FTUNSHADES 2

TNT 3.7

User's Manual

Tnt

Table of Contents

About This Document

This is the user manual for the 3.7 version of the TNT shell as it exists at the end of 2017.

It is meant to be a complete description of the capabilities of the system, written having in mind users interested in operating the command line interface and taking advantage of the scripting capabilities of the embedded Tcl shell.

Although we have attempted to be clear and present the information in a way that can be easily understood, the document requires an understanding of what the FTUNSHADES system is and how it does allow emulating the effects of radiation on any piece of hardware whose description can fit in a FPGA.

Some knowledge of the Tcl scripting language is recommended.

In order to better understand the system, we recommend this document is read with a session of the shell actually open in a machine with at least one FTUNSHADES device connected.

What is TNT

TNT is the suite of Test Analysis Tools for the FTUNSHADES system: a collection of executables to be run from a POSIX command line.

The toolchain can be understood to include a number of parts:

- A set of tools to generate files in several data formats that describe in some way or another hardware as emulated in a FPGA
- An extended Tcl shell that allows users to interface with FTUNSHADES devices, and feed them the data generated by the former.
- A set of tools to generate scripts that allow simulation---not emulation---of circuits at an analogic level.

Introduction to tntsh

In this section we'll provide an introduction to the TNT shell.

The TNT shell is a Tcl shell with an integrated library of commands written in C. These libraries provide all features needed to use an FTUNSHADES device to emulate and debug hardware.

In the following pages we will present the shell and its capabilities, step by step, from the simplest to the more complex.

Launching a Shell Session

If you're using TNT from the UFF web interface, you don't need this section and you can jump to the next one. On the other hand, if you want to forgo the graphic interface and launch the shell directly from the server command line, keep reading.

To start the shell, you'll need an user account with the appropriate permissions in a server with a connected FTUNSHADES device. Usually, you'll connect to this server with an SSH client, but you may also directly log in to the machine.

Once connected, just execute `tntsh`; no arguments are needed:

```
user@/path/to/directory$ tntsh
```

By doing this, you are telling the shell session to use the project in the `/path/to/directory` directory. You may also pass the root directory of the desired project as an argument:

```
user@/home/user$ tntsh /path/to/directory
```

The root folder of the design is important because it is the only place where the application will look for files (with some exceptions that we'll get to later) and the place where all result files and logs will be written to. If you're just getting started, you may create a `foo/` folder somewhere and use that to learn. Even when learning, doing without a dedicated folder is not recommended since there are a number of shell commands that will create files and directories wherever you were when you launched the shell, especially when the log levels of the session are set high. Running the environment from a dedicated folder keeps all these files contained.

Help! I'm Lost In a Command Line Interface

Keep calm and breathe. Don't panic. The shell itself can give you a hand. Just type `help`.

```
% help
Help for the TNT shell 3.7. Type one of the following:

  help dev - FTUNSHADES device
  help xc - expansion cards connected to the motherboard
  help obj - on board memory management
  help load - loaders for several file types
  help bit - target FPGA logic map
  help reg - target FPGA I/O operations
  help clk - emulation time control
  help run - automated campaigns
  help dbg - device debugging
```

As you can see, the help command allows you to access the full reference manual for the TNT shell, exactly the same contents as the one at the end of this particular document. It is not as pretty as a postscript document or web page, but it does the work. Every time you call it, it provides you with a “page” of help that may give an index of all commands related to a particular topic:

```
% help dev
All methods to directly interface with an FTUNSHADES device.

  help dev_ls
  help dev_open
  help dev_close
  help dev_manufacturer
  help dev_description
  help dev_serial
  help dev_timeout
  help dev_error
  help dev_reboot
  help dev_ping
```

Or just explain the mechanics of a single function or variable:

```
% help dev_open
Usage: "dev_open"

  Open the first available device of the type specified in the design
  configuration file. Once opened, the device becomes property of this
  particular session.
```

Of course, you may prefer to read the prettified version of the documentation that we have included at the end of this document. It’s up to you.

Hello, World

So let's begin with the most classic of all beginner examples: the hello, world program.

Of course, we could write this using just the Tcl capabilities of the TNT shell, but we are going to do something a little more complex and much more wasteful: we're asking an actual FTU device for the string we want to print.

```
% dev_open
0
% dev_ping "hello, world"
hello, world
```

Typing this in the TNT shell, and assuming there is a device connected to the server that is accessible by the user and not being used by anyone else, will output "hello, world" to the console.

So how does this work? First, we attempt to acquire and open a device.

```
% dev_open
0
```

Since there is very little TNT can do without a device, this line will be the first one you'll type in most scripts and shell sessions. You'll eventually remember it.

The command returns the acquired device serial number on success, hence the `0`.

Then we have:

```
% dev_ping "hello, world"
hello, world
```

This line sends a string to the device and waits for the specified number of seconds until it responds with exactly the same string. It has had many uses during the debugging stages, but you'll probably not see much of it.

Listing, Opening, and Closing Devices

Let's assume you have a opened TNT shell, with an associated project, and want to check if there are available devices so you can perform your magic.

Listing Devices

You may use the `dev_ls` function. After pressing enter, you will be provided with a list of all available devices in the system, their descriptions and will know if they are available for you to use or not.

```
% dev_ls
{{1 ftu2_xc5vfx70tff1136 {AICIA - Univ. Sevilla} yes}{2 ftu2_xc5vfx70tff1136 {AICIA -
Univ. Sevilla} yes}{3 ftu2_xc5vfx70tff1136 {AICIA - Univ. Sevilla} yes}}
```

If you think that’s unmanageable, you may format it a little:

```
% foreach i [dev_ls] {puts $i}
1 ftu2_xc5vfx70tff1136 {AICIA - Univ. Sevilla} yes
2 ftu2_xc5vfx70tff1136 {AICIA - Univ. Sevilla} yes
3 ftu2_xc5vfx70tff1136 {AICIA - Univ. Sevilla} no
```

Here, there are three available devices. All of them have FX70t target FPGAs and two of them are available for use. We don’t know what is keeping the third one occupied. In general it will be just open in a session by another user.

Opening Devices

So assuming we’re in a project that requires an FX70t FPGA, we can attempt to open one of these with `dev_open`. On success, this function will return the description string of the device it opened:

```
% dev_open
1
```

Because of some quirks of the FTDI library, the `dev_ls` function does not work while there is a device opened, but now that it does belong to our session, we can request information from it with the following functions:

```
% dev_serial
1
% dev_description
ftu2_xc5vfx70tff1136
% dev_manufacturer
AICIA - Univ. Sevilla
```

As you can see, these return the same strings as the `dev_ls` command would.

Closing Devices

Finally, when we are done with the device we can release it:

```
% dev_close
```

It is important to note that a device will remain in the hands of whoever opened it as long as the TNT shell session is active and you don’t close it, so leaving a raw TNT session running with an open device will prevent other users from accessing the device. In the UFF, this is solved by timing out the session if no action is performed—either by a running script or user input—for a specific amount of time, but in the terminal you’re responsible of releasing the device for other users.

TNT as a Hardware Debugger

TNT can be used to debug hardware without implementing it first; in fact, the HADES in FTUNSHADES stands for Hardware Debugging System.

There's a bit of initialization and resource loading required to prepare the target FPGA to do so:

```
# Acquire a device
dev_open
# Populate the bit tree
load_ll "registers.ll"
load_ll "more_registers.ll"
load_pin "pins.pin"
# Create some watches
bit_watch regs
bit_watch more_regs
bit_watch pins
bit_watch more_pins
# Load a bitstream
load_config "bitstream.bit"
# Load a vector set to feed the target FPGA
load_vectors "vectors.dat"
# Configure the target FPGA
xc_configure all
# Move to the first vector
clk_rewind
```

Let's see all those, point by point.

Mapping the Target FPGA

In order to debug a circuit, we must first be able to visualize it. That is what the bit tree does.

A bit tree is a hierarchy of named nodes, similar to a file system where instead of files there are elements of the target FPGA that may take a value, such as internal registers or external pins.

For example, the node whose full path is `"/reg/foo/1"` may correspond to the register whose address list is `(BAADFOOD:DEAD, 600DFOOD:FAAB)`; and `"/pin/baz/clock"` may correspond to the seventh pin of the target FPGA.

Loading Bits From a File

Of course, building bit trees from scratch would be complicated, so we load them from an assortment of files instead.

Registers are loaded from logic location files whose extension is .ll and are usually generated alongside the design. The command to do so is `load_ll` and it behaves as follows.

```
% load_ll counter.ll
% bit_ls
reg/
```

On success, this function adds the registers from the file to the bit tree of the current session. More than one logic location file may be loaded, each one adding their registers to the main tree; this is mostly used to load separately the maps to the user and configuration bits of the target FPGA.

We can also specify a directory of the bit tree as the root for all registers loaded from this file:

```
% load_ll counter.ll foo
% bit_ls
foo/
% bit_ls foo
reg/
```

Pins are loaded from pin files whose extension is usually .pin—the same ones used as a basis to generate .ucf files with the `tnt-pin2ucf` tool. The command to do so is `load_pin`.

```
% load_pin counter.pin
% bit_ls
data_out/ enable rst_high
```

Navigating the Bit Tree

Once the bit tree is loaded we can navigate it. We have seen the most basic usage of the `bit_ls` command:

```
% load_ll counter.ll
% load_pin counter.pin
% bit_ls
reg/ data_out/ enable rst_high
```

Now assume we wanted to know what is inside the `reg/` node. For this, we give `bit_ls` the path of the node we want to see.

```
% bit_ls reg
0 1 2 3 4 5 6 7
```

In this case, that's all right, but we could have attempted to read a node with hundreds or thousands of children. Luckily, the `bit_ls` function allows wildcards:

```
% bit_ls reg/*
reg/0 reg/1 reg/2 reg/3 reg/4 reg/5 reg/6 reg/7
```

We can also match character ranges enclosed in square brackets, but for this to work, we have to enclose the pattern in curly braces:

```
% bit_ls {reg/[01]}
reg/0 reg/1
```

Or even:

```
% bit_ls {reg/[!01]}
reg/2 reg/3 reg/4 reg/5 reg/6 reg/7
```

If we didn't add the enclosing curly braces, the shell would output a message as follows:

```
% bit_ls reg/[01]
invalid command name "01"
```

This happens because for Tcl, the square brackets are an indicator that the enclosed string should be interpreted as a command before the string is passed to the FTU function. To override this, we enclose the string in curly braces, indicating the Tcl interpreter that the data should be passed raw to the `bit_ls` function.

