1300 Henley Court
Pullman, WA 99163
509.334.6306
www.digilentinc.com

Basys MX3 LibPack User guide

1 Overview

BasysMX3 incorporates a wide range of modules. Digilent provides a set of libraries, allowing user easy access to each module's functionality.

Basically, these libraries hide the hardware implementation details; they wrap over the lower level functions that access the registers, allowing the user to call the needed functionality in an easy and intuitive manner.
For more details about each module, please read the BasysMX3 Reference Manual and the Schematics, available on BasysMX3 product page on Digilent website.
The Libraries usage chapter provides some usage information, and the following chapters detail each library.

2 Libraries usage

Normally each library has a .c and .h file. In order to use this library:

- include in your project the .c and .h files corresponding to the module you want to use (for example led.c and led.h). This can be easily done using Add Existing Item options, having copy option checked (right mouse click on the project source folders and project header files folders).
- In your code, include the header of the module
  ```
  #include "led.h"
  ```
- In your code, call the needed functions, for example:
  ```
  LED_Init();
  LED_SetValue(4, 1); //turn on LED4
  ```

Apart from the library source / header file, user should include in the project the config.h file. This file contains the hardware implementation details.

Library dependencies

The following libraries depend on each other in the following manners:

- some libraries call functions from other libraries:
  - AIC (analog input control) and MIC (Microphone) call functions from ADC (Analog Digital converter)
  - AUDIO (audio module) calls functions from MIC (Microphone) in order to implement the audio recording mode.
- AUDIO and MOT libraries share the Timer3.

Read each library chapter for how to address each of these situations.

3 LED

3.1 Overview

The LED library groups the functions that access onboard LEDs (labeled “LD0” – “LD7”).

The library also provides a set of fast access SetValue function macros, one for each LED, defined in led.h.

Library files

The library is implemented in these files: led.c, led.h. Include them in the project.

Include the library header file whenever you want to use the library functions:

```
#include "led.h"
```

LED numbering

The numbering is consistent with the LEDs labeling:

Number	LED
0	LD0
1	LD1
...	
7	LD7

LED values

The library deals with logic value assigned to a LED, having this meaning:

Value	Meaning
0	LED is off (not lighting)
1	LED is on (lighting)

3.2 Library functions

3.2.1 LED_Init

Synopsis

void LED_Init();

Description

This function initializes the hardware involved in the LED module. The pins corresponding to LEDs are initialized as digital outputs. All the LEDs are turned off.

Example

```
#include "led.h"
```

```
LED_Init();
```

3.2.2 LED_SetValue

Synopsis

void LED_SetValue(unsigned char bNo, unsigned char bVal);

Parameters

- bNo – the number of LED whose value will be set. The value must be between 0 and 7.
- bVal – the value to be assigned to the specific LED:

bVal parameter value	Action
0	LD<bNo> is turned off
Other than 0	LD<bNo> is turned on

Description

This function sets the value to the LED specified by bNo to the value specified by bVal.

If the value provided for bNo is not between 0 and 7, the function does nothing.

Example

```
#include "led.h"
LED_Init();
LED_SetValue(3, 1);      // turns on the LD3
```

3.2.3 LED_ToggleValue

Synopsis

void LED_ToggleValue(unsigned char bNo);

Parameters

- bNo – the number of LED whose value will be toggled. The value must be between 0 and 7.

Description

This function toggles the value of the LED specified by bNo. If the LED was off, it is turned on. If the LED was on, it is turned off.

If the value provided for bNo is not between 0 and 7, the function does nothing.

Example

```
#include "led.h"
LED_Init();
LED_ToggleValue(3);      // toggles the value  of LD3
```

3.2.4 LED_SetGroupValue

Synopsis

void LED_SetGroupValue(unsigned char bVal);

Parameters

- bVal – the 8 bit value B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0 that specifies the values for the 8 LEDs.

B_i value	Action
0	LD<i> is turned off
1	LD<i> is turned on

Description

This function sets the value for all 8 LEDs, according to the value provided in bVal. Each bit from bVal corresponds to a LED: Bit 0 (LSB) corresponds to LD0, bit 7 (MSB) corresponds to LD7.

Example

```
#include "led.h"
LED_Init();
LED_SetGroupValue(0xAA);      // assigns alternate values to the
LEDs
```

3.2.5 LEDxSetValue fast access function macros

Synopsis

LEDS_LedxSetValue(val)

Parameters

- bVal – the value to be assigned to the specific LED:

bVal parameter value	Action
0	LD<x> is turned off
1	LD<x> is turned on

Description

For each LED there is a macro function that provides the fastest way to set its value. The bVal parameter specifies the value for the LED (0 for off, 1 for on).

Macro function	Action
LEDS_Led0SetValue	Set the value for LED0
LEDS_Led1SetValue	Set the value for LED1
LEDS_Led2SetValue	Set the value for LED2
LEDS_Led3SetValue	Set the value for LED3
LEDS_Led4SetValue	Set the value for LED4
LEDS_Led5SetValue	Set the value for LED5
LEDS_Led6SetValue	Set the value for LED6
LEDS_Led7SetValue	Set the value for LED7

These macros are defined in led.h.

Example

```
#include "led.h"
LEDS_Led6SetValue(1); // turns on the LD6
```

3.2.6 LED_ConfigurePins

Synopsis

void LED_ConfigurePins()

Description

This function configures the IO pins involved in the LED module as digital output pins.
The function uses pin related definitions from config.h file.
This is a low-level function called by LED_Init(), so user should avoid calling it directly.

Example

```
#include “led.h”
LED_ConfigurePins();
```

4 SWT

4.1 Overview

The SWT library groups the functions that access onboard Switches (labeled “SW0” – “SW7”).

Library files

The library is implemented in these files: swt.c, swt.h. Include them in the project.

Include the library header file whenever you want to use the library functions:

```
#include "swt.h"
```

Switches numbering

The numbering is consistent with the switches labeling:

Number	LED
0	SW0
1	SW1
...	
7	SW7

Switch values

The library deals with logic value assigned to a switch, having this meaning:

Value	Meaning
0	switch is off (position backwards, towards the edge of the board)
1	switch is on (position forward, towards LEDs)

4.2 Library functions

4.2.1 SWT_Init

Synopsis

void SWT_Init();

Description

This function initializes the hardware involved in the SWT module: the pins corresponding to switches are initialized as digital inputs.

Example

```
#include "swt.h"
SWT_Init();
```

4.2.2 SWT_GetValue

Synopsis

unsigned char SWT_GetValue(unsigned char bNo)

Parameters

- bNo – the number of switch whose value will be get. The value must be between 0 and 7.

Return

- unsigned char – the value corresponding to the specified switch:

return value	When
0	SW<bNo> is turned off
1	SW<bNo> is turned on

– 0xFF if bNo is not within 0 - 7.

Description

This function gets the value of the switch specified by bNo (0 or 1). If the value provided for bNo is not between 0 and 7, 0xFF is returned.

Example

```
#include “swt.h”
SWT_Init();
unsigned char val = SWT_GetValue(3);    // gets the value of SW3
```

4.2.3 SWT_GetGroupValue

Synopsis
unsigned char SWT_GetGroupValue()

Return

- unsigned char – the 8 bit value B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0 where each bit B_i corresponds to SW<i>:

B_i has the value	when
0	SW<i> is turned off
1	SW<i> is turned on

Description

This function gets the value of the all 8 switches as a single value on 8 bits.

Each bit from returned value corresponds to a switch: Bit 0 (LSB) corresponds to SW0, bit 7 (MSB) corresponds to SW7.

Example

```
#include “swt.h”
SWT_Init();
unsigned char val = SWT_GetGroupValue();     // gets the value
of all switches
```

4.2.4 SWT_ConfigurePins

Synopsis
void SWT_ConfigurePins()

Description
This function configures the IO pins involved in the SWT module as digital input pins.
The function uses pin related definitions from config.h file.
This is a low-level function called by SWT_Init(), so user should avoid calling it directly.

Example

```
#include “swt.h”
SWT_ConfigurePins();
```

5 BTN

5.1 Overview

The BTN library groups the functions that access onboard Buttons (labeled “BTNU”, “BTNL”, “BTNC”, “BTNR”, “BTND”).

The library just performs basic digital input. No debouncing functionality is provided.

Library files

The library is implemented in these files: btn.c, btn.h. Include them in the project.

Include the library header file whenever you want to use the library functions:

```
#include "btn.h"
```

Buttons designation

Each button can be identified using a number or a letter (uppercase or lowercase), see table below:

Number	Letter	Button
0	'U', 'u'	BTNU
1	'L', 'l'	BTNL
2	'C', 'c'	BTNC
3	'R', 'r'	BTNR
4	'D', 'd'	BTND

Button values

The library deals with logic value assigned to a button, having this meaning:

Value	Meaning
0	button is not pressed
1	button is pressed

Shared pins / functionality

As you can see from the schematic, the S0_PWM signals shares the same PIC32 pin with BTNR and the S1_PWM signals shares the same PIC32 pin with BTND (see SRV chapter). So, servos and buttons resources should be used exclusively.

5.2 Library functions

5.2.1 BTN_Init

Synopsis

void BTN_Init();

Description

This function initializes the hardware involved in the BTN module. The pins corresponding to buttons are initialized as digital inputs.

Example

```
#include "btn.h"
BTN_Init();
```

5.2.2 BTN_GetValue

Synopsis
unsigned char BTN_GetValue(unsigned char btn)

Parameters

- btn – the identification of button whose value will be read. The value can be the number or the letter associated to a button. It must be between 0 and 4, or one of ‘U’, ‘u’, ‘L’, ‘l’, ‘C’, ‘c’, ‘R’, ‘r’, ‘D’, ‘d’.

Number	Letter	Button
0	‘U’, ‘u’	BTNU
1	‘L’, ‘l’	BTNL
2	‘C’, ‘c’	BTNC
3	‘R’, ‘r’	BTNR
4	‘D’, ‘d’	BTND

Return

- unsigned char – the value corresponding to the specified button:

return value	when
0	Corresponding button is not pressed
1	Corresponding button is pressed

– 0xFF if btn is not between 0 and 4, or one of 'U', 'u', 'L', 'l', 'C', 'c', 'R', 'r', 'D', 'd'

Description

This function gets the value of the BTN specified by btn. If the value provided for btn is not between 0 and 4, or one of ‘U’, ‘u’, ‘L’, ‘l’, ‘C’, ‘c’, ‘R’, ‘r’, ‘D’, ‘d’, then the function returns 0xFF.

Example

```
#include “btn.h”
BTN_Init();
unsigned char val = BTN_GetValue(‘C’);  // gets the value of
BTNC
```

5.2.3 BTN_GetGroupValue

Synopsis
unsigned char BTN_GetGroupValue()

Return

- unsigned char – the 8 bit value 0 0 0 B_4 B_3 B_2 B_1 B_0 where bit B_i corresponds to BTN<i>:

Bit	has the value	When	
B_0	0	BTNU	is not pressed
	1		is pressed
B_1	0	BTNL	is not pressed
	1		is pressed
B_2	0	BTNC	is not pressed

	1		is pressed
B_3	0	BTNR	is not pressed
	1		is pressed
B_4	0	BTND	is not pressed
	1		is pressed

Description

This function gets the value of the all 5 buttons as a single value on 8 bits.

The 5 LSB bits from returned value correspond to a button: bit 0 corresponds to BTNU, bit 1 corresponds to BTNL, bit 2 corresponds to BTNC, bit 3 corresponds to BTNR, bit 4 corresponds to BTND (see return value description).

Example

```
#include “btn.h”
BTN_Init();
unsigned char val = BTN_GetGroupValue();     // gets the value
of all buttons
```

5.2.4 BTN _ConfigurePins

Synopsis

void BTN_ConfigurePins()

Description

This function configures the IO pins involved in the BTN module as digital input pins.
The function uses pin related definitions from config.h file.
This is a low-level function called by BTN_Init(), so user should avoid calling it directly.

Example

```
#include “btn.h”
BTN_ConfigurePins();
```

6 RGBLED

6.1 Overview

The RGBLED library groups the functions that access the RGBLED module.

Each color can be identified using the 3 basic components: red, green and blue.

The BasysMX3 provides 3 digital signals to control the RGBLED module, one for each basic color R, G, B: LED8_R, LED8_G and LED8_B.

Using either 0 or 1 values for these signals will only give the user a limited number of colors (2 colors for each component), so most of the time this is not enough in applications using the RGB feature. The solution is to send a sequence of 1 and 0 values on these digital lines, switched rapidly with a frequency

higher than the human perception. The “duty factor” will finally determine the color, as the human eye will “integrate” the discrete illumination values into the final color sensation. The most common ways to implement this are: PWM (pulse width modulation) and PDM (pulse density modulation).

The RGBLED library uses PDM, still provides the PWM implementation using commented code. The advantages of PDM approach are: it saves hardware resources to be used elsewhere (3 output compare modules), the implementation is rather simple and it provides a good example of PDM. The advantages of using PWM are: it uses hardware modules (3 output compare modules), thus decreasing the processor load, and it is well known to many users.

Implementation features (PDM method)

The PDM method implemented in the RGBLED library has the following features:

- The digital pins corresponding to each color (LED8_R, LED8_G and LED8_B) are configured as simple digital outputs.
- Timer5 is configured to generate an interrupt every approx. 300 µs.
- Each color component has a 8 bit resolution: a color is set by providing its components (R, G and B) as 8 bits values.
- In the Timer5 interrupt service routine, for each component color (R, G and B):
    - One 16 bits accumulators are used, one for each color.
    - the 8 bits component color value is added to the corresponding 16 bits accumulator.
    - the 9th bit of the accumulator is considered the carry bit. The resulted carry bits are assigned to the digital pins corresponding to each color (LED8_R, LED8_G and LED8_B).
    - carry bit is cleared in the accumulator.

