BlackLib - User Manual

Beaglebone Black C++ Library

Author: Yiğit Yüce

22.04.2014

blacklib.yigityuce.com

v1.0

CONTENTS

1. ABSTRACT

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

2. KEYWORDS

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

3. INTRODUCTION

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

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 is wrote with C++0x standard, can’t use without add C++0x flag to compiler. For this reason when compiling stage you must add to this flag. ”Compiling C++11 with g++” is explained at this link [3] or if you use eclipse for development, “Eclipse CDT C++11/C++0x support” is explained at this link. [4] You have to do these steps to 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.

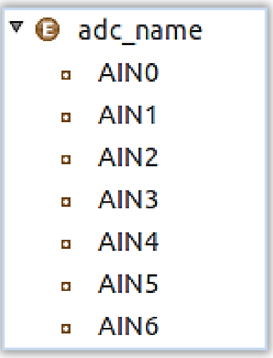
→ Add this code to top of your code.

```
#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

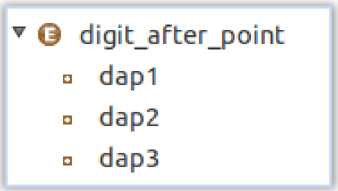
adc_name definition is sub option which you can use when you want read data from Beaglebone Black's analog inputs. You can select which "AINx" input will use for read analog value, by dint of this sub option.

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

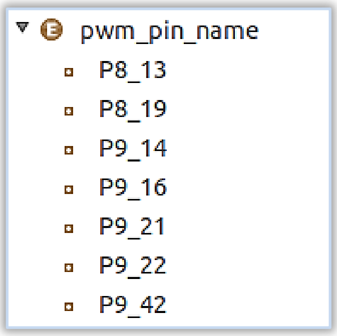
When reading ADC, value is read at mV (millivolt) level. So generally you must scale this millivolt level value. For this reason, "*getParsedValue()*" function is wrote in the library. This function takes one parameter and it can be "*digit_after_point*" type variable. You can define how many digits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

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

b. PWM Definitions

“*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.

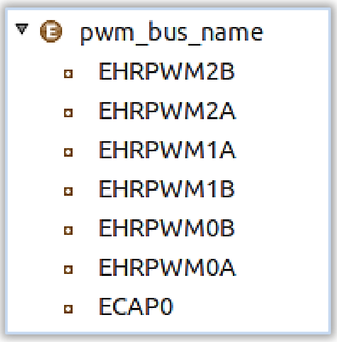
Beaglebone Black has a lot PWM outputs. Available of these outputs defined on the side of page. These PWM outputs located on P8 and P9 ports.



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

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 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.

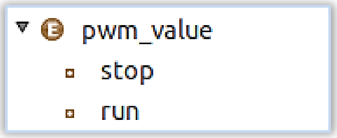


pwm_bus_name
- EHRPWM2B
- EHRPWM2A
- EHRPWM1A
- EHRPWM1B
- EHRPWM0B
- EHRPWM0A
- ECAP0

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 pwm signal must start generating to get from PWM outputs. “*setRunState()*” function is wrote to start or stop generating signal. This function takes one parameter and it can be “*pwm_value*” type variable. This variable’s value can be “*stop*” or “*run*”. “*run*” starts generating signal and “stop” terminates signal. An example shown in below is about this function and usage of “*pwm_value*” variable.

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

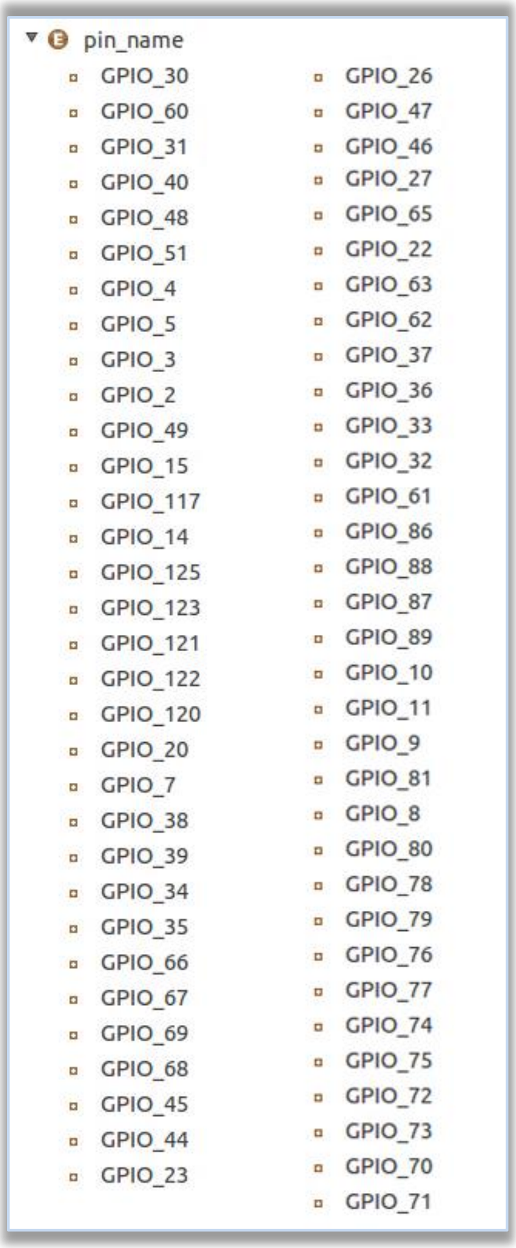
c. GPIO Definitions

"pin_name" is sub option that uses to define which GPIO pin will select. 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 port's names 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 using some pins.

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

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

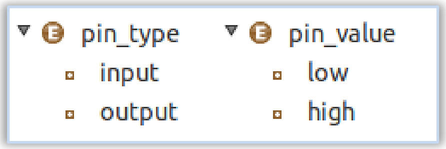


pin_name

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

Gpio pin has to be output type for setting a new pin value. Setting GPIO pin’s direction is performed when initializing BlackGPIO class. If pin type is arranged to input type at this stage, user can’t change this pin’s value later. setValue() function is wrote for setting pin value. This function takes a “*pin_value*” type variable. This variable’s value can be only “*low*” or “*high*”. “low” sets pin to logical 0 state and “high” value sets pin to logical 1 state. An example shown in 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 returns 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 fails, it returns error message.

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

ii. BlackADC::getNumericValue() : int

This function returns 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 integer type analog input value. If file opening 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 returns converted analog input value. It gets value by using getNumericValue() function and converts this value to volt level, according to input parameter. It takes a ***digit_after_point*** type variable. This variable shows conversion level. It returns converted analog input value as float.

