GreatFET One Documentation

Great Scott Gadgets

Feb 03, 2024

USER DOCUMENTATION

1 GreatFET and GNURadio **3**
- 1.1 Requirements . . . 3
- 1.2 Adding our blocks to GNURadio-companion . . . 3

2 Linux Distribution Python Version Matrix **5**

3 Troubleshooting **7**
- 3.1 Why do I get a `No GreatFET devices found!` error when my GreatFET is plugged in? . . . 7

4 Using GreatFET APIs **9**
- 4.1 Getting a GreatFET object . . . 9
- 4.2 Using APIs . . . 9
- 4.3 Debugging . . . 9
- 4.4 LED . . . 10
- 4.5 GPIO . . . 10
- 4.6 Logic Analyzer . . . 10
- 4.7 Pattern Generator . . . 11
- 4.8 UART . . . 11
- 4.9 ADC . . . 11
- 4.10 DAC . . . 11
- 4.11 SWRA124/Chipcon . . . 12
- 4.12 I^2C . . . 12

5 greatfet_i2c **13**
- 5.1 Pin Usage . . . 13
- 5.2 Example Usage . . . 13
- 5.3 API . . . 14

6 greatfet_logic **15**
- 6.1 Pin Usage . . . 15

7 greatfet_msp430 **17**
- 7.1 Pin Usage . . . 17

8 Getting Started with Firmware Development **19**
- 8.1 Fresh install . . . 19
- 8.2 Updating your repository . . . 19
- 8.3 Installing host tools . . . 19
- 8.4 Building and flashing firmware . . . 20
- 8.5 Re-Building and flashing firmware . . . 20

9 LibGreat Verb Signatures **21**
9.1 Describing Verb Signatures . . . 22
9.2 Repeat Specifiers . . . 23
9.3 Length Specifiers . . . 23
9.4 Element Groups . . . 23
9.5 Examples . . . 24
9.6 Omitting Verb Signatures . . . 25

10 GreatFET Classes **27**
10.1 Class Registry . . . 27

11 Neighbors **29**
11.1 already designed . . . 29
11.2 some progress made . . . 29
11.3 rough ideas . . . 30

12 How to Design a Neighbor **31**
12.1 Use a Template . . . 31
12.2 Required Elements . . . 31
12.3 Bonus Row . . . 32
12.4 Orientation . . . 32
12.5 Neighbor Identification . . . 32
12.6 Making Your Neighbor Extra-neighborly . . . 32
12.7 Pin Selection . . . 33

13 Board Naming **35**
13.1 Code Names . . . 35
13.2 GreatFET Trademark . . . 35
13.3 GreatFET and Great Scott Gadgets URLs . . . 35
13.4 Naming Your Neighbor . . . 36

14 I2C Registry **37**

15 Release Process **39**
15.1 tag the release . . . 39
15.2 make the release directory . . . 39
15.3 copy/update RELEASENOTES from previous release . . . 39
15.4 make second clone for firmware build . . . 39
15.5 update the firmware VERSION_STRING and compile firmware . . . 40
15.6 make the release archives . . . 40
15.7 “Draft a new release” on github . . . 40
15.8 announce the release . . . 40

The GreatFET project produces interface tools for hardware hacking, making, and reverse engineering.

CHAPTER

ONE

GREATFET AND GNURADIO

The GreatFET platform can easily interface with GNURadio – and several gnuradio-companion blocks are available to make using GreatFET+GNURadio easier. These blocks include support for various neighbors (e.g. Gladiolus blocks for Software Defined Infrared) and a variety of other inputs and outputs, which are lumped into the 'Software Defined Everything' category.

1.1 Requirements

You'll need an up-to-date GNURadio environment to use our blocks. This means:

- GNURadio >= 3.8, built with python3*
- python 3.6+

It's possible these blocks will work with GNURadio on lower python versions; but these aren't our development targets and aren't fully supported.

1.2 Adding our blocks to GNURadio-companion

The easiest way to add our blocks to GNURadio is to modify our local GNURadio configuration. On Linux and macOS, this file is located at `~/.gnuradio/config.conf`.

Open this file, and find the section that's headed `[grc]`; or create the relevant section if it doesn't exist.

```
[grc]
<existing keys here>
```

We'll want to add the location of our blocks to the configuration file's `local_blocks_path`. We can determine the location of our blocks using the `gf info` command:

```
$ gf info --host
Host tools info:
        host module version: 2019.5.1
        pygreat module version: 2019.9.1
        python version: 3.8.0 (default, Oct 23 2019, 18:51:26)

        module path: /home/user/.local/lib/python3.8/site-packages/greatfet
        command path: /home/user/.local/lib/python3.8/site-packages/greatfet/commands
        gnuradio-companion block path: /home/user/.local/lib/python3.8/site-packages/
↪greatfet/gnuradio
```

In the output above, the last line points out our GRC block path. We'll add this to the `local_block_path` entry in our configuration file:

```
[grc]
local_blocks_path = /home/user/.local/lib/python3.8/site-packages/greatfet/gnuradio
```

If you want to have multiple entries, here – for example, if existing entries are already present – you can separate multiple paths using colons, similar to Linux paths:

```
[grc]
local_blocks_path = /home/user/.local/lib/python3.8/site-packages/greatfet/gnuradio:/
↪home/user/my_blocks
```

The next time you start GRC, you should see new headings (e.g. `SDIR` and `Software Defined Everything`) in your list of available blocks.

CHAPTER

TWO

LINUX DISTRIBUTION PYTHON VERSION MATRIX

	3.5	3.6	3.7	3.8
Ubuntu	16.04 LTS (Xenial)	18.04 LTS (Bionic)	19.04 (Disco)	None
Debian	Oldstable (Stretch)	None	Stable (Buster)	Experimental
Homebrew	Yes	Yes	Default	Yes
Kali			Current	

CHAPTER

THREE

TROUBLESHOOTING

3.1 Why do I get a `No GreatFET devices found!` error when my GreatFET is plugged in?

It is possible that your GreatFET has been connected to the USB1 side, which is where your target device should be plugged in. To fix this you will want to connect your GreatFET to your host device (such as your computer) via the USB0 side. There are USB0 and USB1 labels on the board.

