

Users's Manual for Device Interfaces

Brent Seidel
Phoenix, AZ

September 2, 2024

This document is ©2024, Brent Seidel. All rights reserved.

Note that this is a draft version and not the final version for publication.

Contents

1	Introduction	1
1.1	About the Project	1
1.2	License	1
2	How to Obtain	2
2.1	Dependencies	2
2.1.1	bbs_embed_common	2
2.1.2	bbs_embed_linux	3
2.1.3	bbs_embed_due	3
3	Usage Instructions	5
3.1	Linux Based Raspberry Pi and BeagleBone Black	5
3.2	Arduino Due	5
4	Common API Description	6
4.1	Basic Devices	6
4.1.1	Analog Inputs	7
4.1.2	General-Purpose Input/Output (GPIO)	7
4.1.3	I2C Bus	7
4.1.4	Logging	10
4.1.5	SPI Bus	10
4.2	Higher-Level Device Drivers	11
4.2.1	BBS.embed.gpio.tb6612	11
4.2.2	BBS.embed.I2C.ADS1015	12
4.2.3	BBS.embed.i2c.BME280	16
4.2.4	BBS.embed.i2c.BMP180	18
4.2.5	BBS.embed.i2c.L3GD20H	20
4.2.6	BBS.embed.i2c.LSM303DLHC	24
4.2.7	BBS.embed.i2c.MCP4725	29
4.2.8	BBS.embed.i2c.MCP23008	30
4.2.9	BBS.embed.i2c.MCP23017	31
4.2.10	BBS.embed.i2c.PCA9685	34
4.2.11	BBS.embed.SPI.RA8875	36

5	Linux API Description	37
5.1	Common	37
5.2	Raspberry Pi	37
5.3	BeagleBone Black	37
6	Arduino Due API Description	38
7	Other Stuff	39
	Bibliography	40

List of Tables

4.1	Constants for ADS1015 Mux Mode Configuration	12
4.2	Constants for ADS1015 Programmable Gain Amplifier Configuration	12
4.3	Constants for ADS1015 Data Rate Configuration	13
4.4	Constants for ADS1015 Comparator Queue Configuration	13
4.5	Constants for ADS1015 Miscellaneous Configuration	13
4.6	Constants for BMP180 Conversion Kinds	18
4.7	Constants for L3GD20H Full-Scale Deflection	20
4.8	Constants for L3GD20H Status	21
4.9	Constants for LSM303DLHC Full-Scale Accelerometer	24
4.10	Constants for LSM303DLHC Accelerometer Status	24
4.11	Constants for LSM303DLHC Full-Scale Magnetometer	24
4.12	Constants for LSM303DLHC Magnetometer Status	24
4.13	Constants for MCP4725 Commands	29
4.14	Constants for MCP4725 Power-Down Modes	30
4.15	Measured Values for PCA9685 Controlling Servos	34

Chapter 1

Introduction

1.1 About the Project

This project provides an interface to hardware available on some Linux based systems and the Arduino Due. It consists of two main components: First an abstract set of classes for certain generic hardware items, and second specific classes to interface with the hardware on specific devices. This separation is done to ease porting of software between different devices. The two Linux based devices that are currently supported are the Raspberry Pi and the BeagleBone Black. Other devices may be added by creating a set of specific classes for the device.

1.2 License

This project is licensed using the GNU General Public License V3.0. Should you wish other licensing terms, contact the author.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Chapter 2

How to Obtain

This collection is currently available on GitHub at <https://github.com/BrentSeidel/BBS-BBB-Ada>. Parts are available through alire via “`alr get bbs_embed_common`” and “`alr get bbs_embed_linux`”

2.1 Dependencies

2.1.1 bbs_embed_common

Ada Libraries

The following Ada packages are used:

- `Ada.Integer_Text_IO`
- `Ada.Numerics.Generic_Elementary_Functions` (used only by `lsm303dlhc`)
- `Ada.Real_Time`
- `Ada.Text_IO`
- `Ada.Unchecked_Conversion`

Other Libraries

This library depends on the root package BBS available at <https://github.com/BrentSeidel/BBS-Ada> and through alire via “`alr get bbs`”. Packages external to this library are marked with an asterisk.

- `BBS.embed.GPIO`
- `BBS.embed.i2c`
- `BBS.embed.log`
- `BBS.embed.SPI`
- `BBS.units*`

2.1.2 bbs_embed_linux

Ada Libraries

The following Ada packages are used:

- Ada.Direct_IO
- Ada.IO_Exceptions
- Ada.Long_Integer_Text_IO
- Ada.Strings.Fixed
- Ada.Text_IO
- Interfaces.C

Other Libraries

This library depends on the root package BBS available at <https://github.com/BrentSeidel/BBS-Ada> and through alire via “`alr get bbs`”. Packages external to this library are marked with an asterisk.

- BBS.embed*
- BBS.embed.BBB*
- BBS.embed.GPIO*
- BBS.embed.log*
- BBS.embed.SPI*
- BBS.units*

2.1.3 bbs_embed_due

The Arduino Due requires an appropriate run-time system and cross-compiler.

Ada Libraries

The following Ada packages are used:

- Ada.Interrupts
- Ada.Interrupts.Names
- Ada.Real_Time
- Ada.Synchronous_Task_Control
- Interfaces
- System
- System.Sam3x8

SAM3x8e Stuff

The following SAM3x8e hardware definition packages are used:

- SAM3x8e
- SAM3x8e.ADC
- SAM3x8e.PIO
- SAM3x8e.PMC
- SAM3x8e.TWI
- SAM3x8e.UART

Other Libraries

This library depends on the root package BBS available at <https://github.com/BrentSeidel/BBS-Ada> and through alire via “`alr get bbs`”. Packages external to this library are marked with an asterisk.

- BBS*
- BBS.embed*
- BBS.embed.due.dev
- BBS.embed.due.serial.int
- BBS.embed.due.serial.polled
- BBS.embed.GPIO.Due
- BBS.embed.log*
- BBS.embed.SPI*

Chapter 3

Usage Instructions

This chapter contains high-level instructions on using this library in your project. First, all projects will need to include the `bbs_embed_common` packages to gain access to the base classes and some device drivers that build on these base classes. The second step is platform specific, as described below.

3.1 Linux Based Raspberry Pi and BeagleBone Black

You will need to include the `bbs_embed_linux` packages in your project. The `BBS.embed.rpi` package contains constants for various device names available on the Raspberry Pi. The `BBS.embed.BBB` package contains constants for various devices names on the BeagleBone Black. There is a script, `init-bbb.sh` for the BeagleBone Black or `init-rpi.sh` for the Raspberry Pi that needs to be run to activate some of the devices and set protections on the device files. The script will need to be run as superuser, using the `sudo` command. Among other things, it sets protection on the device files so that your software does not need to run as superuser.

3.2 Arduino Due

This has not been worked on for a while. To use this, you will need an ARM ELF Ada compiler and a board support package for the Arduino Due. It did work with a board support package that I'd cobbled together a few generations of gnat ago. Consider this to be experimental, but it should provide a good start to accessing hardware on the Arduino Due.

Chapter 4

Common API Description

Dealing with hardware can be complex, especially if you want your software to be portable. The various different boards have different devices (or different numbers of devices) available. Sometimes options are available on one board that are not available on another.

