Contents

1 Analog Device Architecture **3**

2 Digital to Analog Converter (`DAC` Block) **5**
- 2.1 Block Function 6
  - 2.1.1 Operating Ranges 6
  - 2.1.2 `AnalogLib` Implementation 6
- 2.2 Calibration 6
- 2.3 Profiling 7
- 2.4 Grendel API Hook 7
  - 2.4.1 Example Usage 8

3 Current Copier Block (`fanout` Block) **9**
- 3.1 Block Function 9
  - 3.1.1 Operating Ranges 9
- 3.2 Calibration 10
- 3.3 Profiling 11
- 3.4 Grendel API Hook 11
  - 3.4.1 Example Usage 11

4 Analog to Digital Converter (`ADC` Block) **13**
- 4.1 Block Function 13
  - 4.1.1 Operating Ranges 14
  - 4.1.2 `AnalogLib` Implementation 14
- 4.2 Calibration 14
- 4.3 Profiling 15
- 4.4 Grendel API Hook 15
  - 4.4.1 Example Usage 15

5 Lookup Table (`LUT` Block) **17**
- 5.1 Block Function 17
  - 5.1.1 `AnalogLib` Implementation 17
- 5.2 Grendel API Hook 18
  - 5.2.1 Example Usage 18

6 Analog Multiplier / Variable Gain Amplifier (`Mult` Block) **19**

7 Analog Integrator (`Integ` Block) 21
7.1 Block Function 21
7.1.1 Operating Ranges 22
7.1.2 `AnalogLib` Implementation 22
7.1.3 Calibration 22
7.1.4 `calibrateClosedLoop` Subroutine 23
7.1.5 `calibrateInitialCond` Subroutine 23
7.2 Profiling 23
7.3 Grendel API Hook 23
7.3.1 Example Usage 23

8 Fast DAC Methods 25
8.1 Fast Measurement 25
8.2 Fast Signal Creation 25

9 Appendix 27
9.1 Measurements 27
9.2 Calibration Utilities 27
9.3 Digitally Settable Code Types 27
9.3.1 Utility Functions 27

Chapter 1

Analog Device Architecture

Chapter 2

Digital to Analog Converter (DAC Block)

The digital to analog converter is a hybrid digital-analog block available on the HCDCv2 device [dacb]. Figures 4.1 and 2.2 presents the complete list of digitally settable codes in each DAC block. The digital to analog converter accepts one digital input that is either read from memory or a lookup table, and produces one analog output current.

Location: Each slice on the HCDCv2 device contains one DAC. The DAC may read values from the lookup tables resident on slice 0 (`source` is `DSRC_LUT0`) or slice 2 (`source` is `DSRC_LUT1`) of the same tile.

static and dynamic Codes: We describe the digitally settable codes resident on the block below:

- `enable`: Determines whether the block is enabled (true) or disabled (false).

- `range`: Configures the current range of the analog current emitted by the DAC. This range cannot be set to low mode (`RANGE_LOW`).

- `inv`: Determines whether the output of the DAC is inverted or not. This is useful if multiple DACs are reading from the same lookup table, as some DACs can by configured to negate the value of the lookup table.

- `source`: Determines where the DAC reads the digital value from. If the code is set to `DSRC_MEM`, the digital code stored in `const_code` is converted to an analog signal. If the code is set to `DSRC_LUT0` or `DSRC_LUT1`, the digital code emitted by the lookup table on slices 0 and 2 respectively is converted to an analog signal. If the DAC is configured to read from the lookup table, it is placed in free-running mode. This allows the DAC to handle dynamic digital values.

code	values
`enable`	`bool_t`
`inv`	`bool_t`
`range`	`range_t`
`source`	`dac_src_t`
`const_code`	256
`pmos`†	8
`nmos`	8
`gain_cal`	64

Figure 2.1: DAC Values [fu.]

code	type
`enable`	static
`inv`	static
`range`	static
`source`	static
`const_code`	dynamic
`pmos`†	hidden

