BlackLib - User Manual

Beaglebone Black C++ Library

Author: Yiğit Yüce

31.01.2014

v1.0

CONTENTS

1. ABSTRACT

This file prepared for BlackLib Library documentation. This document's target group is end users. BlackLib Library that is discussed at this document, wrote with C++ programming language, for Beaglebone Black.

2. KEYWORDS

Beaglebone Black, BlackLib, C++, library, PWM, ADC, GPIO

3. INTRODUCTION

BlackLib wrote with C++ programming language, for Beaglebone Black. You can use Beaglebone Black's ADC, GPIO and PWM features by dint of this library. This library can work with Kernel 3.7 and later versions. This library licensed with LGPL license.

4. LICENSE

For any comment or suggestion please contact the creator of BlackLib Library at ygtyce@gmail.com

5. NEED-TO-KNOW

1. Device Tree

The Device Tree is a data structure for describing hardware. Rather than coding every detail of a device into an operating system, a lot of hardware can be described in a data structure. These structures passed to the operating system at boot time.

The structures can hold any kind of data. Device tree data must be laid out in a structure that operating systems can understand. A "bindings" is a description of how a device is described in the device tree. [1]

2. C++11

C++11 (formerly known as C++0x) is the most recent version of the standard of the C++ programming language. [2]

Functions which wrote with C++0x standard, can't use without add C++0x flag to compiler. For this reason when compiling phase you must add to this flag. ”Compiling C++11 with g++” is explicated at this link [3] or if you use eclipse for development, “Eclipse CDT C++11/C++0x support” is explicated at link. [4] You have to do this step for compile your code which uses this library.

6. USAGE

1. Including Library

You must do this for using the library;

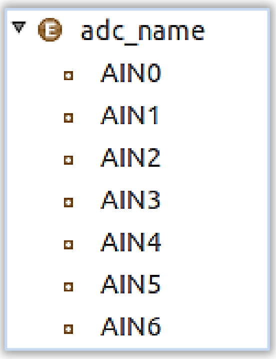
→ Copy the BlackLib directory to your code's directory.
→ Add this code to top of your code.

```
#include "BlackLib/BlackLib.h"
```

2. Special Definitions

After choose Beaglebone Black's which feature to use, you must choose the features' sub options, when appropriate. In our case these options can choose with special definitions. Special definitions are unique for each feature and these names differ from the others. The names defined with enumeration property.

a. ADC Definitions


adc_name
AIN0
AIN1
AIN2
AIN3
AIN4
AIN5
AIN6

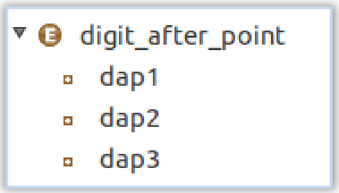
adc_name in ADC definitions, is sub option which you can use when you want read data from Beaglebone Black's analog inputs. You can select which "AINx" input will use for read analog value, by dint of this sub option.

Beaglebone Black has 8 ADCs, even though one of these allocated for Beaglebone Black and disallowed to mount external analog device. Therefore you can read only 7 analog inputs on Beaglebone Black.

ADC name selection is the one and only expected parameter when you initialize the BlackADC class in your code. This parameter can be "*adc_name*" variable. The usage of adc_name option is like below.

```
BlackADC *test = new BlackADC(AIN0);
```

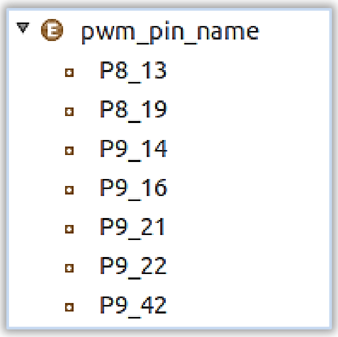
```
BlackADC test(AIN0);
```


digit_after_point
dap1
dap2
dap3

When reading ADC, this read value is mV (millivolt) level. So generally you must scale this millivolt level value. For this reason, "*getParsedValue()*" function are wrote in the library. This function takes one parameter and it can be "*digit_after_point*" type variable. You can define how many digits will exist, after point at converted value and you can make a parsing level selection with this variable. An example below is about this function and usage of "*digit_after_point*" variable.

