MCC DAQ HAT Library Documentation

Release 1.1.0

Measurement Computing

Sep 14, 2018

CONTENTS

CHAPTER

ONE

HARDWARE OVERVIEW

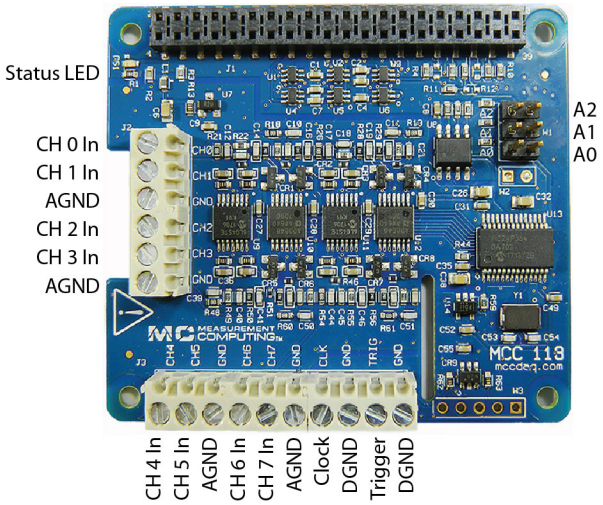
The MCC DAQ HATs are Raspberry Pi add-on boards (Hardware Attached on Top). They adhere to the Raspberry Pi HAT specification, but also extend it to allow stacking up to 8 MCC boards on a single Raspberry Pi.

C and Python libraries, documentation, and examples are provided to allow you to develop your own applications.

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
- Bidirectional scan 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** (CHx): Single-ended analog input terminals.

- **Clock** (CLK): Bidirectional terminal for scan clock input / output. Set the direction with software. Set for input to clock the scans with an external clock signal, or output to use the internal scan clock.
- **Trigger** (TRIG): 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** (GND): Common ground for the analog input terminals.
- **DGND** (GND): 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 jumpers on each additional connected board to set the desired address. Refer to the *Installing multiple boards* topic for more information about the recommended addressing method.

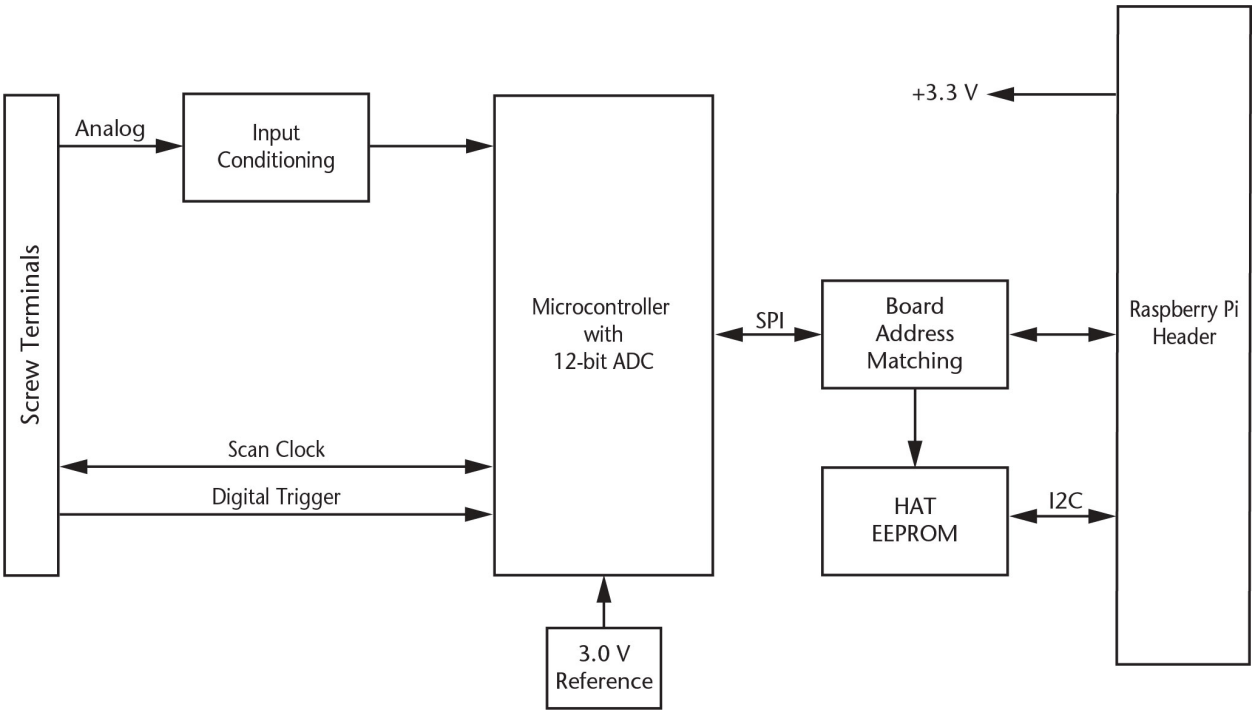
1.1.1.3 Status LED

The LED turns on when the board is connected to a Raspberry Pi with external power applied and flashes when communicating with the board. The LED may be blinked by the user.

1.1.1.4 Header connector

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

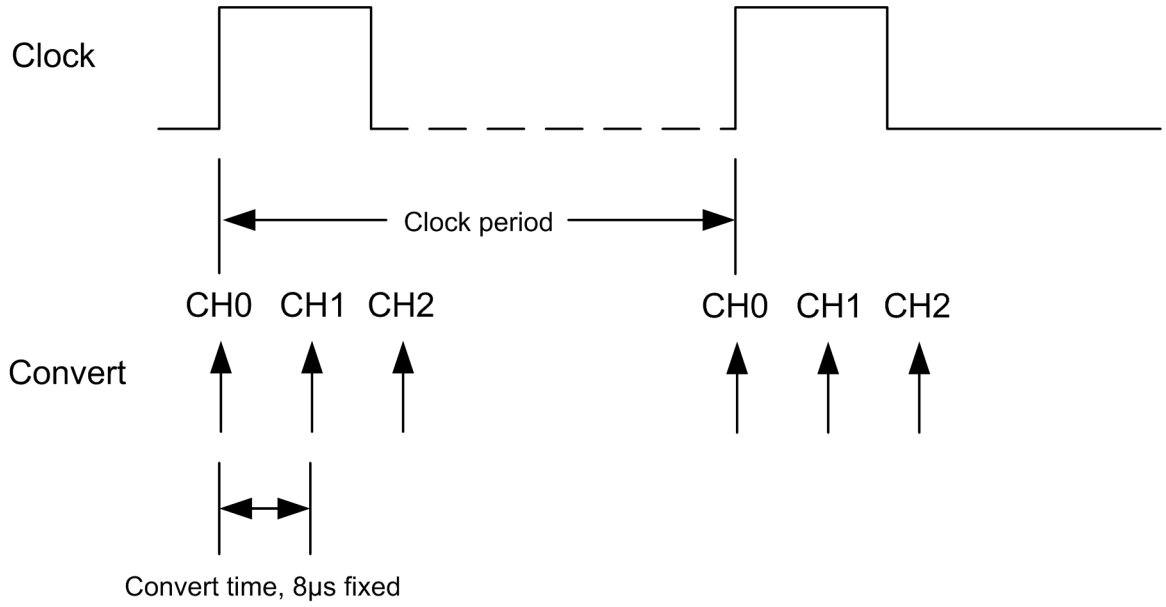
1.1.2 Functional block diagram



1.1.3 Functional details

1.1.3.1 Scan clock

The clock input / output (terminal CLK) is used to output the internal scan clock or apply an external scan clock to the device. The clock input signal may be a 3.3V or 5V TTL or CMOS logic signal, and the output will be 3.3V LVCMOS. A scan occurs for each rising edge of the clock, acquiring one sample from each of the selected channels in the scan. For example, when scanning channels 0, 1, and 2 the conversion activity will be:



1.1.3.2 Trigger

The trigger input (terminal TRIG) is used to hold off the beginning of an analog input scan until the desired condition is met at the trigger input. The trigger input signal may be a 3.3V or 5V TTL or CMOS logic signal. The input condition may be rising edge, falling edge, high level, or low level.

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 scan clock	0.004 S/s to 100 kS/s, software-selectable
	External scan clock	100 kS/s max
Sampling mode		One A/D conversion for each configured channel per clock
Conversion time	Per channel	8 µs
Scan clock source		• Internal scan clock • External scan 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, max (% of reading)	Offset error, max (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	Conditions	Specification
Trigger source		TRIG input
Trigger mode		Software configurable for rising or falling edge, or high or low level
Trigger latency	Internal scan clock	1 µs max
	External scan clock	1 µs + 1 scan 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 scan clock input/output

Table 5. External scan 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 scan clock; active on rising edge Input: Receives scan 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	5.5 V absolute max -0.5 V absolute min 0 V 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

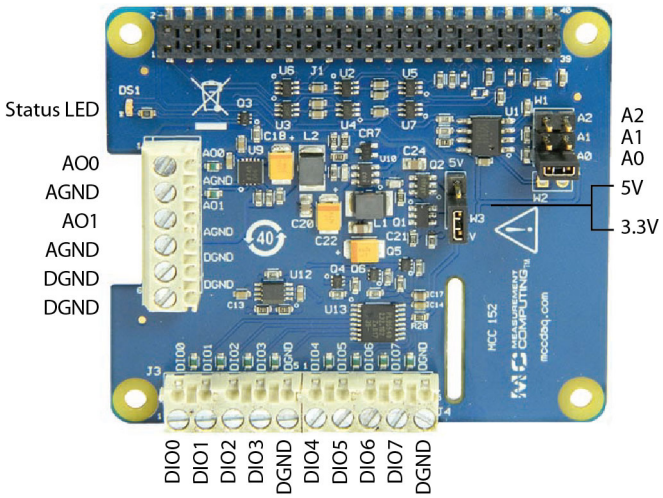
Table 12. Screw terminal pinout

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	Scan clock input / output
14	GND	Digital ground
15	TRIG	Digital trigger input
16	GND	Digital ground

1.2 MCC 152

The MCC 152 is an analog output / digital I/O board with the following features:

- 2 analog outputs
  - 12-bit D/A converter
  - 0 - 5 V outputs
- 8 digital I/O
  - 5 V / 3.3 V selectable
  - Programmable pull-up/pull-down resistors
  - Interrupt on input change



1.2.1 Board components

1.2.1.1 Screw terminals

- **Analog output 0** to **Analog output 1** (AOx): Analog output terminals.
- **Digital I/O 0** to **Digital I/O 7** (DIOx): Digital input/output terminals.
- **AGND** (AGND): Common ground for the analog output terminals.
- **VIO** (VIO): Digital I/O supply voltage (3.3V / 5V, depending on W3 position.)
- **DGND** (DGND): Common ground for the digital I/O terminals.