The common library (`bbs_embed_common` in `alire`) contains base classes for hardware devices and higher-level drivers for devices that attach to the basic hardware, for example devices that connect via an I2C bus.

4.1 Basic Devices

The package `BBS.embed` defines the following types and functions:

```
type addr7 is mod 2**7
  with size => 7;
type int12 is range -(2**11) .. 2**11 - 1
  with size => 12;
type uint12 is mod 2**12
  with size => 12;
```

The `addr7` is used for addressing devices on an I2C bus. the `int12` and `uint12` are used for the return values from typical analog to digital converters and anywhere else a 12 bit number is needed.

```
function uint12_to_int12 is
  new Ada.Unchecked_Conversion(source => uint12, target => int12);
```

This is used to convert from unsigned to signed 12 bit integers. Should the reverse conversion be needed, it would be easy enough to add it here.

```
function highByte(x : uint16) return uint8 is
  (uint8(x / 2**8));
function lowByte(x : uint16) return uint8 is
  (uint8(x and 16#FF#));
```

These are used to extract the MSB and LST from `uint16` values.

4.1.1 Analog Inputs

Analog inputs have one common routine to read the value. Everything else is implementation dependent.

```
function get(self : AIN_record) return uint12 is abstract;
```

Read the value of the specified analog to digital converter.

- *self* - The object for the analog input device.
- Returns the 12-bit value from the analog to digital converter.

4.1.2 General-Purpose Input/Output (GPIO)

A GPIO is a device capable of reading or writing a single bit. The physical characteristics are hardware dependent. Some device specific routines may be needed to convert between input and output.

```
procedure set(self : GPIO_record; value : bit) is abstract;
```

Sets the output value of a GPIO device. The effect if the device is set to input is device specific.

- *self* - The object for the GPIO device.
- *value* - The value to write to the GPIO device.

```
function get(self : GPIO_record) return bit is abstract;
```

Reads the value of a GPIO device. The value returned if the device is set to output is device specific.

- *self* - The object for the GPIO device.
- Returns the value read from the GPIO device.

4.1.3 I2C Bus

An I2C bus can interface with a number of devices on the bus. It operates with the CPU being the master and the addressed device responding. The basic I2C bus uses 7 bit addressing for devices and operates at 100kHz. Any other options (10 bit addressing or higher speeds would be device specific, if supported).

The `BBS.embed.i2c` defines some datatypes. The ones for external use are:

- `err_code` is an enumeration of error statuses that can be returned. The possible values are `none`, `nack`, `ovre`, `invalid_addr`, and `failed`. In most cases you'll just want to compare the returned error to `none`.
- `buff_index` is an `Integer` index into a buffer with a range of 0 .. 127.
- `buffer` is an array of `uint8` and bounds of `buff_index`. It is used for buffering data for I2C bus transfers.

The following routines are used for communicating with devices on the I2C bus. Note that there is no standard about whether multibyte data should be transferred LSB first or MSB first (I've even seen devices that use both depending on which data you're getting). Routines are provided for MSB first (m1 routines) or MSB second (m2 routines) for 16 bit transfers. For longer transfers, use the block transfer routines and decode the data yourself. The 8 and 16 bit routines cover most of the cases.

```
function read(self : in out i2c_interface_record; addr : addr7; reg : uint8;
              error : out err_code) return uint8 is (0);
```

Read a single byte of data from the specified register in the specified device.

- *self* - The I2C interface device to use for communication.
- *addr7* - The 7 bit address of the device to communicate with.
- *reg* - The register address in the device.
- *error* - The error code from the transaction.
- Returns the register contents.

```
function readm1(self : in out i2c_interface_record; addr : addr7; reg : uint8;
                error : out err_code) return uint16 is (0);
```

Read two bytes of data with MSB transferred first from the specified register in the specified device.

- *self* - The I2C interface device to use for communication.
- *addr7* - The 7 bit address of the device to communicate with.
- *reg* - The register address in the device.
- *error* - The error code from the transaction.
- Returns the register contents.

```
function readm2(self : in out i2c_interface_record; addr : addr7; reg : uint8;
                error : out err_code) return uint16 is (0);
```

Read two bytes of data with MSB transferred second from the specified register in the specified device.

- *self* - The I2C interface device to use for communication.
- *addr7* - The 7 bit address of the device to communicate with.
- *reg* - The register address in the device.
- *error* - The error code from the transaction.
- Returns the register contents.

```
procedure read(self : in out i2c_interface_record; addr : addr7; reg : uint8;  
              size : buff_index; error : out err_code) is null;
```

Reads a block of data into the interface record's buffer. The user's code will need to extract the data from that buffer and process it as needed.

- *self* - The I2C interface device to use for communication.
- *addr7* - The 7 bit address of the device to communicate with.
- *reg* - The register address in the device.
- *size* - The number of bytes to transfer.
- *error* - The error code from the transaction.

```
procedure write(self : in out i2c_interface_record; addr : addr7; reg : uint8;  
               data : uint8; error : out err_code) is null;
```

Write a single byte of data to the specified register in the specified device.

- *self* - The I2C interface device to use for communication.
- *addr7* - The 7 bit address of the device to communicate with.
- *reg* - The register address in the device.
- *data* - The data to write.
- *error* - The error code from the transaction.

```
procedure writem1(self : in out i2c_interface_record; addr : addr7; reg : uint8;  
                  data : uint16; error : out err_code) is null;
```

Writes two bytes of data with the MSB transferred first to the specified register in the specified device.

- *self* - The I2C interface device to use for communication.
- *addr7* - The 7 bit address of the device to communicate with.
- *reg* - The register address in the device.
- *data* - The data to write.
- *error* - The error code from the transaction.

```
procedure writem2(self : in out i2c_interface_record; addr : addr7; reg : uint8;  
                  data : uint16; error : out err_code) is null;
```

Writes two bytes of data with the MSB transferred second to the specified register in the specified device.

- *self* - The I2C interface device to use for communication.

- *addr7* - The 7 bit address of the device to communicate with.
- *reg* - The register address in the device.
- *data* - The data to write.
- *error* - The error code from the transaction.

```
procedure write(self : in out i2c_interface_record; addr : addr7; reg : uint8;  
               size : buff_index; error : out err_code) is null;
```

Send the specified number of bytes in the interface record's buffer to the specified device and register.

- *self* - The I2C interface device to use for communication.
- *addr7* - The 7 bit address of the device to communicate with.
- *reg* - The register address in the device.
- *size* - The number of bytes to transfer.
- *error* - The error code from the transaction.

In most cases these routines should only be used when writing a driver for an I2C device.

4.1.4 Logging

This is only for debugging purposes. Should say something here briefly.

4.1.5 SPI Bus

The exposed interface for the SPI bus is much simpler than for the I2C bus. If needed, more routines may be added here, but this hasn't been developed as much as the I2C bus interface.

```
procedure set(self : SPI_record; value : uint8) is abstract;
```

Writes a byte to the SPI bus.

- *self* - The SPI interface device to use for communication.
- *data* - The data to write.

```
function get(self : SPI_record) return uint8 is abstract;
```

Reads a byte from the SPI bus.

- *self* - The SPI interface device to use for communication.
- Returns the byte read from the bus.

4.2 Higher-Level Device Drivers

These higher-level device drivers build on the lower-level devices. Typically these are devices that attach to a databus.