CHAPTER

FOUR

USING GREATFET APIS

4.1 Getting a GreatFET object

A GreatFET object may be created with the following Python code:

```
import greatfet

gf = greatfet.GreatFET()
```

`gf shell` will do this automatically before spawning an IPython instance. `greatfet.GreatFET()` can also take the same keyword arguments as PyUSB's `usb.find()`, to allow specifying a device e.g. by serial number.

4.2 Using APIs

LibGreat provides the general comms API for talking to LibGreat devices.

Generically, APIs can be accessed in Python as `gf.apis.<api_name>`, e.g. `gf.apis.firmware`, for the firmware API. Alternatively, the same objects can be accessed by API names as strings via the `gf.comms.apis` dict.

On the command line, you can list the APIs supported by a GreatFET with `greatfet info -A`.

Some APIs are also additionally exposed as "interfaces" on a GreatFET object, e.g. `gf.<interface>`. These are primarily for convenience—they provide default configurations and simplified interfaces for their relevant APIs.

Some APIs also have command line tools as helpers. These will be subcommands of `greatfet` (which can be shortened to `gf`). Invoking `greatfet` without any arguments will list the currently supported subcommands.

4.3 Debugging

In the event that something does not goes as expected, you can run `greatfet dmesg` on the command line to get a log of events that have occured on the GreatFET. From Python, you can run `gf.read_debug_ring()` to get a string object for the same text.

The debug ring-buffer persists across soft resets (including e.g. firmware flashes), but *not* across hard-resets, like pressing the reset button or unplugging and replugging the device.

`gf.apis.debug.peek` and `gf.apis.debug.poke` can also be used to read from and write to raw memory addresses, for advanced debugging.

4.4 LED

LEDs are a simple place to start. GreatFET Azalea has 4 LEDs, though LED 1 blinks on and off periodically as a "heartbeat" by default. *Note that the LEDs are 1-indexed!*

There is a convenience interface in Python:

```
gf.leds[2].on()
gf.leds[2].off()
gf.leds[2].toggle()
```

As well as a command-line helper tool.

```
$ greatfet led --on 2
$ greatfet led --off 2
$ greatfet led -t 2
```

4.5 GPIO

The GPIO peripheral can be easily controlled from the convenience interface `gf.gpio`.

`gf.gpio.read_pin_state((1, 6))` will read the logic value for GPIO pin 1[6] (pin mappings for peripherals can be found here) Likewise `gf.gpio.set_pin_state((1, 6), 0)` can be used to set the same pin to logic low.

4.6 Logic Analyzer

The logic analysis functionality is easiest from the command-line helper. For example:

```
$ greatfet logic -p out.sr -f 2M -n 4
```

This will sample 4 channels at 2MSPS, and output a Sigrok-compatible file as `out.sr`.

The default pin mappings are as follows:

Channel	Pin
0	J1_P4
1	J1_P6
2	J1_P28
3	J1_P30
4	J2_P36
5	J2_P34
6	J2_P33
7	J1_P7
8	J1_P34
9	J2_P9
10	J1_P25
11	J1_P32
12	J1_P31
13	J2_P3
14	J1_P3
15	J1_P5

4.7 Pattern Generator

Pattern generation functionality can be done from the command-line helper. For example:

```
$ greatfet pattern counter -n 1K -w 8
```

This will generate an 8-bit counter at 1KHz.

The default pin mappings are the same as the mappings for logic analysis.

4.8 UART

This class can be used to talk "serial", and has both a command-line helper tool available and a convenience interface as `gf.uart`.

If you happen to have a USB-serial converter handy, then you can test it out by connecting TXD, RXD, and GND of the USB-serial converter to J1_P34, J1_P33, and any available ground pin (like J1_P1) respectively. Then you can simply run `greatfet uart`, and e.g. on Linux, `dterm /dev/ttyUSB0 115200`. Typing in either terminal will show the respective characters on the other.

Naturally, all of the same functionality can be used from Python via the `uart` interface:

```
data = gf.uart.read()
gf.uart.write(b"Hello, world!")
```

4.9 ADC

The analog to digital converter is easily usable from the command line helper tool. Simply running `greatfet adc` will print the voltage on ADC0 (mapped to J2_P5 by default).

In Python, there is an interface for the default ADC configuration as ADC0. To read a single sample:

```
gf.adc.read_samples(1)
```

The 10-bit digital to analog converter has a command line helper too, with `greatfet dac -S <value>`. Note that the value must be the number to set the DAC too, not a voltage. For example, `greatfet dac -S 512` will set the DAC to ~1.6 volts.

4.10 DAC

GreatFET's digital to analog converter is mapped to J2_P5. The API allows you to either set the raw value loaded into the DAC, or specify a target voltage (which is calculated as `value = (voltage * 1024) / 3.3`). Note however that the voltage must be specified in millivolts, e.g.: `gf.apis.dac.set_voltage(int(2.5 * 1000))` will set the DAC voltage to 2.5.

The command-line helper tool can take either the raw value or a voltage:

```
$ greatfet dac -f raw 776
$ greatfet dac 2.5
```

Both will set the DAC to ~2.5 volts.

4.11 SWRA124/Chipcon

This class is used to debug microcontrollers implementing the CC1110/CC2430/CC2510 debug interface described in SWRA124.

For simple dumping and flashing of firmware, it is easiest to use the command line utility:

```
$ greatfet chipcon --read firmware.bin --length 0x8000
$ greatfet chipcon --write firmware.bin
```

The same functionality, as well as more advanced functionality, can be accessed programmatically through the Python API:

```
cc = gf.create_programmer('chipcon')

# `debug_init` must be called before any debugging can happen.
cc.debug_init()

# Read the entire flash (for a 32k flash).
flash = cc.read_flash(start_address=0, length=32 * 1024)

# Reprogram the flash.
cc.program_flash(flash, erase=True, verify=True)
```

4.12 I²C

Class for communication over I²C buses. Can be used from Python, e.g. `gf.i2c.scan()`, or with the command-line helper, e.g. `gf i2c --scan`.