1.2.1.2 Address jumpers

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

1.2.1.3 DIO Power jumper

- **5V** and **3.3V**: Selects the DIO voltage.

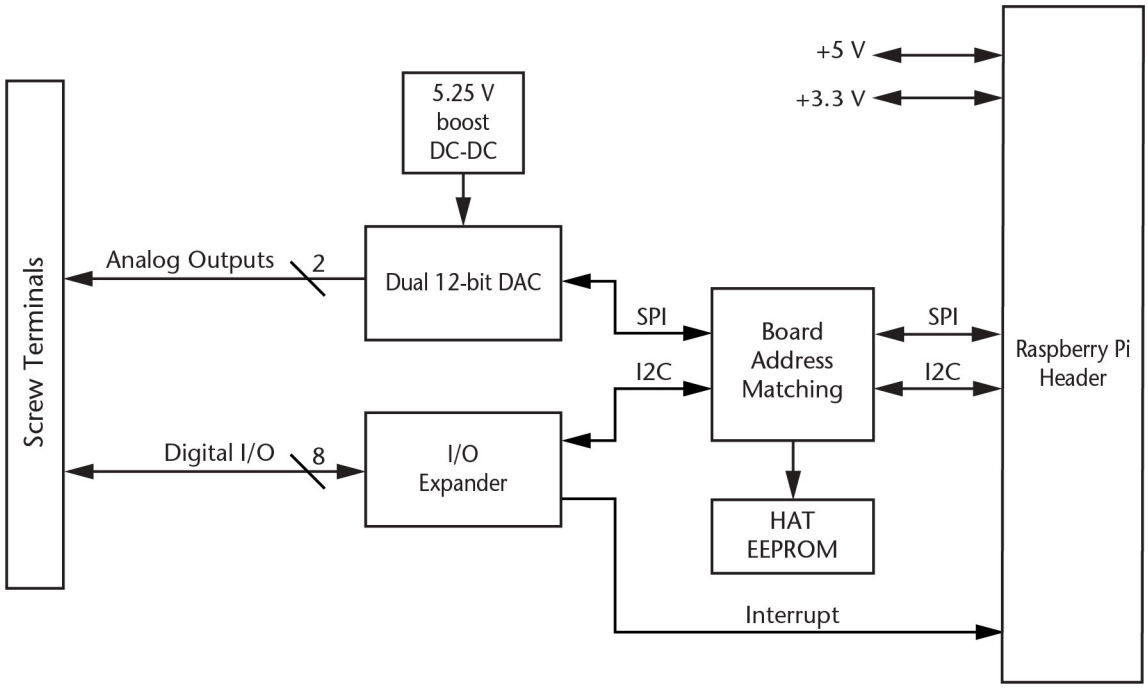
1.2.1.4 Status LED

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

1.2.1.5 Header connector

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

1.2.2 Functional block diagram

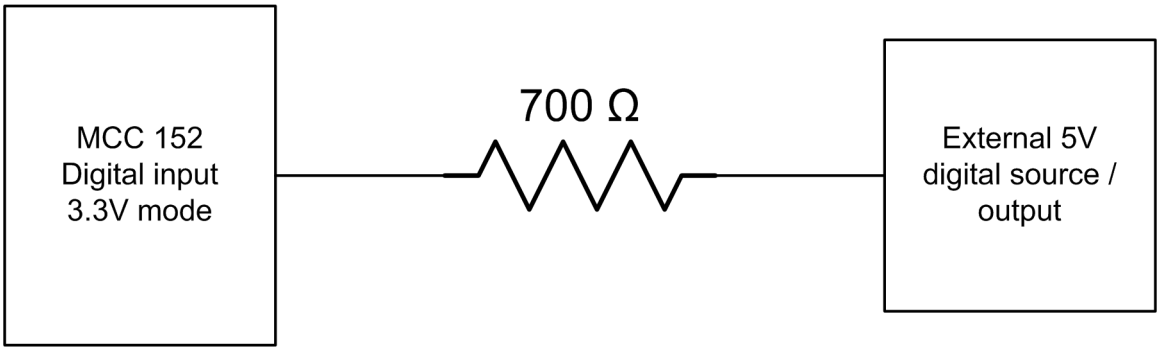


1.2.3 Functional details

1.2.3.1 Mixing 3.3V and 5V digital inputs

The MCC 152 digital inputs are tolerant of 5V signals when the DIO is set to 3.3V operation with jumper W3. However, current can flow into the MCC 152 from the 5V signal so the user must limit this current to avoid raising the voltage of the digital power supply rail (VIO) and possibly damaging components. MCC recommends using a series resistor of 700 ohms or larger.

Example:



1.2.4 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 output

Table 1. Analog output specifications

Parameter	Condition	Specification
Resolution		12 bits, 1 in 4,096
Output range		*0 V to 5.0 V*
Number of channels		2
Write time		12 µs, typ
Power on and reset voltage	Initializes to 000h code	0 V, ±10 mV
Output drive	Each D/A OUT	5 mA, sourcing
Slew rate		0.8 V/µs typ
Zero-scale error	000h code	2 mV typ 10 mV max

Table 2. Analog output accuracy, all values are (±); accuracy tested at no load

Range	Accuracy (LSB)
0 V to 5.0 V	5.0 typ, 45.0 max

Table 3. Analog output accuracy components, all values are (±)

Range	% of FSR	Gain Error at FS (mV)	Offset (mV) (Note 1)	Accuracy at FS (mV)
0 V to 5.0 V	0.08 typ, 0.72 max	4.0 typ, 36.0 max	1.0 typ, 9.0 max	5.0 typ, 45.0 max

Note 1: Zero-scale error may result in a "dead-band" digital input code region. In this case, changes in the requested output voltage may not produce a corresponding change in the output voltage when the voltage is less than 10 mV. The offset error is tested and specified at 10 mV.

Digital input/output

Table 4. Digital I/O specifications

Parameter	Conditions	Specification
Digital input type		CMOS
Number of I/O		8
Configuration		Each bit may be configured as input (power on default) or output
Pull-up configuration		Each bit has a programmable 100 kΩ pull resistor (50 to 150 kΩ range) that may be programmed as pull-up, pull-down, or disabled.
DIO supply voltage (VIO)		5 V or 3.3 V, jumper selectable with jumper W3
Port read time		400 µs typ
Port write time		550 µs typ
Interrupt functionality		Each bit may be configured to generate an interrupt on change when in input mode.
Input low voltage threshold		0.3 x VIO V max
Input high voltage threshold		0.7 x VIO V min
Input voltage limits	Both 3.3 V and 5 V modes	6.5 V absolute max (**Note 2**) –0.5 V absolute min

Parameter	Conditions	Specification
Input voltage recommended range	5 V mode	5.5 V max 0 V min
	3.3 V mode	3.8 V max (**Note 2**) 0 V min
High level output current		10 mA max (**Note 3**)
Low level output current		25 mA max
Output high voltage	VIO = 3.3 V	2.5 V min (IOH = –10 mA)
	VIO = 5 V	4.0 V min (IOH = –10 mA)
Output low voltage	VIO = 3.3 V	0.25 V max (IOL = 10 mA)
	VIO = 5 V	0.2 V max (IOL = 10 mA)

Note 2: When VIO is 3.3V, the input will tolerate voltages up to 6.5 V, but the voltage must be current-limited or it will change the VIO voltage due to current flowing into the MCC 152. An external current limiting resistor of 700 Ω or larger is recommended on each input that is higher than 3.3V when the W3 jumper is in the 3.3V position.

Memory

Table 5. Memory specifications

Parameter	Specification
Non-volatile memory	4 KB (ID and serial storage, no user-modifiable memory)

Power

Table 6. Power specifications

Parameter	Conditions	Specification
Supply current, 5 V supply	Typical, 5V DIO selection	15 mA
	Maximum, 5V DIO selection	35 mA **(Note 4**, **Note 5**)
	Typical, 3.3V DIO selection	10 mA
	Maximum, 3.3V DIO selection	12 mA **(Note 4**)
Supply current, 3.3 V supply (**Note 3**)	Typical, 5V DIO selection	0.01 mA
	Maximum, 5V DIO selection	6 mA
	Typical, 3.3V DIO selection	3.5 mA
	Maximum, 3.3V DIO selection	11 mA **(Note 5**)

Note 3: The power consumed by all DAQ HATs must be within the capacity of the Raspberry Pi power supply. Extra care must be taken with sourcing 3.3 V loads since they are supplied by the regulator on the Raspberry Pi; MCC recommends using the 5 V DIO selection when sourcing large load currents such as LEDs.

Note 4: This specification does not include user loading on analog outputs.

Note 5: This specification does not include user loading on digital outputs or the VIO terminal.

Interface specifications

Table 7. Interface specifications

Parameter	Specification
Raspberry Pi GPIO pins used	GPIO 8, GPIO 10, GPIO 11 (SPI interface) GPIO 2, GPIO 3 (I2C interface) ID_SD, ID_SC (ID EEPROM) GPIO 12, GPIO 13, GPIO 26, (Board address) GPIO 21 (Interrupt)
Data interface type	SPI slave device, CE0 chip select (Analog output) I2C slave device (Digital I/O)
SPI mode	1

SPI clock rate	50 MHz max
I2C address	0x20 to 0x27, depending on board address jumper setting
I2C clock rate	400 kHz max

Environmental

Table 8. 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 9. 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 10. Screw terminal connector specifications

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

Table 11. Screw terminal pinout

Connector J2		
Pin	**Signal name**	**Pin description**
1	AO0	Analog output 0
2	AGND	Analog ground
3	AO1	Analog output 1
4	AGND	Analog ground
5	VIO	Digital supply voltage output (5V or 3.3V)
6	DGND	Digital ground
Connector J3		
Pin	**Signal name**	**Pin description**
7	DIO0	Digital I/O 0
8	DIO1	Digital I/O 1
9	DIO2	Digital I/O 2
10	DIO3	Digital I/O 3
11	DGND	Digital ground
12	DIO4	Digital I/O 4
13	DIO5	Digital I/O 5
14	DIO6	Digital I/O 6
15	DIO7	Digital I/O 7
16	DGND	Digital ground

CHAPTER

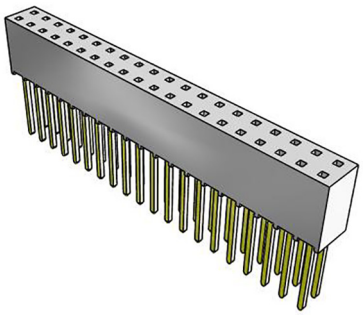
TWO

INSTALLING THE DAQ 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

Follow steps 1-6 in the single board installation procedure for the first HAT board.

