libnewgpio

Version 1.4.1 – 16 June 2016
Kit Bishop

Document History

Version	Date	Change Details
Version 1.4.1	16 June 2016	First full version derived from earlier libnew-gpio code

Contents

1. Background ___ ***5***

2. Pre-requisites ___ ***5***

3. Files Supplied ___ ***5***

4. Usage and Installation ___ ***6***

4.1. Using libnewgpio.a static library ___ **6**

4.2. Using and Installing libnewgpio.so dynamic library ___ **6**

5. Using Makefile ___ ***6***

5.1. Modify Makefile ___ **6**

5.2. Makefile targets ___ **7**

6. Description of the libnewgpio Library ___ ***8***

6.1. GPIOTypes ___ **8**

6.1.1. enum GPIO_Result ___ 8
6.1.2. enum GPIO_Direction ___ 9
6.1.3. enum GPIO_Bit_Order ___ 9
6.1.4. enum GPIO_Irq_Type ___ 9
6.1.5. typedef void (*GPIO_Irq_Handler_Func) (int pinNum, GPIO_Irq_Type type); ___ 9
6.1.6. GPIO_Irq_Handler_Object ___ 10
6.1.7. typedef void (*GPIO_PulseIn_Handler_Func) (int pinNum, long int len); ___ 10
6.1.8. GPIO_PulseIn_Handler_Object ___ 11

6.2. Class GPIOAccess ___ **12**

6.2.1. GPIOAccess Public Methods ___ 12
6.2.1.1. static GPIO_Result setDirection(int pinNum, GPIO_Direction dir); ___ 12
6.2.1.2. static GPIO_Result getDirection(int pinNum, GPIO_Direction &dir); ___ 12
6.2.1.3. static GPIO_Result set(int pinNum, int value); ___ 12
6.2.1.4. static GPIO_Result get(int pinNum, int &value); ___ 12
6.2.1.5. static GPIO_Result setPWM(int pinNum, int freq, int duty, int durationMs = 0); ___ 13
6.2.1.6. static GPIO_Result startPWM(int pinNum, int durationMs = 0); ___ 13
6.2.1.7. static GPIO_Result stopPWM(int pinNum); ___ 14

6.2.1.8. static GPIO_Result getPWMFreq(int pinNum, int &freq); 14
6.2.1.9. static GPIO_Result getPWMDuty(int pinNum, int &duty); 14
6.2.1.10. static GPIO_Result getPWMDuration(int pinNum, int &duration); 14
6.2.1.11. static GPIO_Result isPWMRunning(int pinNum, bool &running); 14
6.2.1.12. static GPIO_Result setIrq(int pinNum, GPIO_Irq_Type type, GPIO_Irq_Handler_Func handler, long int debounceMs = 0); 15
6.2.1.13. static GPIO_Result setIrq(int pinNum, GPIO_Irq_Type type, GPIO_Irq_Handler_Object * handlerObj, long int debounceMs = 0); 15
6.2.1.14. static GPIO_Result resetIrq(int pinNum); 16
6.2.1.15. static GPIO_Result enableIrq(int pinNum); 16
6.2.1.16. static GPIO_Result disableIrq(int pinNum); 16
6.2.1.17. static GPIO_Result enableIrq(int pinNum, bool enable); 17
6.2.1.18. static GPIO_Result irqEnabled(int pinNum, bool &enabled); 17
6.2.1.19. static GPIO_Irq_Type getIrqType(int pinNum); 17
6.2.1.20. static GPIO_Irq_Handler_Func getIrqHandler(int pinNum); 17
6.2.1.21. static GPIO_Irq_Handler_Object * getIrqHandlerObj(int pinNum); 18
6.2.1.22. static void enableIrq(); 18
6.2.1.23. static void disableIrq(); 18
6.2.1.24. static void enableIrq(bool enable); 18
6.2.1.25. static bool irqEnabled(); 18
6.2.1.26. static GPIO_Result setTone(int pinNum, int freq, int durationMs = 0); 19
6.2.1.27. static GPIO_Result startTone(int pinNum, int durationMs = 0); 19
6.2.1.28. static GPIO_Result stopTone(int pinNum); 19
6.2.1.29. static GPIO_Result getToneFreq(int pinNum, int &freq); 20
6.2.1.30. static GPIO_Result getToneDuration(int pinNum, int &duration); 20
6.2.1.31. static GPIO_Result isToneRunning(int pinNum, bool &running); 20
6.2.1.32. static GPIO_Result shiftOut(int dataPinNum, int clockPinNum, int val, long int clockPeriodNS, GPIO_Bit_Order bitOrder = GPIO_MSB_FIRST); 20
6.2.1.33. static GPIO_Result shiftIn(int dataPinNum, int clockPinNum, long int clockPeriodNS, GPIO_Bit_Order bitOrder, int &value); 21
6.2.1.34. static GPIO_Result pulseOut(int pinNum, long int pulseLenUS, int pulseLevel = 1); 21
6.2.1.35. static GPIO_Result pulseIn(int pinNum, int pulseLevel, long int timeoutUS, long int &value); 21
6.2.1.36. static GPIO_Result pulseIn(int pinNum, GPIO_PulseIn_Handler_Func handler , int pulseLevel = 1, long int timeoutUS = 0); 22
6.2.1.37. static GPIO_Result pulseIn(int pinNum, GPIO_PulseIn_Handler_Object * handlerObj , int pulseLevel = 1, long int timeoutUS = 0); 22
6.2.1.38. static GPIO_Result stopPulseIn(int pinNum); 23
6.2.1.39. static GPIO_Result isPulseInRunning(int pinNum, bool &running); 23
6.2.1.40. static GPIO_Result getFrequency(int pinNum, long int sampleTimeMS, long int &freq); 23
6.2.1.41. static bool isPinUsable(int pinNum); 24
6.2.1.42. static bool isAccessOk(); 24

6.3. Class GPIOPin 24

6.3.1. GPIOPin Constructor and Destructor 24
6.3.1.1. Constructor - GPIOPin(int pinNum); 24
6.3.1.2. Destructor - ~GPIOPin(void); 25
6.3.2. GPIOPin Public Methods 25
6.3.2.1. GPIO_Result setDirection(GPIO_Direction dir); 25
6.3.2.2. GPIO_Result getDirection(GPIO_Direction &dir); 25