CHAPTER

FIVE

GREATFET_I2C

greatfet_i2c is a tool for talking to an I2C device with GreatFET.

5.1 Pin Usage

signal	symbol	pin
SDA	I2C0_SDA	J2.39
SCL	I2C0_SCL	J2.40

5.2 Example Usage

scan(GreatFET) - scans the GreatFET I2C Bus for connected I2C devices and displays responses from 7-bit addresses.

```
$ gf i2c -z
I2C address(es):
0x0 W
0x20 W
0x20 R
0x7c W

******** W/R bit set at each valid address ********
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: W- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: WR -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- -- -- -- -- -- W- -- -- --
```

read(GreatFET, address, receive_length, log_function) - reads a 'receive_length' amount of bytes over the I2C bus from a device with the specified 7-bit address. Optionally displays results.

write(GreatFET, address, data, log_function) - writes bytes over the I2C bus to a device with the specified 7-bit address. Optionally displays results.

```
$ gf i2c -a 0x20 -w 0x01 0x02 0x03 -r 2 -v
Trying to find a GreatFET device...
GreatFET One found. (Serial number: 000057cc67e630318c57)
Writing to address 0x20
I2C write status: 0x18
Bytes received from address 0x20:
0x3
0x3
I2C read status: 0x40
```

Transmits bytes '0x01', '0x02', and '0x03' to an I2C device with address 0x20 (Crocus in this case), receives 2 bytes in response, and displays the results.

5.3 API

- I2CDevice(GreatFET, address) - class representation of an I2C device that can send/receive data over the GreatFET I2C bus using the given address
- I2CBus(GreatFET) - class representation of a GreatFET I2C bus
- attach_device(I2CDevice) - attaches a given I2C device to this bus.
- read(address, receive_length) - reads data from the I2C bus.
- write(address, data) - writes data to the I2C bus.
- transmit(address, data, receive_length) - wrapper function for back to back TX/RX.
- scan() - TX/RX over the I2C bus, and receives ACK/NAK in response for valid/invalid addresses.

CHAPTER

SIX

GREATFET_LOGIC

greatfet_logic is a logic analyzer implementation for GreatFET. It uses the SGPIO peripheral in the LPC4330 to monitor the state of 8 pins and combines those into an 8-bit integer streamed to the USB host.

6.1 Pin Usage

signal	symbol	pin
SPGIO0	P0_0	J1.04
SGPIO1	P0_1	J1.06
SGPIO2	P1_15	J1.28
SGPIO3	P1_16	J1.30
SGPIO4	P6_3	J2.36
SGPIO5	P6_6	J2.34
SGPIO6	P2_2	J2.33
SGPIO7	P1_0	J1.07

CHAPTER

SEVEN

GREATFET_MSP430

greatfet_430 implements MSP430 JTAG functions.

7.1 Pin Usage

signal	symbol	pin
TDO	P1_3	J1.40
TDI	P1_4	J1.39
TMS	P1_20	J1.37
TCK	P3_4	J2.28
RST	P4_3	J2.09
TST	P4_2	J2.08

CHAPTER

EIGHT

GETTING STARTED WITH FIRMWARE DEVELOPMENT

Here's some quick info for people with GreatFETs who want to get started with firmware development.

8.1 Fresh install

To get started, you will need to install the dependencies:

```
sudo apt-get install gcc-arm-none-eabi libnewlib-arm-none-eabi cmake make dfu-util␣
↪python-setuptools python-yaml
pip install pyyaml
```

Acquire the code by cloning the repository:

```
git clone --recursive https://github.com/greatscottgadgets/greatfet.git
```

8.2 Updating your repository

You should already have the prerequisites installed as above. Now update the GreatFET repository that you have previously cloned:

```
cd greatfet
git pull
git submodule update
```

8.3 Installing host tools

The host tools are written in Python. To install them, run the following from the root of the cloned repository:

```
pushd libgreat/host/
python setup.py build
sudo python setup.py install
popd

pushd host/
python setup.py build
sudo python setup.py install
popd
```

8.4 Building and flashing firmware

The firmware currently constitutes two parts, `libopencm3` and `greatfet_usb`. We build the library first, then the firmware on top of it:

```
cd firmware/libopencm3
make
cd ../greatfet_usb
mkdir build
cd build
cmake ..
make
```

This will produce a file named `greatfet_usb.bin` which can be written to a GreatFET One using `greatfet_firmware -w greatfet_usb.bin`.

If you need to recover from an empty flash or non-functional firmware, you will need to use DFU to recover. Remember, if the firmware written to flash was non-functional, the DFU version will be too, you will need to return to a known good version to restore GreatFET.

To write the file, first hold the DFU button while resetting the board, `lsusb` will show a line such as `Bus 002 Device 007: ID 1fc9:000c NXP Semiconductors`, which is the NXP LPC4330 in DFU mode. You can write the firmware to the GreatFET One's RAM using `greatfet_firmware -V greatfet_usb.bin`. The firmware will run immediately. If you wish to run from ROM you then need to use `greatfet_firmware` as above.

8.5 Re-Building and flashing firmware

When rebuilding software the following is recommended

```
cd firmware/libopencm3
make clean
make
cd ../greatfet_usb
mkdir build
cd build
cmake ..
make clean
make
```

This will produce a file named `greatfet_usb.bin` which can be written to a GreatFET One using `greatfet_firmware -w greatfet_usb.bin`.