2.1 Block Function

Given a DAC at location [chip,tile,slice], the behavior of the block is dictated by the relation presented below. At a high level, the DAC block converts a digital code to an analog current. The value returned by the function is the value of the current in μA. Any behavior not covered in the algorithm below is undefined:

if `enable` **then**
  if `source` = `DSRC_MEM` **then**
    $2 \cdot sign(\texttt{inv}) \cdot scale(\texttt{range}) \cdot (\texttt{const\_code} - 128) \cdot 128^{-1}$
  else if `source` = `DSRC_LUT0` **then**
    $2 \cdot sign(\texttt{inv}) \cdot scale(\texttt{range}) \cdot (\texttt{lut}[chip, tile, 0] - 128) \cdot 128^{-1}$
  else if `source` = `DSRC_LUT1` **then**
    $2 \cdot sign(\texttt{inv}) \cdot scale(\texttt{range}) \cdot (\texttt{lut}[chip, tile, 2] - 128) \cdot 128^{-1}$

The `inv` code determines whether the output signal should be inverted or not. The `range` code scales the output signal by 1x or 10x. Note that all static and dynamic codes are used in the block function.

2.1.1 Operating Ranges

The magnitude of the analog output is determined by the `range` code of the DAC. The `range` code is limited to medium (`RANGE_MED`) or high (`RANGE_HIGH`) mode:

$\texttt{out} \in scale(\texttt{range}) \cdot [-2\mu A, 2\mu A]$

2.1.2 `AnalogLib` Implementation

The `dac.h` file provides a `computeOutput` function that implements the block function presented above, given a set of dynamic and static codes. The returned value of this function is normalized (divided by $2\mu A$).

2.2 Calibration

The DAC block has three hidden codes:

- `gain_cal`: This code controls the gain of the DAC.
- `pmos` and `nmos`: These codes control the magnitude of the `gain_cal` code.

The `pmos` code is always set to 0. The remaining codes are set by the block's calibration routine. The DAC is calibrated using the following algorithm [daca]:

`table` = `make`()
for `nmos` in 0...7 **do**
  for `gain_cal` in 0...63 with stride 16 **do**
    loss = obj(`nmos`,`gain_cal`)
    `table` ← loss,(`nmos`,`gain_cal`)

```
for gain_cal in 0...63 do
    loss = obj(table.nmos,gain_cal)
    table ← loss,(nmos,gain_cal)
return table.nmos,table.gain_cal
```

At a high level, the calibration algorithm iterates over `nmos` and `gain_cal` codes and computes the loss for each combination of codes. The loss function is computed using the objective function, *obj*. Objective functions are evaluated over a collection of 4 test points unless specified otherwise (`const_code` that encodes 0.0,0.8,-0.8,0.5). The expected behavior is computed using the block function specified in Section 2.1. The DAC block supports three objective functions:

- `CALIB_MINIMIZE_ERROR`: This objective function minimizes the average error between the observed signal and the expected behavior.
- `CALIB_MAXIMIZE_DELTA_FIT`: This objective function minimizes the gain variance and magnitude bias of the block. The magnitude bias b is computed by measuring the signal for test point 0.0. For the nonzero test points, the gain is computed by taking the ratio of the observed to the expected value. The gain variance σ^2 is the computed by taking the variance over computed gains. The final returned loss is $min(\sigma, |b|)$.
- `CALIB_FAST`: This objective function minimizes the error for test point 1.0. This quickly calibrates the gain to have good gain characteristics.