6.3.2.3. GPIO_Result set(int value); 25
6.3.2.4. GPIO_Result get(int &value); 25
6.3.2.5. GPIO_Result setPWM(int freq, int duty, int durationMs = 0); 26
6.3.2.6. GPIO_Result startPWM(int durationMs = 0); 26
6.3.2.7. GPIO_Result stopPWM(); 26
6.3.2.8. GPIO_Result getPWMFreq(int &freq); 27
6.3.2.9. GPIO_Result getPWMDuty(int &duty); 27
6.3.2.10. GPIO_Result getPWMDuration(int &duration); 27
6.3.2.11. GPIO_Result isPWMRunning(bool &running); 27
6.3.2.12. GPIO_Result setIrq(GPIO_Irq_Type type,GPIO_Irq_Handler_Func handler,long int debounceMs=0); 28
6.3.2.13. GPIO_Result setIrq(GPIO_Irq_Type type, GPIO_Irq_Handler_Object * handlerObj, long int debounceMs = 0); 28
6.3.2.14. GPIO_Result resetIrq(); 29
6.3.2.15. GPIO_Result enableIrq(); 29
6.3.2.16. GPIO_Result disableIrq(); 29
6.3.2.17. GPIO_Result enableIrq(bool enable); 29
6.3.2.18. GPIO_Result irqEnabled(bool &enabled); 30
6.3.2.19. GPIO_Irq_Type getIrqType(); 30
6.3.2.20. GPIO_Irq_Handler_Func getIrqHandler(); 30
6.3.2.21. GPIO_Irq_Handler_Object * getIrqHandlerObj(); 30
6.3.2.22. GPIO_Result setTone(int freq, int durationMs = 0); 30
6.3.2.23. GPIO_Result startTone(int durationMs = 0); 31
6.3.2.24. GPIO_Result stopTone(); 31
6.3.2.25. GPIO_Result getToneFreq(int &freq); 31
6.3.2.26. GPIO_Result getToneDuration(int &duration); 32
6.3.2.27. GPIO_Result isToneRunning(bool &running); 32
6.3.2.28. GPIO_Result pulseOut(long int pulseLenUS, int pulseLevel = 1); 32
6.3.2.29. GPIO_Result int pulseIn(int pulseLevel, long int timeoutUS, long int &value); 32
6.3.2.30. GPIO_Result pulseIn(GPIO_PulseIn_Handler_Func handler , int pulseLevel = 1, long int timeoutUS = 0); 33
6.3.2.31. GPIO_Result pulseIn(GPIO_PulseIn_Handler_Object * handlerObj , int pulseLevel = 1, long int timeoutUS = 0); 33
6.3.2.32. GPIO_Result stopPulseIn(); 34
6.3.2.33. GPIO_Result isPulseInRunning(bool &running); 34
6.3.2.34. GPIO_Result int getFrequency(long int sampleTimeMS, long int &freq); 34
6.3.2.35. int getPinNumber(); 35

6.4. Class GPIOShiftIn 36

6.4.1. GPIOShiftIn Constructor and Destructor 36
6.4.1.1. Constructor - GPIOShiftIn(int dataPin, int clockPin); 36
6.4.1.2. Destructor - ~GPIOShiftIn(void); 36
6.4.2. GPIOShiftIn Public Methods 36
6.4.2.1. GPIO_Result read(int &value); 36
6.4.2.2. void setClockPeriodNS(long int clockPerNS); 36
6.4.2.3. long int getClockPeriodNS(); 36
6.4.2.4. void setBitOrder(GPIO_Bit_Order bitOrd); 37
6.4.2.5. GPIO_Bit_Order getBitOrder(); 37
6.4.2.6. GPIOPin * getDataPin(); 37
6.4.2.7. GPIOPin * getClockPin(); 37

6.5. Class GPIOShiftOut 38
6.5.1. GPIOShiftOut Constructor and Destructor 38
6.5.1.1. Constructor - GPIOShiftOut(int dataPin, int clockPin); 38
6.5.1.2. Destructor - ~GPIOShiftOut(void); 38
6.5.2. GPIOShiftOut Public Methods 38
6.5.2.1. GPIO_Result write(int val); 38
6.5.2.2. void setClockPeriodNS(long int clockPerNS); 38
6.5.2.3. long int getClockPeriodNS(); 38
6.5.2.4. void setBitOrder(GPIO_Bit_Order bitOrd); 39
6.5.2.5. GPIO_Bit_Order getBitOrder(); 39
6.5.2.6. GPIOPin * getDataPin(); 39
6.5.2.7. GPIOPin * getClockPin(); 39

6.6. Class RGBLED 40
6.6.1. RGBLED Constructors and Destructor 40
6.6.1.1. Constructor - RGBLED(); 40
6.6.1.2. Constructor - RGBLED(int redPin, int greenPin, int bluePin); 40
6.6.1.3. Destructor - ~RGBLED(void); 40
6.6.2. RGBLED Public Methods 40
6.6.2.1. <void> setColor(int redVal, int greenVal, int blueVal); 40
6.6.2.2. <void> setRed(int redVal); 41
6.6.2.3. <void> setGreen(int greenVal); 41
6.6.2.4. <void> setBlue(int blueVal); 41
6.6.2.5. int getRed(); 41
6.6.2.6. int getGreen(); 41
6.6.2.7. int getBlue(); 42
6.6.2.8. GPIOPin * getRedPin(); 42
6.6.2.9. GPIOPin * getGreenPin(); 42
6.6.2.10. GPIOPin * getBluePin(); 42
6.6.2.11. void setActiveLow(bool actLow); 42
6.6.2.12. bool isActiveLow(); 43

7. Further Development 43

1. Background

libnewgpio is alternative C++ code for accessing the Omega GPIO pins.

The rationale for producing this code was two-fold:

- A desire for GPIO access with different features and capability than **fast-gpio**
- An exercise in developing C++ code for the Omega

libnewgpio consists of static and dynamic link libraries containing the classes used to interact with GPIO pins

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

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

NOTE: Some of the code in the class **GPIOAccess** as described below was derived from code to be found in the **fast-gpio** for the Omega.

2. Pre-requisites

To use the GPIO interrupt handling facilities in **libnewgpio**, your Omega **must** fulfil the following pre-requisites:

- Must have been upgraded to version **0.0.6-b265** or later
- Must have the **kmod-gpio-irq** package installed by running:

```
opkg update
opkg install kmod-gpio-irq
```

3. Files Supplied

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

- **libnewgpio.pdf** – this documentation as a PDF file
- **Makefile** – the Makefile for **libnewgpio** library

- **hdr** – directory containing header (***.h**) files for **libnewgpio** library
- **src** – directory containing source (***.cpp**) files for **libnewgpio** library
- **bin** – directory containing the built library code:
    - **dynamic/libnewgpio.so** – the dynamic link version of the library
    - **static/libnewgpio.a** – the static link version of the library

4. 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.

4.1. Using libnewgpio.a static library

To use **libnewgpio.a** static library you simply need to statically link your program to that library file.

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

4.2. Using and Installing libnewgpio.so dynamic library

To use **libnewgpio.so** dynamic library you need to dynamically link your program to that library file.

For any program that uses **libnewgpio.so** the library file needs to be copied to the **/lib** directory on your Omega.

