LimiFrog Libraries

March, 2016

Release 0.1

Introduction -

The libraries developed for LimiFrog include board support functions as well as a set of STM32 peripheral abstraction functions that build upon either the HAL library or the low-level drivers provided by ST. They also help use the following middlewares : USB, Fat-FS file system, emWin graphics.
The source files can be found under *LimiFrog_SW/libraries/LimiFrog-Lib/*

LimiFrog users are free to use or not this package, or possibly part of it – for example just the board initialization steps.

Specifically, the libraries described here perform:

1- Full board initialization :

→ set-up of all STM32 I/Os and peripherals dedicated to the control of on-board hardware, initialization of on-board chips where required. After running this initialization sequence, the user can immediately access the on-board hardware.

2- Control and exploitation of on-board hardware :

→ for example, simple I2C access to a MEMS sensor ;

→ or, at higher level, API for sensors, etc ;

3- Abstraction of selected STM32 peripherals at a higher level than the HAL library :

→ for example, simplified usage of PWM

4- Support of ST-provided middleware (USB, File System, Graphics Lib) :

→ essentially, «bridging » the generic middleware with the actual hardware and initializing the middleware

The functions documented below are those that may be of interest to a programmer. Some of them rely on lower-level functions which are not described, as not intended to be used directly by the programmer.

Table of Contents

1 PRELIMINARY NOTES

1.1 Global Variables

The LimiFrog library makes use of a few global variables. Specifically, they are structures that are used to configure peripherals dedicated to controlling on-board hardware : I2C2 (dedicated to communication with the MEMS sensors), SPI1 (OLED display), SPI3 (data Flash) and UART3 (BLE module), as well as the USB preipheral in device mode.

> Accesses to the first four peripherals (I2C2, SPI1, SPI3, UART3), therefore to these structures, are performed throughout the code. Therefore it has been decided to declare these structure as global and to declare them as « extern » in a heder file ***LBF_global_variables.h*** which can conveniently be included in any source file that needs acces to these variables. The path to *LBF_global_variables.h* is : *LimiFrog_SW/LimiFrog-lib/inc/LBF_Defs.*

This header file gets pulled by header file ***LBF_Global.h***. Therefore, if the user includes *LBF_Global.h* in his application files he can directly use these global constants without declaring them in his code.

These variables involved are the following :

```
// for I2C2: used on LimiFrog for access to all sensors
      extern I2C_HandleTypeDef hi2c2;     // defined in LBF_I2C2_Init.c
// for SPI1: used on LimiFrog for access to the OLED display
      extern SPI_HandleTypeDef hspi1;     // defined in LBF_SPI1_Init.c
// for SPI3: used on LimiFrog for access to the Data Flash
      extern SPI_HandleTypeDef hspi3;     // defined in LBF_SPI3_Init.c
// for UART3: used on LimiFrog for access to the BlueTooth4.1 (BLE) module
      extern UART_HandleTypeDef huart3; // defined in LBF_USART3_Init.c
```

> In addition :

a driver called PCD (Peripheral Controller Driver) bridges the USB middleware with the USB peripheral hardware. The user hardly needs to know about this lower level driver, but needs to implementing a call to it in the USB interrupt service routine – see OTG_FS_IRQHandler() in *stm32_it.c* template file – where he passes a handle « *hpcd* » to a specific structure, defined in file *usbd_conf.c* as a global variable. For convenience this global varialve is also declared as *extern* in header file *LBF_global_variables.h*. Since the latter gets included in *stm32_it.c,* this saves the user the need to write this declaration in the stm32_it.c file of each project he creates. The variable involved is the following :

```
// PCD for access to USB in device mode
      extern PCD_HandleTypeDef hpcd;      // defined in usbd_conf.c
```

1.2 Custom Types

The LimiFrog library defines a few custom types.

- Those which are used exclusively in a specific file are defined in the header file associated to this file. These custom types are documented along with the functions that use them, further down this document.

- A few custom types are used used to hold the parameters defined by the user in file *User_Configuration.h*, and exploited at several places across the LimiFrog library. These types are defined in file *LBF_custom_types.h, t*he path to which is : *LimiFrog_SW/LimiFrog-lib/inc/LBF_Defs*

1.3 Aliases

File ***LBF_pin_aliases.h*** provides a number of meaningful aliases to designate a specific GPIO of the STM32 according to its rôle in LimiFrog. For example pin GPIO_PIN_7 on port GPIOB (a.k.a PB7) drives the CS input of the data flash on LimiFrog, therefore in file *LBF_pin_aliases.h,* FLASH_CS_PIN is aliased to GPIO_PIN_7 and FLASH_CS_PORT is aliased as GPIOB. The path to *LBF_pin_aliases.h* is *LimiFrog_SW/LimiFrog-lib/inc/LBF_Defs.*

File ***LBF_OnBoard_chip_aliases.h*** provides some aliases for the I2C 7-bit address or « chip ID » of the chips on the LimiFrog board that can be accessed by I2C. For each of them,it also defines an alias for the unique ID register found in each of these chips ((xxx_WHOAMI) and for the expected contents The path to *LBF_OnBoard_chip_aliases.h* is : *LimiFrog_SWLimiFrog-lib/inc/LBF_Defs*

2 FULL BOARD INITIALIZATION

Library files found under: LimiFrog_SW/`libraries/LimiFrog_Lib/xxx/LBF_Board_Inits`

These functions takes care of configuring the STM32 I/Os and on-chip peripherals that are dedicated to communication with on-board hardware and with the extension port.

- **Function** : `void` **`LBF_Board_Inits`**`(void)`
- **Description** : Performs all the low-level hardware intializations required to have the LimiFrog board up and running : set-up of all clocks, NVIC initialization, set-up of all I/Os and STM32 peripherals dedicated to on-board communications, etc.
- **Parameters** : -
- **Return Value**: -
- **Note**: this function requires a valid *User_Configuration.h* file to be present in the list of include files (typically under /*LimiFrog_SW/projects/xxx/inc*, see the proposed template configuration file *…/projects/LimiFrog_PROJECT_TEMPLATE* for guidance), to drive the « selective » part of the

intialization process.

More details :

Function *LBF_Board_Inits()* performs all « fixed » board initializations (those which do not depend on user choices) as well as « selective » initializations (which depend on some user choices, to be expressed as directives in a user configuration file « *User_Configuration.h »).*

Fixed Initializations are performed through a call to ***`LBF_Board_Fixed_Inits()`*** and include :

> setting up the Flash instruction and data cache as well as the Flash prefetch buffer

> setting up the Nested Vector Interrupt Controller of the ARM Cortex core (NVIC) to work with pre-emption priority only, no sub-priority.