For more on pattern matching, see the relevant annex at the end of this manual.

Until now, we have been able to play with a blank slate, but that's not very useful. In this section, we'll learn how to configure the target FPGAs with a bitstream to emulate any circuit.

Loading Bitstreams

The simplest way to do this is to call the `load_config` function:

```
% load_config foo.bit
1
```

If you only need one bitstream to configure the target FPGA, this will be enough and you can skip two sections forward to where we explain how to do exactly that.

If you want to know and do more, we'll start by taking a look at what `load_config` actually does:

```
proc load_config {path} {
    global FPGA_CONFIG
    if {[catch {obj_new} ret] == 0} {
        set name $ret
        if {[catch {load_bit $path $name} ret] == 0} {
            if {[catch {obj_bind $FPGA_CONFIG $name} ret] == 0} {
                return $name
            }
        }
        obj_destroy $name
    }
    error "Can't load bitstream: $ret"
}
```

So step by step, what we must do to load a bitstream on our session is:

- Create a new object in the device with `obj_new`
- Upload an actual bitstream from a file to the object with `load_bit`
- Bind it to the `FPGA_CONFIG` target with `obj_bind`

Also, if this fails, it takes steps to clean up after itself:

- If the data can not actually be loaded, destroy it with `obj_destroy` so it doesn't take up space in the FTUNSHADES device.

Having seen the bullet points, let's get into details:

The `name` returned by the `obj_new` function is a numeric resource identifier that you can use to refer to the object in all subsequent calls.

If the creation fails, instead of returning a name, `obj_new` will raise an error with an appropriate message. Most of the time it will be:

- **No device currently open**: if you forgot to execute `dev_open` or there just isn't an available device. In fact, almost every function of the TNT shell will say this.
- **Out of memory**: if you created too many or too large objects.

Handling Multiple Bitstreams

If you're going to use more than one bitstream, it's recommended you store the names in variables with clear names, as shown below:

```
% set myBS [obj_new]
1
% set myOtherBS [obj_new]
2
```

Now about binding: for the FTUNSHADES device to use an object, we must bind it to a target. The target defines the role of that object; e.g.: the `FPGA_CONFIG` target holds the data that will be used to configure the target FPGAs.

Depending on the device version, binding may be as simple as copying a pointer and will not have further consequences; or it may imply caching the object so future operations may be faster.

Binding is manipulated with the `obj_bind` function. If we give it the name of a valid object, it will replace the previous binding (either none or another object) with it.

```
% obj_bind $FPGA_CONFIG $myOtherBS
```

On the other hand, if 0 is passed, it will unbind whatever was previously bound to the target.

```
% obj_bind $FPGA_CONFIG 0
```

If we wanted to delete an object, we can use the `obj_destroy` command.

```
% obj_destroy $myBS
```

After this, both the object and its associated data will be removed from the FTUNSHADES device.

Configuring the Target FPGA

That's all about managing bitstream objects. After this, the actual configuring of the target FPGAs is pretty much anticlimactic, requiring just a call to `xc_configure`:

```
% xc_configure all
```

The only argument required is the identifier of the expansion whose target FPGA is to be configured or, as in this case, `"all"` to configure all target FPGAs, since it uses the object bound to the `FPGA_CONFIG` target.

Binding and configuring multiple bitstreams in a row is a perfectly valid process that may be used to take advantage or partial reconfiguration. Since once the data is uploaded, all objects reside in the FTUNSHADES device, the cost of this operation is pretty light.

```
foreach bs { $myBS $myOtherBS } {
    obj_bind $FPGA_CONFIG $bs
    xc_configure all
}
```

Finally, the target FPGAs can be cleared and their configuration removed with:

```
% xc_deconfigure all
```

That is pretty much useful only for partial reconfiguration, since a configuration that is not partial will overwrite the previous one.

Inputs and Outputs

It is, of course, entirely possible to emulate a circuit running while all its inputs are set to zero, just not very useful. So we need some data that can be fed to the target FPGAs during emulation.

We call the bit array that contains the concatenated inputs for the target FPGAs each cycle an input vector. The corresponding bit arrays with the concatenated outputs that come out of each one of the target FPGAs each cycle are output vectors.

A full sequence of input vectors intended for a full emulation run we call a vector list. Let's see how we can handle them.

The management of vector list objects is exactly the same as that of bitstream objects. For an in-depth explanation, read again that particular section and just substitute `FPGA_CONFIG` by `FPGA_VECTORS` and `load_bit` by either `load_dat` (if loading from an old vector file) or `load_wave` (if loading from a new vector file). Here, we'll just have a quick look at what can be done before actually playing with the clock.

Loading Vector Sets

This is the simplest way to upload vectors to the device:

```
% load_vectors foo.dat
3
```

This is the code for the `load_vectors` function. If it looks similar, that means there's a good chance you have been reading the sections in order and have at least a halfway decent memory because it's, line by line, the same as `load_config`:

```
proc load_vectors {path} {
    global FPGA_VECTORS
    if {[catch {obj_new} name] == 0} {
        if {[catch {load_dat $path $name} ret] == 0} {
            if {[catch {obj_bind $FPGA_VECTORS $name} ret] == 0} {
                return $name
            }
        }
        obj_destroy $name;
    }
    error "Can't load vector list: $ret";
}
```

The bullet points:

- Use `obj_new` to create a new object and get a name.
- Use `load_dat` to set the data for the bound object.
- Use `obj_bind` to bind the object to the `$FPGA_VECTORS` target.
- Use `obj_destroy` to destroy the object.

The equivalent to `xc_configure` for vectors is the `clk_rewind` command. Despite what its name seems to imply, it does not just rewind the vector pointer, but readies the entire system for emulation. You can see that by checking the clock value with `clk_read` before and after calling it:

```
% clk_read
-1
% clk_rewind
% clk_read
0
```

When the value returned by `clk_read` is less than zero---or higher than the number or loaded input vectors---, no clock operations will work.

Keeping an Eye With Watches

Watches are a way to keep track of the evolution of the values of arbitrarily chosen bits in the bit tree as the emulation advances. By creating a watch on a specific node of the register or pin tree you're telling the TNT session you want the state of all bits under that particular node to be recorded.

Creating Watches

Watches may be created with the `bit_watch` command:

```
% bit_ls
reg/ data_out/ enable rst_high
% bit_watch reg
```

Creating a watch after executing `clk_rewind` may cause a bug that make a timed out error. To avoid this, it is important to create them before configuring the target FPGA.

Listing Watches

At any time, you may request a list of all existing watches with the `bit_watches` command:

```
% bit_watches
reg
```

Reading Bit Logs

When a node in a bit tree has a watch on it, the value of every register inside it will be recorded as the emulation proceeds. This means that creating a watch for `reg/` allows you to request the log of every children of `reg/` (e.g.: `reg/0`).

The logs can be dumped at any moment with the `bit_log` command:

```
% clk_step 10
% bit_log reg golden 0 10
15 00 00 00 00 01 02 03 04 05 06
% bit_log reg faulty 0 10
15 00 00 00 00 01 02 03 04 05 06
```

We start by executing `clk_step` to perform some emulation cycles, because there wouldn't be a log to dump otherwise. We'll see more about the clock in the next section.

The arguments for `bit_log` are as follows:

- Name of a node in the bit tree; it may be a directory, a register, or a pin.
- Name of the card whose values are requested. This may be `golden` (or `g`) and `faulty` (or `f`).
- Optionally, the first cycle of the requested log slice; if no cycle is provided, the last one is used.
- Optionally, the last cycle of the requested log slice; if no cycle is provided, the same as the first is used.

To list the content of all watches at once the `dump_logs` macro is provided:

```
proc dump_logs {args} {
    foreach x [bit_watches] {
        foreach c {g f} {
            puts "$x.$c: [eval bit_log $x $c $args]"
        }
    }
}
```

Its input are the first and last cycles requested, and its output a log of all watches for all target FPGAs:

```
% dump_logs 0 10
/enable.g: 0 0 0 0 1 1 1 1 1 1 1
/enable.f: 0 0 0 0 1 1 1 1 1 1 1
/data_out.g: 00 00 00 00 00 01 02 03 04 05 06
/data_out.f: 00 00 00 00 00 01 02 03 04 05 06
/reg.g: 15 00 00 00 00 01 02 03 04 05 06
/reg.f: 15 00 00 00 00 01 02 03 04 05 06
/rst_high.g: 0 1 1 1 0 0 0 0 0 0 0
/rst_high.f: 0 1 1 1 0 0 0 0 0 0 0
```

Saving logs

The TNT shell provides also the option to save a collection of logs to a value change dump (.vcd) file with the `bit_dump` command:

```
% bit_dump reg all my_dump.vcd
```

The arguments taken by this command are as follows:

- A set of patterns to identify all bits that will be dumped. Each pattern allows the use of wildcard characters "`*'', ``?`", and bracket-enclosed ranges. Multiple patterns may be specified by enclosing a space-separated list in curly braces.
- Which logs related to the specified bits will be saved. Allowed values are `golden` or `g` for the golden target FPGA, `faulty` or `f` for the faulty target FPGA, and `all` or `a` for both.
- The path to the file where data will actually be saved.

In any case, the paths to the bits in the golden FPGA will be saved to the `golden/` directory and the paths to the bits in the faulty FPGA will be saved to the `faulty/` directory.

Deleting Watches

To remove existing watches, use the `bit_unwatch` com:

```
% bit_unwatch reg
% bit_watches
```

Time and the Clock

To actually emulate a circuit, you will use the `clk_` commands. Since you still have done nothing, the current cycle should be `0`. You can see this with the `clk_read` function:

```
% clk_read
0
```

Stepping Cycles

Let's see how to change this. There are four functions that advance the clock: `clk_step`, `clk_skip`, `clk_step_until`, and `clk_skip_until`. Each one changes the logs of those vectors in different ways.

The `clk_step` function emulates the requested number of cycles and records everything that is happening in every one of them. It is probably the most I/O intensive method of the entire shell, because for every step forward, it reads the state of every bit in the target FPGAs input, both sets of FPGA outputs and both values associated with every bit of every watch that has been declared.