Alternatively, you can copy the library 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 library to **/root/lib**
- Added the following lines to my **/etc/profile** file:

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

5. Using Makefile

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

5.1. 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

5.2. 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

6. Description of the libnewgpio Library

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

- **GPIOTypes** – defines a few basic types used elsewhere
  File:
  - **GPIOTypes.h**
- **GPIOAccess** – a class used for direct access to the Omega GPIO hardware.
  Contains only **static** methods for access.
  Files:
  - **GPIOAccess.h**
  - **GPIOAccess.cpp**
- **GPIOPin** – a class used to represent instances of a GPIO pin.
  Contains methods to interact with the specific pin.
  Files:
  - **GPIOPin.h**
  - **GPIOPin.cpp**
- **GPIOShiftIn** – a class used to represent instances of a GPIO shift input on 2 pins.
  Contains methods to perform shift input.
  Files:
  - **GPIOShiftIn.h**
  - **GPIOShiftIn.cpp**
- **GPIOShiftOut** – a class used to represent instances of a GPIO shift output on 2 pins.
  Contains methods to perform shift output.
  Files:
  - **GPIOShiftOut.h**
  - **GPIOShiftOut.cpp**
- **RGBLED** – a class used to represent instances of an RGB led (such as the led on the expansion dock).
  Contains methods to interact with the RGB led.
  Files:
  - **RGBLED.h**
  - **RGBLED.cpp**

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

6.1. GPIOTypes

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

6.1.1. enum GPIO_Result

enum GPIO_Result is used to represent the returned result of GPIO operations. It has values:

- **GPIO_OK = 0** – represents a successful result
- **GPIO_BAD_ACCESS = 1** – indicates a failure to access the GPIO hardware registers

- **GPIO_INVALID_PIN = 2** – indicates that a pin number has been used that is not accessible by GPIO
- **GPIO_INVALID_OP = 3** – indicates that an invalid operation has been attempted on a pin. E.G. attempting to set a pin that is in input mode, or reading a pin that is in output mode

6.1.2. enum GPIO_Direction

enum GPIO_Direction is used to represent the direction for a GPIO pin. It has values:

- **GPIO_INPUT = false** – represents an input pin
- **GPIO_OUTPUT = true** – represents an output pin

6.1.3. enum GPIO_Bit_Order

enum GPIO_Bit_Order is used to represent the bit ordering to be used for shift in and shift out functions. It has values:

- **GPIO_MSB_FIRST = 0** – the most significant bit is first
- **GPIO_LSB_FIRST = 1** – the least significant bit is first

6.1.4. enum GPIO_Irq_Type

enum GPIO_Irq_Type is used to represent the type of interrupt used for a GPIO pin. It has values:

- **GPIO__IRQ_NONE = 0** – indicates no interrupt
- **GPIO_IRQ_RISING = 1** – indicates an interrupt on the rising edge (i.e. low to high change) on a pin
- **GPIO_IRQ_FALLING = 2** – indicates an interrupt on the falling edge (i.e. high to low change) on a pin
- **GPIO_IRQ_BOTH = 3** – indicates an interrupt on the either of a rising edge or on a falling edge on a pin

6.1.5. typedef void (*GPIO_Irq_Handler_Func) (int pinNum, GPIO_Irq_Type type);

GPIO_Irq_Handler_Func represents the type of the function to be specified for handling of an interrupt. Any such function passed for handling of an interrupt will be called when the interrupt occurs.

While the parameters passed are not strictly speaking required for interrupt handling in general, they are provided to allow the same handler to be used for multiple pins and interrupt types. Any actual handler can then (if required) control its action depending upon the pin and interrupt type. If no such distinction is required, these parameters can be ignored in the actual implementation of a passed handler.

Parameters:

- **int pinNum** – the number of the pin for which the handler is being called
- **GPIO_Irq_Type type** – the type of interrupt for which the handler is being called

Returns:

- <none>

6.1.6. GPIO_Irq_Handler_Object

GPIO_Irq_Handler_Object is a pure virtual abstract class that can be used as the base class of an object used to handle an interrupt as an alternative to using a **GPIO_Irq_Handler_Func**.

The form of the class is:

```
class GPIO_Irq_Handler_Object {
public:
    virtual void handleIrq(int pinNum, GPIO_Irq_Type type) = 0;
};
```

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

Parameters:

- **int pinNum** – the number of the pin for which the handler is being called
- **GPIO_Irq_Type type** – the type of interrupt for which the handler is being called

Returns:

- <none>

6.1.7. typedef void (*GPIO_PulseIn_Handler_Func) (int pinNum, long int len);

GPIO_PulseIn_Handler_Func represents the type of the function to be specified for handling of completion of a pulse in operation. Any such function passed for handling of a pulse in will be called when the pulse in completes.

While the pinNum parameter passed is not strictly speaking required for pulse in handling in general, it is provided to allow the same handler to be used for multiple pins and pulse in operations. Any actual handler can then (if required) control its action depending upon the pin. If no such distinction is required, this parameter can be ignored in the actual implementation of a passed handler.

Parameters:

- **int pinNum** – the number of the pin for which the handler is being called
- **long int len** – the length of the pulse input in microseconds

Returns:

- <none>

6.1.8. GPIO_PulseIn_Handler_Object

GPIO_PulseIn_Handler_Object is a pure virtual abstract class that can be used as the base class of an object used to handle completion of a pulse in operation as an alternative to using a **GPIO_PulseIn_Handler_Func**.

The form of the class is:

```
class GPIO_PulseIn_Handler_Object {
public:
    virtual void handlePulseIn(int pinNum, long int len) = 0;
};
```

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

Parameters:

- **int pinNum** – the number of the pin for which the handler is being called
- **long int len** – the length of the pulse input in microseconds

Returns:

<none>

6.2. Class GPIOAccess

The **GPIOAccess** class is the main method by which all access is made to the GPIO hardware.

The class contains only static methods and no instance of this class will ever actually be created hence there are no constructors or destructors.

6.2.1. GPIOAccess Public Methods

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

6.2.1.1. static GPIO_Result setDirection(int pinNum, GPIO_Direction dir);

Sets the direction for a pin.

Parameters:

- **int pinNum** – the number of the pin
- **GPIO_Direction dir** – the direction to set the pin to

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.2. static GPIO_Result getDirection(int pinNum, GPIO_Direction &dir);

Queries the direction of a pin.

Parameters:

- **int pinNum** – the number of the pin
- **GPIO_Direction &dir** – returns the direction of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.3. static GPIO_Result set(int pinNum, int value);

Sets the output state of a pin. Only valid for output pins.

Parameters:

- **int pinNum** – the number of the pin
- **int value** – the value to set the pin to

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.4. static GPIO_Result get(int pinNum, int &value);

Queries the input state of a pin. Only valid for input pins.

Parameters:

- **int pinNum** – the number of the pin
- **int &value** – returns the current state of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.5. *static GPIO_Result setPWM(int pinNum, int freq, int duty, int durationMs = 0);*

Starts the PWM output on a pin with the given frequency and duty values.

NOTE: PWM output is performed by software toggling of the state of the pin. Consequently, the frequency and duty cycle of the output is only reasonably accurate up to a few hundred Hz.

