MCC HAT Library Documentation

Release 0.2.0

Measurement Computing

Jun 14, 2018

CONTENTS

CHAPTER

ONE

HARDWARE OVERVIEW

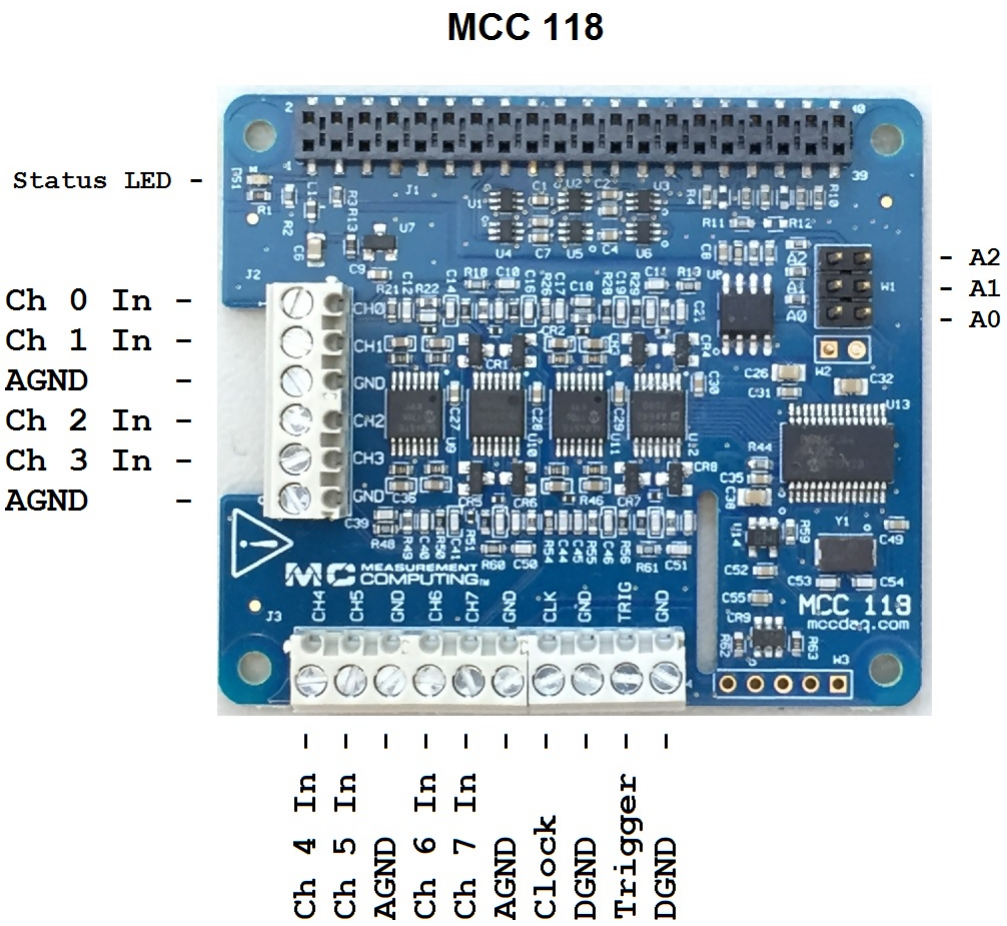
The MCC HATs are Raspberry Pi add-on boards (Hardware Attached on Top). They adhere to the Raspberry Pi HAT specification, but also extend it to allow stacking up to 8 MCC boards on a single Raspberry Pi.

We provide Python and C libraries, documentation, and examples to allow you to develop your own applications using our boards.

1.1 MCC 118

The MCC 118 is an 8-channel analog voltage input board with the following features:

- 12-bit, 100 kS/s A/D converter
- ± 10 V single-ended analog inputs
- Factory calibration with ± 20.8 mV input accuracy
- Hardware sample I/O clock
- Onboard sample buffers
- Digital trigger input



1.1.1 Board components

1.1.1.1 Screw terminals

- **CH 0 In** to **CH 7 In**: Single-ended analog input terminals.
- **Clock**: Bidirectional terminal for pacer I/O. Set the direction with software. Set for input to pace operations with an external clock signal, or output to pace operations with the internal sample clock.
- **Trigger**: External digital trigger input terminal. The trigger mode is software configurable for edge or level sensitive, rising or falling edge, high or low level.
- **AGND**: Common ground for the analog input terminals.
- **DGND**: Common ground for the clock and trigger terminals.

1.1.1.2 Address jumpers

- **A0** to **A2**: Used to identify each HAT when multiple boards are connected. The first HAT connected to the Raspberry Pi must be at address 0 (no jumper). Install a jumper on each additional connected board. Refer to the *Installing multiple boards* discussion for more information about the recommended addressing method.