Assuming we have watches on the `reg/` register and `data_out/` pins, the results of executing a number of steps may be as follows:

```
% clk_step 10
% clk_read
10
% bit_log data_out golden 0 [clk_read]
00 00 00 00 00 01 02 03 04 05 06
```

On the other hand `clk_step_until` does not take arguments. Instead it steps until either a discrepancy is found in the output vectors or the end of input vectors is reached. Apart from this, it behaves just the same as `clk_step`:

```
% clk_step_until
% clk_read
285
% bit_log data_out golden 0 [clk_read]
00 00 00 00 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 14 14 14 14
14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30
31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A
6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87
88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4
A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1
C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE
DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB
FC FD FE FF 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14
```

Unless the user has injected some damage in one of the target FPGAs, no discrepancies will be found and execution will reach the end of vectors.

Skipping Cycles

The previous functions are ideal when we want to know what is actually happening inside the FPGA during execution, but very bad to just reach a point of interest, especially if the point is millions of cycles away from the first one. Luckily, we have alternatives:

```
% clk_skip 10
% clk_read
10
% bit_log data_out golden 0 [clk_read]
00 x x x x x x x x 06
```

And:

```
% clk_skip_until
% clk_read
285
bit_log data_out golden 0 [clk_read]
14 x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x x x x x x x x x x x x 14
```

These emulate the requested number of cycles, but only does the bit-reading after the last step, so the execution is much faster at the cost of not knowing exactly what happened during the missing cycles. Notice those “x”? They mean the log has no information about what the value of the pins was during those particular cycles.

Rewinding the Emulation Clock

After advancing the clock you can not step back because entropy. If you want to go to a previous cycle, you have to reset the clock:

```
% clk_rewind
```

Note that resetting the clock does not reconfigure the target FPGAs, so if there are accumulated errors somewhere inside you may get strange outputs for a few (or more) cycles until the input catches up. To fix this, execute `xc_configure all`.

Manual Injection

While debugging, injections are performed through simple I/O operations in the bit tree:

```
% bit_read reg/0 golden
1
% bit_read reg/0 faulty
1
```

As always, `golden` (or just `g`) makes reference to the control target FPGA, where faults are never injected, while `faulty` (or just `f`) is the experiment target FPGA. In this case the output is just the value of the only bit associated with the only register we have requested, but we can ask for more:

```
% bit_read reg golden
5A
% bit_read reg faulty
5A
```

The output comes in the form of a bit array where the nth bit is the value associated with the nth register matched by the pattern.

You may also overwrite the values:

```
% bit_write reg faulty FF
% bit_read reg golden
5A
% bit_read reg faulty
FF
```

Note that directly writing to bits is not supported by the FPGA external pins, whose values are always determined by the input vectors currently bound to the `$FPGA_VECTORS` target (in the case if the input pins) or the values read from the output pins after every emulation step (in the case of the output pins).

Block Ram Injection

One of the interests of FTUNSHADES2 is the ability to emulate the effects of radiation (SEU) on memory blocks (BRAM) of FPGAs. In this section, we are going to see what we can do about injection on BRAMs and how to do it.

Watches in BRAM

Just as it was done in the registers, watches can be created in order to track the status of arbitrarily chosen BRAM bits in the bit tree using the `bit_watch` command.

```
% bit_ls
address/ ram/ registro/
% bit_watch ram/RAMB36_X2Y25/B:BIT0
```

Creating a watch in a BRAM bit may cause delays of seconds in readings on debug mode.

Creating a watch after executing `clk_rewind` may also cause a bug that make a timed out error. To avoid this, it is important to create them before configuring the target FPGA.

Read BRAM

Block memory contents can be examined by reading memory frames whose addresses appear on logic allocation file (.ll). Although there is not an specific command implemented to read memory frames without causing memory corruption yet, we can use this `dbg_ram_read` Tcl macro to do it properly.

```
proc dbg_ram_read {frm} {
        dbg_frame_read $frm
        dbg_ram_set $frm 465 1
}
```

This macro receives the address of the frame that is wanted to read and prints it to the standard output. A single frame is 1312 hex-encoded bit vector, and is shown here in groups of 32 bits (1 word) ordered from MSB to LSB. The value of these bits is inverse to the value of bits in memory.

Function `dbg_ram_get` doesn't exist because `dbg_bit_get` can be used in BRAM as it is done in registers. This function prints the bit that is wanted to read to the standard output.

```
% dbg_bit_get frame offset
```

Write BRAM

You can modify BRAM contents by writing in the corresponding memory frames that appear in the .ll file. To do this, you can use `dbg_ram_flip` macro, which flips a RAM bit.

```
% dbg_ram_flip frame offset
```

Or `dbg_ram_set` macro, which sets a RAM bit.

```
% dbg_ram_set frame offset
```

TNT as a Campaign Runner

A campaign is an automatic test of an emulated circuit during which a number of errors will be injected in the SEU target FPGA to an user-defined set of registers in an user-defined set of cycles; then its output will be compared to that of the other, where no errors were introduced. From this, the system will generate a number of log files that can be analyzed after the campaign is complete.

Let's see how to perform a simple campaign:

```
# Acquire a device
dev_open
# Populate the register tree
load_ll "registers.ll"
load_ll "more_registers.ll"
# Populate the pin tree
load_pin "pins.pin"
# Load a bitstream to program the target FPGA
load_config "bitstream.bit"
# Load a vector set to feed the target FPGA inputs
load_vectors "vectors.dat"
# Actually execute the campaign
run_random "/" "*" 100
```

Simple, isn't it?

Once the device is open and all required resources loaded, a campaign with default options can be launched using any one of a set of injectors provided by the TNT shell. Injectors are functions that generate the list of `<register,cycle>` pairs where faults will be injected. They also define:

- The lists of target registers and cycles that are candidates to participate in an injection.
- The number of runs that will be executed during that particular campaign.
- The maximum number of injections that will be performed each run.

Anatomy of a Campaign

A campaign is a sequence of individual runs. A run is the process of emulating the design loaded in the target FPGAs by feeding them the entirety of the input vectors provided by the user, and occasionally, flipping the value of a specific bit at a certain time.

At specific moments of the run, the FTUNSHADES device will retrieve the target FPGA outputs and compare them with a "golden" set, obtained by executing a run with no injections. If a mismatch is found between both, an error will be reported.

Checking for Additional Damage

Unless otherwise specified, the device will only start checking for errors after the first injection each run, and only until a maximum number of cycles with discrepancies have happened. You may configure the device to do otherwise with the `run_set` command:

```
% run_set check_residual_damage 1
```

With this, you're telling the device to check for damage from the first cycle of every run. Since you haven't injected, any error found will be a residue of a previous run, hence the name.

Also, you can keep checking for damage after the first error; the following line will keep comparing the outputs until five discrepancies have been found.

```
% run_set damage_per_run 5
```

Note that checking for damage is the costliest operation of the campaign, so changing these values will dramatically increment the time needed to execute it.

Discarding unwanted cycles

By default, campaigns do not only stop checking for errors after `damage_per_run`; they directly stop emulating cycles for that particular run and start the next one. This makes sense if you do not need to know how one run changes the next, but there are situations in which you want to keep on emulating despite not checking for errors:

```
% run_set drop_on_damage 0
```

Note that the value of this setting is by default 1.

Return the Location of Faulty Ouputs

By default, the FTUNSHADES system returns only when damage was found. If you want to return the location, you may use the following:

```
% run_set show_output_xor 1
```

By doing this, the damages.csv file will include a bitmask—in hexadecimal format—in which every bit corresponds to one of the outputs defined in the vector list file. Bits set to 1 are bits where the SEU and GOLD outputs vary.

Readback

It is possible to configure the campaign to perform a dump of the contents of the target FPGA at the end of certain runs. The dumps can then be used to perform an in-depth analysis of how certain damages affect the design.

The `readback_at_error` flag will dump the TFPGA contents to a file at the end of every run in which damage is found. Its value is by default `0`:

```
% run_set readback_at_error 1
```

The `readback_at_time` flag will dump the TFPGA contents to a file at the end of every readback_at_time runs.

```
% run_set readback_at_time 10
```

In any case, the dumps are stored in separate files whose name depends on the run; e.g.: for the 10th run, the file will be `readback_10.bit`.

Also, if any of these flags is set, a `readback_golden.bit` file is produced with a dump of the contents of the TFPGA after the golden run. This is supposed to be the correct state of the design after a run, and can be compared to the others to determine where problems appear.

Dynamically Fix Damages

By default, the FTUNSHADES device keeps executing runs disregarding damage. This means that after a number of injections, the accumulated errors may make the results useless, but there are a number of flags that allow you to sidestep this.

The `unflip_after_run` flag is the fastest and dirtiest one: it just unflips every bit that was flipped at the end of the run. But it doesn't fix damage that may have propagated from those flips, so its usefulness is very limited in many cases.

```
% run_set unflip_after_run 1
```

The `reconfigure_at_error` flag is the slowest and surest of all: after every run in which damage was found, it will fully reconfigure the SEU target FPGA with the bitstream currently bound to the `FPGA_CONFIG` target.

```
% run_set reconfigure_at_error 1
```

The `reconfigure_at_time` flag is kind of a middle ground: It will fully reconfigure the SEU target FPGA with the bitstream currently bound to the `FPGA_CONFIG` target, but only every reconfigure_at_time runs

```
% run_set reconfigure_at_time 10
```

Batch Size

To run campaigns, the FTUNSHADES 2 system operates in batches: it sends a number of runs to be processed to the device, and then reads all the results at once. The number of runs per batch can be specified as follows:

```
% run_set batch_size 1000000
```

A larger batch size will increment the campaign speed, and consume more memory on the device. Also, larger campaign sizes reduce the resolution of the campaign progress bar.

Flow of a Run

All put together, the campaign algorithm is as follows (not actual code, just Python-ish pseudocode):

```
damage_this_run = 0
injected_this_run = False
cycle = 0

for i in input:
    # step a cycle
    if (injected_this_run or check_residual_damage) and
        damage_this_run < damage_per_run:
        # read target FPGA output
        if output != golden_output:
            # write cycle to the response
            if show_output_xor:
                # write the XOR'd outputs to the response
            damage_this_run += 1
    if is_injection_cycle(cycle):
        # flip the chosen bit
        injected_this_run = True
    cycle += 1
if (readback_at_error and damage_this_run > 0) or
    (readback_at_time and run % readback_at_time == 0):
    # write full target FPGA contents to the response
if unflip_after_run:
    # unflip all flipped bits
if (reconfigure_at_error and damage_this_run > 0) or
    (reconfigure_at_time and run % reconfigure_at_time == 0):
    # reconfigure target FPGA
```

The Random Injector

The random injector generates a fixed number of runs, with a fixed number of injections per run, and for every injection, choses randomly from user provided register and cycle lists.

```
% run_random registers cycles runs ?injectionsPerRun? ?seed?
```

It takes the following arguments:

- A list of registers, defined by a pattern like the ones recognized by `bit_ls`.