NOTE: PWM output on a pin is run on a separate thread for that pin. When this method is called the thread will be started (or its data updated if it is already running) and the call to the method then returns. The thread continues to run until one of the following occurs:

- the **stopPWM** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates
- the duration expires

Parameters:

- **int pinNum** – the number of the pin
- **int freq** – sets the PWM frequency in Hz
- **int duty** – sets the PWM duty cycle percentage
- **int durationMs** – sets the duration in milliseconds for which the PWM output is performed. A value of zero give unterminated output. The default value is zero

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.6. *static GPIO_Result startPWM(int pinNum, int durationMs = 0);*

Starts the PWM output on a pin using the last used frequency and duty values for the pin.

Parameters:

- **int pinNum** – the number of the pin
- **int durationMs** – sets the duration in milliseconds for which the PWM output is performed. A value of zero give unterminated output. The default value is zero

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.7. *static GPIO_Result stopPWM(int pinNum);*
Stops any current PWM output on a pin.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.8. *static GPIO_Result getPWMFreq(int pinNum, int &freq);*
Returns the currently set PWM frequency for a pin.

Parameters:

- **int pinNum** – the number of the pin
- **int &freq** – returns the PWM frequency in Hz

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.9. *static GPIO_Result getPWMDuty(int pinNum, int &duty);*
Returns the currently set PWM duty cycle percentage for a pin

Parameters:

- **int pinNum** – the number of the pin
- **int &duty** – returns the PWM duty cycle percentage

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.10. *static GPIO_Result getPWMDuration(int pinNum, int &duration);*
Returns the currently set PWM duration for a pin

Parameters:

- **int pinNum** – the number of the pin
- **int &duration** – returns the PWM duration

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.11. *static GPIO_Result isPWMRunning(int pinNum, bool &running);*
Returns an indication of whether or not PWM is currently running on a pin

Parameters:

- **int pinNum** – the number of the pin
- **bool &running** – returns **true** if PWM is running; **false** if PWM is not running

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.12. *static GPIO_Result setIrq(int pinNum, GPIO_Irq_Type type, GPIO_Irq_Handler_Func handler, long int debounceMs = 0);*

Setups up interrupt handling for a pin with a given handler function.

NOTE: IRQ handling on a pin is run on a separate thread for that pin. When this method is called the thread will be started and the call to the method then returns. The thread continues to run and to call the handler function whenever the relevant interrupt occurs until one of the following occurs:

- the **resetIrq** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates

Parameters:

- **int pinNum** – the number of the pin
- **GPIO_Irq_Type type** – specifies the interrupt type to apply
- **GPIO_Irq_Handler_Func handler** – a pointer to the function to be called to handle the interrupt
- **Long int debounceMs = 0** – specifies an optional debounce period in milliseconds to be applied.
  Default value is **0**
  If value is **0** no debounce handling is applied
  Debounce handling is used to deal with interrupts that come from a potentially noisy mechanical source such as buttons or switches.
  When a non-zero value is used for **debounceMs**, any input signal changes that would normally cause an interrupt but which occur within a time less than the **debounceMs** time since the previous signal change will be ignored so as not to trigger handling of a false signal.

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.13. *static GPIO_Result setIrq(int pinNum, GPIO_Irq_Type type, GPIO_Irq_Handler_Object * handlerObj, long int debounceMs = 0);*

Setups up interrupt handling for a pin with a given handler object.

NOTE: IRQ handling on a pin is run on a separate thread for that pin. When this method is called the thread will be started and the call to the method then returns. The thread continues to run and to call the handler method of the object whenever the relevant interrupt occurs until one of the following occurs:

- the **resetIrq** method is called for the pin

- the process that started the thread (i.e. made the call to this method) terminates

Parameters:

- **int pinNum** – the number of the pin
- **GPIO_Irq_Type type** – specifies the interrupt type to apply
- **GPIO_Irq_Handler_Object handlerObj** – a pointer to the handler object to be used to handle the interrupt
- **long int debounceMs = 0** – specifies an optional debounce period in milliseconds to be applied.
  Default value is **0**
  If value is **0** no debounce handling is applied
  Debounce handling is used to deal with interrupts that come from a potentially noisy mechanical source such as buttons or switches.
  When a non-zero value is used for **debounceMs**, any input signal changes that would normally cause an interrupt but which occur within a time less than the **debounceMs** time since the previous signal change will be ignored so as not to trigger handling of a false signal.

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.14. static GPIO_Result resetIrq(int pinNum);

Removes any interrupt handling for a pin.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.15. static GPIO_Result enableIrq(int pinNum);

Enables interrupt handling for a pin that has previously been disabled by **disableIrq**.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.16. static GPIO_Result disableIrq(int pinNum);

Disables interrupt handling for a pin that has previously been enabled by **enableIrq**.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.17. *static GPIO_Result enableIrq(int pinNum, bool enable);*

Enables or Disables interrupt handling for a pin according to parameter.

Parameters:

- **int pinNum** – the number of the pin
- **bool enable** – indicates whether interrupt handling is to be enabled (**true**) or disabled (**false**)

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.18. *static GPIO_Result irqEnabled(int pinNum, bool &enabled);*

Returns an indication as to whether interrupt handling is currently enabled or disabled for a pin.

Parameters:

- **int pinNum** – the number of the pin
- **bool &enabled** – returns **true** if interrupt handling is enabled; **false** if interrupt handling is disabled

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.19. *static GPIO_Irq_Type getIrqType(int pinNum);*

Returns the current interrupt type for a pin.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- the interrupt type

6.2.1.20. *static GPIO_Irq_Handler_Func getIrqHandler(int pinNum);*

Returns the any currently established interrupt handler function for a pin.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- the interrupt handler function

6.2.1.21. *static GPIO_Irq_Handler_Object * getIrqHandlerObj(int pinNum);*

Returns the any currently established interrupt handler object for a pin.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- pointer to the interrupt handler object

6.2.1.22. *static void enableIrq();*

Enables interrupt handling for all pins with interrupt handling set up.

Parameters:

- <none>

Returns:

- <none>

6.2.1.23. *static void disableIrq();*

Disables interrupt handling for all pins with interrupt handling set up.

Parameters:

- <none>

Returns:

- <none>

6.2.1.24. *static void enableIrq(bool enable);*

Enables or disables interrupt handling for all pins with interrupt handling set up according to parameter.

Parameters:

- **bool enable** – indicates whether interrupt handling is to be enabled (**true**) or disabled (**false**)

Returns:

- <none>

6.2.1.25. *static bool irqEnabled();*

Returns an indication as to whether interrupt handling is currently enabled or disabled for all pins.

Parameters:

- <none>

Returns:

- **true** – interrupt handling is enabled, **false** – interrupt handling is disabled

6.2.1.26. static GPIO_Result setTone(int pinNum, int freq, int durationMs = 0);

Starts the tone output on a pin with the given frequency. Tone output is equivalent to PWM output with a 50% duty cycle.

NOTE: Tone output is performed by software toggling of the state of the pin. Consequently, the frequency of the output is only reasonably accurate up to a few hundred Hz.