1. **Connect all desired field wiring to the installed board - the screw terminals will not be accessible once additional boards are installed above it.**
2. 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.
3. 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.
4. 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

5. 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.
6. Repeat steps 1-5 for each board to be added.
7. 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/mccdaq/daqhats.

3.1 Installation

1. Power off the Raspberry Pi then attach one or more HAT boards (see *Installing the DAQ HAT board*).
2. Power on the Pi and log in. Open a terminal window if using the graphical interface.
3. Update your package list:

```
sudo apt-get update
```

4. **Optional:** Update your installed packages and reboot:

```
sudo apt-get dist-upgrade
sudo reboot
```

5. Install git (if not installed):

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

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

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

7. 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
```

8. **Optional:** Use the firmware update tool to update the firmware on your MCC 118 board(s). The "0" in the example below is the board address. Repeat the command for each MCC 118 address in your board stack. This example demonstrates how to update the firmware on the MCC 118 that is installed at address 0:

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

Note: If you encounter any errors during steps 5 - 7 then uininstall the daqhats library (if installed), go back to step 4, update your installed packages and reboot, then repeat steps 5 - 7.

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
```

You may display a list of the detected boards at any time with:

```
daqhats_list_boards
```

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 DAQ 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 DAQ HAT boards.
hat_error_message()	Return a text description for a DAQ HAT result.
hat_wait_for_interrupt()	Wait for an interrupt to occur.
hat_interrupt_state()	Read the current interrupt status.
hat_interrupt_callback_enable()	Enable an interrupt callback function.
hat_interrupt_callback_disable()	Disable interrupt callback function.

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

Return a list of detected DAQ HAT boards.

It creates the list from the DAQ HAT EEPROM files that are currently on the system. In the case of a single DAQ 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.

const char* **hat_error_message** (int *result*)
Return a text description for a DAQ HAT result code.

Return The error message.

Parameters

- `result`: The *Result code* returned from a DAQ HAT function

int **hat_wait_for_interrupt** (int *timeout*)
Wait for an interrupt to occur.

It waits for the interrupt signal to become active, with a timeout parameter.

Return *RESULT_TIMEOUT*, *RESULT_SUCCESS*, or *RESULT_UNDEFINED*.

Parameters

- `timeout`: Wait timeout in milliseconds. -1 to wait forever, 0 to return immediately.

int **hat_interrupt_state** (void)
Read the current interrupt status.

It returns the status of the interrupt signal. This signal can be shared by multiple boards so the status of each board that may generate must be read and the interrupt source(s) cleared before the interrupt will become inactive.

Return 1 if interrupt is active, 0 if inactive.

int **hat_interrupt_callback_enable** (void(**function*)(void *), void * *user_data*)
Enable an interrupt callback function.

Set a function that will be called when a DAQ HAT interrupt occurs. The function must have a void return type and void * argument, such as:

```
void function(void* user_data)
```

The function will be called when the DAQ HAT interrupt signal becomes active, and cannot be called again until the interrupt signal becomes inactive. Active sources become inactive when manually cleared (such as reading the digital I/O inputs or clearing the interrupt enable.) If not latched, an active source also becomes inactive when the value returns to the original value (the value at the source before the interrupt was generated.)

The user_data argument can be used for passing a reference to anything needed by the callback function. It will be passed to the callback function when the interrupt occurs. Set it to NULL if not needed.

There may only be one callback function at a time; if you call this when a function is already set as the callback function then it will be replaced with the new function and the old function will no longer be called if an interrupt occurs.

The callback function may be disabled with *hat_interrupt_callback_disable()*.

Return *RESULT_SUCCESS* or *RESULT_UNDEFINED*.

Parameters

- `function`: The callback function.
- `user_data`: The data to pass to the callback function.

int **hat_interrupt_callback_disable**(void)
Disable interrupt callbacks.

Removes any callback function from the interrupt handler.

Return *RESULT_SUCCESS* or *RESULT_UNDEFINED*.

4.1.2 Data types and definitions

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

4.1.2.1 HAT IDs

enum HatIDs
Known DAQ HAT IDs.

Values:

HAT_ID_ANY = 0
Match any DAQ HAT 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.

HAT_ID_MCC_152 = 0x0144
MCC 152 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_COMMS_FAILURE = -7
Could not communicate with the device.

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 Option Flags

See individual function documentation for detailed usage information.

OPTS_DEFAULT (0x0000)
Default behavior.

OPTS_NOSCALEDATA (0x0001)
Read / write unscaled data.

OPTS_NOCALIBRATEDATA (0x0002)
Read / write uncalibrated data.

OPTS_EXTCLOCK (0x0004)
Use an external clock source.

OPTS_EXTTRIGGER (0x0008)
Use an external trigger source.

OPTS_CONTINUOUS (0x0010)
Run until explicitly 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_info()	Return information about this device type.
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_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_status()	Read the scan status.
mcc118_a_in_scan_read()	Read scan data and 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).

struct *MCC118DeviceInfo** **mcc118_info**(void)
Return constant device information for all MCC 118s.

Return Pointer to struct *MCC118DeviceInfo*.

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

Passing 0 for count will result in the LED blinking continuously until the board is reset or *mcc118_blink_led()* is called again with a non-zero value for count.

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 active 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_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.

The valid options are:

- *OPTS_NOSCALEDATA*: Return ADC code (a value between 0 and 4095) rather than voltage.
- *OPTS_NOCALIBRATEDATA*: Return data without the calibration factors applied.

The options parameter is set to 0 or *OPTS_DEFAULT* for default operation, which is scaled and calibrated data.

Multiple options may be specified by ORing the flags. For instance, specifying *OPTS_NOSCALEDATA* | *OPTS_NOCALIBRATEDATA* will return the value read from the ADC without calibration or converting to voltage.

The function 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.
- `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 and rate. 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 scan state has defined terminology:

- **Active:** *mcc118_a_in_scan_start()* has been called and the device may be acquiring data or finished with the acquisition. The scan has not been cleaned up by calling *mcc118_a_in_scan_cleanup()*, so another scan may not be started.
- **Running:** The scan is active and the device is still acquiring data. Certain functions like *mcc118_a_in_read()* will return an error because the device is busy.

The valid options are:

- *OPTS_NOSCALEDATA*: Returns ADC code (a value between 0 and 4095) rather than voltage.

- *OPTS_NOCALIBRATEDATA*: Return data without the calibration factors applied.
- *OPTS_EXTCLOCK*: Use an external 3.3V or 5V logic signal at the CLK input as the scan clock. Multiple devices can be synchronized by connecting the CLK pins together and using this option on all but one device so they will be clocked by the single device using its internal clock. **sample_rate_per_channel** is only used for buffer sizing.
- *OPTS_EXTTRIGGER*: Hold off the scan (after calling *mcc118_a_in_scan_start()*) until the trigger condition is met. The trigger is a 3.3V or 5V logic signal applied to the TRIG pin.
- *OPTS_CONTINUOUS*: Scans continuously until stopped by the user by calling *mcc118_a_in_scan_stop()* and writes data to a circular buffer. The data must be read before being overwritten to avoid a buffer overrun error. **samples_per_channel** is only used for buffer sizing.

The options parameter is set to 0 or *OPTS_DEFAULT* for default operation, which is scaled and calibrated data, internal scan clock, no trigger, and finite operation.

Multiple options may be specified by ORing the flags. For instance, specifying *OPTS_NOSCALEDATA* | *OPTS_NOCALIBRATEDATA* will return the values read from the ADC without calibration or converting to voltage.

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 active this function will return *RESULT_BUSY*.

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

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 (finite mode,) or can be used to set a larger scan buffer size than the default value (continuous mode.)
- `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 bitmask.

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 active, *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_status**(uint8_t *address*, uint16_t * *status*, uint32_t * *samples_per_channel*)

Reads status and number of available 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 status of the scan and amount of data in the scan buffer.

Return *Result code*, *RESULT_SUCCESS* if successful, *RESULT_RESOURCE_UNAVAIL* if a scan has not been started 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 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`: Receives the number of samples per channel available in the scan thread buffer.

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 active.

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `status`: Receives the scan status, an ORed combination of the 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 in the scan thread buffer, ignoring **timeout**. 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 (up to the value of **samples_per_channel** or **buffer_size_samples**.)
- `buffer`: The user data buffer that receives the samples.
- `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.

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

This function returns 0 if no scan is active.

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 Device Info

struct MCC118DeviceInfo
MCC 118 constant device information.

Public Members

const uint8_t **NUM_AI_CHANNELS**
The number of analog input channels (8.)

const uint16_t **AI_MIN_CODE**
The minimum ADC code (0.)

const uint16_t **AI_MAX_CODE**
The maximum ADC code (4095.)

const double **AI_MIN_VOLTAGE**
The input voltage corresponding to the minimum code (-10.0V.)

const double **AI_MAX_VOLTAGE**
The input voltage corresponding to the maximum code (+10.0V - 1 LSB.)

const double **AI_MIN_RANGE**
The minimum voltage of the input range (-10.0V.)

const double **AI_MAX_RANGE**
The maximum voltage of the input range (+10.0V.)

4.2.2.2 Trigger Modes

enum TriggerMode
Scan trigger input modes.

Values:

TRIG_RISING_EDGE = 0
Start the scan on a rising edge of TRIG.

TRIG_FALLING_EDGE = 1
Start the scan on a falling edge of TRIG.

TRIG_ACTIVE_HIGH = 2
Start the scan any time TRIG is high.

TRIG_ACTIVE_LOW = 3
Start the scan any time TRIG is low.