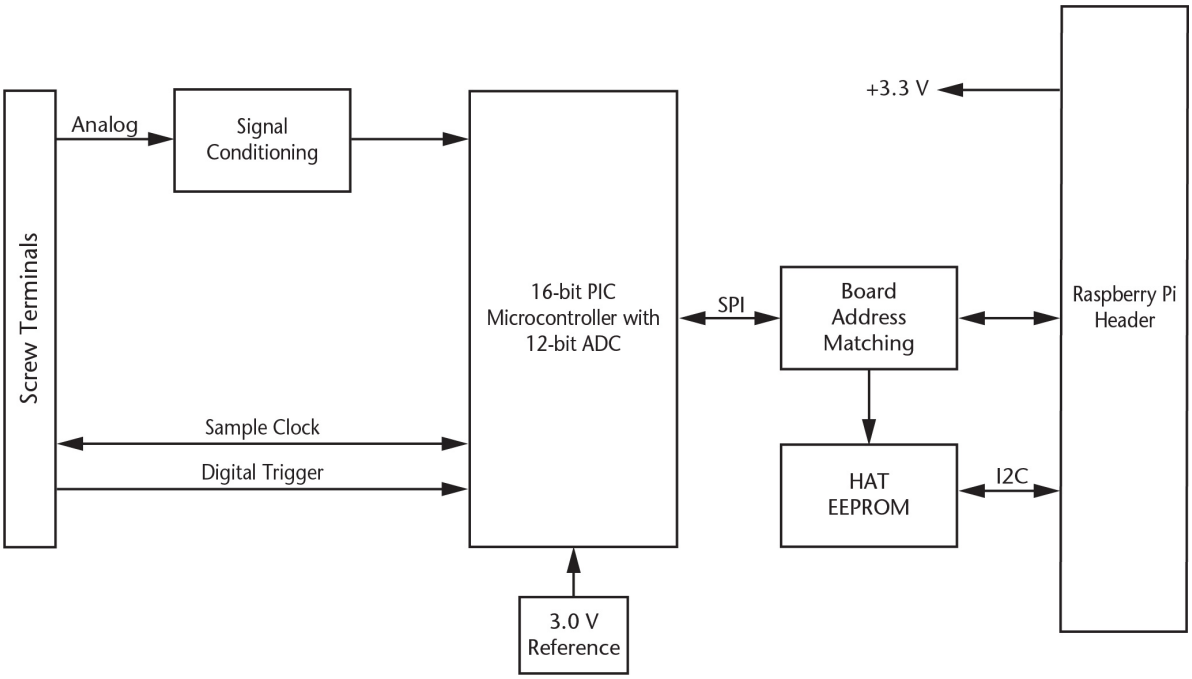
1.1.1.3 Status LED

The LED turns on when the board is connected to a Raspberry Pi with external power applied. You can flash the LED with software.

1.1.1.4 Header connector

The board header is used to connect with the Raspberry Pi. Refer to *Installing the HAT board* for more information about the header connector.

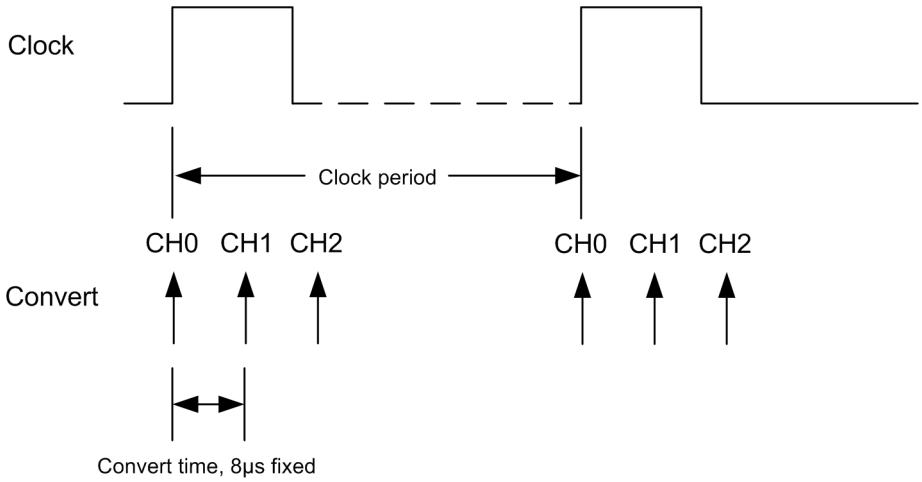
1.1.2 Functional block diagram



1.1.3 Functional details

1.1.3.1 Clock mode

The clock input / output on the MCC 118 is used to output the internal sample clock or apply an external sample clock to the device. Each pulse on the clock initiates a series of conversions of the selected channels in the scan. For example, when scanning channels 0, 1, and 2 the conversion activity will be:



1.1.4 Specifications

All specifications are subject to change without notice.
Typical for 25 °C unless otherwise specified.
Specifications in *italic* text are guaranteed by design.

Analog input

Table 1. General analog input specifications

Parameter	Conditions	Specification
A/D converter type		Successive approximation
ADC resolution		12 bits
Number of channels		8 single-ended
Input voltage range		±10 V
Absolute maximum input voltage	*CHx relative to AGND*	• *±25 V max (power on)* • *±25 V max (power off)*
Input impedance		• *1 MΩ (power on)* • *1 MΩ (power off)*
Input bias current	*10 V input*	*–12 µA*
	0 V input	*2 µA*
	–10 V input	*12 µA*
Monotonicity		*Guaranteed*
Input bandwidth	Small signal (–3 dB)	150 kHz
Maximum working voltage	Input range relative to AGND	±10.1 V max
Crosstalk	Adjacent channels, DC to 10 kHz	–75 dB
Input coupling		DC
Recommended warm-up time		1 minute min
Sampling rate, hardware paced	Internal pacer	0.004 S/s to 100 kS/s, software-selectable
	External pacer	100 kS/s max
Sampling mode		One A/D conversion for each configured channel per clock
Conversion time	Per channel	8 µs
Sample clock source		• Internal sample clock • External sample clock input on terminal CLK
Channel queue		Up to eight unique, ascending channels
Throughput, Raspberry Pi® 2 / 3	Single board	100 kS/s max
	Multiple boards	Up to 320 kS/s aggregate (Note 1)
Throughput, Raspberry Pi A+ / B+	Single board	Up to 100 kS/s (Note 1)
	Multiple boards	Up to 100 kS/s aggregate (Note 1)

Note 1: Depends on the load on the Raspberry Pi processor. The highest throughput may be achieved by using a Raspberry Pi 3.

Accuracy

Analog input DC voltage measurement accuracy

Table 2. DC Accuracy components and specifications. All values are (±)

Range	Gain error (% of reading)	Offset error (mV)	Absolute accuracy at Full Scale (mV)	Gain temperature coefficient (% reading/°C)	Offset temperature coefficient (mV/°C)
±10V	0.098	11	20.8	0.016	0.87

Noise performance

For the peak to peak noise distribution test, the input channel is connected to AGND at the input terminal block, and 12,000 samples are acquired at the maximum throughput.