4.2.1 BBS.embed.gpio.tb6612

This is a driver for the Toshiba TB6612 dual DC motor controller [11]. The device driver is designed to sequence the output to drive a stepper motor, or it can control two DC motors separately. It requires four GPIO output pins.

```
procedure init(self : in out TB6612_record; pin_a : BBS.embed.GPIO.GPIO;  
               pin_b : BBS.embed.GPIO.GPIO; pin_c : BBS.embed.GPIO.GPIO;  
               pin_d : BBS.embed.GPIO.GPIO);
```

Initialize the TB6612 driver with the 4 GPIO devices.

- *self* - The TB6612 device to initialize.
- *pin_a* - The first GPIO pin.
- *pin_b* - The second GPIO pin.
- *pin_c* - The third GPIO pin.
- *pin_d* - The fourth GPIO pin.

```
procedure set_delay(self : in out TB6612_record; wait_time : Natural);
```

Set a time delay between steps to use when stepping the motor a number of steps. If not needed, it can be set to zero.

- *self* - The TB6612 device to modify.
- *wait_time* - The time delay between steps in mS.

```
procedure step(self : in out TB6612_record; steps : Integer);
```

Move the stepper motor a specified number of steps. A negative number will move in the opposite direction as a positive number. Zero steps will do nothing. If no delay has been specified, a default of 5mS between steps will be used.

- *self* - The TB6612 device to modify.
- *steps* - The number of steps to move the motor.

```
procedure stepper_off(self : in out TB6612_record);
```

De-energize the coils for a stepper motor (or both DC motors).

- *self* - The TB6612 device to modify.

```
procedure set_bridge_a(self : in out TB6612_record; value : Integer);
procedure set_bridge_b(self : in out TB6612_record; value : Integer);
```

Each of the two H-bridges can be controlled separately. This would allow two DC motors to be driven.

- *self* - The TB6612 device to modify.
- *value* - A value of zero sets the bridge off. A positive value sets the polarity in one direction. A negative value sets the polarity in the opposite direction.

Note that polarities are not specified as they depend on how the hardware is wired.

4.2.2 BBS.embed.I2C.ADS1015

This is a driver for the Texas Instruments ADS1015 4 channel analog to digital convertor that attaches to an I2C bus [3]. This converter has a wide variety of configuration options. Refer to the datasheet for details. A number of constants have been defined to support the various configuration options (see Tables 4.1, 4.2, 4.3, and 4.4).

Constant	Mux Mode Configuration
<i>mux_a0_a1</i>	Difference between AIN0 and AIN1 (default)
<i>mux_a0_a3</i>	Difference between AIN0 and AIN3
<i>mux_a1_a3</i>	Difference between AIN1 and AIN3
<i>mux_a2_a3</i>	Difference between AIN2 and AIN3
<i>mux_a0_gnd</i>	Single ended AIN0 value
<i>mux_a1_gnd</i>	Single ended AIN1 value
<i>mux_a2_gnd</i>	Single ended AIN2 value
<i>mux_a3_gnd</i>	Single ended AIN3 value

Table 4.1: Constants for ADS1015 Mux Mode Configuration

Constant	PGA Configuration
<i>pga_6_144</i>	Full scale voltage is 6.144V
<i>pga_4_096</i>	Full scale voltage is 4.096V
<i>pga_2_048</i>	Full scale voltage is 2.048V (default)
<i>pga_1_024</i>	Full scale voltage is 1.024V
<i>pga_0_512</i>	Full scale voltage is 0.512V
<i>pga_0_256</i>	Full scale voltage is 0.256V

Table 4.2: Constants for ADS1015 Programmable Gain Amplifier Configuration

There are some additional configuration parameters that don't have constants defined. These just have values of 0 or 1. See Table 4.5.

The datatype `ADS1015_config` is defined as a record containing the configuration values. It has the following fields:

Constant	Data Rate in Samples per Second (S/S)
<i>dr_0128</i>	Data rate is 128S/S
<i>dr_0250</i>	Data rate is 250S/S
<i>dr_0490</i>	Data rate is 490S/S
<i>dr_0920</i>	Data rate is 920S/S
<i>dr_1600</i>	Data rate is 1600S/S (default)
<i>dr_2400</i>	Data rate is 2400S/S
<i>dr_3300</i>	Data rate is 3300S/S

Table 4.3: Constants for ADS1015 Data Rate Configuration

Constant	Comparator Queue Configuration
<i>comp_que_1</i>	Assert after one conversion
<i>comp_que_2</i>	Assert after two conversions
<i>comp_que_3</i>	Assert after three conversion
<i>comp_que_d</i>	Disable comparator (default)

Table 4.4: Constants for ADS1015 Comparator Queue Configuration

Value	Conversion Mode
0	Continuous conversion mode
1	Power-down single-shot mode (default)
Value	Comparator Mode
0	Traditional, with hysteresis (default)
1	Window comparator
Value	Comparator Polarity
0	Active low (default)
1	Active high
Value	Comparator Latching
0	Non-latching comparator (default)
1	Latching comparator

Table 4.5: Constants for ADS1015 Miscellaneous Configuration

- **os** - Operational status, used to start a conversion if in single shot mode. Don't use when setting configuration.
- **mux** - The mux mode (see Table 4.1).
- **pga** - The programmable gain type (see Table 4.2).
- **mode** - Conversion mode (see Table 4.5).
- **dr** - The data rate (see Table 4.3).
- **comp_mode** - The comparator mode (see Table 4.5).

- `comp_pol` - The comparator polarity (see Table 4.5).
- `comp_lat` - The comparator latching (see Table 4.5).
- `comp_que` - The comparator queue configuration (see Table 4.4).

```
procedure configure(self : in out ADS1015_record; port : i2c_interface;  
                  addr : addr7; error : out err_code);
```

Initializes the device to the default configuration.

- *self* - The device to initialize.
- *port* - The I2C interface that the device is connected to.
- *addr* - The I2C address of the device.
- *error* - The I2C error code.

```
procedure configure(self : in out ADS1015_record; port : i2c_interface;  
                  addr : addr7; config : ADS1015_config; error : out err_code);
```

Initialize the device using the specified configuration.

- *self* - The device to initialize.
- *port* - The I2C interface that the device is connected to.
- *addr* - The I2C address of the device.
- *config* - A configuration record containing the desired configuration
- *error* - The I2C error code.

```
procedure change_config(self : in out ADS1015_record;  
                       config : ADS1015_config; error : out err_code);
```

Changes the device configuration to new values

- *self* - The device to modify.
- *config* - A configuration record containing the desired configuration
- *error* - The I2C error code.

```
procedure set_mux(self : in out ADS1015_record;  
                 mux : mux_mode_type; error : out err_code);
```

Changes only the mux mode configuration.

- *self* - The device to modify.
- *mux* - The new mux mode configuration value.

- *error* - The I2C error code.

```
procedure set_gain(self : in out ADS1015_record;  
                  gain : pga_type; error : out err_code);
```

Changes only the converter gain value.

- *self* - The device to modify.
- *gain* - The new gain value.
- *error* - The I2C error code.

```
procedure set_continuous(self : in out ADS1015_record; error : out err_code);
```

Sets the converter to operate in continuous mode.

- *self* - The device to modify.
- *error* - The I2C error code.

```
procedure set_1shot(self : in out ADS1015_record; error : out err_code);
```

Sets the converter to operate in single shot mode.