> configuring all STM32 internal clocks according to the scheme described in Appendix 1
> enabling all GPIO clocks
> configuring all STM32 I/Os that deal with interfacing with the on-board hardware
> initilializing in a suitable mode those STM32 peripherals dedicated to controlling on-board hardare :
- SPI1 for the OLED display,
- SPI3 for the Data Flash IC,
- I2C2 for the MEMS sensors
> initializing the high-precision low-speed external oscillator (32.768KHz) and setting up the RTC to use this clock source

Selective initializations are performed through a call to ***`LBF_Board_Selective_Inits()`*** and include :

> if parameter *USE_OLED* is defined in file *User_Configuration.h* :
configuring the OLED display controller so the OLED is ready for use (high voltage for the panel itself not yet enabled at this stage)
> if parameter *ENABLE_BTLE* is defined in file *User_Configuration.h* :
- retrieving in the data flash file system an exectuble file of specified by BTLE_CODE_FILENAME in file *User_Configuration.h,*
- downloading this firwmare into the BTLE module
- and finally configuring UART3 and associated STM32 I/Os to be ready to communicate with the BLE module, using a baud rate specified by parameter UART_BTLE_BAUDRATE in file *User_Configuration.h*

> configuring the STM32 I/Os that are routed to the extension port, according to directives given by the user in file *User_Configuration.h*

3 ON-BOARD HARDWARE CONTROL

Library files found under: LimiFrog_SW/`libraries/LimiFrog_Lib/xxx/LBF_API`

These functions are provided to interact with on-board hardware in a simple way. They typically rely on peripherals dedicated to this rôle and already initialized when calling `LBF_Board_inits()`.

3.1 LED

Library files found under:

LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_LED_Switches_lowlevAPI.xx

- **Function** : `void LBF_Led_ON(void)`
- **Description** : Turns on the STM32-controlled LED located at the edge of the board
- **Parameters** : -
- **Return Value:** -

- **Function** : `void LBF_Led_OFF(void)`
- **Description** : Turns off the STM32-controlled LED located at the edge of the board
- **Parameters** : -
- **Return Value:** -

- **Function** : `void LBF_Led_TOGGLE(void)`
- **Description** : Toggles the STM32-controlled LED located at the edge of the board
- **Parameters** : -
- **Return Value:** -

3.2 Push-button switch

Library files found under:

LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_LED_Switches_lowlevAPI.xx

- **Function** : `bool LBF_State_Switch1_IsOn(void)`
- **Description** : Returns the state of the general-purpose push-button switch located at the edge of board
- **Parameters** : -
- **Return Value:** true if button being pushed, else false

3.3 Battery

Library files found under: LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_xxx.xx

//TODO

- **Function** : `uint32_t` **`LBF_Get_Battery_Voltage_mV`**`(void)`
- **Description** : Measures the battery voltage (routed to STM32 pin and measured by ADC) and returns the result in mV
- **Parameters** : -
- **Return Value:** Measured battery voltage, in millivolt
- Note : this function includes intialiazing and setting up the ADC and then de-initializing after mesurement is done. To be kept in mind if ADC is also used for other purposes.

3.4 OLED Display

Library files found under:
LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_OLED_lowlevAPI.xx

***Note* : For displaying graphics and characters, the OLED Display can be controlled at a higher level with much more flexibility through the emWin graphics library, if the application can afford it (more memory-consuming). See section 'Middleware' further down that document.**

- **Function** : `void` **`LBF_OLED_Switch_ON`** `(void)`
- **Description** : Provides high voltage to OLED panel (13V) and turn display on with controlled latency as per display datasheet
- **Parameters** : -
- **Return Value:** -

- **Function** : `void` **`LBF_OLED_Switch_OFF`** `(void)`
- **Description** : Turns display off and switches high voltage off with controlled latency as per display datasheet
- **Parameters** : -
- **Return Value:** -

- **Function** : `void` **`LBF_OLED_Clear`** `(void)`

- **Description** : Displays a uniform black screen
- **Parameters** : -
- **Return Value:** -

- **Function** : `void` **`LBF_OLED_Brightness`**`(tbd)`
- **Description** : Changes the brightness of the OLED display.
- **Parameters** : tbd
- **Return Value:** -

  <*TO DO* >

- **Function** : `void` **`LBF_OLED_Fill`**`(uint8_t x, uint8_t y, uint8_t width, uint8_t height, uint16_t color565)`
- **Description** : Fills (in horizontal raster mode) the specified rectangle with the specified color, expressed in RGB565 format (16 bits)
- **Parameters** :

  > *x* : horizontal coordinate of top-left pixel of fill region (x grows from left to right)
  > *y* : vertical coordinate of top-left pixel of fill region (y grows from top to bottom)
  > *width* : width in pixels of the region to fill
  > *height* : height in pixels of the region to fill
  > *color565* : color in RGB565 format that will be used to to fill the rectangle
- **Return Value:** -
- **Note :** alternatively, use `LBF_OLED_Fill_XY allows` to specify coordinates rather than height and width

- **Function** : `void` **`LBF_OLED_Fill_XY`**`(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, uint16_t color565)`
- **Description** : Fills (in horizontal raster mode) the specified rectangle with the specified color, expressed in RGB565 format (16 bits)
- **Parameters** :

  > *x1* : horizontal coordinate of top-left pixel of fill region (x grows from left to right)
  > *y1*: vertical coordinate of top-left pixel of fill region (y grows from top to bottom)
  > *x2* : horizontal coordinate of bottom-right pixel of fill region (x grows from left to right)
  > *y2*: vertical coordinate of bottom-right pixel of fill region (y grows from top to bottom)

> *color565* : color in RGB565 format that will be used to to fill the rectangle

- **Return Value:** -
- **Note :** alternatively, `LBF_OLED_Fill allows` to specify height and width rather than x2, y2

- **Function** : `void` **`LBF_OLED_DisplayBuffer`**`(uint8_t x, uint8_t y, uint8_t width, uint8_t height, uint16_t *buffer)`
- **Description** : Fills (in horizontal raster mode) the specified rectangle with RGB565 pixels stored in a buffer
- **Parameters** :

> *x* : horizontal coordinate of top-left pixel of fill region (x grows from left to right)
> *y* : vertical coordinate of top-left pixel of fill region (y grows from top to bottom)
> *width* : width in pixels of the region to fill
> *height* : height in pixels of the region to fill
> *buffer* : pointer to a buffer of pixes in RGB565 format that will be used to to fill the region

- **Return Value:** -
- **Note :** alternatively, `LBF_OLED_DisplayBuffer_XY allows` to specify coordinates rather than height and width

- **Function** : `void` **`LBF_OLED_DisplayBuffer_XY`**`(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, uint16_t *buffer)`
- **Description** : Fills (in horizontal raster mode) the specified rectangle with RGB565 pixels stored in a buffer
- **Parameters** :

> *x1* : horizontal coordinate of top-left pixel of fill region (x grows from left to right)
> *y1* : vertical coordinate of top-left pixel of fill region (y grows from top to bottom)
> *x2* : horizontal coordinate of bottom-right pixel of fill region
> *y2* : vertical coordinate of bottom-right pixel of fill region
> *buffer* : pointer to a buffer of pixes in RGB565 format that will be used to to fill the region

- **Return Value:** -
- **Note :** alternatively, `LBF_OLED_DisplayBuffer allows` to specify width and height rather than x2 and y2

******* IMPORTANT NOTE : The following 4 functions actually encapsulate emWin functtions, which means this middleware must have been enabled (see LBF_emWin_Init() in section on Middlewares).***

