Arduino_Omega

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

Arduino_Omega is an Arduino library that runs on an Arduino to accept commands from and supply responses to the Omega connected to the Arduino.

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

Note:

- The **Arduino_Omega** library uses I2C to communicate with the Omega. It requires code in the Omega **libnewi2c** library to be running on the Omega to provide access to the Arduino from theOmega. This code is available and documented at https://github.com/KitBishop/Omega-GPIO-I2C-Arduino/tree/master/libnewi2c

Arduino_Omega consists of Arduino library source code containing the classes used to interact with the Omega and a sample Arduino sketch that uses the library.

The library and the sample sketchare described in more details in this document.

The library and sketch require use of the Arduino IDE.

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

2. Files Supplied

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

- **Arduino_Omega.pdf** – this documentation as a PDF file
- **Makefile** – the Makefile for **Arduino_Omega** library that packages the **Arduino_Omega** library in a **.zip** file.
- **Arduino_Omega** – directory containing the library code and associated files
- **Arduino_Omega/examples/omega_access/omega_access.ino** – an example Arduino sketch
- **zip/Arduino_Omega.zip** – the packaged **.zip** file suitable for installation in the Arduino IDE

3. Usage and Installation

Installing and using the library is simple. It primarily consists of installing the library **.zip** file in the Arduino IDE and using it in an Arduino sketch.

3.1. Installing Arduino_Omega.zip library

To install **Arduino_Omega** library for use in a sketch on your Arduino, you simply need to install the **.zip** file as an Arduino library in your Arduino IDE as follows:

- Start the Arduino IDE
- Navigate to menu item **Sketch->Include Library->Add .ZIP Library …**
- Find the **Arduino_Omega.zip** file and click on **Open**

3.2. Using Arduino_Omega library in a Sketch

An Arduino sketch that uses **Arduino_Omega** library must both include the library in the sketch and call required methods in the **setup** code.

To include the library in a sketch:

- In the Arduino IDE, navigate to menu item **Sketch->Include Library->Arduino Omega**

Code required in the **setup** function:

- You must call an **OmegaAccess::begin** method to configure the access and start the Arduino listening for commands from the Omega
- You must register a port to process commands from the Omega by using the **OmegaAccess::registerPort** method

An example sketch is provided that illustrates this, to access this example:

- In the Arduino IDE, navigate to menu item **File->Examples->Arduino Omega->omega_access**

More details on the required methods and the use of ports is described below.

4. Using Makefile

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

4.1. Makefile targets

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

- **make**
  The default target. Packages the library in a zip file in the **zip** directory

- **make clean**
  Removes the packaged zip file

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

6. Description of the Arduino_Omega Library

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

- **OmegaAccessTypes** – defines a few basic types used elsewhere
  File:
  OmegaAccessTypes.h
- **OmegaAccess** – a class that contains static methods only and implements the communication with the Omega and distributes commands and responses to registered **OmegaPort** instances.
  Files:
  OmegaAccess.h
  OmegaAccess.cpp
- **OmegaPort** – a pure virtual abstract class used to as the base class for instances of generic access ports.
  Contain method abstract method to receive commands from and provide response data to the Omega.
  Files:
  OmegaPort.h
  OmegaPort.cpp
- **OmegaArduinoSystemPort** – a class that extends **OmegaPort** to provide access to the Omega to commands sent from the Omega class **ArduinoSystem** for access to Arduino I/O from the Omega.
  Files:
  OmegaArduinoSystemPort.h
  OmegaArduinoSystemPort.cpp
- **omega-access** – a sample Arduino sketch that provides default access to Arduino I/O
  File:
  omega-access.ino

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

6.1. OmegaAccessTypes

The file **OmegaAccessTypes.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 **OmegaArduinoSystemPort** 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

Note that these are the same values as are used for **Arduino_Result** in **ArduinoAccessTypes** in **libarduino** on the Omega.

6.1.3. typedef struct LinkData

Used to represents general data received from the Omega.

```
typedef struct LinkData {
    unsigned char data[32];
    int size;
} LinkData;
```

NOTES:

1. Structure fields:
   - **data** – this is that actual buffer data
   - **size** – this is the number of bytes in the buffer data
2. When used to represent an 8 bit value the fields are used so:
   - size is 1
   - unsigned char value = data[0];
3. When used to represent a 16 bit value the fields are used so:
   - size is 2
   - unsigned int value = data[0] + (data[1] << 8);

   I.E. least significant byte first
4. When used to represent a 32 bit value the fields are used so:
   - size is 4
   - unsigned int value = data[0] + (data[1] << 8) + (data[2] << 16) + (data[3] << 24);

   I.E. least significant byte first

6.1.4. typedef struct ResponseData

Used to represents general data response to be sent to the Omega.

```
typedef struct ResponseData {
    Arduino_Result status;
    unsigned char data[32];
    int size;
} ResponseData;
```

NOTES:

1. Structure fields:
   - **status** – the status of the response
   - **data** – this is that actual buffer data
   - **size** – this is the number of bytes in the buffer data
2. When used to represent an error response the fields are used so:
   - status is an **Arduino_Result** value other than ARDUINO_OK
   - size is 0
3. When used to represent a atatus only response with no associated data the fields are used so:
   - status is the **Arduino_Result** value ARDUINO_OK for success, any other value for error
   - size is 0
4. When used to represent an 8 bit response value the fields are used so:
   - status is the **Arduino_Result** value ARDUINO_OK
   - size is 1
   - data[0] = value;
5. When used to represent a 16 bit response value the fields are used so:
   - status is the **Arduino_Result** value ARDUINO_OK
   - size is 2
   - data[0] = value & 0xff;
   - data[1] = (value >> 8) & 0xff;

   I.E. least significant byte first
6. When used to represent a 32 bit response value the fields are used so:
   - status is the **Arduino_Result** value ARDUINO_OK
   - size is 4
   - data[0] = value & 0xff;
   - data[1] = (value >> 8) & 0xff;
   - data[2] = (value >> 16) & 0xff;
   - data[3] = (value >> 24) & 0xff;

   I.E. least significant byte first

6.2. Class OmegaAccess

The **OmegaAccess** class is used to handle communications received from the Omega and distribute the communication to the relevant registered ports.

