Users's Manual for Device Interfaces

Brent Seidel
Phoenix, AZ

September 19, 2024

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 `BBS.embed.AIN` . . . 7
  - 4.1.2 `BBS.embed.GPIO` . . . 7
  - 4.1.3 `BBS.embed.i2c` . . . 7
  - 4.1.4 `BBS.embed.log` . . . 10
  - 4.1.5 `BBS.embed.SPI` . . . 11
- 4.2 Higher-Level Device Drivers . . . 11
  - 4.2.1 `BBS.embed.gpio.tb6612` . . . 11
  - 4.2.2 `BBS.embed.I2C.ADS1015` . . . 13
  - 4.2.3 `BBS.embed.i2c.BME280` . . . 17
  - 4.2.4 `BBS.embed.i2c.BMP180` . . . 19
  - 4.2.5 `BBS.embed.i2c.L3GD20H` . . . 22
  - 4.2.6 `BBS.embed.i2c.LSM303DLHC` . . . 25
  - 4.2.7 `BBS.embed.i2c.MCP4725` . . . 31
  - 4.2.8 `BBS.embed.i2c.MCP23008` . . . 32
  - 4.2.9 `BBS.embed.i2c.MCP23017` . . . 33
  - 4.2.10 `BBS.embed.i2c.PCA9685` . . . 35
  - 4.2.11 `BBS.embed.SPI.RA8875` . . . 38

5 Linux API Description **56**
5.1 Common . . . 56
5.1.1 BBS.embed.AIN.linux . . . 56
5.1.2 BBS.embed.GPIO.Linux . . . 57
5.1.3 BBS.embed.i2c.linux . . . 58
5.1.4 BBS.embed.log.linux . . . 58
5.1.5 BBS.embed.SPI.Linux . . . 58
5.2 Raspberry Pi . . . 59
5.2.1 Initialization Script . . . 59
5.2.2 BBS.embed.RPI . . . 59
5.3 BeagleBone Black . . . 59
5.3.1 Initialization Script . . . 59
5.3.2 BBS.embed.BBB . . . 60
5.3.3 BBS.embed.LED . . . 60
5.3.4 BBS.embed.PWM . . . 61

6 Arduino Due API Description **63**
6.1 BBS.embed.AIN.due . . . 63
6.2 BBS.embed.due.dev . . . 64
6.3 BBS.embed.Due.GPIO . . . 65
6.4 BBS.embed.due.serial . . . 65
6.5 BBS.embed.due.serial.int . . . 66
6.6 BBS.embed.due.serial.polled . . . 70
6.7 BBS.embed.due . . . 71
6.8 BBS.embed.GPIO.Due . . . 71
6.9 BBS.embed.i2c.due . . . 72
6.10 BBS.embed.log.due . . . 73
6.11 BBS.embed.SPI.Due . . . 73

Indices **74**

Bibliography **76**

List of Tables

4.1 Enumerations for ADS1015 Mux Mode Configuration, Datatype: `mux_mode_type` . . 13
4.2 Enumerations for ADS1015 Programmable Gain Amplifier Configuration, Datatype: `pga_type` 13
4.3 Enumerations for ADS1015 Data Rate Configuration, Datatype: `data_rate_type` . . 13
4.4 Enumerations for ADS1015 Comparator Queue Configuration, Datatype: `comp_que_type` 14
4.5 Enumerations for ADS1015 Operating Mode, Datatype: `mode_type` 14
4.6 Enumerations for ADS1015 Comparator Mode, Datatype: `comp_mode_type` 14
4.7 Enumerations for ADS1015 Comparator Polarity, Datatype: `comp_polarity` 14
4.8 Enumerations for ADS1015 Comparator Latching, Datatype: `comp_latch` 14
4.9 Enumerations for BMP180 Conversion Kinds, Datatype `cvt_type` 20
4.10 Constants for L3GD20H Status 22
4.11 Constants for L3GD20H Full-Scale Deflection 22
4.12 Constants for LSM303DLHC Full-Scale Accelerometer 25
4.13 Constants for LSM303DLHC Accelerometer Status 26
4.14 Constants for LSM303DLHC Full-Scale Magnetometer 26
4.15 Constants for LSM303DLHC Magnetometer Status 26
4.16 Enumerations for MCP4725 Commands, Datatype `CMD_type` 31
4.17 Enumerations for MCP4725 Power-Down Modes, Datatype `Mode_type` 31
4.18 Measured Values for PCA9685 Controlling Servos 36

6.1 SAM3X8E Analog Input Assignment to Arduino Due Pins 64

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

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 BBS.embed.AIN

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 BBS.embed.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;
\indexfunc{get}
```

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 BBS.embed.i2c

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.
- *addr*7 - 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.
- *addr*7 - 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.
- *addr*7 - 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.
- *addr*7 - 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.
- *addr*7 - 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.
- *addr*7 - 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.

- *addr*7 - 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.
- *addr*7 - 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 `BBS.embed.log`

This package provides the definitions for logging capabilities. Since not all embedded systems have access to `Ada.Text_IO`, this package is provided. It defines a logging object that can be subclassed to perform logging on the specific target without having to modify the client packages. This basic class will simply discard any text. Note that multiple logging streams may be defined.

The following are the interface to logging:

```
procedure enable(self : in out log_record);
```

Enables logging on a log stream.

- *self* - The log stream to enable.

```
procedure disable(self : in out log_record)
```

Disables logging on a log stream.

- *self* - The log stream to disable.

```
procedure put(self : log_record; text : String);
```

Writes text to a log stream without adding a newline at the end.

- *self* - The log stream to add text to.
- *text* - The text to add.

```
procedure put_line(self : log_record; text : String) is null;
```

Writes text to a log stream with a newline at the end.

- *self* - The log stream to add text to.
- *text* - The text to add.

The following log streams are defined and can be used for various purposes. These streams may be used by some of the software in this project for debugging purposes. They are initialized to the dummy log stream which simply discards any text logged. Should you wish to use them, you would need to create your own logging class based on `log_record` that actually does something with the text and set the appropriate stream(s) to point to your new subclass.

```
debug : log_ptr := dummy_log'Access;
info  : log_ptr := dummy_log'Access;
error : log_ptr := dummy_log'Access;
```

4.1.5 `BBS.embed.SPI`

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 [12]. 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 [4]. This converter has a wide variety of configuration options. Refer to the datasheet for details. A number of enumerations have been defined to support the various configuration options.

Enumeration	Mux Mode Configuration
a0_a1	Difference between AIN0 and AIN1 (default)
a0_a3	Difference between AIN0 and AIN3
a1_a3	Difference between AIN1 and AIN3
a2_a3	Difference between AIN2 and AIN3
a0_gnd	Single ended AIN0 value
a1_gnd	Single ended AIN1 value
a2_gnd	Single ended AIN2 value
a3_gnd	Single ended AIN3 value

Table 4.1: Enumerations for ADS1015 Mux Mode Configuration, Datatype: `mux_mode_type`

Enumeration	PGA Configuration
pga_6_144	Full scale voltage is 6.144V
pga_4_096	Full scale voltage is 4.096V
pga_2_048	Full scale voltage is 2.048V (default)
pga_1_024	Full scale voltage is 1.024V
pga_0_512	Full scale voltage is 0.512V
pga_0_256	Full scale voltage is 0.256V

Table 4.2: Enumerations for ADS1015 Programmable Gain Amplifier Configuration, Datatype: `pga_type`

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

Table 4.3: Enumerations for ADS1015 Data Rate Configuration, Datatype: `data_rate_type`

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

Enumeration	Comparator Queue Configuration
comp_que_1	Assert after one conversion
comp_que_2	Assert after two conversions
comp_que_3	Assert after three conversion
comp_que_d	Disable comparator (default)

Table 4.4: Enumerations for ADS1015 Comparator Queue Configuration, Datatype: `comp_que_type`

Enumeration	Operating Mode
continuous	Continuous conversion mode
single_shot	Power-down single-shot mode (default)

Table 4.5: Enumerations for ADS1015 Operating Mode, Datatype: `mode_type`

Enumeration	Comparator Mode
traditional	Traditional, with hysteresis (default)
window	Window comparator

Table 4.6: Enumerations for ADS1015 Comparator Mode, Datatype: `comp_mode_type`

Enumeration	Comparator Polarity
act_low	Active low (default)
act_high	Active high

Table 4.7: Enumerations for ADS1015 Comparator Polarity, Datatype: `comp_polarity`

Enumeration	Comparator Latching
no_latch	Non-latching comparator (default)
latch	Latching comparator

Table 4.8: Enumerations for ADS1015 Comparator Latching, Datatype: `comp_latch`

- `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 4.6).

- `comp_pol` - The comparator polarity (see Table 4.7).
- `comp_lat` - The comparator latching (see Table 4.8).
- `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.

```
function present(self : ADS1015_record) return Boolean;
```

Check to see if configured device is present. Returns ***True*** if device is present or ***False*** otherwise. It does this by attempting to read the configuration register.

- Returns ***True*** if the configuration register can be read, and ***False*** otherwise.

```
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 [3]. 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.

```
function present(self : BME280_record) return Boolean;
```

Check to see if configured device is present. Returns ***True*** if device is present or ***False*** otherwise. It does this by attempting to read the ID register and comparing the result to 16#60#.

