Raspberry Pi GPIO Library (in C)

Written by Bill Vandenberk
Judi McCuaig
Bill Gardner
CIS 4900

LOC
Library: 990
Example Files: 397
Device Abstraction Example Files: 170
Total: 1557
18 Total Public Functions
6 Private Functions
12 Files

Contents

About

This document is to be distributed with the RPi_GPIO library written by Bill Vandenberk, 2013. The library is designed to be an easy to use and understand interface for interacting with the general purpose input/output (GPIO) pins on the Raspberry Pi board. The library is modeled on stdio. The functions carry a similar format to those in the stdio library, see API usage for more information.

There are several files included with this library. You should have the following:

```
Report.pdf
Pin_Label.pdf
Device_Wiring.pdf
README.md
source/
        GPIO.c
        RPi_GPIO.h
        Makefile
        examples/
                example0.c
                example1.c
                example2.c
                example3.c
                example4.c
                example5.c
        device_examples/
                led_abs_eg.c
                phr_abs_eg.c
                tws_abs_eg.c
```

The library uses the sysfs method of interacting with the GPIO pins.

Compiling the Library

Compiling the library is easy. Simply enter the source directory and type 'make'. This will build the library libRPi_GPIO.so.

NOTE: If you have a revision 1 Raspberry Pi, you need to uncomment the #define RPi_board_rev_1 in GPIO.c before compiling. This will make the library adopt the older pin numbers.

Installing the Library

Installing the library is just as easy as compiling it. From the source directory type 'make install' you may be prompted for a password for this step. By default, the library installs to /usr/local/lib and the header is copied to /usr/local/include

NOTE: If you have to recompile the library, you should do the install step again.

Compiling With the Library

Sticking with the idea of things being easy here. Once you compile and install the library, all you need to do is #include <RPi_GPIO.h> and link with -lRPi_GPIO.

Running the Example Code

To run the example code, first read through the code and try to understand it. In each file there is a line in the comment header that allows you to compile the program. Once compiled run the executable (with sudo) and watch the magic unfold. If you really want the magic to unfold, you should connect the specified devices to you Raspberry Pi first.

Terms

There are a few definitions you should know before using the library.
Pin – 1. a physical pin on the raspberry pi board. 1. a software representation of a physical pin.
Logic Type – active high or active low. Active high means 3.3v on a pin is true, active low means 0v on a pin is true.
Direction – pins can be defined as strictly input, strictly output or input and output. The direction of the pin specifies whether you plan on using it as an input or an output, or maybe both.
Read – to read from a pin is to get the logic level on the pin (input pins).
Write – to write to a pin is to apply a logic level on the pin (output pins).
Export – exporting a pin means to make its interface available for us to use.
Unexport – unexporting a pin means to make its interface not available for use.

API Usage

Once included, your program has access to all of these functions

RPi_init

```
/* int RPi_init ()

      This function is meant to be run at the beginning of each program that uses 
the library.
      It will prepare the interface required for the library to work.

      Preconditions: the function has not been called before
      Postconditions: all 17 pins are prepared for use with the library

      Parameters: none
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to 
get more details on the failure
*/
```

Pin ADT functions

RPi_popen

```
/* PIN * RPi_popen(int number, enum RPi_logicType logic, enum RPi_direction direc)

      This function will make and return a PIN.
      This needs to be called to read from or write to a pin.
      Note: This funtion mallocs memory which is freed when you call RPi_pclose.

      Preconditions: the specified pin is not opened yet
      Postconditions: the pin is created and returned

      Parameters: int number - an integer in the range 0-16 representing a pin on
the pinmap
                        enum RPi_logicType logic - logic type to apply to this pin
                        enum RPi_direction direc - intended direction of pin
      Returns: PIN, the pin struct on success, NULL on failure - use RPi_errorno
and RPi_errorstr to get more information about the failure
*/
```

<u>RPi_pread</u>

```
/* int RPi_pread(PIN * p, bool * in)

      This function will read the value on an input or bidirectional pin.

      Preconditions: the pin is open and either an in or inout pin
      Postconditions: the value on the pin is read and the value is placed in bool
* in

      Parameters: PIN p - the pin to be read
                        bool * in - the location to store the read value
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

<u>RPi_pwrite</u>

```
/* int RPi_pwrite(PIN * p, bool value)

      This function will put a value on an output or bidirectional pin.