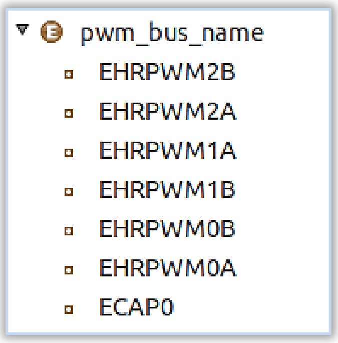
```
float value = test->getParsedValue(dap2);
```

b. PWM Definitions


pwm_pin_name
- P8_13
- P8_19
- P9_14
- P9_16
- P9_21
- P9_22
- P9_42

"*pwm_pin_name* "or "*pwm_bus_name* " are sub options which you can use when you want generate PWM signal from Beaglebone Black's PWM outputs. You can select which PWM output will use for generating pwm signal, by dint of this sub options.

Beaglebone Black has a lot PWM outputs. Available of these outputs defined on the side of page. These PWM outputs located on P8 and P9 ports.


pwm_bus_name
- EHRPWM2B
- EHRPWM2A
- EHRPWM1A
- EHRPWM1B
- EHRPWM0B
- EHRPWM0A
- ECAP0

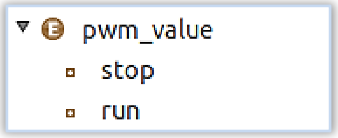
You can use PWM outputs with theirs symbolic names beside their pin names. These symbolic names defined on the side of page. List orders on the side lists are same for both. In other words, "*P8_13*" and "*EHRPWM2B*" are same PWM output like "*P9_42*" and "*ECAP0*".

"*EHRPWM*" represents Enhanced High Resolution Pulse Width Modulation and "*ECAP0*" represents Enhanced Capture Pulse Width Modulation.

Pin or bus name selection is the one and only expected parameter when you initialize the BlackPWM class in your code. This parameter can be "*pwm_pin_name* "or "*pwm_bus_name* "variable. The usages of these options are like below.

```
BlackPWM *test = new BlackPWM(P9_14);
or
BlackPWM *test = new BlackPWM(EHRPWM1A);
```

```
BlackPWM test(P9_14);
or
BlackPWM test(EHRPWM1A);
```


pwm_value
- stop
- run

The signal must start generating for get PWM signal from PWM outputs. "*setSignalState()*" function is wrote in the library for start or stop generating signal. This function takes one parameter and it can be "*pwm_value*" type variable. This variable value can be "*stop*" or "*run*". "*run*" starts generating signal and "stop" stops. An example below is about this function and usage of "*pwm_value*" variable.

```
test->setSignalState(stop);
```