NOTE: Tone output on a pin is run on a separate thread for that pin. When this method is called the thread will be started (or its data updated if it is already running) and the call to the method then returns. The thread continues to run until one of the following occurs:

- the **stopTone** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates
- the duration expires

Parameters:

- **int pinNum** – the number of the pin
- **int freq** – sets the Tone frequency in Hz
- **int durationMs** – sets the duration in milliseconds for which the Tone output is performed. A value of zero give unterminated output. The default value is zero

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.27. static GPIO_Result startTone(int pinNum, int durationMs = 0);

Starts the Tone output on a pin using the last used frequency value for the pin.

Parameters:

- **int pinNum** – the number of the pin
- **int durationMs** – sets the duration in milliseconds for which the Tone output is performed. A value of zero give unterminated output. The default value is zero

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.28. static GPIO_Result stopTone(int pinNum);

Stops any current Tone output on a pin.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.29. static GPIO_Result getToneFreq(int pinNum, int &freq);

Returns the currently set Tone frequency for a pin.

Parameters:

- **int pinNum** – the number of the pin
- **int &freq** – returns Tone frequency in Hz

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.30. static GPIO_Result getToneDuration(int pinNum, int &duration);

Returns the currently set Tone duration for a pin

Parameters:

- **int pinNum** – the number of the pin
- **inr &duration** – returns Tone duration

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.31. static GPIO_Result isToneRunning(int pinNum, bool &running);

Returns an indication of whether or not Tone is currently running on a pin

Parameters:

- **int pinNum** – the number of the pin
- **bool &running** – returns **true** if Tone is running; **false** if Tone is not running

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.32. static GPIO_Result shiftOut(int dataPinNum, int clockPinNum, int val, long int clockPeriodNS, GPIO_Bit_Order bitOrder = GPIO_MSB_FIRST);

Performs an 8 bit shift out operation using indicated data and clock pins. Intended to send an 8 bit value serially to (e.g.) a shift register.

Parameters:

- **int dataPinNum** – the number of the pin for the data
- **int clockPinNum** – the number of the pin for the clock
- **int val** – the value to be sent

- **long int clockPeriodNS** – the duration of each clock cycle in nano-seconds
- **GPIO_BIT_ORDER bitOrder** – the order that the bits are transferred. One of:
    - **GPIO_MSB_FIRST** for most significant bit first
    - **GPIO_LSB_FIRST** for least significant bit first

  Defaults to **GPIO_MSB_FIRST**

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.33. *static GPIO_Result shiftIn(int dataPinNum, int clockPinNum, long int clockPeriodNS, GPIO_Bit_Order bitOrder, int &value);*

Performs an 8 bit shift in operation using indicated data and clock pins. Intended to receive an 8 bit value serially from (e.g.) a shift register.

Parameters:

- **int dataPinNum** – the number of the pin for the data
- **int clockPinNum** – the number of the pin for the clock
- **long int clockPeriodNS** – the duration of each clock cycle in nano-seconds
- **GPIO_BIT_ORDER bitOrder** – the order that the bits are transferred. One of:
    - **GPIO_MSB_FIRST** for most significant bit first
    - **GPIO_LSB_FIRST** for least significant bit first
- **int &value** – returns the value received

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.34. *static GPIO_Result pulseOut(int pinNum, long int pulseLenUS, int pulseLevel = 1);*

Sends a single pulse on the given pin.

Parameters:

- **int pinNum** – the number of the pin
- **long int pulseLenUS** – the length of the pulse in micro-seconds
- **int pulseLevel** – the polarity of the pulse: 1 or 0. Defaults to 1

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.35. *static GPIO_Result pulseIn(int pinNum, int pulseLevel, long int timeoutUS, long int &value);*

Awaits and returns the length of an input pulse on the given pin. Returns when the pulse is received or any time out has expired – whichever comes first.

Parameters:

- **int pinNum** – the number of the pin
- **int pulseLevel** – the polarity of the pulse: 1 or 0. Defaults to 1
- **long int timeoutUS** – the time in micro-seconds to wait for the pulse. If zero, waits indefinitely.
- **long int &value** – returns the length of the pulse in micro-seconds

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.36. static GPIO_Result pulseIn(int pinNum, GPIO_PulseIn_Handler_Func handler , int pulseLevel = 1, long int timeoutUS = 0);

Starts a pulse in operation and returns immediately. The **GPIO_PulseIn_Handler_Func** will be called when the input pulse is completed.

NOTE: pulse input is run on a separate thread for the pin. When this method is called the thread will be started and the call to the method then returns. The thread continues to run until one of the following occurs:

- the pulse in is completed
- the **stopPulseIn** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates
- the timeout expires

Parameters:

- **int pinNum** – the number of the pin
- **GPIO_PulseIn_Handler_Func handler** – the handler function to be called on completion of the pulse input
- **int pulseLevel** – the polarity of the pulse: 1 or 0. Defaults to 1
- **long int timeoutUS** – the time in micro-seconds to wait for the pulse. If zero, waits indefinitely. Defaults to 0

Returns:

- Result of the call as a **GPIO_Result** value

*6.2.1.37. static GPIO_Result pulseIn(int pinNum, GPIO_PulseIn_Handler_Object * handlerObj , int pulseLevel = 1, long int timeoutUS = 0);*

Starts a pulse in operation and returns immediately. The **GPIO_PulseIn_Handler_Object** will be called when the input pulse is completed.

NOTE: pulse input is run on a separate thread for the pin. When this method is called the thread will be started and the call to the method then returns. The thread continues to run until one of the following occurs:

- the pulse in is completed

- the **stopPulseIn** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates
- the timeout expires

Parameters:

- **int pinNum** – the number of the pin
- **GPIO_PulseIn_Handler_Object * handlerObj** – the handler object to be called on completion of the pulse input
- **int pulseLevel** – the polarity of the pulse: 1 or 0. Defaults to 1
- **long int timeoutUS** – the time in micro-seconds to wait for the pulse. If zero, waits indefinitely. Defaults to 0

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.38. static GPIO_Result stopPulseIn(int pinNum);

Stops any current pulse input on a pin.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.39. static GPIO_Result isPulseInRunning(int pinNum, bool &running);

Returns an indication of whether or not pulse input is currently running on a pin

Parameters:

- **int pinNum** – the number of the pin
- **bool &running** – returns **true** if pulse in is running; **false** if pulse in is not running

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.40. static GPIO_Result getFrequency(int pinNum, long int sampleTimeMS, long int &freq);

Attempts to return the input frequency of a signal on the given pin.

NOTE: This method works by measuring the average period of signal changes on the input pin during the given sample time. It does so by using interrupts on the pin and measuring the time between successive interrupts. Consequently:

- The actual value may be influenced by any other processes running on the system

- The accuracy is likely to be improved by a greater sample time at the expense of greater run time
- The method is only reliable up to a few KHz input frequency

Parameters:

- **int pinNum** – the number of the pin
- **long int sampleTimeMS** – the time in milli-seconds over which the input frequency is sampled
- **long int &freq** – returns the detected input frequency in Hz

Returns:

- Result of the call as a **GPIO_Result** value

6.2.1.41. *static bool isPinUsable(int pinNum);*

Returns an indication as to whether or not a specific pin number can be used for a GPIO pin.