- A list of cycles, defined as a comma-separated list of cycle ranges.
- The number of runs that will be executed during this particular campaign.
- Optionally, the number of injections that will be performed each run. If this number is not given, it will automatically be set to 1.
- Optionally, a value that can be used as seed to the pseudo-random number generator.

Two random campaigns with the same arguments, including the seed, are guaranteed to generate the same list of injections.

We have already seen several examples of patterns that match register lists, so we won't go over that again. Instead, we will talk about the cycle lists. As previously mentioned, the candidates are given in the form of a comma-separated list of ranges. Ranges may be a single number or a consecutive list of them, expressed as the first cycle, a dash, and the last cycle. Finally, if the entire string consists in a lone asterisk, all cycles are selected.

The following are valid ranges:

```
"50"
"1,2,3,5,7,9,11,13"
"0-100"
"1,2,3-7,8"
"*"
```

The Exhaustive Injector

The exhaustive (sometimes called combinatorial) injector takes a different approach. It takes only a couple of arguments.

```
% run_exhaustive registers cycles
```

Notice the difference? The full injector does not need a number of runs, because it will do every possible run with the given `registers` and `cycles`, hence the exhaustive name. If you give ten registers and ten cycles, the first ten runs it will inject in every register in sequence in the first cycle, from the eleventh to the twentieth, it will inject in every possible register in sequence in the second cycle, etc.

It's trivial to see that the number of runs generated by this injector is equal to the sizes of the register and cycle lists, multiplied. It's important to be careful, because this number can very easily become very big.

The File Injector

The file injector is a possible solution for when none of the default injectors suffice: an injector that reads the entire campaign from a file provided by the user.

```
% run_file path
```

The campaign description file is pretty bare bones and follows the following schema:

- A register is given by its full path.
- A cycle is given by an integer number.
- An injection is written `cycle:register`, with no spaces.
- A run is a comma-separated (`,`) list of injections, with no spaces, terminated with a semicolon (`;`)

The result is as follows:

```
10:/reg/0;
10:/reg/1;
10:/reg/2;
10:/reg/3;
10:/reg/4;
```

The number of runs is the number of lines in the file, and the maximum number of injections per run the maximum number of register:cycle pairs in a line.

There is a limitation on the way the file must be written. In the same run, the injections must be sorted by cycle. You may imagine the injector reading the file and already in cycle 100 suddenly finding a request to inject in 90 during the same run. Since the system can't go backwards because entropy, you wouldn't like to be asked that, so we'll be polite and do the same to the injector.

All registers mentioned in the file must be loaded in the bit tree (via `load_ll`) before the campaign is launched.

The Custom Injector

The custom injector is the procedural alternative to the file injector. Instead of reading the campaign from a file, it generates all injections from a Tcl function.

For example; let's say we wanted an injector that injected always in the first register, and increments the cycle by one each run. This can, of course, be made with a brief injector, but as far as examples go, it's a nice one:

```
% proc myInjector { regs cycles run injection } { return { 0 run } }
% run_custom foo/reg_1<8> 1-100 100 1 myInjector
```

This is other of those injectors requiring many arguments. And this time they don't even have default values.

- A list of registers, defined by a pattern like the ones recognized by bit_ls.
- A list of cycles, defined as a comma-separated list of cycle ranges.

- The number of runs that will be executed during this particular campaign.
- The maximum number of injections that will be performed each run.
- And the name of a Tcl function that can be called to generate injections.

The command function takes as inputs four integer values that are:

- The size of the register list that resulted from the registers pattern.
- The size of the cycle list that resulted from the cycles pattern.
- The current run number.
- The current injection number.

And it must return a pair of indices, of which the first one refers to the target register list and the second one to the target cycle list.

Custom injectors may be used for a wide variety of strategies.

Define probability of injections

This injector behaves the same as the random one, but the probability of injecting every run is an arbitrary percentage `P`; it takes advantage of the fact that returning an empty list from the injection generator allows to inject less times than the maximum number of injections passed to `run_custom`:

```
proc injector { regs cycles run injection } {
        if { [expr {int(rand()*100)}] < P } {
                set reg [expr {int(rand()*$regs)}];
                set cycle [expr {int(rand()*$cycles)}];
                return "$reg $cycle"
        } else {
                return {};
        }
}
```

The Dummy Injector

The dummy injector does not actually inject. It is used to execute "clean" campaigns, usually so users can provide an alternative hardware-based method of generating SEUs.

To use the dummy injector, you must execute the `run_clean` command. The only argument it requires is the number of runs.

```
% run_clean 100
```

Campaign Output

The output of all campaigns is given in the same format, regardless the choice of injector, and consists on a set of text files that are generated in the same directory, always relative to the project path, whose name is the formatted time stamp of the moment the campaign started:

```
results/YY-MM-DD-hh-mm-ss
```

The files are as follows:

run.tcl

A Tcl script that replicates the campaign if executed. It doubles as a log file for what the campaign actually did.

```
## FTUNSHADES2 campaign script
## Automatically generated by TNT shell 3.6
## Mon, 16 Nov 2015 16:25:59 +0100

run_set batch_size 100000
run_set check_residual_damage 0
run_set damage_per_run 1
run_set drop_on_damage 1
run_set show_output_xor 0
run_set unflip_after_run 0
run_set reconfigure_at_error 0
run_set reconfigure_at_time 0
run_set readback_at_error 0
run_set readback_at_time 0
run_random {/} {*} {10} {1} {0}
```

stats.txt

Some statistics taken while running the actual campaign.

```
## Technical log for...
## Automatically generated by TNT 3.6

Elapsed time ............................ 00:00:00.201489
Upload .................................. 00:00:00.100598
Batch ................................... 00:00:00.100620
Download ................................ 00:00:00.000245
Idle .................................... 00:00:00.000015
Batches processed ....................... 1
Runs executed ........................... 10
Bytes sent .............................. 160
Bytes recv .............................. 80
Runs executed/second .................... 99.3838
Bytes sent/second ....................... 1590.49
Bytes recv/second ....................... 326531
```

reg_names.txt

A list of all registers where faults were injected during the campaign. The purpose of this file is to assign a numeric index to each one of the registers, that is used in the injections.csv file: the first path corresponds to a `0` in, the second to a `1`, etc.

```
/reg/0
/reg/1
/reg/2
/reg/3
/reg/4
/reg/5
/reg/6
/reg/7
```

pin_names.txt

A list of all pins where errors were searched for during the campaign. The purpose of this file is to assign a numeric index to each one of the pins, that is used in the damages.csv file.

```
/reg/0
/reg/1
/reg/2
/reg/3
/reg/4
/reg/5
/reg/6
/reg/7
```

injections.csv

A description of all runs that were executed during the campaign:

Every line corresponds to a particular run and contains a list of pairs `cycle:register`, separated by commas (`,`) and terminated by a semicolon (`;`). All injections in a line are guaranteed to be sorted by cycle.

```
164:7;
227:4;
18:1;
164:7;
65:5;
187:2;
131:0;
141:3;
187:1;
189:6;
```

damages.csv

The results of the full campaign. The contents of this file depend on what information the FTUNSHADES device was configured to collect.

Every line corresponds to a particular run and contains the cycles where damage was detected, separated by commas (`,`) and terminated by a semicolon (`;`).

```
165;
229;
20;
166;
67;
189;
133;
143;
189;
191;
```

If the system was configured to request a XOR of the outputs on error, it will be given for every run where damage was found, after the cycle.

```
165:80;
229:10;
20:01;
166:80;
67:20;
189:04;
133:01;
143:08;
189:02;
191:40;
```

Campaign Post-processing

Additional data may be extracted from the campaign output by using the `tnt-stats` tool. This is independent from `tntsh` and must be executed from the system shell:

```
$ tnt-stats /path/to/my/design/results/YY-MM-DD-hh-mm-ss
```

Executing this will generate the following additional files:

morestats.txt

Contains additional statistics about the campaign extracted from the injection.csv and damages.csv files:

```
Post-process analysis
autmatically generated by tnt-stats

Cycles between injection and damage
Min.: 1
Max.: 2
Mean: 1
```

reg_tree.csv

Contains statistics for all the nodes of the register tree that were involved in the campaign, not just the leaves that correspond to actual registers. This is useful to understand how sections of the design were affected by the damage.

Every line contains the name of a node in the register tree, followed by:

- The number of children it has.
- The number of times (runs) damage was injected in any register under that node
- The number of times (runs) a discrepancy was found in the outputs after damage was injected in that particular node.

```
/,8,10;
/reg/,8,10,7;
/reg/1,1,2,1;
/reg/0,1,1,1;
/reg/3,1,1,1;
/reg/2,1,1,1;
/reg/5,1,1,0;
/reg/4,1,1,1;
/reg/7,1,2,2;
/reg/6,1,1,1;
```

pin_tree.csv

This file will only be generated if the `show_output_xor` variable was set to `1`.

Similar to reg_tree.csv, contains statistics for all the nodes of the bit tree that contain at least one output pin.

Every line contains the name of a node in the bit tree, followed by the number of children it has and the number of times errors were found in any pin under that node.

```
/,8,10;
/reg/,8,10;
/reg/1,1,2;
/reg/0,1,1;
/reg/3,1,1;
/reg/2,1,1;
/reg/5,1,1;
/reg/4,1,1;
/reg/7,1,2;
/reg/6,1,1;
```

faults.csv

Contains an user-friendly version of all data from injections.csv and damages.csv.

