libarduino

Version 1.0.0 – 16 June 2016
Kit Bishop

Document History

Version	Date	Change Details
Version 1.0.0	16 June 2016	First released version

Contents

1. Overview

libarduino is C++ code that runs on the Omega for access to connected Arduino devices.

The rationale for producing this code to provide Omega access to Arduino I/O and to general functionality of code running on the Arduino.

Notes:

- The **libarduino** library uses I2C to communicate with the Arduino via the **libnewi2c** library that is available and documented at https://github.com/KitBishop/Omega-GPIO-I2C-Arduino/tree/master/libnewi2c
- Access to an Arduino from the Omega requires the Arduino to be running code from the **Arduino_Omega** library that is available and documented at https://github.com/KitBishop/Omega-GPIO-I2C-Arduino/tree/master/arduino_omega

libarduino consists of static and dynamic link libraries containing the classes used to interact with Arduino devices.

These library and the main C++ classes are described in more details in this document.

The library was developed on a KUbuntu-14.04 system running in a VirtualBox VM and uses the OpenWrt toolchain for building the code:

The toolchain used can be found at:

- https://s3-us-west-2.amazonaws.com/onion-cdn/community/openwrt/OpenWrt-Toolchain-ar71xx-generic_gcc-4.8-linaro_uClibc-0.9.33.2.Linux-x86_64.tar.bz2

and details of its setup and usage can be found at:

- https://community.onion.io/topic/9/how-to-install-gcc/22

libarduino comes with **NO GUARANTEES** ☺ but you are free to use it and do what you want with it.

2. Files Supplied

libarduino is supplied in files in a GitHub repository at https://github.com/KitBishop/Omega-GPIO-I2C-Arduino/tree/master/libarduino. This repository contains the following important directories and files:

- **libarduino.pdf** – this documentation as a PDF file
- **Makefile** – the Makefile for **libarduino** library
- **hdr** – directory containing header (***.h**) files for **libarduino** library
- **src** – directory containing source (***.cpp**) files for **libarduino** library
- **bin** – directory containing the built library code:
  - **dynamic/libarduino.so** – the dynamic link version of the library
  - **static/libarduino.a** – the static link version of the library

3. Usage and Installation

Installing and using the library is simple. It primarily consists of linking your program that uses the library and for the dynamic link library, copying the library to a suitable location on your Omega.

3.1. Using libarduino.a static library

To use **libarduino.a** static library you simply need to statically link your program to that library file. Since **libarduino** library makes use of **libnewi2c** library for the I2C communication and **libnewgpio** library for GPIO access for interrupt signalling from the Arduino, you will also need to statically link to **libnewi2c.a** and **libnewgpio.a**

Then your program can be copied to and run on the Omega.

3.2. Using and Installing libarduino.so dynamic library

To use **libarduino.so** dynamic library you need to dynamically link your program to that library file. Since **libarduino** library makes use of **libnewi2c** library for the I2C communication and **libnewgpio** library for GPIO access for interrupt signalling from the Arduino, you will also need to dynamically link to **libnewi2c.so** and **libnewgpio.so**

For any program that uses **libarduino.so** the library file needs to be copied to the **/lib** directory on your Omega. The library files **libnewi2c.so** and **libnewgpio.so** will also need to be copied to the **/lib** directory on your Omega.

Alternatively, you can copy the libraries to any location that may be set up in any **LD_LIBRARY_PATH** directory on your Omega. For example, I use the following for testing:

- Created directory **/root/lib**
- Copied the libraries to **/root/lib**
- Added the following lines to my **/etc/profile** file:

```
LD_LIBRARY_PATH=/root/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

4. Using Makefile

A **Makefile** is supplied that can be used to build the library.

4.1. Modify Makefile

The **Makefile** will need modifying:

- You <u>**NEED**</u> to and <u>**MUST**</u> change **TOOL_BIN_DIR** to the "bin" directory of your OpenWrt uClibc toolchain. E.G. make appropriate change to **<xxxx>** in:

  TOOL_BIN_DIR=<xxxx>/OpenWrt-Toolchain-ar71xx-generic_gcc-4.8-linaro_uClibc-0.9.33.2.Linux-x86_64/toolchain-mips_34kc_gcc-4.8-linaro_uClibc-0.9.33.2/bin

- You <u>**MAY**</u> need to change **LIBNEWI2C_DIR** to relative directory of libnewi2c if you are not using the sources as originally supplied.
  The default if using the standard **source** directory structure as supplied is:

  LIBNEW-GPIO_DIR=../libnewi2c

- You <u>**MAY**</u> need to change **LIBNEWGPIO_DIR** to relative directory of libnewgpio if you are not using the sources as originally supplied.
  The default if using the standard **source** directory structure as supplied is:

  LIBNEW-GPIO_DIR=../libnewgpio

4.2. Makefile targets

The **Makefile** implements the following set of targets:

- **make**
  The default target. Performs a complete build of both static and dynamic link versions of the library.
  This is directly equivalent to:
  make static dynamic

- **make static**
  Performs a complete build of just the static link version of the library.

- **make dynamic**
  Performs a complete build of just the dynamic link version of the library.

- **make clean**
  Removes all previous build files, both static and dynamic link versions.
  This is directly equivalent to:
  make clean-static clean-dynamic

- **make clean-static**
  Removes all previous build files for static link versions only
  .

- **make clean-dynamic**
  Removes all previous build files for dynamic link versions only

If the following is added to the **make** command line:

builddep=1

then **libnewi2c** and **libnewgpio** libraries that this library depends on will also be built before building the programs.

5. Omega – Arduino Connection

The connection between the Omega and an Arduino use I2C communication. This section covers three aspects of the connection involved in the communication.

- **Physical Connection** – the wiring involved in connecting the two
- **Logical Connection** – how the two are logically connected via I2C
- **Functional Connection** – functional aspects of communication over the connection

5.1. Physical Connection

The Omega and an Arduino need to be connected physically. This uses I2C connection using three required physical connections and one optional physical connection.

- **Required Physical Connections**
  Since communication is via I2c, the two devices need to be connected on the I2C bus, this requires the following:
  - **I2C SCL**: Omega GPIO Pin **20** <--> Arduino **SCL** pin. This is normally Arduino Analog pin **A5** on an Arduino Uno but some variants of Arduino use a different pin or provide a specific **SCL** pin
  - **I2C SDA**: Omega GPIO Pin **21** <--> Arduino **SDA** pin. This is normally Arduino Analog pin **A4** on an Arduino Uno but some variants of Arduino use a different pin or provide a specific **SDA** pin
  - **Ground**: Omega GND <--> Arduino GND

  When the Omega is plugged in to the Arduino Dock, these connections are automatically established. Any number of additional Arduinos can be connected simultaneously (but see under **Logical Connection** below) by making the necessary connections.

- **Optional Physical Connections**
  Because of the nature of the I2C communications, the main communication is effectively one way under control of the Omega (see **Logical Connection** below). However, a mechanism has been provided in the code that allows an Arduino to signal to the Omega that it has something to communicate. This is done by connecting any suitable Arduino IO pin to any suitable Omega GPIO pin.