4.2.2.3 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 (actively acquiring data.)

4.3 MCC 152 functions and data

4.3.1 Functions

Function	Description
mcc152_open()	Open an MCC 152 for use.
mcc152_is_open()	Check if an MCC 152 is open.
mcc152_close()	Close an MCC 152.
mcc152_info()	Return information about this device type.
mcc152_serial()	Read the serial number.
mcc152_a_out_write()	Write an analog output channel value.
mcc152_a_out_write_all()	Write all analog output channels simultaneously.
mcc152_dio_reset()	Reset the digital I/O to the default configuration.
mcc152_dio_input_read_bit()	Read a digital input.
mcc152_dio_input_read_port()	Read all digital inputs.
mcc152_dio_output_write_bit()	Write a digital output.
mcc152_dio_output_write_port()	Write all digital outputs.
mcc152_dio_output_read_bit()	Read the state of a digital output.
mcc152_dio_output_read_port()	Read the state of all digital outputs.
mcc152_dio_int_status_read_bit()	Read the interrupt status for a single channel.
mcc152_dio_int_status_read_port()	Read the interrupt status for all channels.
mcc152_dio_config_write_bit()	Write a digital I/O configuration item value for a single channel.
mcc152_dio_config_write_port()	Write a digital I/O configuration item value for all channels.
mcc152_dio_config_read_bit()	Read a digital I/O configuration item value for a single channel.
mcc152_dio_config_read_port()	Read a digital I/O configuration item value for all channels.

int **mcc152_open**(uint8_t *address*)
Open a connection to the MCC 152 device at the specified address.

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

Parameters

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

int **mcc152_is_open**(uint8_t *address*)
Check if an MCC 152 is open.

Return 1 if open, 0 if not open.

Parameters

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

int **mcc152_close**(uint8_t *address*)
Close a connection to an MCC 152 device and free allocated resources.

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

Parameters

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

struct *MCC152DeviceInfo** **mcc152_info**(void)
Return constant device information for all MCC 152s.

Return Pointer to struct *MCC152DeviceInfo*.

int **mcc152_serial**(uint8_t *address*, char * *buffer*)
Read the MCC 152 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 **mcc152_a_out_write**(uint8_t *address*, uint8_t *channel*, uint32_t *options*, double *value*)
Perform a write to an analog output channel.

Updates the analog output channel in either volts or DAC code (set the *OPTS_NOSCALEDATA* option to use DAC code.) The voltage must be 0.0 - 5.0 and DAC code 0.0 - 4095.0.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel`: The analog output channel number, 0 - 1.
- `options`: Options bitmask
- `value`: The analog output value.

int **mcc152_a_out_write_all**(uint8_t *address*, uint32_t *options*, double * *values*)
Perform a write to all analog output channels simultaneously.

Update all analog output channels in either volts or DAC code (set the *OPTS_NOSCALEDATA* option to use DAC code.) The outputs will update at the same time.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `options`: Options bitmask
- `values`: The array of analog output values; there must be at least 2 values, but only the first two values will be used.

int **mcc152_dio_reset**(uint8_t *address*)
Reset the digital I/O to the default configuration.

- All channels input
- Output registers set to 1
- Input inversion disabled
- No input latching

- Pull-up resistors enabled
- All interrupts disabled
- Push-pull output type

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

Parameters

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

int **mcc152_dio_input_read_bit**(uint8_t *address*, uint8_t *channel*, uint8_t * *value*)
Read a single digital input channel.

Returns 0 or 1 in **value**. If the specified channel is configured as an output this will return the value present at the terminal.

This function reads the entire input register, so care must be taken when latched inputs are enabled. If a latched input changes between input reads then changes back to its original value, the next input read will report the change to the first value then the following read will show the original value. If another input is read then this input change could be missed so it is best to use *mcc152_dio_input_read_port()* when using latched inputs.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel`: The DIO channel number, 0 - 7.
- `value`: Receives the input value.

int **mcc152_dio_input_read_port**(uint8_t *address*, uint8_t * *value*)
Read all digital input channels simultaneously.

Returns an 8-bit value in **value** representing all channels in channel order (bit 0 is channel 0, etc.) If a channel is configured as an output this will return the value present at the terminal.

Care must be taken when latched inputs are enabled. If a latched input changes between input reads then changes back to its original value, the next input read will report the change to the first value then the following read will show the original value.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `value`: Receives the input values.

int **mcc152_dio_output_write_bit**(uint8_t *address*, uint8_t *channel*, uint8_t *value*)
Write a single digital output channel.

If the specified channel is configured as an input this will not have any effect at the terminal, but allows the output register to be loaded before configuring the channel as an output.

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

Parameters

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

- `channel`: The DIO channel number, 0 - 7.
- `value`: The output value (0 or 1)

int **mcc152_dio_output_write_port**(uint8_t *address*, uint8_t *value*)
Write all digital output channels simultaneously.

Pass an 8-bit value in **value** representing the desired output for all channels in channel order (bit 0 is channel 0, etc.)

If the specified channel is configured as an input this will not have any effect at the terminal, but allows the output register to be loaded before configuring the channel as an output.

For example, to set channels 0 - 3 to 0 and channels 4 - 7 to 1 call:

```
mcc152_dio_output_write(address, 0xF0);
```

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

Parameters

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

int **mcc152_dio_output_read_bit**(uint8_t *address*, uint8_t *channel*, uint8_t * *value*)
Read a single digital output register.

Returns 0 or 1 in **value**.

This function returns the value stored in the output register. It may not represent the value at the terminal if the channel is configured as input or open-drain output.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel`: The DIO channel number, 0 - 7.
- `value`: Receives the output value.

int **mcc152_dio_output_read_port**(uint8_t *address*, uint8_t * *value*)
Read all digital output registers simultaneously.

Returns an 8-bit value in **value** representing all channels in channel order (bit 0 is channel 0, etc.)

This function returns the value stored in the output register. It may not represent the value at the terminal if the channel is configured as input or open-drain output.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `value`: Receives the output values.

int **mcc152_dio_int_status_read_bit**(uint8_t *address*, uint8_t *channel*, uint8_t * *value*)
Read the interrupt status for a single channel.

Returns 0 when the channel is not generating an interrupt, 1 when the channel is generating an interrupt.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel`: The DIO channel number, 0 - 7.
- `value`: Receives the interrupt status value.

int **mcc152_dio_int_status_read_port**(uint8_t *address*, uint8_t * *value*)
Read the interrupt status for all channels.

Returns an 8-bit value in **value** representing all channels in channel order (bit 0 is channel 0, etc.) A 0 in a bit indicates the corresponding channel is not generating an interrupt, a 1 indicates the channel is generating an interrupt.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `value`: Receives the interrupt status value.

int **mcc152_dio_config_write_bit**(uint8_t *address*, uint8_t *channel*, uint8_t *item*, uint8_t *value*)
Write a digital I/O configuration value for a single channel.

There are several configuration items that may be written for the digital I/O. The item is selected with the **item** argument, which may be one of the *DIOConfigItem* values:

- *DIO_DIRECTION*: Set the digital I/O channel direction by passing 0 for output and 1 for input.
- *DIO_PULL_CONFIG*: Configure the pull-up/down resistor by passing 0 for pull-down or 1 for pull-up. The resistor may be enabled or disabled with the *DIO_PULL_ENABLE* item.
- *DIO_PULL_ENABLE*: Enable or disable the pull-up/down resistor by passing 0 for disabled or 1 for enabled. The resistor is configured for pull-up/down with the *DIO_PULL_CONFIG* item.
- *DIO_INPUT_INVERT*: Enable inverting the input by passing a 0 for normal input or 1 for inverted.
- *DIO_INPUT_LATCH*: Enable input latching by passing 0 for non-latched or 1 for latched.

  When the input is non-latched, reads show the current status of the input. A state change in the input generates an interrupt (if it is not masked). A read of the input clears the interrupt. If the input goes back to its initial logic state before the input is read, then the interrupt is cleared.

  When the input is latched, a change of state of the input generates an interrupt and the input logic value is loaded into the input port register. A read of the input will clear the interrupt. If the input returns to its initial logic state before the input is read, then the interrupt is not cleared and the input register keeps the logic value that initiated the interrupt. The next read of the input will show the initial state.

  If the input is changed from latched to non-latched, a read from the input reflects the current terminal logic level. If the input is changed from non-latched to latched input, the read from the input represents the latched logic level.
- *DIO_OUTPUT_TYPE*: Set the output type by writing 0 for push-pull or 1 for open-drain. This setting affects all outputs so is not a per-channel setting and the channel argument will be ignored. It should be set to the desired type before using the *DIO_DIRECTION* item to set channels as outputs.
- *DIO_INT_MASK*: Enable or disable interrupt generation for the input by masking the interrupt. Write 0 to enable the interrupt or 1 to disable it.

All MCC 152s share a single interrupt signal to the CPU, so when an interrupt occurs the user must determine the source, optionally act on the interrupt, then clear that source so that other interrupts may be detected. The current interrupt state may be read with *hat_interrupt_state()*. A user program may wait for the interrupt to become active with *hat_wait_for_interrupt()*, or may register an interrrupt callback function with *hat_interrupt_callback_enable()*. This allows the user to wait for a change on one or more inputs without constantly reading the inputs. The source of the interrupt may be determined by reading the interrupt status of each MCC 152 with *mcc152_dio_int_status_read_bit()* or *mcc152_dio_int_status_read_port()*, and all active interrupt sources must be cleared before the interrupt will become inactive. The interrupt is cleared by reading the input(s) with *mcc152_dio_input_read_bit()* or *mcc152_dio_input_read_port()*.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel`: The digital I/O channel, 0 - 7.
- `item`: The config item, one of *DIOConfigItem*.
- `value`: The config value.

int **mcc152_dio_config_write_port** (uint8_t *address*, uint8_t *item*, uint8_t *value*)

Write a digital I/O configuration value for all channels.