- Returns ***True*** if 16#60# was successfully read from the ID register, and ***False*** otherwise.

```
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 [2]. 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.

```
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.

Enumeration	Conversion Kind
cvt_temp	Convert Temperature
cvt_press0	Convert pressure with no oversampling
cvt_press1	Convert pressure with oversampling of two
cvt_press2	Convert pressure with oversampling of four
cvt_press3	Convert pressure with oversampling of eight

Table 4.9: Enumerations for BMP180 Conversion Kinds, Datatype `cvt_type`

- *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 4.9 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 [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*** is the I2C address of the L3GD20H sensor and is intended for use in the `configure` call.

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

Table 4.10: Constants for L3GD20H Status

Enumeration	Status
fs_245dpsr	Full-Scale deflection is 245°/S
fs_500dps	Full-Scale deflection is 500°/S
fs_2000dpsr	Full-Scale deflection is 2000°/S

Table 4.11: Enumerations for L3GD20H Full-Scale Deflection

```
type rotations is
   record
      x : integer;
      y : integer;
      z : integer;
   end record;
```

The datatype `rotations` is a record that holds the raw rotation values from the X, Y, and Z sensors.

```
type rotations_dps is
   record
      x : BBS.units.rot_d_s;
      y : BBS.units.rot_d_s;
      z : BBS.units.rot_d_s;
   end record;
```

The datatype `rotations_dps` is a record that holds the rotation values in degrees per second from the X, Y, and Z sensors.

```
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 4.11).
- *error* - The error code from any I2C transactions.

```
function get_temp(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_temp(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 4.10 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.

4.2.6 `BBS.embed.i2c.LSM303DLHC`

This is a driver for the STMicrosystems LSM303DLHC three-axis digital output gyroscope [11]. 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.

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

Table 4.12: Constants for LSM303DLHC Full-Scale Accelerometer

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

Table 4.13: Constants for LSM303DLHC Accelerometer Status

Constant	Full-Scale Deflection
fa_1_3_gauss	Full scale is 1.3 gauss
fs_1_9_gauss	Full scale is 1.9 gauss
fs_2_5_gauss	Full scale is 2.5 gauss
fs_4_0_gauss	Full scale is 4.0 gauss
fs_4_7_gauss	Full scale is 4.7 gauss
fs_5_6_gauss	Full scale is 5.6 gauss
fs_8_1_gauss	Full scale is 8.1 gauss

Table 4.14: Constants for LSM303DLHC Full-Scale Magnetometer

Constant	Status
mag_lock	Data output register lock.
mag_drdy	Data ready

Table 4.15: Constants for LSM303DLHC Magnetometer Status

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 4.12.
- *mag_fs* - The acceleration full scale value. See Table 4.14.
- *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 4.13 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_temp(self : LSM303DLHC_record; error : out err_code)
   return integer;
```

Return the device temperature as an `Integer` in units of $\frac{1}{8}$°C.

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

```
function get_temp(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 °C.

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

Return the device temperature in °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 4.15 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 [6].

Enumeration	Command
Fast_Write	Fast write.
Write_CMD	Write CMD.
Write_EEPROM	Write EEPROM.

Table 4.16: Enumerations for MCP4725 Commands, Datatype CMD_type

Enumeration	Power-Down Mode
PD_Normal	Normal mode.
PD_1k	1kΩ to ground.
PD_100k	100kΩ to ground.
PD_500k	500kΩ to ground.

Table 4.17: Enumerations for MCP4725 Power-Down Modes, Datatype Mode_type

```
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.
- *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 4.16 for available commands.
- *mode* - The power-down mode to use. See Table 4.17 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[5]. 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 MCB23008_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[7]. 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.

```
function present(port : i2c_interface;
                 addr : addr7) return boolean;
```

Check to see if a MCP23017 is present at the specified I2C address. First it checks if the address is in the range for the MCP23017 and then if a read of one of the device registers completes successfully.

- *port* - The I2C bus object that the MCP23017 is connected to.

- *addr* - The I2C address of the device to check.

```
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[9]. 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 4.18 for some measured values.

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

Table 4.18: Measured Values for PCA9685 Controlling Servos

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.

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`

This is the driver for the RAiO RA8875 LCD controller that attaches to the SPI bus[8]. There are a couple of hardware considerations. First, it appears that the RA8875 does not tri-state its SPI output, which means that it doesn't work well with other devices on the SPI bus. This can be fixed by adding a tri-state buffer. Second, while the RA8875 operates at 3.3V, the LCD I used requires 5V for the backlight. This is not insurmountable, but something to be aware of.

The intention of this driver is to provide an Ada interface to the various text and graphics primitives offered by the RA8875. While it works, it still needs some work and should be considered to be experimental. Use at your own risk, or you can use this as a starting point for writing your own driver.

There are a number of constants defined. I expect that most of these are used internal to the driver and should be moved out of the spec. There are also some types that are used in the API. These are documented below.

```
type RA8875_sizes is (RA8875_480x272, RA8875_800x480);
```

The datatype `RA8875_sizes` is an enumeration defined for the screen sizes supported by the RA8875. The sizes supported by the RA8875 are 320x240, 320x480, 480x272, 640x480, and 800x480. Right now I only have an 800x480 panel for testing so nothing is tested for other sizes.

```
type RA8875_LAYER is (LAYER1, LAYER2);
```

The datatype `RA8875_LAYER` is used to select the currently active layer for the display. In some configurations, only one layer is allowed and this option is ignored.

```
type RA8875_MWCR0_MODE is (graphic, text);
```

The datatype `RA8875_MWCR0_MODE` is used to determine if the RA8875 is operating in text mode or graphics mode.

```
type RA8875_MWCR0_CURDIR is (LRTD, RLTD, TDLR, DTLR);
```

The datatype `RA8875_MWCR0_CURDIR` is used to determine the memory write direction for graphics mode. The options are left to right top down (*`LRTD`*), right to left top down (*`RLTD`*), top down left to right (*`TDLR`*), and down to top left to right (*`DTLR`*).

```
type RA8875_MWCR1_GCURS_ENABLE is (disable, enable);
```

The datatype `RA8875_MWCR1_GCURS_ENABLE` is used to determine if the graphics cursor should be enabled or disabled.

```
type RA8875_MWCR1_GCURS_SET is range 0 .. 7;
```

The datatype `RA8875_MWCR1_GCURS_SET` is used to select which set of graphics cursors to use.

```
type RA8875_MWCR1_WRITE_DEST is (LAYER, CGRAM, GCURS, PATTERN);
```

The datatype `RA8875_MWCR1_WRITE_DEST` is used to select the destination for graphics writing.

```
type R5G6B5_color is record
   R : uint8 range 0 .. 31;
   G : uint8 range 0 .. 63;
   B : uint8 range 0 .. 31;
end record;
```

The datatype `R5G6B5_color` is used to represent colors with five bits for the red, six bits for the green, and five bits for the blue channels. This adds up to sixteen bits.

```
   type R3G3B2_color is record
      R : uint8 range 0 .. 7;
      G : uint8 range 0 .. 7;
      B : uint8 range 0 .. 3;
   end record
     with pack, size => 8;
   for R3G3B2_color use
      record
         B at 0 range 0 .. 1;
         G at 0 range 2 .. 4;
         R at 0 range 5 .. 7;
      end record;
\begin{lstlisting}
\indextype{R3G3B2\_color}
The datatype \datatype{R3G3B2\_color} is used to represent colors with three bits for red,
This adds up to eight bits.

\begin{lstlisting}
type RA8875_FNCR0_Code_Page is
     (RA8875_FNCR0_ISO8859_1, RA8875_FNCR0_ISO8859_2,
      RA8875_FNCR0_ISO8859_3, RA8875_FNCR0_ISO8859_4);
```

The datatype `RA8875_FNCR0_Code_Page` is used to select which code page to use when translating character codes to glyphs on the display.

```
type RA8875_ELLIPSE_PART is (RA8875_ELLIPSE_LL, RA8875_ELLIPSE_UL,
                             RA8875_ELLIPSE_UR, RA8875_ELLIPSE_LR);
```

The datatype `RA8875_ELLIPSE_PART` is used to select which part of an ellipse is drawn. The options are lower-left (*`RA8875_ELLIPSE_LL`*), upper-left (*`RA8875_ELLIPSE_UL`*), upper-right (*`RA8875_ELLIPSE_UR`*), and lower-right (*`RA8875_ELLIPSE_LR`*).

```
type RA8875_LTPR0_SCROLL_MODE is (LAYER12_SIMULTANEOUS, LAYER1_ONLY,
                                  LAYER2_ONLY, BUFFERED);
```

The datatype `RA8875_LTPR0_SCROLL_MODE` is used to identify the mode for scrolling. Both layers can be scrolled simultaneously, either of the two layers can be selected to scroll independently, or layer 2 can be used as a scroll buffer.

```
type RA8875_LTPR0_DISP_MODE is (ONLY_LAYER1, ONLY_LAYER2, LIGHTEN, TRANSPARENT,
                                BOOL_OR, BOOL_AND, FLOATING, RESERVED);
```

The datatype `RA8875_LTPR0_DISP_MODE` is used to identify the way to display the two layers. Each layer can be displayed by itself or in various combinations.

```
type RA8875_GCursor is array (0 .. 31, 0 .. 31) of integer range 0 .. 3
  with Pack;
```

The datatype `RA8875_GCursor` represents a bitmap for the graphics cursor. The pixel values are:

- 0 - GCC0 color
- 1 - GCC1 color
- 2 - Background color
- 3 - Inverse of background color

Note that the coordinates for GCursor are reversed from what one would expect. The Y axis coordinate is the first array index and the X axis coordinate is the second array index. One can think of it as being in row, column order.