Table 3. Noise performance specifications

Range	Counts	LSBrms
±10 V	5	0.76

External digital trigger

Table 4. External digital trigger specifications

Parameter	Specification
Trigger source	TRIG input
Trigger mode	Software configurable for edge or level sensitive, rising or falling edge, high or low level.
Trigger latency	Internal pacer: 1 µs max External pacer: 1 µs + 1 pacer clock cycle max
Trigger pulse width	125 ns min
Input type	Schmitt trigger, weak pull-down to ground (approximately 10 K)
Input high voltage threshold	2.64 V min
Input low voltage threshold	0.66 V max
Input voltage limits	5.5 V absolute max –0.5 V absolute min 0 V recommended min

External sample clock input/output

Table 5. External sample clock I/O specifications

Parameter	Specification
Terminal name	CLK
Terminal types	Bidirectional, defaults to input when not sampling analog channels
Direction (software-selectable)	Output: Outputs internal sample clock; active on rising edge Input: Receives sample clock from external source; active on rising edge
Input clock rate	100 kHz max
Input clock pulse width	400 ns min
Input type	Schmitt trigger, weak pull-down to ground in input mode (approximately 10 K), protected with 150 Ω series resistor
Input high voltage threshold	2.64 V min
Input low voltage threshold	0.66 V max
Input voltage limits	5.5V absolute max -0.5V absolute min 0V recommended min
Output high voltage	3.0 V min (IOH = -50 µA) 2.65 V min (IOH = –3 mA)
Output low voltage	0.1 V max (IOL = 50 µA) 0.8 V max (IOL = 3 mA)
Output current	±3 mA max

Memory

Table 6. Memory specifications

Parameter	Specification
Data FIFO	7 K (7,168) analog input samples
Non-volatile memory	4 KB (ID and calibration storage, no user-modifiable memory)

Power

Table 7. Power specifications

Parameter	Conditions	Specification
Supply current, 3.3V supply	Typical	35 mA
	Maximum	55 mA

Interface specifications

Table 8. Interface specifications

Parameter	Specification
Raspberry Pi ™ GPIO pins used	GPIO 8, GPIO 9, GPIO 10, GPIO 11 (SPI interface) ID_SD, ID_SC (ID EEPROM) GPIO 12, GPIO 13, GPIO 26, (Board address)
Data interface type	SPI slave device, CE0 chip select
SPI mode	1
SPI clock rate	10 MHz, max

Environmental

Table 9. Environmental specifications

Parameter	Specification
Operating temperature range	0 °C to 55 °C
Storage temperature range	–40 °C to 85 °C
Humidity	0% to 90% non-condensing

Mechanical

Table 10. Mechanical specifications

Parameter	Specification
Dimensions (L × W × H)	65 × 56.5 × 12 mm (2.56 × 2.22 × 0.47 in.) max

Screw terminal connector

Table 11. Screw terminal connector specifications

Parameter	Specification
Connector type	Screw terminal
Wire gauge range	16 AWG to 30 AWG

Table 12. Screw terminal pinout

Connector J2		
Pin	**Signal name**	**Pin description**
1	CH0	Channel 0
2	CH1	Channel 1
3	GND	Analog ground
4	CH2	Channel 2
5	CH3	Channel 3
6	GND	Analog ground
Connector J3		
Pin	**Signal name**	**Pin description**
7	CH4	Channel 4
8	CH5	Channel 5
9	GND	Analog ground
10	CH6	Channel 6
11	CH7	Channel 7
12	GND	Analog ground
13	CLK	Sample clock input / output
14	GND	Digital ground
15	TRIG	Digital trigger input
16	GND	Digital ground

CHAPTER

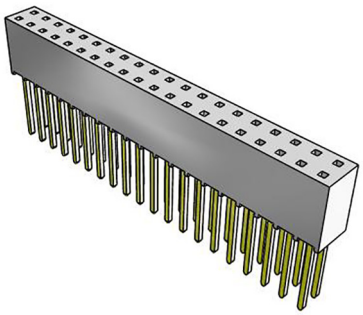
TWO

INSTALLING THE HAT BOARD

2.1 Installing a single board

1. Power off the Raspberry Pi.
2. Locate the 4 standoffs. A typical standoff is shown here:

3. Attach the 4 standoffs to the Raspberry Pi by inserting the male threaded portion through the 4 corner holes on the Raspberry Pi from the top and securing them with the included nuts from the bottom.
4. Install the 2x20 receptacle with extended leads onto the Raspberry Pi GPIO header by pressing the female portion of the receptacle onto the header pins, being careful not to bend the leads of the receptacle. The 2x20 receptacle looks like:



5. **The HAT must be at address 0.** Remove any jumpers from the address header locations A0-A2 on the HAT board.
6. Insert the HAT board onto the leads of the 2x20 receptacle so that the leads go into the holes on the bottom of the HAT board and come out through the 2x20 connector on the top of the HAT board. The 4 mounting holes in the corners of the HAT board must line up with the standoffs. Slide the HAT board down until it rests on the standoffs.
7. Insert the included screws through the mounting holes on the HAT board into the threaded holes in the standoffs and lightly tighten them.

2.2 Installing multiple boards

1. Follow steps 1-6 in the single board installation procedure for the first HAT board.
2. Connect all desired field wiring to the first board because the screw terminals will not be accessible once additional boards are installed above it.
3. Install the standoffs of the additional board by inserting the male threaded portions through the 4 corner holes of the installed HAT board and threading them into the standoffs below.
4. Install the next 2x20 receptacle with extended leads onto the leads of the previous 2x20 receptacle by pressing the female portion of the new receptacle onto the previous receptacle leads, being careful not to bend the leads of either receptacle.
5. Install the appropriate address jumpers onto address header locations A0-A2 of the new HAT board. The recommended addressing method is to have the addresses increment from 0 as the boards are installed, i.e. 0, 1, 2, and so forth. **There must always be a board at address 0.** The jumpers are installed in this manner (install jumpers where "Y" appears):

Address	A0	A1	A2
0			
1	Y		
2		Y	
3	Y	Y	
4			Y
5	Y		Y
6		Y	Y
7	Y	Y	Y

6. Insert the new HAT board onto the leads of the 2x20 receptacle so that the leads go into the holes on the bottom of the HAT board and come out through the 2x20 connector on the top of the HAT board. The 4 mounting holes in the corners of the HAT board must line up with the standoffs. Slide the HAT board down until it rests on the standoffs.
7. Repeat steps 2-6 for each board to be added.
8. Insert the included screws through the mounting holes on the top HAT board into the threaded holes in the standoffs and lightly tighten them.