RGBLED implemented using PWM

As mentioned above, the library source file (rgbled.c) contains as commented code the PWM implementation of the functionality needed for RGBLED module.

It has the following features:

- The digital pins corresponding to each color (LED8_R, LED8_G and LED8_B) are mapped to the output of OC3, OC4, OC5 modules of PIC32.
- The OC3, OC4, OC5 and Timer2 modules are configured to work together to generate PWM.
- Every time a color is set, the OCxRS register sets the appropriate PWM.

Library files

The library is implemented in these files: rgbled.c, rgbled.h. Include them in the project.

Include the library header file whenever you want to use the library functions:

```
#include "rgbled.h"
```

6.2 Library functions

6.2.1 RGBLED_Init

Synopsis

void RGBLED_Init();

Description

This function initializes the hardware involved in the RGBLED module: the pins corresponding to R, G and B colors are initialized as digital outputs and Timer 5 in configured.

Example

```
#include "rgbled.h"
RGBLED_Init();
```

6.2.2 RGBLED_SetValue

Synopsis

void RGBLED_SetValue(unsigned char bValR, unsigned char bValG, unsigned char bValB);

Parameters

- bValR – the value that corresponds to R component of the color.
- bValG – the value that corresponds to G component of the color.
- bValB – the value that corresponds to B component of the color.

Description

This function sets the color value by providing the values for the 3 components: R, G and B, as 3 separate 8 bits values.

Example

```
#include "rgbled.h"
RGBLED_Init();
RGBLED_SetValue(0xFF, 0x7F, 0); // set color R = 0xFF, G =
0x7FF, B = 0
```

RGBLED_SetValueGrouped

Synopsis

void RGBLED_SetValueGrouped(unsigned int uiValRGB);

Parameters

- uiValRGB – the value that groups in the 3 LSB bytes the color values in this pattern: xxxxxxxxRRRRRRRRGGGGGGGGBBBBBBBB.

Bits	Content
31-24	<don't care>
23-16	Byte corresponding to R color

15-8	Byte corresponding to G color
7-0 (LSB byte)	Byte corresponding to B color

Description

This function sets the color value by providing the values for the 3 components R, G and B as a 24 bits value, placed in the LSB 3 bytes of the uiValRGB like this: bits 23-16 correspond to color R, bits 15-8 correspond to color G, bits 7-0 (LSB byte) correspond to color B.

Example

```
#include "rgbled.h"
RGBLED_Init();
RGBLED_SetValueGrouped(0xFF7F00); // set color R = 0xFF, G =
0x7FF, B = 0
```

6.2.3 RGBLED_Close

Synopsis

void RGBLED_Close();

Description

This function can be called when RGBLED library is no longer needed: it stops the Timer5 and turns off the RGBLED.

Example

```
#include "rgbled.h"
RGBLED_Close();
```

6.2.4 RGBLED_Timer5Setup

Synopsis

void RGBLED_Timer5Setup ()

Description

This function configures the Timer1 to be used by RGBLED module.
The timer will generate interrupts every 300 us. The period constant is computed using TMR_TIME definition (located in this source file) and peripheral bus frequency definition (PB_FRQ, located in config.h).
This is a low-level function called by RGBLED_Init(), so user should avoid calling it directly.

Example

```
#include “ssd.h”
RGBLED_Timer5Setup (); // initialize Timer5
```

6.2.5 RGBLED _ConfigurePins

Synopsis
void RGBLED_ConfigurePins()
Description
This function configures the IO pins involved in the RGBLED module as digital output pins.
The function uses pin related definitions from config.h file.
This is a low-level function called by RGBLED_Init(), so user should avoid calling it directly.
Example

```
#include “rgbled.h”
RGBLED_ConfigurePins();
```

7 I2C

7.1 Overview

The I2C library groups the functions that access the I2C1 hardware interface of PIC32.

This library is used in the implementation of ACL library.

It also can be used for other I^2C devices connected to the I2C connector situated under the LCD.

The function is providing the basic I2C functions: initialization, read and write.

Library files

The library is implemented in these files: i2c.c, i2c.h. Include them in the project.

Include the library header file whenever you want to use the library functions:

```
#include "i2c.h"
```

7.2 Library functions

7.2.1 I2C_Init

Synopsis
void I2C_Init(unsigned int i2cFreq)
Parameters

- unsigned int i2cFreq - I2C clock frequency (Hz)
  (for example 400000 value sent as parameter corresponds to 400 kHz)

Description

This function configures the I2C1 hardware interface of PIC32, according to the provided frequency. In order to compute the baud rate value, it uses the peripheral bus frequency definition (PB_FRQ, located in config.h).

This is a low-level function called by Init(), so user should avoid calling it directly.

Example

```
#include “i2c.h”
I2C_Init(400000);
```

7.2.2 I2C_Write

Synopsis

unsigned char I2C_Write(unsigned char slaveAddress,
unsigned char* dataBuffer,
unsigned char bytesNumber,
unsigned char stopBit)

Parameters

- unsigned char slaveAddress - I2C address of the slave device.
- unsigned char* dataBuffer - Pointer to a buffer storing the bytes to be transmitted.
- unsigned char bytesNumber - Number of bytes to be transmitted.
- unsigned char stopBit - Stop condition control.

stopBit	Meaning
0	A stop condition will not be sent
1	A stop condition will be sent

Return

- unsigned char

Return Value	Meaning
0	Success
0xFF	the slave address was not acknowledged by the device error
0xFE	timeout error

Description

This function writes a number of bytes to the specified I2C slave. It returns the status of the operation: success or I2C errors (the slave address was not acknowledged by the device or timeout error).

This is a low-level function, so user should avoid calling it directly.

Example

```
#include “i2c.h”
unsigned char rgVals[2], bResult;
rgVals[0] = 0x2A;// register address
rgVals[1] = 1;      // register value
bResult = I2C_Write(0x1D, rgVals, 2, 1);
```

7.2.3 I2C_Read

Synopsis

unsigned char I2C_Read(unsigned char slaveAddress,
unsigned char* dataBuffer,
unsigned char bytesNumber)

Parameters

- unsigned char slaveAddress - I2C address of the slave device.
- unsigned char* dataBuffer - Pointer to a buffer where received bytes will be placed.
- unsigned char bytesNumber - Number of bytes to be read.

Return

- unsigned char

Return Value	Meaning
0	success
0xFF	the slave address was not acknowledged by the device error
0xFE	timeout error

Description

This function writes a number of bytes to the specified I2C slave. It returns the status of the operation: success or I2C errors (the slave address was not acknowledged by the device or timeout error).

This is a low-level function, so user should avoid calling it directly.

Example

```
#include “i2c.h”

unsigned char bResult;
I2C_Read(0x1D, &bResult, 1);
```

8 ACL

8.1 Overview

The ACL library groups the functions that access onboard accelerometer. It is a 3-Axis, 12-bit, Digital accelerometer, exposing an I²C digital interface.

The library implements I²C access to the onboard accelerator device and provides basic functions to configure the accelerometer and provide the accelerometer values (raw values and g values).

The Basys MX3 board also provides a digital pin (ACL_INT2) that can be used as an interrupt raised by the accelerometer. The library implements no functionality related to this pin, it just configures the pin as a digital input, so user should properly configure the accelerometer related to this interrupt.

The ACL library uses I2C library for the I²C related functionality.

Raw values and g Values

For each of the 3 axis, the value retrieved from the accelerator (over I^2C) is an 2's complement 12 bit value. It is considered as raw value, and is represented on 2 bytes.

The library provides conversion of the raw values to the values expressed in terms of g.

The conversion is done in the following manner:

- Compute the conversion constant: value in g corresponding for each count of the raw value. This value only changes when accelerator full scale range is changed.
  - the selected accelerator full scale range corresponds to a range of g values.
  - for example, for ±2g full scale range, the g values are between -2g and +1.999g, making a 4 g range.
  - the range of g values is implemented using 12 bits raw value.
  - for each count from the 12 bits raw value, it corresponds a value in g:
    - (<full scale range> / 2^12)
  - for example, for ±2g full scale range, the value corresponding to each count is (4/2^12)g.
- To compute the value in g, the raw value is multiplied with the conversion constant (value corresponding to each count), considering the sign from the 12 bit raw value.

Float values computing

The method of converting the raw values in g values is implemented using float values computing. This costs performance, so user should be aware of it. The library is implemented so that the conversion constant (value corresponding to each count) is pre-computed when the full scale range is set.

If user chooses to use the raw values instead of g values (use ACL_ReadRawValues instead of ACL_ReadGValues), no float values computing is involved.

Library files

The library is implemented in these files: acl.c, acl.h. Include them in the project, together with i2c.c and i2c.h.

Include the library header file whenever you want to use the library functions:

```
#include "acl.h"
```

8.2 Library functions

8.2.1 ACL_Init

Synopsis

void ACL_Init();

Description

This function initializes the hardware involved in the ACL module: the I^2C1 module of PIC32 is configured to work at 400 khz, the ACL is initialized at ±2g full scale range and the ACL_INT2 pin is configured as digital input.

Example

```
#include "ACL.h"
ACL_Init();
```

8.2.2 ACL_SetRange

Synopsis

unsigned char ACL_SetRange(unsigned char bRange)

Parameters

- unsigned char bRange - The full scale range identification.

Value	Full scale range
0	±2g
1	±4g
2	±8g

Return

- unsigned char

Return Value	Meaning
0	success
0xFF	the slave address was not acknowledged by the device error
0xFE	timeout error

Description

This function sets the full scale range. It sets the according bits in the 0x0E: XYZ_DATA_CFG register. The function also pre-computes the fGRangeLSB to be used when converting raw values to g values.

It returns the status of the operation: success or I2C errors (the slave address was not acknowledged by the device or timeout error).

Example

```
#include “acl.h”
ACL_Init();
ACL_SetRange(0); // +/- 2g
```

8.2.3 ACL_ReadRawValues

Synopsis

void ACL_ReadRawValues(unsigned char *rgRawVals)

Parameters

- unsigned char *rgRawVals - Pointer to a buffer where the received bytes will be placed. It will contain the 6 bytes, one pair for each of to the 3 axes:

Index	Byte content	B7	B6	B5	B4	B3	B2	B1	B0
0	MSB of X reading	X11	X10	X9	X8	X7	X6	X5	X4
1	LSB of X reading	X3	X2	X1	X0	0	0	0	0
2	MSB of Y reading	Y11	Y10	Y9	Y8	Y7	Y6	Y5	Y4
3	LSB of Y reading	Y3	Y2	Y1	Y0	0	0	0	0
4	MSB of Z reading	Z11	Z10	Z9	Z8	Z7	Z6	Z5	Z4
5	LSB of Z reading	Z3	Z2	Z1	Z0	0	0	0	0

In the above table, the raw value for each axis is a 12 bits value: X11-X0, Y11-Y0, Z11-Z0, the 0 bit being the LSB bit.

Description

This function returns the acceleration value in terms of g, computed from the 2 bytes acceleration raw values (2's complement on 12 bites).
Each raw value is represented on 12 bits so it will be represented on 2 bytes: The MSB byte contains the 8 MSB bits of the raw value, while the LSB byte contains the 4 LSB bits of the raw value padded right with 4 bits of 0.
Computing the acceleration in terms of g is done with this formula:
(<full scale range> / 2^12)*<raw value>
The (<full scale range> / 2^12) term is pre-computed every time the range is set, using global variable fGRangeLSB.
This function involves float values computing, so avoid using it intensively when performance is an issue.

Example

```
#include “acl.h”
unsigned char rgRawVals[6];
ACL_Init();
ACL_ReadRawValues(rgRawVals);
```

8.2.4 ACL_ReadGValues

Synopsis

void ACL_ReadGValues(float *rgGVals)

Parameters

- float *rgGVals - Pointer to a buffer where the acceleration values expressed in g will be placed.
  It will contain the 3 float values, one for each of to the 3 axes:

Index	Content
0	Acceleration on X axis (in terms of g)
1	Acceleration on Y axis (in terms of g)
2	Acceleration on Z axis (in terms of g)

Description

This function provides the acceleration values for the three axes, as float values in terms of g. The raw values are acquired using ACL_ReadRawValues. Then, for each axis, the 2 bytes are converted to a float value in terms of in g.

This function involves float values computing, so avoid using it intensively when performance is an issue.

Example

```
#include “acl.h”
float rgACLGVals[3];
ACL_Init();
ACL_ReadGValues(rgACLGVals);
```

8.2.5 ACL_ConvertRawToValueG

Synopsis

float ACL_ConvertRawToValueG(unsigned char *rgRawVals)

Parameters

- unsigned char *rgRawVals - Pointer to a buffer that contains the 2 bytes corresponding to the raw value:

Byte	Byte content	B7	B6	B5	B4	B3	B2	B1	B0
0	MSB of the raw value	V11	V10	V9	V8	V7	V6	V5	V4
1	LSB of the raw value	V3	V2	V1	V0	0	0	0	0

In the above table, the raw value is a 12 bits value: V11-V0, the 0 bit being the LSB bit.

Description

This function reads the module raw values for the three axes. Each raw value is represented on 12 bits, so it will be represented on 2 bytes: The MSB byte contains the 8 MSB bits, while the LSB byte contains the 4 LSB bits, padded right with 4 bits of 0.

The raw values are obtained by reading the 6 consecutive registers starting with "0x01: OUT_X_MSB".

It returns the status of the operation: success or I2C errors (the slave address was not acknowledged by the device or timeout error).