There are several configuration items that may be written for the digital I/O. They are written for all channels at once using the 8-bit value passed in **value**, where each bit corresponds to a channel (bit 0 is channel 0, etc.) The item is selected with the **item** argument, which may be one of the *DIOConfigItem* values:

- *DIO_DIRECTION*: Set the digital I/O channel directions by passing 0 in a bit for output and 1 for input.
- *DIO_PULL_CONFIG*: Configure the pull-up/down resistors by passing 0 in a bit for pull-down or 1 for pull-up. The resistors may be enabled or disabled with the *DIO_PULL_ENABLE* item.
- *DIO_PULL_ENABLE*: Enable or disable pull-up/down resistors by passing 0 in a bit for disabled or 1 for enabled. The resistors are configured for pull-up/down with the *DIO_PULL_CONFIG* item.
- *DIO_INPUT_INVERT*: Enable inverting inputs by passing a 0 in a bit for normal input or 1 for inverted.
- *DIO_INPUT_LATCH*: Enable input latching by passing 0 in a bit for non-latched or 1 for latched.

  When the input is non-latched, reads show the current status of the input. A state change in the corresponding input generates an interrupt (if it is not masked). A read of the input clears the interrupt. If the input goes back to its initial logic state before the input is read, then the interrupt is cleared.

  When the input is latched, a change of state of the input generates an interrupt and the input logic value is loaded into the input port register. A read of the input will clear the interrupt. If the input returns to its initial logic state before the input is read, then the interrupt is not cleared and the input register keeps the logic value that initiated the interrupt. The next read of the input will show the initial state.

  If the input is changed from latched to non-latched, a read from the input reflects the current terminal logic level. If the input is changed from non-latched to latched input, the read from the input represents the latched logic level.
- *DIO_OUTPUT_TYPE*: Set the output type by writing 0 for push-pull or 1 for open-drain. This setting affects all outputs so is not a per-channel setting. It should be set to the desired type before using *DIO_DIRECTION* to set channels as outputs.
- *DIO_INT_MASK*: Enable or disable interrupt generation for specific inputs by masking the interrupts. Write 0 in a bit to enable the interrupt from that channel or 1 to disable it.

All MCC 152s share a single interrupt signal to the CPU, so when an interrupt occurs the user must determine the source, optionally act on the interrupt, then clear that source so that other interrupts may be detected. The current interrupt state may be read with *hat_interrupt_state()*. A user program may wait for the interrupt to become active with *hat_wait_for_interrupt()*, or may register an interrrupt callback function with *hat_interrupt_callback_enable()*. This allows the user to wait for a change on one or more inputs without constantly reading the inputs. The source of the interrupt may be determined by reading the interrupt status of each MCC 152 with *mcc152_dio_int_status_read_bit()* or *mcc152_dio_int_status_read_port()*, and all active interrupt sources must be cleared before the interrupt will become inactive. The interrupt is cleared by reading the input(s) with *mcc152_dio_input_read_bit()* or *mcc152_dio_input_read_port()*.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `item`: The config item, one of *DIOConfigItem*.
- `value`: The config value.

int **mcc152_dio_config_read_bit** (uint8_t *address*, uint8_t *channel*, uint8_t *item*, uint8_t * *value*)

Read a digital I/O configuration value for a single channel.

There are several configuration items that may be read for the digital I/O. The item is selected with the **item** argument, which may be one of the *DIOConfigItem* values:

- *DIO_DIRECTION*: Read the digital I/O channel direction setting, where 0 is output and 1 is input.
- *DIO_PULL_CONFIG*: Read the pull-up/down resistor configuration where 0 is pull-down and 1 is pull-up.
- *DIO_PULL_ENABLE*: Read the pull-up/down resistor enable setting where 0 is disabled and 1 is enabled.
- *DIO_INPUT_INVERT*: Read the input invert setting where 0 is normal input and 1 is inverted.
- *DIO_INPUT_LATCH*: Read the input latching setting where 0 is non-latched and 1 is latched.
- *DIO_OUTPUT_TYPE*: Read the output type setting where 0 is push-pull and 1 is open-drain. This setting affects all outputs so is not a per-channel setting and the channel argument is ignored.
- *DIO_INT_MASK*: Read the interrupt mask setting where 0 enables the interrupt and 1 disables it.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `channel`: The digital I/O channel, 0 - 7.
- `item`: The config item, one of *DIOConfigItem*.
- `value`: Receives the config value.

int **mcc152_dio_config_read_port** (uint8_t *address*, uint8_t *item*, uint8_t * *value*)

Read a digital I/O configuration value for all channels.

There are several configuration items that may be read for the digital I/O. They are read for all channels at once, returning an 8-bit value in **value**, where each bit corresponds to a channel (bit 0 is channel 0, etc.) The item is selected with the **item** argument, which may be one of the *DIOConfigItem* values:

- *DIO_DIRECTION*: Read the digital I/O channels direction settings, where 0 for a bit is output and 1 is input.

- *DIO_PULL_CONFIG*: Read the pull-up/down resistor configurations where 0 for a bit is pull-down and 1 is pull-up.
- *DIO_PULL_ENABLE*: Read the pull-up/down resistor enable settings where 0 for a bit is disabled and 1 is enabled.
- *DIO_INPUT_INVERT*: Read the input invert settings where 0 for a bit is normal input and 1 is inverted.
- *DIO_INPUT_LATCH*: Read the input latching settings where 0 for a bit is non-latched and 1 is latched.
- *DIO_OUTPUT_TYPE*: Read the output type setting where 0 is push-pull and 1 is open-drain. This setting affects all outputs so is not a per-channel setting.
- *DIO_INT_MASK*: Read the interrupt mask settings where 0 enables the interrupt from the corresponding channel and 1 disables it.

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

Parameters

- `address`: The board address (0 - 7). Board must already be opened.
- `item`: The config item, one of *DIOConfigItem*.
- `value`: Receives the config value.

4.3.2 Data types and definitions

4.3.2.1 Device Info

struct MCC152DeviceInfo
MCC 152 constant device information.

Public Members

const uint8_t **NUM_DIO_CHANNELS**
The number of digital I/O channels (8.)

const uint8_t **NUM_AO_CHANNELS**
The number of analog output channels (2.)

const uint16_t **AO_MIN_CODE**
The minimum DAC code (0.)

const uint16_t **AO_MAX_CODE**
The maximum DAC code (4095.)

const double **AO_MIN_VOLTAGE**
The output voltage corresponding to the minimum code (0.0V.)

const double **AO_MAX_VOLTAGE**
The output voltage corresponding to the maximum code (+5.0V - 1 LSB.)

const double **AO_MIN_RANGE**
The minimum voltage of the output range (0.0V.)

const double **AO_MAX_RANGE**
The maximum voltage of the output range (+5.0V.)

4.3.2.2 DIO Config Items

enum DIOConfigItem
DIO Configuration Items.

Values:

DIO_DIRECTION = 0
Configure channel direction.

DIO_PULL_CONFIG = 1
Configure pull-up/down resistor.

DIO_PULL_ENABLE = 2
Enable pull-up/down resistor.

DIO_INPUT_INVERT = 3
Configure input inversion.

DIO_INPUT_LATCH = 4
Configure input latching.

DIO_OUTPUT_TYPE = 5
Configure output type.

DIO_INT_MASK = 6
Configure interrupt mask.

CHAPTER

FIVE

PYTHON LIBRARY REFERENCE

The Python library is organized as a global method for listing the DAQ 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 DAQ HAT boards.
interrupt_state()	Read the current DAQ HAT interrupt status.
wait_for_interrupt()	Wait for a DAQ HAT interrupt to occur.
interrupt_callback_enable()	Enable an interrupt callback function.
interrupt_callback_disable()	Disable interrupt callback function.

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

Return a list of detected DAQ HAT boards.

Scans certain locations for information from the HAT EEPROMs. Verifies the contents are valid HAT EEPROM contents and returns a list of namedtuples containing information on the HAT. Info will only be returned for DAQ 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 DAQ HATs found. Otherwise, return only DAQ HATs with ID matching this value.

Returns

A list of namedtuples, the number of elements match the number of DAQ HATs found. Each namedtuple will contain the following field names

- **address** (int): device address
- **id** (int): device product ID, identifies the type of DAQ HAT
- **version** (int): device hardware version
- **product_name** (str): device product name

Return type list

daqhats.**interrupt_state**()

Read the current DAQ HAT interrupt status

Returns the status of the interrupt signal, True if active or False if inactive. The signal can be shared by multiple DAQ HATs so the status of each board that may generate an interrupt must be read and the interrupt source(s) cleared before the interrupt will become inactive.

Returns The interrupt status.

Return type bool

daqhats.**wait_for_interrupt**(*timeout*)
Wait for an interrupt from a DAQ HAT to occur.

Pass a timeout in seconds. Pass -1 to wait forever or 0 to return immediately. If the interrupt has not occurred before the timeout elapses the function will return 0.

Returns The interrupt status - True = interrupt active, False = interrupt inactive.

Return type bool

daqhats.**interrupt_callback_enable**(*callback*, *user_data*)
Enable an interrupt callback function.

Set a function that will be called when a DAQ HAT interrupt occurs.

The function will be called when the DAQ HAT interrupt signal becomes active, and cannot be called again until the interrupt signal becomes inactive. Active sources become inactive when manually cleared (such as reading the digital I/O inputs or clearing the interrupt enable.) If not latched, an active source also becomes inactive when the value returns to the original value (the value at the source before the interrupt was generated.)

There may only be one callback function at a time; if you call this when a function is already set as the callback function then it will be replaced with the new function and the old function will no longer be called if an interrupt occurs. The data argument to this function will be passed to the callback function when it is called.

The callback function must be encapsulated in a *HatCallback* class. For example:

```
def my_function(data):
    # This is my callback function.
    print("The interrupt occurred, and returned {}.".format(data))
    data[0] += 1

value = [0]
callback = HatCallback(my_function)
interrupt_enable_callback(callback, value)
```

In this example *my_function()* will be called when the interrupt occurs, and the list *value* will be passed as the user_data. Inside the callback it will be received as *data*, but will still be the same object so any changes made will be present in the original *value*. Every time the interrupt occurs *value[0]* will be incremented and a higher number will be printed.

An integer was not used for *value* because integers are immutable in Python so the original *value* would never change.

The callback may be disabled with *interrupt_callback_disable()*.

Parameters

- **callback** (*HatCallback*) – The callback function encapsulated in a *HatCallback* class.
- **user_data** (*object*) – callback function.

Raises Exception – Internal error enabling the callback.

daqhats.**interrupt_callback_disable**()
Disable interrupt callbacks.

Raises `Exception` – Internal error disabling the callback.

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

MCC_152 = 324
MCC 152 ID

5.1.2.2 Trigger modes

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

RISING_EDGE = 0
Start the scan on a rising edge of TRIG.

