new-gpio

Version 1.2 – 18 January 2016
Kit Bishop

Document History		
Version	**Date**	**Change Details**
Version 1.0	26 November 2015	Initial version
Version 1.1	5 December 2015	Added PWM control methods in GPIOAccess and GPIOPin classes in libnew-gpio.so library Added PWM control in new-gpiotest program Some reorganisation of contents of this document
Version 1.2	18 January 2016	Added IRQ functionality to GPIOAccess and GPIOPin classes in libnew-gpio.so library Added control of IQ in new-gpiotest program Changes in method of obtaining result of method calls Additional minor changes to parameters to new_gpiotest program Add section on pre-requisites for IRQ usage

Contents

1. Background

new-gpio 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

new-gpio consists of two main components:

- **libnew-gpio.so** – a dynamic library containing the classes used to intercat with GPIO pins
- **new-gpiotest** – a simple test program for interacting with GPIO pins using **libnew-gpio.so**

These components are described in more details in this document, as are the files contained in the package supplied with this document.

The software was developed under NetBeans 8.1 running on a KUbuntu-14.04 system running in a VirtualBox VM.

Details of setting up the tool-chain for building and the usage of NetBeans can be found at:

- How to install gcc (https://community.onion.io/topic/9/how-to-install-gcc)
- Using NetBeans to compile C/C++ code for Omega (https://community.onion.io/topic/125/using-netbeans-to-compile-c-c-code-for-omega)

new-gpio 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 in **fast-gpio**

2. Pre-requisites

To use theGPIO interrupt handling facilities in new-gpio, 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

new-gpio is supplied in an archive file named **new-gpio-1.2.tar.bz2**. This archive contains the following:

- **new-gpio.pdf** – this documentation as a PDF file
- **libnew-gpio.so** – the built dynamic library
- **new-gpiotest** – the built test program
- **new-gpio-src** – a directory containing the source files for the **libnew-gpio.so** library
- **new-gpiotest-src** – a directory containing the source file for the **new-gpiotest** program

4. Basic Installation

Installing the software is simple. It primarily consists of copying the library and test program to suitable locations on your Omega.