CHAPTER

NINE

LIBGREAT VERB SIGNATURES

`libgreat` verbs are designed to be self-describing: each verb provided by a libgreat device includes a small body of metadata that can be queried by the host:

```
{
    // The name of the verb. These are typically named like C function names.
    .name = "sum_and_difference",

    // The handler function for the given verb. This is the verb definition we
    // provided above.
    .handler = example_verb_sum_and_difference,

    // A short piece of documentation for the verb.
    .doc = "Computes the sum and difference of two ints.",

    // The signature for the verb's arguments. This roughly matches python's
    // struct.pack format; see the wiki documentation for more information.
    .in_signature = "<II",

    // The signature for the verb's return values. This roughly matches python's
    // struct.pack format; see the wiki documentation for more information.
    .out_signature = "<II",

    // The names of the arguments to the verb.
    .in_param_names = "a, b",

    // The names of the return values for the verb.
    .out_param_names = "sum, difference"
},
```

This meta-data can include machine-parseable descriptions of each verb's signatures in the form of a short, machine-readable string. These strings include a description of the arguments to the function (the in-signature) and of the function's multiple return values (the out-signature); and effectively describe the data formats sent to and from the device during execution of a verb.

Providing descriptions of these signatures is optional, but it's highly recommended that you do so whenever possible: the `libgreat` host library can use these signatures to generate code for you – making device communications transparent to your code!

9.1 Describing Verb Signatures

Verb signatures are provided in a format that's heavily inspired by Python's struct module – in fact, the formats are mostly identical. To provide some additional flexibility, we support a few additional format characters. The standard and added format characters are described below:

Table 1: Verb Signatures

Format	Bytes	C-Type	Python Type	libgreat parse function	libgreat response function
x *(1)*	1	none	none	comms_argument_read_buffer(trans, 1, NULL)	comms_argument_read_buffer(trans, 1, NULL)
c	1	char	string of length 1	comms_argument_parse_uint8_t	comms_response_add_uint8_t
b	1	int8_t	integer	comms_argument_parse_int8_t	comms_response_add_int8_t
B	1	uint8_t	integer	comms_argument_parse_uint8_t	comms_response_add_uint8_t
?	1	bool / _Bool	integer	comms_argument_parse_bool	comms_response_add_bool
h	2	int16_t	integer	comms_argument_parse_int16_t	comms_response_add_int16_t
H	2	uint16_t	integer	comms_argument_parse_uint16_t	comms_response_add_uint16_t
i	4	int32_t	integer	comms_argument_parse_int32_t	comms_response_add_int32_t
I	4	uint32_t	integer	comms_argument_parse_uint32_t	comms_response_add_uint32_t
l	4	int32_t	integer	comms_argument_parse_int32_t	comms_response_add_int32_t
L	4	uint32_t	integer	comms_argument_parse_uint32_t	comms_response_add_uint32_t
q	8	int64_t	integer	comms_argument_parse_int64_t	comms_response_add_int64_t
Q	8	uint64_t	integer	comms_argument_parse_uint64_t	comms_response_add_uint64_t
f	4	float	float	comms_argument_parse_float	comms_response_add_float
d	8	double	float	comms_argument_parse_double	comms_response_add_double
s *(2)(3)(6)*		char[]	string	comms_argument_read_buffer	comms_response_add_raw
p *(2)(6)*		char[]	string	comms_argument_parse_uint8_t / comms_argument_read_buffer	comms_response_add_uint8_t / comms_response_add_raw
S *(4)*		char[]	string	comms_argument_read_string	comms_response_add_string
X *(3)(5)(6)*		uint8_t	bytes of length 1	comms_argument_read_string	comms_response_add_string

note number	description
(1)	null padding byte; rarely used
(2)	see python docs
(3)	typically used with a numeric prefix
(4)	encodes a null-terminated string
(5)	encodes raw bytes; repeated elements are merged into a single entry
(6)	numeric prefixes behave differently; see below

`libgreat` data is always of standard size, and *always* little-endian. **Accordingly, every non-empty method signature must begin with a '<'.** Exceptions are made for methods that expect no arguments or return no values, which can provide an empty string.

9.2 Repeat Specifiers

Most types can be modified with a numeric *repeat specifier*; this acts the same as if the element were repeated multiple times. For example:

```
4I
```

is exactly equivalent to:

```
IIII
```

This matches the behavior of Python's pack and unpack. Unless denoted with note `(6)` in the table above, each type supports a repeat specifier.

`libgreat` adds one additional repeat specifier: a repeat specifier of `*` specifies that all a remaining data or arguments should be interpereted as instances of the provided type. Accordingly, a verb with an in-signature of `<*I` accepts any number of `uint32_t` arguments (including zero); a verb with an out-signature of `<II*B` would always return two 32-bit integers, followed by any number of single bytes.

9.3 Length Specifiers

A handful of format specifiers interpret numeric prefixes as *element lengths*, rather than repeat counts. These types interpret these specifiers as documented below:

type	interpretation
s	the specified element represents a string of N characters, where N is the length specifier
p	the specified element represents a pascal string of maximum length N, where N is the length specifier
X	the specified element represents a string of N bytes, where N is the length specifier

For the `s` and `X` specifiers, a length specifier of `*` indicates that the relevant string can be expected to take up all of the remaining data. Note that the format `S` does accepts a *repeat specifier* and **not** a *length specifier*, so the string `32S` denotes 32 null-terminated strings.

9.4 Element Groups

`libgreat`'s format strings add one additional feature: *format groups*. Format groups use parenthesis to create *groups of elements*, which are handled slightly differently:

- On the python side, each format group accepts a single tuple (or list) that should contain each of the parenthesized types. So, the group `<(IIB)` would expect a single tuple contianing three integers, which would be packed as two consecutive `uint32_ts` followed by a `uint8_t`.
- Each format group can accept a *repeat specifier*; so the string `<8(IB)` would denote eight pairs of one `uint32_t` and one `uint8_t`. A repeat specifier of `*` is also acceptable, which implies that the entire remainder of the arguments accepted or data parsed will consist of pairs of `uint32_t` and `uint8_t`.