FALLING_EDGE = 1
Start the scan on a falling edge of TRIG.

ACTIVE_HIGH = 2
Start the scan any time TRIG is high.

ACTIVE_LOW = 3
Start the scan any time TRIG is low.

5.1.2.3 Scan / read option flags

class daqhats.**OptionFlags**
Scan / read option flags. See individual methods for detailed descriptions.

DEFAULT = 0
Use default behavior.

NOSCALEDATA = 1
Read / write unscaled data.

NOCALIBRATEDATA = 2
Read / write uncalibrated data.

EXTCLOCK = 4
Use an external clock source.

EXTTRIGGER = 8
Use an external trigger source.

CONTINUOUS = 16
Run until explicitly stopped.

5.1.3 HatError class

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

Parameters

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

5.1.4 HatCallback class

class daqhats.**HatCallback**(*function*)
DAQ HAT interrupt callback function class.

This class handles passing Python functions to the shared library as a callback then retrieving the passed user data as a Python object when the function is called from the library.

The callback function should have a single argument (optional) that is a Python object the user provides to *interrupt_callback_enable()* that will be passed to the callback function whenever it is called.

Parameters **function** (*function*) – the function to be called when an interrupt occurs.

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.info()	Get info about this device type.
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_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.

static info()
Return constant information about this type of device.

Returns

a namedtuple containing the following field names

- **NUM_AI_CHANNELS** (int): The number of analog input channels (8.)
- **AI_MIN_CODE** (int): The minimum ADC code (0.)
- **AI_MAX_CODE** (int): The maximum ADC code (4095.)
- **AI_MIN_VOLTAGE** (float): The voltage corresponding to the minimum ADC code (-10.0.)
- **AI_MAX_VOLTAGE** (float): The voltage corresponding to the maximum ADC code (+10.0 - 1 LSB)
- **AI_MIN_RANGE** (float): The minimum voltage of the input range (-10.0.)
- **AI_MAX_RANGE** (float): The maximum voltage of the input range (+10.0.)

Return type namedtuple

firmware_version()
Read the board firmware and bootloader versions.

Returns

a namedtuple containing the following field names

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

Return type namedtuple

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.

Setting count to 0 will cause the LED to blink continuously until blink_led() is called again with a non-zero count.

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

a namedtuple containing the following field names

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

Return type namedtuple

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 active 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*: Start the scan when the TRIG input transitions from low to high.
- *TriggerModes.FALLING_EDGE*: Start the scan when the TRIG input transitions from high to low.
- *TriggerModes.ACTIVE_HIGH*: Start the scan when the TRIG input is high.
- *TriggerModes.ACTIVE_LOW*: Start the scan 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.

a_in_read(*channel*, *options=<OptionFlags.DEFAULT: 0>*)
Perform a single reading of an analog input channel and return the value.

options is an ORed combination of OptionFlags. Valid flags for this method are:

- *OptionFlags.DEFAULT*: Return a calibrated voltage value. Any other flags will override DEFAULT behavior.
- *OptionFlags.NOSCALEDATA*: Return an ADC code (a value between 0 and 4095) rather than voltage.
- *OptionFlags.NOCALIBRATEDATA*: Return data without the calibration factors applied.

Parameters

- **channel** (*int*) – The analog input channel number, 0-7.
- **options** (*int*) – ORed combination of *OptionFlags*, *OptionFlags.DEFAULT* if unspecified.

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*, *options*)
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 state has defined terminology:

- **Active**: *a_in_scan_start()* has been called and the device may be acquiring data or finished with the acquisition. The scan has not been cleaned up by calling *a_in_scan_cleanup()*, so another scan may not be started.
- **Running**: The scan is active and the device is still acquiring data. Certain methods like *a_in_read()* will return an error because the device is busy.

The scan options that may be used are:

- *OptionFlags.DEFAULT*: Return scaled and calibrated data, internal scan clock, no trigger, and finite operation. Any other flags will override DEFAULT behavior.
- *OptionFlags.NOSCALEDATA*: Return ADC codes (values between 0 and 4095) rather than voltage.
- *OptionFlags.NOCALIBRATEDATA*: Return data without the calibration factors applied.
- *OptionFlags.EXTCLOCK*: Use an external 3.3V or 5V logic signal at the CLK input as the scan clock. Multiple devices can be synchronized by connecting the CLK pins together and using this flag on all but one device so they will be clocked by the single device using its internal clock. **sample_rate_per_channel** is only used for buffer sizing.
- *OptionFlags.EXTTRIGGER*: Hold off the scan (after calling *a_in_scan_start()*) until the trigger condition is met. The trigger is a 3.3V or 5V logic signal applied to the TRIG pin.
- *OptionFlags.CONTINUOUS*: Scans continuously until stopped by the user by calling *a_in_scan_stop()* and writes data to a circular buffer. The data must be read before being overwritten to avoid a buffer overrun error. **samples_per_channel** is only used for buffer sizing.

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 active this method 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 (finite mode,) or or can be used to set a larger scan buffer size than the default value (continuous mode.)
- **sample_rate_per_channel** (*float*) – The per-channel rate of the internal scan clock, or the expected maximum rate of an external scan clock, max 100,000.0.
- **options** (*int*) – An ORed combination of *OptionFlags* flags that control the scan.

Raises

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

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 active).

a_in_scan_status()

Read scan status and number of available samples per channel.

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 the number of samples per channel available in the scan thread buffer.

Returns

a namedtuple containing the following field names:

- **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.
- **samples_available** (int): The number of samples per channel currently in the scan buffer.

Return type namedtuple

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

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 a negative number to return all available samples immediately and ignore **timeout** 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 (up to **samples_per_channel**.) If the timeout is met and the specified number of samples have not been read, then the function will return all the available samples and the timeout status set.

Returns

a namedtuple containing the following field names:

- **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 namedtuple

Raises

- *HatError* – A scan is not active, the board is not initialized, does not respond, or responds incorrectly.
- `ValueError` – Incorrect argument.

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 namedtuple 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 a negative number 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 namedtuple containing the following field names:

- **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 namedtuple

Raises

- *HatError* – A scan is not active, the board is not initialized, does not respond, or responds incorrectly.
- ValueError – Incorrect argument.

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 active, 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 device stops acquiring data 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.

5.3 MCC 152 class

5.3.1 Methods

class daqhats.**mcc152**(*address=0*)
The class for an MCC 152 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
mcc152.info()	Get info about this device type.
mcc152.serial()	Read the serial number.
mcc152.a_out_write()	Write an analog output channel.
mcc152.a_out_write_all()	Write all analog output channels.
mcc152.dio_reset()	Reset the digital I/O to the default configuration.
mcc152.dio_input_read_bit()	Read a digital input.
mcc152.dio_input_read_port()	Read all digital inputs.
mcc152.dio_input_read_tuple()	Read all digital inputs as a tuple.
mcc152.dio_output_write_bit()	Write a digital output.
mcc152.dio_output_write_port()	Write all digital outputs.
mcc152.dio_output_write_dict()	Write multiple digital outputs with a dictionary.
mcc152.dio_output_read_bit()	Read the state of a digital output.
mcc152.dio_output_read_port()	Read the state of all digital outputs.
mcc152.dio_output_read_tuple()	Read the state of all digital outputs as a tuple.
mcc152.dio_int_status_read_bit()	Read the interrupt status for a single channel.
mcc152.dio_int_status_read_port()	Read the interrupt status for all channels.
mcc152.dio_int_status_read_tuple()	Read the interrupt status for all channels as a tuple.
mcc152.dio_config_write_bit()	Write a digital I/O configuration item value for a single channel.
mcc152.dio_config_write_port()	Write a digital I/O configuration item value for all channels.
mcc152.dio_config_write_dict()	Write a digital I/O configuration item value for multiple channels.
mcc152.dio_config_read_bit()	Read a digital I/O configuration item value for a single channel.
mcc152.dio_config_read_port()	Read a digital I/O configuration item value for all channels.
mcc152.dio_config_read_tuple()	Read a digital I/O configuration item value for all channels as a tuple.

static info()

Return constant information about this type of device.

Returns

a namedtuple containing the following field names

- **NUM_DIO_CHANNELS** (int): The number of digital I/O channels (8.)
- **NUM_AO_CHANNELS** (int): The number of analog output channels (2.)
- **AO_MIN_CODE** (int): The minimum DAC code (0.)
- **AO_MAX_CODE** (int): The maximum DAC code (4095.)
- **AO_MIN_VOLTAGE** (float): The voltage corresponding to the minimum DAC code (0.0.)
- **AO_MAX_VOLTAGE** (float): The voltage corresponding to the maximum DAC code (+5.0 - 1 LSB)
- **AO_MIN_RANGE** (float): The minimum voltage of the output range (0.0.)
- **AO_MAX_RANGE** (float): The maximum voltage of the output range (+5.0.)

Return type namedtuple

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.

a_out_write(*channel*, *value*, *options=<OptionFlags.DEFAULT: 0>*)
Write a single analog output channel value. The value will be limited to the range of the DAC without raising an exception.

options is an OptionFlags value. Valid flags for this method are:

- *OptionFlags.DEFAULT*: Write a voltage value (0 - 5).
- *OptionFlags.NOSCALEDATA*: Write a DAC code (a value between 0 and 4095) rather than voltage.

Parameters

- **channel** (*int*) – The analog output channel number, 0-1.
- **value** (*float*) – The value to write.
- **options** (*int*) – An *OptionFlags* value, *OptionFlags.DEFAULT* if unspecified.

Raises

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

a_out_write_all(*values*, *options=<OptionFlags.DEFAULT: 0>*)
Write all analog output channels simultaneously.

options is an OptionFlags value. Valid flags for this method are:

- *OptionFlags.DEFAULT*: Write voltage values (0 - 5).
- *OptionFlags.NOSCALEDATA*: Write DAC codes (values between 0 and 4095) rather than voltage.

Parameters

- **values** (*list of float*) – The values to write, in channel order. There must be at least two values, but only the first two will be used.
- **options** (*int*) – An *OptionFlags* value, *OptionFlags.DEFAULT* if unspecified.

Raises

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

dio_reset()
Reset the DIO to the default configuration.

- All channels input
- Output registers set to 1

- Input inversion disabled
- No input latching
- Pull-up resistors enabled
- All interrupts disabled
- Push-pull output type

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

dio_input_read_bit(*channel*)
Read a single digital input channel.