Assumptions: XXX

2.3 Profiling

XXX profiling algorithm here XXX

2.4 Grendel API Hook

`Grendel` supports configuring DAC blocks using the `use_dac` command. We present the general formulation of the `use_dac` command below:

use_dac chip tile slice **src** dsrc **sgn** sign **val** value **rng** range

The `use_dac` command accepts a DAC location in the form of a chip, tile and slice index and several additional arguments (`dsrc`, `sign`, `value` and `range`) which are used set the static and dynamic codes in the block:

- `dsrc` argument: This argument sets the `source` static code for the block, and accepts `mem`, `lut0` and `lut1` as input values. These input values correspond to `DSRC_MEM`, `DSRC_LUT0` and `DSRC_LUT1` respectively.
- `sign` argument:This argument sets the `inv` static code for the block, and accepts `+` or `-` as values, where the `-` value sets the `inv` code to true.

- `val` argument: This argument indirectly sets the `const_code` dynamic code of the block, and accepts a floating point value between -1 and 1. The `val` argument is converted to a digital code using the following function:

$$min(value \cdot 128 + 128, 255)$$

- `rng`: This argument sets the `range` static code in the block, and accepts `m` and `h` as input values. The `m` input value corresponds to `RANGE_MED` and the `h` input value corresponds to `RANGE_HIGH`.

2.4.1 Example Usage

The following invocation configures the DAC on chip 1, tile 3, slice 0 to emit an analog signal of $10\mu A$:

```
use_dac 1 3 0 src mem sgn + val 0.5 rng h
```

The following invocation configures the DAC on chip 0, tile 0, slice 0 to emit an analog signal of $0.25\mu A$:

```
use_dac 0 0 0 src mem sgn + val 0.125 rng m
```

The following invocation configures the DAC on chip 0, tile 2, slice 1 to convert the output of the LUT at chip 0, tile 2, slice 2 to an analog signal. The resulting analog signal is scaled up by ten:

```
use_dac 0 2 1 src lut1 sgn + val 0.0 rng h
```

Chapter 3

Current Copier Block (`fanout` Block)

The current copier block is an analog block available on the HCDCv2 device [fan]. Figures 3.1 and 3.2 presents a complete summary of the digitally settable codes for the block. The current copier accepts one analog input (`in`) and produces three analog outputs (`out0`, `out1` and `out2`), where the analog outputs are copies of the provided signal.

Location: Each slice of the HCDCv2 device contains two current copier blocks. Given a slice at [chip,tile,slice], the two current copiers on the slice are written as [chip,tile,slice,0] and [chip,tile,slice,1].

code	values
`enable`	`bool_t`
`third`	`bool_t`
`range`	`range_t`
`inv[out0Id]`	`bool_t`
`inv[out1Id]`	`bool_t`
`inv[out2Id]`	`bool_t`
`nmos`	8
`pmos`†	8
`port_cal[out0Id]`	64
`port_cal[out1Id]`	64
`port_cal[out2Id]`	64

Figure 3.1: Fanout Values [fu.]

code	type
`enable`	static
`range`	static
`inv[out0]`	static
`inv[out1]`	static
`inv[out2]`	static
`nmos`	hidden
`pmos`†	hidden
`port_cal[out0]`	hidden
`port_cal[out1]`	hidden
`port_cal[out2]`	hidden

Figure 3.2: Fanout Code Types[fu.]

3.1 Block Function

The behavior of output i (`out`i) the current copier is dictated by the relation presented below. We write the analog input as *in* in the presented relation. The value returned by the function is the value of the current in μA. Any behavior not covered by this algorithm is undefined.

if `enable` **then**
  sign(`inv`[`out`i]) `in`

The `inv` code for the output i determines the whether the copied signal should be inverted or not. Note that all static codes, with the exception of the `range` and `third` codes, are used in the block function. The `range` code is used to configure the current limitations of the block, and the `third` code determines if the third output (`out2`) of the current copier is in use.