The interest is to provide very simple printf-like means to display messages on the screen ****

- **Function** : `void` **`LBF_OLED_PrintString`**`(char* string)`
- **Description** : Displays the specified string on the OLED, using default or the latest specified GUI settings. If bottom of screen was reached, clears the screen and starts back at top of screen.
- **Parameters** : *string* : pointer to the string of characters to be displayed
- **Return Value:** -
- **Note** : will hang if emWin Gfx middleware has not been initialized (using e.g. `LBF_emWin_Init()`, see section on middlewares)

- **`Function :`** `void` **`LBF_OLED_PrintDec`**`(int32_t SignedInteger)`
- **Description** : Displays the specified signed 32-bit integer in decimal format on the OLED, using default or the latest specified GUI settings. If bottom of screen was reached, clears the screen and starts back at top of screen.
- **Parameters** : *SignedInteger* : the signed integer value to display in decimal format
- **Return Value:** -
- **Note** : will hang if emWin Gfx middleware has not been initialized (using e.g. `LBF_emWin_Init()`, see section on middlewares)

- **Function** : `void` **`LBF_OLED_PrintHex`**`(uint16_t Unsigned16)`
- **Description** : Displays the specified unsigned 16-bit integer in hexadecimal format on the OLED, using default or the latest specified GUI settings. If bottom of screen was reached, clears the screen and starts back at top of screen.
- **Parameters** : *Unsigned16* : the 16-bit value to display in hexadecimal format
- **Return Value:** -
- **Note** : will hang if emWin Gfx middleware has not been initialized (using e.g. `LBF_emWin_Init()`, see section on middlewares)

- **Function** : `void` **`LBF_OLED_Overwrite_CurrentLine`**`(void)`
- **Description** : Rewinds screen pointer to the start of the current line and erases it.

- **Parameters** : -
- **Return Value:** -
- **Note** : will hang if emWin Gfx middleware has not been initialized (using e.g. `LBF_emWin_Init()`, see section on middlewares)

******* Next are functions that may be used for low-level interface to the OLED display controller *******

- **Function** : `void` **`LBF_OLED_SendCmd`** `(uint8_t Value)`
- **Description** : Sends a command byte over SPI to the OLED. Signal CS and RS are managed (to enable bus and signal a command type value)
- **Parameters** :
  > *Value* : the 8-bit value to send
- **Return Value:** -

- **Function** : `void` **`LBF_OLED_WriteReg`** `(uint8_t RegAdd, uint8_t RegValue)`
- **Description** : Writes the specified data byte into the specified register of the OLED controller using the SPI interface.
- **Parameters** :
  > *RegAdd* : The address of the register to write
  > *RegValue* : The value to write
- **Return Value:** -

- **Function** : `void` **`LBF_OLED_SPI1_16bTransferStream_DMA1Ch3`** `(uint16_t* pTxBuffer, uint32_t TxLength)`
- **Description** : Sends a data stream over SPI1 to the OLED. DMA1 peripheral is supposed to have been initialized, with SPI data frame at 16bits (if using *LBF_Board_Init()* to initialize the board, this was catered for when initializing SPI1). Signals CS and RS to OLED are not touched.
- **Parameters** : *Value* : the 8-bit value to send
- **Return Value:** -

3.5 Data Flash

Library files found under:

LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_FLASH_lowlevAPI.xx

Do not confuse the Data Flash, a 64Mbit chip dedicated to data storage on LimiFrog, with the embedded Flash memory of the STM32 (program memory).

***Note* : the Data Flash can also be used as a FAT File System, through the use of the Fat-Fs library. See section on middleware further down this document.**

- **Function** : `void LBF_FLASH_EraseBulk(void)`
- **Description** : Electrically erases the full Data Flash (this takes some time)
- **Parameters** : -
- **Return Value:** -

- **Function** : uint8_t `LBF_FLASH_SendByte(uint8_t TxByte)`
- **Description** : Transfers a byte to the Data Flash over SPI in full-duplex mode, simultaneously receive a byte from the Flash (dummy or useful depending on context)
- **Parameters** : *TxByte* : the byte to send to the Data Flash
- **Return Value:** the byte received from the Flash

- **Function** : `void LBF_FLASH_WriteBuffer(uint8_t* pBuffer, uint32_t WriteAddr, uint32_t NumByteToWrite)`
- **Description** : Writes into the Data Flash, starting at the specified adress, the specified number of data bytes located in the buffer specified through a pointer to its head. This function takes care of chopping the operation into a number of a successive page accesses.
- **Parameters** :
  > *pBuffer* : pointer to the buffer that contains the data bytes to write into the Data Flash
  > *WriteAddr* : First address to write in the Data Flash
  > *NumByteToWrite* : Number of consecutive bytes to write into the Data Flash
- **Return Value:** -

- **Function** : `void LBF_FLASH_ReadBuffer(uint8_t* pBuffer, uint32_t ReadAddr, uint32_t NumByteToRead)`

- **Description** : Reads from the Data Flash, starting at the adress specified by *ReadAddr*, the number of data bytes specified by *NumByteToRead* and places them in a buffer pointer by *pBuffer*.
- **Parameters** :
  > *pBuffer* : pointer to the buffer for storing the data bytes read from the Data Flash
  > Read*Addr* : First address to read in the Data Flash
  > *NumByteToRead* : Number of consecutive bytes to read from the Data Flash
- **Return Value:** -

- **Function** : `uint32_t` **`LBF_FLASH_ReadID`**`(void)`
- **Description** : Returns the Unique ID read from the Data Flash
- **Parameters** : -
- **Return Value:** Unique ID of the Data Flash

3.6 Sensors

> Sensors can be controlled at low level using I2C bus #2 (which is dedicated to sensor control on LimiFrog).

Library files found under:
LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_I2Csensors_lowlevAPI.xx

Meaningful aliases for all sensor registers are available in the header files present under :
/LimiFrog_SW/libraries/Sensor_APIs/xxx/inc

> They can also be controlled using IC vendor-supplied API. These drivers are being integrated into the LimiFrog API
<TO DO >

> Finally, a number of simple, high-level functions will be provided to obtain sensors results in a simple way for selected use cases (e.g. single shot acquisition, etc.)
<TO DO >

- **Function** : `void` **`LBF_I2CSensors_WriteSingleReg`** `(uint8_t ChipID, uint16_t RegAdd, uint8_t RegVal)`
- **Description** : Writes the specified value at the specified address of the sensor identified by I2C

address *ChipID* (transactions occur over I2C bus #2, dedicated to sensors on LimiFrog).