- *self* - The device to modify.
- *error* - The I2C error code.

```
procedure start_conversion(self : in out ADS1015_record; error : out err_code);
```

Start a conversion when in single shot mode. No effect in continuous mode.

- *self* - The device to modify.
- *error* - The I2C error code.

```
function conversion_done(self : in out ADS1015_record; error : out err_code)  
    return Boolean;
```

Checks if conversion is in progress. Will always return *False* (conversion in progress) while in continuous mode. Returns *True* when no conversion is in progress.

- *self* - The device to initialize.
- *error* - The I2C error code.
- Returns a conversion in progress flag.

```
function get_result(self : in out ADS1015_record; error : out err_code)  
    return uint12;
```

Returns the conversion value.

- *self* - The device to initialize.
- *error* - The I2C error code.
- Returns the conversion value

4.2.3 BBS.embed.i2c.BME280

This is a driver for the Bosch BME280 temperature, pressure, and humidity sensor that attaches to an I2C bus [2]. A number of constants are defined, but most of them are intended only for internal use. The constant *addr* is the I2C address of the BME280 sensor and is intended for use in the *configure* call.

```
procedure configure(self : in out BME280_record; port : i2c_interface;  
                  addr : addr7; error : out err_code);
```

Called to configure a BME280 device. This needs to be called before the device can be used.

- *self* - The BME280 device to configure.
- *port* - The I2C bus object that the BME280 is connected to.
- *addr* - The I2C address of the device.
- *error* - The error code from any I2C transactions.

```
procedure start_conversion(self : BME280_record; error : out err_code);
```

Instruct the BME280 to start converting temperature, pressure, and humidity readings. These are converted at the same time.

- *self* - The BME280 device to instruct.
- *error* - The error code from any I2C transactions.

```
function data_ready(self : BME280_record; error : out err_code) return boolean;
```

Checks to see if conversion is complete. The user software should wait until conversion is complete before attempting to read otherwise the results will be undefined.

- *self* - The BME280 device to instruct.
- *error* - The error code from any I2C transactions.
- Returns *True* if the conversion is complete and *False* otherwise.

```
procedure read_data(self : in out BME280_record; error : out err_code);
```

Instructs the BME280 to read the converted temperature, pressure, and humidity values into BME280 object and compute calibrated values. There is less overhead to read all three at once.

- *self* - The BME280 device to instruct.
- *error* - The error code from any I2C transactions.

```
procedure get_raw(self : BME280_record; raw_temp : out uint32;  
                 raw_press : out uint32; raw_hum : out uint32);
```

Return the raw, uncompensated values after *read_data()* has been called. This is primarily for debugging purposes.

- *self* - The BME280 device to instruct.
- *raw_temp* - The raw temperature value.
- *raw_press* - The raw pressure value.
- *raw_hum* - The raw humidity value.

```
function get_t_fine(self : BME280_record) return int32;
```

Returns the `t_fine` value after `read_data()` has been called. This is primarily for debugging purposes.

- *self* - The BME280 device to instruct.
- Returns the `t_fine` value.

```
function get_temp(self : BME280_record) return integer;
```

Returns the calibrated temperature value as an **Integer**. The LSB unit is 0.01°C.

- *self* - The BME280 device to instruct.
- Returns the temperature in units of 0.01°C.

```
function get_temp(self : BME280_record) return BBS.units.temp_c;
```

```
function get_temp(self : BME280_record) return BBS.units.temp_f;
```

```
function get_temp(self : BME280_record) return BBS.units.temp_k;
```

Returns the temperature in units of °C, °F, or K, depending on datatype of the destination.

- *self* - The BME280 device to instruct.
- Returns the temperature in units of °C, °F, or K.

```
function get_press(self : BME280_record) return integer;
```

Returns the calibrated pressure value as an **Integer**. The LSB unit is $\frac{1}{256}$ Pa.

- *self* - The BME280 device to instruct.
- Returns the pressure in units of $\frac{1}{256}$ Pa.

```
function get_press(self : BME280_record) return BBS.units.press_p;
```

```
function get_press(self : BME280_record) return BBS.units.press_mb;
```

```
function get_press(self : BME280_record) return BBS.units.press_atm;
```

```
function get_press(self : BME280_record) return BBS.units.press_inHg;
```

Returns that pressure in units of Pa, mB, Atm, or inHg, depending on the datatype of the destination.

- *self* - The BME280 device to instruct.

- Returns the pressure in units of Pa, mB, Atm, or inHg.

```
function get_hum(self : BME280_record) return integer;
```

Returns the calibrated relative humidity as an **Integer**. The LSB unit is $\frac{1}{1024}\%$ humidity.

- *self* - The BME280 device to instruct.
- Returns the humidity in units of $\frac{1}{1024}\%$ humidity.

```
function get_hum(self : BME280_record) return float;
```

Returns the relative humidity as a percentage relative humidity.

- *self* - The BME280 device to instruct.
- Returns the humidity as a percentage relative humidity.

4.2.4 BBS.embed.i2c.BMP180

This is a driver for the Bosch BMP180 temperature and pressure sensor that attaches to an I2C bus [1]. It has been discontinued by Bosch and is not recommended for new projects. A number of constants are defined, but most of them are intended only for internal use. The constant *addr* is the I2C address of the BMP180 sensor and is intended for use in the **configure** call.

Constant	Conversion Kind
<i>cvt_temp</i>	Convert Temperature
<i>cvt_press0</i>	Convert pressure with no oversampling
<i>cvt_press1</i>	Convert pressure with oversampling of two
<i>cvt_press2</i>	Convert pressure with oversampling of four
<i>cvt_press3</i>	Convert pressure with oversampling of eight

Table 4.6: Constants for BMP180 Conversion Kinds

```
procedure configure(self : in out BMP180_record; port : i2c_interface;
                  addr : addr7; error : out err_code);
```

Called to configure a BMP180 device. This needs to be called before the device can be used.

- *self* - The BMP180 device to configure.
- *port* - The I2C bus object that the BMP180 is connected to.
- *addr* - The I2C address of the device.
- *error* - The error code from any I2C transactions.

```
procedure start_conversion(self : in out BMP180_record;
                          kind : uint8; error : out err_code);
```

Instruct the BMP180 to start converting temperature or pressure.

- *self* - The BMP180 device to instruct.
- *kind* - The kind of conversion to start. See Table 4.6 for options.
- *error* - The error code from any I2C transactions.

```
function data_ready(self : BMP180_record; error : out err_code)
    return boolean;
```

Checks to see if conversion is complete. The user software should wait until conversion is complete before attempting to read otherwise the results will be undefined.

- *self* - The BMP180 device to instruct.
- *error* - The error code from any I2C transactions.
- Returns *True* if the conversion is complete and *False* otherwise.

```
function get_temp(self : in out BMP180_record; error : out err_code)
    return float;
```

Returns the calibrated temperature value as a **Float** in °C.

- *self* - The BMP180 device to instruct.
- *error* - The error code from any I2C transactions.
- Returns the temperature as a **Float** in °C.

```
function get_temp(self : in out BMP180_record; error : out err_code)
    return integer;
```

Returns the calibrated temperature value as an **Integer**. The LSB unit is 0.1°C.