The API is documented below.

Low Level Methods

These are primarily intended for supporting the other, higher level routines.

```
procedure setup(self : in out RA8875_record; CS : GPIO.GPIO; screen : SPI_ptr);
```

Initialize and configure the RA8875 device. Use this is no hardware reset GPIO is connected

- *self* - The RA8875 device to configure.
- *CS* - The GPIO used as the device chip select.
- *screen* - The SPI bus that the RA8875 is connected to.

```
procedure setup(self : in out RA8875_record; CS : GPIO.GPIO;
                RST : GPIO.GPIO; screen : SPI_ptr);
```

Initialize and configure the RA8875 device. Use this if a GPIO is connected to use as a hardware reset.

- *self* - The RA8875 device to configure.
- *CS* - The GPIO used as the device chip select.

- *RST* - The GPIO used as a hardware reset.
- *screen* - The SPI bus that the RA8875 is connected to.

```
procedure hwReset(self : in out RA8875_record);
```

Send a hardware reset command to the RA8875, if a GPIO has been assigned for hardware reset. Otherwise, it does nothing.

- *self* - The RA8875 device to reset.

```
procedure swReset(self : in out RA8875_record);
```

Send a software reset command to the RA8875. This can be done even if no hardware reset GPIO has been configured.

- *self* - The RA8875 device to reset.

```
procedure writeCmd(self : RA8875_record; value : uint8);
```

Send a command to the RA8875.

- *self* - The RA8875 device to send a command to.
- *value* - The command as a `uint8`.

```
procedure writeData(self : RA8875_record; value : uint8);
```

Send data to the RA8875.

- *self* - The RA8875 device to send data to.
- *value* - The data as a `uint8`.

```
function readStatus(self : RA8875_record) return uint8;
```

Read a status value from the RA8875.

- *self* - The RA8875 device to get status from.
- Returns the status as a `uint8`.

```
function readData(self : RA8875_record) return uint8;
```

Read data from the RA8875.

- *self* - The RA8875 device to get data from.
- Returns the data as a `uint8`.

```
procedure writeReg(self : RA8875_record; reg : uint8; value : uint8);
```

Writes data to a RA8875 register.

- *self* - The RA8875 device containing the register.
- *reg* - The register number as a `uint8`.
- *value* - The data as a `uint8`.

```
function readReg(self : RA8875_record; reg : uint8) return uint8;
```

Reads data from a RA8875 register.

- *self* - The RA8875 device containing the register.
- *reg* - The register number as a `uint8`.
- Returns the data as a `uint8`.

Configuration Methods

```
procedure configure(self : in out RA8875_record; size : RA8875_sizes);
```

Configures the LCD size for use by the RA8875. Currently only 480x272 and 800x480 are supported and only 800x480 has been tested.

- *self* - The RA8875 device to control.
- *size* - The LCD size as a `RA8875_sizes`.

```
procedure setSleep(self : RA8875_record; state : boolean);
```

Sets the sleep state of the RA8875.

- *self* - The RA8875 device to control.
- *state* - If ***`True`***, puts the RA8875 to sleep and turns the display off. If ***`False`*** just turns the display off.

```
procedure setDisplay(self : RA8875_record; state : boolean);
```

Turns the display on or off.

- *self* - The RA8875 device to control.
- *state* - If ***`True`***, turns the display on. If ***`False`*** turns the display off.

```
procedure GPIOX(self : RA8875_record; state : boolean);
```

Set the state of the GPIOX pin. This is apparently used in the AdaFruit breakout board, though I'm not sure what it's used for.

- *self* - The RA8875 device to control.
- *state* - Sets the state of the GPIOX pin on or off.

```
procedure PWM1config(self : RA8875_record; state : boolean; clock : uint8);
```

Configures PWM channel 1.

- *self* - The RA8875 device to control.
- *state* - Sets the state of the PWM Channel 1 to disable (*False*) or enable (*True*).
- *clock* - Sets the clock divide ratio.

```
procedure PWM2config(self : RA8875_record; state : boolean; clock : uint8);
```

Configures PWM channel 2.

- *self* - The RA8875 device to control.
- *state* - Sets the state of the PWM Channel 2 to disable (*False*) or enable (*True*).
- *clock* - Sets the clock divide ratio.

```
procedure PWM1out(self : RA8875_record; value : uint8);
```

Sets PWM channel 1 duty cycle.

- *self* - The RA8875 device to control.
- *value* - Sets the duty cycle of PWM Channel 1 in units of $\frac{1}{256}$.

```
procedure PWM2out(self : RA8875_record; value : uint8);
```

Sets PWM channel 1 duty cycle.

- *self* - The RA8875 device to control.
- *value* - Sets the duty cycle of PWM Channel 1 in units of $\frac{1}{256}$.

```
procedure setDisplayCtrl(self : RA8875_record; layer : uint8; hdir : uint8;
                         vdir : uint8);
```

Sets some of the RA8875 display control parameters.

- *self* - The RA8875 device to control.
- *layer* - 0 For one layer configuration, 128 for two layer configuration.
- *hdir* - 0 For increasing SEG number, 8 for decreasing SEG number.
- *vdir* - 0 For increasing COM number, 4 for decreasing COM number.

```
procedure setWriteCtrl0(self : RA8875_record; mode : RA8875_MWCR0_MODE;
                        cursorVisible : boolean; cursorBlink : boolean;
                        writeDir : RA8875_MWCR0_CURDIR;
                        WriteCursorIncr : boolean; ReadCursorIncr : boolean);
```

Configures the RA8875 memory write control register 0.

- *self* - The RA8875 device to control.
- *mode* - Text or graphics mode.
- *cursorVisible* - Set to ***True*** to make the cursor visible.
- *cursorBlink* - Set to ***True*** to make the cursor blink.
- *writeDir* - The direction to write data in graphics mode.
- *WriteCursorIncr* - Set to ***True*** to increment the cursor when writing memory.
- *ReadCursorIncr* - Set to ***True*** to increment the cursor when reading memory.

```
procedure setWriteCtrl1(self : RA8875_record;
                        cursorEnable : RA8875_MWCR1_GCURS_ENABLE;
                        GCursorSelect : RA8875_MWCR1_GCURS_SET;
                        writeDest : RA8875_MWCR1_WRITE_DEST;
                        layer : RA8875_LAYER);
```

Configures the RA8875 memory write control register 1.

- *self* - The RA8875 device to control.
- *cursorEnable* - Selects if the graphics cursor is enabled or not.
- *GCursorSelect* - Selects which graphics cursor to use.
- *writeDest* - Selects the destination for graphics writing.
- *writeDir* - The direction to write data in graphics mode.
- *layer* - Selects the layer for graphics writing (used only if the resolution is ≤480x4000 or the color depth is 8 bits/pixel).

Text Methods

```
procedure textMode(self : RA8875_record);
```

Sets the RA8875 to text mode.

- *self* - The RA8875 device to control.

```
procedure textColor(self : RA8875_record; bg : R5G6B5_color; fg : R5G6B5_color);
```

Sets the foreground and background color for text.

- *self* - The RA8875 device to control.
- *bg* - The background color.
- *fg* - The foreground color.

```
procedure textSetCodePage(self : RA8875_record; page : RA8875_FNCR0_Code_Page);
```

Selects the code page to used when translating character codes to glyphs on the screen. The options are: *RA8875_FNCR0_ISO8859_1*, *RA8875_FNCR0_ISO8859_2*, *RA8875_FNCR0_ISO8859_3*, and *RA8875_FNCR0_ISO8859_4*.

- *self* - The RA8875 device to control.
- *page* - The code page to use.

```
procedure textSetAttribute(self : RA8875_record; align : boolean;
                           transparent : boolean;
                           rotate : boolean; h_size : uint8; v_size : uint8);
```

Sets the attributes in the font control register for drawing text.

- *self* - The RA8875 device to control.
- *align* - Enables or disables text alignment.
- *transparent* - If *True*, the background is transparent instead of the background color.
- *rotate* - If *True*, rotate text by 90°.
- *h_size* - Horizontal size scale factor (1 .. 4).
- *v_size* - Vertical size scale factor (1 .. 4).

```
procedure textSetLineHeight(self : RA8875_record; size : uint8);
```

Set the font line distance setting.

- *self* - The RA8875 device to control.
- *size* - The distance between lines of text in pixels.

```
procedure textSetFontWidth(self : RA8875_record; size : uint8);
```

Set the spacing between characters in pixels.

- *self* - The RA8875 device to control.
- *size* - The distance between characters in pixels.

```
procedure textWrite(self : RA8875_record; str : string);
```

Writes a string of text to the display using the currently selected attributes.

- *self* - The RA8875 device to control.
- *str* - The text to write.

Graphics Methods

```
procedure graphicsMode(self : RA8875_record);
```

Sets the RA8875 to graphics mode.

- *self* - The RA8875 device to control.

```
procedure drawColor(self : RA8875_record; color : R5G6B5_color);
```

Set the color for drawing graphics. The graphics commands will use this color until it is changed.