Example

```
#include “acl.h”
unsigned char rgRawVals[6];
ACL_Init();
unsigned char rgRawVals[6];
ACL_ReadRawValues(rgRawVals);
rgGVals[0] = ACL_ConvertRawToValueG(rgRawVals);
rgGVals[1] = ACL_ConvertRawToValueG(rgRawVals + 2);
rgGVals[2] = ACL_ConvertRawToValueG(rgRawVals + 4);
```

8.2.6 ACL_GetDeviceID

Synopsis

unsigned char ACL_GetDeviceID()

Parameters

- unsigned char bRange - The full scale range identification.

Value	Full scale range
0	±2g
1	±4g
2	±8g

Return

- unsigned char

Return Value	Meaning
Any value not 0xFE and 0xFF	Device ID
0xFF	the slave address was not acknowledged by the device error
0xFE	timeout error

Description

This function returns the device ID. It obtains it by reading "0x0D: WHO_AM_I Device ID" register.

If errors occur, it returns the I2C error (the slave address was not acknowledged by the device or timeout error).

Example

```
#include “acl.h”
Unsigned char devID;
ACL_Init();
devID = ACL_GetDeviceID();
```

8.2.7 ACL_SetRegister

Synopsis

unsigned char ACL_SetRegister(unsigned char bAddress, unsigned char bValue)

Parameters

- unsigned char bAddress - The register address.
- unsigned char bValue - The value to be written in the register.

Return

- unsigned char

Return Value	Meaning
0	Success
0xFF	the slave address was not acknowledged by the device error
0xFE	timeout error

Description

This function writes the specified value to the register specified by its address.

It returns the status of the operation: success or I2C errors (the slave address was not acknowledged by the device or timeout error).

Example

```
#include “acl.h”
ACL_SetRegister(0x2A, 1);
```

8.2.8 ACL_GetRegister

Synopsis

unsigned char ACL_GetRegister(unsigned char bAddress)

Parameters

- unsigned char bAddress - The register address.

Return value

- unsigned char - The register value.

Description

This function returns the value of the register specified by its address.

Example

```
#include “acl.h”
unsigned char bVal;
bVal = ACL_GetRegister(ACL_XYZDATACFG);
```

8.2.9 ACL_Close

Synopsis

void ACL_Close()

Description

This functions releases the hardware involved in the ACL library: it closes the I2C1 interface.

Example

```
#include “acl.h”
ACL_Close();
```

8.2.10 ACL_ConfigurePins

Synopsis
void ACL_ConfigurePins()
Description
This function configures the digital pins involved in the ACL module: the ACL_INT2 pin is configured as digital input.
The function uses pin related definitions from config.h file.
This is a low-level function called by ACL_Init(), so user should avoid calling it directly.
Example

```
#include “acl.h”
ACL_ConfigurePins();
```

9 Analog Input Control – AIC

9.1 Overview

This library groups the functions that implement the AIC module. The library is used to handle analog input control on thumbwheel potentiometer or analog input connectors labeled AIC.

The analog value is converted to digital value using 10 bits analog to digital conversion implemented by ADC library.

Include aic.c file as well as adc.c and adc.h in the project when this library is needed.

9.2 Library functions

9.2.1 AIC_Init

Synopsis
void AIC_Init ()

Description

This function initializes the hardware involved in the AIC module: the ADC_AN2 pin is configured as analog input, the ADC module is configured by calling the ADC configuration function from adc.c.

Example

```
#include “aic.h”
AIC_Init();     //initializes the AIC module
```

9.2.2 AIC_Val

Synopsis
unsigned int AIC_Val()
Return value

- unsigned int - the 16 LSB bits contain the result of analog to digital conversion of AIC value

Description

This function returns the digital value corresponding to the AIC analog pin (thumbwheel potentiometer or analog input connectors labeled AIC) as the result of analog to digital conversion performed by the ADC module.

Example

```
#include “aic.h”
Unsigned int val = AIC_Val();    // Read the digital value
corresponding to the AIC module
```

9.2.3 AIC_ConfigurePins

Synopsis
void AIC_ConfigurePins()
Description
This function configures the ADC_AN2 pin as analog input.
The function uses pin related definitions from config.h file.
This is a low-level function called by AIC_Init(), so user should avoid calling it directly.
Example

```
#include “aic.h”
AIC_ConfigurePins();
```

10 ADC

10.1 Overview

This library groups the functions that implement the ADC module. The ADC module implements basic 10 bits ADC (analog to Digital converter) functionality provided by the PIC32.

The ADC library provides functions for ADC configuration and reading the 10 bits value from the ADC module.

Include adc.c in the project when ADC is used standalone, or when the AIC and MIC libraries are needed.

Used by other libraries

The ADC library is used by AIC and MIC libraries, as these libraries perform analog pins reading.

Used as a standalone library

The ADC library can be used stand alone for reading any analog input pin. The analog function of the PIC32 pins is shown by the "ANx" reference in the pin function. The x is the number of the analog pin (0-27).

For example, the following PMOD pins have analog functionality:

Pmod pin	Schematic name	PIC32 pin
PMODA_4	JA4	AN16/C1IND/RPG6/SCK2/PMA5/RG6
PMODA_8	JA8	AN17/C1INC/RPG7/PMA4/RG7
PMODA_9	JA9	AN18/C2IND/RPG8/PMA3/RG8
PMODA_10	JA10	AN19/C2INC/RPG9/PMA2/RG9
PMODB_9	JB9	AN24/RPD1/RD1

For example, the PMODA_4 (AN16/C1IND/RPG6/SCK2/PMA5/RG6) is the analog pin 16.

10.2 Library functions

10.2.1 ADC_Init

Synopsis
void ADC_Init()

Description

This function initializes the analog to digital converter module of PIC32 in manual sampling mode.

Example

```
#include "adc.h"
ADC_Init();     //initializes the ADC module
```

10.2.2 ADC_AnalogRead

Synopsis
unsigned int ADC_AnalogRead(unsigned char analogPIN)

Parameters

- unsigned char analogPIN - the number of the analog pin that must be read.

Return value

- unsigned int - the 16 LSB bits contain the result of analog to digital conversion of the analog value of the specified pin

Description

This function returns the digital value corresponding to the analog pin, as the result of analog to digital conversion performed by the ADC module.

Example

```
#include “adc.h”
return ADC_AnalogRead(2);    // Read the ADC Value for analog pin 2
```

11 AUDIO

11.1 Overview

This library groups the functions that implement the AUDIO module. The library implements the generating audio function using a sine of a certain frequency, recording and playback functionality. In order to generate audio using a sine of a certain frequency, the sine values are produced using PWM on OC1.

Audio modes

There are 4 audio modes supported by AUDIO library. The mode is configured when the AUDIO library is initialized, it is passed as an argument in AUDIO_Init() function.

Mode	Name	Features
0	Generate sound using sine	Generates sound using a sine wave: - A fix number of sine values, covering one sine period, are stored in an array. - Using the sine values, PWM is produced using OC1, at a frequency of 48 kHz given by Timer3.
1	Mirror	Acquires samples from MIC and generates Audio output accordingly, live: - MIC is sampled using the MIC library at a frequency of 16 kHz given by Timer3. - According to each acquired sample, the audio out is generated using OC1, at a frequency of 16 kHz given by Timer3.
2	Record	Records values acquired from MIC, saving them into a buffer: - MIC is sampled using the MIC library at a frequency of 16 kHz given by Timer3. - the sampled values are placed in rgAudioBuf (which is a large static character array).
3	Play recorded	Plays the values from a buffer to the Audio output: - This function uses the samples from rgAudioBuf (which is a large static character array), stored with Record mode. - According to each acquired sample, the audio out is generated using OC1, at a frequency of 16 kHz given by Timer3.

As this module has microphone related capabilities, MIC library functions are used inside AUDIO library.

Include the file in the project together with mic.c, mic.h, adc.c, adc.h, config.h when this library is needed.

Shared resources

As shown on Hardware resources map, the AUDIO library shares the Timer3 with MOT library in PH/EN mode library. If the user only use one of the MOT and AUDIO libraries, then there is no conflict. If the user needs to use both MOT in PH/EN mode and AUDIO, then the AUDIO library should be initialized first.

11.2 Library functions

11.2.1 AUDIO_Init

Synopsis

void AUDIO_Init(unsigned char bMode)

Parameters

- unsigned char bMode - the mode for the Audio module.

Value	Mode
0	Generate sound using sine
1	Mirror
2	Record
3	Play recorded

Description

This function initializes the AUDIO module, in the mode indicated by parameter bMode.

The output pin corresponding to AUDIO module (A_OUT) is configured as digital output and is mapped to OC1.
According to the specified mode, Timer3 is set and specific initialization is performed:

Mode 0 (Generate sound using a sine wave.) - Timer3 is initialized at a frequency of 48 kHz. The playback is initialized with the sine definition string.

Mode 1 (mirror) - The MIC module is initialized. Timer3 is initialized at a frequency of 16 kHz.

Mode 2 (record) - The MIC module is initialized. Timer3 is initialized at a frequency of 16 kHz. The record is initialized with the large buffer rgAudioBuf.

Mode 3 (play recorded) - Timer3 is initialized at a frequency of 16 kHz. The playback is initialized with the large buffer rgAudioBuf, where eventually MIC values were stored.

OC1 module is configured to work with Timer3.

The timer period constant is computed using TMR_FREQ_SINE and TMR_FREQ_SOUND definitions (located in this source file) and peripheral bus frequency definition (PB_FRQ, located in config.h).

Example

```
#include “audio.h”
AUDIO_Init(0);  // generates a sound using the sine wave
```

11.2.2 AUDIO_GetAudioMode

Synopsis
unsigned char AUDIO_GetAudioMode()
Return value

- unsigned char - the current audio mode.

Value	Mode
0	Generate sound using sine
1	Mirror
2	Record
3	Play recorded

Description

This function returns the audio mode that was selected when the AUDIO was initialized.

Example

```
#include “audio.h”
…
if(AUDIO_GetAudioMode() != 0)
{
      // the AUDIO module was initialized in Mode 0.
}
```

11.2.3 AUDIO_InitPlayBack

Synopsis
void AUDIO_InitPlayBack(unsigned short *pAudioSamples1, int cntBuf1)
Parameters

- unsigned short *pAudioSamples1 - pointer to a buffer where the 2 bytes values to be played are stored.
- int cntBuf1 - the buffer dimension (the number of values that are stored in the buffer).

Description

This function initializes the playback buffer, by providing a pointer to a buffer where the 2 bytes values to be played are stored and the buffer dimension. This function is called by by AUDIO_Init(), when Mode 3 is initialized.

Example

```
#include “audio.h”
…
AUDIO_InitPlayBack(rgAudioBuf, RECORD_SIZE);
```

11.2.4 AUDIO_InitRecord

Synopsis

void AUDIO_InitPlayBack(unsigned short *pAudioSamples1, int cntBuf1)

Parameters

- unsigned short *pAudioSamples1 - pointer to a buffer where the recorded 2 bytes values will be stored.
- int cntBuf1 - the buffer dimension (the number of values that can be stored in the buffer).

Description

This function initializes the record buffer, by providing a pointer to pointer to a buffer where the recorded 2 bytes values will be stored and the buffer dimension. This function is called by by AUDIO_Init(), when Mode 2 is initialized.

Example

```
#include “audio.h”
…
AUDIO_InitRecord(rgAudioBuf, RECORD_SIZE);
```

11.2.5 AUDIO_ConfigurePins

Synopsis

void AUDIO_ConfigurePins()

Description

This function configures the output pin corresponding to AUDIO module (A_OUT) as digital output and maps it to OC1.
The function uses pin related definitions from config.h file.
This is a low-level function called by AUDIO_Init(), so user should avoid calling it directly.

Example

```
#include “audio.h”
AUDIO_ConfigurePins();
```

12 IRDA

12.1 Overview

This library groups the functions that implement the IRDA module. The library implements the control for it and it also maps the UART5 interface over the IRDA transmit and receive pins. So, in order to send a byte over IRDA, the byte is sent over UART5, thus UART5 serializes each bit to be transmitted over IRDA. Receiving bits from IRDA are accumulated in a byte in UART5 receive buffer.
Include the file together with utils.c and utils.h in the project when this library is needed.

12.2 Library functions

12.2.1 IRDA_Init

Synopsis
void IRDA_Init(unsigned int baud)
Parameters

- unsigned int baud - UART baud rate.
  for example 115200 corresponds to 115200 baud.

Description

This function initializes the hardware involved in the IRDA module:

The following digital pins are configured as digital outputs: IRDA_PDOWN, IRDA_TX.

The following digital pins are configured as digital inputs: IRDA_RX.

IRDA module is set in SMIR mode.

The IRDA_TX and IRDA_RX are mapped over UART5 interface, which is configured to work at the specified baud.

Example

```
#include “IRDA.h”
IRDA_Init(9600);// initialize IRDA, baud 9600, no parity and 1 stop
bit
```

12.2.2 IRDA_UARTPutChar

Synopsis
void IRDA_UARTPutChar (char ch)
Parameters

- char ch - the character to be transmitted over IRDA.

Description

This function transmits a character over IRDA, by transmitting it over UART5.

Example

```
#include “IRDA.h”
IRDA_UARTPutChar(‘D’);     // transmits ‘D’ character over IRDA
```

12.2.3 IRDA_UART_AvaliableRx

Synopsis
unsigned char IRDA_UART_AvaliableRx()
Return value

- unsigned char - Receive Buffer Data Available bit:

return value	When
0	Receive buffer is empty
1	Receive buffer has data, at least one more character can be read

Description

This function returns UART5 Receive Buffer Data Available bit.

It returns 1 if the receive buffer has data (at least one more character can be read).

It returns 0 if the receive buffer is empty.