- *self* - The BMP180 device to instruct.
- *error* - The error code from any I2C transactions.
- Returns the temperature in units of 0.1°C.

```
function get_temp(self : in out BMP180_record; error : out err_code)
    return BBS.units.temp_c;
function get_temp(self : in out BMP180_record; error : out err_code)
    return BBS.units.temp_f;
function get_temp(self : in out BMP180_record; error : out err_code)
    return BBS.units.temp_k;
```

Returns the temperature in units of °C, °F, or K, depending on datatype of the destination.

- *self* - The BMP180 device to instruct.
- *error* - The error code from any I2C transactions.

- Returns the temperature in units of °C, °F, or K.

```
function get_press(self : BMP180_record; error : out err_code)
    return integer;
```

Returns the calibrated pressure value as an **Integer**. The LSB unit is 1Pa.

- *self* - The BMP180 device to instruct.
- *error* - The error code from any I2C transactions.
- Returns the temperature in units of 0.1°C.

```
function get_press(self : BMP180_record; error : out err_code)
    return BBS.units.press_p;
function get_press(self : BMP180_record; error : out err_code)
    return BBS.units.press_mb;
function get_press(self : BMP180_record; error : out err_code)
    return BBS.units.press_atm;
function get_press(self : BMP180_record; error : out err_code)
    return BBS.units.press_inHg;
```

Returns that pressure in units of Pa, mB, Atm, or inHg, depending on the datatype of the destination.

- *self* - The BMP180 device to instruct.
- *error* - The error code from any I2C transactions.
- Returns the pressure in units of Pa, mB, Atm, or inHg.

4.2.5 BBS.embed.i2c.L3GD20H

This is a driver for the STMicrosystems L3GD20H three-axis digital output gyroscope [9]. This device has a number of operating modes that have not been implemented in this driver. A number of constants are defined, but most of them are intended only for internal use. The constant *addr* is the I2C address of the L3GD20H sensor and is intended for use in the **configure** call.

Constant	Full-Scale Deflection
<i>fs_245dps</i>	Full scale is 245°/S (default)
<i>fs_500dps</i>	Full scale is 500°/S
<i>fs_2000dps</i>	Full scale is 2000°/S

Table 4.7: Constants for L3GD20H Full-Scale Deflection

Two utility datatypes are defined for holding rotations. The first, **rotations** is a record with **x**, **y**, and **z** components all of **Integer** type holding the raw rotation values from the sensor. The second, **rotations_dps** is similar, but the components are all of type **BBS.units.rot_d_s** for rotation in degrees per second.

Constant	Status
<i>zyx_or</i>	X,Y,Z axis data overrun - New data has overwritten previous data before it was read
<i>z_or</i>	Z axis data overrun - New data has overwritten previous data before it was read
<i>y_or</i>	Y axis data overrun - New data has overwritten previous data before it was read
<i>x_or</i>	X axis data overrun - New data has overwritten previous data before it was read
<i>zyxda</i>	X,Y,Z axis new data available
<i>zda</i>	Z axis new data available
<i>yda</i>	Y axis new data available
<i>xda</i>	X axis new data available

Table 4.8: Constants for L3GD20H Status

```
procedure configure(self : in out L3GD20H_record; port : i2c_interface;
                  addr : addr7; error : out err_code);
```

Called to configure the L3GD20H device. This must be done before using the device.

- *self* - The L3GD20H device to configure.
- *port* - The I2C bus object that the L3GD20H is connected to.
- *addr* - The I2C address of the device.
- *error* - The error code from any I2C transactions.

```
procedure configure(self : in out L3GD20H_record; port : i2c_interface;
                  addr : addr7; deflection : uint8; error : out err_code);
```

Called to configure the L3GD20H device. This must be done before using the device.

- *self* - The L3GD20H device to configure.
- *port* - The I2C bus object that the L3GD20H is connected to.
- *addr* - The I2C address of the device.
- *deflection* - Set the full-scale deflection (See constants in Table 4.7).
- *error* - The error code from any I2C transactions.

```
function get_temperature(self : L3GD20H_record; error : out err_code)
  return integer;
```

Return the device temperature in °C.

- *self* - The L3GD20H device to read.
- *error* - The error code from any I2C transactions.
- Returns an **Integer** representing the temperature in °C.

```
function get_rotation_x(self : L3GD20H_record; error : out err_code)
    return integer;
function get_rotation_y(self : L3GD20H_record; error : out err_code)
    return integer;
function get_rotation_z(self : L3GD20H_record; error : out err_code)
    return integer;
```

Return the rotation around the specified axis, x , y , or z as an **Integer** containing the raw sensor value.

- *self* - The L3GD20H device to read.
- *error* - The error code from any I2C transactions.
- Returns an **Integer** representing the rotation around the specified axis.

```
function get_rotations(self : L3GD20H_record; error : out err_code)
    return rotations;
```

Return a **rotations** record containing the raw sensor values for the rotations around each of the axis.

- *self* - The L3GD20H device to read.
- *error* - The error code from any I2C transactions.
- Returns a **rotations** record containing the rotations around all axis.

```
function get_temperature(self : L3GD20H_record; error : out err_code)
    return BBS.units.temp_c;
```

Return the device temperature in °C.

- *self* - The L3GD20H device to read.
- *error* - The error code from any I2C transactions.
- Returns a **BBS.units.temp_c** representing the temperature in °C.

```
function get_rotation_x(self : L3GD20H_record; error : out err_code)
    return BBS.units.rot_d_s;
function get_rotation_y(self : L3GD20H_record; error : out err_code)
    return BBS.units.rot_d_s;
function get_rotation_z(self : L3GD20H_record; error : out err_code)
    return BBS.units.rot_d_s;
```

Return the rotation around the specified axis, x , y , or z as a **rotations_dps** containing the rotation in °/S.

- *self* - The L3GD20H device to read.
- *error* - The error code from any I2C transactions.

- Returns a `rotations_dps` representing the rotation around the specified axis.

```
function get_rotations(self : L3GD20H_record; error : out err_code)
    return rotations_dps;
```

Return a `otations_dps` record containing the rotation in °/S around each of the axis.

- *self* - The L3GD20H device to read.
- *error* - The error code from any I2C transactions.
- Returns a `rotations_dps` record containing the rotations around all axis.

```
function get_status(self : L3GD20H_record; error : out err_code)
    return uint8;
```

Return the device status. The constants in Table 4.8 can be used to decode the status.

- *self* - The L3GD20H device to read.
- *error* - The error code from any I2C transactions.
- Returns an `uint8` representing the device status.

```
function data_ready(self : L3GD20H_record; error : out err_code)
    return boolean;
```

Checks if the sensor has data ready.

- *self* - The L3GD20H device to read.
- *error* - The error code from any I2C transactions.
- Returns *True* if data is ready.

```
function measure_offsets(self : in out L3GD20H_record) return boolean;
```

When stationary, the sensors may not report 0. This function should be called when the sensor is stationary. It reads the rotations several times and averages the results. This is used to calculate offset values. The offset values are used when returning the rotations in `rotations_dps`.

This function returns *True* if the measurement was successful - that is all of the values measured are reasonably close to the mean. If it returns false, the sensor may be moving.

- *self* - The L3GD20H device to read.
- *error* - The error code from any I2C transactions.
- Returns *True* if the offset measurement was successful.