      Preconditions: the pin is open and either an out or inout pin
      Postconditions: the given value is placed on the pin

      Parameters: PIN p - the pin to be written to
                        bool value - the value to put on the pin
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

<u>RPi_pdirection</u>

```
/* int RPi_pdirection(PIN * p, enum RPi_direction dire)

      This function will change the direction of a pin.

      Preconditions: the pin is open
      Postconditions: the pins direction is changed to the given one.

      Parameters: PIN p - the pin that is to be changed
                        enum RPi_direction - the new direction of the pin
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

RPi_pclose
```
/* int RPi_pclose(PIN * p)

      This function destroys a PIN and frees the memory affiliated with it.

      Preconditions: the specified pin is open
      Postconditions: the specified pin is closed

      Parameters: PIN p - the pin to be destroyed
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

LED Device Abstraction Functions

RPi_LED_open
```
/* LED * RPi_LED_open(int pin);

      This function opens a connection to an LED. This needs to be called before
you can turn the LED on or off. Note: this function mallocs memory that is freed
when you call RPi_LED_close

      Preconditions: the pin that the LED is connected to is not in use already
      Postconditions: an led is created and returned

      Parameters: int pin - the pin that the led is connected to
      Returns: LED *, the created LED
*/
```

RPi_LED_on
```
/* int RPi_LED_on(LED * l);

      This function will turn an LED on.

      Preconditions: the led is open
      Postconditions: the led is turned on

      Parameters: LED * l - the LED to turn on
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

RPi_LED_off
```
/* int RPi_LED_off(LED * l);

      This function turns an LED off.

      Preconditions: the led is open
      Postconditions: the led is turned off

      Parameters: LED * l - the LED to turn off
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

RPi_LED_toggle
```
/* int RPi_LED_toggle(LED * l);

      This function will toggle the current state of an LED.

      Preconditions: the led is open
      Postconditions: the led is turned on if it was previously off and off if it
was previously on

      Parameters: LED * l - the LED to toggle
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

RPi_LED_close
```
/* int RPi_LED_close(LED * l);

      This function closes the connection to an LED. This also frees all memory
associated with the LED.

      Preconditions: the led is open
      Postconditions: the led is closed and the memory is freed

      Parameters: LED * l - the LED to close
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

3-Way-Switch Device Abstraction Functions

RPi_TWS_open
```
/* TWS * RPi_TWS_open(int pin1, int pin2);

      This function creates a connection to a three way switch. Note: this
function mallocs memory that is freed when you call RPi_TWS_close

      Preconditions: the pins that are being used for the 3-way-switch are not
already in use
      Postconditions: the 3-way-switch is created and returned

      Parameters: int pin1 - the first input pin (also connected to ground via a
resistor),
                        int pin2 - the second input pin (also connected to ground
via a resistor)
      Returns: a three-way-switch on success, NULL on failure
*/
```

RPi_TWS_readPosition
```
/* int RPi_TWS_readPosition(TWS * s, int * readto);

      This function reads the position of a 3-way-switch

      Preconditions: the tws is open
      Postconditions: the position of the tws is placed in int * readto

      Parameters: TWS * s - the switch to get the position for,
                        int * readto - the int to read the position into
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

<u>RPi_TWS_close</u>

```
/* int RPi_TWS_close(TWS * s);

      This function will close a three way switch connection

      Preconditions: the tws is open
      Postconditions: the tws is closed and the memory related to it is freed

      Parameters: TWS * s - the switch to close
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

Photoresistor Device Abstraction Functions

<u>RPi_PHR_open</u>

```
/* PHR * RPi_PHR_open(int pin);

      This function creates a photoresistor

      Preconditions: the pins for the photoresistor are not already in use
      Postconditions: the photoresistor is created and returned

      Parameters: int pin - the pin number that the photo resistor is hooked up to
      Returns: PHR *, the photoresistor
*/
```

<u>RPi_PHR_read</u>

```
/* int RPi_PHR_read(PHR * r, bool * readto);

      This function reads the photoresistor. False value means it is dark, true
value means it is bright

      Preconditions: the phr is open
      Postconditions: the value on the phr is placed in bool * readto

      Parameters: PHR * r - the resistor to read from, bool * readto - the boolean
to put the value into
      Returns: 0 on success
*/
```

<u>RPi_PHR_close</u>

```
/* int RPi_PHR_close(PHR * r);

      This function closes the connection to a photoresistor

      Preconditions: the phr is open
      Postconditions: the phr is closed and memory related to it is freed

      Parameters: PHR * r - the photoresistor you are no longer using
      Returns: 0 on success, 1 on failure - use RPi_errorno and RPi_errorstr to
get more details on the failure
*/
```

Error Handling Functions

RPi_errorno

```
/* int RPi_errorno()

      This function can be used to get the error number of the last funciton call
(if a function does not return 0, you can use this function to see what went
wrong.)

      Preconditions: there was an error within the library
      Postconditions: the last error code is returned

      Parameters: none.
      Returns: the last error number
*/
```

Possible Error Codes are 1-11 and unknown error (anything else).

