PFS Driver Board Firmware Documentation

Deliverable for Caltech

October 2, 2017

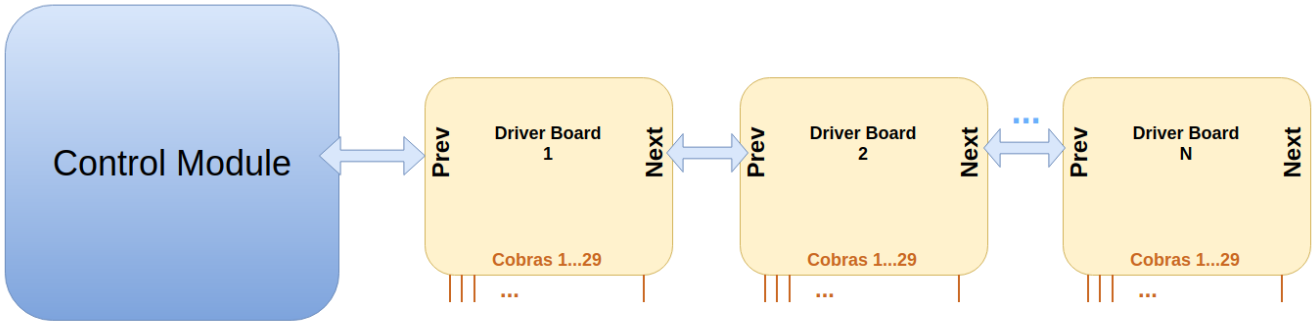
Prepared by: Keith Rumney

Jet Propulsion Laboratory
California Institute of Technology

Table of Contents

1. Firmware Overview

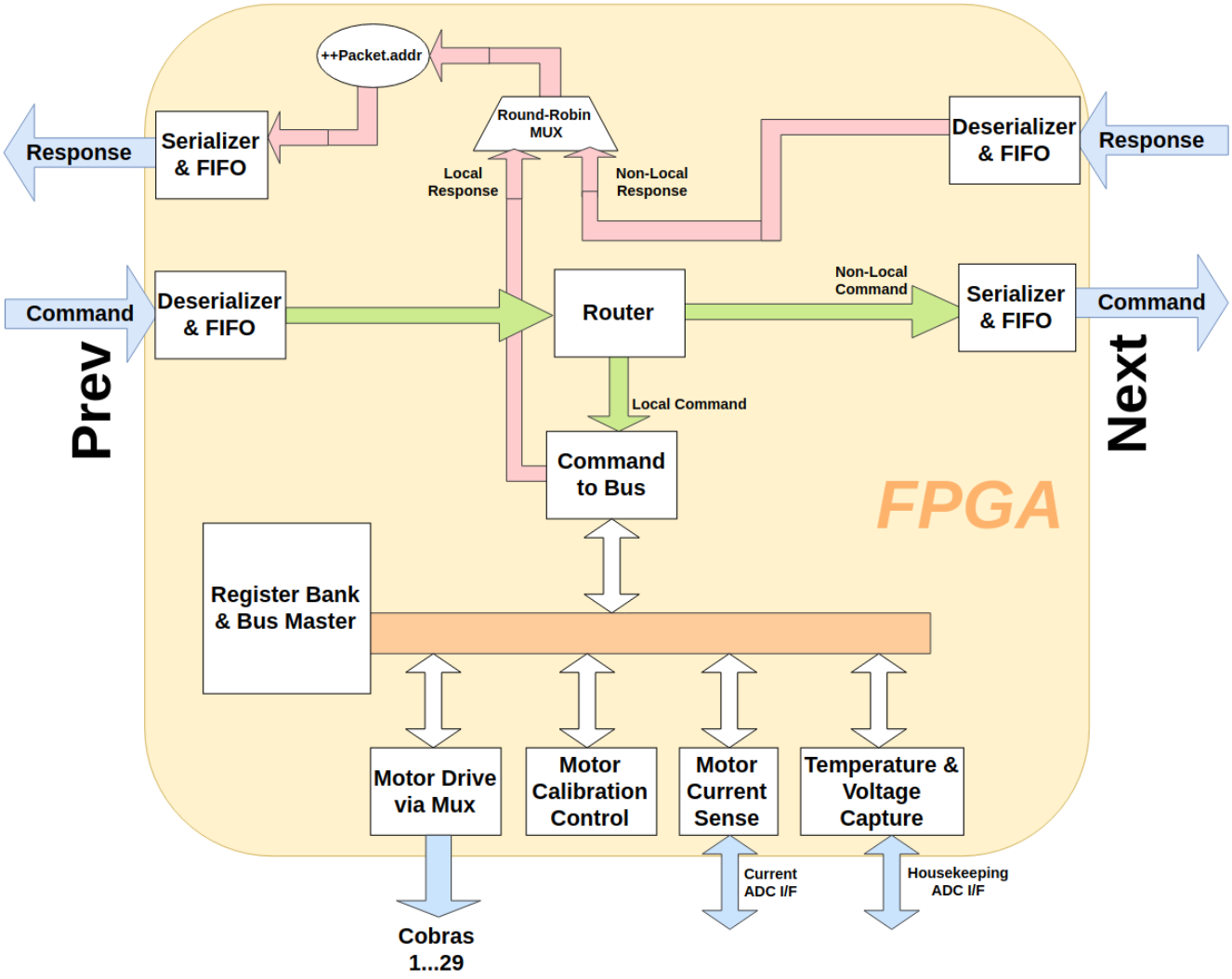
The driver board firmware exist primarily to control the 29 Cobras (piezo-electric motors) attached to the board. It contains two communication Flex connectors, named **Previous** and **Next**. The driver boards use these flex connectors to form a chained topology, where communication flows downstream from one board to the next. The first board in the chain is connected to a Control Module which generates all command sent down the chain. The FPGA used on the driver boards is the ProASIC3E A3PE3000 in a 484FBGA package.