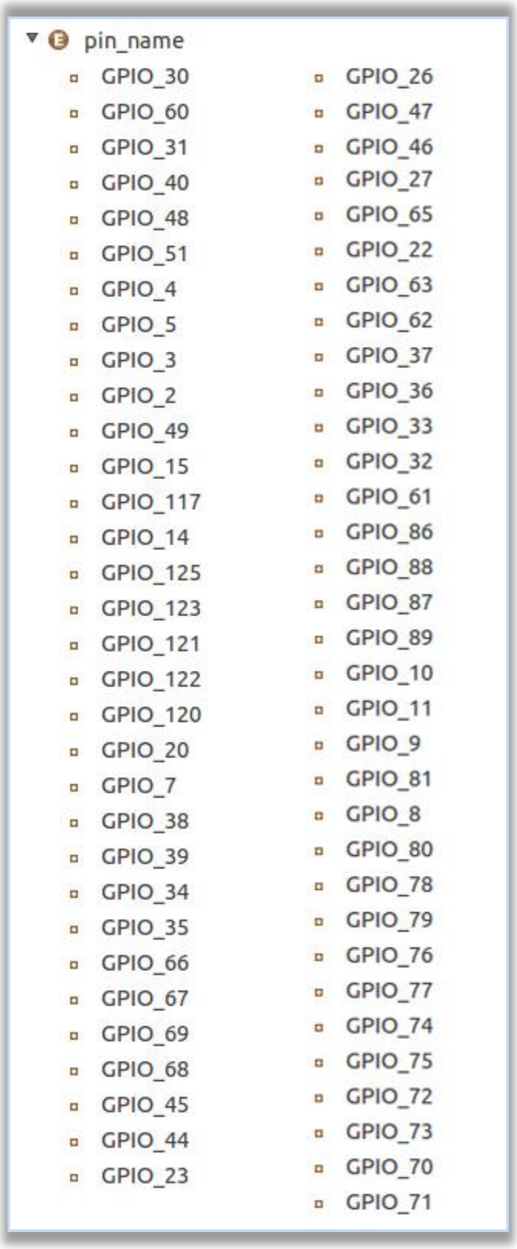
c. GPIO Definitions

"pin_name" is sub option that use for define which GPIO pin will use. You can select which GPIO pin of Beaglebone Black will use, by dint of this sub option.

Beaglebone Black has two GPIO ports and each of these ports have 46 pins. These ports are P8 and P9. But entire pins can't use directly by end user. Some of these pins use for Beaglebone Black's own features such as HDMI output, SD card, LCD screen etc. Hence you should make decision which pins or features will use, before start your project. Even sometimes you may be obliged to load Device Tree for use some pins.

General purpose pins can use two types. These types can be input or output. Input pins use for read digital data from outside, output pins use for generate digital data to outside world. Digital data can be logical 0 or 1. Pin type selection can perform with "*pin_type*" option.

Two parameters are expected from you, when you initialize BlackGPIO class into your code. These are pin number and pin type selections. These parameters are "pin_name" and "pin_type" type variables. Usage of these options is like below.

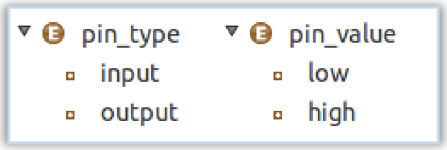


pin_name

GPIO_30	GPIO_26
GPIO_60	GPIO_47
GPIO_31	GPIO_46
GPIO_40	GPIO_27
GPIO_48	GPIO_65
GPIO_51	GPIO_22
GPIO_4	GPIO_63
GPIO_5	GPIO_62
GPIO_3	GPIO_37
GPIO_2	GPIO_36
GPIO_49	GPIO_33
GPIO_15	GPIO_32
GPIO_117	GPIO_61
GPIO_14	GPIO_86
GPIO_125	GPIO_88
GPIO_123	GPIO_87
GPIO_121	GPIO_89
GPIO_122	GPIO_10
GPIO_120	GPIO_11
GPIO_20	GPIO_9
GPIO_7	GPIO_81
GPIO_38	GPIO_8
GPIO_39	GPIO_80
GPIO_34	GPIO_78
GPIO_35	GPIO_79
GPIO_66	GPIO_76
GPIO_67	GPIO_77
GPIO_69	GPIO_74
GPIO_68	GPIO_75
GPIO_45	GPIO_72
GPIO_44	GPIO_73
GPIO_23	GPIO_70
	GPIO_71

```
BlackGPIO *test = new BlackGPIO(GPIO_30, input);
                        or
BlackGPIO *test = new BlackGPIO(GPIO_60, output);
```

```
BlackGPIO test(GPIO_30, input);
              or
BlackGPIO test(GPIO_60, output);
```



Pin has to be output type for set a new GPIO pin value. Setting GPIO pin's direction process performs when initializing BlackGPIO class. If pin type is arranged to input type at this stage, user can't change this pin's value. setValue() function are wrote in the library for setting pin value. This function takes a "*pin_value*" type variable. This variable value can be only "*low*" or "*high*". "low" value sets pin to logical 0 state and "high" value sets pin to logical 1 state. An example below is about this function and usage of "`pin_value`" variable.

```
test->setValue(high);
```