Each line corresponds to a run, and contains first the cycles and registers where faults were injected, then the cycles where discrepancies were found in the outputs.

```
{164:reg/7},{165};
{227:reg/4},{229};
{18:reg/1},{20};
{164:reg/7},{166};
{65:reg/5},{67};
{187:reg/2},{189};
{131:reg/0},{133};
{141:reg/3},{143};
{187:reg/1},{189};
{189:reg/6},{191};
```

If the `show_output_xor` flag was set, a bitmask with the actual discrepancies will be printed alongside the cycles where they were found. See the description of damages.csv for details.

```
{164:reg/7},{165:80};
{227:reg/4},{229:10};
{18:reg/1},{20:01};
{164:reg/7},{166:80};
{65:reg/5},{67:20};
{187:reg/2},{189:04};
{131:reg/0},{133:01};
{141:reg/3},{143:08};
{187:reg/1},{189:02};
{189:reg/6},{191:40};
```

Tools to generate files

In this section, we'll explain the tools that generate specific files required by the TNT shell, or can be used to manage its output.

Generate User Constraints From Pin Info

During the generation of a design, an ucf file is usually required to provide a mapping of the pins on the TFPGA; for this, the TNT toolchain provides a tool called `tnt-pin2ucf` that can generate such file from the same file tntsh uses to load the pin names.

A pin file has the following format:

```
--control
clk
--input
foo 1
bar 1 7
--bidir
--output
```

Only the `--control` section must have one---and only one---element, that is the clock signal for the entire circuit. All other sections may be empty, but include usually lines of two types:

- Single pins, on the form `name id`, identify only one pin;
- Multiple pins, on the form `name first last`, identify a range of pins.

To generate the ucf file, invoke `tnt-pin2ucf` as follows:

```
$ tnt-pin2ucf <board> <pin file>
```

Where `<board>` is the name of the TFPGA model.

This will dump the ucf code to stdout. A file may be specified with the `-o` or `--output` option:

```
$ tnt-pin2ucf <board> <pin file> -o <ucf file>
```

Generate I/O Data from Value Change Dump

When emulating circuits, in addition to the design of the circuit itself, we require the data that is passed as input and, optionally, the output that corresponds with that input.

Tnt provides the `tnt-vcd2dat` to create vector (dat) files from a value change dump (vcd) file generated by simulating the design, and the same description of the pins that is used when invoking `tnt-pin2ucf`.

The invocation is as follows:

```
$ tnt-vcd2dat <board> <vcd file> <pin file>
```

Where `<board>` is the name of the TFPGA model.

This will dump the ucf code to stdout. A file may be specified with the `-o` or `--output` option:

```
$ tnt-vcd2dat <board> <vcd file> <pin file> -o <dat file>
```

Generate Additional Logic Location Info

To observe the results of injected faults in an emulated circuit, we need a map of where in the TFPGA are the elements of the design mapped: for user bits, this information is automatically generated, and tntsh knows how to load it, but there are many bits in the design---specifically, those that do not correspond to actual elements but determine how those elements are used---, for which no map is usually provided.

The tnt toolchain provides an application named `tnt-ebd2ll` that solves this problem: basically it is feed an ebd file, that is actually provided when the design is generated, and provides a logic location file where the configuration bits of the design are named in a hierarchy.

Names for the nodes in the resulting hierarchy are not user friendly, and in fact do not reflect what element the bits are controlling, but it allows users to inject and observe bits that are not naturally exposed.

To invoke the application:

```
$ tnt-ebd2ll <ebd file>
```

This will dump the ll code to stdout. A file may be specified with the `-o` or `--output` option:

```
$ tnt-ebd2ll <ebd file> -o <ll file>
```

Compare Binary Files

When readback of the TFPGA memory is enabled during campaign, a number of files are produced, containing binary dumps. Despite not being strictly necessary---these kind of utilities already exist around---, tnt provides a couple of tools to compare and format these dumps.

A binary file may be formatted with the `tnt-dump` tool:

```
$ tnt-dump <binary file>
```

This will dump the file to stdout; by default it assumes the following:

- The data is formatted in hexadecimal numbers; if you want instead a string of ones and zeroes, use the `-b` or `--binary` option.
- Every line includes 16 columns of hexadecimal numbers; you may change this with the `-c` or `--columns` option.
- The bytes are assumed to be sorted in big endian; to assume little endian, use the `-l` or `--little-endian` option.
- The bits are assumed to be sorted with the less significant one first; to assume the opposite use the `-m` or `--msb-first` option.

Finally, there are three flags that change the operation mode of the tool:

- Use the `-r` or `--reverse` option to generate a binary stream from formatted data.
- Use the `-1` or `--only-ones` option to print the indices of all bits set to 1 in a binary file.
- Use the `-0` or `--only-zeroes` option to print the indices of all bits set to 0 in a binary file.

We also provide a `tnt-mask` tool that performs boolean operations, bit by bit, in binary files, and is useful for comparison purposes, specially when combined with `tnt-mask`:

```
$ tnt-mask --and <file1> <file2>
$ tnt-mask --or <file1> <file2>
$ tnt-mask --xor <file1> <file2>
$ tnt-mask --nand <file1> <file2>
```

The raw bitstream will be written to stdout; you can redirect it into a file or into the `tnt-dump` tool to have it formatted in some way.

If `file1` is not provided, the data will be read from stdin, allowing to chain together several calls to the tool.

For example: to see the indices of the bytes where two bitstreams differ:

```
$ tnt-mask --xor <file1> <file2> | tnt-dump -1
```

Another example: to see if one of a collection of bits defined in the `mask` file differ between two files:

```
$ tnt-mask --xor <file1> <file2> | tnt-mask --and <mask> | tnt-dump -1
```

Analog Circuits

Starting with version 3.7, tnt provides a set of tools to perform analysis on analog circuits: not a hardware-accelerated emulation, but a simple simulation that depends on external tools; specifically it depends on the Ocean suite, and the input and output files for the toolchain are determined by what is needed by it.

Instrumentation

Instrumentation is the process that takes the design of a circuit and adds elements (instruments) that keep track of the behavior of other elements.

Input circuits for this, come in SPECTRE format, usually with the scs extension.

To add instruments to measure a circuit, we use the `tnt-instrument` tool; in its simpler form, it takes two arguments:

- The name of the technology where the circuit will be emulated
- The path to the scs file containing the circuit.

So a valid way to invoke the command would be:

```
tnt-instrument ihp25 foo.scs
```

This will dump the instrumented netlist to stdout and generate two additional files:

- The node list (in this case foo.nodes) containing all sources where faults can be injected during simulation
- The source list (in this case foo.sources) containing all nodes that can be watched during simulation.

The technology determines which elements of the circuit will be instrumented. Additional technologies can be added to `tnt-instrument` by creating a stf file, that contains a list of the elements that must be instrumented.

By default, Tnt comes bundled with the following files:

- tsmc65.stf
- umc180.stf
- ihp013.stf
- st130.stf
- on05.stf
- ihp25.stf

The actual list of technologies installed at any moment can be printed by passing the `-t` or `--tech` argument to `tnt-instrument`:

```
$ tnt-instrument -t
   tsmc65
   umc180
   ihp013
   st130
   on05
   ihp25
```

By default, `tnt-instrument` writes the instrumented circuit to the standard output. This can be changed by plain redirection or using the `-o` or `--output` option:

```
$ tnt-instrument -o foo.instr.scs ihp25 foo.scs
```

You can also overwrite the old scs file with the new one using the `-i` or `--in-place` option:

```
$ tnt-instrument -i ihp25 foo.scs
```

You can also use `-n` or `--nodes` to specify a custom path for the node list, and `-s` or `--sources` to specify a custom path for the source list.

Ocean Projects

Once we have a circuit we can debug, we create a script to determine how the circuit is to be debugged with the `tnt-ocean` tool

The ultimate purpose of `tnt-ocean` is to produce a script that can be feed to Ocean alongside the netlist produced with `tnt-instrument` to simulate the analog circuit and analyze it.

To do so, we start by initializing the project—preferably in an empty folder—with the `--init` option:

```
$ tnt-ocean --init
$ ls
config   inject   watch
```

As you can see, the project is initialized with three files:

- **config** contains the miscellaneous options about the inputs, outputs, and internals of the experiment to be run.
- **inject** contains the definitions of all the injectors to be used during the simulation to add damages.
- **watch** contains the definitions of all the elements to be watched during the simulation, and what kind of behaviour should trigger a warning.

The Config File

This is the main configuration file for a project, and consists in a number of sections labelled `[section]`, each containing a number of `key = value` lines. Comments explaining what the file does start with `#`.

```
#------------------------------------------------------------------------------
# Input options
#------------------------------------------------------------------------------

[input]

# The technology is used to specify the techfile. Currently supported
# technologies and the key word for specification here are:
#
#    Technology                                        key word
#    ST Microelectronics 130nm 1P/6M 1.2V CMOS          st130
#    UMC 180nn 1P/6M 1.8V CMOS                          umc180
#    IHP Microelectronics 250nm 5M 2.5V SiGe BiCMOS     ihp25

technology=
```

The Input Section

technology specifies the technology on which the design runs; values require that a valid *techfile* exists. Known values are:

- **st130**: ST Microelectronics 130nm 1P/6M 1.2V CMOS
- **umc180**: UMC 180nn 1P/6M 1.8V CMOS
- **ihp25**: IHP Microelectronics 250nm 5M 2.5V SiGe BiCMOS

watch_path points to the *watch* file that was created alongside *config* and contains the definitions of all signals to be watched during the simulation.

inject_path points to the *inject* file that was created alongside *config* and contains the definition of all elements where faults may be injected during the simulations, as well as parameters that define what kind of faults.

source_path points to a sourcelist generated by `tnt-instrument`; this is a list of all extraneous sources inserted in the design to debug it.

The Output Section

script_path is the location and name of the Ocean script that will result when running `tnt-ocean`

csv_path is used to specify the file with analysis results that will be generated by the Ocean script.

The Analysis Section

netlist_path points to the design; this is usually an modified netlist as generated by `tnt-instrument`. For Spectre's sake, this file ought to be part of a complete Cadence project.

gold_path is used to store temporary data about the control run

rad_path is used to store temporary data about the experiment runs

heuristic determines which algorithm will be used in the analyze function to profile all watches. This determines the arguments that must be provided for each element in the watch file. Valid values are:

- **deviation_recovery**: compares the difference between the non-irradiated signal and the observed one, measuring the time that it is greater than a user defined threshold and the maximum difference between both signals
- **time_ranges**: measures the total time a signal is inside the five ranges defined by four user provided thresholds during a fixed time period.

analysis_time determines how long the analysis will last. Must be expressed on time units such as *s* (seconds), *ms* (milliseconds), *us* (microseconds) …

analysis_unit divides the output of all watched elements before printing them to the CSV file. A good choice produces more compact and legible data.

analysis_step is the sampling rate for the simulation (i.e. time between two reads of every declared watch)

analysis_type determines the kind of analysis that will be performed in the circuit. This is directly passed as argument to the simulator; see its documentation for more information.

analysis_accuracy allows the following values: conservative, moderate, liberal. This is directly passed to the Spectre simulator; see its documentation for more information.

initial_injected_charge is the default value for the charge in each source

initial_impact_time is the default initial injection time. Must be greater than *analysis_time*, and follows the same formatting conventions

enable_plotting enables plotting of signals if set to true.

analysis_initial_time is only used by the *time_ranges* heuristic. Determines the initial time for sampling.