3.1.1 Operating Ranges

The magnitude of the analog input *in* must fall within the current limits of the current copier. These limits are determined by the `range` code:

`in` $\in scale(\texttt{range}) \cdot [-2\mu A, 2\mu A]$

`out0` $\in scale(\texttt{range}) \cdot [-2\mu A, 2\mu A]$
`out1` $\in scale(\texttt{range}) \cdot [-2\mu A, 2\mu A]$
`out2` $\in scale(\texttt{range}) \cdot [-2\mu A, 2\mu A]$

3.2 Calibration

The fanout block has five hidden codes:

- `port_cal[out0]`, `port_cal[out1]`, `port_cal[out2]`: These bias correction codes control the currents injected into the `out0`, `out1` and `out2` outputs of the current copier. These injected currents are used to correct for any unwanted biases in the block. A `port_cal` value of 32 approximately corresponds to an injected current of zero.

- `pmos` and `nmos`: These current reference codes affect the magnitude of the bias correction codes. They correspond to iref currents in the schematic of the block.

The `pmos` code is always set to 3 and the `nmos` code is always set to 0. The remaining codes are set by the block's calibration routine. The calibration routine implements the following algorithm:

```
ctbl = make_table()
tbl0 = make_table()
for cal0 in 0...64 with stride=4 do
    for cal1 in 0...64 with stride=4 do
        for cal2 in 0...64 with stride=4 do
            loss = obj(cal0,cal1,cal2)
            ctbl ← loss,(cal0,cal1,cal2)
tbl = make_table()
for i in -3..3 do
    cal0 = ctbl.cal0+i
    for j in -3..3 do
        cal1 = ctbl.cal1+j
        for k in -3..3 do
            cal2 = ctbl.cal2+k
            loss = obj(cal0,cal1,cal2)
            tbl ← loss,(cal0,cal1,cal2)
return tbl.cal0,tbl.cal1,tbl.cal2
```

At a high level, the calibration algorithm independently finds the best coarse-grained assignments for the `port_cal[out0Id]`, `port_cal[out1Id]` and `port_cal[out2Id]` bias correction codes. It then fine tunes the coarse-grained set of assignments by doing a local search in the vicinity of values around the coarse-grained set of assignments.

The objective function used by the fanout calibration routine XXX

3.3 Profiling

3.4 Grendel API Hook

`Grendel` supports configuring fanout blocks using the `use_fanout` command. We present the general formulation of the `use_fanout` command below:

use_fanout chip tile slice index **sgn** sign0 sign1 sign2 **rng** range **three**|**two**

The `use_fanout` command accepts a fanout location in the form of a chip, tile, slice and index and several additional arguments (`sign0`, `sign1`, `sign2`, `range`, `three`, and `two`) that set the static and dynamic codes of the block:

- `sign0`,`sign1` and `sign2` arguments: The sign arguments set the `inv` static codes for the first, second and third outputs respectively. These arguments accept `+` and `-` as values. If the sign argument is `-`, the corresponding `inv` code is set to true.
- `range` argument: The range argument sets the `range` static code of the block. This argument accepts `m` and `h` as values, where `m` corresponds to `RANGE_MED` and `h` corresponds to `RANGE_HIGH`.
- `three` or `two` argument: This argument determines if the third output (`out2`) is enabled. If this argument is set to `three`, the third output is enabled.

3.4.1 Example Usage

The following invocation configures the fanout on chip 0, tile 0, slice 0, index 0 to copy the input signal, where the input signal is within $[-2\mu A, 2\mu A]$. The first and second outputs are negated, and the third output is enabled.

use_fanout 0 0 0 0 **sgn** − − + **rng** m **three**

The following invocation configures the fanout on chip 1, tile 0, slice 2, index 1 to copy the input signal, where the input signal is within $[-20\mu A, 20\mu A]$. The second output is negated, and the third output is disabled.

use_fanout 1 0 2 1 **sgn** + − + **rng** h **two**

Chapter 4

Analog to Digital Converter (ADC Block)

The analog to digital converter is a hybrid digital-analog block available on the HCDCv2 device [?]. Figures **??** and 4.2 presents the complete list of digitally settable codes for each ADC block. The analog to digital converter accepts one analog input (`in`), and emits one digital output that is then read by a lookup table.