9.5 Examples

It may help to consider an example RPC with the following meta-data:

```
{ .name = "sum_polar", .handler = example_verb_sum_polar, .in_signature = "<*(II)",
    .out_signature = "<II", .in_param_names = "magnitudes_and_angles", .out_param_names␣
↪= "sum_magnitude, sum_angle",
    .doc = "Sums together a collection of polar coordinates." },
```

The method's in-signature, `<*(II)`, demonstrates that the method expects any number of *two-element pairs*, which each contain a pair of integers. Accordingly, we might call it as follows:

```
# Assuming the RPC is available as gf.apis.example.sum_polar:
magnitude, angle = gf.apis.example.sum_polar((1, 2,), (3, 4),)
```

Each argument will be intepreted as a pair of 32-bit integers; so the resultant data stream will wind up looking like:

```
<uint32_t '1'><uint32_t '2'><uint32_t '3'><uint32_t '4'>
```

On the device side, we might read the data as follows:

```
static int example_verb_sum_polar(struct command_transaction *trans)
{
    uint32_t sum_magnitude = 0, sum_angle = 0;

    // While there's still data available in the string, grab vectors the data-stream.
    while (comms_argument_data_remaining(trans)) {

        // Read the next pair of vector components from the data stream...
        uint32_t magnitude = comms_argument_parse_uint32_t(trans);
        uint32_t angle = comms_argument_parse_uint32_t(trans);

        // ... do your math here.
        // <left as an exercise to the reader>
    }

    // Check to make sure we actually got all the pairs we tried to read.
    // If we didn't, this function will fail out!
    if (!comms_transaction_okay(trans)) {
        return EBADMSG;
    }

    // And respond with the relevant data.
    comms_response_add_uint32_t(trans, sum_magnitude);
    comms_response_add_uint32_t(trans, sum_angle);
    return 0;
}
```

In this case, we repeatedly call `comms_argument_parse_uint32_t` to capture each piece of the input stream; using `comms_argument_data_remaining` to check how much data is left.

9.6 Omitting Verb Signatures

In some cases, we may not exactly be able to describe our data format using the strings above; or we may not know the data format until run-time. In these cases, the verb signature can be replaced with the string `"*"`, which indicates that the signature is too complex to be handled automatically.

Using this signature allows us to be flexible, but comes at a significant cost: the host code can no longer automatically generate RPC methods for us. It becomes our responsibility to provide code on the host side for to interface with these verbs. Typically, this is accomplished using the `execute_raw_command` method of the `CommsBackend` class. See the on-line help for more documentation:

```
from pygreat.comms import CommsBackend
help(CommsBackend.execute_raw_command)
```

CHAPTER

TEN

GREATFET CLASSES

The new *GreatFET Communications Protocol* uses a simple Class/Verb scheme to organize protocol-independent commands. **Classes** organize groups of related functions, while **verbs** provide the functions themselves. This very much parallels the organization of GoodFET commands.

Examples of *classes*:

- SPI functionality
- FaceDancer ("GreatDancer") functionality
- Debug utilities
- GPIO

Examples of *verbs*:

- Configure the SPI bus to set e.g. polarity
- Read a USB1 status register (for consumption by the FaceDancer host)
- Return the contents of the device's debug ring
- Set a particular pin to a given value

Each class is identified by a unique **32-bit integer**. Each verb is identified by a *separately-namespaced* **32-bit integer**. To add a class to GreatFET, first reserve its number in the Class Registry below.

10.1 Class Registry

Class Number	Class Name	Description
0x0	core	core device identification functionality
0x1	firmware	verbs for working with/updating device firmware
0x10	debug	debug utilities
0x11	selftest	utilities for self-testing libgreat-based boards
0x100	example	example verbs meant to illustrate the comms protocol
0x101	spi_flash	verbs for programming SPI flashes
0x102	heartbeat	control the GreatFET's idle/"heartbeat" LED
0x103	gpio	raw control over the GreatFET's GPIO pins
0x104	greatdancer	remote control over the GreatFET's USB1 port in device mode, for e.g. FaceDancer
0x105	usbhost	remote control over the GreatFET's USB port in host mode, for e.g. FaceDancer

continues on next page

Table 1 – continued from previous page

Class Number	Class Name	Description
0x106	glitchkit c	control over the GlitchKit API, and control over simple triggers
0x107	glitchkit_usb	control over functionality intended to help with timing USB fault injection
0x108	i2c	communication as an I2C controller
0x109	spi	communication as SPI controller
0x10A	leds	control over a given board's LEDs
0x10B	jtag	functions for debugging over JTAG
0x10C	jtag_msp430	MSP430 specific JTAG functions
0x10D	logic_analyzer	allows one to use the GreatFET's SGPIO interface as a logic analyzer
0x10E	sdir	Functionality for Software Defined Infrared
0x10F	usbproxy	Firmware functionality supporting USBProxy
0x110	pattern_generator	allows one to use the GreatFET's SGPIO interface as a pattern generator
0x111	adc	analog to digital converter functionality
0x112	uart	functionality for talking 'serial'
0x113	usb_analysis	functionality for USB analysis e.g. with Rhododendron
0x114	swra124	debugging/programming for TI cc111x, cc243x, and cc251x
0x115	loadables	API for loading and running small firmware extensions
0x116	clock_gen	clock generation / control
0x117	benchmarking	measurement of GreatFET communications and functions
0x118	can	functionality for communication over CAN
0x119	SWD	functionality for communicating with ARM SWD interfaces

CHAPTER

ELEVEN

NEIGHBORS

11.1 already designed

- Begonia: a GIMME for GreatFET
- Crocus: nRF24L01+
- Daffodil: through-hole prototyping
- Scorzonera: interface for FLIR PM-series thermal cameras (AM7969 & RS232)
- Spot: a chonky indicator LED