CHAPTER

THREE

INSTALLING AND USING THE LIBRARY

The project is hosted at https://github.com/nwright98/daqhats.

3.1 Installation

1. Power off the Raspberry Pi then attach one or more HAT boards, using unique address settings for each. When using a single board, leave it at address 0 (all address jumpers removed.) One board must always be at address 0 so the OS reads a HAT EEPROM and initializes the hardware correctly.
2. Power on the Pi and log in. Open a terminal window if using the graphical interface.
3. If git is not already installed, update installation packages and install it:

```
sudo apt-get update
sudo apt-get install git
```

4. Download this package to your user folder with git:

```
cd ~
git clone https://github.com/nwright98/daqhats
```

5. Build and install the shared library and optional Python support. The installer will ask if you want to install Python 2 and Python 3 support. It will also detect the HAT board EEPROMs and save the contents if needed:

```
cd ~/daqhats
sudo ./install.sh
```

6. [Optional] To update the firmware on your MCC 118 board(s) use the firmware update tool. The "0" in the example below is the board address. The line with the "-b" option updates the bootloader. Repead the two commands for each MCC 118 address in your board stack:

```
mcc118_firmware_update -b 0 ~/daqhats/tools/MCC_118.hex
mcc118_firmware_update 0 ~/daqhats/tools/MCC_118.hex
```

You can now run the example programs under ~/daqhats/examples and create your own programs.

To uninstall the package use:

```
cd ~/daqhats
sudo ./uninstall.sh
```

If you change your board stackup and have more than one HAT board attached you must update the saved EEPROM images for the library to have the correct board information:

```
sudo daqhats_read_eeproms
```

3.2 Creating a C program

- The daqhats headers are installed in /usr/local/include/daqhats. Add the compiler option `-I/usr/local/include` in order to find the header files when compiling, and the include line `#include <daqhats/daqhats.h>` to your source code.
- The shared library, libdaqhats.so, is installed in /usr/local/lib. Add the linker option `-ldaqhats` to include this library.
- Study the example programs, example makefile, and library documentation for more information.

3.3 Creating a Python program

- The Python package is named *daqhats*. Use it in your code with `import daqhats`.
- Study the example programs and library documentation for more information.

CHAPTER

FOUR

C LIBRARY REFERENCE

The C library is organized as a global function for listing the HAT boards attached to your system, and board-specific functions to provide full functionality for each type of board. The library may be used with C and C++.

4.1 Global functions and data

4.1.1 Functions

Function	Description
hat_list()	Return a list of detected MCC HAT boards.

int **hat_list** (uint16_t *filter_id*, struct *HatInfo* * *list*)

Return a list of detected MCC HAT boards.

It creates the list from the HAT EEPROM files that are currently on the system. In the case of a single HAT at address 0 this information is automatically provided by Raspbian. However, when you have a stack of multiple boards you must extract the EEPROM images using the daqhats_read_eeproms tool.

Example usage:

```
int count = hat_list(HAT_ID_ANY, NULL);

if (count > 0)
{
    struct HatInfo* list = (struct HatInfo*)malloc(count * sizeof(struct␣
↪HatInfo));
    hat_list(HAT_ID_ANY, list);

    // perform actions with list

    free(list);
}
```

Return The number of boards found.

Parameters

- `filter_id`: An optional *ID* filter to only return boards with a specific ID. Use *HAT_ID_ANY* to return all boards.
- `list`: A pointer to a user-allocated array of struct *HatInfo*. The function will fill the structures with information about the detected boards. You may have an array of the maximum number of boards

(*MAX_NUMBER_HATS*) or call this function while passing NULL for list, which will return the count of boards found, then allocate the correct amount of memory and call this function again with a valid pointer.

4.1.2 Data types and definitions

MAX_NUMBER_HATS 8
The maximum number of MCC HATs that may be connected.

4.1.2.1 HAT IDs

enum HatIDs
Known MCC HAT IDs.

Values:

HAT_ID_ANY = 0
Match any MCC ID in *hat_list()*.

HAT_ID_MCC_118 = 0x0142
MCC 118 ID.

HAT_ID_MCC_118_BOOTLOADER = 0x8142
MCC 118 in firmware update mode ID.

4.1.2.2 Result Codes

enum ResultCode
Return values from the library functions.

Values:

RESULT_SUCCESS = 0
Success, no errors.

RESULT_BAD_PARAMETER = -1
A parameter passed to the function was incorrect.

RESULT_BUSY = -2
The device is busy.

RESULT_TIMEOUT = -3
There was a timeout accessing a resource.

RESULT_LOCK_TIMEOUT = -4
There was a timeout while obtaining a resource lock.

RESULT_INVALID_DEVICE = -5
The device at the specified address is not the correct type.

RESULT_RESOURCE_UNAVAIL = -6
A needed resource was not available.

RESULT_UNDEFINED = -10
Some other error occurred.

4.1.2.3 HatInfo structure

struct HatInfo
Contains information about a specific board.

Public Members

uint8_t **address**
The board address.

uint16_t **id**
The product ID, one of *HatIDs*.

uint16_t **version**
The hardware version.

char HatInfo::product_name[256]
The product name.

4.1.2.4 Analog Input / Scan Options

OPTS_NOSCALEDATA (0x0001)
Return ADC code instead of scaled data (voltage, temperature, etc.)

OPTS_NOCALIBRATEDATA (0x0002)
Return uncalibrated data.

OPTS_EXTCLOCK (0x0004)
Use an external sample clock.

OPTS_EXTTRIGGER (0x0008)
Use an external trigger.

OPTS_CONTINUOUS (0x0010)
Scan until stopped.