- **Parameters** :
  > *ChipID* : 7-bit I2C address of the target chip
  > *RegAdd* : Address of the target register in the target chip's memory map
  > *RegVal* : Register value to write
- **Return Value:** -

- **Function** : `void` **`LBF_I2C_Sensors_WriteMultipleReg`** `(uint8_t ChipID, uint16_t RegAdd, uint8_t* pVal, uint16_t NumByteToWrite )`
- **Description** : Write the sequence of *NumByteToWrite* values located in a buffer pointed by *pVal* into consecutive addresses starting at address *RegAdd* of the sensor identified by I2C address *ChipID* (transactions occur over I2C bus #2, dedicated to sensors on LimiFrog).
- **Parameters** :
  > *ChipID* : 7-bit I2C address of the target chip
  > *RegAdd* : Address of the first target register in the target chip's memory map
  > p*Val* : pointer to a buffer that contains the sequence of values to write
  > *NumByteToWrite :* number of consecutive bytes to write into the target chip
- **Return Value:** -

- **Function** : `uint8_t` **`LBF_I2C_Sensors_ReadSingleReg`** `(uint8_t ChipID, uint16_t RegAdd)`
- **Description** : Return value read from address *RegAdd* of the sensor identified by I2C address *ChipID* (transactions occur over I2C bus #2, dedicated to sensors on LimiFrog).
- **Parameters** :
  > *ChipID* : 7-bit I2C address of the target chip
  > *RegAdd* : Address of the target register in the target chip's memory map
- **Return Value:** the value read from the target register

- **Function** : `void` **`LBF_I2C_Sensors_ReadMultipleReg`** `(uint8_t ChipID, uint16_t RegAdd, uint8_t* pVal, uint16_t NumByteToRead )`
- **Description** : Read the sequence of *NumByteToRead* values starting at address *RegAdd* of the sensor identified by I2C address *ChipID* and place them in a buffer pointed by pVal (transactions occur over I2C bus #2, dedicated to sensors on LimiFrog).
- **Parameters** :
  > *ChipID* : 7-bit I2C address of the target chip
  > *RegAdd* : Address of the first target register in the target chip's memory map

> p*Val* : pointer to a buffer where to store the sequence of values read back from the target chip
> *NumByteToRead :* number of consecutive bytes to read from the target chip

- **Return Value:** -

- **Function** : `void` **`LBF_I2C_Sensors_RmodWSingleReg`** `(uint8_t ChipID, uint16_t RegAdd, uint8_t RegMask, uint8_t RegUpdateVal)`
- **Description** : Read value *RegVal* from address *RegAdd* of the sensor identified by I2C address *ChipID,* overwrite bits indicated by *RegMask* (positions with bit set at 1 in *RegMask)* with corresponding bits of *RegUpdateVal*, and write back the result at *RegAdd* (transactions occur over I2C bus #2, dedicated to sensors on LimiFrog).
- **Parameters** : -
- **Return Value:** -

3.7 BlueTooth Low-Energy

Upon power-up, the Blue Tooth Low-Energy module (which embeds a Cortex-M0) does not contain any firmware. The LimiFrog initialization function described in section 2. of this document can handle downloading a firmware into the BlueTooth module as part of the board initialization sequence.

For this, the user simply sets a couple of directives in configuration file *User_Configuration.h* to enable BTLE and to indicate under what name he has stored the binary firmware file on the File System (Data Flash).
Note : Copying this firmware from a PC onto the (Data Flash) File System can be done by connecting LimiFrog to PC through USB, with LimiFrog running a program that enables USB device mass storage.

The initialization function then takes care of downloading (over UART3, automatically set up to that aim) that firmware into the BlueTooth module RAM. The BlueTooth module will boot from there.

The STM32 can then exchange with the BTLE module over UART #3.

Library files found under
LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_UART_BLE_lowlevAPI.xx

- **Function** : `void` **`LBF_UART_BLE_SendByte`**`(uint8_t TxByte)`
- **Description** : Send a byte over the Tx pin of UART3, which connects to the BLE module.
  If hardware flow control has been enabled at init then UART3 CTS is taken into account to pause transmission over Tx if needed.
- **Parameters** :

  > *TxByte*: The 8-bit value to send

- **Return Value:** -

- **Function** : `uint8_t` **`LBF_UART_BLE_ReceiveByte`** `(void)`
- **Description** : Receives a data byte from UART3 which connects to the BLE module. If hardware flow control has been enabled at init then UART3 RTS is managed to signal readyness to the BLE module.
- **Parameters** : -
- **Return Value:** the data byte received

- **Function** : `void` **`LBF_UART_BLE_SendString`**`(char* pString)`
- **Description** : Send a string of characters over the Tx pin of UART3 which connects to the BLE module. If hardware flow control has been enabled at init then UART3 CTS is taken into account to pause transmission over Tx.
- **Parameters** : *pString*: pointer to the first character of the string
- **Return Value: -**

- **Function** : `void` **`LBF_UART_BLE_SendString_SWFlowControl`**`(char* pString)`
- **Description** : Send a string of characters over the Tx pin of UART3 which connects to the BLE module. If hardware flow control has been enabled at init then UART3 CTS is taken into account to pause transmission over Tx. In addition, software flow control is supported : transmission is paused upon reception of XOFF, until reception of next XON.
- **Parameters** : *pString*: pointer to the first character of the string
- **Return Value: -**

TO DO :

- **Function** : `void` **`LBF_UART_BLE_SendStream_DMA`**`(uint8_t* pTxBuf, uint16_t NumBytes)`
- **Description** : Send a string of bytes over the Tx pin of UART3 which connects to the BLE module, using the DMA channel associated to UART3. If hardware flow control has been enabled at init then UART3 CTS is taken into account to pause transmission over Tx.
- **Parameters** :
  > *pTxBuf*: pointer to the head of the buffer containing the data to send
  > *NumBytes*: the number of bytes to send

- **Return Value: -**

TO DO :

- **Function** : `void` **`LBF_UART_BLE_ReceiveStream_DMA`**`(uint8_t* pRxBuf, uint16_t NumBytes)`
- **Description** : Receives a string of bytes over the Rx pin of UART3 which connects to the BLE module, using the DMA channel associated to UART3.
- **Parameters** :
  - > *pRxBuf*: pointer to the head of the buffer where to store the received data
  - > *NumBytes*: the number of bytes to receive
- **Return Value: -**

3.8 Power Supplies

Library files found under LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_PWR_lowlevAPI.xx

OLED-related

- **Function** : `bool` **`LBF_Check_VDDH_On`**`(void)`
- **Description** : Checks whether the high voltage VDDH (13V) for the OLED display has been enabled or not.
- **Parameters** : -
- **Return Value:** *True* if VDDH has been enabled, else *false*

- **Function** : `void` **`LBF_Turn_VDDH_On`**`(void)`
- **Description** : Enable high voltage VDDH (13V) to OLED display
- **Parameters** : -
- **Return Value: -**

- **Function** : `void` **`LBF_Turn_VDDH_Off`**`(void)`
- **Description** : Disable high voltage VDDH (13V) to OLED display

- **Parameters** : -
- **Return Value: -**

VCC_LDO - related

- **Function** : `void` **`LBF_Enable_LDO`**`(void)`
- **Description** : Enable LDO, VCC_LDO (3.3V) available on extension port
- **Parameters** : -
- **Return Value: -**

- **Function** : `void` **`LBF_Disable_LDO`**`(void)`
- **Description** : Disable LDO
- **Parameters** : -
- **Return Value: -**

4 OTHER CONTROL FUNCTIONS

NOTE :

The peripheral control functions described below allow to use commonly needed peripherals in a simple way, with a quick prototyping objective in mind.