Writing Injectors

The inject file determines where, when, and how will damages be injected during the simulation process. It contains a list of `inject` blocks.

```
inject name1, name2,...:
    Q = charge1, charge2...;
    t = time1, time2...;
```

This means that all elements with the names `name1`, `name2`, etc. are injected with charges from the `Q` list, at times from the `t` list.

The elements in the inject file can be either written by hand using as template the sources in the source list generated by tnt-instrument, or they can be automatically managed with the `tnt-inject` tool.

You can use `tnt-inject -i` with the source list generated by tnt-instrument to create a file containing injection declarations for all sources, but all blocks will be initially commented:

```
$ tnt-inject -i foo.sources foo.inject
```

To list all commented blocks in this file, use the `-E` or `--list-enabled` option, and to list all uncommented blocks, use the `-D` or `--list-disabled` option:

```
$ tnt-inject -D foo.inject
$ tnt-inject -E foo.inject
```

To enable a block that was disabled, use the `-e` or `--enable` option, and to disable a block that was enabled use the `-d` or `--disable` option:

```
$ tnt-inject -e node foo.inject
$ tnt-inject -d node foo.inject
```

Finally, to set the charge and time lists of specific blocks, use the `-s` or `--set` option:

```
$ tnt-inject -s node.Q:1,2,3,4 foo.inject
```

Of course, you can also print the value associated with a particular key using the `-g` or `--get` option:

```
$ tnt-inject -g node.Q foo.inject
1,2,3,4
```

Writing Watches

The watch file determines what elements are analyzed during the simulation process; as the inject file contains `inject` blocks, the watch file contains `watch` blocks, but the formats allowed depend on the value of the `heuristic` field in the config file.

If the chosen heuristic is **deviation_recovery**, the watch bloks are:

```
watch <name> = <expression> :
    threshold = <number> ;
```

Whereas if heuristic is **time_ranges**, they are:

```
watch <name> = <expression> :
    thr_min = <number> ;
    thr_max = <number> ;
    thr_med_max = <number> ;
    thr_med_min = <number> ;
```

In the same way the inject file can be generated by `tnt-inject` from the source list, the watch file can be generated by `tnt-watch` from the node list, but it requires an extra argument to be passed: the name of the heuristic used to analyze the elements.

```
$ tnt-watch -i foo.sources -h deviation_recovery foo.watch
```

To list all commented blocks in this file, use the `-E` or `--list-enabled` option, and to list all uncommented blocks, use the `-D` or `--list-disabled` option:

```
$ tnt-watch -D foo.watch
$ tnt-watch -E foo.watch
```

To enable a block that was disabled, use the `-e` or `--enable` option, and to disable a block that was enabled use the `-d` or `--disable` option:

```
$ tnt-watch -e node foo.watch
$ tnt-watch -d node foo.watch
```

Finally, to set the charge and time lists of specific blocks, use the `-s` or `--set` option:

```
$ tnt-watch -s node.Q:1,2,3,4 foo.watch
```

Of course, you can also print the value associated with a particular key using the `-g` or `--get` option:

```
$ tnt-watch -g node.Q foo.watch
1,2,3,4
```

Generating the Final Script

Once the config, inject, and watch files are ready, you may invoke `tnt-ocean` again to generate a Lisp script from them:

```
$ tnt-ocean config
```

The tool takes the paths to the inject and watch files from lines in the config file, so you only need to specify this last one.

If all goes right, a new lisp script will appear, in the path specified by the value of the `output.script_path` element in the config file.

Reference Manual

This appendix is a summary of the Tcl library that allows interfacing with FTUNSHADES systems. The same documentation is available through the `help` command, albeit formatted in a more limited way.

General Commands

help command

Print help

Param. *command* name of a Tcl command

exit

Prints an error message

Overrides the native Tcl **exit** command

Devices

dev_ls

List devices

List all FTUNSHADES devices connected to the machine where the shell is running. This function will refuse to run if a device is currently open, because FTDI quirks.

For every existing device, the function returns a list containing its serial number, description, manufacturer, and a last field whose value will be "yes" if the device can be opened or "no" if it is currently in use by other session.

The strings returned by this command are the same as would be returned by dev_string.

Return a list of all devices

dev_open serial

Open device

Given the serial number of a device that is not currently in use, acquire and prepare it to be used by the current session. If no serial number is given, the first available device will be opened instead. On success, returns the serial number of the opened device.

Param. *serial* optionally, the desired serial number
Return the serial string of the opened device

dev_close

Close device

Close the device bound to this particular session. Once closed, the device becomes free again to be opened by any other session.

dev_manufacturer

Return device manufacturer string

Return a string

dev_description

Return device description string

Return a string

dev_serial

Return device serial string

Return a string

dev_timeout seconds

Set timeout for read operations

Reset the timeout value for read operations in the USB connected to the device.

Param. *seconds* desired timeput
Return the previous timeout value

dev_flush

Flush the USB buffers on a device

Return the number of flushed bytes

dev_error

Query motherboard error status

Return a string

dev_reboot

Reboot connected device

Send a signal to the device bound to this session that bypasses the command interpreter to force a hard reset.

dev_ping string

Ping device

Request the device currently bound to this session to send back string.

Param. *string* the string to be sent
Return *string*

dev_info

Print help

Return a string

Expansion Cards

xc_target card

Select expansion card

Select the board that will be target of board-specific operations. For the FTUNSHADES 2 system, the valid targets are:

- "golden" (o "g"): Golden daughter board
- "faulty" (or "f"): Faulty daughter board
- "all" (or "a"): All boards at once

Note that the "all" value must be only used to execute commands that do not read from the expansion cards. Doing otherwise will produce unexpected and probably fatal results.

Param. *card* the card to select

xc_error card

Query expansion card error status

Return a string describing the current error state of an expansion card on the device currently bound to this session.

If cardName is not specified, the last cardName specified will be used.

Executing this function with a cardName value of all produces undefined results.

Param. *card* the card to query (optional)
Return a string

xc_reboot card

Reboot expansion card

Send a signal to a expansion card on the device bound to this session that bypasses the command interpreter to force a hard reset.

If cardName is not specified, the last cardName specified will be used.

Param. *card* the card to reboot (optional)

xc_ping string card

Ping expansion card

Request a expansion card on the device currently bound to this session to send back string.

If cardName is not specified, the last cardName specified will be used.

Executing this function with a cardName value of all produces undefined results.

Param. *string* data to send
Param. *card* the card to ping (optional)
Return *string*

xc_configure card

Configure TFPGA on expansion card

Configure the target FPGAs on an expansion card on the device currently bound to this session with the bitstream object bound to FPGA_CONFIG.

If cardName is not specified, the last cardName specified will be used.

Param. *card* the card to configure (optional)

xc_deconfigure card

Deconfigure TFPGA on expansion card

Clear the configuration of the FPGA on an expansion card on the device currently bound to this session.

If cardName is not specified, the last cardName specified will be used.

Param. *card* the card to deconfigure (optional)

xc_is_configured card

Query configuration status of TFPGA on expansion card

Return 1 if the FPGA on an expansion card on the device currently bound to this session is correctly configured; 0 otherwise.

If cardName is not specified, the last cardName specified will be used.

Executing this function with a cardName value of all produces undefined results.

Param. *card* the card to query (optional)

is_obj name

Determine if name corresponds to an object

Param. *name* name of an object
Return a boolean

obj_new

Create object

Create a new object on the device and return its name

Return the object name

obj_destroy name

Destroy object

Destroy the object reference by name. If name is currently bound to a target, it will be unbound.

Param. *name* the object to destroy

obj_bind target name

Bind object

If called with a valid object name, bind the object associated with the name to target. If called with 0, unbind the object bound to target.

Target must be one of: FPGA_CONFIG, FPGA_VECTORS, CAMPAIGN_RESULTS.

Param. *target* target to bind *name* to
Param. *name* object to bind to *target*

obj_ls

List objects

Return a string list

File Loading

load_bit path name

Load bitstream

This is a low level utility to load bitstreams. Most of the time, users should prefer to use load_config instead.

Open and read the contents of the file found in path. If it contains a valid bitstream for the target FPGAs, initialize the data of the object whose name is given with the data from the file.

Param. *path* a bit file
Param. *name* an object to load the data to

load_dat path name

Load vectors

This is a low level utility to load vectors. Most of the time, users should prefer to use load_vectors instead.

Open and read the contents of the file found in path. If it contains a valid vector list for the target FPGAs, initialize the data of the object whose name is given with the data from the file.

Param. *path* a vector file
Param. *name* an object to load the data to

load_ll path mount_point

Load registers

Open and read the contents of the logic location file found in path. If it contains a valid register tree, merge it with the session bit tree. If mountPoint is specified, it will be used as the root directory for the new bits.

Param. *path* a ll file
Param. *mount_pount* node in the bit tree that is considered root for all
paths in *path*

load_pin path mount_point

Load pins

Open and read the contents of the pin file found in path. If it contains a valid pin tree, merge it with the session bit tree. If mountPoint is specified, it will be used as the root directory for the new bits.

Param. *path* a pin file
Param. *mount_pount* node in the bit tree that is considered root for all
paths in *path*

Bit Tree

bit_rm patterns

Remove nodes

Param. *patterns* patterns to match nodes in the bit tree

bit_type path

Return node type

Param. *path* path of a node in the bit tree

bit_cd path

Change current node

Param. *path* path of a node in the bit tree

bit_wd

Print current node

bit_ls pattern

List nodes

Param. *pattern* pattern to match nodes in the bit tree

bit_glob patterns

Glob nodes

Param. *patterns* patterns to match nodes in the bit tree

bit_watch path

Create watch

Creates a watch associated with path, so the value of all associated bits are read from both target FPGAs after every step and skip instruction.