4.2 MCC 118 functions and data

4.2.1 Functions

Function	Description
mcc118_open()	Open an MCC 118 for use.
mcc118_is_open()	Check if an MCC 118 is open.
mcc118_close()	Close an MCC 118.
mcc118_blink_led()	Blink the MCC 118 LED.
mcc118_firmware_version()	Get the firmware version.
mcc118_serial()	Read the serial number.
mcc118_calibration_date()	Read the calibration date.
mcc118_calibration_coefficient_read()	Read the calibration coefficients for a channel.
mcc118_calibration_coefficient_write()	Write the calibration coefficients for a channel.
mcc118_a_in_num_channels()	Get the number of analog input channels.
mcc118_a_in_read()	Read an analog input value.
mcc118_trigger_mode()	Set the external trigger input mode.
mcc118_a_in_scan_actual_rate()	Read the actual sample rate for a set of scan parameters.
mcc118_a_in_scan_start()	Start a hardware-paced analog input scan.
mcc118_a_in_scan_buffer_size()	Read the size of the internal scan data buffer.
mcc118_a_in_scan_read()	Read scan data / status.
mcc118_a_in_scan_channel_count()	Get the number of channels in the current scan.
mcc118_a_in_scan_stop()	Stop the scan.
mcc118_a_in_scan_cleanup()	Free scan resources.

int **mcc118_open**(uint8_t *address*)
Open a connection to the MCC 118 device at the specified address.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7).

int **mcc118_close**(uint8_t *address*)
Close a connection to an MCC 118 device and free allocated resources.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7).

int **mcc118_is_open**(uint8_t *address*)
Check if an MCC 118 is open.

Return 1 if open, 0 if not open.

Parameters

- `address`: The board address (0 - 7).

int **mcc118_blink_led**(uint8_t *address*, uint8_t *count*)
Blink the LED on the MCC 118.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7).
- `count`: The number of times to blink (0 - 255).

int **mcc118_firmware_version**(uint8_t *address*, uint16_t * *version*, uint16_t * *boot_version*)
Return the board firmware and bootloader versions.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `version`: Receives the firmware version. The version will be in BCD hexadecimal with the high byte as the major version and low byte as minor, i.e. 0x0103 is version 1.03.
- `boot_version`: Receives the bootloader version. The version will be in BCD hexadecimal as above.

int **mcc118_serial**(uint8_t *address*, char * *buffer*)
Read the MCC 118 serial number.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `buffer`: Pass a user-allocated buffer pointer to receive the serial number as a string. The buffer must be at least 9 characters in length.

int **mcc118_calibration_date**(uint8_t *address*, char * *buffer*)
Read the MCC 118 calibration date.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `buffer`: Pass a user-allocated buffer pointer to receive the date as a string (format "YYYY-MM-DD"). The buffer must be at least 11 characters in length.

int **mcc118_calibration_coefficient_read**(uint8_t *address*, uint8_t *channel*, double * *slope*, double * *offset*)
Read the MCC 118 calibration coefficients for a single channel.

The coefficients are applied in the library as:

```
calibrated_ADC_code = (raw_ADC_code * slope) + offset
```

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel`: The channel number (0 - 7).

- `slope`: Receives the slope.
- `offset`: Receives the offset.

int **mcc118_calibration_coefficient_write**(uint8_t *address*, uint8_t *channel*, double *slope*, double *offset*)

Temporarily write the MCC 118 calibration coefficients for a single channel.

The user can apply their own calibration coefficients by writing to these values. The values will reset to the factory values from the EEPROM whenever *mcc118_open()* is called. This function will fail and return *RESULT_BUSY* if a scan is running when it is called.

The coefficients are applied in the library as:

```
calibrated_ADC_code = (raw_ADC_code * slope) + offset
```

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel`: The channel number (0 - 7).
- `slope`: The new slope value.
- `offset`: The new offset value.

int **mcc118_a_in_num_channels**(void)

Return the number of analog input channels on the MCC 118.

Return The number of channels.

int **mcc118_a_in_read**(uint8_t *address*, uint8_t *channel*, uint32_t *options*, double * *value*)

Perform a single reading of an analog input channel and return the value.

Will return *RESULT_BUSY* if called while a scan is running.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel`: The analog input channel number, 0 - 7.
- `options`: Options bitmask (only *OPTS_NOSCALEDATA* and *OPTS_NOCALIBRATEDATA* are supported)
- `value`: Receives the analog input value.

int **mcc118_trigger_mode**(uint8_t *address*, uint8_t *mode*)

Set the trigger input mode.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `mode`: One of the *trigger mode* values.

int **mcc118_a_in_scan_actual_rate**(uint8_t *channel_count*, double *sample_rate_per_channel*, double * *actual_sample_rate_per_channel*)

Read the actual sample rate per channel for a requested sample rate.

The internal scan clock is generated from a 16 MHz clock source so only discrete frequency steps can be achieved. This function will return the actual rate for a requested channel count, rate and burst mode setting. This function does not perform any actions with a board, it simply calculates the rate.

Return *Result code*, *RESULT_SUCCESS* if successful, *RESULT_BAD_PARAMETER* if the scan parameters are not achievable on an MCC 118.

Parameters

- `channel_count`: The number of channels in the scan.
- `sample_rate_per_channel`: The desired sampling rate in samples per second per channel, max 100,000.
- `actual_sample_rate_per_channel`: The actual sample rate that would occur when requesting this rate on an MCC 118, or 0 if there is an error.

int **mcc118_a_in_scan_start**(uint8_t *address*, uint8_t *channel_mask*, uint32_t *samples_per_channel*, double *sample_rate_per_channel*, uint32_t *options*)

Start a hardware-paced analog input scan.

The scan runs as a separate thread from the user's code. The function will allocate a scan buffer and read data from the device into that buffer. The user reads the data from this buffer and the scan status using the *mcc118_a_in_scan_read()* function. *mcc118_a_in_scan_stop()* is used to stop a continuous scan, or to stop a finite scan before it completes. The user must call *mcc118_a_in_scan_cleanup()* after the scan has finished and all desired data has been read; this frees all resources from the scan and allows additional scans to be performed.