A register bank exists that has entries for 64 motors (29 cobras * 2 motors per cobra = 58) so that each motor can have parameters related to drive frequency, peak current, and run-time settings stored. The 64-58=6 unused motor entries are useful to store other values such as board voltages, temperatures, and global settings that apply to all motors.

There are only drive signals for 8 cobras provided by the FPGA that go to four groups of cobras, group A (1..8), group B (9..16), group C (17..24), and group D (25..29). There are four separate enable signals for these groups that enable their motor drivers, and the firmware only ever enables one group at a time when driving motors. Within a group 2 adjacent cobras both large and small motors are driven together, while the others remain idle.

2. FPGA Block Diagram



3. Source Code File Structure

File tree:

```
.
├── constraints
│   ├── physical.pdc
│   ├── synp_timing.sdc
│   └── timing.sdc
├── designer
│   ├── drvr_board.adb
│   ├── drvr_board.pdb
│   └── pfs_top.edn
├── src
│   ├── ad7265.v
│   ├── adc_current_if.v
│   ├── adc_temps_if.v
│   ├── calibrator.v
│   ├── cal_norm.v
│   ├── clk_counter.v
│   ├── cmd_to_bus.v
│   ├── comm.v
│   ├── deserializer.v
│   ├── divider.v
│   ├── fifo.v
│   ├── mm_bus_read.v
│   ├── mm_bus_write.v
│   ├── mm_col_control.v
│   ├── mm_states.v
│   ├── mm_throttle.v
│   ├── motor_driver.v
│   ├── motor_mux.v
│   ├── pfs_bus_master.v
│   ├── pfs_bus.v
│   ├── pfs_pkgs.v
│   ├── pfs_top.v
│   ├── pfs.v
│   ├── roundrobin_mux.v
│   ├── router.v
│   ├── rst_deb.v
│   └── serializer.v
├── synthesis
│   └── pfs_driver_board.prj
└── validation
    ├── ad7265_tb
    │   ├── ad7265_sim.v
    │   ├── ad7265_tb.do
    │   └── ad7265_tb.v
    ├── adc_currents_if_tb
    │   ├── adc_current_if_tb.v
    │   └── adc_current_tb.do
    ├── adc_temp_if_tb
    │   ├── adc_temp_if_tb.v
    │   └── adc_temp_tb.do
    ├── calibrator_tb
    │   ├── calibrator_tb.do
    │   └── calibrator_tb.v
    ├── cal_norm_tb
    │   ├── cal_norm_tb.v
```

```
    │   └── tb.do
    ├── motor_mux_tb
    │   ├── motor_mux_tb.do
    │   └── motor_mux_tb.v
    └── utils.do

11 directories, 48 files
```