- *self* - The RA8875 device to control.
- *color* - The color to draw.

```
procedure drawRect(self : RA8875_record; x1 : uint16; y1 : uint16; x2 : uint16;
                   y2 : uint16; fill : boolean);
```

Draws a rectangle on the display.

- *self* - The RA8875 device to control.
- $x1$ - The X-coordinate of one corner.
- $y1$ - The Y-coordinate of one corner.
- $x2$ - The opposite X-coordinate.
- $y2$ - The opposite Y-coordinate.
- *fill* - `True` for fill, `False` for outline.

```
procedure drawRndRect(self : RA8875_record; x1 : uint16; y1 : uint16; x2 : uint16;
                      y2 : uint16; rad : uint16; fill : boolean);
```

Draws a rectangle on the display.

- *self* - The RA8875 device to control.
- $x1$ - The X-coordinate of one corner.
- $y1$ - The Y-coordinate of one corner.
- $x2$ - The opposite X-coordinate.
- $y2$ - The opposite Y-coordinate.
- *rad* - The radius of the corner curves.
- *fill* - `True` for fill, `False` for outline.

```
procedure drawLine(self : RA8875_record; x1 : uint16; y1 : uint16; x2 : uint16;
                   y2 : uint16);
```

Draw a line between two points.

- *self* - The RA8875 device to control.
- $x1$ - The X-coordinate of one end of the line.
- $y1$ - The Y-coordinate of one end of the line.
- $x2$ - The X-coordinate of the other end of the line.
- $y2$ - The Y-coordinate of the other end of the line.

```
procedure drawCircle(self : RA8875_record; x : uint16; y : uint16; rad : uint16;
                     fill : boolean);
```

Draw a circle.

- *self* - The RA8875 device to control.
- x - The X-coordinate of the center of the circle.
- y - The Y-coordinate of the center of the circle.
- *rad* - The radius of the circle.
- *fill* - `True` for fill, `False` for outline.

```
procedure drawTriangle(self : RA8875_record; x1 : uint16; y1 : uint16;
                       x2 : uint16; y2 : uint16; x3 : uint16; y3 : uint16;
                       fill : boolean);
```

Draw a triangle.

- *self* - The RA8875 device to control.
- $x1$ - The X-coordinate of the first point.
- $y1$ - The Y-coordinate of the first point.
- $x2$ - The X-coordinate of the second point.
- $y2$ - The Y-coordinate of the second point.
- $x3$ - The X-coordinate of the third point.
- $y3$ - The Y-coordinate of the third point.
- *fill* - `True` for fill, `False` for outline.

```
procedure drawEllipse(self : RA8875_record; x : uint16; y : uint16;
                      hRad : uint16; vRad : uint16; fill : boolean);
```

Draw an ellipse oriented either horizontally or vertically.

- *self* - The RA8875 device to control.
- *x* - The X-coordinate of the center of the circle.
- *y* - The Y-coordinate of the center of the circle.
- *hRad* - The radius in the horizontal direction.
- *vRad* - The radius in the vertical direction.
- *fill* - ***True*** for fill, ***False*** for outline.

```
procedure drawEllipseSegment(self : RA8875_record; x : uint16; y : uint16;
                             hRad : uint16;  vRad : uint16;
                             seg : RA8875_ELLIPSE_PART; fill : boolean);
```

Draw a segment of an ellipse oriented either horizontally or vertically.

- *self* - The RA8875 device to control.
- *x* - The X-coordinate of the center of the circle.
- *y* - The Y-coordinate of the center of the circle.
- *hRad* - The radius in the horizontal direction.
- *vRad* - The radius in the vertical direction.
- *seg* - Which segment to draw.
- *fill* - ***True*** for fill, ***False*** for outline.

```
procedure drawRect(self : RA8875_record; x1 : uint16; y1 : uint16; x2 : uint16;
                   y2 : uint16; color : R5G6B5_color; fill : boolean);
```

Draws a rectangle on the display.

- *self* - The RA8875 device to control.
- *x*1 - The X-coordinate of one corner.
- *y*1 - The Y-coordinate of one corner.
- *x*2 - The opposite X-coordinate.
- *y*2 - The opposite Y-coordinate.
- *color* - The color for the rectangle.
- *fill* - ***True*** for fill, ***False*** for outline.

```
procedure drawRndRect(self : RA8875_record; x1 : uint16; y1 : uint16; x2 : uint16;
                   y2 : uint16; rad : uint16; color : R5G6B5_color; fill : boolean);
```

Draws a rectangle on the display.

- *self* - The RA8875 device to control.
- *x*1 - The X-coordinate of one corner.
- *y*1 - The Y-coordinate of one corner.
- *x*2 - The opposite X-coordinate.
- *y*2 - The opposite Y-coordinate.
- *rad* - The radius of the corner curves.
- *color* - The color for the rectangle.
- *fill* - ***True*** for fill, ***False*** for outline.

```
procedure drawLine(self : RA8875_record; x1 : uint16; y1 : uint16; x2 : uint16;
                   y2 : uint16; color : R5G6B5_color);
```

Draw a line between two points.

- *self* - The RA8875 device to control.
- *x*1 - The X-coordinate of one end of the line.
- *y*1 - The Y-coordinate of one end of the line.
- *x*2 - The X-coordinate of the other end of the line.
- *y*2 - The Y-coordinate of the other end of the line.
- *color* - The color for the line.

```
procedure drawCircle(self : RA8875_record; x : uint16; y : uint16; rad : uint16;
                     color : R5G6B5_color; fill : boolean);
```

Draw a circle.

- *self* - The RA8875 device to control.
- *x* - The X-coordinate of the center of the circle.
- *y* - The Y-coordinate of the center of the circle.
- *rad* - The radius of the circle.
- *color* - The color for the circle.
- *fill* - ***True*** for fill, ***False*** for outline.

```
procedure drawTriangle(self : RA8875_record; x1 : uint16; y1 : uint16;
                       x2 : uint16; y2 : uint16; x3 : uint16; y3 : uint16;
                       color : R5G6B5_color; fill : boolean);
```

Draw a triangle.

- *self* - The RA8875 device to control.
- $x1$ - The X-coordinate of the first point.
- $y1$ - The Y-coordinate of the first point.
- $x2$ - The X-coordinate of the second point.
- $y2$ - The Y-coordinate of the second point.
- $x3$ - The X-coordinate of the third point.
- $y3$ - The Y-coordinate of the third point.
- *color* - The color for the triangle.
- *fill* - `True` for fill, `False` for outline.

```
procedure drawEllipse(self : RA8875_record; x : uint16; y : uint16; hRad : uint16;
                      vRad : uint16; color : R5G6B5_color; fill : boolean);
```

Draw an ellipse oriented either horizontally or vertically.

- *self* - The RA8875 device to control.
- *x* - The X-coordinate of the center of the circle.
- *y* - The Y-coordinate of the center of the circle.
- *hRad* - The radius in the horizontal direction.
- *vRad* - The radius in the vertical direction.
- *color* - The color for the ellipse.
- *fill* - `True` for fill, `False` for outline.

```
procedure drawEllipseSegment(self : RA8875_record; x : uint16; y : uint16;
                             hRad : uint16; vRad : uint16; seg : RA8875_ELLIPSE_PART
                             color : R5G6B5_color; fill : boolean);
```

Draw a segment of an ellipse oriented either horizontally or vertically.

- *self* - The RA8875 device to control.
- *x* - The X-coordinate of the center of the circle.
- *y* - The Y-coordinate of the center of the circle.
- *hRad* - The radius in the horizontal direction.
- *vRad* - The radius in the vertical direction.
- *seg* - Which segment to draw.

- *color* - The color for the ellipse segment.
- *fill* - *True* for fill, *False* for outline.

```
procedure waitPoll(self : RA8875_record; reg : uint8; flag : uint8);
```

Repeatedly reads a register and checks for a bit to be cleared. This is commonly used at the end of drawing routines to ensure that the drawing is complete before proceeding. This is intended primarily for use by other of the RA8875 routines and not by user code.

- *self* - The RA8875 device to check.
- *reg* - The register to check.
- *flag* - The set of bits to check. This value is ANDed with the register value and execution proceeds if the result is zero.

Touch Methods

```
procedure enableTouch(self : RA8875_record; state : boolean);
```

Enable or disable touch processing.

- *self* - The RA8875 device to configure.
- *state* - *True* to enable touch, *False* to disable it.

```
function checkTouched(self : RA8875_record) return boolean;
```

Checks if the RA8875 has recorded a touch event.

- *self* - The RA8875 device to check.
- Returns *True* if a touch event has occured.

```
procedure readTouchRaw(self : RA8875_record; x : out uint16; y : out uint16);
```

Reads the raw touch location.

- *self* - The RA8875 device to read.
- *x* - The raw X-coordinate of the touch location
- *y* - The raw Y-coordinate of the touch location.

```
procedure readTouchCal(self : RA8875_record; x : out uint16; y : out uint16);
```

Reads the calibrated touch location. Note that the calibration assumes a linear response.

- *self* - The RA8875 device to read.
- *x* - The calibrated X-coordinate of the touch location

- *y* - The calibrated Y-coordinate of the touch location.

```
procedure touchCalibrate(self : in out RA8875_record);
```

Run a touch calibration process. This requires the user to touch the top, bottom, left, and right edges of the touch screen. Based on this, the limits of the touch sensor are computed and added to the RA8875 object.