11.2 some progress made

- Canna: basic 2-channel CAN neighbor
- Edelweiss: CC1310 (depending on progress, could consider newer cc1352 that supports more)
- Foxglove: advanced level shifting and probing (support for +/- 12v would be awesome…)
- Gladiolus: IR
- Heliopsis: googly eyes
- Hydrangea: NFC
- Indigo: ChipWhisperer 20-Pin Connector + Voltage Glitching
- Jasmine: Lithium Polymer battery pack/charger
- Kniphofia: iCE40 FPGA capable of remapping neighbor pins
- Lily: RF power detector
- Lucky Bamboo: quadruple 2.4 GHz ADF7242 transceiver for monitoring Bluetooth LE advertising channels
- Magnolia: TDR, dual ethernet (LAN tap)
- Narcissus: a jig for testing GreatFET One
- Orchid: Breakout board for common hardware hacking interfaces
- Peony: SDR based on AT86RF215IQ
- Quince: 2.4 GHz SDR using 1 bit ADC over SGPIO
- Rhododendron: Hi-Speed USB passive sniffer
- Stellaria: experimental random number generator

- Tulip: an awesome blinky neighbor
- Ursinia: Grove base

11.3 rough ideas

- RNG
- DAQ
- pulse generation (TTL)
- arbitrary waveform generation
- magstripe
- barcode
- dual cc1101
- cc1200 + cc2500 for rfcat-like functionality in Sub-GHz and 2.5 GHz
- cc1352 for WiFi, BLE 5, Zigbee, Thread, Wireless M-Bus, 802.15.4g, 6LoWPAN, KNX RF, Wi-SUN®, and 2FSK/4FSK proprietary protocols
- Fieldbus Neighbor with CAN, RS-232, TIA-422, TIA-485 (possibly support for two of these neighbors operating at the same time for MitM and other dual interface abilities)
- ODB-II ?
- DTMF
- NFC/RFID
- DPA
- JTAG (like bus blaster and/or Black Magic Probe) (could be satisfied by Foxglove)
- SX1257 (SDR, no FPGA needed)
- SD card
- clock generator (for multi-HackRF)
- ultrasound
- USB Type-C sniffer
- LCD
- nRF8001
- general comms: Ethernet, Bluetooth, Wi-Fi, etc.?
- one or more of the Cypress Wi-Fi and/or Bluetooth chips formerly owned by Broadcom
- KNX
- mmWave sensor/radar
- energy harvesting (buck/boost)
- USB Type-C power delivery, accessory mode, alternate mode breakout/sniff/hack
- weather station
- zero insertion force neighbor

CHAPTER

TWELVE

HOW TO DESIGN A NEIGHBOR

This guide will help you design a neighbor, an add-on board for GreatFET.

12.1 Use a Template

The easiest way to get started is to use a neighbor template with KiCad to edit the design. Clone the neighbor-template repository locally:

```
git clone https://github.com/greatfet-hardware/neighbor-template.git
```

Then in KiCad select File > New > Project from Template. In the Project Template Selector click the User Templates tab and then click the Browse button. In the file browser select the neighbor-template directory. Two GreatFET Neighbor icons should now appear in the Project Template Selector. Select either the two-layer template or the four-layer template depending on how many copper layers you would like your PCB to have and click OK.

Type a name for your project and have fun designing it! In the schematic editor you will see some components that have already been added for you. Some of these are required (see below), and others are optional elements that you may wish to delete. Pay attention to the various tips below while you work on your design.

If this is your first time using KiCad, you may wish to go through Getting to Blinky or one of the other KiCad tutorials before editing your neighbor.

12.2 Required Elements

Every neighbor should connect to both of the 2x20, 2.54mm pin headers (J1 and J2) on GreatFET. You can use female stackable headers mounted on the top of your neighbor, or you can use male headers mounted on the bottom if you do not want your neighbor to be stackable. For mechanical stability, use 2x20 headers (not headers with fewer pins) even if you don't need to use very many pins.

Our favorite header part is Samtec SSQ-120-23-G-D, but it is a bit expensive due to the gold plating and low insertion force features. Any female header from the Samtec SSQ series with 10 mm post length (e.g. SSQ-120-03-T-D) should be fine. To get the correct post length, look for a "3" in the last numeric position of the part number (SSQ-xxx-x3-x-x). An important feature of the SSQ series is that the posts have a square cross sections. We have experienced unreliable connections with lower cost headers that have flat, rectangular post cross sections.

Except for +5V, all pins on the neighbor interface are referenced to 3.3 V.

You may power your neighbor from the 3.3 V supply (VCC) provided by the GreatFET, but a stack of neighbors may draw a total of only 150 mA from this supply. If you need more current or a different voltage, use +5V on J2 and implement your own voltage regulation. As a general rule, if your neighbor draws more than 50 mA consider adding your own voltage regulation to ensure that other neighbors are able to draw from VCC.

12.3 Bonus Row

The bonus row of pins (J7) is not required. Design your neighbor without using bonus row pins if you can. However, you may wish to keep J7 as an unpopulated header anyway (see Orientation below).

12.4 Orientation

It is possible for a user to plug in a neighbor rotated 180 degrees. This could be very bad! In order to discourage such accidents, give the user visual cues that indicate the correct orientation.

The most obvious visual cue is the PCB shape. Use the template to match the shape of the GreatFET. If you need a different shape, try to maintain aspects of the GreatFET shape, especially the curved west edge, to help the user recognize the orientation. Alternatively you could indicate the curves with markings on the silkscreen layer of your neighbor.

Another cue is the position of the mounting holes. The west mounting holes are closer together than the east mounting holes. Your neighbor should have all four mounting holes if possible. Be careful not to place components or route traces through the keep-out area around each mounting hole.

A third cue is the bonus row (J7). Even if you don't need the bonus row, keep J7 as an unpopulated header if you can. This makes the orientation easy to see, and it also allows the user to install a stackable header in J7 if desired.

Last, orient any major text in the same way as the title text on GreatFET.

12.5 Neighbor Identification

It is neighborly to enable automated detection of your neighbor. We like I2C for this because any number of neighbors can be detected over the same pair of I2C pins without conflict. If you have a use for an I/O expander or other I2C device on your neighbor, configure it with a unique address and reserve that address on the *I2C Registry*. Document your I2C address in 7-bit format even if the datasheet for your I2C part uses 8-bit format.