Location: Each even slice on the HCDCv2device contains one ADC. The ADC may write values to the lookup tables resident on slice 0 or slice 2.

static and dynamic Codes: We describe the digitally settable codes resident on the block below:

- `enable`: Determines whether the block is enabled (true) or disabled (false).
- `range`: Configures the current limit of the analog input. This code cannot be set to `RANGE_LOW`
- `test` codes: Places the block in various test modes. Currently unused, and therefore set to false.

code	values
`enable`	`bool_t`
`range`	`range_t`
`test_en`	`bool_t`
`test_adc`	`bool_t`
`test_i2v`	`bool_t`
`test_rs`	`bool_t`
`test_rsinc`	`bool_t`
`pmos`	8
`pmos2`	8
`nmos`	8
`i2v_cal`	64
`upper`	64
`lower`	64
`upper_fs`	4
`lower_fs`	4

Figure 4.1: ADC values [fu.]

code	values
`enable`	static
`range`	static
`test_en`	static
`test_adc`	static
`test_i2v`	static
`test_rs`	static
`test_rsinc`	static
`pmos`	hidden
`pmos2`	hidden
`nmos`	hidden
`i2v_cal`	hidden
`upper`	hidden
`lower`	hidden
`upper_fs`	hidden

4.1 Block Function

The behavior of block is dictated by the relation presented below. At a high level, the ADC converts an analog signal to a digital code. The value returned by the function is the digital value (0-255) emitted by the ADC. Any behavior not covered by the algorithm below is undefined:

if `enable` **then**

$min((scale(\texttt{range})^{-1} \cdot \texttt{in}) \cdot 128 + 128, 255)$

Since the `range` code affects the current limit accepted by the ADC, it also introduces and implicit gain into the function governing the behavior of the block. More concretely, if `range` code is set to `RANGE_HIGH`, the analog signal is scaled down by 10x (scaled by 0.1) before being converted to a digital signal.

4.1.1 Operating Ranges

The current range of the analog input `in` is determined by the `range` code of the ADC. The `range` code is limited to medium (`RANGE_MED`) or high (`RANGE_HIGH`) mode:

$\texttt{in} \in scale(\texttt{range}) \cdot [-2\mu A, 2\mu A]$

4.1.2 `AnalogLib` Implementation

The `adc.h` file provides a `computeOutput` function that implements the block function presented above, given a set of dynamic and static codes and an analog input value. The analog input value is provided to the function in the form of a normalized (divided by $2\mu A$), floating point value.

4.2 Calibration

The ADC block has eight hidden codes:

- `i2v_cal`: This code controls the gain of the internal current-to-voltage converter in the ADC.
- `upper_fs`,`lower_fs`,`upper` and `lower`: These codes affect how the analog signal is mapped a digital value.
- `pmos`, `pmos2` and `nmos`: These codes control the magnitude of the `i2v_cal` code.

The `pmos` and `pmos2` codes are always set to XXX and XXX respectively. The remaining six codes are set using the block's calibration routine. The ADC is calibrated using the following algorithm [**?**].

for fs in 0..3 **do**
  lowerFs = fs
  upperFs = fs
  for spread in 0..31 **do**
    for lsign in [-1,1] **do**
      for usign in [-1,1] **do**
        lower = $31 + spread \cdot lsign$
        upper = $31 + spread \cdot usign$
        for nmos in 0..7 **do**
          for i2v in 0..63 with stride=16 **do**
            score = obj(lowerFs,upperFs,lower,upper,nmos,i2v)
            tbl $\leftarrow$ score,(lowerFs,upperFs, lower,upper,nmos,i2v)

for i2v in 0..63 **do**
  score = obj(tbl.lowerFs,tbl.upperFs,tbl.lower,tbl.upper,tbl.nmos,i2v)
  table ← score,(tbl.lowerFs,tbl.upperFs.tbl.lower,tbl.upper,tbl.nmos,i2v)

return tbl.lowerFs,tbl.upperFs,tbl.lower,tbl.upper,tbl.nmos,tbl.i2v

At a high level, the calibration routine finds the best combination of `lowerFs`, `upperFs`, `lower`, `nmos` and `upper` values, and then finds the best `i2v_cal` code for the best combination of these codes. The algorithm doesn't exhaustively iterate over every `lower` and `upper` code value. The algorithm used to iterate over `lower` and `upper` codes is lifted from the original calibration routine implemented by the hardware designer.
Assumptions: XXX