3. Functions That Interact With End Users

a. ADC Management Functions

i. BlackADC::getValue() : string

This function gives analog input value. It reads specified file from path, where defined at BlackADC::ainPath variable. This file holds analog input voltage at milivolt level. It returns string type analog input value. If file opening process fails, it returns error message.

```
cout << "AIN value = " << test->getValue() << " mV";
```

ii. BlackADC::getNumericValue() : int

This function gives analog input value. It reads specified file path, where defined at BlackADC::ainPath variable. This file holds analog input voltage at milivolt level. It returns integer type analog input value. If opening file process fails, it returns -1.

```
int x = test->getNumericValue() + 100;
cout << "AIN value + 0,1V = " << x << " mV";
```

iii. BlackADC::getParsedValue(digit_after_point) : float

This function gives converted analog input value. It reads value by using getNumericValue() function and parses this value according to entered parameter. Read value becomes to volt level after parsing process. It takes a ***digit_after_point*** type variable. This variable shows parsing level. It returns float type value of parsed analog input.

```
float x = test->getParsedValue(dap3);
cout << "AIN(at a,bcd format)= " << x << " Volt";
```

iv. BlackADC::fail() : bool

This function uses for general debugging. It returns true if any error occur, else false.

```
if ( test->fail() )
{
    cout << "Error: General error occurred.";
}
```

v. BlackADC::fail(BlackADC::flags) : bool

This function uses for specific debugging. You can use this, after use BlackADC member functions in your code. It takes a ***flags*** type variable and this variable uses for find out status of an error which you want. It returns value of selected error. (true or false)

```
if ( test->fail(ReadErr) )
{
    cout << "Error: Analog input, file read error.";
}
```

b. PWM Management Functions

i. BlackPWM::getValue() : string

This function gives percentage value of duty cycle. It calls getNumericPeriodValue() and getNumericDutyValue() functions for find out the period and duty value of pwm. After do that it calculates percentage value by using these values. It returns string type percentage value of duty cycle.

```
cout << “PWM percentage value = “ << test->getValue();
```

ii. BlackPWM::getPeriodValue () : string

This function gives period value of pwm signal. It reads specified file path, where defined at BlackPWM::periodPath variable. This file holds pwm period value at nanosecond (ns) level. It returns string type, nanosecond (ns) level period value. If opening file process fails, it returns error message.

```
cout << “PWM period value = “ << test->getPeriodValue();
```

iii. BlackPWM::getDutyValue (): string

This function gives duty value of pwm signal. It reads specified file path, where defined at BlackPWM::dutyPath variable. This file holds pwm duty value at nanosecond (ns) level. It returns string type, nanosecond (ns) level duty value. If opening file process fails, it returns error message.

```
cout << “PWM duty value = “ << test->getDutyValue();
```

iv. BlackPWM::getRunValue (): string

This function gives run value of pwm signal. It reads specified file path, where defined at BlackPWM::runPath variable. This file holds pwm run value. It can be only 1 or 0. It returns string type, run value. If opening file process fails, it returns error message.

```
cout << "PWM run value = " << test->getRunValue();
```

v. BlackPWM::getPolarityValue (): string

This function gives polarity value of pwm signal. It reads specified file path, where defined at BlackPWM::polarityPath variable. This file holds pwm polarity value. It can be only 1 or 0. It returns string type, polarity value. If opening file process fails, it returns error message.

```
cout << "PWM polarity value = " << test->getPolarityValue();
```

vi. BlackPWM::getNumericValue (): float

This function gives percentage value of duty cycle as numerically. It calls getNumericPeriodValue() and getNumericDutyValue() functions, for find out the period and duty value of pwm signal. After do that it calculates percentage value by using these values. It returns float type percentage value of duty cycle.

```
float x = test->getNumericValue();
cout << "PWM percentage value = " << x;
```

vii. BlackPWM::getNumericPeriodValue () : uint32_t

This function gives period value of pwm signal as numerically. It calls getPeriodValue() function, for find out the pwm period value. After do that, this string type period value is converted to uint32_t type by using strtoimax() function. It returns uint32_t type, nanosecond (ns) level period value.

```
uint32_t x = test->getNumericPeriodValue();
cout << "PWM period value (ns) = " << x << endl;
cout << "PWM period value (us) = " << x/1000 << endl;
cout << "PWM period value (ms) = " << x/1000000;
```

viii. BlackPWM::getNumericDutyValue () : uint32_t

This function gives duty value of pwm signal as numerically. It calls getDutyValue() function, for find out the pwm duty value. After do that, this string type duty value is converted to uint32_t type by using strtoimax() function. It returns uint32_t type, nanosecond (ns) level duty value.

```
uint32_t x = test->getNumericDutyValue();
cout << "PWM duty value (ns) = " << x << endl;
cout << "PWM duty value (us) = " << x/1000 << endl;
cout << "PWM duty value (ms) = " << x/1000000;
```