The buffer size will be allocated as follows:

Finite mode: Total number of samples in the scan

Continuous mode (buffer size is per channel): Either **samples_per_channel** or the value in the following table, whichever is greater

Sample Rate	Buffer Size (per channel)
Not specified	10 kS
0-100 S/s	1 kS
100-10k S/s	10 kS
10k-100k S/s	100 kS

Specifying a very large value for **samples_per_channel** could use too much of the Raspberry Pi memory. If the memory allocation fails, the function will return *RESULT_RESOURCE_UNAVAIL*. The allocation could succeed, but the lack of free memory could cause other problems in the Raspberry Pi. If you need to acquire a high number of samples then it is better to run the scan in continuous mode and stop it when you have acquired the desired amount of data. If a scan is already running this function will return *RESULT_BUSY*.

Return *Result code*, *RESULT_SUCCESS* if successful, *RESULT_BUSY* if a scan is already running.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel_mask`: A bit mask of the channels to be scanned. Set each bit to enable the associated channel (0x01 - 0xFF.)
- `samples_per_channel`: The number of samples to acquire for each channel in the scan.

- `sample_rate_per_channel`: The sampling rate in samples per second per channel, max 100,000. When using an external sample clock set this value to the maximum expected rate of the clock.
- `options`: The options for the scan. This is a bitmask and may be an ORed combination of:
  - *OPTS_NOSCALEDATA*: Return ADC codes instead of voltage.
  - *OPTS_NOCALIBRATEDATA*: Return uncalibrated values.
  - *OPTS_EXTCLOCK*: Use an external sample clock on the CLK input.
  - *OPTS_EXTTRIGGER*: Use an external trigger source on the TRIG input.
  - *OPTS_CONTINUOUS*: Scan until stopped (samples_per_channel is only used for buffer allocation.)

int **mcc118_a_in_scan_buffer_size**(uint8_t *address*, uint32_t * *buffer_size_samples*)

Returns the size of the internal scan data buffer.

An internal data buffer is allocated for the scan when *mcc118_a_in_scan_start()* is called. This function returns the total size of that buffer in samples.

Return *Result code*, *RESULT_SUCCESS* if successful, *RESULT_RESOURCE_UNAVAIL* if a scan is not currently running under this instance of the device, or *RESULT_BAD_PARAMETER* if the address is invalid or buffer_size_samples is NULL.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `buffer_size_samples`: Receives the size of the buffer in samples. Each sample is a **double**.

int **mcc118_a_in_scan_read**(uint8_t *address*, uint16_t * *status*, int32_t *samples_per_channel*, double *timeout*, double * *buffer*, uint32_t *buffer_size_samples*, uint32_t * *samples_read_per_channel*)

Reads status and multiple samples from an analog input scan.

The scan is started with *mcc118_a_in_scan_start()* and runs in a background thread that reads the data from the board into an internal scan buffer. This function reads the data from the scan buffer, and returns the current scan status.

Return *Result code*, *RESULT_SUCCESS* if successful, *RESULT_RESOURCE_UNAVAIL* if a scan is not currently running under this instance of the device.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `status`: Receives the scan status, an ORed combination of the following flags:
  - *STATUS_HW_OVERRUN*: The device scan buffer was not read fast enough and data was lost.
  - *STATUS_BUFFER_OVERRUN*: The thread scan buffer was not read by the user fast enough and data was lost.
  - *STATUS_TRIGGERED*: The trigger conditions have been met.
  - *STATUS_RUNNING*: The scan is running.
- `samples_per_channel`: The number of samples per channel to read. Specify **-1** to read all available samples, or **0** to only read the scan status and not return data. If buffer does not contain enough space then the function will read as many samples per channel as will fit in buffer.

- `timeout`: The amount of time in seconds to wait for the samples to be read. Specify a negative number to wait indefinitely or **0** to return immediately with whatever samples are available.
- `buffer`: The buffer to read samples into. May be NULL if samples_per_channel is **0**.
- `buffer_size_samples`: The size of the buffer in samples. Each sample is a **double**.
- `samples_read_per_channel`: Returns the actual number of samples read from each channel. May be **NULL** if samples_per_channel is **0**.

int **mcc118_a_in_scan_channel_count**(uint8_t *address*)
Return the number of channels in the current analog input scan.

Return The number of channels, 0 - 8.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.

int **mcc118_a_in_scan_stop**(uint8_t *address*)
Stops an analog input scan.

The scan is stopped immediately. The scan data that has been read into the scan buffer is available until *mcc118_a_in_scan_cleanup()* is called.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.

int **mcc118_a_in_scan_cleanup**(uint8_t *address*)
Free analog input scan resources after the scan is complete.

Return *Result code*, *RESULT_SUCCESS* if successful.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.

4.2.2 Data definitions

4.2.2.1 Trigger Modes

enum TriggerMode
Scan trigger input modes.

Values:

TRIG_RISING_EDGE = 0
Trigger on a rising edge.

TRIG_FALLING_EDGE = 1
Trigger on a falling edge.

TRIG_ACTIVE_HIGH = 2
Trigger any time the signal is high.

TRIG_ACTIVE_LOW = 3
Trigger any time the signal is low.

4.2.2.2 Scan Status Flags

STATUS_HW_OVERRUN (0x0001)
A hardware overrun occurred.

STATUS_BUFFER_OVERRUN (0x0002)
A scan buffer overrun occurred.

STATUS_TRIGGERED (0x0004)
The trigger event occurred.

STATUS_RUNNING (0x0008)
The scan is running.

CHAPTER

FIVE

PYTHON LIBRARY REFERENCE

The Python library is organized as a global method for listing the HAT boards attached to your system, and board-specific classes to provide full functionality for each type of board. The Python package is named *daqhats*.

5.1 Global methods and data

5.1.1 Methods

Method	Description
hat_list()	Return a list of detected MCC HAT boards.

daqhats.**hat_list**(*filter_by_id=0*)

Return a list of detected MCC HAT boards.

Scans certain locations for information from the HAT EEPROMs. Verifies the contents are valid HAT EEPROM contents and returns a list of dictionaries containing information on the HAT. Info will only be returned for MCC HATs. The EEPROM contents are stored in /etc/mcc/hats when using the daqhats_read_eeproms tool, or in /proc/device-tree in the case of a single HAT at address 0.