  When this is done, the code provides a method by which the Arduino can provide data to be made available to the Omega and signal this fact by changing the state of the Arduino pin. The Omega catches this signal by using interrupt handling on its GPIO pin and from that uses I2C to query for the signalled data.

  So long as the Arduino pin and Omega GPIO pin are connected and referred to in the relevant calls to the code, all the handling of the communications is dealt with seamlessly by the provided code (see code description below and for the Arduino code in **Arduino_Omega** library)

5.2. Logical Connection

As has been mentioned, the Omega – Arduino connection uses I2C.

When communicating on I2C one end acts as the I2C **master** and the other end acts as an I2C **slave**.

An I2C **master** can communicate with multiple I2C **slaves**, the **master** specifies the I2C **address** of the **slave** to communicate with it. All communication is initiated by the **master**.

In Omega – Arduino communication:

- The Omega is the I2C master – when the Omega communicates with an Arduino, it uses the relevant I2C **address**
- An Arduino is an I2C slave – to participate in communication with the Omega, it listens on the relevant I2C **address**

Consequently:

- The Omega and the Arduino must use the same **address** to communicate
- One Omega can communicate with multiple Arduinos so long as each Arduino uses a different **slave address**
- When using multiple Arduinos connected to the Omega, each Arduino must use a different **slave address** to avoid conflict
- Because the Arduino operates as a **slave** it cannot act as a **master** to access other attached I2C devices – this must be done by the Omega accessing these devices directly

5.3. Functional Connection

The communication between the Omega and the Arduino uses the concept of different **ports** to represent different areas of functionality. For example, one might have a separate **port** for access to each device (e.g. shield) attached to the Arduino.

The communication uses a **port** number to specify which area of functionality to communicate with. For each **port** there can be different sets of **commands** that are used to access specific functions on the **port**

There are two code components involved in Omega – Arduino **port** communication:

- **On the Omega end**, the class **ArduinoPort** (described below) provides object instances that have generic functionality to send commands and data to the **port** on the Arduino and retrieve the responses from the Arduino
- **On the Arduino end**, there must be a specific class that descends from the abstract class **OmegaPort** (see documentation on Arduino library **Arduino_Omega** library) that accepts commands and data from the Omega and makes responses available (according to the desired functionality) for retrieval by the Omega

The normal usage and flow of communication on a **port** is as follows:

- On the Omega, call one of the **ArduinoPort** methods **sendCmd**, **send8**, **send16**, **send32** or **sendBuffer** with command and any required data. The method to use depends on the data expectations of the Arduino for processing of the command.
- The Arduino receives the command and any data, performs any required actions, and composes and makes available a response that includes any return data and a status indicator. This always occurs even if the command is not recognised or has an error in received data or execution.
- On the Omega, immediately after calling one of the methods **sendCmd**, **send8**, **send16**, **send32** or **sendBuffer**, call one of the methods **getStatus**, **get8**, **get16**, **get32** or **getBuffer** to retrieve the Arduinos response including any data and status. The specific retrieval method to use depends on the expected response type from the Arduino.

The provided code supplies one such pair of predefined code for **ArduinoPort** (on the Omega) and **OmegaPort** (on the Arduino) to provide functional access from the Omega to Arduino general I/O functions. These are:

- **On the Omega end**, the class **ArduinoSystem** (see below) provides access to Arduino I/O using an instance of **ArduinoPort**
- **On the Arduino end**, the class **OmegaArduinoSystemPort** (see documentation on Arduino library **Arduino_Omega** library) extends the class **OmegaPort** to accept and respond to commands and data from the port used by **ArduinoSystem**

6. Description of the libarduino Library

The **libarduino** library contains three main components for access and usage of **libarduino** and some components that have internal usage only. These main components and their source files are:

- **ArduinoAccessTypes** – defines a few basic types used elsewhere
  File:
  ArduinoAccessTypes.h
- **ArduinoPort** – a class used to represent instances of generic access to Arduino ports.
  Contains methods to send commands to and retrieve data from an Arduino port.
  Files:
  ArduinoPort.h
  ArduinoPort.cpp
- **ArduinoSystem** – a class used to represent instances of access to Arduino system I/O functionality via an Arduino port.
  Contains methods to perform Arduino I/O actions on the Arduino.
  Files:
  ArduinoSystem.h - includes some defines and types used in the access
  ArduinoSystem.cpp

The contents of these main components are described in following sections.

6.1. I2CTypes

The file **I2CTypes.h** contains definitions of some basic types used elsewhere.

6.1.1. Defines

The following #define items are provided for convenience:

- **#define DEFAULT_ARDUINO_DEV_ADDR 0x08**
  Defines the default I2C device address for the Arduino in the Arduino Dock as used conventionally in existing Onion Omega I2C code

- **#define DEFAULT_ARDUINO_SYSPORT 0**
  Defines the default port number for **ArduinoSystem** if not otherwise supplied

6.1.2. enum Arduino_Result

enum Arduino_Result is used to represent the returned result of Arduino access operations. It has values:

- **ARDUINO_OK = 0** – represents a successful result
- **ARDUINO_UNKNOWN_COMMAND = 1** – indicates that an unrecognised command was used
- **ARDUINO_DATA_ERR = 2** – indicates that the data supplied was in an invalid form
- **ARDUINO_I2C_ERR = 3** – indicates that an I2C error occurred during communication
- **ARDUINO_BAD_PIN = 4** – indicates that an invalid Arduino pin number was used
- **ARDUINO_BAD_PORT = 5** – indicates that a port number was not recognised

- **ARDUINO_SIG_PENDING = 6** – indicates an attempt by the Arduino to signal the Omega when the Omega had not already acted on an earlier signal
- **ARDUINO_BAD_SIG = 7** – indicates that signal data is not in a valid form
- **ARDUINO_BAD_READ_LEN = 8** – indicates that the length of data read was not what was expected
- **ARDUINO_NO_LINK = 9** – indicates that an error occurred in communication
- **ARDUINO_TIMEOUT = 10** – indicates that an I2C timeout occurred during communication

6.1.3. typedef void (*Arduino_Sig_Handler_Func) (unsigned char devAddr, unsigned char portN, I2C_Data &linkData);

Arduino_Sig_Handler_Func represents the type of the function to be specified for handling of an interrupt received when the Arduino signals the Omega. Any such function passed for handling of the interrupt will be called when the signal occurs.

While the parameters passed are not strictly speaking required for signal handling in general, they are provided to allow the same handler to be used for multiple Arduino device addresses and ports. Any actual handler can then (if required) control its action depending upon the device address and port. If no such distinction is required, these parameters can be ignored in the actual implementation of a passed handler.

Parameters:

- **unsigned char devAddr** – the I2C device address for which the handler is being called
- **unsigned char portN** – the port number for which the handler is being called
- **I2C_Data &linkData** – the data supplied by the Arduino when it raised the signal

Returns:

- <none>

6.1.4. Arduino_Sig_Handler_Object

Arduino_Sig_Handler_Object is a pure virtual abstract class that can be used as the base class of an object used to handle a signal as an alternative to using a **Arduino_Sig_Handler_Func**.

The form of the class is:

```
class Arduino_Sig_Handler_Object {
public:
    virtual void handleSignalData(unsigned char devAddr, unsigned char portN,
I2C_Data &linkData) = 0;
};
```

Any object actually used as an instance of **Arduino_Sig_Handler_Object** must inherit from this class and provide a non-abstract method for **handleSignalData**. When an instance of any such class is used to handle a signal, the **handleSignalData** method of the object is called to handle the signal. The **handleSignalData** method has the following characteristics:

Parameters:

- **unsigned char devAddr** – the I2C device address for which the handler is being called
- **unsigned char portN** – the port number for which the handler is being called
- **I2C_Data &linkData** – the data supplied by the Arduino when it raised the signal

Returns:

- <none>

6.1.5. typedef struct I2C_Data

This actually comes from **libnewi2c**. It is used to represents general buffer data in transfers.

```
typedef struct I2C_Data {
    int size;
    unsigned char data[I2C_BUFFER_SIZE];
} I2C_Data;
```

NOTE:

- **size** – this is the number of bytes in the buffer data
- **data** – this is that actual buffer data

6.2. Class ArduinoPort

The **ArduinoPort** class represents instances of an Arduino port access.

6.2.1. ArduinoPort Constructors

6.2.1.1. Constructor - ArduinoPort(unsigned char portN);

Creates a new ArduinoPort instance with given port number on the default Arduino access I2C device address. The default Arduino access I2C device address is 0x08

Parameters:

- **unsigned char portN** – the port number

6.2.1.2. Constructor - ArduinoPort(unsigned char devAddr, unsigned char portN);

Creates a new ArduinoPort instance with given port number on the given Arduino access I2C device address.

Parameters:

- **unsigned char devAddr** – the I2C device address for Arduino
- **unsigned char portN** – the port number

6.2.2. ArduinoPort Public Methods

Note that in general, the success or failure of any method is returned as an **Arduino_Result** value from each method.

6.2.2.1. Arduino_Result sendCmd(unsigned char cmd);
Sends a command to the port with no data.

Parameters:

- **unsigned char cmd** – the command to send

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.2. Arduino_Result send8(unsigned char cmd, unsigned char val);
Sends a command to the port with an 8 bit data value.

Parameters:

- **unsigned char cmd** – the command to send
- **unsigned char val** – the data value to send

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.3. Arduino_Result send16(unsigned char cmd, unsigned int val);
Sends a command to the port with a 16 bit data value.

Parameters:

- **unsigned char cmd** – the command to send
- **unsigned int val** – the data value to send

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.4. Arduino_Result send32(unsigned char cmd, unsigned long val);
Sends a command to the port with a 32 bit data value.

Parameters:

- **unsigned char cmd** – the command to send
- **unsigned long val** – the data value to send

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.5. Arduino_Result sendBuffer(unsigned char cmd, I2C_Data i2cData);
Sends a command to the port with arbitrary buffer data.

Parameters:

- **unsigned char cmd** – the command to send
- **I2C_Data i2cData** – the buffer data to send, includes length of data

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.6. *Arduino_Result getStatus();*

Retrieves a status only response from the port.

Parameters:

- <none>

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.7. *Arduino_Result get8(unsigned char & val);*

Retrieves an 8 bit data response from the port.

Parameters:

- **unsigned char &val** – returns the value retrieved

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.8. *Arduino_Result get16(unsigned int & val);*

Retrieves a 16 bit data response from the port.

Parameters:

- **unsigned int &val** – returns the value retrieved

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.9. *Arduino_Result get32(unsigned long & val);*

Retrieves a 32 bit data response from the port.

Parameters:

- **unsigned long &val** – returns the value retrieved

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.10. Arduino_Result getBuffer(I2C_Data & i2cData, int numBytes);
Retrieves an arbitrary buffer data response from the port.

Parameters:

- **I2C_Data &i2cData** – returns the buffer data including length
- **int numBytes** – the expected number of bytes to b retrieved

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.11. void setResponseDelayMS(unsigned long delayMS);
Because it is possible that actions on the Arduino in executing a command may take some time to perform, it may be necessary to delay retrieving a response. This method sets the delay time to use when retrieving a response.

Parameters:

- **unsigned long delayMS** – the delay in milliseconds to apply

Returns:

- <none>

6.2.2.12. unsigned long getResponseDelayMS();
Returns current response delay.

Parameters:

- <none>

Returns:

- The current response delay in milliseconds

6.2.2.13. void setRetryDelay(unsigned int delayMS);
Sets the retry delay on the device used for the port. When accessing the device for the port, the specified time will be waited before making any retries. The default delay is 1 millisecond.

Parameters:

- **unsigned int delayMS** – the delay time in milliseconds between retries

Returns:

- <none>

6.2.2.14. *void setRetryCount(int count);*

Sets the retry count on the device used for the port. When accessing the device for the port, any failed operation will be retried this number of times before the access is abandoned with a time out. The default retry count is 10.

Parameters:

- **int count** – the number of times to retry a failed access. If this value is less than 0, an unlimited number of retries will be performed.

Returns:

- <none>

6.2.2.15. *unsigned int getRetryDelay();*

Returns the current retry delay on the device for the port.

Parameters:

- <none>

Returns:

- the current retry delay in milliseconds

6.2.2.16. *int getRetryCount();*

Returns the current retry count on the device for the port.

Parameters:

- <none>

Returns:

- the current retry count

6.2.2.17. *Arduino_Result setSignalHandler(int pin, Arduino_Sig_Handler_Func handler);*

Sets the Omega pin that will receive signal interrupts from the Arduino and the function to be called to handle the interrupts. This method and the similar one which takes an **Arduino_Sig_Handler_Object** parameter are mutually exclusive – only one or the other can be in effect at any one time.

Parameters:

- **int pin** – the Omega pin number to be used to catch the signal interrupts
- **Arduino_Sig_Handler_Func handler** – the function to handle the signal interrupts

Returns:

- Result of the call as an **Arduino_Result** value

*6.2.2.18. Arduino_Result setSignalHandler(int pin, Arduino_Sig_Handler_Object * handlerObj);*

Sets the Omega pin that will receive signal interrupts from the Arduino and the object to be used to handle the interrupts. This method and the similar one which takes an **Arduino_Sig_Handler_Func** parameter are mutually exclusive – only one or the other can be in effect at any one time.

Parameters:

- **int pin** – the Omega pin number to be used to catch the signal interrupts
- **Arduino_Sig_Handler_Object handlerObj** – the object that is to handle the signal interrupts via calls to its **handleSignalData** method

Returns:

- Result of the call as an **Arduino_Result** value

6.2.2.19. Arduino_Result reboot();

Reboots the Arduino used for the port as if the reset button had been pressed on the Arduino.

Parameters:

- <none>

Returns:

- Result of the call as an **Arduino_Result** value

6.3. Class ArduinoSystem

The **ArduinoSystem** class represents instances of an Arduino system I/O access that is performed using an **ArduinoPort** instance.

6.3.1. ArduinoSystem Defines and Types

The file ArduinoSystem uses some basic types etc used by the class.

6.3.1.1. Defines

The following #define items are provided for convenience:

- Defines the maximum Arduino pin number that can be used
  - **#define MAX_PIN 19**

- Symbolic defines for Arduino Digital pin numbers
  - **#define D0 0**
  - **#define D1 1**
  - **#define D2 2**
  - **#define D3 3**
  - **#define D4 4**
  - **#define D5 5**

    - **#define D6 6**
    - **#define D7 7**
    - **#define D8 8**
    - **#define D9 9**
    - **#define D10 10**
    - **#define D11 11**
    - **#define D12 12**
    - **#define D13 13**

- Symbolic defines for Arduino Analog pin numbers
    - **#define A0 14**
    - **#define A1 15**
    - **#define A2 16**
    - **#define A3 17**
    - **#define A4 18**
    - **#define A5 19**

- Symbolic defines for Arduino SPI related pins
    - **#define SS D10**
    - **#define MOSI D11**
    - **#define MISO D12**
    - **#define SCK D13**

- Symbolic defines for Arduino I2C related pins
    - **#define SDA A4**
    - **#define SCL A5**

6.3.1.2. enum ArduinoPinMode

enum ArduinoPinMode is used to represent the mode of an Arduino pin. It has values:

- **INPUT = 0x0** – mode is input
- **OUTPUT = 0x1** – mode is output
- **INPUT_PULLUP = 0x2** – mode is input with pull up

6.3.1.3. enum ArduinoPinVal

enum ArduinoPinVal is used to represent the state of an Arduino pin. It has values:

- **LOW = 0x0** – represents a pin in the low (i.e. off) state
- **HIGH = 0x1** – represents a pin in the high (i.e. on) state

6.3.1.4. enum ArduinoARefMode

enum ArduinoARefMode is used to represent the analog reference mode to be used by the Arduino. It has values:

- **DEFAULT = 0** – use the default setting for reference voltage