Parameters:

- **int pinNum** – the number of the pin

Returns:

- **true** or **false –** indicating whether or not **pinNum** is a valid GPIO pin

6.2.1.42. *static bool isAccessOk();*

Returns an indication as to whether or not the GPIO hardware is accessible.

Parameters:

- <none>

Returns:

- **true** or **false –** indicating whether or not the hardware is accessible

6.3. Class GPIOPin

The **GPIOPin** class represents instances of a GPIO pin.

6.3.1. GPIOPin Constructor and Destructor

6.3.1.1. *Constructor - GPIOPin(int pinNum);*

Creates a new GPIOPin instance for a given pin.

Parameters:

- **int pinNum** – the pin number

6.3.1.2. *Destructor - ~GPIOPin(void);*
Destroys an instance of a GPIOPin.

NOTE: This also ensures that any threads for the pin are terminated.

Parameters:

- <none>

6.3.2. GPIOPin Public Methods
Note that in general, the success or failure of any method is returned as a **GPIO_Result** value from each method.

6.3.2.1. *GPIO_Result setDirection(GPIO_Direction dir);*
Sets the direction of the GPIOPin.

Parameters:

- **GPIO_Direction dir** – the direction to set the pin to

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.2. *GPIO_Result getDirection(GPIO_Direction &dir);*
Obtains the current direction of the GPIOPin

Parameters:

- **GPIO_Direction &dir** – returns the current direction of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.3. *GPIO_Result set(int value);*
Sets the value of the GPIOPin.

Parameters:

- **int value** – the value to set the pin to

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.4. *GPIO_Result get(int &value);*
Directly returns the value of the GPIOPin.

Parameters:

- **int &value** – returns the current value of the pin

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.5. GPIO_Result setPWM(int freq, int duty, int durationMs = 0);

Starts the PWM output on the GPIOPin with the given frequency and duty values.

NOTE: PWM output is performed by software toggling of the state of the pin. Consequently, the frequency and duty cycle of the output is only reasonably accurate up to a few hundred Hz.

NOTE: PWM output on a pin is run on a separate thread for that pin. When this method is called the thread will be started (or its data updated if it is already running) and the call to the method then returns. The thread continues to run until one of the following occurs:

- the **stopPWM** method is called for the pin
- the GPIOPin destructor for the pin is called
- the process that started the thread (i.e. made the call to this method) terminates
- the duration expires

Parameters:

- **int freq** – sets the PWM frequency in Hz
- **int duty** – sets the PWM duty cycle percentage
- **int durationMS** – the duration in milliseconds to keep the PWM running. A value of zero gives unterminated output. The default is zero

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.6. GPIO_Result startPWM(int durationMs = 0);

Starts the PWM output on the GPIOPin using the last used frequency and duty values

Parameters:

- **int durationMS** – the duration in milliseconds to keep the PWM running. A value of zero gives unterminated output. The default is zero

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.7. GPIO_Result stopPWM();

Stops any current PWM output on the GPIOPin

Parameters:

- <none>

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.8. GPIO_Result getPWMFreq(int &freq);

Returns the currently set PWM frequency for the GPIOPin

Parameters:

- **int &freq** – returns the PWM frequency in Hz

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.9. GPIO_Result getPWMDuty(int &duty);

Returns the currently set PWM duty cycle percentage for the GPIOPin

Parameters:

- **int &duty** – returns the PWM duty cycle percentage

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.10. GPIO_Result getPWMDuration(int &duration);

Returns the currently set PWM duration for the GPIOPin

Parameters:

- **int &duration** – returns the PWM duration

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.11. GPIO_Result isPWMRunning(bool &running);

Returns an indication of whether or not PWM is currently running on the GPIOPin

Parameters:

- **bool &running** – returns **true** if PWM is running; **false** if PWM is not running

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.12. GPIO_Result setIrq(GPIO_Irq_Type type,GPIO_Irq_Handler_Func handler,long int debounceMs=0);

Setups up interrupt handling for the GPIOPin with a given handler function.

NOTE: IRQ handling on a pin is run on a separate thread for that pin. When this method is called the thread will be started and the call to the method then returns. The thread continues to run and to call the handler function whenever the relevant interrupt occurs until one of the following occurs:

- the **resetIrq** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates

Parameters:

- **GPIO_Irq_Type type** – specifies the interrupt type to apply
- **GPIO_Irq_Handler_Func handler** – a pointer to the function to be called to handle the interrupt
- **Long int debounceMs = 0** – specifies an optional debounce period in milliseconds to be applied.
  Default value is **0**
  If value is **0** no debounce handling is applied
  Debounce handling is used to deal with interrupts that come from a potentially noisy mechanical source such as buttons or switches.
  When a non-zero value is used for **debounceMs**, any input signal changes that would normally cause an interrupt but which occur within a time less than the **debounceMs** time since the previous signal change will be ignored so as not to trigger handling of a false signal.

Returns:

- Result of the call as a **GPIO_Result** value

*6.3.2.13. GPIO_Result setIrq(GPIO_Irq_Type type, GPIO_Irq_Handler_Object * handlerObj, long int debounceMs = 0);*

Setups up interrupt handling for the GPIOPin with a given handler object.

NOTE: IRQ handling on a pin is run on a separate thread for that pin. When this method is called the thread will be started and the call to the method then returns. The thread continues to run and to call the handler method of the object whenever the relevant interrupt occurs until one of the following occurs:

- the **resetIrq** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates

Parameters:

- **GPIO_Irq_Type type** – specifies the interrupt type to apply
- **GPIO_Irq_Handler_Object handlerObj** – a pointer to the handler object to be used to handle the interrupt
- **long int debounceMs = 0** – specifies an optional debounce period in milliseconds to be applied.
  Default value is **0**
  If value is **0** no debounce handling is applied

Debounce handling is used to deal with interrupts that come from a potentially noisy mechanical source such as buttons or switches.
When a non-zero value is used for **debounceMs**, any input signal changes that would normally cause an interrupt but which occur within a time less than the **debounceMs** time since the previous signal change will be ignored so as not to trigger handling of a false signal.

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.14. GPIO_Result resetIrq();

Removes any interrupt handling for the GPIOPin.

Parameters:

- <none>

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.15. GPIO_Result enableIrq();

Enables interrupt handling for the GPIOPin that has previously been disabled by **disableIrq**.

Parameters:

- <none>

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.16. GPIO_Result disableIrq();

Disables interrupt handling for the GPIOPin that has previously been enabled by **enableIrq**.

Parameters:

- <none>

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.17. GPIO_Result enableIrq(bool enable);

Enables or Disables interrupt handling for the GPIOPin according to parameter.

Parameters:

- **bool enable** – indicates whether interrupt handling is to be enabled (**true**) or disabled (**false**)

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.18. GPIO_Result irqEnabled(bool &enabled);

Returns an indication as to whether interrupt handling is currently enabled or disabled for the GPIOPin.