Constant	Full-Scale Deflection
<i>fs_2g</i>	Full scale is 2G (default)
<i>fs_4g</i>	Full scale is 4G
<i>fs_8g</i>	Full scale is 8G
<i>fs_16gs</i>	Full scale is 16G

Table 4.9: Constants for LSM303DLHC Full-Scale Accelerometer

Constant	Status
<i>accel_stat_zyxor</i>	X,Y,Z axis data overrun - New data has overwritten previous data before it was read
<i>accel_stat_zor</i>	Z axis data overrun - New data has overwritten previous data before it was read
<i>accel_stat_yor</i>	Y axis data overrun - New data has overwritten previous data before it was read
<i>accel_stat_xorr</i>	X axis data overrun - New data has overwritten previous data before it was read
<i>accel_stat_zyxda</i>	X,Y,Z axis new data available
<i>accel_stat_zda</i>	Z axis new data available
<i>accel_stat_yda</i>	Y axis new data available
<i>accel_stat_xda</i>	X axis new data available

Table 4.10: Constants for LSM303DLHC Accelerometer Status

Constant	Full-Scale Deflection
<i>fa_1.3_gauss</i>	Full scale is 1.3 gauss
<i>fs_1.9_gauss</i>	Full scale is 1.9 gauss
<i>fs_2.5_gauss</i>	Full scale is 2.5 gauss
<i>fs_4.0_gauss</i>	Full scale is 4.0 gauss
<i>fs_4.7_gauss</i>	Full scale is 4.7 gauss
<i>fs_5.6_gauss</i>	Full scale is 5.6 gauss
<i>fs_8.1_gauss</i>	Full scale is 8.1 gauss

Table 4.11: Constants for LSM303DLHC Full-Scale Magnetometer

Constant	Status
<i>mag_lock</i>	Data output register lock.
<i>mag_drdy</i>	Data ready

Table 4.12: Constants for LSM303DLHC Magnetometer Status

4.2.6 BBS.embed.i2c.LSM303DLHC

This is a driver for the STMicrosystems LSM303DLHC three-axis digital output gyroscope [10]. This device has a number of operating modes that have not been implemented in this driver. A number of constants are defined, but most of them are intended only for internal use. The constant *addr_accel* is the I2C address of the LSM303DLHC accelerometer sensor; the constant *addr_mag* is the I2C address of the magnetometer sensor. They are intended for use in the *configure* call.

Two utility datatypes are defined for holding accelerations. The first, **accelerations** is a record with **x**, **y**, and **z** components all of **Integer** type holding the raw acceleration values from the sensor. The second, **accelerations_g** is similar, but the components are all of type **BBS.units.accel_g** for acceleration in units on 1 earth gravity.

Two utility datatypes are defined for holding magnetic fields. The first, **magnetism** is a record with **x**, **y**, and **z** components all of **Integer** type holding the raw magnetometer values from the sensor. The second, **magnetism_gauss** is similar, but the components are all of type **BBS.units.mag_g** for magnetism in units on 1 gauss.

```
procedure configure(self : in out LSM303DLHC_record; port : i2c_interface;
                   accel : addr7; mag : addr7; error : out err_code);
```

Called to configure the LSM303DLHC device. This must be done before using the device.

- *self* - The LSM303DLHC device to configure.
- *port* - The I2C bus object that the LSM303DLHC is connected to.
- *accel* - The I2C address of the device accelerometer.
- *mag* - The I2C address of the device magnetometer.
- *error* - The error code from any I2C transactions.

```
procedure configure(self : in out LSM303DLHC_record;
                   port : i2c_interface; addr_accel : addr7; addr_mag : addr7;
                   accel_fs : uint8; mag_fs : uint8; error : out err_code);
```

Called to configure the LSM303DLHC device. This must be done before using the device.

- *self* - The LSM303DLHC device to configure.
- *port* - The I2C bus object that the LSM303DLHC is connected to.
- *addr_accel* - The I2C address of the device accelerometer.
- *addr_mag* - The I2C address of the device magnetometer.
- *accel_fs* - The acceleration full scale value. See Table 4.9.
- *mag_fs* - The acceleration full scale value. See Table 4.11.
- *error* - The error code from any I2C transactions.

```
procedure calibrate_accel(self : in out LSM303DLHC_record);
```

The calibrate accel procedure can be called when the sensor is stationary in a 1G acceleration or gravitational field. It takes multiple measurements of the X, Y, and Z acceleration and computes the average of $X^2 + Y^2 + Z^2$. This value should be 1.0. A more sophisticated approach would be to compute a calibration value for each of the axis separately, but that would require the sensor to be precisely positioned three time.

- *self* - The LSM303DLHC device to calibrate.

```
function get_acceleration_x(self : LSM303DLHC_record; error : out err_code)
    return integer;
function get_acceleration_y(self : LSM303DLHC_record; error : out err_code)
    return integer;
function get_acceleration_z(self : LSM303DLHC_record; error : out err_code)
    return integer;
```

Return the acceleration along the specified axis, *x*, *y*, or *z* as an **Integer** containing the raw sensor value.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns an **Integer** representing the acceleration along the specified axis.

```
function get_accelerations(self : LSM303DLHC_record; error : out err_code)
    return accelerations;
```

Return the acceleration along all axis, *x*, *y*, or *z* as an **accelerations** containing the raw sensor value.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns an **accelerations** record containing the acceleration along the all axis.

```
function get_acceleration_x(self : LSM303DLHC_record; error : out err_code)
    return BBS.units.accel_g;
function get_acceleration_y(self : LSM303DLHC_record; error : out err_code)
    return BBS.units.accel_g;
function get_acceleration_z(self : LSM303DLHC_record; error : out err_code)
    return BBS.units.accel_g;
```

Return the acceleration along the specified axis, *x*, *y*, or *z* as a **BBS.units.accel_g** containing the acceleration in units of 1 gravity.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns a **BBS.units.accel_g** representing the acceleration along the specified axis.

```
function get_accelerations(self : LSM303DLHC_record; error : out err_code)
    return accelerations_g;
```

Return the acceleration along all axis, *x*, *y*, or *z* as an **accelerations_g** containing the acceleration in units of 1 gravity.

- *self* - The LSM303DLHC device to read.

- *error* - The error code from any I2C transactions.
- Returns an `accelerations_g` record containing the acceleration along the all axis.

```
function get_accel_status(self : LSM303DLHC_record; error : out err_code)
    return uint8;
```

Return the accelerometer status. The constants in Table 4.10 can be used to decode the status.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns an `uint8` representing the device status.

```
function accel_data_ready(self : LSM303DLHC_record; error : out err_code)
    return boolean;
```

Checks if the sensor has data ready.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns *True* if data is ready.

```
function get_temperature(self : LSM303DLHC_record; error : out err_code)
    return integer;
```

Return the device temperature as an **Integer** in units of $\frac{1}{8}^{\circ}\text{C}$.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns an **Integer** representing the temperature in units of $\frac{1}{8}^{\circ}\text{C}$.

```
function get_temperature(self : LSM303DLHC_record; error : out err_code)
    return float;
```

Return the device temperature as a **Float** in units of 1°C .