4.3 Profiling

XXX

4.4 Grendel API Hook

`Grendel` supports configuring ADC blocks using the `use_adc` command. We present the general formulation of the `user_adc` command below:

use_adc chip tile slice **rng** range

The `use_adc` command accepts an ADC location in the form of a chip, tile and slice and a range argument (`range`) which is used to determine the range static code of the ADC. If the `range` argument is set to `m`, the `range` static code is set to `RANGE_MED`. If the `range` argument is set to `h`, the `range` static code is set to `RANGE_HIGH`.

4.4.1 Example Usage

This invocation configures the ADC on chip 0, tile 3, slice 2 to accept an analog signal in $[-2, 2]\mu A$.

use_adc 0 3 2 **rng** m

This invocation configures the ADC on chip 1, tile 0, slice 1 to accept an analog signal in $[-20, 20]\mu A$.

use_adc 1 0 1 **rng** h

Chapter 5

Lookup Table (LUT Block)

The digital to analog converter is a digital block available on the HCDCv2 device [?]. Figures 5.1 and 5.2 present the complete list of digitally settable codes in each LUT block. The digital to analog converter accepts one digital input and produces one digital output. The function the LUT block implements is set by the end-user.

Location: Each slice on the HCDCv2 device contains one LUT. The LUT may read values from ADCs resident on slice 0 (`source` is `LSRC_ADC0`) or slice 2 (`source` is `LSRC_ADC1`) on the same tile.

static and dynamic Codes: The lookup table accepts one static code, `source`, that determines which ADC the LUT should read from. If `source` is `LSRC_ADC0`, it reads from the ADC on slice 0 of the same tile. If `source` is `LSRC_ADC1`, it reads from the ADC on slice 2 of the same tile.

code	values
source	`lut_source_t`

Figure 5.1: LUT Values [fu.]

code	values
source	static

Figure 5.2: LUT Types [fu.]

5.1 Block Function

Given a LUT as location [chip,tile,slice], the behavior of the block is dictated by the function encoded in the lookup table. The function is implemented as a 256-value array (`MAP`), where each value is an 8-bit number. The index into the 256 value array is the input value, and the value at that cell is the output value. We formally describe the behavior of the block with the following relation:

if $\texttt{source} = \texttt{LSRC\_ADC0}$ **then**
$TABLE[\texttt{adc}[chip, tile, 0]]$
else if $\texttt{source} = \texttt{LSRC\_ADC1}$ **then**
$TABLE[\texttt{adc}[chip, tile, 2]]$

5.1.1 `AnalogLib` Implementation

This function is written to the LUT block by setting each input-output pair using the `setLut` function [?]. The `addr` argument is the digital code of the input, and the `data` argument is the digital code of the output.

5.2 Grendel API Hook

`Grendel` supports configuring the LUT block using `use_lut` and `write_lut` functions. The `use_lut` function sets the ADC the LUT reads from, and the `write_lut` function sets the one-input one-output function the LUT implements:

```
use_lut 0 3 0 src source
write_lut chip tile slice [input] function
```

Both commands accept a LUT location in the form of a chip, tile and slice. The `use_lut` command accepts a `source` argument that determines which ADC to read from. The `source` value may be either `adc0` or `adc1`. The `write_lut` command accepts an function and input argument. The function argument is a python expression and the input argument specifies the name of the input variable that appears in the function. The provided function will be executed on decimal values between [-1,1], and must produce values within the range [-1,1].