- **EXTERNAL = 1** – use an external reference voltage

- **INTERNAL = 2** – use the internal reference voltage

6.3.1.5. enum ArduinoBitOrder

enum ArduinoBitOrder is used to represent the bit ordering to be used for shift in and shift out operations on an Arduino pin. It has values:

- **LSBFIRST = 0** – use least significant bit first
- **MSBFIRST = 1** – use most significant bit first

6.3.2. ArduinoSystem Constructors

6.3.2.1. Constructor - ArduinoSystem();

Creates a new ArduinoSystem instance that uses the default Arduino access I2C device address and the default ArduinoPort number. The default Arduino access I2C device address is 0x08. The default port number is 0

Parameters:

- <none>

6.3.2.2. Constructor - ArduinoSystem(unsigned char portN);

Creates a new ArduinoSystem instance that uses the default Arduino access I2C device address and the given ArduinoPort number. The default Arduino access I2C device address is 0x08.

Parameters:

- **unsigned char portN** – the port number

6.3.2.3. Constructor - ArduinoSystem(unsigned char devAddr, unsigned char portN);

Creates a new ArduinoSystem instance that uses the given Arduino access I2C device address and the given ArduinoPort number.

Parameters:

- **unsigned char devAddr** – the I2C device address for Arduino
- **unsigned char portN** – the port number

6.3.3. ArduinoSystem Public Methods

The majority of the methods provide Omega access to the Arduino functions of the same name on the Arduino.

Note that in general, the success or failure of any method is returned as an **Arduino_Result** value from each method.

6.3.3.1. Arduino_Result pinMode(unsigned char pin, ArduinoPinMode mode);

Sets the pin mode for the given Arduino pin.

Parameters:

- **unsigned char pin** – the Arduino pin number
- **ArduinoPinMode mode** – the mode to use

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.2. Arduino_Result digitalWrite(unsigned char pin, ArduinoPinVal val);

Sets an Arduino pin to the given state

Parameters:

- **unsigned char pin** – the Arduino pin number
- **ArduinoPinVal val** – the state to set

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.3. Arduino_Result digitalRead(unsigned char pin, ArduinoPinVal & val);

Gets the state of an Arduino pin

Parameters:

- **unsigned char pin** – the Arduino pin number
- **ArduinoPinVal &val** – returns the state

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.4. Arduino_Result analogReference(ArduinoARefMode mode);

Sets the analog reference mode to be used on the Arduino.

Parameters:

- **ArduinoARefMode mode** – the mode to be used

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.5. Arduino_Result analogRead(unsigned char pin, unsigned int & val);

Reads the analog value from an Arduino analog pin.

Parameters:

- **unsigned char pin** – the Arduino pin number
- **unsigned int &val** – returns the analog value

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.6. Arduino_Result analogWrite(unsigned char pin, unsigned char val);

Writes an analog value to an Arduino pin.

NOTE: The Arduino actually uses PWM output on a digital pin to perform this operation. Not all Arduino pins can perform PWM output – consult documentation on your Arduino.

Parameters:

- **unsigned char pin** – the Arduino pin number
- **unsigned char val** – the value to output, 0 gives a PWM output with 0% duty cycle, 255 gives a PWM output with 100% duty cycle.

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.7. Arduino_Result tone(unsigned char pin, unsigned int freq, unsigned long durationMS = 0);

Outputs a continuous tone on an Arduino pin.

Parameters:

- **unsigned char pin** – the Arduino pin number
- **unsigned int freq** – the frequency of the tone in Hz
- **unsigned long durationMS** – the duration for which the tone sounds in milliseconds. If 0, the duration is indefinite. The tone output continues until the duration expires or the **noTone** function is used on the same pin.

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.8. Arduino_Result noTone(unsigned char pin);

Terminates tone output on an Arduino pin.

Parameters:

- **unsigned char pin** – the Arduino pin number

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.9. Arduino_Result shiftOut(unsigned char dataPin, unsigned char clockPin, ArduinoBitOrder order, unsigned char value);

Performs a serial shift output of a value on a data pin on the Arduino clocked by a clock pin.

Parameters:

- **unsigned char dataPin** – the Arduino pin number for the data
- **unsigned char clockPin** – the Arduino pin number for the clock
- **ArduinoBitOrder order** – the bit order to use for transferring the data on the data pin
- **unsigned char value** – the value to output

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.10. Arduino_Result shiftIn(unsigned char dataPin, unsigned char clockPin, ArduinoBitOrder order, unsigned char & value);