Parameters:

- **bool &enabled** – returns **true** if interrupt handling is enabled; **false** if interrupt handling is disabled

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.19. GPIO_Irq_Type getIrqType();

Returns the current interrupt type for the GPIOPin.

Parameters:

- <none>

Returns:

- the interrupt type

6.3.2.20. GPIO_Irq_Handler_Func getIrqHandler();

Returns the any currently established interrupt handler function for the GPIOPin.

Parameters:

- <none>

Returns:

- the interrupt handler function

*6.3.2.21. GPIO_Irq_Handler_Object * getIrqHandlerObj();*

Returns the any currently established interrupt handler object for the GPIOPin.

Parameters:

- <none>

Returns:

- pointer to the interrupt handler object

6.3.2.22. GPIO_Result setTone(int freq, int durationMs = 0);

Starts the tone output on the pin with the given frequency. Tone output is equivalent to PWM output with a 50% duty cycle.

NOTE: Tone output is performed by software toggling of the state of the pin. Consequently, the frequency of the output is only reasonably accurate up to a few hundred Hz.

NOTE: Tone output on a pin is run on a separate thread for that pin. When this method is called the thread will be started (or its data updated if it is already running) and the call to the method then returns. The thread continues to run until one of the following occurs:

- the **stopTone** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates
- the duration expires

Parameters:

- **int freq** – sets the Tone frequency in Hz
- **int durationMs** – sets the duration in milliseconds for which the Tone output is performed. A value of zero give unterminated output. The default value is zero

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.23. GPIO_Result startTone(int durationMs = 0);

Starts the Tone output on the pin using the last used frequency value for the pin.

Parameters:

- **int durationMs** – sets the duration in milliseconds for which the Tone output is performed. A value of zero give unterminated output. The default value is zero

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.24. GPIO_Result stopTone();

Stops any current Tone output on the pin.

Parameters:

- <none>

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.25. GPIO_Result getToneFreq(int &freq);

Returns the currently set Tone frequency for the pin.

Parameters:

- **int &freq** – returns the Tone frequency in Hz

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.26. GPIO_Result getToneDuration(int &duration);

Returns the currently set Tone duration for the pin

Parameters:

- **int &duration** – returns the Tone duration

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.27. GPIO_Result isToneRunning(bool &running);

Returns an indication of whether or not Tone is currently running on the pin

Parameters:

- **bool &running** – returns **true** if Tone is running; **false** if Tone is not running

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.28. GPIO_Result pulseOut(long int pulseLenUS, int pulseLevel = 1);

Sends a single pulse on the pin.

Parameters:

- **long int pulseLenUS** – the length of the pulse in micro-seconds
- **int pulseLevel** – the polarity of the pulse: 1 or 0. Defaults to 1

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.29. GPIO_Result int pulseIn(int pulseLevel, long int timeoutUS, long int &value);

Awaits and returns the length of an input pulse on the pin. Returns when the pulse is received or any time out has expired – whichever comes first.

Parameters:

- **int pulseLevel** – the polarity of the pulse: 1 or 0.
- **long int timeoutUS** – the time in micro-seconds to wait for the pulse. If zero, waits indefinitely
- **long int &value** – returns the length of the pulse in micro-seconds

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.30. GPIO_Result pulseIn(GPIO_PulseIn_Handler_Func handler , int pulseLevel = 1, long int timeoutUS = 0);

Starts a pulse in operation and returns immediately. The **GPIO_PulseIn_Handler_Func** will be called when the input pulse is completed.

NOTE: pulse input is run on a separate thread for the pin. When this method is called the thread will be started and the call to the method then returns. The thread continues to run until one of the following occurs:

- the pulse in is completed
- the **stopPulseIn** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates
- the timeout expires

Parameters:

- **GPIO_PulseIn_Handler_Func handler** – the handler function to be called on completion of the pulse input
- **int pulseLevel** – the polarity of the pulse: 1 or 0. Defaults to 1
- **long int timeoutUS** – the time in micro-seconds to wait for the pulse. If zero, waits indefinitely. Defaults to 0

Returns:

- Result of the call as a **GPIO_Result** value

*6.3.2.31. GPIO_Result pulseIn(GPIO_PulseIn_Handler_Object * handlerObj , int pulseLevel = 1, long int timeoutUS = 0);*

Starts a pulse in operation and returns immediately. The **GPIO_PulseIn_Handler_Object** will be called when the input pulse is completed.

NOTE: pulse input is run on a separate thread for the pin. When this method is called the thread will be started and the call to the method then returns. The thread continues to run until one of the following occurs:

- the pulse in is completed
- the **stopPulseIn** method is called for the pin
- the process that started the thread (i.e. made the call to this method) terminates
- the timeout expires

Parameters:

- **GPIO_PulseIn_Handler_Object * handlerObj** – the handler object to be called on completion of the pulse input
- **int pulseLevel** – the polarity of the pulse: 1 or 0. Defaults to 1

- **long int timeoutUS** – the time in micro-seconds to wait for the pulse. If zero, waits indefinitely. Defaults to 0

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.32. GPIO_Result stopPulseIn();

Stops any current pulse input on the pin.

Parameters:

- <none>

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.33. GPIO_Result isPulseInRunning(bool &running);

Returns an indication of whether or not pulse input is currently running on the pin

Parameters:

- **bool &running** – returns **true** if pulse in is running; **false** if pulse is not running

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.34. GPIO_Result int getFrequency(long int sampleTimeMS, long int &freq);

Attempts to return the input frequency of a signal on the pin.

NOTE: This method works by measuring the average period of signal changes on the input pin during the given sample time. It does so by using interrupts on the pin and measuring the time between successive interrupts. Consequently:

- The actual value may be influenced by any other processes running on the system
- The accuracy is likely to be improved by a greater sample time at the expense of greater run time
- The method is only reliable up to a few KHz input frequency

Parameters:

- **long int sampleTimeMS** – the time in milli-seconds over which the input frequency is sampled
- **long int &freq** – returns the detected input frequency in Hz

Returns:

- Result of the call as a **GPIO_Result** value

6.3.2.35. *int getPinNumber();*

Returns the pin number for the GPIOPin

Parameters:

- <none>

Returns:

- The pin number

6.4. Class GPIOShiftIn

The **GPIOShiftIn** class represents instances of a shift input using 2 GPIO pins – one for the shift data and one for the shift clock.

6.4.1. GPIOShiftIn Constructor and Destructor

6.4.1.1. Constructor - GPIOShiftIn(int dataPin, int clockPin);

Creates a new GPIOShiftIn instance specific that uses the specified pins.

Parameters:

- **int dataPin** – the pin number to be used for the data
- **int clockPin** – the pin number to be used for the clock

6.4.1.2. Destructor - ~GPIOShiftIn(void);

Destroys an instance of a GPIOShiftIn.

Parameters:

- <none>

6.4.2. GPIOShiftIn Public Methods

6.4.2.1. GPIO_Result read(int &value);

Reads the data on the data pin using the clock on the clock pin.

