off	Problem loading installed package
SYSTEM_ERR	On	Blink	System error
-	On	On	(not used, should not occur)

The detailed meaning of these conditions is described below.

BOOTING The system is currently booting. Unless a new image is being configured this should only take a few seconds, but during image installation this can take a few minutes.

SYSTEM_OK Booting has completed and the system is running normally.

ATTENTION User attention is required. Either a fresh installating is prompting for a MAC address, or no system packages have been installed. Connect a serial port in the first case, connect to the administration web page on port 8080 in the second case.

NW_ERR A network error has been detected.

This function is not currently implemented.

ZPKG_ERR An installed package has failed to start. Try power-cycling Panda first, if that fails check the logs and the serial port for relevant diagnostic messages.

SYSTEM_ERR An internal system error has been detected.

This function is not currently implemented.

Index