ix. BlackPWM::setDutyPercent (float) : bool

This function sets percentage value of duty cycle. If entered parameter is in range (between 0 and 100), this function sets duty value without changing period value. For calculate new duty value, the current period value multiplies by entered percentage and this product subtracts from period value. After do that, this calculated value is recorded to duty file. It takes a ***float*** type variable and this variable uses for set new percentage value. It returns true if setting process is successful, else false.

```
float x = 41,0;
test->setDutyPercent(x);
```

x. BlackPWM::setPeriodTime (uint32_t) : bool

This function sets period value of pwm signal. If entered parameter is in range (between 0 or current duty value and 10^9), this function sets period value by recording entered value to period file. It takes a ***uint32_t*** type variable and this variable uses for set new period value. It returns true if setting process is successful, else false.

```
uint32_t x = 400000000;
test->setPeriodTime(x);
```

xi. BlackPWM::setSpaceRatioTime (uint32_t) : bool

This function sets space time value of pwm signal. If entered parameter is in range (between 0 and current period value), this function sets duty value by recording entered value to duty file. It takes a ***uint32_t*** type variable and this variable uses for set new space time. It returns true if setting process is successful, else false.

```
uint32_t x = 400000000;
test->setSpaceRatioTime(x);
```

xii. BlackPWM::setLoadRatioTime (uint32_t) : bool

This function sets load time value of pwm signal. If entered parameter is in range (between 0 and current period value), this function sets duty value. For calculate new duty value, the entered load time value subtracts from the current period value. After do that, this calculated value is recorded to duty file. It takes a ***uint32_t*** type variable and this variable uses for set load time. It returns true if setting process is successful, else false.

```
uint32_t x = 400000000;
test->setLoadRatioTime(x);
```

xiii. BlackPWM::setPolarity (pwm_polarity) : bool

This function sets polarity type of pwm signal. It sets polarity value to 1 or 0, by value of entered parameter. This polarity value is recorded to polarity file. It takes a ***pwm_polarity*** type variable and this variable uses for set polarity type. It returns true if setting process is successful, else false.

```
test->setPolarity(reverse);
```

xiv. BlackPWM::setRunState (pwm_value) : bool

This function sets run state of pwm signal. It sets run value to 1 or 0, by value of entered parameter. This run value is recorded to run file. It takes a ***pwm_value*** type variable and this variable uses for set run state. It returns true if setting process is successful, else false.

```
test->setRunState(stop);
```

xv. BlackPWM::toggleRunState () : void

This function toggles run state of pwm signal. It sets run value to 1 or 0, by taking existing value into consideration. This new run value is recorded to run file.

```
test-> toggleRunState();
```

xvi. BlackPWM::tooglePolarity () : void

This function toggles polarity type of pwm signal. It sets polarity value to 1 or 0, by taking existing value into consideration. This new polarity value is recorded to polarity file.

```
test-> tooglePolarity();
```

xvii. BlackPWM::isRunning () : bool

This function checks run state of pwm signal. It calls getRunValue() function and evaluates return value of this function. It returns false if run value equals to 0, else true.

```
if ( test->isRunning() )
{
    cout << "Pwm signal is running now.";
}
```

xviii. BlackPWM::isPolarityStraight () : bool

This function checks polarity state of pwm signal. It calls getPolarityValue() function and evaluates return value of this function. It returns true if polarity value equals to 0, else false.

```
if ( test->isPolarityStraight() )
{
    cout << "Pwm signal has straight polarity.";
}
```

xix. BlackPWM::isPolarityReverse () : bool

This function checks polarity state of pwm signal. It calls getPolarityValue() function and evaluates return value of this function. It returns false if polarity value equals to 0, else true.

```
if ( test->isPolarityReverse() )
{
    cout << "Pwm signal has reverse polarity.";
}
```

xx. BlackPWM::fail () : bool

This function uses for general debugging. It returns true if any error occur, else false.

```
        if ( test->fail() )
        {
            cout << "Error: General error occurred.";
        }
```