Performs a serial shift input of a value from a data pin on the Arduino clocked by a clock pin.

Parameters:

- **unsigned char dataPin** – the Arduino pin number for the data
- **unsigned char clockPin** – the Arduino pin number for the clock
- **ArduinoBitOrder order** – the bit order to use for transferring the data on the data pin
- **unsigned char &value** – returns the value input

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.11. Arduino_Result pulseIn(unsigned char pin, ArduinoPinVal pulseType, unsigned long & val);

Inputs the duration of a single pulse received on an Arduino pin.

Parameters:

- **unsigned char pin** – the Arduino pin number
- **ArduinoPinVal pulseType** – indicates whether a high level or low level pulse is to be input
- **unsigned long &val** – returns the duration of the pulse detected in micro-seconds.

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.12. Arduino_Result pulseIn(unsigned char pin, ArduinoPinVal pulseType, unsigned long timeOutUS, unsigned long & val);

Inputs the duration of a single pulse received on an Arduino pin with timeout

Parameters:

- **unsigned char pin** – the Arduino pin number
- **ArduinoPinVal pulseType** – indicates whether a high level or low level pulse is to be input
- **unsigned long timeOutUS** – a timeout period in micro-seconds to apply when awaiting the pulse
- **unsigned long &val** – returns the duration of the pulse detected in micro-seconds.

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.13. void setResponseDelayMS(unsigned long delayMS);

Because it is possible that actions on the Arduino in executing a command may take some time to perform, it may be necessary to delay retrieving a response. This method sets the delay time to use when retrieving a response.

Parameters:

- **unsigned long delayMS** – the delay in milliseconds to apply

Returns:

- <none>

6.3.3.14. unsigned long getResponseDelayMS();

Returns current response delay.

Parameters:

- <none>

Returns:

- The current response delay in milliseconds

6.3.3.15. void setRetryDelay(unsigned int delayMS);

Sets the retry delay on the device used for the port. When accessing the device for the port, the specified time will be waited before making any retries. The default delay is 1 millisecond.

Parameters:

- **unsigned int delayMS** – the delay time in milliseconds between retries

Returns:

- <none>

6.3.3.16. void setRetryCount(int count);

Sets the retry count on the device used for the port. When accessing the device for the port, any failed operation will be retried this number of times before the access is abandoned with a time out. The default retry count is 10.

Parameters:

- **int count** – the number of times to retry a failed access. If this value is less than 0, an unlimited number of retries will be performed.

Returns:

- <none>

6.3.3.17. unsigned int getRetryDelay();

Returns the current retry delay on the device for the port.

Parameters:

- <none>

Returns:

- the current retry delay in milliseconds

6.3.3.18. int getRetryCount();

Returns the current retry count on the device for the port.

Parameters:

- <none>

Returns:

- the current retry count

6.3.3.19. Arduino_Result setSignalHandler(int pin, Arduino_Sig_Handler_Func handler);

Sets the Omega pin that will receive signal interrupts from the Arduino and the function to be called to handle the interrupts. This method and the similar one which takes an **Arduino_Sig_Handler_Object** parameter are mutually exclusive – only one or the other can be in effect at any one time.

Parameters:

- **int pin** – the Omega pin number to be used to catch the signal interrupts
- **Arduino_Sig_Handler_Func handler** – the function to handle the signal interrupts

Returns:

- Result of the call as an **Arduino_Result** value

*6.3.3.20. Arduino_Result setSignalHandler(int pin, Arduino_Sig_Handler_Object * handlerObj);*

Sets the Omega pin that will receive signal interrupts from the Arduino and the object to be used to handle the interrupts. This method and the similar one which takes an **Arduino_Sig_Handler_Func** parameter are mutually exclusive – only one or the other can be in effect at any one time.

Parameters:

- **int pin** – the Omega pin number to be used to catch the signal interrupts
- **Arduino_Sig_Handler_Object handlerObj** – the object that is to handle the signal interrupts via calls to its **handleSignalData** method

Returns:

- Result of the call as an **Arduino_Result** value

6.3.3.21. Arduino_Result reboot();

Reboots the Arduino used for the port as if the reset button had been pressed on the Arduino.

Parameters:

- <none>

Returns:

- Result of the call as an **Arduino_Result** value

7. Further Development

Development of **libarduino** is on-going. There will be changes and additions to the code in the future.