If you don't have a use for any particular I2C function, consider using an I2C EEPROM for identification. We've added one to the template to make this easy. This will allow your neighbor to participate in our scheme for identifying multiple neighbors that all share an I2C address on their EEPROMs.

The EEPROM is optional; you can remove it if you don't want it. In most of our designs we have removed the EEPROM and instead have used another I2C device with a unique address. If you're not sure whether or not you want to keep the EEPROM in your design, we suggest that you do. You can always choose to not populate it.

12.6 Making Your Neighbor Extra-neighborly

An extra-neighborly neighbor is stackable and allows the user maximum flexibility and hackability. Even if you haven't thought of an application that would require stacking of your neighbor with other neighbors, install stackable headers if you can.

Unfortunately not all neighbors will be electrically compatible with each other even if they fit together mechanically, but there are some things you can do to maximize compatibility.

If you have an option to use I2C as the primary interface to your neighbor, use it and configure your neighbor with an I2C address that is unused by other neighbors. *I2C Registry* your I2C address on this wiki. If you can provide solderable jumpers or some other means for a user to reconfigure the I2C address on your neighbor, do so. This could

allow a user to solve unforeseen conflicts or to use multiple copies of your neighbor simultaneously. See Crocus for an example. Document your I2C address in 7-bit format even if the datasheet for your I2C part uses 8-bit format.

Solder jumpers can also be used to provide alternative pin selections for any pin. See Begonia for an example.

If your neighbor uses SPI as its primary interface, consider ways to avoid conflicts with your chip select pin. One way would be to use an I2C I/O expander for your chip select. See Crocus for an example.

Use no more power than you need. Stacked neighbors may only draw a total of 150 mA from the 3.3 V supply, so a power hungry neighbor may be incompatible with other neighbors. We like to add a voltage regulator to neighbors that can draw more than 50 mA. Also don't draw current when you don't need it. A great option is providing the ability to switch on and off a voltage regulator or load switch from an I2C I/O expander.

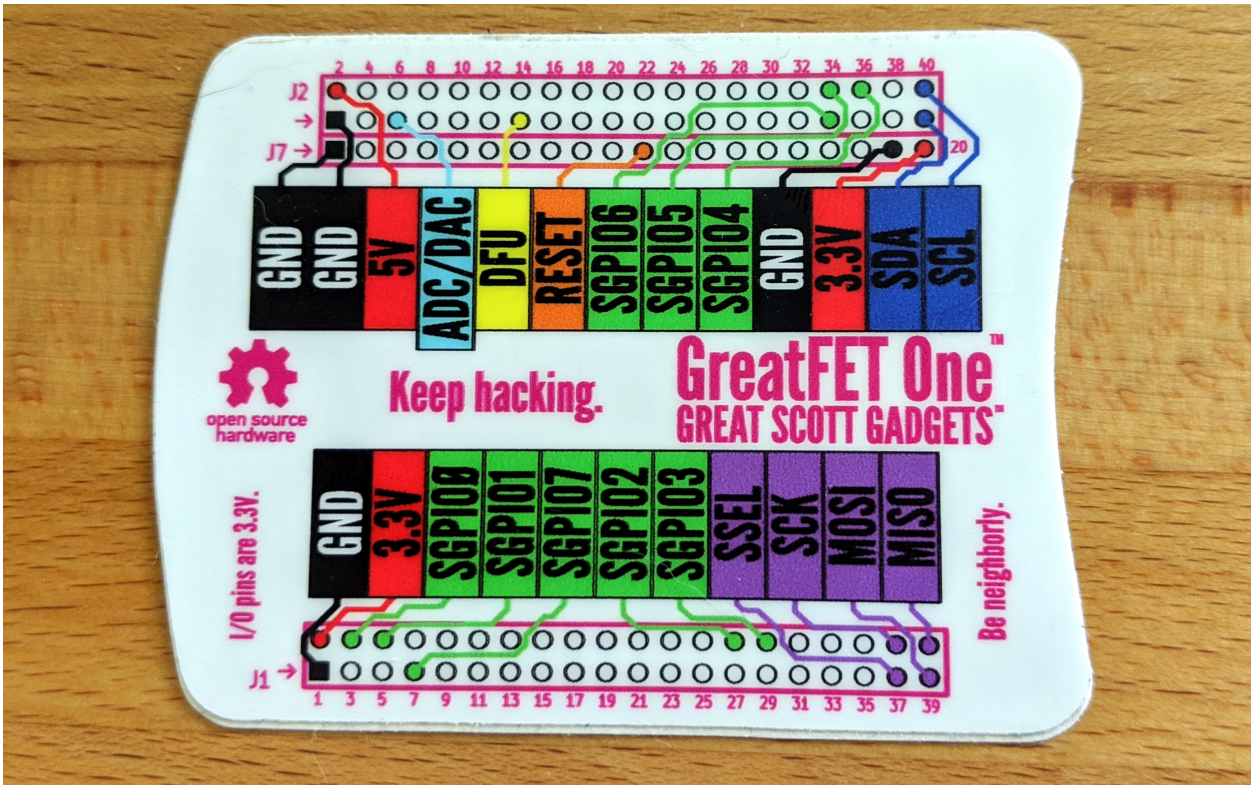
Provide a way to power down or disable your neighbor so that it does not draw power or unnecessarily load I/O pins while it is not in use. Two neighbors that are not compatible for simultaneous use may still be compatible if only one is enabled at a time by software.

Do not load analog pins when you don't need to. There is a very limited number of ADC/DAC pins, so conflicts are quite likely. It may make sense to use a small analog switch to enable or disable a connection to one of these pins. See Jasmine for an example. Another option is to provide your own ADC or DAC. This can be a good option if you expect your neighbor to use the analog function 100% of the time. See Gladiolus for an example of both.

Break out all your pins. If your neighbor has a component with unused pins, break them out to test points or unpopulated headers so that future people can experiment with them if desired.