xxi. BlackPWM::fail (BlackPWM::flags) : bool

This function uses for specific debugging. You can use this, after use BlackPWM member functions in your code. It takes a ***flags*** type variable and this variable uses for find out status of an error which you want. It returns value of selected error. (True or false)

```
        if ( test->fail(outOfRangeErr) )
        {
            cout << "Error: Pwm, entered value is out of range.";
        }
```

c. GPIO Management Functions

i. BlackGPIO::getValue() : string

This function gives GPIO pin value. It checks pin status by calling isReady() function. If pin is ready, it reads specified file from path, where defined at BlackGPIO::valuePath variable. This file holds gpio pin value. It returns string type gpio pin value. If file opening process fails or pin isn't ready, it returns error message.

```
cout << "GPIO pin value = " << test->getValue();
```

ii. BlackGPIO::setValue (gpio_value) : bool

This function sets GPIO pin value. If pin type is output and pin is ready, it sets pin value to 1 or 0, by value of entered parameter. This value is recorded to value file. It takes a ***gpio_value*** type variable and this variable uses for set value of pin. It returns true if setting process is successful, else false.

```
test->setValue(low);
```

iii. BlackGPIO::isHigh () : bool

This function checks value of GPIO pin. It calls getValue() function and evaluates return value of this function. It returns false if pin value equals to 0, else true.

```
if ( test->isHigh() )
{
    cout << "GPIO value is 1.";
}
```

iv. BlackGPIO::toggleValue () : void

This function toggles value of gpio pin. If pin type is output, it sets pin value to 1 or 0, by taking existing value into consideration. This new pin value is recorded to value file.

```
test-> toggleValue();
```

v. BlackGPIO::fail () : bool

This function uses for general debugging. It returns true if any error occur, else false.

```
if ( test->fail() )
{
    cout << "Error: General error occurred.";
}
```

vi. BlackGPIO::fail (BlackGPIO::flags) : bool

This function uses for specific debugging. You can use this, after use BlackGPIO member functions in your code. It takes a ***flags*** type variable and this variable uses for find out status of an error which you want. It returns value of selected error. (True or false)

```
if ( test->fail(exportErr) )
{
    cout << "Error: Pin didn't export.";
}
```

4. Debugging and Error Flags

Error flags are controlled after called member functions. The objectives of flags are problem tracking. If problems occur when functions are running, the flags record these problems. Error types vary by function's task. For this reason you should know which flag must be controlled after which function ran. For instance, if you check error that records read function failures, after calling function that executes writing process, this control result will be incorrect.

Using flags which belong to errorCore or errorCore{ADC, PWM, GPIO} data structures, are not necessary for end user. These flags are made for developers who want to write code on cores of library.

a.ADC Error Flags and Usage

Flag Name	Description	Inherited Data Structure	Place and Reason Of Use
cpmgrErr	Error of capemgr name finding	errorCore > errorCoreADC	Anywhere in your code, before using capemgr name
ocpErr	Error of ocp name finding	errorCore > errorCoreADC	Anywhere in your code, before using ocp name
dtErr	Error of device tree loading	errorCoreADC	For checking device tree loading process, after initialization of BlackADC class
helperErr	Error of helper name finding	errorCoreADC	For checking ADC device driver, after initialization of BlackADC class and device tree loading process
readErr	Error of file reading	----	For checking accuracy, after reading analog value process[1]

[1] You can check this flag after these functions.

- **getValue()**
- **getNumericValue()**
- **getParsedValue()**

b. PWM Error Flags and Usage

Flag Name	Description	Inherited Data Structure	Place and Reason Of Use
cpmgrErr	Error of capemgr name finding	errorCore > errorCorePWM	Anywhere in your code, before using capemgr name
ocpErr	Error of ocp name finding	errorCore > errorCorePWM	Anywhere in your code, before using ocp name
dtErr	Error of device tree loading	errorCorePWM	For checking device tree loading process, after initialization of BlackPWM class
dtSubSystemErr	Error of device tree loading	errorCorePWM	For checking second device tree loading process, after initialization of BlackPWM class
pwmTestErr	Error of pwm_test name finding	errorCorePWM	For checking PWM device driver, after initialization of BlackPWM class and device tree loading processes
initializeErr	Error of initialization	errorCorePWM	If you try to initialize pwm one more time, this error will occur. Under normal conditions you can initialize pwm only once from BlackPWM class.
periodFileErr	Error of period file opening	----	After called a function which opens period file[1]
dutyFileErr	Error of duty file opening	----	After called a function which opens duty file[2]