They only address a selection of STM32 on-chip peripherals though, in selected use cases.

Programmers are free to use these functions or rely on their own code, of course.

4.1 General Purpose Input/Output (GPIO) :

Library files found under

LimiFrog_SW/libraries/LimiFrog_Lib/inc/LBF_API/LBF_GPIO_lowlevAPI.h

- **Function** : `void` **`GPIO_HIGH`** `( GPIO_TypeDef* GPIO_Port_Name, uint16_t GPIO_Pin_Number )`
- **Description** : Sets to Logic_1 pin *GPIO_Pin_Number* of port *GPIO_Port_Name*.
- **Parameters** :

  > *GPIO_Port_Name* : can be GPIOA to GPIOC (or alias)

> *GPIO_Pin_Number* : can be 0 to 15 (or alias)

- **Return Value:** -
- **Note** : See file *LBF_pin_aliases.h* for a proposal list of meaningful port name and pin number aliases.

- **Function** : `void` **`GPIO_LOW`** `( GPIO_TypeDef* GPIO_Port_Name, uint16_t GPIO_Pin_Number )`
- **Description** : Sets to Logic_0 pin *GPIO_Pin_Number* of port *GPIO_Port_Name*.
- **Parameters** :

  > *GPIO_Port_Name* : can be GPIOA to GPIOC (or alias)

  > *GPIO_Pin_Number* : can be 0 to 15 (or alias)
- **Return Value:** -
- **Note** : See file *LBF_pin_aliases.h* for a proposal list of meaningful port name and pin number aliases.

- **Function** : `void` **`GPIO_TOGGLE`** `( GPIO_TypeDef* GPIO_Port_Name, uint16_t GPIO_Pin_Number )`
- **Description** : Toggles the logic level on pin *GPIO_Pin_Number* of port *GPIO_Port_Name*.
- **Parameters** :

  > *GPIO_Port_Name* : can be GPIOA to GPIOC (or alias)

  > *GPIO_Pin_Number* : can be 0 to 15 (or alias)
- **Return Value:** -
- **Note** : See file *LBF_pin_aliases.h* for a proposal list of meaningful port name and pin number aliases.

- **Function** : bool **`IS_GPIO_RESET`** `( GPIO_TypeDef* GPIO_Port_Name, uint16_t GPIO_Pin_Number )`
- **Description** : Evaluates the logic level on pin *GPIO_Pin_Number* of port *GPIO_Port_Name*.
- **Parameters** :

  > *GPIO_Port_Name* : can be GPIOA to GPIOC (or alias)

  > *GPIO_Pin_Number* : can be 0 to 15 (or alias)
- **Return Value:** true if level detected is Logic_0, else false

- **Note** : See file *LBF_pin_aliases.h* for a proposal list of meaningful port name and pin number aliases.

- **Function** : bool **`IS_GPIO_SET`** `( GPIO_TypeDef* GPIO_Port_Name, uint16_t GPIO_Pin_Number )`
- **Description** : Evaluates the logic level on pin *GPIO_Pin_Number* of port *GPIO_Port_Name*.
- **Parameters** :

  > *GPIO_Port_Name* : can be GPIOA to GPIOC (or alias)

  > *GPIO_Pin_Number* : can be 0 to 15 (or alias)
- **Return Value:** true if level detected is Logic_1, else false
- **Note** : See file *LBF_pin_aliases.h* for a proposal list of meaningful port name and pin number aliases.

4.2 Extension Port Control – Preset Configurations

`Library files found under LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_ExtPort_lowlevAPI.xx`

The functions described below set those STM32 peripherals which can be made available on the extension port in pre-defined (or « preset ») configurations. These preset configurations are representative of typical requirements of external systems with which one may want to interface across the extension port.

If the proposed configuration does not fit, of course the user is free to program the peripheral in any way he/she wants.

NOTE : These functions only deal with programming the STM32 peripheral. The setting of the corresponding I/Os into the relevant alternate function mode is supposed to have been done earlier – e.g., at board init through the use of function *LBF_Board_Init()*, with the relevant definitions in user file *User_Configuration.h*.

I2C1 over extension port

- **Function** : `void` **`LBF_Init_PresetConf_I2C1`** `(I2C_HandleTypeDef* hi2c )`
- **Description** : Puts I2C1 into the following preset configuration : 400kHz SCL clock, 7-bit I2C target adress (ChipID), dual addressing mode disabled, general call disabled. The peripheral clock APB1 is assumed to be 20MHz (else SCL clock will be different).
- **Parameters** : *hi2c* : pointer to a structure which the underlying HAL driver will fill to perform the initialization (see note below)
- **Return Value:** -

- **Note:** the user needs to declare a structure of type *I2C_HandleTypeDef* in his code prior to calling this function, passing a pointer to this structure as argument. He can then, if he so wishes, also use this handle to call other hAL functions, for example to send or receive data – he may also work at lower level if he prefers.

- **Function** : `void LBF_DeInit_PresetConf_I2C1 (I2C_HandleTypeDef* hi2c )`

SPI-over-USART2 on extension port

- **Function** : `void LBF_Init_PresetConf_SPIoverUSART2 (USART_HandleTypeDef* hSpiUsart, uint32_t User_Configured_BaudRate )`
- **Description** : Puts USART2 into an SPI master emulation mode, at 8-bit per frame, full-duplex, MSB first (as required by the SPI protocol), with the following pin correspondance : SPI MOSI = USART2 Tx, SPI MISO = USART2_Rx, SPI CK = USART2_CK. Chip select must be handled through a GPIO.
- **Parameters** :

  > *hSpiUsart* : pointer to a structure which the underlying HAL driver will fill to perform the initialization (see note below)
  > *User_Configured_BaudRate*: required baud rate i.e. clock frequency (in Hertz) for the SPI transmission. Maximum supported value is Fsystem_clock / 16.
- **Return Value: -**
- **Note:** the user needs to declare a structure of type *USART_HandleTypeDef* in his code prior to calling this function, passing a pointer to this structure as argument. He can then, if he so wishes, also use this handle to call otherhAL functions, for example to send or receive data – he may also work at lower level if he prefers.

- **Function** : `void LBF_DeInit_PresetConf_SPIoverUSART2( USART_HandleTypeDef* hSpiUsart )`

UART4 on extension port

- **Function** : `void LBF_Init_PresetConf_UART4 (UART_HandleTypeDef* huart, uint32_t User_Configured_BaudRate )`
- **Description** : Puts UART4 into the following preset configuration : 8-bit word length, 1 stop bit, no parity, x16 oversampling, full duplex(Rx and Tx enabled), no hardware flow control
- **Parameters** :

> *huart* : pointer to a structure which the underlying HAL driver will fill to perform the initialization (see note below)

> *User_Configured_BaudRate*: required baud rate i.e. clock frequency (in Hertz) for the SPI transmission. Maximum supported value is Fsystem_clock / 16.

- **Return Value: -**
- **Note:** the user needs to declare a structure of type *UART_HandleTypeDef* in his code prior to calling this function, passing a pointer to this structure as argument. He can then, if he so wishes, also use this handle to call otherhAL functions, for example to send or receive data – he may also work at lower level if he prefers.