- *self* - The RA8875 device to calibrate.

```
procedure setTouchCalibration(self : in out RA8875_record; top : uint16;
                              bottom : uint16; left : uint16; right : uint16);
```

Set the touch sensor limits for the RA8875. This can be useful if the limits are already known and avoids running through the touch calibration process.

- *self* - The RA8875 device to calibrate.
- *top* - The screen top limit.
- *bottom* - The screen bottom limit.
- *left* - The screen left limit.
- *right* - The screen right limit.

```
procedure getTouchCalibration(self : RA8875_record; top : out uint16;
                              bottom : out uint16; left : out uint16;
                              right : out uint16);
```

Read the touch sensor limits from the RA8875. This can be useful to save the limits for use with `setTouchCalibration`.

- *self* - The RA8875 device to read.
- *top* - The screen top limit.
- *bottom* - The screen bottom limit.
- *left* - The screen left limit.
- *right* - The screen right limit.

Region and Layer Methods

```
procedure scroll(self : RA8875_record; hStart : uint16; vStart : uint16;
                 hEnd : uint16; vEnd : uint16; hOffset : uint16; vOffset : uint16);
```

Scroll a region on the display

- *self* - The RA8875 device to scroll.
- *hStart* - The region starting horizontal coordinate.

- *vStart* - The region starting vertical coordinate.
- *hEnd* - The region ending horizontal coordinate.
- *vEnd* - The region ending vertical coordinate.
- *hOffset* - The horizontal scroll amount.
- *vOffset* - The vertical scroll amount.

```
procedure setActiveWindow(self : RA8875_record; top : uint16; bottom : uint16;
                          left : uint16; right : uint16);
```

Sets the current active window on the display.

- *self* - The RA8875 device to configure.
- *top* - The top of the new active window.
- *bottom* - The bottom of the new active window.
- *left* - The left side of the new active window.
- *right* - The right side of the new active window.

```
procedure screenActive(self : RA8875_record);
```

Sets the entire display area to be the active window.

- *self* - The RA8875 device to configure.

```
procedure setLayers(self : RA8875_record; layer : RA8875_LAYER);
```

Sets the number of layers available for use. May be one or two. If two layers are selected, the graphics depths is reduced to allow two layers.

- *self* - The RA8875 device to configure.
- *layer* - *LAYER1* for one layer, *LAYER2* for two layers.

```
procedure selectLayer(self : RA8875_record; layer : RA8875_LAYER);
```

Selects which layer to use.

- *self* - The RA8875 device to configure.
- *layer* - The currently active layer.

```
procedure setLayerSetting0(self : RA8875_record;
                           scroll : RA8875_LTPR0_SCROLL_MODE;
                           float : boolean; display : RA8875_LTPR0_DISP_MODE);
```

Sets layer transparency register 0 in the RA8875.

- *self* - The RA8875 device to configure.
- *scroll* - Sets the scrolling mode.
- *float* - Enables or disables floating window transparency display.
- *display* - Sets the display mode for mixing layers 1 and 2.

Cursor Methods

```
procedure setTextCursorPos(self : RA8875_record; x : uint16; y : uint16);
```

Sets the text cursor position.

- *self* - The RA8875 device to configure.
- *x* - The X-coordinate of the cursor.
- *y* - The Y-coordinate of the cursor.

```
procedure setGraphCursorColors(self : RA8875_record; color0 : R3G3B2_color;
                               color1 : R3G3B2_color);
```

Set the graphics cursor colors.

- *self* - The RA8875 device to configure.
- *color*0 - The first color to use.
- *color*1 - The second color to use.

```
procedure setGraphCursorPos(self : RA8875_record; x : uint16; y : uint16);
```

Sets the graphics cursor position.

- *self* - The RA8875 device to configure.
- *x* - The X-coordinate of the cursor.
- *y* - The Y-coordinate of the cursor.

```
procedure setGraphCursor(self : RA8875_record; curs : RA8875_MWCR1_GCURS_SET;
                         data : RA8875_GCursor);
```

Sets the graphics cursor.

- *self* - The RA8875 device to configure.
- *curs* - The graphics cursor set.
- *data* - The bitmap for the graphics cursor.

```
procedure selectGraphCursor(self : RA8875_record; curs : RA8875_MWCR1_GCURS_SET;
                            enable : RA8875_MWCR1_GCURS_ENABLE);
```

Selects the graphics cursor.

- *self* - The RA8875 device to configure.
- *curs* - The graphics cursor set.
- *enable* - Enables or disables the graphics cursor.

Miscellaneous Methods

```
procedure fillScreen(self : RA8875_record; color : R5G6B5_color);
```

Fills the display with a solid color.

- *self* - The RA8875 device to configure.
- *color* - The color to use.

Chapter 5

Linux API Description

Linux defines some common methods for interfacing with devices. This is typically using device files. However, each platform may have different devices. So, the Linux API description describes the implemented common API. Then there is a discussion of Raspberry PI specific and BeagleBone Black specific items. Since the differences are manly in which device files to use, it should be fairly easy to add new boards.

5.1 Common

This section describes the implementation differences between the base common software and the Linux implementation.

5.1.1 BBS.embed.AIN.linux

As the analog inputs are handled using device files, the name of the device file has to be passed to a configure routine which then opens the file. When finished, the file should be closed.

```
procedure configure(self : in out Linux_AIN_record;
                    port : string);
```

Configures a Linux analog input. This needs to be done before the analog input can be used.

- *self* - The analog input to configure.
- *port* - The name of the device file for the analog input.

```
procedure close(self : in out Linux_AIN_record);
```

Closes a Linux analog input device file. This should be done when the software is finished with the device.

- *self* - The analog input to close.

5.1.2 BBS.embed.GPIO.Linux

As the general-purpose I/O pins are handled using device files (sometimes multiple files), the name of the device file has to be passed to a configure routine which then opens the file. When finished, the file should be closed.

```
procedure configure(self : in out Linux_GPIO_record;
                    pin : string; port : string; dir : direction);
```

In some cases, pins may have multiple uses. In these cases, a pin control file needs to be accessed to select the use for the pin.

- *self* - The GPIO device to configure.
- *pin* - The path to the pin control file for this GPIO.
- *port* - This is a path to a directory containing various GPIO control files.
- *dir* - The direction (input or output) for the GPIO pin.

```
procedure configure(self : in out Linux_GPIO_record;
                    port : string; dir : direction);
```

For dedicated pins without pin control files, a simpler form can be used for configuration.

- *self* - The GPIO device to configure.
- *port* - This is a path to a directory containing various GPIO control files.
- *dir* - The direction (input or output) for the GPIO pin.

```
procedure set_dir(self : in out Linux_GPIO_record;
                  port : String; dir : direction);
```

Sets the direction of a pin. This is used by the `configure` procedure, but can be used whenever a GPIO pin needs to change between input and output.

- *self* - The GPIO device to configure.
- *dir* - The direction (input or output) for the GPIO pin.

```
procedure close(self : in out Linux_GPIO_record);
```

Closes a GPIO device file. This should be done when the software is finished with the device.

- *self* - The GPIO device to close.

5.1.3 BBS.embed.i2c.linux

This package basically provides an Ada wrapper around a bunch of C `ioctl` calls.

```
procedure configure(self : in out linux_i2c_interface_record; i2c_file : string;
                    SCL : string; SDA : string);
```

Configure the I2C interface on a BeagleBone Black or other systems that have multiple functions on the I2C pins. This configuration procedure sets the pins to the I2C function.

- *self* - The I2C device to configure.
- *i2c_file* - The path to the I2C device file. This is opened as a C file.
- *SCL* - The path to the pin control file for the SCL pin for the I2C bus.
- *SDA* - The path to the pin control file for the SDA pin for the I2C bus.

```
procedure configure(self : in out linux_i2c_interface_record; i2c_file : string);
\indexfunc{configure}
```

Configure the I2C interface on a Raspberry PI or other systems that have dedicated pins for the I2C interface. This would also work on a system with shared pins if the pins had already been set to the I2C function.

- *self* - The I2C device to configure.
- *i2c_file* - The path to the I2C device file. This is opened as a C file.

```
procedure close(self : in out linux_i2c_interface_record);
```

Closes a I2C device file. This should be done when the software is finished with the device.

- *self* - The I2C device to close.

5.1.4 BBS.embed.log.linux

The Linux log implementation just prints the log messages to the console using `Ada.Text_IO.Put` and `Ada.Text_IO.Put_Line`. No new functions are added to the class.

5.1.5 BBS.embed.SPI.Linux

This is mostly Ada wrappers around some C I/O calls.

```
procedure configure(self : in out Linux_SPI_record; SPI_file : string;
                    SCL : string; SDA : string);
```

Configure the SPI interface on a BeagleBone Black or other systems that have multiple functions on the SPI pins. This configuration procedure sets the pins to the SPI function.
This function is not yet implemented. Do not attempt to use.

```
procedure configure(self : in out Linux_SPI_record; SPI_file : string);
```

Configure the SPI interface on a Raspberry PI or other systems that have dedicated pins for the SPI interface. This would also work on a system with shared pins if the pins had already been set to the SPI function.

- *self* - The SPI device to configure.
- *SPI_file* - The path to the SPI device file. This is opened as a C file.

```
procedure close(self : in out Linux_SPI_record);
```

Closes a SPI device file. This should be done when the software is finished with the device.