Constraints/

- physical.pdc – Physical Pin Constraints for FPGA
- synp_timing.sdc – Timing File used with Synplify for synthesis
- timing.sdc – Timing File used with Libero SoC for Place & Route

Designer/

- drvr_board.adb - Libero SoC Designer project file
- drvr_board.pdb - Driver Board FPGA Programming file
- pfs_top.edn - Netlist generated from Synplify Pro project (in synthesis/)

Src/

- *.v – System Verilog Source code explained in Section 4

Synthesis/

- pfs_driver_board.prj – Project file for Synplify Pro

Validation/

- utils.do – Tcl script used in each testbench
- *_tb – Various test-benches described in Section 6

4. Source Code Module Description

ad7265.v
The driver used to control the AD7265 ADC, takes a request from the user to read one of the analog inputs to the ADC and flags a ready signal when the conversion is done and the 12-bit data is available.

- 16Mhz clock and slows it down to a 4Mhz clock to be used for the ADC serial interface. The clock was slowed down due to significant delay on the data lines from the ADC at 16Mhz.
- There are two seperate MUXes in the ADC (A and B), both data channels are deserialized and the user is given both 12-bit values.
- Serial Data is driven by the ADC on the falling edge, and acquired on the rising edge at 4Mhz in this module.

adc_current_if.v
Controls the Current Sense ADC and hooks up to the internal bus. Works in tandem with calibrator.v

- uses ad7265.v for ADC driver
- Module waits for a motor to be running under calibration and then starts sampling ADC. x16 12-bit ADC values are accumulated into a 16-bit value that is provided for the calibrator.
- ADC is sampled for entire time motor is driven at a given frequency, and the latest possible 16-bit accumulated value is stored, since current peaks towards the end of a calibration motor drive.

adc_temps_if.v
Controls the Housekeeping ADC and hooks up to the internal bus. Regularly samples the temperatures and voltage on a timer and stores the values into the register bank using the bus.

- uses ad7265.v for ADC driver
- A timer goes off every ~262ms to sample the ADC, timer runs on a 1Mhz clock
- There are 3 different ADC inputs to the mux to sample, 2 for temperature and 1 for voltage. Only one input is sampled when the timer goes off so the effective sampling rate of each input is ~768ms.
- When the timer goes off the selected ADC input is sampled 16 times and accumulated into a 16-bit value, and then stored in the register bank.

calibrator.v
This module is trigged by the calibrate bit being set in the register bank, it is polling the bus to see this bit go high. This module collects all the settings relevant to calibrating motors and then loops through calibrating all 58 usable motors if calibration is enabled for that motor. Current is detected while sweeping the frequency for each motor to pick the frequency with peak current and store it as the motors operating frequency. This is explained in more detail in section 5-d.

- Loads the register in reg bank telling the adc_current_if the correct ADC input to collect current data from
- During peak current detection, currents at higher frequencies are normalized to be less than they would otherwise be. This is also what Newscale does during calibration, since higher frequencies naturally generate higher current, and can interfere with finding the local peak in

current due to the increasing slope in current with frequency. The cal_norm.v module applies this normalization.

cal_norm.v
Takes in an un-normalized current "curr" and normalizes it by returning "curr_norm = curr - (curr/per)" where per is the period of the motor frequency is 62.5ns steps. This normalization is effectively identical to new-scale's method.

- Uses divider.v
- This normalization is very subtle, consider a typical example where motor frequency is ~63Khz so that per=254 and measured current curr=45000. curr_norm = 45000 - 177 = 44823, the normalized current decreases about 0.39%. In another case where motor frequency is ~105Khz so that per=152, curr_norm = 45000 - 296 = 44704 or a 0.66% decrease.

clk_counter.v
Takes an input frequency parameter and output frequency parameter, and attempts to create an output clock that is of the desired output frequency via division. The ratio between input and output frequency should be an even integer for best results.