Returns 0 if the input is low, 1 if it is high.

If the specified channel is configured as an output this will return the value present at the terminal.

This method reads the entire input register even though a single channel is specified, so care must be taken when latched inputs are enabled. If a latched input changes between input reads then changes back to its original value, the next input read will report the change to the first value then the following read will show the original value. If another input is read then this input change could be missed so it is best to use *dio_input_read_port()* or *dio_input_read_tuple()* when using latched inputs.

Parameters **channel** (*int*) – The DIO channel number, 0-7.

Returns The input value.

Return type int

Raises

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

dio_input_read_port()
Read all digital input channels.

Returns the values as an integer with a value of 0 - 255. Each channel is represented by a bit in the integer (bit 0 is channel 0, etc.)

The value of a specific input can be found by examining the bit at that location. For example, to act on the channel 3 input:

```
inputs = mcc152.dio_input_read_port()
if (inputs & (1 << 3)) == 0:
    print("channel 3 is 0")
else:
    print("channel 3 is 1")
```

If a channel is configured as an output this will return the value present at the terminal.

Care must be taken when latched inputs are enabled. If a latched input changes between input reads then changes back to its original value, the next input read will report the change to the first value then the following read will show the original value.

Returns int: The input values.

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

dio_input_read_tuple()
Read all digital inputs at once as a tuple.

Returns a tuple of all input values in channel order. For example, to compare the channel 1 input to the channel 3 input:

```
inputs = mcc152.dio_input_read_tuple()
if inputs[1] == inputs[3]:
    print("channel 1 and channel 3 inputs are the same")
```

If a channel is configured as an output this will return the value present at the terminal.

Care must be taken when latched inputs are enabled. If a latched input changes between input reads then changes back to its original value, the next input read will report the change to the first value then the following read will show the original value.

Returns tuple of int: The input values.

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

dio_output_write_bit(*channel*, *value*)
Write a single digital output channel.

If the specified channel is configured as an input this will not have any effect at the terminal, but allows the output register to be loaded before configuring the channel as an output.

Parameters

- **channel** (*int*) – The digital channel number, 0-7.
- **value** (*int*) – The output value, 0 or 1.

Raises

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

dio_output_write_port(*values*)
Write all digital output channel values.

Pass an integer in the range of 0 - 255, where each bit represents the value of the associated channel (bit 0 is channel 0, etc.) If a specified channel is configured as an input this will not have any effect at the terminal, but allows the output register to be loaded before configuring the channel as an output.

To change specific outputs without affecting other outputs first read the output values with dio_output_read_port(), change the desired bits in the result, then write them back.

For example, to set channels 0 and 2 to 1 without affecting the other outputs:

```
values = mcc152.dio_output_read_port()
values |= 0x05
mcc152.dio_output_write_port(values)
```

Parameters **values** (*integer*) – The output values.

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

dio_output_write_dict(*value_dict*)
Write multiple digital output channel values.

Pass a dictionary containing channel:value pairs. If a channel is repeated in the dictionary then the last value will be used. If a specified channel is configured as an input this will not have any effect at the terminal, but allows the output register to be loaded before configuring the channel as an output.

For example, to set channels 0 and 2 to 1 without affecting the other outputs:

```
values = { 0:1, 2:1 }
mcc152.dio_output_write_dict(values)
```

Parameters **value_dict** (*dictionary*) – The output values in a dictionary of channel:value pairs.

Raises

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

dio_output_read_bit(*channel*)

Read a single digital output channel value.

This function returns the value stored in the output register. It may not represent the value at the terminal if the channel is configured as input or open-drain output.

Parameters **channel** (*int*) – The digital channel number, 0-7.

Returns The output value.

Return type int

Raises

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

dio_output_read_port()

Read all digital output channel values.

Returns the values as an integer with a value of 0 - 255. Each channel is represented by a bit in the integer (bit 0 is channel 0, etc.) The value of a specific output can be found by examining the bit at that location.

This function returns the values stored in the output register. They may not represent the value at the terminal if the channel is configured as input or open-drain output.

Returns int: The output values.

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

dio_output_read_tuple()

Read all digital output channel values at once as a tuple.

Returns a tuple of all output values in channel order. For example, to compare the channel 1 output to the channel 3 output:

```
outputs = mcc152.dio_output_read_tuple()
if outputs[1] == outputs[3]:
    print("channel 1 and channel 3 outputs are the same")
```

This function returns the values stored in the output register. They may not represent the value at the terminal if the channel is configured as input or open-drain output.

Returns tuple of int: The output values.

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

dio_int_status_read_bit(*channel*)
Read the interrupt status for a single channel.

Returns 0 if the input is not generating an interrupt, 1 if it is generating an interrupt.

Returns int: The interrupt status value.

Raises

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

dio_int_status_read_port()
Read the interrupt status for all channels.

Returns the values as an integer with a value of 0 - 255. Each channel is represented by a bit in the integer (bit 0 is channel 0, etc.) The status for a specific input can be found by examining the bit at that location. Each bit will be 0 if the channel is not generating an interrupt or 1 if it is generating an interrupt.

Returns int: The interrupt status values.

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

dio_int_status_read_tuple()
Read the interrupt status for all channels as a tuple.

Returns a tuple of all interrupt status values in channel order. Each value will be 0 if the channel is not generating an interrupt or 1 if it is generating an interrupt. For example, to see if an interrupt has occurred on channel 2 or 4:

```
status = mcc152.dio_int_status_read_tuple()
if status[2] == 1 or status[4] == 1:
    print("an interrupt has occurred on channel 2 or 4")
```

Returns tuple of int: The status values.

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

dio_config_write_bit(*channel*, *item*, *value*)
Write a digital I/O configuration value for a single channel.

There are several configuration items that may be written for the digital I/O. The item is selected with the **item** argument, which may be one of the *DIOConfigItem* values:

- *DIOConfigItem.DIRECTION*: Set the digital I/O channel direction by passing 0 for output and 1 for input.
- *DIOConfigItem.PULL_CONFIG*: Configure the pull-up/down resistor by passing 0 for pull-down or 1 for pull-up. The resistor may be enabled or disabled with the *DIOConfigItem.PULL_ENABLE* item.
- *DIOConfigItem.PULL_ENABLE*: Enable or disable the pull-up/down resistor by passing 0 for disabled or 1 for enabled. The resistor is configured for pull-up/down with the *DIOConfigItem.PULL_CONFIG* item.
- *DIOConfigItem.INPUT_INVERT*: Enable inverting the input by passing a 0 for normal input or 1 for inverted.

- *DIOConfigItem.INPUT_LATCH*: Enable input latching by passing 0 for non-latched or 1 for latched.

  When the input is non-latched, reads show the current status of the input. A state change in the input generates an interrupt (if it is not masked). A read of the input clears the interrupt. If the input goes back to its initial logic state before the input is read, then the interrupt is cleared.

  When the input is latched, a change of state of the input generates an interrupt and the input logic value is loaded into the input port register. A read of the input will clear the interrupt. If the input returns to its initial logic state before the input is read, then the interrupt is not cleared and the input register keeps the logic value that initiated the interrupt. The next read of the input will show the initial state.

  If the input is changed from latched to non-latched, a read from the input reflects the current terminal logic level. If the input is changed from non-latched to latched input, the read from the input represents the latched logic level.

- *DIOConfigItem.OUTPUT_TYPE*: Set the output type by writing 0 for push-pull or 1 for open-drain. This setting affects all outputs so is not a per-channel setting and the channel argument is ignored. It should be set to the desired type before using *DIOConfigItem.DIRECTION* item to set channels as outputs.

- *DIOConfigItem.INT_MASK*: Enable or disable interrupt generation by masking the interrupt. Write 0 to enable the interrupt or 1 to mask (disable) it.

  All MCC 152s share a single interrupt signal to the CPU, so when an interrupt occurs the user must determine the source, optionally act on the interrupt, then clear that source so that other interrupts may be detected. The current interrupt state may be read with *interrupt_state()*. A user program may wait for the interrupt to become active with *wait_for_interrupt()*, or may register an interrupt callback function with *interrupt_callback_enable()*. This allows the user to wait for a change on one or more inputs without constantly reading the inputs. The source of the interrupt may be determined by reading the interrupt status of each MCC 152 with *dio_int_status_read_bit()*, *dio_int_status_read_port()* or *dio_int_status_read_tuple()*, and all active interrupt sources must be cleared before the interrupt will become inactive. The interrupt is cleared by reading the input with *dio_input_read_bit()*, *dio_input_read_port()*, or *dio_input_read_tuple()*.

  Parameters

  - **channel** (*integer*) – The digital I/O channel, 0 - 7
  - **item** (*integer*) – The configuration item, one of *DIOConfigItem*.
  - **value** (*integer*) – The configuration value.

  Raises

  - *HatError* – the board is not initialized, does not respond, or responds incorrectly.
  - `ValueError` – an argument is invalid.

dio_config_write_port (*item*, *value*)

Write a digital I/O configuration value for all channels.

There are several configuration items that may be written for the digital I/O. They are written for all channels at once using an 8-bit value passed in **value**, where each bit corresponds to a channel (bit 0 is channel 0, etc.) The item is selected with the **item** argument, which may be one of the *DIOConfigItem* values.

- *DIOConfigItem.DIRECTION*: Set the digital I/O channel directions by passing 0 in a bit for output and 1 for input.

- *DIOConfigItem.PULL_CONFIG*: Configure the pull-up/down resistors by passing 0 in a bit for pull-down or 1 for pull-up. The resistors may be enabled or disabled with the *DIOConfigItem.PULL_ENABLE* item.
- *DIOConfigItem.PULL_ENABLE*: Enable or disable pull-up/down resistors by passing 0 in a bit for disabled or 1 for enabled. The resistors are configured for pull-up/down with the *DIOConfigItem.PULL_CONFIG* item.
- *DIOConfigItem.INPUT_INVERT*: Enable inverting inputs by passing a 0 in a bit for normal input or 1 for inverted.
- *DIOConfigItem.INPUT_LATCH*: Enable input latching by passing 0 in a bit for non-latched or 1 for latched.