Parameters **filter_by_id** (*int*) – If this is *HatIDs.ANY* return all MCC HATs found. Otherwise, return only HATs with ID matching this value.

Returns

A list of dictionaries, the number of elements match the number of HATs found. Each dictionary will contain the following keys

address (int): device address

id (int): device product ID, identifies the type of MCC HAT

version (int): device hardware version

product_name (str): device product name

Return type list

5.1.2 Data

5.1.2.1 Hat IDs

class daqhats.**HatIDs**
Known MCC HAT IDs.

ANY = 0
Match any MCC ID in *hat_list()*

MCC_118 = 322
MCC 118 ID

5.1.2.2 Trigger modes

class daqhats.**TriggerModes**
Scan trigger input modes.

RISING_EDGE = 0
Trigger on a rising edge.

FALLING_EDGE = 1
Trigger on a falling edge.

ACTIVE_HIGH = 2
Trigger any time the signal is high

ACTIVE_LOW = 3
Trigger any time the signal is low.

5.1.3 HatError class

exception daqhats.**HatError**(*address*, *value*)
Exceptions raised for MCC HAT specific errors.

Parameters

- **address** (*int*) – the address of the board that caused the exception.
- **value** (*str*) – the exception description.

5.2 MCC 118 class

5.2.1 Methods

class daqhats.**mcc118**(*address=0*)
The class for an MCC 118 board.

Parameters **address** (*int*) – board address, must be 0-7.

Raises *HatError* – the board did not respond or was of an incorrect type

Methods

Method	Description
mcc118.blink_led()	Blink the MCC 118 LED.
mcc118.firmware_version()	Get the firmware version.
mcc118.address()	Read the board's address.
mcc118.serial()	Read the serial number.
mcc118.calibration_date()	Read the calibration date.
mcc118.calibration_coefficient_read()	Read the calibration coefficients for a channel.
mcc118.calibration_coefficient_write()	Write the calibration coefficients for a channel.
mcc118.a_in_num_channels()	Get the number of analog input channels.
mcc118.a_in_read()	Read an analog input channel.
mcc118.trigger_mode()	Set the external trigger input mode.
mcc118.a_in_scan_actual_rate()	Read the actual sample rate for a requested sample rate.
mcc118.a_in_scan_start()	Start a hardware-paced analog input scan.
mcc118.a_in_scan_buffer_size()	Read the size of the internal scan data buffer.
mcc118.a_in_scan_read()	Read scan status / data (list).
mcc118.a_in_scan_read_numpy()	Read scan status / data (NumPy array).
mcc118.a_in_scan_channel_count()	Get the number of channels in the current scan.
mcc118.a_in_scan_stop()	Stop the scan.
mcc118.a_in_scan_cleanup()	Free scan resources.

firmware_version()
Read the board firmware and bootloader versions.

Returns

versions containing the following keys

version (string): The firmware version, i.e "1.03".
bootloader_version (string): The bootloader version, i.e "1.01".

Return type dictionary

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

serial()
Read the serial number.

Returns The serial number.

Return type string

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

blink_led(*count*)
Blink the MCC 118 LED.

Parameters **count** (*int*) – The number of times to blink (max 255).

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

calibration_date()
Read the calibration date.

Returns The calibration date in the format "YYYY-MM-DD".

Return type string

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

calibration_coefficient_read(*channel*)
Read the calibration coefficients for a single channel.

The coefficients are applied in the library as:

```
calibrated_ADC_code = (raw_ADC_code * slope) + offset
```

Returns

coefficients containing the following keys

slope (float): The slope.
offset (float): The offset.

Return type dictionary

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

calibration_coefficient_write(*channel*, *slope*, *offset*)
Temporarily write the calibration coefficients for a single channel.

The user can apply their own calibration coefficients by writing to these values. The values will reset to the factory values from the EEPROM whenever the class is initialized. This function will fail and raise a HatError exception if a scan is running when it is called.

The coefficients are applied in the library as:

```
calibrated_ADC_code = (raw_ADC_code * slope) + offset
```

Parameters

- **slope** (*float*) – The new slope value.
- **offset** (*float*) – The new offset value.

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

trigger_mode(*mode*)
Set the external trigger input mode.

The available modes are:

- *TriggerModes.RISING_EDGE*: Trigger when the TRIG input transitions from low to high.
- *TriggerModes.RISING_EDGE*: Trigger when the TRIG input transitions from high to low.
- *TriggerModes.ACTIVE_HIGH*: Trigger when the TRIG input is high.
- *TriggerModes.ACTIVE_LOW*: Trigger when the TRIG input is low.

Parameters **mode** (*TriggerModes*) – The trigger mode.

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

static a_in_num_channels()
Return the number of analog input channels.

Returns the number of channels.

Return type int

a_in_read(*channel*, *scaled=True*, *calibrated=True*)
Perform a single reading of an analog input channel and return the value.

Parameters

- **channel** (*int*) – The analog input channel number, 0-7.
- **scaled** (*bool*) – True to return voltage, False to return ADC code.
- **calibrated** (*bool*) – True to apply calibration to the value, False to return uncalibrated value.

Returns the read value

Return type float

Raises

- *HatError* – the board is not initialized, does not respond, or responds incorrectly.
- ValueError – the channel number is invalid.

a_in_scan_actual_rate(*channel_count*, *sample_rate_per_channel*)
Read the actual sample rate per channel for a requested sample rate.

The internal scan clock is generated from a 16 MHz clock source so only discrete frequency steps can be achieved. This function will return the actual rate for a requested channel count and rate setting.

This function does not perform any actions with a board, it simply calculates the rate.

Parameters

- **channel_count** (*int*) – The number of channels in the scan, 1-8.
- **sample_rate_per_channel** (*float*) – The desired per-channel rate of the internal sampling clock, max 100,000.0.

Returns the actual sample rate

Return type float

Raises ValueError – a scan argument is invalid.

a_in_scan_start(*channel_mask*, *samples_per_channel*, *sample_rate_per_channel*, *continuous=False*, *external_clock=False*, *external_trigger=False*, *scaled=True*, *calibrated=True*)
Start a hardware-paced analog input channel scan.