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns a **Float** representing the temperature in $^{\circ}\text{C}$.

```
function get_temperature(self : LSM303DLHC_record; error : out err_code)
    return BBS.units.temp_c;
```

Return the device temperature in $^{\circ}\text{C}$.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns a `BBS.units.temp_c` representing the temperature in °C.

```
function get_magnet_x(self : LSM303DLHC_record; error : out err_code)
    return integer;
function get_magnet_y(self : LSM303DLHC_record; error : out err_code)
    return integer;
function get_magnet_z(self : LSM303DLHC_record; error : out err_code)
    return integer;
```

Return the magnetic field along the specified axis, *x*, *y*, or *z* as an **Integer** containing the raw sensor value.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns an **Integer** representing the magnetic field along the specified axis.

```
function get_magnetism(self : LSM303DLHC_record; error : out err_code)
    return magnetism;
```

Return the magnetic field along all axis, *x*, *y*, or *z* as an **magnetism** containing the raw sensor value.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns an **magnetism** record containing the magnetic field along the all axis.

```
function get_magnet_x(self : LSM303DLHC_record; error : out err_code)
    return BBS.units.mag_g;
function get_magnet_y(self : LSM303DLHC_record; error : out err_code)
    return BBS.units.mag_g;
function get_magnet_z(self : LSM303DLHC_record; error : out err_code)
    return BBS.units.mag_g;
```

Return the magnetic field along the specified axis, *x*, *y*, or *z* as a `BBS.units.mag_g` in units of gauss.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns an `BBS.units.mag_g` representing the magnetic field along the specified axis in units of gauss.

```
function get_magnetism(self : LSM303DLHC_record; error : out err_code)
    return magnetism_gauss;
```

Return the magnetic field along all axis, x , y , or z as an `magnetism_gauss` in units of gauss.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns an `magnetism_gauss` record containing the magnetic field along the all axis.

```
function get_mag_status(self : LSM303DLHC_record; error : out err_code)
    return uint8;
```

Return the magnetometer status. The constants in Table 4.12 can be used to decode the status.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns an `uint8` representing the device status.

```
function mag_data_ready(self : LSM303DLHC_record; error : out err_code)
    return boolean;
```

Checks if the sensor has data ready.

- *self* - The LSM303DLHC device to read.
- *error* - The error code from any I2C transactions.
- Returns *True* if data is ready.

4.2.7 BBS.embed.i2c.MCP4725

This is a driver for the Microchip MCP4725 digital to analog convertor that attaches to an I2C bus [5].

Constant	Command
<i>Fast_Write</i>	Fast write.
<i>Write_CMD</i>	Write CMD.
<i>Write_EEPROM</i>	Write EEPROM.

Table 4.13: Constants for MCP4725 Commands

```
procedure configure(self : in out MCP4725_record; port : i2c_interface;
    addr : addr7; error : out err_code);
```

Called to configure the MCP4725 device. This must be done before using the device.

- *self* - The MCP4725 device to configure.
- *port* - The I2C bus object that the MCP4725 is connected to.

Constant	Power-Down Mode
<i>PD_Normal</i>	Normal mode.
<i>PD_1k</i>	1k Ω to groupd.
<i>PD_100k</i>	100k Ω to ground.
<i>PD_500k</i>	500k Ω to ground.

Table 4.14: Constants for MCP4725 Power-Down Modes

- *addr* - The I2C address of the device.
- *error* - The error code from any I2C transactions.

```
procedure set(self : in out MCP4725_record; value : uint12;
             err : out err_code);
```

The DAC is only single channel, so this just sets the value using fast write and PD normal mode.

- *self* - The MCP4725 device to set.
- *value* - The output value to use for the DAC.
- *error* - The error code from any I2C transactions.

```
procedure set(self : in out MCP4725_record; cmd : CMD_type; mode : Mode_type;
             value : uint12; err : out err_code);
```

General set command. Use the defined constants for the command and mode. Other values may cause unexpected behavior.

- *self* - The MCP4725 device to set.
- *cmd* - The command to use. See Table 4.13 for available commands.
- *mode* - The power-down mode to use. See Table 4.14 for available modes.
- *value* - The output value to use for the command and mode.
- *error* - The error code from any I2C transactions.

4.2.8 BBS.embed.i2c.MCP23008

This is a driver for the Microchip MCP23008 8-bit I/O port extender[4]. The driver does not support all of the options that the device has. Should more options be needed, the driver can be extended to support them. The device can be configured to be at one of eight different I2C addresses. Constants are defined for each of the possible addresses. They are *addr_0* through *addr_7*. Refer to your hardware documentation to determine which address you need.

```
procedure configure(self : in out MCP23008_record; port : i2c_interface;
                  addr : addr7; error : out err_code);
```

Called to configure the MCP23008 device. This must be done before using the device.

- *self* - The MCP23008 device to configure.
- *port* - The I2C bus object that the MCP23008 is connected to.
- *addr* - The I2C address of the device.
- *error* - The error code from any I2C transactions.

```
procedure set_dir(self : MCB23008_record; dir : uint8;  
                 error : out err_code);
```

Set the direction (read(0)/write(1)) for each of the output bits. The direction bits are packed into a `uint8`.

- *self* - The MCP23008 device to configure.
- *dir* - The `uint8` containing the direction bits.
- *error* - The error code from any I2C transactions.

```
procedure set_data(self : MCB23008_record; data : uint8;  
                 error : out err_code);
```

Sets the output bits. Bits are packed into a `uint8`.

- *self* - The MCP23008 device to set.
- *data* - The `uint8` containing the output bits.
- *error* - The error code from any I2C transactions.

```
function read_data(self : MCB23008_record; error : out err_code)  
  return uint8;
```

Read the port. Bits are packed into a `uint8`.

- *self* - The MCP23008 device to set.
- *error* - The error code from any I2C transactions.
- Returns the status if the bits packed into a `uint8`.

4.2.9 BBS.embed.i2c.MCP23017

This is a driver for the Microchip MCP23017 8-bit I/O port extender[6]. The driver does not support all of the options that the device has. Should more options be needed, the driver can be extended to support them. The device can be configured to be at one of eight different I2C addresses. Constants are defined for each of the possible addresses. They are *addr_0* through *addr_7*. Refer to your hardware documentation to determine which address you need.

```
procedure configure(self : in out MCP23017_record; port : i2c_interface;  
                 addr : addr7; error : out err_code);
```

Called to configure the MCP23017 device. This must be done before using the device.

- *self* - The MCP23017 device to configure.
- *port* - The I2C bus object that the MCP23017 is connected to.
- *addr* - The I2C address of the device.
- *error* - The error code from any I2C transactions.

```
procedure set_dir(self : MCP23017_record; dir : uint16;  
                 error : out err_code);
```

Set the direction (read(0)/write(1)) for each of the output bits. The direction bits are packed into a `uint16`.

- *self* - The MCP23017 device to configure.
- *dir* - The `uint16` containing the direction bits.
- *error* - The error code from any I2C transactions.

```
function get_dir(self : MCP23017_record;  
                error : out err_code) return uint16;
```

Read the direction (read(0)/write(1)) for each of the output bits. The direction bits are packed into a `uint16`.

- *self* - The MCP23017 device to read.
- *error* - The error code from any I2C transactions.
- Returns the direction bits packed into a `uint16`.

```
procedure set_polarity(self : MCP23017_record; dir : uint16;  
                     error : out err_code);
```

Set the polarity (normal(0)/inverted(1)) for each of the input bits. The direction bits are packed into a `uint16`.

- *self* - The MCP23017 device to configure.
- *dir* - The `uint16` containing the polarity bits.
- *error* - The error code from any I2C transactions.

```
function get_polarity(self : MCP23017_record;  
                     error : out err_code) return uint16;
```

Read the polarity (normal(0)/inverted(1)) for each of the input bits. The direction bits are packed into a `uint16`.