```
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 call BlackADC member functions in your code. It takes a ***flags*** type variable and it is used for finding out status of selected error. It returns value of selected error.

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

b. PWM Management Functions

i. BlackPWM::getValue() : string

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

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

ii. BlackPWM::getPeriodValue () : string

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

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

iii. BlackPWM::getDutyValue (): string

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

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

iv. BlackPWM::getRunValue (): string

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

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

v. BlackPWM::getPolarityValue (): string

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

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

vi. BlackPWM::getNumericValue (): float

This function returns numeric percentage value of duty cycle. It calls getNumericPeriodValue() and getNumericDutyValue() functions, for finding out the period and duty values of pwm. After do that it calculates percentage value. 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 returns numeric period value of pwm signal. It calls getPeriodValue() function, for finding out the period value. After do that, this period value is converted to uint32_t type by using strtoimax() function. It returns uint32_t type 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 returns numeric duty value of pwm signal. It calls getDutyValue() function, for finding out the duty value. After do that, this duty value is converted to uint32_t type by using strtoimax() function. It returns uint32_t type 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 input parameter is in range (from 0 to 100) this function changes duty value without changing period value. For calculating new duty value, the current period multiplies by entered percentage and this product subtracts from period. After do that this calculated value is saved to duty file. It takes a ***float*** type variable and this variable uses for setting new percentage value. It returns true if setting new value 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 input parameter is in range (from 0 to 10^9 or greater than current duty value) this function changes period value by saving entered value to period file. It takes a ***uint32_t*** type variable and this variable uses for setting new period value. It returns true if setting new value 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 input parameter is in range (from 0 to current period value) this function changes duty value by saving entered value to duty file. It takes a ***uint32_t*** type variable and this variable uses for setting new space time. It returns true if setting new value 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 input parameter is in range (from 0 to current period value) this function changes duty value. For calculating new duty value, the entered value subtracts from the current period value. After do that this calculated value is saved to duty file. It takes a ***uint32_t*** type variable and this variable uses for setting load time. It returns true if setting new value is successful, else false.