The scan runs as a separate thread from the user's code. This function will allocate a scan buffer and start the thread that reads data from the device into that buffer. The user reads the data from the scan buffer and the scan status using the *a_in_scan_read()* function. *a_in_scan_stop()* is used to stop a continuous scan, or to stop a finite scan before it completes. The user must call *a_in_scan_cleanup()* after the scan has finished and all desired data has been read; this frees all resources from the scan and allows additional scans to be performed.

The scan buffer size will be allocated as follows:

Finite mode: Total number of samples in the scan.

Continuous mode: Either **samples_per_channel** or the value in the table below, whichever is greater.

Sample Rate	Buffer Size (per channel)
Not specified	10 kS
0-100 S/s	1 kS
100-10k S/s	10 kS
10k-100k S/s	100 kS

Specifying a very large value for samples_per_channel could use too much of the Raspberry Pi memory. If the memory allocation fails, the function will raise a HatError with this description. The allocation could succeed, but the lack of free memory could cause other problems in the Raspberry Pi. If you need to acquire a high number of samples then it is better to run the scan in continuous mode and stop it when you have acquired the desired amount of data. If a scan is already running this function will raise a HatError.

Parameters

- **channel_mask** (*int*) – A bit mask of the desired channels (0x01 - 0xFF).
- **samples_per_channel** (*int*) – The number of samples to acquire per channel.
- **sample_rate_per_channel** (*float*) – The per-channel rate of the internal sampling clock, or the expected maximum rate of an external sampling clock, max 100,000.0.
- **continuous** (*bool*) – False for a finite scan, True for a scan that runs until stopped by the user.
- **external_clock** (*bool*) – False to use the internal sampling clock, True to use an external clock on the CLK pin.
- **external_trigger** (*bool*) – False to start the scan immediately, True to start the scan when the external trigger conditions (set with *trigger_mode()*) are met on the TRIG pin.
- **scaled** (*bool*) – True to return voltage, False to return A/D code.
- **calibrated** (*bool*) – True to apply calibration to the value, False to return uncalibrated value.

Raises

- *HatError* – a scan is already running; memory could not be allocated; the board is not initialized, does not respond, or responds incorrectly.
- ValueError – a scan argument is invalid.

Examples

```
>>> a_in_scan_start(channel_mask = 0x05, samples_per_channel = 1000, sample_
↪rate_per_channel = 10000)
```

This will include channels 0 and 2 in the scan. 1000 samples will be acquired per channel, resulting in 2000 samples being read (samples_per_channel * number of channels). The sample rate per channel is 10,000 Hz, so the ADC will convert at a frequency of 20,000 Hz (sample_rate_per_channel * number of channels). The scan will start immediately.

a_in_scan_buffer_size()

Read the internal scan data buffer size.

An internal data buffer is allocated for the scan when *a_in_scan_start()* is called. This function returns the total size of that buffer in samples.

Returns the buffer size in samples

Return type int

Raises *HatError* – the board is not initialized or no scan buffer is allocated (a scan is not running).

a_in_scan_read(*samples_per_channel*, *timeout*)
Read scan status and data (as a list).

The analog input scan is started with *a_in_scan_start()* and runs in the background. This function reads the status of that background scan and optionally reads sampled data from the scan buffer.

Parameters

- **samples_per_channel** (*int*) – The number of samples per channel to read from the scan buffer. Specify -1 to read all available samples or 0 to only read the scan status and return no data.
- **timeout** (*float*) – The amount of time in seconds to wait for the samples to be read. Specify a negative number to wait indefinitely, or 0 to return immediately with the samples that are already in the scan buffer. If the timeout is met and the specified number of samples have not been read, then the function will return with the amount that has been read and the timeout status set.

Returns

a dictionary containing the following keys:

running (bool): True if the scan is running, False if it has stopped or completed.
hardware_overrun (bool): True if the hardware could not acquire and unload samples fast enough and data was lost.
buffer_overrun (bool): True if the background scan buffer was not read fast enough and data was lost.
triggered (bool): True if the trigger conditions have been met and data acquisition started.
timeout (bool): True if the timeout time expired before the specified number of samples were read.
data (list of float): The data that was read from the scan buffer.

Return type dictionary

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

a_in_scan_read_numpy(*samples_per_channel*, *timeout*)
Read scan status and data (as a NumPy array).

This function is similar to *a_in_scan_read()* except that the *data* key in the returned dictionary is a NumPy array of float64 values and may be used directly with NumPy functions.

Parameters

- **samples_per_channel** (*int*) – The number of samples per channel to read from the scan buffer. Specify -1 to read all available samples or 0 to only read the scan status and return no data.
- **timeout** (*float*) – The amount of time in seconds to wait for the samples to be read. Specify a negative number to wait indefinitely, or 0 to return immediately with the samples that are already in the scan buffer. If the timeout is met and the specified number of samples have not been read, then the function will return with the amount that has been read and the timeout status set.

Returns

a dictionary containing the following keys:

running (bool): True if the scan is running, False if it has stopped or completed.
hardware_overrun (bool): True if the hardware could not acquire and unload samples fast enough and data was lost.
buffer_overrun (bool): True if the background scan buffer was not read fast enough and data was lost.
triggered (bool): True if the trigger conditions have been met and data acquisition started.
timeout (bool): True if the timeout time expired before the specified number of samples were read.
data (NumPy array of float64): The data that was read from the scan buffer.

Return type dictionary

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

a_in_scan_channel_count()
Read the number of channels in the current analog input scan.

Returns the number of channels (0 if no scan is running, 1-8 otherwise)

Return type int

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

a_in_scan_stop()
Stops an analog input scan.

The scan is stopped immediately. The scan data that has been read into the scan buffer is available until *a_in_scan_cleanup()* is called.

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

a_in_scan_cleanup()
Free analog input scan resources after the scan is complete.

This will free the scan buffer and other resources used by the background scan and make it possible to start another scan with *a_in_scan_start()*

Raises *HatError* – the board is not initialized, does not respond, or responds incorrectly.

address()
Return the device address.

INDEX