- *self* - The SPI device to close.

5.2 Raspberry Pi

This section covers items specific to the Raspberry Pi.

5.2.1 Initialization Script

This initialization script needs to be run after the Raspberry Pi is booted and before attempting to run software using this package. It contains shell commands to activate the GPIO pins and set the protection on the pin control files so that non-root users can write to them. It also sets the protection on the I2C device file for non-root access. The script is called `init-bbb.sh` and is in the project root directory. It needs to be run as root (typically using the `sudo` command).

5.2.2 `BBS.embed.RPI`

This package contains string constants for the available GPIO pins, I2C device, and SPI device. These constants should be used rather that using string literals for the path names.

5.3 BeagleBone Black

This section covers items specific to the BeagleBone Black. Note that the BeagleBone Black has more I/O options than the Raspberry Pi. This includes PWM outputs, analog inputs, and four on-board LEDs.

5.3.1 Initialization Script

This initialization script needs to be run after the BeagleBone Black is booted and before attempting to run software using this package. It contains shell commands to activate the various I/O options and set the protection on the pin control files and device files so that non-root users can write to them. The script is called `init-rpi.sh` and is in the project root directory. It needs to be run as root (typically using the `sudo` command). Read through the script if you are interested in more details about what it does. It also seems that the PWM devices get renumbered on boot. This script does some wildcard matches and creates hard links to the appropriate device files.

5.3.2 BBS.embed.BBB

This package contains string constants for the available I/O device files. These constants should be used rather than using string literals for the path names. They are more extensive than the Raspberry Pi's I/O. The comments in this package may give some insight into the I/O devices.

5.3.3 BBS.embed.LED

This package is for controlling the BeagleBone Black's four on-board LEDs. This will probably always be unique to the BeagleBone Black. This package hasn't changed much since the initial development.

```
subtype led_num is Integer range 0 .. 3;
```

Since there are four LEDs, the datatype `led_num` is defined to identify them.

```
type led_state is (off, on);
```

The datatype `led_state` is used to turn LEDs on or off.

```
type led_states is array (led_num) of led_state;
```

The datatype `led_states` combines a state for all of the LEDs into a single array.

```
procedure open(l : in led_num);
```

Open a specific LED device file. This needs to be done before the LED can be used.

- *l* - The LED to open.

```
procedure open;
```

Open all of the LED device files. This makes all LEDs ready to be used.

```
procedure set(l : in led_num; s : in led_state);
```

Sets the state of a specific LED.

- *l* - The LED to change.
- *s* - The state for the LED.

```
procedure set(led0 : in led_state; led1 : in led_state;
              led2 : in led_state; led3 : in led_state);
```

Sets the state of all LEDs by individually specifying the state.

- *led*0 - The state for LED 0.
- *led*1 - The state for LED 1.
- *led*2 - The state for LED 2.
- *led*3 - The state for LED 3.

```
procedure set(leds : in led_states);
```

Set the state of all LEDs by providing a `led_states` array.

- *leds* - The array containing states for all LEDs.

```
procedure close(l : in led_num);
\indexfunc{close}
```

Close a specific LED device file.

- *l* - The LED to close.

```
procedure close;
\indexfunc{close}
```

Close all of the LED device files.

5.3.4 BBS.embed.PWM

This package is for controlling the BeagleBone Black's PWM outputs.

```
type nanoseconds is range 0 .. integer'Last;
```

The Linux PWM driver uses period and duty-cycle (actually time high) expressed in nanoseconds. By the time that this is translated to the device, the actual resolution may be something different. It is also not clear what the maximum value of period may be. The maximum value may also depend on the specific PWM device being accessed. Some experimentation would be in order to determine what works for your application.

```
procedure configure(self : not null access PWM_record'class;
                    pin : string; index : pwm_range);
```

Configure a PWM device. The pin control file and the PWM number must correspond, otherwise things will not work correctly. Pin should be one of the pin constants.

- *self* - The PWM object to configure.
- *pin* - The path to the pin control file for the PWM.
- *index* - The PWM number.

```
procedure enable(self : not null access PWM_record'class; state : boolean);
```

Enables or disables a PWN device.

- *self* - The PWM object to configure.
- *state* - `True` to enable, `False` to disable.

```
procedure set_period(self : not null access PWM_record'class; period : nanoseconds);
```

Set the period of the PWM. This is the value used by Linux in nS. Must be zero or positive. Some of the PWM devices may have additional restrictions on the range.

- *self* - The PWM object to configure.
- *period* - The time in nS between pulses on the PWM given as `nanoseconds`.

```
procedure set_period(self : not null access PWM_record'class; period : Duration);
```

Set the period of the PWM. This is the value used by Linux in nS. Must be zero or positive. Some of the PWM devices may have additional restrictions on the range.

- *self* - The PWM object to configure.
- *period* - The time in nS between pulses on the PWM given as `Duration`.

```
procedure set_high(self : not null access PWM_record'class; high : nanoseconds);
```

Set the time that the output is in the high state. This must be between zero and the period.

- *self* - The PWM object to configure.
- *high* - The time the output is in the high state in nS given as `nanoseconds`.

```
procedure set_high(self : not null access PWM_record'class; high : Duration);
```

Set the time that the output is in the high state. This must be between zero and the period.

- *self* - The PWM object to configure.
- *high* - The time the output is in the high state in nS given as `Duration`.

```
procedure set_rate(self : not null access PWM_record'class; rate : BBS.units.freq_hz
```

Sets the rate in Hz. This is essentially the inverse of `set_period`. The rate is given as a floating point number and is converted to an integer number of nanoseconds.

- *self* - The PWM object to configure.
- *rate* - The frequency of the PWM.

```
procedure set_duty(self : not null access PWM_record'class; duty : float)
     with pre => (duty >= 0.0) and (duty <= 100.0);
```

Sets the duty cycle. The duty cycle is a percentage of the period that the output is high. The input is a floating point number and is converted to the appropriate number of nanoseconds.

- *self* - The PWM object to configure.
- *duty* - The duty cycle of the PWM in percent (this must be between 0.0 and 100.0 inclusive).

Chapter 6

Arduino Due API Description

The Arduino Due is based on the Atmel SAM3X8E ARM Cortex-M3 CPU[1]. This has not been worked on lately due to other projects (one day, I hope to get back to it) and should be considered to be experimental, but may be of some use to others working with an Arduino Due. The packages listed here may depend on some `sam3x8e.*` packages. The `sam3x8e.*` are autogenerated from the `ATSAM3X8E.svd` file and they provide many of the low-level constants and datatypes for access the SAM3X hardware.

The following sections describe device specific additions or alterations to the common classes as well as unique device specific routines.

6.1 `BBS.embed.AIN.due`

This section describes the Arduino Due implementation of analog inputs.

Note that enabling an analog input will supersede any other use for that pin. The architecture has defined 16 analog inputs, 0-15, with input 15 being used to measure the CPU temperature.

Do not use channels marked as unused. It may cause problems. Using AD15, the CPU temperature sensor also seems to cause trouble with tasking.

```
subtype AIN_Num is Integer range 0 .. 11;
```

Define a subtype for the Arduino analogs inputs. These are mapped to the analog channels via an array defined in the private section of this package. Using this prevents the unused channels from being accessed.

```
procedure enable_ain(self : Due_AIN_record; b : Boolean);
```

Enable or disable a specified analog input object.

- *self* - The analog input device to configure.
- *b* - *`True`* to enable, *`False`* to disable.

```
procedure setup_ain;
```

Setup the analog to digital controller. This needs to be done before using the analog input.

SAM3X8E Input	Arduino Due Pin
AD0	AD07.
AD1/WKUP1	AD06.
AD2	AD05.
AD3	AD04.
AD4	AD03.
AD5	AD02.
AD6	AD01.
AD7	AD00.
AD8	Unused
AD9	Unused
AD10	AD08.
AD11	AD09.
AD12	AD10.
AD13	AD11/TXD3.
AD14	Unused
AD15	CPU temp sensor.

Table 6.1: SAM3X8E Analog Input Assignment to Arduino Due Pins

```
procedure enable_ain(c : AIN_Num; b : Boolean);
```

Enable or disable a specified analog input channel. Generally, the object oriented call should be used instead.

- *c* - The Arduino analog input number to configure.
- *b* - *True* to enable, *False* to disable.

```
procedure start;
```

Start an analog to digital conversion.

```
procedure free_run(b : Boolean);
```

Set free running conversion.

- *b* - *True* to enable free running conversion, *False* to disable it.

```
function get(c : AIN_Num) return UInt12;
```

Read an ADC value from a channel. Generally, the object oriented call should be used instead.

- *c* - The Arduino analog input number
- Returns the 12-bit converted value.

6.2 BBS.embed.due.dev

This package contains a set of constants for the device numbers for the SAM3X devices. It was collected here as it wasn't really available anywhere else. This is primarily for internal use only.

6.3 BBS.embed.Due.GPIO

This package contains a collection of pin objects for the Arduino Pins. If memory is tight for your application, just copy out the ones that you need. There does not seem to be much rhyme or reason for how the CPU pins are assigned to the Arduino pins.

6.4 BBS.embed.due.serial

This package is the root of the serial I/O functionality on the Arduino Due. It has four of its serial ports wired to the headers. These ports are numbered 0 through 3. Port 0 is also wired to the programming USB connector and is used as the default port, if none is specified.