- Example: 16000Khz in and 300Khz out is accomplished by a counter that counts up to 27 cycles on the 16000Khz input domain clock and generates a 296Khz output.

cmd_to_bus.v
Takes incoming bus commands and buffers them in a FIFO, and requests bus access to run commands in the FIFO. If the bus command is a read is a response is provided as an output to the module once completed.

- Uses. fifo.v

comm.v + deserializer.v + serializer.v
Handles serial communication for off-board communication with one receive and one transmit channel (clock + data pair). Consist of a deserializer + FIFO for the receive channel, and a FIFO + serializer for the transmit channel.

- Uses fifo.v
- The serializer and deserializer both deal with packets that have the format of {Start bit, Data Payload, Checksum bit, Stop bit} this will be described in section 5-a

divider.v
Takes a 16-bit divisor and dividend and returns a 16-bit quotient (= dividend/divisor) valid once done goes high. The cycles the divider takes is variable.

fifo.v
Takes parameters for desired depth and data width, and cascades the FIFO4K18 Microsemi primitive to achieve this

motor_mux.v mm_bus_read.v mm_bus_write.v mm_col_control.v mm_states.v mm_throttle.v
Checks for the run bit to get set in the reg bank, at begins looping through cobras, running two adjacent cobras both large and small motors at the same time. The multiplexing and drive signals are described in section 5-d.

- The loop has to repeat until the specified number of steps has been run for each motor. After running each motor it's entry for number of steps to run in the register bank is updated, decrementing until it reaches 0.
- Also takes care of the "Sleeps" which are dead steps which must occur before a motor begins running it's steps.

motor_driver.v
Takes in settings for a given motor and starts driving it when commanded, for the time, frequency, and direction set by the settings. Drives the 4 motor drive signals (2 per phase) and flags ready when drive time is up.

- Charge control is implemented by delaying the low side of both phases by 812.5ns (13 clock cycles at 16Mhz)

pfs.v
Connects together mostly all of the design, this effectively implements the FPGA block diagram.

pfs_bus.v
This is the Bus interface, defining the different interfaces that a master and slave will have.

pfs_bus_master.v
The sole master on the bus, the gatekeeper to the register bank for the 64 (58 actual) motors described in section 5-b. This takes independent read/write requests from each of the slaves and arbitrates them, servicing in a round-robin fashion. Also creates the soft_reset pulse signal used by other modules.

pfs_pkgs.v
Holds parameters, typedefinitions, and functions for the design.

pfs_top.v
The top level block holding pfs.v, glue logic, and other I/O buffers.

- Includes clock generation of a 1Mhz clock and ~300Khz clock for PSU switching
- Power on Reset debounced for 20ms that is used for pfs.v
- Power on Reset debounced for 4usec that is used for 300Khz clock gen
- Renaming the motor drive signals

roundrobin_mux.v
Takes a parameter for MUXIN to specify number of inputs, this is used as a 2-input mux to arbitrate the telemetry/responses that need to be sent out on the response channel of the boards Previous interface. It is used to mux local and non-local responses.

router.v
Takes in a command and looks at it's target field (address) and if it is 0 accepts the command as local, otherwise declares it non-local and decrements the target field and requests the command be passed on via the boards Next interface.

rst_deb.v
Debounces a reset for the time specified in micro-seconds by the input parameter HOLD_TIME_US. This is used to hide glitches in an external reset signal.

5. Detailed Descriptions

5-a. Communication

Both commands and responses sent serially between driver boards have the same format. Within the packet is a field for board number which is really a hop-count.

Commands flow down the chain of boards with the hop-count being decremented and the command is valid for the board where hop-count is 0.

Responses flow up the chain with the hop-count incrementing and the control module determines which board was the source by the number of hops.