- *self* - The MCP23017 device to read.

- *error* - The error code from any I2C transactions.
- Returns the polarity bits packed into a `uint16`.

```
procedure set_pullup(self : MCP23017_record; dir : uint16;  
                    error : out err_code);
```

Enable/Disable weak pullup resistors (disable(0)/enable(1)) for each of the output bits. The bits are packed into a `uint16`.

- *self* - The MCP23017 device to configure.
- *dir* - The `uint16` containing the pullup bits.
- *error* - The error code from any I2C transactions.

```
function get_pullup(self : MCP23017_record;  
                   error : out err_code) return uint16;
```

Read weak pullup resistors (disable(0)/enable(1)) for each of the output bits. The bits are packed into a `uint16`.

- *self* - The MCP23017 device to read.
- *error* - The error code from any I2C transactions.
- Returns the pullup bits packed into a `uint16`.

```
procedure set_data(self : MCP23017_record; data : uint16;  
                 error : out err_code);
```

Sets the output bits. Bits are packed into a `uint16`.

- *self* - The MCP23017 device to set.
- *data* - The `uint16` containing the output bits.
- *error* - The error code from any I2C transactions.

```
function get_data(self : MCP23017_record; error : out err_code)  
               return uint16;
```

Read the port. Bits are packed into a `uint16`.

- *self* - The MCP23017 device to read.
- *error* - The error code from any I2C transactions.
- Returns the status if the bits packed into a `uint16`.

4.2.10 BBS.embed.i2c.PCA9685

This is a driver for the NXP Semiconductors PCA9685 16-channel, 12-bit PWM with I2C bus LED controller[8]. The driver does not support all of the options that the device has, but these can be added if needed. The device can be configured to respond to one of four different I2C addresses. The constants `addr_0` through `addr_3` are defined for these addresses.

In addition to controlling LEDs, it can control other PWM devices such as servo motors. Note that LED brightness is controlled by the duty cycle and any duty cycle is valid. Servos are controlled by the pulse width which should range from 1.5 to 2.5 mS. See Table 4.15 for some measured values. Note that all measured numbers are approximate. There are probably a few counts left before hitting full scale movement. It's also entirely possible that these values may vary with time, temperature, or other factors.

Servo	Min-position	Max-position
SG90	500mS	2100mS
SG99	450mS	2050mS
SG5010	500mS	2100mS

Table 4.15: Measured Values for PCA9685 Controlling Servos

There are some things to keep in mind:

1. Test your own servos to determine their appropriate values.
2. If you want any sort of precision, you need some sort of position feed-back to the program.
3. The documentation that says that the pulse width for servos should range from 1.5 to 2.5 mS may not be accurate.

PWM channels are 0 to 15. Channel 16 is the all call channel. For each channel there is a 12 bit counter and two thresholds: the on and the off threshold. When the counter is equal to the on threshold, the output turns on. When the counter is equal to the off threshold, the output turns off. This allows the pulses to be staggered between the channels, if needed.

For driving servo motors, the `servo_range` datatype is defined as a `Float` with a range from -1.0 to 1.0. Once the `set_servo_range` procedure has been called, the `set_servo` procedure can be used to set the servo position using `servo_range` rather than figuring out the settings for the duty cycle. Thus, when changing between servos with different characteristics, all that needs to change is the `set_servo_range` call.

```
procedure configure(self : in out PS9685_record; port : i2c_interface;
                  addr : addr7; error : out err_code);
```

Called to configure the PCA9685 device. This must be done before using the device.

- `self` - The PCA9685 device to configure.
- `port` - The I2C bus object that the PCA9685 is connected to.
- `addr` - The I2C address of the device.

- *error* - The error code from any I2C transactions.

```
procedure set(self : PS9685_record; chan : channel;  
             on : uint12; off : uint12; error : out err_code);
```

Set on and off times for a specific channel.

- *self* - The PCA9685 device to configure.
- *chan* - The channel number.
- *on* - The on time for the channel.
- *off* - The off time for the channel.
- *error* - The error code from any I2C transactions.

```
procedure set_full_on(self : PS9685_record; chan : channel;  
                    error : out err_code);
```

Sets the specified channel to full on.

- *self* - The PCA9685 device to configure.
- *chan* - The channel number.
- *error* - The error code from any I2C transactions.

```
procedure set_full_off(self : PS9685_record; chan : channel;  
                    error : out err_code);
```

Sets the specified channel to full off.

- *self* - The PCA9685 device to configure.
- *chan* - The channel number.
- *error* - The error code from any I2C transactions.

```
procedure sleep(self : PS9685_record; state : boolean;  
              error : out err_code);
```

If state is *True*, send the device to sleep, otherwise wake it up.

- *self* - The PCA9685 device to configure.
- *state* - *True* for sleep, *False* for wake.
- *error* - The error code from any I2C transactions.

```
procedure set_servo_range(self : in out PS9685_record; chan : channel;  
                        min : uint12; max : uint12);
```

Sets the maximum and minimum duty cycles for a channel. Once these are set, a servo motor can be controlled using the `set_servo` procedure using a servo position in the range on -1.0 to 1.0.

- *self* - The PCA9685 device to configure.
- *chan* - The channel number.
- *min* - The minimum duty cycle for the servo, corresponding to servo position -1.0.
- *max* - The maximum duty cycle for the servo, corresponding to servo position 1.0.

```
procedure set_servo(self : PS9685_record; chan : channel;  
                   position : servo_range; error : out err_code);
```

Once the servo range has been set by the `set_servo_range` procedure, the servo can be controlled by this function using a servo position from -1.0 to 1.0 rather than the duty cycle.

- *self* - The PCA9685 device to configure.
- *chan* - The channel number.
- *position* - The servo position in a range from -1.0 to 1.0.
- *error* - The error code from any I2C transactions.

4.2.11 BBS.embed.SPI.RA8875

[7]

Chapter 5

Linux API Description

5.1 Common

5.2 Raspberry Pi

5.3 BeagleBone Black

Chapter 6

Arduino Due API Description

Chapter 7

Other Stuff

If there is anything else that should be added, additional chapters may be added as needed.

Bibliography

- [1] Bosch. BMP180 digital pressure sensor, April 2013.
- [2] Bosch. BME280 combined humidity and pressure sensor, February 2024.
- [3] Texas Instruments. ADS1015 12-bit, 3.3-ksps, 4-channel, delta-sigma ADC with PGA, oscillator, VREF, comparator, and I2C, January 2018.
- [4] Microchip. MCP23008 8-bit I/O expander with serial interface, 2007.
- [5] Microchip. MCP4725 12-bit digital-to-analog converter with EEPROM memory, 2009.
- [6] Microchip. MCP23017 16-bit I/O expander with serial interface, 2016.
- [7] RAiO. RA8875 character/graphic TFT LCD controller, March 2013.
- [8] NXP Semiconductors. PCA9685 16-channel, 12-bit PWM fm+I²C-bus LED controller, April 2015.
- [9] STMicroelectronics. L3GD20 low power 3-axis gyroscope, I2C/SPI digital output, February 2013.
- [10] STMicrosystems. LSM303DLHC ultra-compact high-performance ecompass module: 3d accelerometer and 3d magnetometer, November 2013.
- [11] Toshiba. TB6612FNG driver IC for dual DC motor, October 2014.