  When the input is non-latched, reads show the current status of the input. A state change in the corresponding input generates an interrupt (if it is not masked). A read of the input clears the interrupt. If the input goes back to its initial logic state before the input is read, then the interrupt is cleared. When the input is latched, a change of state of the input generates an interrupt and the input logic value is loaded into the input port register. A read of the input will clear the interrupt. If the input returns to its initial logic state before the input is read, then the interrupt is not cleared and the input register keeps the logic value that initiated the interrupt. The next read of the input will show the initial state.

  If the input is changed from latched to non-latched, a read from the input reflects the current terminal logic level. If the input is changed from non-latched to latched input, the read from the input represents the latched logic level.
- *DIOConfigItem.OUTPUT_TYPE*: Set the output type by writing 0 for push-pull or 1 for open-drain. This setting affects all outputs so is not a per-channel setting. It should be set to the desired type before using *DIOConfigItem.DIRECTION* to set channels as outputs.
- *DIOConfigItem.INT_MASK*: Enable or disable interrupt generation for specific inputs by masking the interrupt. Write 0 in a bit to enable the interrupt from that channel or 1 to mask (disable) it.

  All MCC 152s share a single interrupt signal to the CPU, so when an interrupt occurs the user must determine the source, optionally act on the interrupt, then clear that source so that other interrupts may be detected. The current interrupt state may be read with *interrupt_state()*. A user program may wait for the interrupt to become active with *wait_for_interrupt()*, or may register an interrupt callback function with *interrupt_callback_enable()*. This allows the user to wait for a change on one or more inputs without constantly reading the inputs. The source of the interrupt may be determined by reading the interrupt status of each MCC 152 with *dio_int_status_read_bit()*, *dio_int_status_read_port()* or *dio_int_status_read_tuple()*, and all active interrupt sources must be cleared before the interrupt will become inactive. The interrupt is cleared by reading the input with *dio_input_read_bit()*, *dio_input_read_port()*, or *dio_input_read_tuple()*.

  Parameters

  - **item** (*integer*) – The configuration item, one of *DIOConfigItem*.
  - **value** (*integer*) – The configuration value.

  Raises

  - *HatError* – the board is not initialized, does not respond, or responds incorrectly.
  - `ValueError` – an argument is invalid.

dio_config_write_dict(*item*, *value_dict*)

Write a digital I/O configuration value for multiple channels.

There are several configuration items that may be written for the digital I/O. They are written for multiple channels at once using a dictionary of channel:value pairs. If a channel is repeated in the dictionary then the last value will be used. The item is selected with the **item** argument, which may be one of the *DIOConfigItem* values:

- *DIOConfigItem.DIRECTION*: Set the digital I/O channel directions by passing 0 in a value for output and 1 for input.
- *DIOConfigItem.PULL_CONFIG*: Configure the pull-up/down resistors by passing 0 in a value for pull-down or 1 for pull-up. The resistors may be enabled or disabled with the *DIOConfigItem.PULL_ENABLE* item.
- *DIOConfigItem.PULL_ENABLE*: Enable or disable pull-up/down resistors by passing 0 in a value for disabled or 1 for enabled. The resistors are configured for pull-up/down with the *DIOConfigItem.PULL_CONFIG* item.
- *DIOConfigItem.INPUT_INVERT*: Enable inverting inputs by passing a 0 in a value for normal input or 1 for inverted.
- *DIOConfigItem.INPUT_LATCH*: Enable input latching by passing 0 in a value for non-latched or 1 for latched.

  When the input is non-latched, reads show the current status of the input. A state change in the corresponding input generates an interrupt (if it is not masked). A read of the input clears the interrupt. If the input goes back to its initial logic state before the input is read, then the interrupt is cleared. When the input is latched, a change of state of the input generates an interrupt and the input logic value is loaded into the input port register. A read of the input will clear the interrupt. If the input returns to its initial logic state before the input is read, then the interrupt is not cleared and the input register keeps the logic value that initiated the interrupt. The next read of the input will show the initial state.

  If the input is changed from latched to non-latched, a read from the input reflects the current terminal logic level. If the input is changed from non-latched to latched input, the read from the input represents the latched logic level.
- *DIOConfigItem.OUTPUT_TYPE*: Set the output type by writing 0 for push-pull or 1 for open-drain. This setting affects all outputs so is not a per-channel setting. It should be set to the desired type before using *DIOConfigItem.DIRECTION* to set channels as outputs.
- *DIOConfigItem.INT_MASK*: Enable or disable interrupt generation for specific inputs by masking the interrupt. Write 0 in a value to enable the interrupt from that channel or 1 to mask (disable) it.

  All MCC 152s share a single interrupt signal to the CPU, so when an interrupt occurs the user must determine the source, optionally act on the interrupt, then clear that source so that other interrupts may be detected. The current interrupt state may be read with *interrupt_state()*. A user program may wait for the interrupt to become active with *wait_for_interrupt()*, or may register an interrupt callback function with *interrupt_callback_enable()*. This allows the user to wait for a change on one or more inputs without constantly reading the inputs. The source of the interrupt may be determined by reading the interrupt status of each MCC 152 with *dio_int_status_read_bit()*, *dio_int_status_read_port()* or *dio_int_status_read_tuple()*, and all active interrupt sources must be cleared before the interrupt will become inactive. The interrupt is cleared by reading the input with *dio_input_read_bit()*, *dio_input_read_port()*, or *dio_input_read_tuple()*.

For example, to set channels 6 and 7 to output:

```
values = { 6:0, 7:0 }
mcc152.dio_config_write_dict(DIOConfigItem.DIRECTION, values)
```

Parameters

- **item** (*integer*) – The configuration item, one of *DIOConfigItem*.
- **value_dict** (*dictionary*) – The configuration values for multiple channels in a dictionary of channel:value pairs.

Raises

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

dio_config_read_bit (*channel*, *item*)

Read a digital I/O configuration value for a single channel.

There are several configuration items that may be read for the digital I/O. The item is selected with the **item** argument, which may be one of the DIOConfigItems values:

- *DIOConfigItem.DIRECTION*: Read the digital I/O channel direction setting, where 0 is output and 1 is input.
- *DIOConfigItem.PULL_CONFIG*: Read the pull-up/down resistor configuration where 0 is pull-down and 1 is pull-up.
- *DIOConfigItem.PULL_ENABLE*: Read the pull-up/down resistor enable setting where 0 is disabled and 1 is enabled.
- *DIOConfigItem.INPUT_INVERT*: Read the input inversion setting where 0 is normal input and 1 is inverted.
- *DIOConfigItem.INPUT_LATCH*: Read the input latching setting where 0 is non-latched and 1 is latched.
- *DIOConfigItem.OUTPUT_TYPE*: Read the output type setting where 0 is push-pull and 1 is open-drain. This setting affects all outputs so is not a per-channel setting and the channel argument is ignored.
- *DIOConfigItem.INT_MASK*: Read the interrupt mask setting where 0 in a bit enables the interrupt and 1 disables it.

Parameters

- **channel** (*integer*) – The digital I/O channel, 0 - 7.
- **item** (*integer*) – The configuration item, one of *DIOConfigItem*.

Returns int: The configuration item value.

Raises

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

dio_config_read_port (*item*)

Read a digital I/O configuration value for all channels.

There are several configuration items that may be read for the digital I/O. They are read for all channels at once, returning an 8-bit integer where each bit corresponds to a channel (bit 0 is channel 0, etc.) The item is selected with the **item** argument, which may be one of the *DIOConfigItem* values:

- *DIOConfigItem.DIRECTION*: Read the digital I/O channels direction settings, where 0 in a bit is output and 1 is input.
- *DIOConfigItem.PULL_CONFIG*: Read the pull-up/down resistor configurations where 0 in a bit is pull-down and 1 is pull-up.
- *DIOConfigItem.PULL_ENABLE*: Read the pull-up/down resistor enable settings where 0 in a bit is disabled and 1 is enabled.
- *DIOConfigItem.INPUT_INVERT*: Read the input inversion settings where 0 in a bit is normal input and 1 is inverted.
- *DIOConfigItem.INPUT_LATCH*: Read the input latching settings where 0 in a bit is non-latched and 1 is latched.
- *DIOConfigItem.OUTPUT_TYPE*: Read the output type setting where 0 is push-pull and 1 is open-drain. This setting affects all outputs so is not a per-channel setting.
- *DIOConfigItem.INT_MASK*: Read the interrupt mask settings where 0 in a bit enables the interrupt and 1 disables it.

Parameters **item** (*integer*) – The configuration item, one of *DIOConfigItem*.

Returns int: The configuration item value.

Raises

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

dio_config_read_tuple(*item*)

Read a digital I/O configuration value for all channels as a tuple.

There are several configuration items that may be read for the digital I/O. They are read for all channels at once, returning a tuple in channel order. The item is selected with the **item** argument, which may be one of the *DIOConfigItem* values:

- *DIOConfigItem.DIRECTION*: Read the digital I/O channels direction settings, where 0 in a value is output and 1 is input.
- *DIOConfigItem.PULL_CONFIG*: Read the pull-up/down resistor configurations where 0 in a value is pull-down and 1 is pull-up.
- *DIOConfigItem.PULL_ENABLE*: Read the pull-up/down resistor enable settings where 0 in a value is disabled and 1 is enabled.
- *DIOConfigItem.INPUT_INVERT*: Read the input inversion settings where 0 in a value is normal input and 1 is inverted.
- *DIOConfigItem.INPUT_LATCH*: Read the input latching settings where 0 in a value is non-latched and 1 is latched.
- *DIOConfigItem.OUTPUT_TYPE*: Read the output type setting where 0 is push-pull and 1 is open-drain. This setting affects all outputs so is not a per-channel setting.
- *DIOConfigItem.INT_MASK*: Read the interrupt mask settings where 0 in a value enables the interrupt and 1 disables it.

Parameters **item** (*integer*) – The configuration item, one of *DIOConfigItem*.

Returns tuple of int: The configuration item values.

Raises

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

address()
Return the device address.

5.3.2 Data

5.3.2.1 DIO Config Items

class daqhats.**DIOConfigItem**
Digital I/O Configuration Items.

DIRECTION = 0
Configure channel direction

PULL_CONFIG = 1
Configure pull-up/down resistor

PULL_ENABLE = 2
Enable pull-up/down resistor

INPUT_INVERT = 3
Configure input inversion

INPUT_LATCH = 4
Configure input latching

OUTPUT_TYPE = 5
Configure output type

INT_MASK = 6
Configure interrupt mask

INDEX

M

N

O

P

R

S

T

W