Example

```
#include “IRDA.h”
if(IRDA_AvaliableRx())
{
     // if there is an available received character over UART
     c = IRDA_GetCharPoll();
}
```

12.2.4 IRDA_UART_GetChar

Synopsis

unsigned char IRDA_UART_GetChar()

Parameters

- unsigned int timeout - the number of times to check for an available received byte
- char *error - pointer to a value that returns the error status

value	Meaning
0	no timeout
1	timeout

Return value

- unsigned char - the byte received over IRDA, by receiving it from UART5

Description

This function receives a character over IRDA.

It waits until a byte is received over UART5.

Then, it returns the byte.

When after a number of times (specified by timeout parameter) no value is available on UART5,

timeout condition is met: *error parameter will return 1 (otherwise it returns 0).

Example

```
#include “IRDA.h”
while(IRDA_UART_AvaliableRx())
{
      IRDA_UART_GetChar(65000, &rx_err);;
}
```

12.2.5 IRDA_ConfigureIRDAOverUART5

Synopsis

void IRDA_ConfigureIRDAOverUART5(unsigned int baud)

Parameters

- unsigned int baud - UART baud rate.
  for example 9600 corresponds to 9600 baud.

Description

This function configures the UART1 hardware interface of PIC32, according to the provided baud rate, no parity and 1 stop bit, with no interrupts.

In order to compute the baud rate value, it uses the peripheral bus frequency definition (PB_FRQ, located in config.h).

This is a low-level function called by IRDA_Init(), so user should avoid calling it directly.

Example

```
#include “IRDA.h”
IRDA_ ConfigureIRDA0verUART5(9600);   // initialize IRDA over
UART5, baud 9600, no parity and 1 stop bit
```

12.2.6 IRDA_ConfigurePins

Synopsis

void IRDA_ConfigurePins()

Description

The following digital pins are configured as digital outputs: IRDA_PDOWN, IRDA_TX.

The following digital pins are configured as digital inputs: IRDA_RX.

The function uses pin related definitions from config.h file.

This is a low-level function called by IRDA_Init(), so user should avoid calling it directly.

Example

```
#include “IRDA.h”
IRDA_ConfigurePins();
```

12.2.7 IRDA_Close

Synopsis

void IRDA_Close()

Description

This functions releases the hardware involved in IRDA library: it turns off the UART5 interface.

Example

```
#include “IRDA.h”
IRDA_Close();   // close IRDA module
```

13 LCD

13.1 Overview

The LCD module is a simple LCD 2x16 display.

The LCD controller contains a character-generator ROM (CGROM) with 192 preset 5×8 character patterns, a character-generator RAM (CGRAM) that can hold 8 user-defined 5×8 characters, and a display data RAM (DDRAM) that can hold 40 character codes for each line, so more than the displayed 16 characters (available using the shift display command).

This library groups the functions that implement the LCD device, using specific LCD control functions. It is accessed in a "parallel like" approach. These are some features of LCD library:

- Low level read and write functionality are implemented using command / data pins, according to the parallel port approach specified above.
- Basic commands are implemented using the low level read and write functions.
- The initialization sequence is implemented according to the implemented commands.
- Different access functions are implemented
    - write to LCD screen
    - access the CGRAM and DDRAM memories

Include the lcd.c together with config.h, utils.c and utils.h in the project when this library is needed.

13.2 Library functions

13.2.1 LCD_Init

Synopsis
void LCD_Init()

Description

This function initializes the hardware used in the LCD module:

The following digital pins are configured as digital outputs: LCD_DISP_RS, LCD_DISP_RW, LCD_DISP_EN

The following digital pins are configured as digital inputs: LCD_DISP_RS.

The LCD initialization sequence is performed, the LCD is turned on.

Example

```
#include “lcd.h”
LCD_Init();     // initialize LCD
```

13.2.2 LCD_ WriteByte

Synopsis
void LCD_WriteByte(unsigned char bData)
Parameters

- unsigned char bData - the data to be written to LCD, over the parallel interface

Description

This function writes a byte to the LCD.

It implements the parallel write using LCD_DISP_RS, LCD_DISP_RW, LCD_DISP_EN,

LCD_DISP_RS pins, and data pins.

For a better performance, the data pins are accessed using a pointer to

the register byte where they are allocated.

This is a low-level function called by LCD write functions, so user should avoid calling it directly.

The function uses pin related definitions from config.h file.

Example

```
#include “lcd.h”
LCD_WriteByte(bCmd);  // writes bCmd byte on the parallel interface of
LCD
```

13.2.3 LCD_ ReadByte

Synopsis
unsigned char LCD_ReadByte()
Return value

- unsigned char - the data read from LCD, over the parallel interface.

Description

This function reads a byte from the LCD.

It implements the parallel read using LCD_DISP_RS, LCD_DISP_RW, LCD_DISP_EN,

LCD_DISP_RS pins, and data pins.

This is a low-level function called by LCD_ReadStatus function, so user should avoid calling it directly.

The function uses pin related definitions from config.h file.

Example

```
#include “lcd.h”
unsigned char bStatus = LCD_ReadByte();     // reads a byte over the
parallel interface of LCD
```

13.2.4 LCD_ ReadStatus

Synopsis
unsigned char LCD_ReadStatus()
Return value

- unsigned char - the status byte that was read.

Description

Reads the status of the LCD.

It clears the RS and calls LCD_ReadByte() function.

The function uses pin related definitions from config.h file.

Example

```
#include “lcd.h”
unsigned char bStatus = LCD_ReadStatus();  // reads the status from
LCD
```

13.2.5 LCD_ WriteCommand

Synopsis
void LCD_WriteCommand(unsigned char bCmd)
Parameters

- unsigned char bCmd - the command code byte to be written to LCD

Description

Writes the specified byte as command.

It clears the RS and writes the byte to LCD.

The function uses pin related definitions from config.h file.

Example

```
#include “lcd.h”
LCD_WriteCommand(cmdLcdClear);   // writes clear display command to the
LCD
```

13.2.6 LCD_ WriteDataByte

Synopsis
void LCD_WriteDataByte(unsigned char bData)
Parameters

- unsigned char bData - the data byte to be written to LCD

Description

Writes the specified byte as data.

It sets the RS and writes the byte to LCD.

The function uses pin related definitions from config.h file.

This is a low-level function called by LCD write functions, so user should avoid calling it directly.

Example

```
#include “lcd.h”
LCD_WriteDataByte(bVal);    // writes bVal byte as data to the LCD
```

13.2.7 LCD_InitSequence

Synopsis

void LCD_WriteDataByte(unsigned char bDisplaySetOptions)

Parameters

- unsigned char bDisplaySetOptions - display options.

  Possible options (to be OR-ed):

option	Meaning
displaySetOptionDisplayOn	display ON
displaySetOptionCursorOn	cursor ON
displaySetBlinkOn	cursor blink ON

Description

This function performs the initializing (startup) sequence.

The LCD is initialized according to the parameter bDisplaySetOptions.

Example

```
#include “lcd.h”
LCD_InitSequence(displaySetOptionDisplayOn);
```

13.2.8 LCD_DisplaySet

Synopsis

void LCD_DisplaySet(unsigned char bDisplaySetOptions)

Parameters

- unsigned char bDisplaySetOptions - display options.

  Possible options (to be OR-ed):

option	Meaning
displaySetOptionDisplayOn	display ON

displaySetOptionCursorOn	cursor ON
displaySetBlinkOn	cursor blink ON

Description

The LCD is initialized according to the parameter bDisplaySetOptions.

If one of the above mentioned optios is not OR-ed, it means that the OFF action is performed for it.

Example

```
#include “lcd.h”
LCD_DisplaySet(displaySetOptionDisplayOn);
```

13.2.9 LCD_ DisplayClear

Synopsis
void LCD_DisplayClear()

Description

Clears the display and returns the cursor home (upper left corner, position 0 on row 0).

Example

```
#include “lcd.h”
LCD_DisplayClear();
```

13.2.10 LCD_ReturnHome

Synopsis
void LCD_ReturnHome()

Description

Returns the cursor home (upper left corner, position 0 on row 0).

Example

```
#include “lcd.h”
LCD_LCD_ReturnHome();
```

13.2.11 LCD_DisplayShift

Synopsis
void LCD_DisplayShift (unsigned char fRight)
Parameters

- unsigned char fRight- specifies display shift direction:

value	Meaning
1	Shift right
0	Shift right

Description

Shifts the display one position right or left, depending on the fRight parameter.

Example

```
#include “lcd.h”
LCD_DisplayShift(1);  // shift display right one position
```

13.2.12 LCD_CursorShift

Synopsis

void LCD_CursorShift(unsigned char fRight)

Parameters

- unsigned char fRight- specifies cursor shift direction:

value	Meaning
1	Shift right
0	Shift right

Description

Shifts the display one position right or left, depending on the fRight parameter.

Example

```
#include “lcd.h”
LCD_CursorShift(1);  // shift cursor right one position
```

13.2.13 LCD_WriteStringAtPos

Synopsis

void LCD_WriteStringAtPos(char *szLn, unsigned char idxLine, unsigned char idxPos)

Parameters

- char *szLn - string to be written to LCD
- int idxLine - line where the string will be displayed

value	Meaning
0	first line of LCD
1	second line of LCD

- unsigned char idxPos - the starting position of the string within the line

value	Meaning
0	first line of LCD

39	last position for DDRAM for one line

Description

Displays the specified string at the specified position on the specified line.

It sets the corresponding write position and then writes data bytes when the device is ready.

Strings longer than 40 characters are trimmed.

It is possible that not all the characters will be visualized, as the display only visualizes 16 characters for one line.

Example

```
#include “lcd.h”
LCD_WriteStringAtPos(“Digilent”, 0, 0);     // write string on the
first line, first position
```

13.2.14 LCD_ SetWriteCgramPosition

Synopsis

void LCD_SetWriteCgramPosition(unsigned char bAdr)

Parameters

- unsigned char bAdr - the write location. The position in CGRAM where the next data write operations will put bytes.

Description

Sets the DDRAM write position. This is the location where the next data write operation will be performed.

Writing to a location auto-increments the write location.

This is a low-level function called by LCD_WriteBytesAtPosCgram(), so user should avoid calling it directly.

Example

```
#include “lcd.h”
LCD_SetWriteCgramPosition(bAdr);
```

13.2.15 LCD_WriteBytesAtPosCgram

Synopsis

void LCD_SetWriteCgramPosition(unsigned char bAdr)

Parameters

- unsigned char *pBytes - pointer to the string of bytes.
- unsigned char len - the number of bytes to be written

- unsigned char bAdr - the position in CGRAM where bytes will be written

Description

Writes the specified number of bytes to CGRAM starting at the specified position.

This allows user characters to be defined.

It sets the corresponding write position and then writes data bytes when the device is ready.

Example

```
#include “lcd.h”
LCD_WriteBytesAtPosCgram(userDefArrow, 8, posCgramChar0);
```

13.2.16 LCD_ConfigurePins

Synopsis

void LCD_ConfigurePins()

Description

This function configures the digital pins involved in the LCD module:

The following digital pins are configured as digital outputs: LCD_DISP_RS, LCD_DISP_RW, LCD_DISP_EN

The following digital pins are configured as digital inputs: LCD_DISP_RS.

The function uses pin related definitions from config.h file.

This is a low-level function called by LCD_Init(), so user should avoid calling it directly.

Example

```
#include “lcd.h”
LCD_ConfigurePins();
```

14 MIC

14.1 Overview

This library groups the functions that implement the AIC module. The library is used to handle analog input from the microphone.

The analog value is converted to digital value using 10 bits analog to digital conversion implemented by ADC library.

Include mic.c file as well as adc.c and adc.h in the project when this library is needed.

14.2 Library functions

14.2.1 MIC_Init

Synopsis
void MIC_Init ()

Description

This function initializes the hardware involved in the MIC module: the MIC pin is configured as analog input, the ADC module is configured by calling the ADC configuration function from adc.c.

Example

```
#include “mic.h”
MIC_Init();     //initializes the MIC module
```

14.2.2 MIC_Val

Synopsis
unsigned int MIC_Val()
Return value

- unsigned int - the 16 LSB bits contain the result of analog to digital conversion of MIC value

Description

This function returns the digital value corresponding to the MIC analog pin as the result of analog to digital conversion performed by the ADC module.

It can be used to sample the microphone input.

Example

```
#include “adc.h”
Unsigned int val = MIC_Val();    // Read the digital value
corresponding to the MIC module
```

14.2.3 MIC_ConfigurePins

Synopsis
void MIC_ConfigurePins()
Description
This function configures the MIC pin as analog input.
The function uses pin related definitions from config.h file.
This is a low-level function called by MIC_Init(), so user should avoid calling it directly.
Example

```
#include “mic.h”
```

```
MIC_ConfigurePins();
```

15 MOT

15.1 Overview

This library groups the functions that implement the MOT module. There are two modes: PH/EN and IN/IN. After common initialization functions, separate functions are provided for each mode. The separate functions contain “PhEn” or “InIn” in their name.

Include the mot.c file in the project, together with config.h when this library is needed.

PH/EN mode

In PH/EN mode, two DC motors are commanded using 2 digital pins: Phase (direction) and Enable (PWM) for each. OC2 and OC3 are configured together with Timer3 to generate PWM for these motors.

IN/IN mode

In IN/IN mode, one stepper motor is commanded by controlling 4 digital pins.

In order to generate commands corresponding to a number of steps in a specific direction, a 4 bit value is used. Each bit corresponds to a pin. The initial value is 1100b. Every time steps are performed, the value is rotated left or right (according to the stepper direction) a number of positions (according to number of steps).