12.7 Pin Selection

Choosing which pins (on J1, J2, and J7) to use for your neighbor can be a tricky process because there are so many options! These tips can help:



- If your neighbor uses a common function such as I2C or SPI that is labeled on the GreatFET One pinout sticker, use the pins suggested by the sticker.
  - exception: The default CS (SPI chip select) can be used by only one SPI peripheral at a time. It is labeled on the sticker so that people plugging in an external device have an easy-to-find default. You should use some other GPIO pin or a pin controlled by an I2C I/O expander for this function.
  - exception: The default ADC/DAC pin can be used by only one analog circuit at a time. It is similarly labeled for external use. Avoid using it for your neighbor unless you expect it to be used sparingly and you provide a way to switch off loading of that pin.
- Avoid using pins on the sticker for functions other than the use labeled on the sticker. For example, you could use J1 pin 40 for GPIO instead of for SPI CIPO, but doing so would be a poor practice because other users and neighbor designers will expect that pin to be available for its CIPO function.
- If your neighbor requires high speed parallel streaming with up to 16 pins, the SGPIO peripheral is probably your best choice. There are various SGPIO pins available, but we have defined a set of default pin assignments (shown on the sticker and used by greatfet logic) for 8-pin interfaces. Be aware that you will probably not be able to use SGPIO simultaneously with another neighbor using SGPIO.
- Use the pinout tables in the Azalea README file (hint: clone the repository and view the file locally) or this interactive table to find pins that have functions you require.
- If there is a neighbor that you specifically hope will be compatible with your neighbor, check its documentation and design files to find out what pins it uses.

CHAPTER

THIRTEEN

BOARD NAMING

13.1 Code Names

When we start working on a neighbor or other hardware design related to the GreatFET project, we assign a code name to the board. Our code names are names of flowers, and we usually choose a name starting with the next letter in the alphabet. We update the *list of neighbors* as soon as a name is selected.

13.2 GreatFET Trademark

"GreatFET" and "GreatFET One" are trademarks of Great Scott Gadgets. In order to make it easier for end users to know if their GreatFET-related devices are products of Great Scott Gadgets, we typically do not license these trademarks to others. If you create GreatFET-related products, please do not use "GreatFET" in the product name. If you would like to manufacture a clone of GreatFET One, for example, you could use the code name Azalea as the product name or as a part of the product name.

It is perfectly acceptable (and encouraged!) to use "GreatFET" in descriptive text such as "a GreatFET neighbor" or "compatible with GreatFET One" or "designed for GreatFET".

We try to make it easy to follow these guidelines by using code names and avoiding trademarks in our published hardware designs .

13.3 GreatFET and Great Scott Gadgets URLs

If you use one of our URLs such as https://greatscottgadgets.com/, https://greatscottgadgets.com/greatfet/, or https://github.com/greatscottgadgets/greatfet on a label of your own product, please clarify that it is our site, not yours. Here is a good way to do that:

> Designed for GreatFET: https://greatscottgadgets.com/greatfet/

An even better option is to use a URL pointing to your own informative web site (which links to ours where appropriate).

13.4 Naming Your Neighbor

If you create a neighbor or other GreatFET-related hardware design, we suggest that you follow our practice of using flowery code names, but you are welcome to choose whatever name you like. Just be careful of our trademark guidelines above. Please add your neighbor to the *list*!

CHAPTER

FOURTEEN

I2C REGISTRY

The following I2C addresses (in 7-bit format) are reserved by the I2C specification or by certain neighbors:

address	R/W	neighbor
0x00	W	reserved, general call
0x00	R	reserved, start byte
0x01		reserved, CBUS
0x02		reserved, different bus formats
0x03		reserved, future purposes
0x04-0x07		reserved, high speed controller code
0x10-0x11		Violet
0x18-0x1f		Operacake (HackRF add-on in optional neighbor mode)
0x27		Crocus
0x21		Foxglove
0x26-0x27		Narcissus
0x20		Jasmine
0x50		neighbor identification EEPROM
0x60		Gladiolus
0x78-0x7B		reserved, 10-bit peripheral addressing
0x7C-0x7F	R	reserved, device ID

Looking for known I2C addresses for things other than GreatFET neighbors? Check out Adafruit's handy list.

Confused about 7-bit vs. 8-bit I2C addresses? So is everyone! We try to use 7-bit addresses whenever possible because that is how they are specified. Total Phase has a nice article on the subject that we suggest reading to make sense of it all.

CHAPTER

FIFTEEN

RELEASE PROCESS

This is the process for tagging and publishing a release. Change the release version number and paths as appropriate. Release version numbers are in the form YYYY.MM.N where N is the release number for that month (usually 1).

15.1 tag the release

```
git tag -a v2013.07.1 -m 'release 2013.07.1'
git push --tags
```

15.2 make the release directory

```
cd /tmp
git clone ~/src/greatfet
cd greatfet
rm -rf .git*
mkdir firmware-bin
cd ..
mv greatfet greatfet-2013.07.1
```

15.3 copy/update RELEASENOTES from previous release

- prepend the current release notes to previous release notes

15.4 make second clone for firmware build

```
git clone --recursive ~/src/greatfet
cd greatfet/firmware/libopencm3
make
cd ..
```

15.5 update the firmware VERSION_STRING and compile firmware

```
sed -i 's/git-${VERSION}/2013.07.1/' cmake/greatfet-common.cmake
mkdir build
cd build
cmake ..
make
cp flash_stub/flash_stub.dfu /tmp/greatfet-2013.07.1/firmware-bin/
cp greatfet_usb/greatfet_usb.bin /tmp/greatfet-2013.07.1/firmware-bin/
```

15.6 make the release archives

```
tar -cJvf greatfet-2013.07.1.tar.xz greatfet-2013.07.1
zip -r greatfet-2013.07.1.zip greatfet-2013.07.1
```

15.7 “Draft a new release” on github

- call it “release 2013.07.1”
- paste release notes (just for this release, not previous)
- upload .tar.xz and .zip files

15.8 announce the release

- irc
- greatfet mailing list
- twitter