Users's Manual for Device Interfaces

Brent Seidel
Phoenix, AZ

August 27, 2024

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

Contents

Chapter 1

Introduction

1.1 About the Project

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

1.2 License

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

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

Chapter 2

How to Obtain

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

2.1 Dependencies

2.1.1 bbs_embed_common

Ada Libraries

The following Ada packages are used:

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

Other Libraries

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

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

2.1.2 bbs_embed_linux

Ada Libraries

The following Ada packages are used:

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

Other Libraries

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

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

2.1.3 bbs_embed_due

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

Ada Libraries

The following Ada packages are used:

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

SAM3x8e Stuff

The following SAM3x8e hardware definition packages are used:

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

Other Libraries

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

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

Chapter 3

Usage Instructions

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

3.1 Linux Based Raspberry Pi and BeagleBone Black

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

3.2 Arduino Due

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

Chapter 4

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.

4.1 Common

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.

4.1.1 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.

Analog Inputs

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

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

Read the value of the specified analog to digital converter.

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

General-Purpose Input/Output (GPIO)

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

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

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

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

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

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

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

I2C Bus

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

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

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

- `buffer` is an array of `uint8` and bounds of `buff_index`. It is used for buffering data for I2C bus transfers.

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

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

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

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

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

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

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

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

SPI Bus

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

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

Writes a byte to the SPI bus.

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

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

Reads a byte from the SPI bus.

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

4.1.2 Higher-Level Device Drivers

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

`BBS.embed.gpio.tb6612`

This is a driver for the Toshiba TB6612 dual DC motor controller [2]. 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.

`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 [1]. This converter has a wide variety of configuration options. Refer to the datasheet for details. A number of constants have been defined to support the various configuration options (see Tables 4.1, 4.2. 4.3, and 4.4).

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

Table 4.1: Constants for ADS1015 Mux Mode Configuration

Constant	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: Constants for ADS1015 Programmable Gain Amplifier Configuration

Constant	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: Constants for ADS1015 Data Rate Configuration

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

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

Constant	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: Constants for ADS1015 Comparator Queue Configuration

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

Table 4.5: Constants for ADS1015 Miscellaneous Configuration

- `os` - Operational status, used to start a conversion if in single shot mode. Don't use when setting configuration.
- `mux` - The mux mode (see Table 4.1).
- `pga` - The programmable gain type (see Table 4.2).
- `mode` - Conversion mode (see Table 4.5).
- `dr` - The data rate (see Table 4.3).
- `comp_mode` - The comparator mode (see Table 4.5).
- `comp_pol` - The comparator polarity (see Table 4.5).
- `comp_lat` - The comparator latching (see Table 4.5).
- `comp_que` - The comparator queue configuration (see Table 4.4).

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

Initializes the device to the default configuration.

- *self* - The device to initialize.

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

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

Initialize the device using the specified configuration.

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

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

Changes the device configuration to new values

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

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

Changes only the mux mode configuration.

- *self* - The device to modify.
- *mux* - The new mux mode configuration value.
- *error* - The I2C error code.

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

Changes only the converter gain value.

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

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

Sets the converter to operate in continuous mode.

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

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

Sets the converter to operate in single shot mode.

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

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

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

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

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

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

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

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

Returns the conversion value.

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

4.2 Linux

4.2.1 Raspberry Pi

4.2.2 BeagleBone Black

4.3 Arduino Due

Chapter 5

Other Stuff

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

Bibliography

[1] Texas Instruments. 12-bit, 3.3-ksps, 4-channel, delta-sigma adc with pga, oscillator, vref, comparator, and i2c, January 2018.

[2] Toshiba. Tb6612fng driver ic for dual dc motor, October 2014.