Example:

Value	Action	New value
1100b	1 step direction 1 (rotate right)	0110b
0110b	2 steps direction 0 (rotate left)	1001b

Shared resources

As shown on Hardware resources map, the MOT library in PH/EN mode shares the Timer3 with AUDIO library. If the user only use one of the MOT and AUDIO libraries, then there is no conflict. If the user needs to use both MOT in PH/EN mode and AUDIO, then the AUDIO library should be initialized first.

PWM frequency

As shown above, when using MOT in PH/EN mode Timer3 is shared with AUDIO library.

When initializing the MOT library in PH/EN mode:

- if Timer3 is off then Timer3 is initialized to work at a frequency of 20 kHz.

- if Timer3 is ON (initialized by AUDIO), then Timer3 is not altered. The MOT library will use the AUDIO timer frequencies (16 kHz or 48 kHz, depending on the selected audio mode).

15.2 Library functions

15.2.1 MOT_Init

Synopsis
void MOT_Init(unsigned char bMode1)
Parameters

- unsigned char bMode1 – the mode for the MOT module.

Value	Mode
0	PH/EN mode
1	IN/IN mode

Description

This function initializes the hardware involved in the MOT module, in a specific mode (PH/EN or IN/IN).

The motor command pins are initialized as digital outputs.

For PH/EN mode, the AIN2 and BIN2 are mapped over OC2 and OC3, OC2 and OC3 are properly initialized.

For IN/IN mode, the stepper is initialized to work with 0b1100 value rotated one position over 4 bits.

Example

```
#include “mot.h”
MOT_Init();     // initializes the MOT module
```

15.2.2 MOT_SetPhEnMotor1

Synopsis
void MOT_SetPhEnMotor1(unsigned char bDir, unsigned char bSpeed)
Parameters

- unsigned short bDir - the rotation direction for DC motor 1

Value	Mode
0	turn left
1	turn right

- unsigned char bSpeed - the rotation speed for DC motor 1

Description

This function configures the DC motor 1 by providing the rotation direction and the rotation speed. The direction speed is provided in values between 0 - 255, which are used to configure the PWM on OC2 accordingly. Naming left / write the rotation direction is a simple convention.

The function is supposed to be called in the PH/EN mode, otherwise the function will have no result or effect.

Example

```
#include “mot.h”
MOT_SetPhEnMotor1(0, 128);        // configures DC motor 1 rotation:
direction left, PWM 50%.
```

15.2.3 MOT_SetPhEnMotor2

Synopsis

void MOT_SetPhEnMotor2(unsigned char bDir, unsigned char bSpeed)

Parameters

- unsigned short bDir - the rotation direction for DC motor 2

Value	Action
0	turn left
1	turn right

- unsigned char bSpeed - the rotation speed for DC motor 2

Description

This function configures the DC motor 2 by providing the rotation direction and the rotation speed. The direction speed is provided in values between 0 - 255, which are used to configure the PWM on OC2 accordingly. Naming left / write the rotation direction is a simple convention.

The function is supposed to be called in the PH/EN mode, otherwise the function will have no result or effect.

Example

```
#include “mot.h”
MOT_SetPhEnMotor2(0, 128);        // configures DC motor 2 rotation:
direction left, PWM 50%.
```

15.2.4 MOT_InInInitStep

Synopsis

void MOT_InInInitStep(unsigned char bInitVal, unsigned char noBits)

Parameters

- unsigned char bInitVal - the byte containing the bits that will be rotated for each step
- unsigned char noBits - the number of bits that will be rotated for each step

Description

This function configures the Step motor command parameters. Performing the steps is done by calling MOT_InInPerformStep.

The function is supposed to be called in the IN/IN mode, otherwise the function will have no result or effect.

Example

```
#include “mot.h”
```

```
MOT_InInInitStep (0x0C, 4); // 0b1100 rotated one position over 4 bits
```

15.2.5 MOT_InInPerformStep

Synopsis

void MOT_InInPerformStep(int noSteps, unsigned char bDir)

Parameters

- int noSteps - the number of steps to be performed
- unsigned char bDir - the stepper motor rotation direction

Value	Action
0	turn left
1	turn right

Description

This function commands the stepper motor to perform a number of steps in a certain direction.

For this, a 4 bit value is used. Each bit corresponds to a pin. The initial value is 1100b.

Every time steps are performed, the value is rotated left or right (according to the stepper direction)

a number of positions (according to number of steps).

Naming left / write the rotation direction is a simple convention.

The function is supposed to be called in the IN/IN mode, otherwise the function will have no result or effect.

Example

```
#include “mot.h”
MOT_InInInitStep (0x0C, 4); // 0b1100 rotated one position over 4 bits
```

15.2.6 MOT_ConfigureOCs

Synopsis

void MOT_ConfigureOCs()

Description

This function configures the OC2 and OC3 to work together with Timer3 in order to generate PWM in PH/EN mode.

This is a low-level function called by MOT_Init(), so user should avoid calling it directly.

MOT library in mode PH/EN shares the Timer3 with AUDIO library.

If the Timer3 is ON (initialized by AUDIO module), then Timer3 is not altered. The MOT library will use the AUDIO timer frequencies (16 kHz or 48 kHz).

If the Timer3 is OFF, then Timer3 is initialized at a frequency of 20kHz.

The timer period constant is computed using TMR_FREQ_MOT definition (located in this source file) and peripheral bus frequency definition (PB_FRQ, located in config.h).

Example

```
#include “mot.h”
MOT_ConfigureOCs();
```

15.2.7 MOT_ConfigurePins

Synopsis

void MOT_ConfigurePins()

Description

This function configures the IO pins involved in the MOT module as digital output pins.

The function uses pin related definitions from config.h file.

This is a low-level function called by MOT_Init(), so user should avoid calling it directly.

Example

```
#include “mot.h”
MOT_ConfigurePins();
```

15.2.8 MOT_ PhEnComputeOCFromSpeed

Synopsis

unsigned short MOT_PhEnComputeOCFromSpeed(unsigned char bSpeed)

Parameters

- unsigned char bSpeed - the rotation speed for DC motor 1

Return value

- unsigned short - the OCxRS value that corresponds to the specified speed

Description

This function computes the value to be assigned to the OCxRS register in order to implement the PWM corresponding to the specified speed.

This is the formula: OCxRS = speed*(PR3 + 1) / 256, where PR3 is the Timer3 period, 256 is the number of speed values (duty resolution).

This is a low-level function called by MOT_SetPhEnMotor() functions.

Example

```
#include “mot.h”
OC2RS = MOT_PhEnComputeOCFromSpeed(bSpeed);
```

15.2.9 MOT_Close

Synopsis
void MOT_Close()
Description
This functions releases the hardware involved in MOT library: it turns off the OC2, OC3 and Timer3 interfaces.
Example

```
#include “mot.h”
MOT_Close();
```

16 PMODS

16.1 Overview

This library groups the functions that implement the PMODS functionality. Pins from PMODA and PMODB can be initialized as digital input / output pins, their value can be accessed using set / get functions. Include the pmods.c in the project, together with config.h when this library is needed.

16.2 Library functions

16.2.1 PMODS_ InitPin

Synopsis
void PMODS_InitPin(unsigned char bPmod, unsigned char bPos, unsigned char bDir, unsigned char pullup, unsigned char pulldown)
Parameters

- unsigned char bPmod - the PMOD where the pin is located

value	PMOD
0	PMODA
1	PMODB

- unsigned char bPos - the pin position in the Pmod (allowed values 1-4, 7-10)

value	pin	if
1	JA1	bPmod = 0
	JB1	bPmod = 1
2	JA2	bPmod = 0
	JB2	bPmod = 1
3	JA3	bPmod = 0
	JB3	bPmod = 1
4	JA4	bPmod = 0
	JB4	bPmod = 1
7	JA7	bPmod = 0
	JB7	bPmod = 1
8	JA8	bPmod = 0
	JB8	bPmod = 1
9	JA9	bPmod = 0

	JB9	bPmod = 1
10	JA10	bPmod = 0
	JB10	bPmod = 1

- unsigned char bDir - the pin direction

value	Direction
0	Output
1	Input

- unsigned char pull-up - the pull-up property of the pin

value	Direction
0	No pull-up
1	Pull-up

- unsigned char pull-down - the pull- down property of the pin

value	Direction
0	No pull- down
1	Pull- down

Description

This function configures the pins located in the PMODA and PMODB connectors

to be used as digital input / output, also allowing pullup and pulldown properties to be specified.

This function uses pin related definitions from config.h file.

If the bPmod and bPos do not specify a valid pin, nothing happens.

Example

```
#include “pmods.h”
PMODS_InitPin(0, 1, 0, 0, 0); // Initialize JA1 output, no pull-up, no pull-down
```

16.2.2 PMODS_GetValue

Synopsis

unsigned char PMODS_GetValue(unsigned char bPmod, unsigned char bPos)

Parameters

- unsigned char bPmod - the PMOD where the pin is located

value	PMOD
0	PMODA
1	PMODB

- unsigned char bPos - the pin position in the Pmod (allowed values 1-4, 7-10)

value	pin	if
1	JA1	bPmod = 0
	JB1	bPmod = 1
2	JA2	bPmod = 0
	JB2	bPmod = 1
3	JA3	bPmod = 0
	JB3	bPmod = 1
4	JA4	bPmod = 0

	JB4	bPmod = 1
7	JA7	bPmod = 0
	JB7	bPmod = 1
8	JA8	bPmod = 0
	JB8	bPmod = 1
9	JA9	bPmod = 0
	JB9	bPmod = 1
10	JA10	bPmod = 0
	JB10	bPmod = 1

Return value

- the value of the digital pin or error

value	when
0	the value of the digital pin
1	
0xFF	bPmod and bPos do not specify a valid pin

Description

This function returns the value of a digital pin located in the PMODA and PMODB connectors, specified by bPmod and bPos.

If bPmod and bPos do not specify a valid pin, 0xFF is returned.

This function uses pin related definitions from config.h file.

Example

```
#include “pmods.h”
val = PMODS_GetValue(0, 1);    // Return value of JA1
```

16.2.3 PMODS_SetValue

Synopsis

void PMODS_SetValue(unsigned char bPmod, unsigned char bPos, unsigned char bVal)

Parameters

- unsigned char bPmod - the PMOD where the pin is located

value	PMOD
0	PMODA
1	PMODB

- unsigned char bPos - the pin position in the Pmod (allowed values 1-4, 7-10)

value	pin	if
1	JA1	bPmod = 0
	JB1	bPmod = 1
2	JA2	bPmod = 0
	JB2	bPmod = 1
3	JA3	bPmod = 0
	JB3	bPmod = 1

4	JA4	bPmod = 0
	JB4	bPmod = 1
7	JA7	bPmod = 0
	JB7	bPmod = 1
8	JA8	bPmod = 0
	JB8	bPmod = 1
9	JA9	bPmod = 0
	JB9	bPmod = 1
10	JA10	bPmod = 0
	JB10	bPmod = 1

- unsigned char bVal - the value to be assigned to the digital pin

value	when
0	the value to be assigned
1	to the digital pin

Description

This function assigns the digital value specified by bVal to the digital

pin located in the PMODA and PMODB connectors, specified by bPmod and bPos.

If bPmod and bPos do not specify a valid pin, nothing is performed.

This function uses pin related definitions from config.h file.

Example

```
#include “pmods.h”
val = PMODS_ SetValue(1, 7, 1);     // Set value 1 to JB7
```

17 SPIFLASH

17.1 Overview

This library groups the functions that implement the SPIFLASH device. The library implements SPI access to the onboard SPI Flash memory over SPI1 hardware interface of PIC32 and provides basic functions to configure the SPI Flash memory, write and read functions to access SPI Flash memory bytes.

Include the spiflash.c in the project, together with config.h, when this library is needed.

17.2 Library functions

17.2.1 SPIFLASH_Init

Synopsis

void SPIFLASH_Init()

Description

This function initializes the hardware involved in the SPIFLASH module:

The following digital pins are configured as digital outputs (SPIFLASH_CE, SPIFLASH_SCK, SPIFLASH_SI).

The following digital pins are configured as digital inputs (SPIFLASH_SO).

The SPIFLASH_SI and SPIFLASH_SO are mapped over the SPI1 interface.

The SPI1 module of PIC32 is configured to work at 1 Mhz, polarity 0 and edge 1.

Example

```
#include “spiflash.h”
SPIFLASH_Init();//initializes the SPIFLASH module
```

17.2.2 SPIFLASH_ConfigureSPI

Synopsis

void SPIFLASH_ConfigureSPI(unsigned int spiFreq, unsigned char pol, unsigned char edge)

Parameters

- unsigned int spiFreq - SPI clock frequency (Hz)

  for example, 1000000 corresponds to 1 MHz

- unsigned char pol - SPI Clock Polarity, similar to CKP field of SPIxCON

Value	Meaning
1	Idle state for clock is a high level; active state is a low level
0	Idle state for clock is a low level; active state is a high level

- unsigned char edge - SPI Clock Edge, similar to CKE field of SPIxCON

Value	Meaning
1	Serial output data changes on transition from active clock state to Idle clock state (see CKP bit)
0	Serial output data changes on transition from Idle clock state to active clock state (see CKP bit)

Description

This function configures the SPI1 hardware interface of PIC32, according to the provided parameters. In order to compute the baud rate value, it uses the peripheral bus frequency definition (PB_FRQ, located in config.h). This is a low-level function called by initialization functions, so user should avoid calling it directly.

Example

```
#include “spiflash.h”
SPIFLASH_ConfigureSPI(1000000, 0, 1); // configures SPI to work at 1
Mhz, polarity 0 and edge 1.
```

17.2.3 SPIFLASH_TransferBytes

Synopsis

void SPIFLASH_TransferBytes(int bytesNumber, unsigned char *pbRdData, unsigned char *pbWrData)

Parameters

- int bytesNumber - Number of bytes to be transferred.