Param. *path* path of a node in the bit tree

bit_unwatch path

Destroy watch

Destroys the watch associated with path. The logs associated with the bits contained in path may be destroyed if no more watches exist on them.

Param. *path* path of a node in the bit tree

bit_watches

List watches

Returns a list of the names of all nodes with watches on the register tree.

Return a string list

bit_log patterns chan cycle end

Print past values of watched bits

If all bits matching pattern have active watches on them, return the slice of the log in the [start,end] range for the given channel.

- If start is not given, the current clock cycle will be used instead.
- If end is not given, the same value as start will be used instead.
- Valid values for channel are "golden", "g" for the golden target FPGA and "faulty", "f" for the faulty target FPGA.

Param. *patterns* patterns to match nodes in the bit tree
Param. *chan* name of a expansion card
Param. *cycle* first cycle to retrieve
Param. *end* last cycle to retrieve
Return a string list

bit_is_leaf path

Print help

bit_read patterns chan

Read current bit values

Get the value of the bits matching pattern in the target FPGA that corresponds to channel.

Param. *patterns* patterns to match nodes in the bit tree
Param. *chan* name of a expansion card
Return a string

bit_write patterns chan value

Write current bit values

Set the value of the bits matching pattern in the target FPGA that corresponds to channel.

Param. *patterns* patterns to match nodes in the bit tree
Param. *chan* name of a expansion card
Param. *value* a string

bit_dump patterns mask path

Dump past values of warched bits to file

Save the full log for the bits identified by pattern to the file whose path is given.

Debugging

clk_read

Return current emulation cycle

Return the current cycle

clk_rewind

Go back to first emulation cycle

clk_step cycles

Emulate and record all values

Advance the given number of cycles, recording the value of every log every step. If cycles are not specified, it will advance one cycle.

Param. *cycles* number of cycles to emulate

clk_skip cycles

Emulate and record last value

Advance the given number of cycles, only stopping at the end to record the value of every log after last step. The values of the cycles-1 cycles before it will be marked as unknown.

Note that since one cycle is always recorded, clk_skip 1 is equivalent to clk_step 1.

Param. *cycles* number of cycles to emulate

clk_step_until

Emulate and record all values

Advance until the end of vectors is reached or a discrepancy is found in the output vectors, recording the value of every log every step. If cycles are not specified, it will advance one cycle.

clk_skip_until

Emulate and record last value

Advance until the end of vectors is reached or a discrepancy is found in the output vectors, only stopping at the end to record the value of every log after last step. The values of the cycles before it will be marked as unknown.

Campaigns

run_set key val

Set campaign properties

Will set the value of the given variable whose name is key, that must be one of the following strings:

batch_size: Determines the number of injections that are sent at once every time to the device. This value does not change the campaign results, but fine-tuning it may make it end faster. By default, it is set to 1024.

check_residual_damage: If set to 1, the output vectors will be checked for damage the cycles before injection every run. Defaults to 0.

damage_per_run: If set to a number greater than zero, runs stop the nth cycle a discrepancy is found in the outputs. May make certain campaigns faster. Defaults to 1.

drop_on_damage: If set to 1, the campaign not only stops checking for damage after damage_per_run, but stop emulating cycles for that run and jump to the next. If set to 0, errors are not checked but cycles are emulated. Defaults to 0.

show_output_xor: If set to 1, a bitmask of the bytes where the GOLD and SEU output vectors vary is returned for every cycle where damage is detected. Defaults to 0.

unflip_after_run: If set to 1, after the run ends, it attempts to fix the damage to the emulated circuit by unflipping the bits associated with these registers that we changed. Depending on the internals of the FPGA, this may work or not (certain registers propagate when flipped and make this solution unfeasible). Defaults to 0.

reconfigure_at_error: If set to 1, the SEU target FPGA is reconfigured after every run where damage is detected. Defaults to 0.

reconfigure_at_time: If set to any value other than 0, the SEU target FPGA is reconfigured every **reconfigure_at_time** runs. Defaults to 0.

readback_at_error: If set to 1, the internal state of the SEU target FPGA is dumped to a file after every run where damage is detected. Defaults to 0.

readback_at_time: If set to any value other than 0, the internal state of the SEU target FPGA is dumped to a file every **readback_at_time** runs. Defaults to 0.

Param. *key* name of a property
Param. *val* value for *key*

run_get key

Get campaign properties

Param. *key* name of a property
Return the value of *key*

run_random regs cycles nruns ninjs seed

Run random campaign

Every injection takes one register from the target register list and one cycle from the target cycle list, randomly. If a seed is given, the random number generator is feed it before starting.

Param. *regs* list of target registers
Param. *cycles* list of target cycles
Param. *nruns* number of runs
Param. *ninjs* number of injections per run

run_exhaustive patterns cycles

Run exhaustive campaign

The campaign will inject in every possible combination of the target cycle and register lists. The total number of runs will be the sizes of both, multiplied.

Param. *regs* list of target registers
Param. *cycles* list of target cycles

run_custom regs cycles nruns ninjs command

Run custom campaign

Generates injections with a user provided function that takes four parameters:

- The size of the register list that resulted from the registers pattern.
- The size of the cycle list that resulted from the cycles pattern.
- The current run number.
- The current injection number.

And it must return a pair of indices, of which the first one refers to the target register list and the second one to the target cycle list.

Param. *regs* list of target registers
Param. *cycles* list of target cycles
Param. *nruns* number of runs

Param. *ninjs* number of injections per run
Param. *command* name of a Tcl command

run_file path

Run file campaign

Run campaign using the file injector, that reads injections from a user-provided file.

The target register and target cycle lists are determined by reading the file, so there's no need to manually provide them. Still, the registers referenced must exist in the register tree, and there must be vectors for every cycle up the the greatest one used in the file.

Every line in the file corresponds to a run and must be a comma-separated list of cycle:register pairs, as follows:

```
10:/reg/0;
10:/reg/1,10:/reg/2;
10:/reg/3;
10:/reg/4,10:/reg/5,10:/reg/6;
10:/reg/7;
```

Remember that for every run, injection pairs must be sorted by cycle.

Param. *path* a injection file

run_clean nruns

Run clean campaign

Execute a campaign in which no injections are performed. The only argument required is the number of runs.

Param. *nruns* number of runs

run_progress

Print progress of currenc campaign

run_abort

Abort current campaign

run_status

Return 1 of there is campaign active, 0 otherwise

Miscellanea

dbg_smap_bt_enable

Enable SelectMap backtrace

After this function is called, every command sent to the SelectMap will be registered in a list that can be read with *dbg_smap_dump*.

dbg_smap_bt_disable

Disable SelectMap backtrace

Stop registering SelectMap events.

dbg_smap_bt

Dump SelectMap backtrace

Write the contents of the SelectMap backtrace to the output; the backtrace will be emptied. Note that the backtrace will not contain elements unless *dbg_smap_backtrace* was called and then some SelectMap events generated.

dbg_frame_read frame

Read a full frame

Write the formatted contents of the frame *frame* to the standard output. Note that this function will not take measures to prevent corruption of RAM frames.

Param. *frame* index of the frame to read

dbg_bit_set frame offset value

Set the value of a bit

Change the value of bit *offset* in frame *frame* to *value*. Note that this function will not take measures to prevent corruption of RAM frames.

Param. *frame* index of the frame containing the bit
Param. *offset* index of the bit in the frame
Param. *value* new value for the bit

dbg_bit_get frame offset

Get the value of a bit

Return the value of bit *offset* in frame *frame*. Note that this function will not take measures to prevent corruption of RAM frames.

Param. *frame* index of the frame containing the bit
Param. *offset* index of the bit in the frame
Return bit value

dbg_bit_flip frame offset

Flip the value of a bit

Flip the value of bit *offset* in frame *frame*. Note that this function will not take measures to prevent corruption of RAM frames.

Param. *frame* index of the frame containing the bit
Param. *offset* index of the bit in the frame

dbg_read_frames start len path

Read consecutive frames

Write the formatted contents of *len* frames starting at *start* to the file whose path is *path*. Note that this function will not take measures to prevent corruption of RAM frames.

Param. *start* index of the first frame to read
Param. *len* number of frames to read
Param. *path* path to a file where output will be written to.

Throubleshooting

FTUNSHADES is an experimental system. The good of it is that we keep pushing new features all the time. The bad is that sometimes you'll find the system behaving in unexpected ways. And by unexpected we mean broken.

So we have included a section on what to do when things go pear-shaped, with some recipes we have developed along the way.

Do This If You Can't Open a Supposedly Available Device

When multiple users access the same TNT installation in a server, it may happen that `dev_open` fails with the following message:

```
% dev_ls
Serial  Description           Manufacturer           Available
1       ftu2_xc5vfx70tff1136  AICIA - Univ. Sevilla  yes
% dev_open
/usr/local/var/tnt/LCK.1: File exists
```

The file `/usr/local/var/tnt/LCK.1`---or one very similar to it: the directory depends on installation options and the number on the index of the device you're trying to open---is created by `dev_open` to ensure two users don't attempt to use the same FTUNSHADES device at the same time. In normal circumstances, it is destroyed by `dev_close`, but if the session is killed in such a way the device is not released and the file deleted, and the `umask` for `tntsh` is not correctly set, when a different user attempts to open the device, it will be unable to read or modify the file lock, and because of that, the device will not be opened.

Normal circumstances, i.e.: those that prevent this problem, are as follows:

- The `umask` for the `tntsh` process must be `0002`.
- The `/usr/local/var/tnt` folder belongs to the `ftu` group.
- All users that are supposed to use the TNT shell belong to the `ftu` group.
- This includes the one used by the HTTP server to provide the UFF, if there is one.

This ensures new files created inside it are readable and modifiable.

Error Messages From FTUNSHADES Devices

Tcl knows how to tell apart the return values associated with a successful function call and those associated with an error. In the inside, there are some possible values being returned (TCL_OK, TCL_ERROR, etc.). Functions in the TNT shell will return the expected values associated with an Ok code on success and an error message associated with an error code on failure.

```
% load_bit $FPGA_CONFIG "file_larger_than_512MB.bit"
Out of memory
```

The error messages you may get are:

Ok: Everything is peachy. Except this is not an error code and shouldn't be output to the shell. Unless you are calling the `dev_error` command to flush an uncaught error from the FTUNSHADES device, but if so we'll assume you know what you're doing.