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

xiii. BlackPWM::setPolarity (pwm_polarity) : bool

This function sets polarity of pwm signal. It sets polarity value to 1 or 0. The input parameter is converted to 1 or 0 and this value is saved to polarity file. It takes a ***pwm_polarity*** type variable and this variable uses for setting polarity type. It returns true if setting new polarity 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. The input parameter is converted to 1 or 0 and this value is saved to run file. It takes a ***pwm_value*** type variable and this variable uses for setting run state. It returns true if setting new run state 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 value of current state. This new value is saved 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 value of current polarity. This new value is saved 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. It returns false if return 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. It returns true if return 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. It returns false if return 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 call BlackPWM member functions in your code. It takes a ***flags*** type variable and this variable uses for finding out status of selected error. It returns value of selected error.

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

c. GPIO Management Functions

i. BlackGPIO::getValue() : string

This function returns value of GPIO pin. It checks pin ready state 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 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 value of GPIO pin. If pin type is output and pin is ready, it sets pin value to 1 or 0. The input parameter is converted to 1 or 0 and this value is saved to value file. It takes a ***gpio_value*** type variable and this variable uses for setting value of pin. It returns true if setting new value 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. It returns false if return 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, this function sets pin value to 1 or 0, by value of current state. This new value is saved 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 call BlackGPIO member functions in your code. It takes a ***flags*** type variable and this variable uses for finding out status of selected error. It returns value of selected error.

```
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 will be controlled after which function ran. For instance, if you check read error flag, 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	Defined Structure	Place and Reason Of Use
cpmgrErr	capemgr name finding error	errorCore	Anywhere in your code, before using capemgr name
ocpErr	ocp name finding error	errorCore	Anywhere in your code, before using ocp name
dtErr	device tree loading error	errorCoreADC	For checking device tree loading process, after initialization of BlackADC class
helperErr	helper name finding error	errorCoreADC	For checking ADC device driver, after initialization of BlackADC class and device tree loading process
readErr	file reading error	errorADC	For checking accuracy, after reading analog value process[1]

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

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

b. PWM Error Flags and Usage

Flag Name	Description	Defined Structure	Place and Reason Of Use
cpmgrErr	capemgr name finding error	errorCore	Anywhere in your code, before using capemgr name
ocpErr	ocp name finding error	errorCore	Anywhere in your code, before using ocp name
dtErr	device tree loading error	errorCorePWM	For checking device tree loading process, after initialization of BlackPWM class
dtSubSystemErr	device tree loading error	errorCorePWM	For checking second device tree loading process, after initialization of BlackPWM class
pwmTestErr	pwm_test name finding error	errorCorePWM	For checking PWM device driver, after initialization of BlackPWM class and device tree loading processes
initializeErr	Initialization error	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	period file opening error	errorPWM	After called a function which opens period file[1]
dutyFileErr	duty file opening error	errorPWM	After called a function which opens duty file[2]

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

- **getValue()**
- **getNumericValue()**
- **getPeriodValue()**
- **getNumericPeriodValue()**
- **setPeriodTime()**
- **setSpaceRatioTime()**
- **setLoadRatioTime()**

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

- **getValue()**
- **getNumericValue()**
- **getDutyValue()**
- **getNumericDutyValue()**
- **setDutyPercent()**
- **setPeriodTime()**
- **setSpaceRatioTime()**
- **setLoadRatioTime()**

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

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

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

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

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

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

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

c. GPIO Error Flags and Usage

Flag Name	Description	Defined Structure	Place and Reason Of Use
exportFileErr	export file opening error	errorCoreGPIO	After pin exporting process
directionFileErr	direction file opening error	errorCoreGPIO	After pin direction setting process
initializeErr	Initialization error	errorCoreGPIO	If you try to initialize GPIO one more time, this error will occur. Under normal conditions you can initialize gpio only once from BlackGPIO class.
exportErr	exporting pin error	errorGPIO	If read or write errors occur, you can check this error. In addition to these errors, if exportErr occurred also, this situation shows pin is not exported or unexported by manually after initialization.[1]
directionErr	pin direction error	errorGPIO	If read or write errors occur, you can check this error. In addition to these errors, if directionErr occurred also, this situation shows pin direction changed by manually after initialization.[2]
readErr	value file reading error	errorGPIO	For checking accuracy, after reading process[3]
writeErr	value file writing error	errorGPIO	For checking accuracy, after writing process[4]
forcingErr	pin value forcing error	errorGPIO	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 call these functions.
- **getValue()**
- **isHigh()**
- **toggleValue()**
- **setValue()**

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

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

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

[5] You can check this flag after call 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 sets 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.

Please visit our website for detailed technical documents. **blacklib.yigityuce.com**

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.