- **Function** : `void` **`LBF_DeInit_PresetConf_UART4`** `(UART_HandleTypeDef* huart)`

UART2 on extension port

- **Function** : void **`LBF_Init_Presetconf_UART2`** `(UART_HandleTypeDef* huart, bool cts_flowcontrol, bool rts_flowcontrol, uint32_t User_Configured_BaudRate)`
- **Description** : Puts UART2 into the following preset configuration : 8-bit word length, 1 stop bit, no parity, x16 oversampling, full duplex(Rx and Tx enabled). Hardware flow control on CTS and/or RTS is possible.
- **Parameters** :

  > *huart* : pointer to a structure which the underlying HAL driver will fill to perform the initialization (see note below)

  > *cts_flowcontrol*: if set, hardware flow control is enabled on the UART4_CTS pin

  > *rts_flowcontrol*: if set, hardware flow control is enabled on the UART4_RTS pin

  > *User_Configured_BaudRate*: required baud rate i.e. clock frequency (in Hertz) for the SPI transmission. Maximum supported value is Fsystem_clock / 16.

- **Return Value: -**
- **Note:** the user needs to declare a structure of type *UART_HandleTypeDef* in his code prior to calling this function, passing a pointer to this structure as argument. He can then, if he so wishes, also use this handle to call otherhAL functions, for example to send or receive data – he may also work at lower level if he prefers.

- **Function** : `void` **`LBF_DeInit_PresetConf_UART2`** `(UART_HandleTypeDef* huart) hi2c )`

4.3 External Interrupts :

Library files found under
LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_EXTI_ExtPort_lowlevAPI.xx

- **Custom Type**:

```
typedef enum {
RISING_TRIGGER_FALSE = 0,
RISING_TRIGGER_TRUE = 1
}
IT_Rising_Trigger_t;
```

- **Description** : Used in below functions to specify if an external IT source should be rising-edge triggered or not

- **Custom Type**:

```
typedef enum {
FALLING_TRIGGER_FALSE = 0,
FALLING_TRIGGER_TRUE = 1
}
IT_Falling_Trigger_t;
```

- Description : Used in below functions to specify if an external IT source should be falling-edge triggered or not

- **Function** : `void LBF_Enable_EXTI_ExtPort(uint16_t GPIO_Pin, IT_RisingTrigger_t IT_RisingTrigger, IT_FallingTrigger_t IT_FallingTrigger);`

- **Description** : Enables the EXTI interrupt line associated to the specified pin. It is assumed that the GPIO pin of interest has already been setup as external interrupt source (and disabled) – this is covered by the *LBF_Init()* function based on the directives found in user file *User_Configuration.h*.

- **Parameters** :
  > *GPIO_Pin* : pin number (or alias) of the GPIO pin available on the extension port which needs to be enabled as EXTI interrupt line. See file *LBF_pin_aliases.h* for a proposal list of meaningful port name and pin number aliases.
  > *IT_RisingTrigger* : set to `RISING_TRIGGER_TRUE` to specify that a rising edge must trigger an EXTI interrupt, or to `RISING_TRIGGER_FALSE` to ignore a rising edge. It is possible to specify both rising and falling edge triggers.
  > *IT_FallingTrigger* : set to `FALLING_TRIGGER_TRUE` to specify that a rsing edge must trigger an EXTI interrupt, or to `FALLING_TRIGGER_FALSE` to ignore a falling edge It is possible to specify

both rising and falling edge triggers.

- **Return Value:** -
- **Notes :**

  1) All pins of same numbers on ports GPIOA, GPIOB, GPIOC are associated to the same EXTI line (for example PA0, PB0 and PC0 can all drive EXTI0). It is recommended to enable only one of them at a time. If only using the pins available on the extension port, this is guaranteed by design.However, a couple of STM32 pins that bear IT from on-board chips share the same pin number as some GPIOs available on the extension port. These are INT1 from LSM6DS3 (on PC6 – same pin number as CONN_POS2 = PA6) and INT from VL6180X (on PB2 – same pin number as CONN_POS5 = PA2).

  2) The EXTI group to which this EXTI belongs must still be enabled at NVIC level to be visible by the STM32's Cortex ARM core. See function *LBF_Enable_NVIC_EXT_ExtPort* below.

- **Function** : `void` **`LBF_Disable_EXTI_ExtPort`**`(uint16_t  GPIO_Pin);`
- **Description** : Disables the EXTI interrupt line associated to the specified pin.
- **Parameters** :
  > *GPIO_Pin* : pin number (or alias) of the GPIO pin available on the extension port which needs to be enabled as EXTI interrupt line. See file *LBF_pin_aliases.h* for a proposal list of meaningful port name and pin number aliases.
- **Return Value:** -

- **Function** : `void` **`LBF_SoftIT_EXTI_ExtPort`**`(uint16_t  GPIO_Pin);`
- **Description** : Generates a software interrupt (sets the interrupt pending flag) on the EXTI interrupt line associated to the specified pin.
- **Parameters** :
  > *GPIO_Pin* : pin number (or alias) of the GPIO pin available on the extension port which needs to be enabled as EXTI interrupt line. See file *LBF_pin_aliases.h* for a proposal list of meaningful port name and pin number aliases.
- **Return Value:** -

- **Function** : bool **`LBF_IsPending_EXTI_ExtPort`**`(uint16_t  GPIO_Pin);`
- **Description** : Indicates if an interrupt is pending on the EXTI line associated to the specified pin.
- **Parameters** :
  > *GPIO_Pin* : pin number (or alias) of the GPIO pin available on the extension port which needs to be enabled as EXTI interrupt line. See file *LBF_pin_aliases.h* for a proposal list of meaningful port

name and pin number aliases.

- **Return Value:** *true* if EXTI interrupt is pending, else *false*

- **Function** : void **`LBF_Clear_EXTI_ExtPort`**`(uint16_t GPIO_Pin);`
- **Description** : Clears any pending interrupt on the EXTI line associated to the specified pin.
- **Parameters** :
  > *GPIO_Pin* : pin number (or alias) of the GPIO pin available on the extension port which needs to be enabled as EXTI interrupt line. See file *LBF_pin_aliases.h* for a proposal list of meaningful port name and pin number aliases.
- **Return Value:** -

- **Function** : void **`LBF_Enable_NVIC_EXTI_ExtPort`**`(uint16_t GPIO_Pin);`
- **Description** : Enables the NVIC for the *group* of interrupt sources containing the EXTI line associated to the specified pin (see also note below). Also sets the priority of this interrupt group according to *__EXTIxx_IRQn_PRIO* specified in user file I*T_Priorities_UserDefinable.h*.
- **Parameters** :
  > *GPIO_Pin* : based on this parameter (pin number or alias), the correspoding EXTI line is identified and it is the group to which this EXTI belongs that gets enabled at NVIC level
- **Return Value:** -
- **Note :** Some interrupt sources are combined before entering the NVIC. In the case of EXTI interrupts from STM32 GPIOs, EXTI0 to EXTI4 are independent, EXTI5 to EXTI9 are grouped into EXTI5_9 and EXTI10 to EXTI15 are grouped into EXTI15_10. Therefore, calling for example *LBF_Enable_NVIC_EXTI_ExtPort(CONN_POS11_PIN), CONN_POS11_PIN* being an alias for *GPIO_Pin_9,* will make not only EXTI9 but also EXTI8, EXTI7, EXTI6 and EXTI5 visible to the NVIC.