The packet is formated as follows:

Name	Start Bit	Payload	Parity Bit	Stop Bit
# Bits	1 bit	34 bits	1 bit	1 bit
Value	1	*(Explained below)*	Bitwise XOR of Payload	0

The Payload is as follows, It is a read/write command for the register bank of the driver board specified by the board number field. (Note this same format is used whether it is a command or response):

Name	Board Number	Rd/Wrn	Address	Data
# Bits	7 bits	1 bit	10 bits	16 bits
Value	Acts as the "Hop Count"	1 = Rd, 0 = Wr	Register Bank Address (described in section 5-b)	Filled with the write data on a write command, filled with the read data on a read response

5-b. Register Bank

The 10-bit Address for the register bank is as follows:

Name	Row	Column	Reg Number
# Bits	4 bits	2 bits	4 bits
Description	(explained below)	(explained below)	(explained below)

The row and column together form the 64 unique motor entries, with 16 registers available for each motor entry. The data-width of registers is 16-bits.

As described earlier there are 29 cobras and given it's two stages that forms 58 motors, the other 6 motor entries are pseudo-motors and are used for storing information such as run time, calibration parameters, ADC values, etc.
Below is the mapping of Row,Column pair to motors:

	Column 0	**Column 1**	**Column 2**	**Column 3**
Row 0	Cobra 1, Large	Cobra 9, Large	Cobra 17, Large	Cobra 25, Large
Row 1	Cobra 2, Large	Cobra 10, Large	Cobra 18, Large	Cobra 26, Large
Row 2	Cobra 1, Small	Cobra 9, Small	Cobra 17, Small	Cobra 25, Small
Row 3	Cobra 2, Small	Cobra 10, Small	Cobra 18, Small	Cobra 26, Small
Row 4	Cobra 3, Large	Cobra 11, Large	Cobra 19, Large	Cobra 27, Large
Row 5	Cobra 4, Large	Cobra 12, Large	Cobra 20, Large	Cobra 28, Large
Row 6	Cobra 3, Small	Cobra 11, Small	Cobra 19, Small	Cobra 27, Small
Row 7	Cobra 4, Small	Cobra 12, Small	Cobra 20, Small	Cobra 28, Small
Row 8	Cobra 5, Large	Cobra 13, Large	Cobra 21, Large	Cobra 29, Large
Row 9	Cobra 6, Large	Cobra 14, Large	Cobra 22, Large	-
Row 10	Cobra 5, Small	Cobra 13, Small	Cobra 21, Small	Cobra 29, Small
Row 11	Cobra 6, Small	Cobra 14, Small	Cobra 22, Small	-
Row 12	Cobra 7, Large	Cobra 15, Large	Cobra 23, Large	-
Row 13	Cobra 8, Large	Cobra 16, Large	Cobra 24, Large	-
Row 14	Cobra 7, Small	Cobra 15, Small	Cobra 23, Small	ADC
Row 15	Cobra 8, Small	Cobra 16, Small	Cobra 24, Small	Command

Normal motor entries have the following registers:

0	Pulse Period	Frequency defined as period via 62.5ns cycles
1	Pulse Length	duration of a single step runtime in micro-seconds
2	# Steps	Number of times to run
3	Min Period	Calibration minimum period
4	Max Period	Calibration maximum period
5	Period Step	Calibration period increment
6	Result Period	Calibration Resulting maximum period
7	Result ADC	16-bit peak current detected during calibration from ADC
8	# Sleeps	Number of times to sleep before starting the # steps

The **ADC pseudo-motor entry** has the following registers:

0	ADC0 Control	Used to store the 4-bit address specifying which ADC input to read for the adc_current_if
1	ADC0 Value	For debug, the last ADC Current captured is stored here
2	Temperature 0	Temperature 0 (temp 1 in schematic) read from the housekeeping ADC
3	Temperature 1	Temperature 1 (temp 2 in schematic) read from the housekeeping ADC
4	Voltage	Voltage read from the housekeeping ADC

The **command pseudo-motor entry** has the following registers:

0	Version	Holds Version Number (Currently 20)
1	Sandbox	Not Used
2	Control	Bit 0 can be set to start the motor mux, Bit 1 can be set to start a calibration, Bit 15 can be set to do a soft reset
3	Status	Not used
4	Motor Enable 0	A bit-mask specifying which motor to enable for all the motors in column 0 of the regbank. Bit 15 is motor at row 15, Bit 14 at row 14, ... Bit 0 at row 0.
5	Motor Enable 1	Same, but for column 1
6	Motor Enable 2	Same, but for column 2
7	Motor Enable 3	Same, but for column 3
8	Motor Direction 0	A bit-mask like Motor Enable 0, specifying the direction of motor rotation for all motors in column 0
9	Motor Direction 1	Same, but for column 1
10	Motor Direction 2	Same, but for column 2
11	Motor Direction 3	Same, but for column 3
12	Step Length	The minimum number of time (in micro-seconds) a step can last when it is being run on cobras, sometimes called "interleave". This is described in section 5-d.
13	Calibration Sleep	The pulse length to for all motors when running a calibrate

5-c. Calibration

Calibration requires certain registers to be setup by the user before starting, explained here.

The user first sets motor.min_period, motor.max_period, and motor.period_step for each motor. The Pulse length in calibration is set globally for all motors via command.calibration_sleep. The values for

command.motor_enable and command.motor_direction must also be set to specify which motors to calibrate and which direction to drive them in. Finally, by setting bit 1 of command.control the calibration sequence begins, implementing the following pseudo-code:

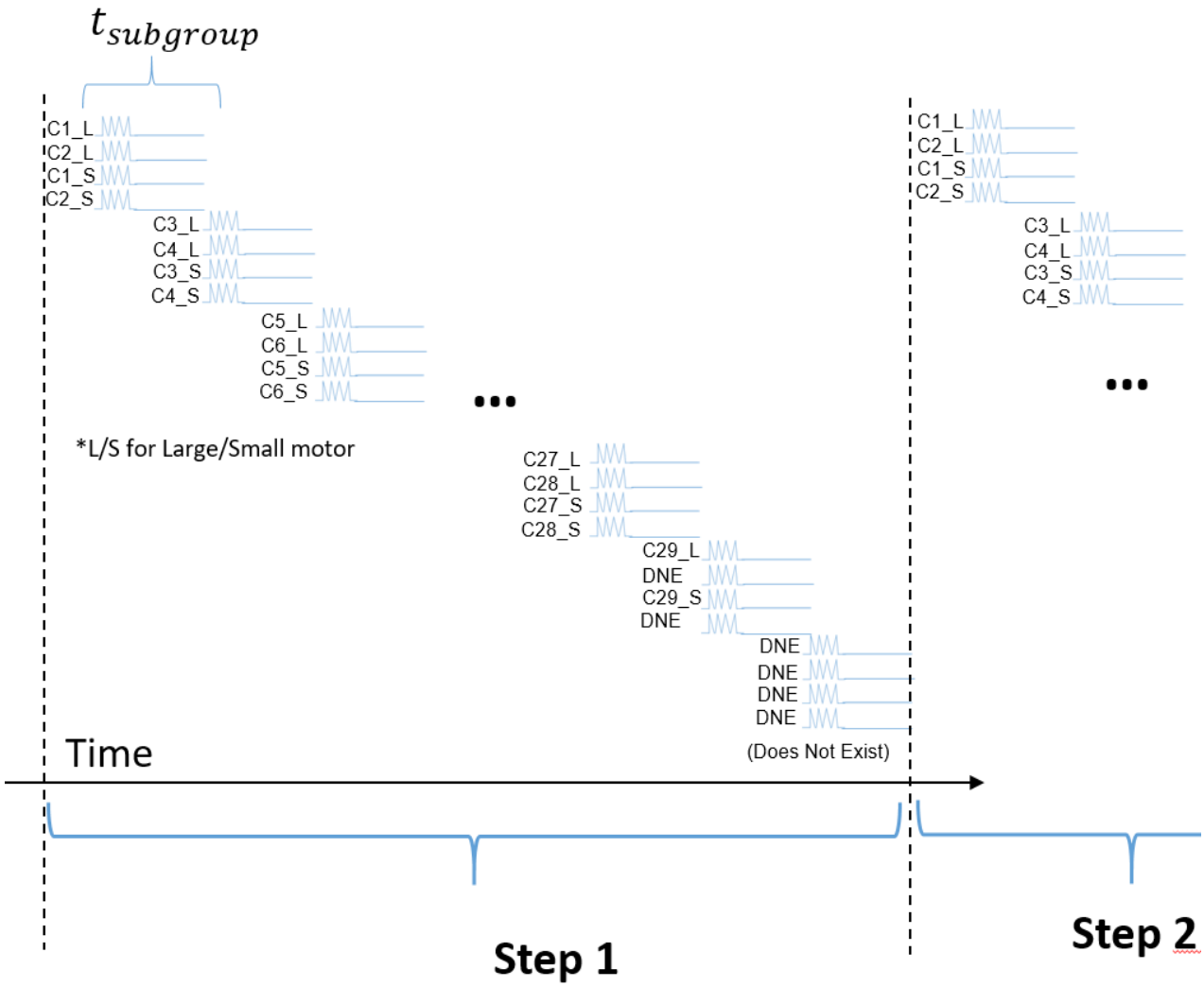
```
For Cobras in [1 … 29]:
   For motor in [3.4mm, 2.4mm]:
      If motor_enables[motor]:
         peak_current = 0
         peak_period = 0
        normalized_current = 0

        # Calibration loop for the selected motor
        For period in [motor.min_period … motor.max_period, motor.period_step]:
           (load params and set motor.pulse_period = period)
           (run the motor)
           (get the adc_val for current detected)
           if adc_val > normalized_current:
              peak_current = adc_val
              peak_period = period
             normalized_current = normalize(peak_current, peak_period)

        # after completing the calibration loop for the given motor:
        (Store motor.result_adc = peak_current, and motor.result_period,pulse_period = peak_period)
```