4.1. Installing libnew-gpio.so

Copy the **libnew-gpio.so** file 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
```

4.2. Installing the new-gpiotest Program

Copy the **new-gpiotest** program file to any suitable directory on your Omega from which you wish to run it.

5. Description of the libnew-gpio.so Library

The **libnew-gpio.so** library contains three main components for access and usage of new-gpio and two 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

The internal use only components and their source files are:

- **GPIOPwmPin** – a support class used only internal by **GPIOAccess** to provide PWM facilities for a pin
  Files:

  GPIOPwmPin.h
  GPIOPwmPin.cpp
- **GPIOIrqInf** – defines internal type for support of GPIO interrups
  File:

  GPIOIrqInf.h

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

5.1. GPIOTypes

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

5.1.1. 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

5.1.2. 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

5.1.3. 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

5.1.4. 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. Ifno 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>

5.1.5. GPIO_Irq_Handler_Object

GPIO_Irq_Handler_Object is an 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>

5.2. 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.

5.2.1. GPIOAccess Public Methods

Note that in general, the success or failure of any method can be ascertained by calling the **getlastResult** immediately after the call to the particular method.

5.2.1.1. static void 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:

- <none>

5.2.1.2. static GPIO_Direction getDirection(int pinNum);

Queries the direction of a pin.

Parameters:

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

Returns:

- The current direction of the pin

5.2.1.3. static void 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:

- <none>

5.2.1.4. static int get(int pinNum);

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

Parameters:

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

Returns:

- The current state of the pin

5.2.1.5. static void setPWM(int pinNum, int freq, int duty);

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

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

Parameters:

- **int pinNum** – the number of the pin
- **int freq** – sets the PWM frequency in Hz
- **int duty** – sets the PWM duty cycle percentage

Returns:

- <none>

5.2.1.6. static void startPWM(int pinNum);

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

Returns:

- <none>

5.2.1.7. static void stopPWM(int pinNum);

Stops any current PWM output on a pin.

Parameters:

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

Returns:

- <none>

5.2.1.8. static int getPWMFreq(int pinNum);

Returns the currently set PWM frequency for a pin.

Parameters:

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

Returns:

- The PWM frequency in Hz

5.2.1.9. static int getPWMDuty(int pinNum);

Returns the currently set PWM duty cycle percentage for a pin

Parameters:

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

Returns:

- The PWM duty cycle percentage

5.2.1.10. static void 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:

- <none>

*5.2.1.11. static void 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:

- <none>

5.2.1.12. static void resetIrq(int pinNum);

Removes any interrupt handling for a pin.

Parameters:

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

Returns:

- <none>

5.2.1.13. static void 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:

- <none>

5.2.1.14. static void 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:

- <none>

5.2.1.15. static void 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:

- <none>

5.2.1.16. static bool irqEnabled(int pinNum);

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

Parameters:

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

Returns:

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

5.2.1.17. 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

5.2.1.18. 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

*5.2.1.19. 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

5.2.1.20. static void enableIrq();

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

Parameters:

- <none>

Returns:

- <none>

5.2.1.21. static void disableIrq();

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

Parameters:

- <none>

Returns:

- <none>

5.2.1.22. 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>

5.2.1.23. 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

5.2.1.24. static bool isPWMRunning(int pinNum);

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

Parameters:

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

Returns:

- **true** if PWM is running; **false** if PWM is not running

5.2.1.25. 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

5.2.1.26. 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

5.2.1.27. static GPIO_Result getLastResult();

Returns the result of the latest call to other methods.

Parameters:

- <none>

Returns:

- The result of the last method call

5.3. Class GPIOPin

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

5.3.1. GPIOPin Constructor and Destructor

5.3.1.1. Constructor - GPIOPin(int pinNum);

Creates a new GPIOPin instance for a given pin.

Parameters:

- **Int pinNum** – the pin number

5.3.1.2. Destructor - ~GPIOPin(void);

Destroys an instance of a GPIOPin.

NOTE: This also ensures that any PWM thread for the pin is terminated.

Parameters:

- <none>

5.3.2. GPIOPin Public Methods

Note that in general, the success or failure of any method can be ascertained by calling the **getlastResult** immediately after the call to the particular method.

5.3.2.1. <void> setDirection(GPIO_Direction dir);

Sets the direction of the GPIOPin.

Parameters:

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

Returns:

- <none>

5.3.2.2. GPIO_Result getDirection();

Obtains the current direction of the GPIOPin

Parameters:

- <none>

Returns:

- The current direction of the pin

5.3.2.3. void set(int value);

Sets the value of the GPIOPin.

Parameters:

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

Returns:

- <none>

5.3.2.4. int get();

Directly returns the value of the GPIOPin.

Parameters:

- <none>

Returns:

- the current value of the pin

5.3.2.5. void setPWM(int freq, int duty);

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

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

Parameters:

- **int freq** – sets the PWM frequency in Hz
- **int duty** – sets the PWM duty cycle percentage

Returns:

- <none>

5.3.2.6. void startPWM();

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

Parameters:

- <none>

Returns:

- <none>

5.3.2.7. void stopPWM();

Stops any current PWM output on the GPIOPin

Parameters:

- <none>

Returns:

- <none>

5.3.2.8. int getPWMFreq();

Returns the currently set PWM frequency for the GPIOPin

Parameters:

- <none>

Returns:

- The PWM frequency in Hz

5.3.2.9. int getPWMDuty();

Returns the currently set PWM duty cycle percentage for the GPIOPin

Parameters:

- <none>

Returns:

- The PWM duty cycle percentage

5.3.2.10. bool isPWMRunning();

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

Parameters:

- <none>

Returns:

- **true** if PWM is running; **false** if PWM is not running

5.3.2.11. void 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:

- <none>

*5.3.2.12. void 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:

- <none>

5.3.2.13. void resetIrq();

Removes any interrupt handling for the GPIOPin.

Parameters:

- <none>

Returns:

- <none>

5.3.2.14. void enableIrq();

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

Parameters:

- <none>

Returns:

- <none>

5.3.2.15. void disableIrq();

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

Parameters:

- <none>

Returns:

- <none>

5.3.2.16. void 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:

- <none>

5.3.2.17. bool irqEnabled();

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

Parameters:

- <none>

Returns:

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

5.3.2.18. GPIO_Irq_Type getIrqType();

Returns the current interrupt type for the GPIOPin.

Parameters:

- <none>

Returns:

- the interrupt type

5.3.2.19. GPIO_Irq_Handler_Func getIrqHandler();

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

Parameters:

- <none>

Returns:

- the interrupt handler function

5.3.2.20. 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

5.3.2.21. int getPinNumber();

Returns the pin number for the GPIOPin

Parameters:

- <none>

Returns:

- The pin number

5.3.2.22. GPIO_Result getLastResult();

Returns the result of the latest call to other methods.

Parameters:

- <none>

Returns:

- The result of the last method call

6. Usage of the new-gpiotest Program

The **new-gpiotest** program accepts a set of parameters to control its operation.

The program will document it's usage when the command **new-gpiotest help** is used.

In addition, the usage is shown whenever any errors are detected in the parameters.

The usage information displayed is:

```
./new-gpiotest
Usage
Commands - one of:
        - ./new-gpiotest <op> <pin> <val>
        - ./new-gpiotest pwm <pin> <freq> <duty>
                Starts PWM output on pin
        - ./new-gpiotest pwmstop <pin>
```

```
                Stops PWM output on pin
        - ./new-gpiotest irq <pin> <irqtype> <irqcmd> <debounce>
                Enables IRQ handling on pin
        - ./new-gpiotest irqstop <pin>
                Terminates IRQ handling on pin
Where:
        <op> is one of:
                info - to display info on pin(s)
                set - to set pin(s) value
                get - to get and return pin value
                setd - to set pin(s) direction
                getd - to get and return pin direction
                help - to display usage
        <pin> is one of
                0, 1, 6, 7, 8, 12, 13, 14, 15, 16, 17, 18, 19, 23, 26, all
                A <pin> of all can only be used for an <op> of:
                        info, set, setd
        <val> is only required for set and setd:
                for set, <val> is 0 or 1
                for setd, <val> is in or out
        <freq> is PWM frequency in Hz > 0
        <duty> is PWM duty cycle % in range 0 to 100
        <irqtype> is the type for IRQ and is one of:
                falling, rising, both
        <irqcmd> is the shell command to be executed when the IRQ occurs
                Must be enclosed in " characters if it contains
                spaces or other special characters
                If it starts with the string [debug],
                debug output is displayed first
        <debounce> is optional debounce time for IRQ in milliseconds
                Defaults to 0 if not supplied
```

Notes:

1. The return value from the command will be one of the following:
    - **255 (-1)** – indicates an error has occurred – either in the parameters or in executing the command
    - **0** – indicates normal successfully completion for an operation (**<op>**) other than **get** or **getd**
    - For a successful **get** operation:
        - **0** – indicates the pin is **off**
        - **1** – indicates the pin is **on**
    - For a successful **getd** operation:
        - **0** – indicates the pin is an **input** pin
        - **1** – indicates the pin is an **output** pin

2. When the **pwm** operation is used, the program forks a separate process to perform the PWM output.
   This separate process continues after the program returns until such time as the **pwmstop** operation is performed on the same pin.
   The ID of the separate process can be discovered by running:
   new-gpiotest info <pin-number>

3. When the **irq** operation is used, the program forks a separate process to monitor and respond to pin state changes.

Each time the relevant pin undergoes the relevant change in state, the **<irqcmd>** command specified is run.
This separate process continues after the program returns until such time as the **irqstop** operation is performed on the same pin.
The ID of the separate process can be discovered by running:

new-gpiotest info <pin-number>

7. Further Development

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

7.1. For the Future

In addition, it is intended that further work be done in the future based on **new-gpio**. In particular:

- Similar code for **i2c** access
- Java class wrappers that will provide access to GPIO and i2c from Java on the Omega