5.2.1 Example Usage

The following invocation configures the LUT at chip 0, tile 3, slice 0 to read digital values from the ADC at chip 0, tile 3, slice 2. The LUT is then configured to implement the function $0.5sgn(in) \cdot \sqrt{in}$.

```
use_lut 0 3 0 src adc1
write_lut 0 3 0 [in_] ((0.5)*math.copysign(1,in_)*(math.sqrt(abs(in_))))
```

The following invocation configures the LUT at chip 0, tile 3, slice 2 to read digital values from the ADC at chip 0, tile 3, slice 0. The LUT is then configured to implement the function $sin(in^2)$.

```
use_lut 0 3 2 src adc0
write_lut 0 3 2 [in_] (sin(in_*in_))
```

Chapter 6

Analog Multiplier / Variable Gain Amplifier (Mult Block)

code	values
enable	bool_t
vga	bool_t
inv	bool_t
range[in0]	range_t
range[in1]	range_t
range[out]	range_t
gain_code	256
pmos	8
nmos	8
gain_cal	64
port_cal[in0]	64
port_cal[in1]	64
port_cal[out]	64

Figure 6.1: Integrator Values [fu.]

Chapter 7

Analog Integrator (`Integ` Block)

The analog integrator is an analog block available on the HCDCv2 device [?]. Figures 7.1 and 7.2 present the complete list of digitally settable codes in the integrator block. The analog integrator accepts one analog input current (`in`) and produces one analog output current (`out`).
Location: Each slice on the HCDCv2 device contains one integrator.
static and dynamic Codes: We describe the digitally settable codes resident on the block below:

- `enable`: Determines whether the block is enabled (true) or disabled (false).
- `inv`: Determines whether the output of the integrator is inverted or not.
- `range[in]`: Configures the current range of the input current accepted by the integrator.
- `range[out]`: Configures the current range of the output current produced by the integrator. If this code is set to `RANGE_HIGH` or `RANGE_LOW`, an implicit gain is introduced into the block function. Refer to the description of `range[in]` for a detailed description of how the code values map to current limits.
- `ic_code`: Configures the initial condition of the integrator.

7.1 Block Function

The behavior of the block is dictated by the relation presented below. At a high level, the integrator integrates the input signal. The value returned by the function is the value of the current in μA. Any behavior not covered by the algorithm is undefined:

code	values
`enable`	`bool_t`
`exception`	`bool_t`
`inv`	`bool_t`
`range[in]`	`range_t`
`range[out]`	`range_t`
`ic_code`	256
`pmos`	8
`nmos`	8
`gain_cal`	64
`port_cal[in]`	64
`port_cal[out]`	64

Figure 7.1: Integrator Values [fu.]

code	type
`enable`	static
`exception`	static
`inv`	static
`range[in]`	static
`range[out]`	static
`ic_code`	dynamic
`pmos`	hidden
`nmos`	hidden
`gain_cal`	64
`port_cal[in]`	64
`port_cal[out]`	64

Figure 7.2: Integrator Types [fu.]

if `enable` **then**
$\texttt{out} = \int sign(\texttt{inv}) \cdot scale(\texttt{range[out]}) \cdot scale(\texttt{range[in]})^{-1}\omega \cdot \texttt{in}$
$\texttt{out}(0) = 2 \cdot sign(\texttt{inv}) \cdot scale(\texttt{range[out]})(\texttt{ic\_code} - 128) \cdot 128.0^{-1}$

Time Constant: The time constant is of the integrator is $scale(\texttt{range[out]} \cdot scale(\texttt{range[in]})^{-1} \cdot \omega)$. The nominal time constant ω is 126000.

7.1.1 Operating Ranges

The current range of the analog input and output of the integrator is determined by the value of the `range` code. The equations below describe the current range of the input and output:

$\texttt{in} \in scale(\texttt{range[in]}) \cdot [-2\mu A, 2\mu A]$
$\texttt{out} \in scale(\texttt{range[out]}) \cdot [-2\mu A, 2\mu A]$

7.1.2 `AnalogLib` Implementation

The `int.h` file provides convenience functions that implement the block function described above:

- `computeOutput`: Given a set of static and dynamic codes and an analog input value, this function returns the expected analog output in μA. This function assumes that the current state of the integrator is 0.
- `computeTimeConstant`: Given a set of static and dynamic codes, this function returns the expected time constant.
- `computeInitCond`: Given a set of static and dynamic codes, this function returns the expected initial value of the analog output in μA.