- unsigned char *pbRdData - Pointer to a buffer storing the received bytes.
- unsigned char *pbWrData - Pointer to a buffer storing the bytes to be transmitted.

Description

This function implements transfer of a number of bytes over SPI1.
It transmits the bytes from pbWrData and receives the bytes in pbRdData.
This function properly handles Slave Select (SPIFLASH_CE) pin.

Example

```
#include “spiflash.h”
unsigned char bWrBytes[2], bRdBytes[2];

SPIFLASH_TransferBytes(2, bRdBytes, bWrBytes);   // send and receive 2 bytes over SPI1
```

17.2.4 SPIFLASH_SendOneByteCmd

Synopsis

void SPIFLASH_SendOneByteCmd(unsigned char bCmd)

Parameter

- unsigned char bCmd - the command ID

Description

This function sends one byte command over SPI.

Example

```
#include “spiflash.h”
SPIFLASH_SendOneByteCmd(SPIFLASH_CMD_ERASE_ALL);
```

17.2.5 SPIFLASH_ReleasePowerDownGetDeviceID

Synopsis

unsigned char SPIFLASH_ReleasePowerDownGetDeviceID()

Return value

- unsigned char - the Device ID provided by Deep-Power-Down / Device ID command

Description

This function implements the Release from Deep-Power-Down / Device ID command.
The obtained Device ID is returned.

Example

```
#include “spiflash.h”
unsigned char bVal = SPIFLASH_ReleasePowerDownGetDeviceID();
```

17.2.6 SPIFLASH_GetStatus

Synopsis
unsigned char SPIFLASH_GetStatus()
Return value

- unsigned char - the Status Register-1

Description
This function reads the Status Register-1 and returns it.
Example

```
#include “spiflash.h”
unsigned char bVal = SPIFLASH_GetStatus();
```

17.2.7 SPIFLASH_WaitUntilNoBusy

Synopsis
void SPIFLASH_WaitUntilNoBusy()
Description
This function reads the Status Register-1 and waits until Busy flag is not set.
Example

```
#include “spiflash.h”
SPIFLASH_WaitUntilNoBusy();
```

17.2.8 SPIFLASH_ WriteEnable

Synopsis
void SPIFLASH_WriteEnable()
Description
This function calls the Write Enable command.
Example

```
#include “spiflash.h”
SPIFLASH_WriteEnable();
```

17.2.9 SPIFLASH_ WriteDisable

Synopsis
void SPIFLASH_WriteDisable()
Description
This function calls the Write Disable command.
Example

```
#include “spiflash.h”
SPIFLASH_WriteDisable();
```

17.2.10 SPIFLASH_Erase4k

Synopsis
void SPIFLASH_Erase4k()
Parameter

- unsigned int addr - the address where the sector starts

Description
This functions performs a sector erase: erases the 4k sector starting at the address.
Example

```
#include “spiflash.h”
SPIFLASH_Erase4k(0);
```

17.2.11 SPIFLASH_ Erase64k

Synopsis
void SPIFLASH_Erase64k()
Parameter

- unsigned int addr - the address where the block starts

Description
This functions performs a block erase: erases the 64k block starting at the address.
Example

```
#include “spiflash.h”
SPIFLASH_Erase64k(0);
```

17.2.12 SPIFLASH_EraseAll

Synopsis
void SPIFLASH_EraseAll()

Description
This functions performs a chip erase: erases all memory within the device.
Example

```
#include “spiflash.h”
SPIFLASH_EraseAll(0);
```

17.2.13 SPIFLASH_ProgramPage

Synopsis
void SPIFLASH_ProgramPage(unsigned int addr, unsigned char *pBuf, unsigned int len)
Parameter

- unsigned int addr - The memory address where data will be written.
- unsigned char *pBuf - Pointer to a buffer storing the bytes to be written.
- int len - Number of bytes to be written.

Description
This functions calls the Page Program command:
it allows from one byte to 256 bytes (a page) of data to be programmed at previously erased memory locations.

Example

```
#include “spiflash.h”
SPIFLASH_ProgramPage(0, pBuf, 1);     // writes one byte from pBuf
```

17.2.14 SPIFLASH_Read

Synopsis
void SPIFLASH_ProgramPage(unsigned int addr, unsigned char *pBuf, unsigned int len)

Parameter

- unsigned int addr - The memory address from where the data will be read.
- unsigned char *pBuf - Pointer to a buffer storing the read bytes.
- int len - Number of bytes to be read.

Description
This functions calls the Read Data command:
it allows one or more data bytes to be sequentially read from the memory.

Example

```
#include “spiflash.h”
SPIFLASH_Read(0, pBuf, 1); // read one byte into pBuf
```

17.2.15 SPIFLASH_RawTransferByte

Synopsis
unsigned char SPIFLASH_RawTransferByte(unsigned char bVal)

Parameters

- unsigned char bVal - the byte to be transmitted over SPI

Return value

- unsigned char - the byte received over SPI

Description
This function implements basic byte transfer over SPI1. It transmits the parameter bVal and returns the received byte.
This function does not handle Slave Select (SPIFLASH_CE) pin, use SPIFLASH_TransferBytes for SPI transfer.
This is a low-level function called by SPIFLASH_TransferBytes(), so user should avoid calling it directly.

Example

```
#include “spiflash.h”
pbRdData[i] = SPIFLASH_RawTransferByte(pbWrData[i]);
```

17.2.16 SPIFLASH_Close

Synopsis
void SPIFLASH_Close()
Description
This functions releases the hardware involved in SPIFLASH library: it turns off the SPI1 interface.
Example

```
#include “spiflash.h”
SPIFLASH_Close();
```

17.2.17 SPIFLASH_ConfigurePins

Synopsis
void SPIFLASH_ConfigurePins()
Description
This function configures the digital pins involved in the SPIFLASH module:
The following digital pins are configured as digital outputs: SPIFLASH_CE, SPIFLASH_SCK, SPIFLASH_SI.
The following digital pins are configured as digital inputs: SPIFLASH_SO.
The SPIFLASH_SI and SPIFLASH_SO are mapped over the SPI1 interface.
The function uses pin related definitions from config.h file.
This is a low-level function called by SPIFLASH_Init(), so user should avoid calling it directly.
Example

```
#include “spiflash.h”
SPIFLASH_ConfigurePins();
```

18 SPIJA

18.1 Overview

This library implements the SPI2 (hardware interface of PIC32) functionality over the PMODA pins:

Pmod pin	Schematic name	PIC32 pin	Name / SPI Function
PMODA_1	JA1	RPC2/RC2	SPIJA_CE
PMODA_2	JA2	RPC1/RC1	SPIJA_SI
PMODA_3	JA3	RPC4/CTED7/RC4	SPIJA_SO
PMODA_4	JA4	AN16/C1IND/RPG6/SCK2/PMA5/RG6	SPIJA_SCK

The library provides basic functions to configure SPI (including pin mapping) and SPI transfer functions.

Include the spija.c file in the project, together with config.h, when this library is needed.

18.2 Library functions

18.2.1 SPIJA_Init

Synopsis
void SPIJA_Init()

Description

This function initializes the hardware involved in the SPIJA module:

The following digital pins are configured as digital outputs: SPIJA_CE (JA1), SPIJA_SCK(JA2), SPIJA_SI(JA3)

The following digital pins are configured as digital inputs: SPIJA_SO(JA4).

The SPIJA_SI and SPIJA_SO are mapped over the SPI2 interface.

The SPI2 module of PIC32 is configured to work at 1 Mhz, polarity 0 and edge 1.

Example

```
#include “spija.h”
SPIJA_Init();   //initializes the SPIJA module
```

18.2.2 SPIJA_ConfigureSPI

Synopsis
void SPIJA_ConfigureSPI(unsigned int spiFreq, unsigned char pol, unsigned char edge)
Parameters

- unsigned int spiFreq - SPI clock frequency (Hz)
  for example, 1000000 corresponds to 1 MHz
- unsigned char pol - SPI Clock Polarity, similar to CKP field of SPIxCON

Value	Meaning
1	Idle state for clock is a high level; active state is a low level
0	Idle state for clock is a low level; active state is a high level

- unsigned char edge - SPI Clock Edge, similar to CKE field of SPIxCON

Value	Meaning
1	Serial output data changes on transition from active clock state to Idle clock state (see CKP bit)
0	Serial output data changes on transition from Idle clock state to active clock state (see CKP bit)

Description
This function configures the SPI2 hardware interface of PIC32, according to the provided parameters. In order to compute the baud rate value, it uses the peripheral bus frequency definition (PB_FRQ, located

in config.h). This is a low-level function called by initialization functions, so user should avoid calling it directly.

Example

```
#include “spija.h”
SPIJA_ConfigureSPI(1000000, 0, 1); // configures SPI to work at 1 Mhz,
polarity 0 and edge 1.
```

18.2.3 SPIJA_TransferBytes

Synopsis

void SPIJA_TransferBytes(int bytesNumber, unsigned char *pbRdData, unsigned char *pbWrData)

Parameters

- int bytesNumber - Number of bytes to be transferred.
- unsigned char *pbRdData - Pointer to a buffer storing the received bytes.
- unsigned char *pbWrData - Pointer to a buffer storing the bytes to be transmitted.

Description

This function implements transfer of a number of bytes over SPI2.
It transmits the bytes from pbWrData and receives the bytes in pbRdData.
This function properly handles Slave Select (SPIJA_CE) pin.

Example

```
#include “spija.h”
unsigned char bWrBytes[2], bRdBytes[2];

SPIJA_TransferBytes(2, bRdBytes, bWrBytes); // send and receive 2 bytes
over SPI2
```

18.2.4 SPIJA_RawTransferByte

Synopsis

unsigned char SPIJA_RawTransferByte(unsigned char bVal)

Parameters

- unsigned char bVal - the byte to be transmitted over SPI

Return value

- unsigned char - the byte received over SPI

Description

This function implements basic byte transfer over SPI2. It transmits the parameter bVal and returns the received byte.
This function does not handle Slave Select (SPIJA_CE) pin, use SPIJA_TransferBytes for SPI transfer.
This is a low-level function called by SPIJA_TransferBytes(), so user should avoid calling it directly.

Example

```
#include “spija.h”
pbRdData[i] = SPIJA_RawTransferByte(pbWrData[i]);
```

18.2.5 SPIJA_Close

Synopsis
void SPIJA_Close()
Description
This functions releases the hardware involved in SPIJA library: it turns off the SPI2 interface.
Example

```
#include “spija.h”
SPIJA_Close();
```

18.2.6 SPIJA_ConfigurePins

Synopsis
void SPIJA_ConfigurePins()
Description
This function configures the digital pins involved in the SPIJA module:
The following digital pins are configured as digital outputs: SPIJA_CE (JA1), SPIJA_SCK(JA2), SPIJA_SI(JA3)
The following digital pins are configured as digital inputs: SPIJA_SO(JA4).
The SPIJA_SI and SPIJA_SO are mapped over the SPI2 interface.
The function uses pin related definitions from config.h file.
This is a low-level function called by SPIJA_Init(), so user should avoid calling it directly.
Example

```
#include “spija.h”
SPIJA_ConfigurePins();
```

19 SRV

19.1 Overview

This library handles the 2 servo modules SERVO 1 and SERVO 2.

Servo pins are mapped over OC5 and OC4 that are properly configured in order to generate PWM for each servo at a period of 20 ms. For each servo, the control function allows to set the pulse duration in microseconds. Include the file in the project, together with config.h when this library is needed.

Servo pins are mapped over OC5 and OC4 that are properly configured (period 20 ms). Library functions allow to specify for each servo the pulse duration.

Shared pins / functionality

As you can see from the schematic, the S0_PWM signals shares the same PIC32 pin with BTNR and the S1_PWM signals shares the same PIC32 pin with BTND (see BTN chapter). So, servos and buttons resources should be used exclusively.

19.1.1 SRV_Init

Synopsis
void SRV_Init()

Description

This function initializes the hardware involved in the SRV module:

The Servo 1 PWM and Servo 2 PWM pins are configured as digital out and mapped to OC5 and OC4. The OC5 and OC4 module of PIC32 are configured with a period of 20 ms given by Timer2.

Example

```
#include “srv.h”
SRV_Init();     // initializes the SRV module
```

19.1.2 SRV_SetPulseMicroseconds1

Synopsis
void SRV_SetPulseMicroseconds1(unsigned short usVal)

Parameters

- unsigned short usVal - the pulse width in microseconds

Description

This function configures the output compare 5 (corresponding to servo 1) according to the specified pulse width.

Example

```
#include “srv.h”
SRV_SetPulseMicroseconds1(1500);      // set a pulse width of 1.5 ms
```

19.1.3 SRV_SetPulseMicroseconds2

Synopsis
void SRV_SetPulseMicroseconds2(unsigned short usVal)

Parameters

- unsigned short usVal - the pulse width in microseconds

Description

This function configures the output compare 4 (corresponding to servo 2) according to the specified pulse width.

Example

```
#include “srv.h”
SRV_SetPulseMicroseconds2(2000);      // set a pulse width of 2 ms
```

19.1.4 SRV_ConfigureOCs

Synopsis
void SRV_ConfigureOCs()

Description

This function configures the output compares and timer involved in the SRV module. The OC5 and OC4 module of PIC32 are configured with a period of 20 ms given by Timer2.

This is a low-level function called by SRV_Init(), so user should avoid calling it directly.

Example

```
#include “srv.h”
SRV_ConfigureOCs();
```

19.1.5 SRV_ConfigurePins

Synopsis
void SRV_ConfigurePins()

Description

This function configures the digital pins involved in the SRV module:

The servo 1 and servo 2 pins are defined as digital out and mapped to OC5 and OC4.