Parameters:

- **int &value** – returns the value read

Returns:

- Result of the call as a **GPIO_Result** value

6.4.2.2. void setClockPeriodNS(long int clockPerNS);

Sets the period of one clock cycle on the clock pin.

Parameters:

- **long int clockPerNS** – the clock period in nano-seconds

Returns:

- <none>

6.4.2.3. long int getClockPeriodNS();

Returns the period of one clock cycle on the clock pin.

Parameters:

- <none>

Returns:

- the clock period in nano-seconds

6.4.2.4. void setBitOrder(GPIO_Bit_Order bitOrd);

Sets the bit order to be used for the shift in.

Parameters:

- **GPIO_Bit_Order bitOrd** – the bit order to be used

Returns:

- <none>

6.4.2.5. GPIO_Bit_Order getBitOrder();

Returns the bit order used for the shift in.

Parameters:

- <none>

Returns:

- the bit order used

*6.4.2.6. GPIOPin * getDataPin();*

Returns a reference to the GPIOPin instance used for the data.

Parameters:

- <none>

Returns:

- A reference to the GPIOPin

*6.4.2.7. GPIOPin * getClockPin();*

Returns a reference to the GPIOPin instance used for the clock.

Parameters:

- <none>

Returns:

- A reference to the GPIOPin

6.5. Class GPIOShiftOut

The **GPIOShiftOut** class represents instances of a shift output using 2 GPIO pins – one for the shift data and one for the shift clock.

6.5.1. GPIOShiftOut Constructor and Destructor

6.5.1.1. Constructor - GPIOShiftOut(int dataPin, int clockPin);

Creates a new GPIOShiftOut instance specific that uses the specified pins.

Parameters:

- **int dataPin** – the pin number to be used for the data
- **int clockPin** – the pin number to be used for the clock

6.5.1.2. Destructor - ~GPIOShiftOut(void);

Destroys an instance of a GPIOShiftOut.

Parameters:

- <none>

6.5.2. GPIOShiftOut Public Methods

6.5.2.1. GPIO_Result write(int val);

Writes a value on the data pin using the clock on the clock pin.

Parameters:

- **int val** – the value to be written

Returns:

- Result of the call as a **GPIO_Result** value

6.5.2.2. void setClockPeriodNS(long int clockPerNS);

Sets the period of one clock cycle on the clock pin.

Parameters:

- **long int clockPerNS** – the clock period in nano-seconds

Returns:

- <none>

6.5.2.3. long int getClockPeriodNS();

Returns the period of one clock cycle on the clock pin.

Parameters:

- <none>

Returns:

- the clock period in nano-seconds

6.5.2.4. void setBitOrder(GPIO_Bit_Order bitOrd);

Sets the bit order to be used for the shift out.

Parameters:

- **GPIO_Bit_Order bitOrd** – the bit order to be used

Returns:

- <none>

6.5.2.5. GPIO_Bit_Order getBitOrder();

Returns the bit order used for the shift out.

Parameters:

- <none>

Returns:

- the bit order used

*6.5.2.6. GPIOPin * getDataPin();*

Returns a reference to the GPIOPin instance used for the data.

Parameters:

- <none>

Returns:

- A reference to the GPIOPin

*6.5.2.7. GPIOPin * getClockPin();*

Returns a reference to the GPIOPin instance used for the clock.

Parameters:

- <none>

Returns:

- A reference to the GPIOPin

6.6. Class RGBLED

The **RGBLED** class represents instances of an RGB led that uses 3 GPIO pins to control the led. A specific constructor is provided that directly represents and controls access to the Omega Expansion Dock led.

6.6.1. RGBLED Constructors and Destructor

6.6.1.1. Constructor - RGBLED();

Creates a new RGBLED instance specific for access to the expansion dock led.

Use of this constructor is equivalent to:

- **RGBLED(17, 16, 15);**

Parameters:

- <none>

6.6.1.2. Constructor - RGBLED(int redPin, int greenPin, int bluePin);

Creates a new RGBLED instance that uses the given pins.

Parameters:

- **int redPin** – the pin number for the red component of the led
- **int greenPin** – the pin number for the green component of the led
- **int bluePin** – the pin number for the blue component of the led

6.6.1.3. Destructor - ~RGBLED(void);

Destroys an instance of an RGBLED.

Parameters:

- <none>

6.6.2. RGBLED Public Methods

6.6.2.1. <void> setColor(int redVal, int greenVal, int blueVal);

Sets the colour of the led according to the parameters.

Parameters:

- **int redVal** – the value for the red component – in the range 0 (off) to 100 (fully on).
- **int greenVal** – the value for the green component – in the range 0 (off) to 100 (fully on).
- **int blueVal** – the value for the blue component – in the range 0 (off) to 100 (fully on).

Returns:

- <none>

6.6.2.2. *<void> setRed(int redVal);*

Sets the red component of the led according to the parameter.

Parameters:

- **int redVal** – the value for the red component – in the range 0 (off) to 100 (fully on).

Returns:

- <none>

6.6.2.3. *<void> setGreen(int greenVal);*

Sets the green component of the led according to the parameter.

Parameters:

- **int greenVal** – the value for the green component – in the range 0 (off) to 100 (fully on).

Returns:

- <none>

6.6.2.4. *<void> setBlue(int blueVal);*

Sets the blue component of the led according to the parameter.

Parameters:

- **int blueVal** – the value for the blue component – in the range 0 (off) to 100 (fully on).

Returns:

- <none>

6.6.2.5. *int getRed();*

Returns the setting of the red component of the led.

Parameters:

- <none>

Returns:

- The current value for the red component

6.6.2.6. *int getGreen();*

Returns the setting of the green component of the led.

Parameters:

- <none>

Returns:

- The current value for the green component

6.6.2.7. *int getBlue();*

Returns the setting of the blue component of the led.

Parameters:

- <none>

Returns:

- The current value for the blue component

6.6.2.8. *GPIOPin * getRedPin();*

Returns a reference to the GPIOPin used to control the red component of the led.

Parameters:

- <none>

Returns:

- Reference to the GPIOPin for the red component

6.6.2.9. *GPIOPin * getGreenPin();*

Returns a reference to the GPIOPin used to control the green component of the led.

Parameters:

- <none>

Returns:

- Reference to the GPIOPin for the green component

6.6.2.10. *GPIOPin * getBluePin();*

Returns a reference to the GPIOPin used to control the blue component of the led.

Parameters:

- <none>

Returns:

- Reference to the GPIOPin for the blue component

6.6.2.11. *void setActiveLow(bool actLow);*

Sets whether the led uses active low control (i.e. a low value is output to turn a component on) or active high control (i.e. a high value is output to turn a component on).

By default, activeLow is set to true as is used by the expansion dock led.

Parameters:

- **bool actLow** – sets whether activeLow is enabled or not

Returns:

- <none>

6.6.2.12. bool isActiveLow();

Returns whether or not activeLow is set for the RGBLED.

Parameters:

- <none>

Returns:

- Whether or not activeLow is set

7. Further Development

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