Communication is available in both polled and interrupt driven variants. Polled is unbuffered and has very little memory overhead, but puts more load on the CPU. It is most useful when memory is tight or serial I/O needs are minimal. It would also be useful as a method of last resort for printing messages from an exception handler.

Interrupt driven copies messages into a buffer and uses an interrupt handler to feed characters into the UART as needed. The main thread of software can continue processing while this is happening. This provides more processing for user software at the expense of requiring memory for the buffers.

This package contains definitions and code that is common to both methods. The interrupt or polled specific definitions and code are in the appropriate sub-package.

```
type port_id is new Integer range 0 .. 3;
```

This type is used to identify which of the four available serial ports is being used.

```
procedure init(baud_rate : SAM3x8e.UInt32);
```

Initialize the serial port attached to the programming USB port. It looks like this is equivalent to channel zero.

- *baud_rate* - The desired baud rate for the USB programming port.

```
procedure init(chan_num : port_id; baud_rate : SAM3x8e.UInt32);
```

Initialize the serial port attached to the specified channel.

- *chan_num* - The channel number of the port to initialize.
- *baud_rate* - The desired baud rate for the USB programming port.

```
function tx_ready return Boolean;
```

Check if the USB programming port transmitter is ready.

- Returns *True* if the transmitter is ready.

```
function tx_ready(chan : port_id) return Boolean;
```

Check if the specified serial port transmitter is ready.

- *chan* - The channel number of the serial port to check.
- Returns *True* if the transmitter is ready.

```
function rx_ready return Boolean;
```

Check if the USB programming port receiver is ready.

- Returns *True* if the transmitter is ready.

```
function rx_ready(chan : port_id) return Boolean;
```

Check if the specified serial port receiver is ready.

- *chan* - The channel number of the serial port to check.
- Returns *True* if the receiver is ready.

```
function tx_empty return Boolean;
```

Check if the USB programming port transmitter is empty.

- Returns *True* if the transmitter is empty.

```
function tx_empty(chan : port_id) return Boolean;
```

Check if the specified serial port transmitter is empty.

- *chan* - The channel number of the serial port to check.
- Returns *True* if the transmitter is empty.

6.5 BBS.embed.due.serial.int

This is an interrupt driven serial package that can be used to print text with reduced overhead for the user code. Characters are written to a buffer which is sent to the UART under control of interrupts.

An even more processor efficient option would be to use DMA for handling the I/O. This is left for a future project.

There are two goals for this driver. The first is to provide console I/O for a person to communicate with the device. The second is to be able to communicate with other devices. There are many features that could be added, but it should be kept fairly simple and primitive.

Many of the routines can be called with a channel number, a serial port object, or nothing (this is similar to `Ada.Text_IO`). If no channel or object is provided, the default serial port (USB programming port, or channel 0) is used.

```
function init(c : port_id; baud : SAM3x8e.UInt32) return serial_port;
```

Initialize a channel and return the serial port object for that channel.

- *c* - The channel number.

- *baud* - The baud rate.
- Returns the serial port object for the channel.

```
function get_port(c : port_id) return serial_port;
```

Return the serial port object for a channel.

- *c* - The channel number.
- Returns the serial port object for the channel.

```
procedure put(c : Character);
procedure put(chan : port_id; c : Character);
procedure put(self : not null access serial_port_record'class; c : Character);
```

Send a character to a serial port. There are three options. The default with no port or channel identifier sends the character to port 0 (the USB programming port).

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- *c* - The character to send.

```
procedure put(s : string);
procedure put(chan : port_id; s : string);
procedure put(self : not null access serial_port_record'class; s : String);
```

Send a string to a serial port. There are three options. The default with no port or channel identifier sends the character to port 0 (the USB programming port).

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- *s* - The string to send.

```
procedure put_line(s : string);
procedure put_line(chan : port_id; s : string);
procedure put_line(self : not null access serial_port_record'class; s : String);
```

Send a string to a serial port, followed by a CR/LF combination. There are three options. The default with no port or channel identifier sends the character to port 0 (the USB programming port).

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- *s* - The string to send.

```
procedure new_line;
procedure new_line(chan : port_id);
procedure new_line(self : not null access serial_port_record'class);
```

Send a CR/LF combination to a serial port for a new line. There are three options. The default with no port or channel identifier sends the character to port 0 (the USB programming port).

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.

```
procedure enable_rs485(chan : port_id; d : BBS.embed.GPIO.Due.Due_GPIO_ptr)
  with pre => ((d.ctrl = BBS.embed.GPIO.Due.PIOA'Access) or
                  (d.ctrl = BBS.embed.GPIO.Due.PIOB'Access) or
                  (d.ctrl = BBS.embed.GPIO.Due.PIOC'Access) or
                  (d.ctrl = BBS.embed.GPIO.Due.PIOD'Access));
procedure enable_rs485(self : not null access serial_port_record'class;
                       d : BBS.embed.GPIO.Due.Due_GPIO_ptr)
  with pre => ((d.ctrl = BBS.embed.GPIO.Due.PIOA'Access) or
                  (d.ctrl = BBS.embed.GPIO.Due.PIOB'Access) or
                  (d.ctrl = BBS.embed.GPIO.Due.PIOC'Access) or
                  (d.ctrl = BBS.embed.GPIO.Due.PIOD'Access));
```

Enables RS-485 mode on an I/O channel. It requires an initialized digital I/O pin record. If d.ctrl isn't pointing to a PIO control record, bad things can happen, so make this a precondition. This hasn't really been tested. Use with caution

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- *d* - The GPIO pin record to use as a RS-485 control pin.

```
procedure flush(chan : port_id);
procedure flush(self : not null access serial_port_record'class);
```

Wait until transmit buffer is empty.

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.

```
procedure rx_enable(chan : port_id; b : Boolean);
procedure rx_enable(self : not null access serial_port_record'class; b : Boolean);
```

Enable or disable the RX interrupt.

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- *b* - `True` to enable, `False` to disable.

```
function rx_ready return Boolean;
function rx_ready(chan : port_id) return Boolean;
function rx_ready(self : not null access serial_port_record'class)
   return Boolean;
```

Check to see if characters are available in the buffer

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- Returns *True* if characters are available in the buffer.

```
function get return Character;
function get(chan : port_id) return Character;
function get(self : not null access serial_port_record'class)
   return Character;
```

Read a character from the buffer. If no character is available, wait until one is and then read it.

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- Returns the next character from the buffer.

```
function peek return Character;
function peek(chan : port_id) return Character;
function peek(self : not null access serial_port_record'class)
   return Character;
```

Return the next character in the receive buffer without removing it. If no character is available, wait until one is and then read it.

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- Returns the next character from the buffer.

```
procedure get_line(s : out String; l : out Natural);
procedure get_line(chan : port_id; s : in out String; l : out Natural);
procedure get_line(self : not null access serial_port_record'class;
                   s : in out String; l : out Natural);
```

Return a line of text, waiting as needed until a line is complete.

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- Returns the next line of text from the buffer.

```
procedure set_echo(chan : port_id; b : Boolean);
procedure set_echo(self : not null access serial_port_record'class; b : Boolean);
```

Enables or disables echo of received characters.

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- *b* - `True` to enable echoing of characters.

```
procedure set_del(chan : port_id; b : Boolean);
procedure set_del(self : not null access serial_port_record'class; b : Boolean);
```

Enables or disables deleting of received characters.

- *chan* - The serial port's channel number.
- *self* - The serial port object to use.
- *b* - `True` to enable deleting of characters.

6.6 BBS.embed.due.serial.polled

This is a very simple serial package that can be used to print some debugging information. It uses polling to wait for each character to be passed to the UART. This makes it not particularly efficient. However, it will work even if the interrupts get all screwed up, which makes it useful in a last resort exception handler to get a message printed.

```
procedure put(c : Character);
procedure put(chan : port_id; c : Character);
```

Write a character to a serial port. It does a busy wait on the `UART_SR TXRDY` (transmit ready) bit. It does a loop until the value of the bit is 1 and then write the character. If no channel is specified in the call, channel 0 (the USB programming port) is used.

- *chan* - The serial port's channel number.
- *c* - The character to transmit.

```
procedure put(s : string);
procedure put(chan : port_id; s : string);
```

Write a string to a serial port. If no channel is specified in the call, channel 0 (the USB programming port) is used.

- *chan* - The serial port's channel number.
- *s* - The string to transmit.

```
procedure put_line(s : string);
procedure put_line(chan : port_id; s : string);
```

Write a string to a serial port, followed by a CR/LF combination. If no channel is specified in the call, channel 0 (the USB programming port) is used.

- *chan* - The serial port's channel number.
- *s* - The string to transmit.

```
function get return Character;
function get(chan : port_id) return Character;
```

Read a character from serial port - wait for one to be present, if necessary.

- *chan* - The serial port's channel number.
- Returns the character from the serial port.

6.7 BBS.embed.due

This is the root package for the Arduino Due specific packages. It currently has no other contents.