The function uses pin related definitions from config.h file.

This is a low-level function called by SRV_Init(), so user should avoid calling it directly.

Example

```
#include “srv.h”
SRV_ConfigurePins();
```

20 SSD

20.1 Overview

This library deals with seven segment display modules. The module contains 4 digits, each showing 7 segments value and a decimal point.

As digits share the cathodes, in order to be able to display different information on each digit, periodically each digit is refreshed, while the others are disabled. This happens faster than the human can notice.

These are the features of the SSD implementation:

- One array contains constant values for the segments configurations (one bit for each segment) corresponding to various digits (0-9, A-F, H). See Digit Value explanation, below.
- When user selects the values to be displayed, the values are used as index in this segments configuration table and the resulting configuration bytes are stored in global variables.
- Timer1 is used to generate interrupts every approx. 3 ms .
- Every time the interrupt handler routine is called, the following operations are performed:
    - the next digit becomes the current digit, in a circular approach (thus each digit will be addressed once after 4 calls of the interrupt handler routine)
    - all digits are deactivated by outputting 1 to their corresponding anodes.
    - the cathodes are outputted according to the segments information corresponding to the current digit
    - the current digit is activated (0 is outputted to its corresponding anode)

The library provides functions for setting the information to be displayed.

Include the ssd.c in the project, together with config.h, when this library is needed.

Digit value

Some fixed configurations of the segments are defined inside the SSD library. They correspond to the visual representations of some characters to be displayed. Each of these configurations are identified by the index in this table, chosen to be the most intuitive in the following manner:

Index	Character
0	'0'
1	'1'
2	'2'
3	'3'
4	'4'
5	'5'
6	'6'
7	'7'
8	'8'
9	'9'
10	'A'
11	'b'
12	'C'
13	'd'
14	'E'
15	'F'
16	'H'

The index can be considered as the represented "digit value".

Decimal point value

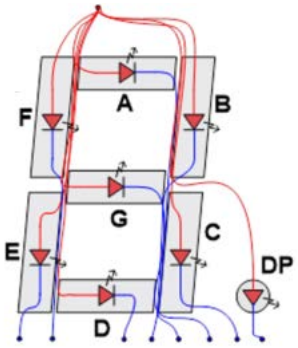
Each digit allows to command the decimal point. In order to specify the values for the decimal points in the library functions calls, this condition is used:

Value for Decimal Point	When

1	Decimal point is turned on
0	Decimal point is turned off

Segments naming

Each digit contains seven segments, referenced using SA-SG letters or DP like this:



Digits numbering

The Basys MX3 board contains 4 seven segment digits. The rightmost is considered to be the first, and the leftmost is the fourth. This convention reflects the hardware schematic naming, e.g. pins AN0 – AN3.

20.2 Library functions

20.2.1 SSD_Init

Synopsis

void SSD_Init()

Description

This function initializes the hardware involved in the SSD module:

The pins corresponding to SSD module are initialized as digital outputs.

The Timer1 is initialized to generate interrupts every approx. 3 ms.

Example

```
#include “ssd.h”
SSD_Init(0);    //initializes the SSD module
```

20.2.2 SSD_WriteDigits

Synopsis

void SSD_WriteDigits(unsigned char d1, unsigned char d2, unsigned char d3, unsigned char d4, \
unsigned char dp1, unsigned char dp2, unsigned char dp3, unsigned char dp4)

Parameters

- unsigned char d1 - the value to be represented on first (rightmost) digit of SSD. Should be between 0 and 16, according to the digit value explained in the Overview chapter.
- unsigned char d2 - the value to be represented on second digit of SSD. Should be between 0 and 16, according to the digit value explained in the Overview chapter.
- unsigned char d3 - the value to be represented on third digit of SSD. Should be between 0 and 16, according to the digit value explained in the Overview chapter.
- unsigned char d4 - the value to be represented on fourth (leftmost) digit of SSD. Should be between 0 and 16, according to the digit value explained in the Overview chapter.
- unsigned char dp1 - the value corresponding to the decimal point for the first (rightmost) digit of SSD, according to the decimal point value explained in the Overview chapter.
- unsigned char dp2 - the value corresponding to the decimal point for the second digit of SSD, according to the decimal point value explained in the Overview chapter.
- unsigned char dp3 - the value corresponding to the decimal point for the third digit of SSD, according to the decimal point value explained in the Overview chapter.
- unsigned char dp2 - the value corresponding to the decimal point for the fourth (leftmost) digit of SSD, according to the decimal point value explained in the Overview chapter.

Description

This function sets the 4 values and 4 decimal points to be displayed on the 4 SSD digits. If d1, d2, d3 or d4 is outside 0 - 16, the corresponding SSD digits will display nothing.

Example

```
#include “ssd.h”
…
SSD_WriteDigits(4, 3, 2, 1, 0, 0, 0, 0); // displays “1234” on the SSD digits, decimal points off
```

20.2.3 SSD_WriteDigitsGrouped

Synopsis

void SSD_WriteDigitsGrouped(unsigned int val, unsigned char dp)

Parameters

- unsigned int val - each of the 4 bytes contains the value to be displayed on a SSD digit:

Byte	contains
0 (LSB)	the value to be displayed on first (rightmost) digit of SSD
1	the value to be displayed on second digit of SSD

2	the value to be displayed on third digit of SSD
4	the value to be displayed on fourth (leftmost) digit of SSD

- unsigned char dp - each of the 4 LSB bits contains the value corresponding to a decimal point of a SSD digit.

Bit	corresponds to
0 (LSB)	the digital point corresponding to the first (rightmost) digit of SSD
1	the digital point corresponding to the second digit of SSD
2	the digital point corresponding to the third digit of SSD
3	the digital point corresponding to the fourth (leftmost) digit of SSD

Description

This function sets the 4 values and 4 decimal points to be displayed on the 4 SSD digits. Each of the 4 bytes of val parameter contains the value to be displayed on a SSD digit (as detailed above), and each of the 4 LSB bits of dp parameter contains the value corresponds to decimal point of a SSD digit. If a value in the 4 bytes of val is outside 0 - 16, then the corresponding SSD digits will display nothing.

Example

```
#include “ssd.h”
…
SSD_WriteDigits(0x4321, 0x00); // displays “1234” on the SSD digits,
decimal points off
```

20.2.4 SSD_GetDigitSegments

Synopsis

unsigned char SSD_GetDigitSegments(unsigned char d)

Parameters

- unsigned char d1 - the value to be represented on first (rightmost) digit of SSD. Should be between 0 and 16, according to the digit value explained in the Overview chapter.

Return value

- unsigned char - value containing segments configuration. Bits 0-6 correspond to the seven segments:

Bit	corresponds to	Value
0 (LSB)	Segment A	0 for segment ON
		1 for segment off
1	Segment B	0 for segment ON
		1 for segment off
2	Segment C	0 for segment ON
		1 for segment off
3	Segment D	0 for segment ON

		1 for segment off
4	Segment E	0 for segment ON
		1 for segment off
5	Segment F	0 for segment ON
		1 for segment off
6	Segment G	0 for segment ON
		1 for segment off

- 0xFF If d is outside 0 - 16

Description

This function returns the byte containing the segments configuration for the digit received as parameter. If d is outside 0 - 16, then 0xFF is returned.

Example

```
#include “ssd.h”
…
SSD_WriteDigits(0x4321, 0x00); // displays “1234” on the SSD digits,
decimal points off
```

20.2.5 SSD_Timer1Setup

Synopsis

void SSD_Timer1Setup()

Description

This function configures the Timer1 to be used by SSD module.
The timer will generate interrupts every 3 ms. The period constant is computed using TMR_TIME definition (located in this source file) and peripheral bus frequency definition (PB_FRQ, located in config.h).
This is a low-level function called by SSD_Init(), so user should avoid calling it directly.

Example

```
#include “ssd.h”
SSD_Timer1Setup(); // initialize Timer1
```

20.2.6 SSD_ConfigurePins

Synopsis

void SSD_ConfigurePins()

Description

This function configures the IO pins involved in the SSD module as digital output pins.
The function uses pin related definitions from config.h file.
This is a low-level function called by SSD_Init(), so user should avoid calling it directly.

Example

```
#include “ssd.h”
SSD_ConfigurePins();
```

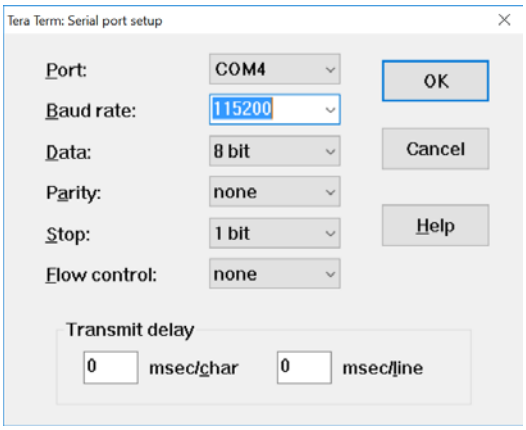
21 UART

21.1 Overview

This library implements the UART4 hardware interface of PIC32 functionality connected to the USB – UART interface labeled UART. It provides basic functions to configure UART and transmit / receive functions.

PC side of communication

Most of the time, the USB – UART interface is used for communicating with a PC. Connecting the USB connector labeled UART to PC will create a COM port, allowing a terminal to communicate with this UART interface. Set the baud rate according to the library initialization and leave the others with default values.



Free USB-COM port drivers, available from www.ftdichip.com, convert USB packets to UART/serial port data.

Receive functions on interrupt or polling

The UART receive can be done both with interrupts and with polling methods. Separate functions are provided for the two approaches, for example UART_GetString (retrieves a CR+LF terminated string using interrupt approach) and UART_GetStringPoll (retrieves a string using polling approach).

The two modes provide separate initialization functions: UART_Init (for interrupt approach) and UART_InitPoll (for polling approach).

In the interrupt approach, UART is configured so that it triggers receive interrupts. In the UART interrupt ISR, the received characters are placed into a buffer. When terminating CR+LF characters are detected, the UART interrupt ISR signals the availability of a CR+LF terminated string (using a global variable), so that subsequent call to UART_GetString will return the received string.

In the polling approach, the function UART_GetStringPoll checks the availability of a received character. While characters are available, they are placed in the received characters buffer.

Library files

The library is implemented in these files: uart.c, uart.h. Include them in the project.

Include the library header file whenever you want to use the library functions:

```
#include "uart.h"
```

21.2 Library functions

21.2.1 UART_Init

Synopsis

void UART_Init(unsigned int baud)

Parameters

- unsigned int baud - UART baud rate.
  for example 115200 corresponds to 115200 baud.

Description

This function initializes the hardware involved in the UART module, in the UART receive with interrupt mode.

The UART_TX digital pin is configured as digital output.

The UART_RX digital pin is configured as digital input.

The UART_TX and UART_RX are mapped over the UART4 interface.

The UART4 module of PIC32 is configured to work at the specified baud, no parity and 1 stop bit.

Example

```
#include “uart.h”
UART_Init(9600); // initialize UART in the receive interrupt mode, baud
9600, no parity and 1 stop bit
```

21.2.2 UART_InitPoll

Synopsis

void UART_InitPoll(unsigned int baud)

Parameters

- unsigned int baud - UART baud rate.
  for example 115200 corresponds to 115200 baud.

Description

This function initializes the hardware involved in the UART module, in the UART receive without interrupts (polling method).

The UART_TX digital pin is configured as digital output.

The UART_RX digital pin is configured as digital input.

The UART_TX and UART_RX are mapped over the UART4 interface.

The UART4 module of PIC32 is configured to work at the specified baud, no parity and 1 stop bit.

Example

```
#include “uart.h”
UART_InitPoll(9600);  // initialize UART in the receive polling mode,
baud 9600, no parity and 1 stop bit
```

21.2.3 UART_PutChar

Synopsis

void UART_PutChar(char ch)

Parameters

- char ch - the character to be transmitted over UART.

Description

This function transmits a character over UART4.

Example

```
#include “uart.h”
UART_PutChar(‘D’);    // transmits ‘D’ character over UART4
```

21.2.4 UART_PutString

Synopsis

void UART_PutString(char szData[])

Parameters

- char szData[] - the zero terminated string containing characters to be transmitted over UART.

Description

This function transmits all the characters from a zero terminated string over UART4. The terminator character is not sent.

Example

```
#include “uart.h”
UART_PutString(“Digilent”);// sends 8 characters : ‘D’, ‘i’, ‘g’, ‘i’, ‘l’, ‘e’, ‘n’, ‘t’
```

21.2.5 UART_AvaliableRx

Synopsis

unsigned char UART_AvaliableRx()

Return value

- unsigned char - Receive Buffer Data Available bit:

return value	When
0	Receive buffer is empty
1	Receive buffer has data, at least one more character can be read

Description

This function returns UART4 Receive Buffer Data Available bit.

It returns 1 if the receive buffer has data (at least one more character can be read).

It returns 0 if the receive buffer is empty.

Example

```
#include “uart.h”
if(UART_AvaliableRx())
{
     // if there is an available received character over UART
     c = UART_GetCharPoll();
}
```

21.2.6 UART_GetCharPoll

Synopsis

unsigned char UART_GetCharPoll()

Return value

- unsigned char - the byte received over UART

Description

This function waits until a byte is received over UART4. Then, it returns the byte.

This implements the polling method of receive one byte.

It returns 0 if the receive buffer is empty.