Notes:

1. One and only one call may be made to an **OmegaAccess::begin** method to configure and initialise processing of communication from the Omega
2. At least one call should be made to the **OmegaAccess::registerPort** method with the port ta handle specific communications from a port on the Omega

6.2.1. OmegaAccess Public Methods

Note that all methods are static methods only.

6.2.1.1. static void begin(unsigned char i2cAddr, int sigPin);

Configures and start the access with the given information.

Parameters:

- **unsigned char i2cAddr** – the I2C address on which communications are to be listened for
- **int sigPin** – the Arduino pin to be used for signalling to the Omega. This pin must be connected to the pin in the Omega that is used to receive signals from the Arduino.
  If this has a value <0, no signalling can be performed to the Omega.

Returns:

- <none>

6.2.1.2. static void begin(unsigned char i2cAddr);

Configures and start the access with the given information but no signal pin.

Note:

- This is equivalent to **OmegaAccess::begin(i2cAddr, -1)**

Parameters:

- **unsigned char i2cAddr** – the I2C address on which communications are to be listened for

Returns:

- <none>

6.2.1.3. static void begin();

Configures and start the access using the default I2C address (0x08) and no signal pin.

Note:

- This is equivalent to **OmegaAccess::begin(DEFAULT_ARDUINO_DEV_ADDR, -1)**

Parameters:

- <none>

Returns:

- <none>

*6.2.1.4. static Arduino_Result registerPort(OmegaPort * omegaPort);*

Registers a port to receive communications. Any communications from the Omega with a port number that matches that of the given port will be dispatched to the **processCommand** method of the port.

Parameters:

- **OmegaPort * omegaPort** – a port to receive commands from the Omega

Returns:

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

6.3. Class OmegaPort

The **OmegaPort** class is a pure virtual abstract class that is to be used as the super class of any class to be used to receive commands from the Omega.

As such, it is essential that any descendent class provide an implementation of the **processCommand** method.

6.3.1. OmegaPort Constructors

6.3.1.1. Constructor - OmegaPort(unsigned char portN);

Creates a new OmegaPort instance with the given port number.

Parameters:

- **unsigned char portN** – the port number. Must be in the range 0 to 15

6.3.2. OmegaPort Public Methods

6.3.2.1. virtual ResponseData processCommand(unsigned char cmd, LinkData linkData) = 0;

The virtual method used to process commands from the Omega. Must be overridden and implemented in any descendent class.

Parameters:

- **unsigned char cmd** – the command number received from the Omega
- **LinkData linkData** – the associated data received with the command from the Omega. If there is no such data, the **size** field of linkdata will be 0

Returns:

- The response data to be returned to the Omega as a result of processing the command.

6.3.2.2. Arduino_Result signalOmega(LinkData sigData);

Signals the Omega and makes the supplied data available to the Omega.

Note: This method is dependent upon the following factors:

- **OmegaAccess** must have been configured with a valid **sigPin**
- The **sigPin** on the Arduino must be physically connected to a GPIO pin on the Omega
- The GPIO pin on the Omega must be specified in a **setSignalHandler** method call on an **ArduinoPort** on the Omega

Parameters:

- **LinkData sigData** – the data to be communicated to the linked **ArduinoPort** on the Omega

Returns:

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

6.4. Class OmegaArduinoSystemPort

The **OmegaArduinoSystemPort** class descends from the **OmegaPort** class and provides a **processCommand** implementation that handles commands from an Omega **ArduinoSystem** instance to provide access to Arduino I/O functionality.

Note that this class inherits the **signalOmega** method from **OmegaPort** class.

6.4.1. OmegaArduinoSystemPort Constructors

6.4.1.1. Constructor - OmegaArduinoSystemPort (unsigned char portN);

Creates a new **OmegaArduinoSystemPort** instance with the given port number.

Parameters:

- **unsigned char portN** – the port number. Must be in the range 0 to 15

6.4.1.2. Constructor - OmegaArduinoSystemPort ();

Creates a new **OmegaArduinoSystemPort** instance with default port number.

Note:

- This is equivalent to **OmegaArduinoSystemPort(DEFAULT_ARDUINO_SYSPORT)**

Parameters:

- <none>

6.5. Sketch omega-access

The file **omega-access.ino** contains a basic Arduino Sketch that provides access to the Arduino I/O via a **OmegaArduinoSystemPort** instance when used with a default **ArduinoSystem** instance on the Omega.

The relevant code in the setup method is:

```
// Set up Omega access for default access with I2C address 0x08
// using access to Arduino System pin functionality on default port 0
//
    OmegaAccess::begin();
    OmegaAccess::registerPort(new OmegaArduinoSystemPort());
```

7. Further Development

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