6.8 BBS.embed.GPIO.Due

This is a description of the Arduino Due specific GPIO functions. For the Arduino Due, pin records in the package `BBS.embed.Due.GPIO` are used to define each pin, rather than using device files like Linux based systems.

```
type direction is (gpio_input, gpio_output, funct_a, funct_b);
```

The datatype `direction` is used to identify the function of a pin. As is typical, most pins have multiple possible functions.

```
procedure config(self : in out Due_GPIO_record;
                 pin : Due_GPIO_record; dir : direction);
```

Configures a pin to be controlled by the PIO controller. Output is enabled or disabled based on the value of dir.

- *self* - The GPIO record to configure.
- *pin* - This should be one of the pin records from the package `BBS.embed.Due.GPIO`. The data from this record are used to populate *self*.
- *dir* - Indicates the function for the pin.

```
procedure config(self : in out Due_GPIO_record; dir : direction);
```

Configures a pin to be controlled by the PIO controller. Output is enabled or disabled based on the value of dir. The values in *self* are assumed to already be assigned. Typically, this would be used to re-configure a pin.

- *self* - The GPIO record to configure.

- *dir* - Indicates the function for the pin.

```
procedure pullup(self : Due_GPIO_record; val : Bit);
```

Enable or disable pullup on a pin.

- *self* - The GPIO record to configure.
- *val* - Set to 1 to enable pullup. Set to 0 to disable pullup.

6.9 BBS.embed.i2c.due

This is a description of the Arduino Due specific I2C bus functions.

```
type speed_type is (low100, high400);
```

The datatype `speed_type` is used to select the speed of the I2C interface.

```
type port_id is  new Integer range 0 .. 1;
```

The datatype `port_id` is used to select one of the two I2C ports on the Arduino Due.

```
function get_interface(d : port_id) return due_i2c_interface;
```

The interface records are declared in the private section. This returns an access to the record for the specified channel.

- *d* - The specified I2C channel.
- Returns an access to the interface record for the specified channel.

```
procedure init(chan : port_id; speed : speed_type);
```

Initializes the specified I2C channel and sets the speed.

- *d* - The specified I2C channel.
- *speed* - The selected speed for the channel.

```
function get_activity(self : in out due_i2c_interface_record) return uint32;
```

Returns the I2C activity counter for the specified channel. This is primarily useful for debugging to ensure that the device is actually processing transactions.

- *d* - The specified I2C channel.
- Returns the activity counter for the specified channel.

```
function is_busy(self : in out due_i2c_interface_record) return Boolean;
```

Returns the busy status for the specified channel.

- *d* - The specified I2C channel.
- Returns the busy status for the specified channel. ***True*** for busy, ***False*** for not busy.

6.10 BBS.embed.log.due

The Arduino Due specific logging uses the interrupt driven serial driver to sent the log messages to serial channel 0. No Arduino Due specific routines exist.

6.11 BBS.embed.SPI.Due

This is a description of the Arduino Due specific SPI bus functions. I don't think that this has been tested, but it might work.

```
procedure configure(self : in out Due_SPI_record);
```

Configure the SPI interface. The SAM3X8E core actually has two SPI interfaces, but SPI1 does not seem to be present on the chip pinout.

- *self* - The SPI device record to configure.

List of Datatypes

accelerations, 26
accelerations_g, 26
addr7, 6
ADS1015_config, 15
AIN_Num, 63

buff_index, 8
buffer, 8

CMD_type, 31
comp_latch, 14
comp_mode_type, 14
comp_polarity, 14
comp_que_type, 14
cvt_type, 20

data_rate_type, 13
direction, 71

err_code, 8

fsd, 22

int12, 6

led_num, 60
led_state, 60
led_states, 60

magnetism, 26
magnetism_gauss, 26
Mode_type, 31
mode_type, 14
mux_mode_type, 13

nanoseconds, 61

pga_type, 13
port_id, 65, 72

R5G6B5_color, 39
RA8875_ELLIPSE_PART, 39
RA8875_FNCR0_Code_Page, 39
RA8875_GCursor, 40
RA8875_LAYER, 38
RA8875_LTPR0_DISP_MODE, 40
RA8875_LTPR0_SCROLL_MODE, 40
RA8875_MWCR0_CURDIR, 38
RA8875_MWCR0_MODE, 38
RA8875_MWCR1_GCURS_ENABLE, 39
RA8875_MWCR1_GCURS_SET, 39
RA8875_MWCR1_WRITE_DEST, 39
RA8875_sizes, 38
rotations, 22
rotations_dps, 22

servo_range, 36
speed_type, 72

uint12, 6

List of Functions/Procedures

accel_data_ready, 29

calibrate_accel, 27
change_config, 15
checkTouched, 51
close, 56–59
config, 71
configure, 15, 17, 19, 23, 26, 27, 31–33, 36, 42, 56–59, 61, 73
conversion_done, 16

data_ready, 18, 20, 25
disable, 10
drawCircle, 47, 49
drawColor, 46
drawEllipse, 47, 50
drawEllipseSegment, 48, 50
drawLine, 47, 49
drawRect, 46, 48
drawRndRect, 46, 48
drawTriangle, 47, 49

enable, 10, 61
enable_ain, 63, 64
enable_rs485, 68
enableTouch, 51

fillScreen, 55
flush, 68
free_run, 64

get, 7, 11, 64, 69, 71
get_accel_status, 28
get_acceleration_x, 27, 28
get_acceleration_y, 27, 28
get_acceleration_z, 27, 28
get_accelerations, 28
get_activity, 72
get_data, 35
get_dir, 34
get_hum, 19
get_interface, 72
get_line, 69
get_mag_status, 30
get_magnet_x, 29, 30
get_magnet_y, 29, 30
get_magnet_z, 29, 30
get_magnetism, 30
get_polarity, 34
get_port, 67
get_press, 19, 21
get_pullup, 35
get_raw, 18
get_result, 17
get_rotation_x, 23, 24
get_rotation_y, 23, 24
get_rotation_z, 23, 24
get_rotations, 24
get_status, 25
get_t_fine, 18
get_temp, 18, 20, 21, 23, 24, 29
getTouchCalibration, 52
GPIOX, 42
graphicsMode, 46

highByte, 6
hwReset, 41

init, 12, 65, 66, 72
is_busy, 72

lowByte, 6

mag_data_ready, 31
measure_offsets, 25

new_line, 68

open, 60

peek, 69
present, 15, 17, 33
pullup, 72
put, 10, 67, 70
put_line, 11, 67, 71
PWM1config, 43
PWM1out, 43
PWM2config, 43
PWM2out, 43

read, 8, 9
read_data, 18, 33
readData, 41
readm1, 8
readm2, 8
readReg, 42
readStatus, 41
readTouchCal, 51
readTouchRaw, 51
rx_enable, 68
rx_ready, 66, 69

screenActive, 53
scroll, 52

selectGraphCursor, 54
selectLayer, 53
set, 7, 11, 31, 32, 36, 60, 61
set_1shot, 16
set_bridge_a, 12
set_bridge_b, 12
set_continuous, 16
set_data, 33, 35
set_del, 70
set_delay, 12
set_dir, 32, 34, 57
set_duty, 62
set_echo, 70
set_full_off, 37
set_full_on, 37
set_gain, 16
set_high, 62
set_mux, 16
set_period, 61, 62
set_polarity, 34
set_pullup, 35
set_rate, 62
set_servo, 38
set_servo_range, 37
setActiveWindow, 53
setDisplay, 42
setDisplayCtrl, 43
setGraphCursor, 54
setGraphCursorColors, 54
setGraphCursorPos, 54
setLayers, 53
setLayerSetting0, 53
setSleep, 42
setTextCursorPos, 54
setTouchCalibration, 52
setup, 40
setup_ain, 63
setWriteCtrl0, 43
setWriteCtrl1, 44
sleep, 37
start, 64
start_conversion, 16, 17, 20
step, 12
stepper_off, 12
swReset, 41

textColor, 44
textMode, 44
textSetAttribute, 45
textSetCodePage, 45
textSetFontWidth, 45
textSetLineHeight, 45
textWrite, 45
touchCalibrate, 52
tx_empty, 66
tx_ready, 65

uint12_to_int12, 6

waitPoll, 51
write, 9, 10
writeCmd, 41
writeData, 41
writem1, 9
writem2, 9
writeReg, 41

Bibliography

[1] Atmel. SAM3X/SAM3A series SMART ARM-based MCU, March 2015.

[2] Bosch. BMP180 digital pressure sensor, April 2013.

[3] Bosch. BME280 combined humidity and pressure sensor, February 2024.

[4] Texas Instruments. ADS1015 12-bit, 3.3-ksps, 4-channel, delta-sigma ADC with PGA, oscillator, VREF, comparator, and I2C, January 2018.

[5] Microchip. MCP23008 8-bit I/O expander with serial interface, 2007.

[6] Microchip. MCP4725 12-bit digital-to-analog converter with EEPROM memory, 2009.

[7] Microchip. MCP23017 16-bit I/O expander with serial interface, 2016.

[8] RAiO. RA8875 character/graphic TFT LCD controller, April 2016.

[9] NXP Semiconductors. PCA9685 16-channel, 12-bit PWM fm+I^2C-bus LED controller, April 2015.

[10] STMicroelectronics. L3GD20 low power 3-axis gyroscope, I2C/SPI digital output, February 2013.

[11] STMicrosystems. LSM303DLHC ultra-compact high-performance ecompass module: 3d accelerometer and 3d magnetometer, November 2013.

[12] Toshiba. TB6612FNG driver IC for dual DC motor, October 2014.