Example

```
#include “uart.h”
c = UART_GetCharPoll();     // wait until a character is received over
UART and return it
```

21.2.7 UART_GetStringPoll

Synopsis

unsigned char UART_GetStringPoll(unsigned char *pText)

Parameters

- unsigned char *pText - Pointer to a buffer to store the received bytes

Return value

- unsigned char - the received status

return value	When
0	no received bytes are available
1	at least one received byte is available, the received characters are placed in pText buffer

Description

This function returns a zero terminated string received over UART4, using polling method.

While a received bytes is available, this function calls repeatedly UART_GetCharPoll until no values are received over UART4.

It returns 0 if no received bytes are available, and returns 1 if at least one byte was received.

Example

```
#include “uart.h”
char buff[100];
UART_GetStringPoll(buff);
```

21.2.8 UART_GetString

Synopsis

unsigned char UART_GetString(char* pchBuff, int cchBuff)

Parameters

- char* pchBuff - pointer to a char buffer to hold the received zero terminated string
- int cchBuff - size of the buffer to hold the zero terminated string

Return value

- unsigned char - the received status

return value	When
>0	the number of characters contained in the string
0	a CR+LF terminated string hasn't been received
-2	a buffer underrun occurred (user buffer not large enough)
-3	an invalid (0 char count) CR+LF terminated string was received

Description

This function returns a zero terminated string to be received over UART4, received using interrupt method. The interrupt recognizes a string having up to cchRxMax - 2 characters, followed by a carriage return and a line feed ("\r\n", CRLF). The CRLF is stripped from the string and a NULL character ('\0') is appended. The number of characters contained in the zero terminated string is returned as a value greater than 0.

Example

```
#include “uart.h”
char buff[100];
UART_GetString(buff, 100);
```

21.2.9 UART_ConfigureUart

Synopsis

void UART_ConfigureUart(unsigned int baud)

Parameters

- unsigned int baud - UART baud rate.
  for example 115200 corresponds to 115200 baud.

Description

This function configures the UART4 hardware interface of PIC32, according to the provided baud rate, no parity and 1 stop bit, with no interrupts.

In order to compute the baud rate value, it uses the peripheral bus frequency definition (PB_FRQ, located in config.h).

This is a low-level function called by UART_Init(), so user should avoid calling it directly.

Example

```
#include “uart.h”
UART_ConfigureUart(9600);  // initialize UART in the receive polling
mode, baud 9600, no parity and 1 stop bit
```

21.2.10 UART_ConfigureUartRXInt

Synopsis

void UART_ConfigureUartRXInt(unsigned int baud)

Parameters

- unsigned int baud - UART baud rate.
  for example 115200 corresponds to 115200 baud.

Description

This function configures the UART4 hardware interface of PIC32, according to the provided baud rate, no parity and 1 stop bit, with no interrupts (by calling the UART_ConfigureUart function) and additionally configures the interrupt on RX.

This is a low-level function called by initialization functions, so user should avoid calling it directly.

Example

```
#include “uart.h”
UART_ConfigureUartRXInt(9600);  // initialize UART in the receive
polling mode, baud 9600, no parity and 1 stop bit
```

21.2.11 UART_ConfigurePins

Synopsis

void UART_ConfigurePins()

Description

This function configures the digital pins involved in the UART module:
The UART_TX digital pin is configured as digital output.

The UART_RX digital pin is configured as digital input.

The UART_TX and UART_RX are mapped over the UART4 interface.

The function uses pin related definitions from config.h file.

This is a low-level function called by UART_Init(), so user should avoid calling it directly.

Example

```
#include “uart.h”
UART_ConfigurePins();
```

21.2.12 UART_Close

Synopsis
void UART_Close()

Description

This functions releases the hardware involved in UART library: it turns off the UART4 interface.

Example

```
#include “uart.h”
UART_Close();   // close UART module
```

22 UARTJB

22.1 Overview

This library implements the UART1 (hardware interface of PIC32) functionality over the PMODB pins:

Pmod pin	Schematic name	PIC32 pin	UART mapped pin
PMODB_2	JB2	RPD11/PMCS1/RD11	TX1
PMODB_3	JB3	RPD10/PMCS2/RD10	RX1

The library provides basic functions to configure UART (including pin mapping) and transmit / receive functions.

This setup is useful for the Basys MX3 board to communicate with another UART device, using two wires UART.

Receive functions on interrupt or polling

The UART receive can be done both with interrupts and with polling methods. Separate functions are provided for the two approaches, for example UARTJB_GetString (retrieves a CR+LF terminated string using interrupt approach) and UARTJB_GetStringPoll (retrieves a string using polling approach).

The two modes provide separate initialization functions: UARTJB_Init (for interrupt approach) and UARTJB_InitPoll (for polling approach).

In the interrupt approach, UARTJB is configured so that it triggers receive interrupts. In the UARTJB interrupt ISR, the received characters are placed into a buffer. When terminating CR+LF characters are detected, the UARTJB interrupt ISR signals the availability of a CR+LF terminated string (using a global variable), so that subsequent call to UARTJB_GetString will return the received string.

In the polling approach, the function UARTJB_GetStringPoll checks the availability of a received character. While characters are available, they are placed in the received characters buffer.

Library files

The library is implemented in these files: uart.c, uart.h. Include them in the project.

Include the library header file whenever you want to use the library functions:

```
#include "uart.h"
```

22.2 Library functions

22.2.1 UARTJB_Init

Synopsis

void UARTJB_Init(unsigned int baud)

Parameters

- unsigned int baud - UART baud rate.
  for example 115200 corresponds to 115200 baud.

Description

This function initializes the hardware involved in the UARTJB module, in the UART receive with interrupt mode.

The JB2 digital pin is configured as digital output, and mapped over U1TX.

The JB3 digital pin is configured as digital input, and mapped over U1RX.

The UART1 module of PIC32 is configured to work at the specified baud, no parity and 1 stop bit.

Example

```
#include “uartjb.h”
UARTJB_Init(9600);    // initialize UARTJB in the receive interrupt
mode, baud 9600, no parity and 1 stop bit
```

22.2.2 UARTJB_InitPoll

Synopsis

void UARTJB_InitPoll(unsigned int baud)

Parameters

- unsigned int baud - UART baud rate.
  for example 115200 corresponds to 115200 baud.

Description

This function initializes the hardware involved in the UARTJB module, in the UART receive without interrupts (polling method).

The JB2 digital pin is configured as digital output, and mapped over U1TX.

The JB3 digital pin is configured as digital input, and mapped over U1RX.

The UART1 module of PIC32 is configured to work at the specified baud, no parity and 1 stop bit.

Example

```
#include “uartjb.h”
UARTJB_InitPoll(9600);      // initialize UARTJB in the receive polling mode, baud 9600, no parity and 1 stop bit
```

22.2.3 UARTJB_PutChar

Synopsis

void UARTJB_PutChar(char ch)

Parameters

- char ch - the character to be transmitted over UART.

Description

This function transmits a character over UART1.

Example

```
#include “uartjb.h”
UARTJB_PutChar(‘D’);  // transmits ‘D’ character over UART1
```

22.2.4 UARTJB_PutString

Synopsis

void UARTJB_PutString(char szData[])

Parameters

- char szData[] - the zero terminated string containing characters to be transmitted over UART.

Description

This function transmits all the characters from a zero terminated string over UART1. The terminator character is not sent.

Example

```
#include “uartjb.h”
UARTJB_PutString(“Digilent”);    // sends 8 characters : ‘D’, ‘i’, ‘g’, ‘i’, ‘l’, ‘e’, ‘n’, ‘t’
```

22.2.5 UARTJB_AvaliableRx

Synopsis

unsigned char UARTJB_AvaliableRx()

Return value

- unsigned char - Receive Buffer Data Available bit:

return value	When
0	Receive buffer is empty
1	Receive buffer has data, at least one more character can be read

Description

This function returns UART1 Receive Buffer Data Available bit.

It returns 1 if the receive buffer has data (at least one more character can be read).

It returns 0 if the receive buffer is empty.

Example

```
#include “uartjb.h”
if(UARTJB_AvaliableRx())
{
     // if there is an available received character over UART
     c = UARTJB_GetCharPoll();
}
```

22.2.6 UARTJB_GetCharPoll

Synopsis

unsigned char UARTJB_GetCharPoll()

Return value

- unsigned char - the byte received over UART

Description

This function waits until a byte is received over UART1. Then, it returns the byte.

This implements the polling method of receive one byte.

It returns 0 if the receive buffer is empty.

Example

```
#include "uartjb.h"
c = UARTJB_GetCharPoll();  // wait until a character is received over
UART and return it
```

22.2.7 UARTJB_GetStringPoll

Synopsis

unsigned char UARTJB_GetStringPoll(unsigned char *pText)

Parameters

- unsigned char *pText - Pointer to a buffer to store the received bytes

Return value

- unsigned char - the received status

return value	When
0	no received bytes are available
1	at least one received byte is available, the received characters are placed in pText buffer

Description

This function returns a zero terminated string received over UART1, using polling method.

While a received bytes is available, this function calls repeatedly UARTJB_GetCharPoll until no values are received over UART1.

It returns 0 if no received bytes are available, and returns 1 if at least one byte was received.

Example

```
#include "uartjb.h"
char buff[100];
UARTJB_GetStringPoll(buff);
```

22.2.8 UARTJB_GetString

Synopsis

unsigned char UARTJB_GetString(char* pchBuff, int cchBuff)

Parameters

- char* pchBuff - pointer to a char buffer to hold the received zero terminated string
- int cchBuff - size of the buffer to hold the zero terminated string

Return value

- unsigned char - the received status

return value	When
>0	the number of characters contained in the string
0	a CR+LF terminated string hasn't been received
-2	a buffer underrun occurred (user buffer not large enough)
-3	an invalid (0 char count) CR+LF terminated string was received

Description

This function returns a zero terminated string to be received over UART1, received using interrupt method. The interrupt recognizes a string having up to cchRxMax - 2 characters, followed by a carriage return and a line feed ("\r\n", CRLF). The CRLF is stripped from the string and a NULL character ('\0') is appended. The number of characters contained in the zero terminated string is returned as a value greater than 0.

Example

```
#include “uartjb.h”
char buff[100];
UARTJB_GetString(buff, 100);
```

22.2.9 UARTJB_ConfigureUart

Synopsis

void UARTJB_ConfigureUart(unsigned int baud)

Parameters

- unsigned int baud - UART baud rate.
  for example 115200 corresponds to 115200 baud.

Description

This function configures the UART1 hardware interface of PIC32, according to the provided baud rate, no parity and 1 stop bit, with no interrupts.

In order to compute the baud rate value, it uses the peripheral bus frequency definition (PB_FRQ, located in config.h).

This is a low-level function called by UARTJB_Init(), so user should avoid calling it directly.

Example

```
#include “uartjb.h”
UARTJB_ConfigureUart(9600);// initialize UARTJB in the receive polling mode, baud 9600, no parity and 1 stop bit
```

22.2.10 UARTJB_ConfigureUartRXInt

Synopsis
void UARTJB_ConfigureUartRXInt(unsigned int baud)
Parameters

- unsigned int baud - UART baud rate.
  for example 115200 corresponds to 115200 baud.

Description

This function configures the UART1 hardware interface of PIC32, according to the provided baud rate, no parity and 1 stop bit, with no interrupts (by calling the UARTJB_ConfigureUart function) and additionally configures the interrupt on RX.

This is a low-level function called by initialization functions, so user should avoid calling it directly.

Example

```
#include “uartjb.h”
UARTJB_ConfigureUartRXInt(9600); // initialize UARTJB in the receive
polling mode, baud 9600, no parity and 1 stop bit
```

22.2.11 UARTJB_ConfigurePins

Synopsis
void UARTJB_ConfigurePins()
Description
This function configures the digital pins involved in the UARTJB module:
The JB2 digital pin is configured as digital output, and mapped over U1TX.

The JB3 digital pin is configured as digital input, and mapped over U1RX.

The function uses pin related definitions from config.h file.

This is a low-level function called by UARTJB_Init(), so user should avoid calling it directly.
Example

```
#include “uartjb.h”
UARTJB_ConfigurePins();
```

22.2.12 UARTJB_Close

Synopsis
void UARTJB_Close()

Description

This functions releases the hardware involved in UARTJB library: it turns off the UART1 interface.

Example

```
#include “uartjb.h”
UARTJB_Close(); // close UARTJB module
```

23 Utils

23.1 Overview

This library implements the delay functionality used in other libraries. The delay is implemented using loop, so the delay time is not exact. For exact timing use timers.

Include the file in the project, together with utils.h, when this when this library is needed.

Library files

The library is implemented in these files: utils.c, utils.h. Include them in the project.

Include the library header file whenever you want to use the library functions:

```
#include "utils.h"
```

23.2 Library functions

23.2.1 DelayAprox10Us

Synopsis
void DelayAprox10Us (unsigned int t100usDelay)
Parameters

- unsigned int t100usDelay- the amount of time you wish to delay in hundreds of microseconds

Description

This procedure delays program execution for the specified number of microseconds. This delay is not precise.

This routine is written with the assumption that the system clock is 40 MHz.

Example

```
#include “utils.h”
DelayAprox10Us(100);  // configure a delay of 100us
```

24 Hardware resources map

Library	Module
SSD	Timer1
SRV	Timer2
MOT (in PH/EN mode)	Timer3
RGBLED	Timer5
UART	UART4
UARTJB	UART1
SRV	OC4
SRV	OC5
MOT	OC2
MOT	OC3
AUDIO	Timer3

Commented functionality

Library	Action
RGBLED	Timer2, OC3, OC4 and OC5

25 Basys MX3 LibPack versions

Version	Release date	Note
1.0	April 14, 2017	All libraries source files have this datetime: “04/13/2017 04:14 PM”
1.1	June 20, 2017	All libraries source files have this datetime: “06/20/2017 03:59 PM”