Some Notes:

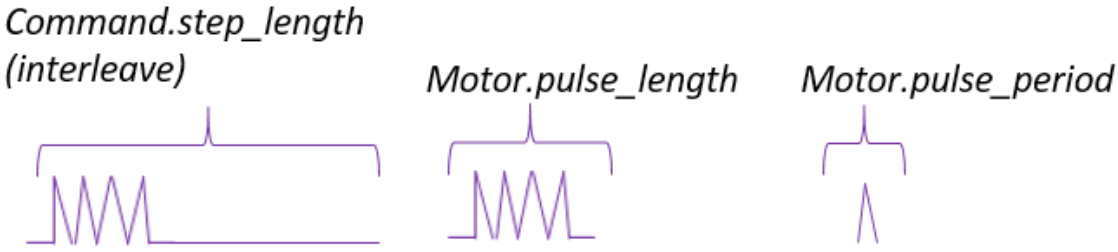
- This means that calibration sweeps through all 3.4mm cobras in order, followed by all 2.4mm cobras in order.
- Motor sweeps start at the smallest period which is the highest frequency, and ends at the lowest frequency via the highest period
- motor.period_step defines the motor period increment of the sweep, this is usually set to 1
- There is a normalization function normalize(current, period) which is implemented in cal_norm.v. It is used to normalize the current before it used the next for comparison, it slightly decreases the measured current based on frequency. Higher frequency currents are decreased more than lower frequency currents.
- An FPGA output controls a bypass current path to drive motors without power going through the current sense resistor, this bypass is disabled until calibration completes on all motors

5-d. Running Motors

When the run bit is set at command.control[0], the motor mux begins it's state machine. It is called a motor mux because there are only enough drive signals for 8 cobras provided by the FPGA, covering one of the four columns in the register bank. There is a 4 bit output for column enable, and only one column is enabled at a time. This method works since the firmware only ever drives 2 cobras at a time. A diagram of driving motors is shown on the next page:



- $t_{subgroup} = \max(stepLength, \max(motors.pulseLengths))$
- The effective $motor.pulseLength$ in $t_{subgroup}$ will be 0 if $motor.numSteps$ is 0 or if $motor.numSleeps$ is not 0



Some Notes:

- There is 16 sub-groups, and the time each sub group takes depends on the pulse lengths set for the motors in this group
- For each motor, if the **motor.num_sleeps is not 0** then this number must first decrement each step until it reaches 0. The motor **is NOT run** during this time. When **motor.num_sleeps is 0** then motor.num_steps decrements until it is also 0, the motor **IS run** during this time.
- Once all motors reach 0 sleeps and 0 steps left the motor mux completes and bit 0 of command.control clears

5-e. Conversions

ADC samples are stored in the regbank under ADC.temperature0, ADC.temperature1, ADC.voltage, and the current for all physical motors at motor.result_adc. These values can be converted to a voltage seen at the ADC, and from there to their respective units via the conversion table presented:

- $V_{ADC} = 2.5V * \left(\frac{x}{65535}\right)$

Value	Conversion from ADC Voltage
Temperature	$\frac{V_{ADC} * 1000}{5.99}\ °Kelvin$
10V Voltage	$\frac{V_{ADC} * 98.7}{16.2} Volts$
Motor Current	$\frac{V_{ADC}}{100 * 0.020} mAmps$

For each physical motor, motor.pulse_period specifies the motor frequency in terms of number of clock cycles, it can be converted to frequency as presented:

Value	
Motor Frequency	$\frac{16000}{motor.period} Khz$

6. Verification

All testbenches have their own subdirectory in validation/ and within their subdirectory is a x.do file which is to be run in questasim with the "do x.do" command. The x_tb.v file is the top level testbench file.

ad7265_tb/
Contains a model of the AD7265 ADC called "ad7265_sim.v". This testbench runs the low level ad7265.v file to verify it interacts properly with the model of the ADC.

adc_currents_if_tb/
Tests "adc_current_if.v" by hooking it up to the ADC model and also to the bus with regbank. This verifies current sampling from the correct ADC channel and storage into the regbank.

adc_temp_if_tb/
Tests "adc_temp_if.v" by hooking it up to the ADC model and also to the bus with regbank. This verifies the proper sampling of temperatures and voltages and their storage in the regbank.

calibrator_tb/
Tests "calibrator.v" by hooking up it up to the bus, regbank, adc_currents_if, and also motor_mux. Verifies that the calibrator can configure the motor mux properly to run one motor at a time, and then record the peak current and store it.

cal_norm_tb/
Tests the normalization performed during calibration, "cal_norm.v", for different motor periods (frequencies).

motor_mux_tb/
Tests "motor_mux.v" by hooking it up to a bus and regbank. Testbench manually sets up parameters for a motor entry in the regbank, and then sets the run-bit in command.control to start the motor_mux. The drive signals for the motor can then be verified on the waveform.

7. Tools used

Synthesis: Synplify Pro N-2017.09

Place and Route: Libero SoC v11.8 SP1

Simulation: Questasim 10.6C_1

Firmware Programming: Libero v11.8 Flash Pro

8. Firmware Parameters

FPGA: ProASIC3E A3PE3000 484FBGA

For the delivered firmware with Checksum 0xAC45:

- Usage: 23% Cells, 27% of Block-Ram
- Environment: -35-60ºC
- Least Slack under Max conditions: 21.68ns on 16Mhz clk_prv domain

9. Firmware Programming

To program the firmware a PC with Libero SoC V11.8 SP1 is needed and also a Microsemi FLASHPRO4 USB programmer plugged into the PC and the Driver Board.

1. With the driver board on, open the Libero SoC program FlashPro and create a new project stored in any location
2. In the table for Programmers look for the FLASHPRO4 to be detected, if it is not try hitting Refresh/Rescan for Programmers
3. Hit the Configure Device button in the center of the window and browse to the firmware file designer/drvr_board.pdb
4. There should then be a button titled "PROGRAM" to the right of the Configure Device button, click it and watch the programmer status in the table
5. The programming should take about 1-2 minutes and the programmer status should go through the states of ERASE, PROGRAM, and VERIFY. If the status turns to FAILED then verify proper connectivity of the programmer to the driver board, and power off the driver board for 30 seconds before repowering
6. Upon completion repower the board

To verify the firmware is loaded it can be plugged into a control module and a diagnostic command can be run with the control module to count boards verifying the board is accounted for.