RPi_errorstr

```
/* char * RPi_errorstr(int err)

      This function will return an english readable explanation of an errorno
code.

      Preconditions: none
      Postconditions: the string affiliated with the error code is returned

      Parameters: int err - the error code as gotten from errorno()
      Returns" char *, a string containing information about the error
*/
```

Possible Error Strings are as follows.

Errorno	Error String
1	Unable to create PIN, memory not available.
2	Unable to interact with board, permissions are probably set up incorrectly.
3	Invalid pin number.
4	Recieved an argument that was NULL.
5	Attempt to write to an input only pin.
6	Attempt to read from a write only pin.
7	Unable to create LED, memory not available.
8	Unable to create three-way-switch, memory not available.
9	Unable to create photoresistor, memory not available.
10	Unable to use pin. It appears that it is already in use.
11	Unable to close pin. It appears to already be closed.
Other	Invalid error code.

Deliverables

There are several things this project needs to be a success.

1. No memory leaks.
2. Pin conrol works correctly.
3. Library follows state transition diagram (last in this document).
4. 6 well commented, simple example programs.
5. 3 device abstractions including at least an LED and a switch.
6. Example code for each device.
7. A working PIN ADT.
8. A library demo.

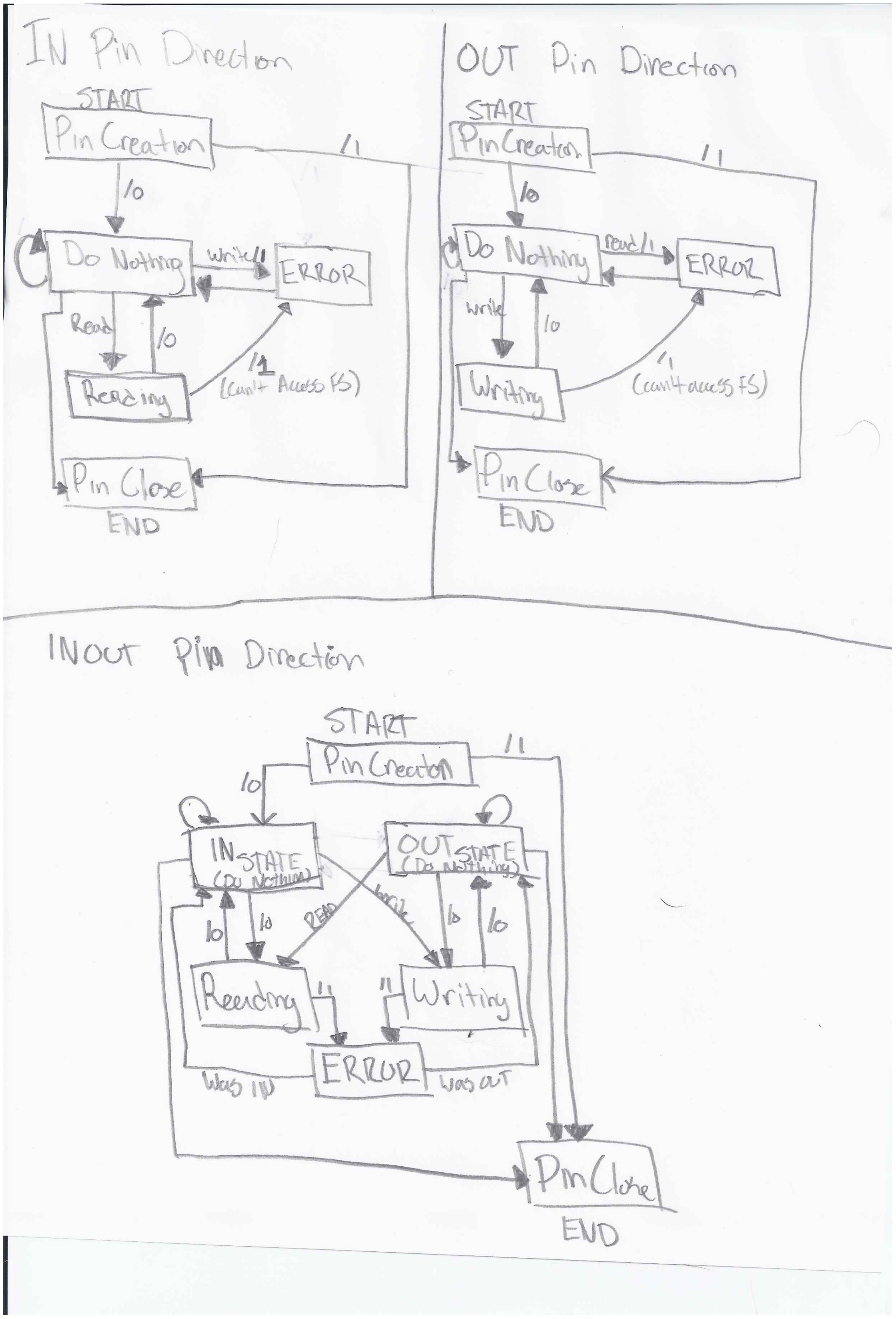
Below is proof that the library is working, has no memory leaks, and has 6 example programs. The project also contains the device abstraction examples and a state transition diagram (below)