[1] You can check this flag after these functions.
- **getValue()**
- **getNumericValue()**
- **getPeriodValue()**
- **getNumericPeriodValue()**
- **setPeriodTime()**
- **setSpaceRatioTime()**
- **setLoadRatioTime()**

[2] You can check this flag after these functions.
- **getValue()**
- **getNumericValue()**
- **getDutyValue()**
- **getNumericDutyValue()**
- **setDutyPercent()**
- **setPeriodTime()**
- **setSpaceRatioTime()**
- **setLoadRatioTime()**

runFileErr	Error of run file opening	----	After called a function which opens run file[1]
polarityFileErr	Error of polarity file opening	----	After called a function which opens polarity file[2]
outOfRangeErr	Error of entered value range	----	After called a function which takes numeric variable and uses this variable for make an adjustment[3]

[1] You can check this flag after these functions.

- **getRunValue()**
- **isRunning()**
- **setRunState()**
- **toggleRunState()**

[2] You can check this flag after these functions.

- **getPolarityValue()**
- **isPolarityStraight()**
- **isPolarityReverse()**
- **setPolarity()**
- **tooglePolarity()**

[3] You can check this flag after these functions.

- **setDutyPercent()**
- **setPeriodTime()**
- **setSpaceRatioTime()**
- **setLoadRatioTime()**

c. GPIO Error Flags and Usage

Flag Name	Description	Inherited Data Structure	Place and Reason Of Use
exportFileErr	Error of export file opening	errorCoreGPIO	After pin exporting process
directionFileErr	Error of direction file opening	errorCoreGPIO	After pin direction setting process
initializeErr	Error of initialization	errorCoreGPIO	If you try to initialize GPIO one more time, this error will occur. Under normal conditions you can initialize pwm only once from BlackGPIO class.
exportErr	Error of exporting pin	----	If reading or writing error occurs, you can check this error. In addition to these errors, if exportErr occurred also, this situation shows pin didn't export or unexported by manually after class initialization.[1]
directionErr	Error of pin direction	----	If reading or writing error occurs, you can check this error. In addition to these errors, if directionErr occurred also, this situation shows pin direction changed by manually after class initialization.[2]
readErr	Error of value file reading	----	For checking accuracy, after reading process[3]
writeErr	Error of value file writing	----	For checking accuracy, after writing process[4]
forcingErr	Error of pin value forcing	----	If you try to write value to input type pin, this error will occur. You can check this error after writing process[5]

[1] You can check this flag after these functions.
- **getValue()**
- **isHigh()**
- **toggleValue()**
- **setValue()**

[2] You can check this flag after these functions.
- **getValue()**
- **isHigh()**
- **toggleValue()**
- **setValue()**

[3] You can check this flag after these functions.
- **getValue()**
- **isHigh()**
- **toggleValue()**

[4] You can check this flag after these functions.
- **setValue()**

[5] You can check this flag after these functions.
- **setValue()**

Some of GPIO flags can be set at the same time. When these times, you must evaluate these flags together for understand the actual reason of error. Error tracking algorithm set flags properly, by taking occurred problems into consideration. For this reason, some of these combinations have special meanings. This situation is valid for GPIO flags only. “GpioFlagsUsageTable” document which exists in document files directory, is prepared for these combinations usage.

7. BIBLIOGRAPHY

[1] "Device Tree," [Online]. Available: http://www.devicetree.org/Main_Page.

[2] "C++11," Wikipedia, 2012. [Online]. Available: http://en.wikipedia.org/wiki/C%2B%2B11.

[3] "Compiling C++11 with g++ - Stack Overflow," Stack Overflow, 2012. [Online]. Available: http://stackoverflow.com/questions/10363646/compiling-c11-with-g.

[4] "Eclipse CDT C++11/C++0x support - Stack Overflow," Stack Overflow, 2012. [Online]. Available: http://stackoverflow.com/questions/9131763/eclipse-cdt-c11-c0x-support.