- **Function** : void **`LBF_Disable_NVIC_EXTI_ExtPort`**`(uint16_t GPIO_Pin);`
- **Description** : Disables the NVIC for the *group* of interrupt sources containing the EXTI line associated to the specified pin (see also note below).
- **Parameters** :
  > *GPIO_Pin* : *PIO_Pin* : based on this parameter (pin number or alias), the correspoding EXTI line is identified and it is the group to which this EXTI belongs that gets diabled at NVIC level
- **Return Value:** -
- **Note :** CAUTION - Since the complete group of EXTI which contains the EXTI associated to the specified pin gets disabled, care must be taken that disabling the NVIC may impact the visibility of another IT that would be sharing the same EXTI group. For example, calling

`LBF_Disable_NVIC_EXTI_ExtPort(CONN_POS11_PIN),` with CONN_POS11_PIN being an alias for GPIO_Pin_9, will disable EXTI5_9 at NVIC level and therefore also stop the STM32 from servicing any interrupt raised by EXT5 to EXTI8.

4.4 Timers :

Library files found under
LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_Timer_lowlevAPI.xx

- **Defined constants :**

```
#define LBF_TIMER_UNIT_US     1
#define LBF_TIMER_UNIT_MS     1000
```

Description : Useful defines to specify prescale clock period in a self-explanatory way in timer setup functions of LimiFrog API

- **Custom Type**:

```
typedef enum {
TIMER1 = 1,
TIMER2 = 2,
TIMER3 = 3,
TIMER4 = 4,
TIMER5 = 5,
TIMER6 = 6,
TIMER7 = 7,
TIMER8 = 8,
TIMER15 = 15,
TIMER16 = 16,
TIMER17 = 17
}
LBF_TimerID_t;
```

- **Description** : Timer Identification, used as parameters for the functions described below. Only timers present in the STM32L476 are listed

- **Custom Type**:

```
typedef enum {
CHANNEL1 = 1,
CHANNEL2 = 2,
CHANNEL3 = 3,
```

```
CHANNEL4 = 4
}
LBF_ChannelID_t;
```

- **Description** : Channel Identification within a given timer, used as parameters for the functions described below.

- **Function** : `void LBF_Timer_Setup( TIM_HandleTypeDef* htim, LBF_TimerID_t TimerID, uint16_t Timer_TimeUnit_us, uint32_t Period_as_TimerUnits )`
- **Description** : Configures the timer identified by *TimerID,* with a user-specified period expressed in a user-specified time-unit (typically in micro-seconds or in millisonds). The time unit corresponds to the period of the 'prescaled clock' in the timer hardare. The timer is set to up-counting mode.
- **Parameters** :

  > *htim*: pointer to a structure which the underlying HAL driver will fill to perform the initialization (see note below)
  > *TimerID* : the identity (number) of the timer to set up
  > *Timer_TimeUnit_us* : the prescaled clock period to use for the target timer, expressed in micro-seconds (us). Can therefore be regarded as time unit for that timer. Aliases LBF_TIMER_UNIT_US or LBF_TIMER_UNIT_MS can conveniently be used to get the timer to use, respectively, one micro-second or one milli-second as time unit.
  > *Period_as_TimerUnits* : Timer period, expressed in the formerly defined time unit (i.e. in number of prescaled clock periods)
- **Return Value:** -
- **Notes:**

  1) the user needs to declare a structure of type *TIM_HandleTypeDef* in his code prior to calling this function, passing a pointer to this structure as argument. He can then, if he so wishes, also use this handle to call other HAL functions, for example to send or receive data – he may also work at lower level if he prefers.

  2) This function requires the clock of the timer peripheral to be lower than 65.536 MHz, i.e. the system clock is >65.536MHz then the APB prescaler must be at least 4. This is the case if using the proposed LBF_Board_Inits() function which includes initialization of peripheral clocks.

  //TODO – implement an automatic check of timer peripheral clock

- **Function** : `void LBF_Timer_Start_ITout( TIM_HandleTypeDef* htim )`
- **Description** : Starts the timer identified by *htim* with generation of an interrupt at the end of each period
- **Parameters** : *htim* : pointer to the structure of type *TIM_HandleTypeDef* that contains the

configuration parameters of the selected timer. Typically this pointer is first created by the user (through the declaration of a structure of type *TIM_HandleTypeDef)* and then passed to functions such as this one and others to operate on this timer

- **Return Value:** -

- **Function** : `void` **`LBF_Timer_Stop`**`( TIM_HandleTypeDef* htim )`
- **Description** : Stops the timer identified by *htim*.
- **Parameters** : *htim* : pointer to the structure of type *TIM_HandleTypeDef* that contains the configuration parameters of the selected timer. Typically this pointer is first created by the user (through the declaration of a structure of type *TIM_HandleTypeDef)* and then passed to functions such as this one and others to operate on this timer
- **Return Value:** -

EXAMPLES :

`> TIM_HandleTypeDef hTim;`

Effect : A structure of type *TIM_HandleTypeDef* is declared, it can then be used (by HAL) to set-up and handle a timer

`> LBF_Timer_Setup( &hTim, TIMER4, TIMER_UNIT_MS, 1500 );`

Effect : Timer #4 is initialized to operate with a 1.5 s period, initialization is performed using structure hTim declared above

`> LBF_Timer_StartITout( &hTim );`

Effect : Start Timer #4 (the timer to which hTim4 was associated through the previous function call) with generation of an IT each time the period elapses.

`> LBF_Timer_Stop( &hTim );`

Effect : Stop Timer #4 (the timer to which hTim4 was associated through the previous function call)

- **Function** : `void` **`LBF_PWMchannel_Setup`**`( TIM_HandleTypeDef* htim, TimChannelID_t ChannelID, uint32_t Pulse_as_TimerUnits )`
- **Description** : Initializes the specified channel of the timer handled by htim to generate a pulse at high logic level of the specified duration when enabled (actual pulse generation does not start yet)
- **Parameters** :
  > *htim* : pointer to the structure of type *TIM_HandleTypeDef* that contains the configuration

parameters of the selected timer. Typically this pointer is first created by the user (through the declaration of a structure of type *TIM_HandleTypeDef)* and then passed to functions such as this one and others to operate on this timer.

> Channel_*ID* : the identity (number) of the channel to use
> *Pulse_as_TimerUnits* : High pulse duration, expressed in the formerly defined time unit (i.e. in number of prescaled clock periods) – typically in ms or us.