MEMTEST RESULTS

```
pi@raspberrypi ~/gpio/CIS4900/examples $ sudo valgrind ./example0
==2853== Command: ./example0
==2853== HEAP SUMMARY:
==2853==     in use at exit: 0 bytes in 0 blocks
==2853==   total heap usage: 50 allocs, 50 frees, 9,274 bytes allocated
==2853== All heap blocks were freed -- no leaks are possible
==2853== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 6)

pi@raspberrypi ~/gpio/CIS4900/examples $ sudo valgrind ./example1
==2857== Command: ./example1
==2857== HEAP SUMMARY:
==2857==     in use at exit: 0 bytes in 0 blocks
==2857==   total heap usage: 47 allocs, 47 frees, 8,877 bytes allocated
==2857== All heap blocks were freed -- no leaks are possible
==2857== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 6)

pi@raspberrypi ~/gpio/CIS4900/examples $ sudo valgrind ./example2
==2861== Command: ./example2
Press enter to toggle the LEDs
Press enter to terminate
==2861== HEAP SUMMARY:
==2861==     in use at exit: 0 bytes in 0 blocks
==2861==   total heap usage: 60 allocs, 60 frees, 11,226 bytes allocated
==2861== All heap blocks were freed -- no leaks are possible
==2861== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 6)

pi@raspberrypi ~/gpio/CIS4900/examples $ sudo valgrind ./example3
==2865== Command: ./example3
==2865== HEAP SUMMARY:
==2865==     in use at exit: 0 bytes in 0 blocks
==2865==   total heap usage: 137 allocs, 137 frees, 25,284 bytes allocated
```

```
==2865== All heap blocks were freed -- no leaks are possible
==2865== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 6)

pi@raspberrypi ~/gpio/CIS4900/examples $ sudo valgrind ./example4
==2869== Command: ./example4
==2869== HEAP SUMMARY:
==2869==     in use at exit: 0 bytes in 0 blocks
==2869==   total heap usage: 69 allocs, 69 frees, 12,790 bytes allocated
==2869== All heap blocks were freed -- no leaks are possible
==2869== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 6)

pi@raspberrypi ~/gpio/CIS4900/examples $ sudo valgrind ./example5
==2873== Command: ./example5
==2873== HEAP SUMMARY:
==2873==     in use at exit: 0 bytes in 0 blocks
==2873==   total heap usage: 68 allocs, 68 frees, 12,405 bytes allocated
==2873== All heap blocks were freed -- no leaks are possible
==2873== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 6)
```

Possible State Transitions


IN Pin Direction
START
Pin Creation
/0
/1
Do Nothing
write/1
ERROR
Read
/0
Reading
/1 (can't Access FS)
Pin Close
END
OUT Pin Direction
START
Pin Creation
/0
/1
Do Nothing
read/1
ERROR
write
/0
Writing
/1 (can't access FS)
Pin Close
END
INOUT Pin Direction
START
Pin Creation
/0
/1
IN STATE (Do Nothing)
OUT STATE (Do Nothing)
/0
/0
READ
write
/0
/0
Reading
Writing
/1
/1
ERROR
Was IN
Was OUT
Pin Close
END