7.1.3 Calibration

The integrator has five hidden codes:

- `port_cal[in]`: This code injects a bias correction current into the analog input of the integrator.
- `port_cal[out]`: This code injects a bias correction current into the analog output of the integrator.
- `gain_cal`: This code controls the gain of the initial condition.
- `pmos` and `nmos`: These codes control the magnitude of the `gain_cal` code.

The `pmos` code is always set of XXX. The remaining four codes are set using the block's calibration routine. The calibration algorithm is broken up into two subroutines:

- **calibrateClosedLoop**: This routine configures the device to implement the idiomatic circuit presented in Figure **??**. For each `nmos` code, this algorithm finds the set of `port_cal[in]` and `port_cal[out]` assignments that brings the output of this circuit closest to zero.

- **calibrateInitialCond**: This routine finds the best `nmos` and `gain_cal` code that minimizes the loss of the objective function *obj*. This algorithm works with the set of `port_cal[in]` and `port_cal[out]` assignments computed by the `closedLoop` routine.

7.1.4 `calibrateClosedLoop` Subroutine

```
bias0 = measure out0 of fan
bias1 = measure out1 of fan
bias= bias0+bias1
for nmos in 0..7 do
    port_cal[out] ← 32
    gain_cal ← 32
    tbls[nmos] = make()
    for calIn in 0..63 do
        port_cal[in] ← calIn
        obs = measure out of integ
        loss = abs(obs-bias)
        tbl ← loss,(calIn,32)
for nmos in 0..7 do
    port_cal[in] ← tbls[nmos].calIn
    for calOut in 0..63 do
        port_cal[out] ← calOut
        obs = measure out of integ
        loss = abs(obs-bias)
        tbl ← loss,(calIn,calOut)
return tbls
```

7.1.5 `calibrateInitialCond` Subroutine

7.2 Profiling

7.3 Grendel API Hook

7.3.1 Example Usage

Chapter 8

Fast DAC Methods

8.1 Fast Measurement

8.2 Fast Signal Creation

Chapter 9

Appendix

9.1 Measurements

XXX describe how I2V XXX

9.2 Calibration Utilities

calibration table [?]: XXX

9.3 Digitally Settable Code Types

Figure ?? presents a summary of the types of digitally settable codes. All other digitally settable codes are unsigned integers.

`bool_t`	`true`
	`false`
`range_t`	`RANGE_MED`
	`RANGE_HIGH`
	`RANGE_LOW`

9.3.1 Utility Functions

sign(`bool_t` inverted): The sign function returns a constant coefficient of -1 if inverted is set, otherwise it returns a coefficient of 1. Refer to `sign_to_coeff` in `util.h` [uti].
scale(`range_t` range): The range function a constant coefficient that corresponds to the selected current range. The function returns 1.0 if the range is `RANGE_MED`, 10.0 if the range is `RANGE_HIGH` and 0.1 if the range is `RANGE_LOW`. Refer to `range_to_coeff` in `util.h` [uti].

Bibliography

[daca] `lab_bench/lib/AnalogLib/dac_calib.cpp`.

Dac calibration routine.

[dacb] `lab_bench/lib/AnalogLib/dac.h`.

Header file containing DAC class definition.

[fan] `lab_bench/lib/AnalogLib/fan.h`.

Header file containing Fanout class definition.

[fu.] `lab_bench/lib/AnalogLib/fu.h`.

Header file containing major type definitions for analog firmware.

[uti] `lab_bench/lib/AnalogLib/util.h`.

Header and source files providing utility functions.