- **Return Value:** -
- **Note** : The selected timer must have been set up beforehand through the use of function *LBF_Timer_Setup*

- **Function** : `void` **`LBF_PWMchannel_Start`**`( TIM_HandleTypeDef* htim, TimChannelID_t ChannelID )`
- **Description** : Start the specified PWM channel of the timer handled by htim
- **Parameters** :
  > *htim* : pointer to the structure of type *TIM_HandleTypeDef* that contains the configuration parameters of the selected timer. Typically this pointer is first created by the user (through the declaration of a structure of type *TIM_HandleTypeDef)* and then passed to functions such as this one and others to operate on this timer.

  > Channel_*ID* : the identity (number) of the channel to use
- **Return Value:** -
- **Note** : The selected PWM channel musthave been set up beforehand through the use of function *LBF_PWMchannel_Setup*

- **Function** : `void` **`LBF_PWMchannel_Stop`**`( TIM_HandleTypeDef* htim, TimChannelID_t ChannelID )`
- **Description** : Stop the specified PWM channel of the timer handled by htim
- **Parameters** :
  > *htim* : pointer to the structure of type *TIM_HandleTypeDef* that contains the configuration parameters of the selected timer. Typically this pointer is first created by the user (through the declaration of a structure of type *TIM_HandleTypeDef)* and then passed to functions such as this one and others to operate on this timer.

  > Channel_*ID* : the identity (number) of the channel to use
- **Return Value:** -

- **Function** : `void` **`LBF_Update_Pulse`**`( TIM_HandleTypeDef* htim, TimChannelID_t`

`ChannelID, uint32_t Pulse_as_TimerUnits )`

- **Description** : Updates on-the-fly the pulse duration of the specified PWM channel of the timer handled by *htim* – allowing to modulate its duty cycle,
- **Parameters** :

  > *htim* : pointer to the structure of type *TIM_HandleTypeDef* that contains the configuration parameters of the selected timer. Typically this pointer is first created by the user (through the declaration of a structure of type *TIM_HandleTypeDef)* and then passed to functions such as this one and others to operate on this timer.

  > Channel_*ID* : the identity (number) of the channel to use
  > *Pulse_as_TimerUnits* : High pulse duration, expressed in the formerly defined time unit (i.e. in number of prescaled clock periods) – typically in ms or us.
- **Return Value:** -

4.5 Services and Utilities

Library files found under LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_API/LBF_Services_lowlevAPI.xx

- Function : `void` **`LBF_Delay_ms`** `(uint16_t nTime) void LBF_Delay_ms (uint16_t nTime)`
- **Description** : Loops until the delay specified in milliseconds by *nTime* is elapsed. ***LBF_Delay_ms* is based on Timer3 rather than SysTick**. This also means it is safer to avoid using Timer3 for other purposes in the application (even though, technically, it is sufficient to make sure LBF_Delay_ms() is never invoked while Timer3 is running for some other purpose).
- **Parameters** : *nTime* : the number of millisecond to wait – Maximum 60000ms = 1mn (greater values will be clipped)
- **Return Value:** -

Library files found under LimiFrog_SW/libraries/LimiFrog_Lib/xxx/Debug_Utilities :

- **Function** : `void` **`LBF_Led_StopNBlinkOnFalse`**`( bool Status )`
- **Description** : Enters infinite loop with flashing LED is *Status* is false, else continue. For debug and diagnosis.
- **Parameters** : *Status* : the boolean that will (if *false*) make LED to flash and program to enter infinite loop or(if *true*) let it continue

- **Return Value:** -

- **Function** : `void` **`LBF_Led_BlinkOnFalse_FixedOnTrue`**`( bool Status )`
- **Description** : Enters infinite loop, with flashing LED is *Status* is false, or with fixed LED on if *Status* is true. For debug an diagnosis.
- **Parameters** : *Status* : the boolean that will cause LED to flash if false and to stay fixed if true.
- **Return Value:** -

4.6 Embedded String Functions

Library files found under LimiFrog_SW/libraries/LimiFrog_Lib/xxx/Debug_Utilities/xprintf.xx

`xprintf``, etc.`

Compact string IO library provided by ChaN.

Documentation on : http://elm-chan.org/fsw/strf/xprintf.html

5 MIDDLEWARE

5.1 USB device stack

ST provides a full USB stack supporting different classes, as part as the Cube environment.

LimiFrog's software package includes the « glue » to use the Data Flash as storage media when the USB is used in Mass Storage mode. The LimiFrog library provides functions to start and stop USB in mass storage mode with no further intervention by the application.

Library files found under LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_USB_MassStorage

- **Function** : `bool` **`LBF_LaunchUSB_MassStorage`**`(void)`
- **Description** : Configures the USB stack in Mass Storage class using the data Flash as storage media, and enables the stack. After calling this function USB mass storage is operational, an external host device will see LimiFrog as a USB drive. Returns true if operation was successful, else false.
- **Parameters** : -

- **Return value:** true if success, false if problem

- **Function** : `void` **`LBF_StopUSB_MassStorage`**`(void)`
- **Description** : Stops USB operation in Mass Storage mode.
- **Parameters** : -
- **Return Value:** -

5.2 FatFs File System

ST provides a FatFs middleware, which relies on ChaN's FatFs open-source library. This is a FAT file system optimized for embedded systems.

LimiFrog's software package includes the « glue » to use the Data Flash as storage media for this FAT File System. The LimiFrog library provides an nitialization function that allows to then directly use the Fat-Fs file operations listed under « **Application Interface** » in the **FatFs documentation :** http://elm-chan.org/fsw/ff/00index_e.html (f_open, f_read, f_opendir, etc.)

Library files found under LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_DataFlash_FatFS

- **Function** : `bool` **`LBF_FatFS_Init`**`(void)`
- **Description** : Initializes the Data Flash as FatFs file system. After this call the FatFs peripheral is mounted and ready to use. Returns true if operation was successful, else false.
- **Parameters** : -
- **Return value:** true if success, false if problem

- **Function** : `bool` **`LBF_FatFS_DeInit`**`(void)`
- **Description** : De-initializes the FatFs file system, driver is unlinked.
- **Parameters** : -
- **Return Value:** true if success, else false

5.3 emWin Graphics Library

ST provides, under the name StemWin, an implementation of Segger's emWin professional graphics library ported to the STM32. Refer tp emWin's documentation for a full description of this very rich API.
LimiFrog's software package includes the « glue » to interface this generic library with the OLED display

controller. The LimiFrog library provides an initialization function that allows to then use the emWin API to perform many graphics operation.

Library files found under LimiFrog_SW/libraries/LimiFrog_Lib/xxx/LBF_STemWin

- **Function** : `bool` **`LBF_emWin_Init`**`(void)`
- **Description** : Initializes the emWin graphics library (Note : this reserves some RAM).
- **Parameters** : -
- **Return value:** true if success, false if problem

Then the user can use the API provided by emWin, document in document « ***Graphic Library with Graphical User Interface - User & Reference Guide*** » by Segger (included in documentation package provided with LimiFrog). **Functions from this ibrary are all prefixed GUI**_.

- **Function** : void **`LBF_emWin_DeInit`**`(void)`
- **Description** : De-initializes the emWin graphics library. Clears emWin internal data from memory.
- **Parameters** : -
- **Return value:**