Unexpected instruction: The state machine received a command when it expected data. If you get this, the most probable explanation is that there is a mismatch between the versions of TNT and the embedded software.

Unexpected data: The state machine received data when it expected a command. Same as the previous one.

Invalid instruction: The state machine received a command it does not recognize. Probably means a version mismatch, too.

Value out of range: The state machine received a value that is not in the expected range. Examples may be a cycle greater than the size of the current vector list or a object type identifier that does not match with the ones the board manages internally.

Invalid operation in the current context: You requested an operation that can not be performed given the current internal state of the FTUNSHADES device. This may happen, for example, if you attempt to configure the target FPGAs without uploading a bitstream first.

Out of memory: You uploaded too much data, and it just doesn't fit in the device's memory. Most of the time it can be fixed by destroying some resident objects, but it may also mean you have attempted to use provide a vector list that does not fit, or set a campaign batch size too big.

What to Do If the Device Says Gibberish or Dies

There are some error conditions of which the FTUNSHADES device is not aware, or is unable to answer user queries.

One of the less problematic ones is the reverse of the`'unexpected instruction',*"unexpected data", and*`*'invalid instruction*' codes. It happens when the FTUNSHADES board has sent a response larger than the TNT library expected. That means that whenever you request more data, you get a response for a command you sent some time ago, first. This error can be easily identified by pinging the board.

```
% dev_ping hello
bad response: '.hell'
```

You can see we received as many characters as we sent, but not the same ones. The "`o`" character from the ping command is still in the USB buffers and will contaminate the response of the next command. To fix this, you can execute `dev_flush`:

```
% dev_flush
1
```

The function will return the number of bytes removed from the buffer.

There are other possible error conditions, not as easily fixable. In many cases, these conditions prevent the board from responding. Again, try pinging:

```
% dev_ping hello
ftu_usb_recv: timed out after 1 seconds (0 bytes read)
```

This error will normally be caused by an abnormal condition of the service FPGAs and the motherboard expecting a response from them that is not going to come. When this kind of error happens, the data on board the different device components is all but lost and you must force a reboot. The simplest way to do so is with the `dev_panic` command:

```
% dev_panic
```

This command is pretty throrough, rebooting all modules of the FTUNSHADES device and erasing the configuration on the target FPGAs. The source code for it is as follows:

```
proc dev_panic {} {
    if {[catch dev_reboot} ret] != 0} {
        error "Can't reboot mother board: $ret"
    } elseif {[catch xc_reboot gold} ret] != 0} {
        error "Can't reboot GOLD daughter board: $ret"
    } elseif {[catch xc_reboot seu} ret] != 0} {
        error "Can't reboot SEU daughter board: $ret"
    } elseif {[catch xc_deconfigure gold} ret] != 0} {
        error "Can't deconfigure target GOLD FPGA: $ret"
    } elseif {[catch xc_deconfigure seu} ret] != 0} {
        error "Can't deconfigure target SEU FPGA: $ret"
    }
}
```

If this does not work… well, then it's going to be a hardware problem. Try turning it off and on again.

Then, contact your system's administrator.

Batch Sizes and Out of Memory Errors

The FTUNSHADES2 devices have 0.5GB of on board memory that is used in a number of operations.

This memory has to fit the bitstreams to configure the target FPGAs, vectors to feed them, and during campaigns it must also contain the campaign description (times and places for injection) and output.

This means that it is possible to launch campaigns too big for the device to handle, and although errors will be raised in the device, the way campaigns work means that it'll be quite a time since you launch the campaign to he point it fails.

To prevent these problems, you can do some things.

Ensure there are no unneeded bitstream or vector objects on the device. When you start loading these from file, the first object will have an id of 1, the second of 2, etc. If you get other numbers, that means you left something on the device. Release those objects manually or execute `dev_panic` to clean up everything.

Adjust the `batch_size` variable. This determines the number of runs that will be executed in each batch of your campaign. The problem is that "correct" values are pretty fuzzy. Large batch sizes (as is the default) are perfectly good when the amount of data you request every run is small, but enabling such flags as `readback_on_error` and `readback_on_time` will greatly increase the amount of memory required per batch, since the requested data has to reside in the intermediate memory while the batch is being run, and the motherboard will not send them to the server until the batch is complete.

Additionally, a large `batch_size` will reduce the campaign resolution. Campaigns can only be stopped between batches, at the point the batch data is retrieved and new data is sent, so large values can result in single batch campaigns that can't be stopped.

On the other hand, the larger the `batch_size` the faster the campaign, since the USB connection with the server is one of the worst bottlenecks in the system.

Segmentation Fault on Clock Command After Watch Creation

This is a known issue related to the bit log API that handles storage of large waveforms in the server's memory.

It is triggered by code like the following:

```
% dev_open 1
% load_ll counter8bit.ll
% load_config counter8bit.bit
% load_vectors counter8bit.dat
% xc_configure all
% clk_rewind
% bit_watch reg
% clk_step 10
```

This code causes `tntsh` to crash with a message like:

```
Program received signal SIGSEGV, Segmentation fault.
```

The problem seems to be the creation of a watch immediately after the `clk_rewind` instruction, but this isn't always the case.

This can be prevented by creating the watches before `clk_rewind` is called; just flip those two lines:

```
% bit_watch reg
% clk_rewind
% clk_step 10
```

Other Things You May find Confusing

This appendix contains some lengthy explanations of topics that haven't been touched as in depth in the rest of the manual, as well as a troubleshooting guide.

The Bit Tree

Conversions at Loading Time

When elements are added to the bit tree from a logic location or pin file, the paths are changed to prevent collisions with certain TNT shell features:

Square brackets (`[]`), curly braces (`{}`), and angles (`<>`) are all converted to slashes (`/`). In the case of square brackets and angles, this is done because they are special characters in the Tcl language and in patterns recognized by the `fnmatch` function, and escaping them is exceedingly laborious and time-consuming.

Also, these characters are used to designate nodes that share a common meaning, such as `reg<0>`, `reg<1>`, `reg<2>`... so, to better index them in the bit tree, they become `reg/0`, `reg/1`, `reg/2`...

Rules for Pattern Matching

By default, the FTUNSHADES libraries depend on the `fnmatch` C function to match names of individual nodes in the bit tree. This means that slashes are only matched by other slashes and the rest of the characters are matched to patterns that may include the following wildcards:

- The asterisk (`*`) matches zero or more characters.
- The question mark (`?`) matches exactly one character.
- Ranges of characters enclosed in square brackets, e.g.: `[a-zA-Z]` match one of any single character within the ranges.
- Ranges of characters enclosed in square brackets preceded by an exclamation mark, e.g.: `[!0-9]` match one of any character that is not in the ranges.

It is very important to note that anytime you want to use a bracket-enclosed range in a pattern from the Tcl shell, you must enclose the full string in braces. This will not work.

```
% bit_ls foo/[0-9]
% bit_ls "foo/[0-9]"
```

Typing any of these two would result in an error message:

```
invalid command name "0-9"
```

This happens because Tcl thinks it is supposed to parse the bracket content and execute it as a expression. Enclosing the string in curly braces prevents this from happening

```
% bit_ls {foo/[0-9]}
```

Sorting Order for Bit Tree Nodes

The first thing that must be noticed is that the slash (`/`) character is in fact a separator in the register name that separates different levels in the hierarchy. This means that it is not interpreted alphanumerically. Whenever a sorting operation is performed, it is between the tokens in the full path that lie among the slashes. These tokens are assumed to be the names of nodes in the register tree.

The child nodes of every directory in the bit tree are sorted lexicographically except when the name of both nodes can be parsed into an integer; in this case, they are sorted numerically.

Why do this? Well. This is because it seems to be standard in logic location files to enumerate consecutive registers in a certain way, because the numbers provided by the programs that generate the logic location files do not fill the numbers with zeroes in the left, so lexicographical sorting is not the same as numeric sorting.

You can see the difference between the sorting options here:

\noindent \begin{center} \begin{tabular}{|c|c|} \hline Lexicographical Sorting & Numeric Sorting\tabularnewline \hline \hline foo/0 & foo/0\tabularnewline foo/1 & foo/1\tabularnewline foo/10 & foo/2\tabularnewline foo/2 & foo/3\tabularnewline foo/3 & foo/4\tabularnewline foo/4 & foo/5\tabularnewline foo/5 & foo/6\tabularnewline foo/6 & foo/7\tabularnewline foo/7 & foo/8\tabularnewline foo/8 & foo/9\tabularnewline foo/9 & foo/10\tabularnewline \hline \end{tabular} \par\end{center}

Attacking Configuration Bits a.k.a. FPGA Mode In Depth

When a design is not only tested by putting it into an FPGA, but actually intended to go into production this way, it becomes vulnerable to damages in not only the user registers, but also the configuration bits.

From the point of view of the bitstream used to program the target FPGAs, there is no difference between those elements that define how the circuit behaves and the ones that hold the data. The problem comes from the lack of a register map that can be used to target them, so as part of the toolchain, we have created a process that can extract that information from the design files and dump it to a secondary logic location file usually called `config.ll`.

Producing a Config Location File

A config location file is just a logic location file for the configuration bits of the target FPGA. This is not naturally produced by the same tools that generate the design, but by a tool from TNT called `tnt-ebd2ll`.

This tool must be invoked from command line (not the Tcl shell) and requires a binary .ebd file, that is produced along the rest of the design. Invocation is pretty simple:

```
$ tnt-ebd2ll foo.ebd
```

This will produce a `config.ll` file in the same directory it was called.

Using a Config Location File

With this file in hand, targeting the registers is easy. First, load the registers:

```
% load_ll "config.ll"
% bit_ls
config/
```

Then, execute the campaign, targeting the registers in `config/`:

```
% load_config "bar.bit"
% load_vectors "qux.dat"
% run_random "config/" "0-100" 100
```

Partial Reconfiguration of the Target FPGAs

It is perfectly possible to configure the target FPGAs “in pieces” with data from multiple bitstreams. First, we program the devices as usual:

```
% load_config "whole.bit"
1
% xc_configure all
```

Then, we load the additional bitstreams we want, and this time keep the names:

```
% set fragment1 [load_config "fragment1.bit"]
2
% set fragment2 [load_config "fragment2.bit"]
3
% set fragment3 [load_config "fragment3.bit"]
4
```

At any moment we can perform a partial reconfiguration as follows:

```
% obj_bind $FPGA_CONFIG $fragment1
2
% xc_configure all
```