Draft D

LimiFrog API

December, 2015

TO DO :

> Implement and add more return codes (success/error) – align on usual conventions

> Add de-init functions in general and on extension port in particular

> Add non-blocking peripheral accesses

> Add missing functionalities (e.g. simplfied ADC setup, etc)

> Could specify in which C file each function is found

> Make naming conventions more consistent

LimiFrog API consists of :

- board support functions :

  > **full STM32 initialization**

  > **simple control of on-board hardware**

- a selection of high-level, **easy to use STM32 peripheral control functions**, focusing on a number of common use cases

- plus functions to support ST-provided **middleware** (USB, File System, Graphics Lib) on LimiFrog's hardware

LimiFrogAPI has been developed with a prototyping objective in mind -- possibly trading off performance optimization for ease of use in some places. If you need something more optimized you may either use your own choice of libraries, or perhaps start from this package and rework sections that need optimization for your application.

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 FULL BOARD INITIALIZATION

These functions take care of configuring all STM32 I/Os and on-chip peripherals that are decicated to communication with on-board hardware and with the extension port. They also initialize a number of integrated circuits present on the board so they are ready for use.

1.1 Fixed Initializations

- **Function** :   `void LBF_Board_Fixed_Inits(void)`
- **Description** : Performs all « fixed » initializations, which do not depend on some user choices. This includes configuring the clocks, the Interrupt Controller, all « fixed-role » GPIOs and peripherals, etc.
- **Parameters** : -
- **Return Value**: -

- **Function** : `void` **`LBF_Board_Selective_Inits`**`(void)`
- **Description** : Performs all « selective » initializations, which depend on some user choices, expressed as directives in a user configuration file « *User_Configuration.h ».*
  Selective initializations include :
  > configuring the STM32 IO that are routed to the extension port according to user's wishes
  > configuring the BlueTooth Low-Energy Module if its usage is enabled by the user
  > initializing the OLED display controller if usage of the OLED is enabled by the user
- **Parameters** : -
- **Return Value:** -
- **Notes:** The extension port connects to GPIOs of the STM32, which are largely configurable and support many interfaces. Through the config file and this selective initialization function, LimiFrog API offers to easily set up a number of pre-defined configurations. They don't intend to cover all possible requirements, though – they are rather a selection of the most typical needs – and the user is free to write his own code or adapt this for his specific needs. Predefined configurations include :
  - **I2C1** – on extension port positions 9 (SCL) and 10 (SDA) : 400KHz, single-master memory-mapped access to external chip using an 8-bit base address.

  - **SPI** – on extension port positions 3-4-5 and 1 (CK, MISO, MOSI and nCS) : using USART2 configured in SPI imode, user-specified speed (max CoreClk/16), with idle SCK clock level = low and data sampling (MISO/MOSI) expected on rising SCK edge.
  - **UART4** – on extension port positions 7 (Rx) and 8 (Tx) : 1 stop bit,no parity, with user-specified baud rate (max CoreClk/16)
  - **UART2** – on extension port positions 4-5 (Rx-Tx) and optionally 7-8 (RTS-CTS): : 1 stop bit,no parity, with user-specified baud rate (max CoreClk/16), and optinal hardware flow control
  - **CAN** (Controller Area Network) bus – on extension port positions 9 (Rx) and 10 (Tx) *: <preset configuration tbd>*
  - **GPIO** – on any extension port position within 1, 3-5, 7-10 : configurable as either input with optional pull-up/down or output (optionally open-drain). The drive strength is preset to «low» (to limit switching noise).
  - **External Interrupt Request (IRQ**) – on any extension port position within 1, 3-5, 7-10 : configurable as either rising-edge or falling-edge triggered.
  - **PWM** output – on extension port positions 1, 3-7, 8-10 : configured as edge-aligned, upcounting timer, positive pulse, with period and pulse duration expressable un micro-seconds or milli-seconds, with the following predefined assignment :
  - Pos. #1 : PWM from STM32 Timer 16 (single channel timer)
  - Pos. #3 : PWM from STM32 LP-Timer 2 (single channel)
  - Pos. #4 : PWM from STM32 Timer 2, Channel 4
  - Pos. #5 : PWM from STM32 Timer 2, Channel 3
  - Pos. #7 : PWM from STM32 Timer 2, Channel 2
  - Pos. #8 : PWM from STM32 Timer 2, Channel 1
  - Pos. #9 : PWM from STM32 Timer 4, Channel 3

- Pos. #10 : PWM from STM32 Timer 4, Channel 4

Positions 2, 6 and 11 cannot bear a PWM signal. PWM signals from the same timer will all use the same period, but the duty cycle can be different for each channel of the same timer.

Some connector positions support other Timer/Channel assignments than those preset by this function. The user has to write his own code in case he wants to use a such a different assignment

- **Analog-to-Digital** Conversion – on extension port positions 1, 3-5, 7-8, with the following assignment :
- Pos. #1 : input to STM32 ADC channel 11
- Pos. #3 : input to STM32 ADC channel 9
- Pos. #4 : input to STM32 ADC channel 8
- Pos. #5 : input to STM32 ADC channel 7
- Pos. #7 : input to STM32 ADC channel 6
- Pos. #8 : input to STM32 ADC channel 5

- **Digital-to-Analog** conversion – on extension port position 3

2 ON-BOARD HARDWARE CONTROL

These functions are provided to interact with on-board hardware in a simple way.

2.1 Battery

<TO DO : Port Code from L1>

- **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.

2.2 LED

- **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:** -

2.3 Push-button switches

- **Function** : `bool` **`LBF_State_Switch1_IsOn`**`(void)`
- **Description** : Returns the state of push-button switch #1 located at edge of board
- **Parameters** : -
- **Return Value:** true if button being pushed, else false

- **Function** : `bool` **`LBF_State_Switch2_IsOn`**`(void)`
- **Description** : Returns the state of push-button switch #2 located at edge of board
- **Parameters** : -
- **Return Value:** true if button being pushed, else false

2.4 OLED Display

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. See section 'Middleware' further down that document.**

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

- **Description** : Provides high voltage 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_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 – WRITE CODE>

- **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 screen region with RGB565 pixels stored in a buffer

- **Parameters** :

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

- **Return Value:** -

******* IMPORTANT NOTE : The following 4 functions actually encapsulate emWin functtions, which means this middleware must have been enabled. 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. (emWin must have been enabled first)
- **Parameters** : *string (In)* : pointer to the string of characters to be displayed
- **Return Value:** -

- **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. (emWin must have been enabled first)
- **Parameters** : *SignedInteger (In)* : the signed integer value to display in decimal format
- **Return Value:** -

- **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. (emWin must have been enabled first)
- **Parameters** : *Unsigned16 (In)* : the 16-bit value to display in hexadecimal format
- **Return Value:** -

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

******* 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 (In)* : 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 (In)* : The address of the register to write
  > *RegValue (In)* : The value to write
- **Return Value:** -

- **Function** : `void LBF_OLED_DataStart (void)`
- **Description** : Signals to the OLED that all subsequent data that will be transfered over SPI will be for the DDRAM (Display Data RAM) of the OLED, until instructed otherwise (typically, through OLED_DataEnd function).
- **Parameters** : -
- **Return Value:** -

- **Function** : `void LBF_OLED_SendData (uint16_t Value)`
- **Description** : Transfers 16 bit of RGB565 data over SPI towards the OLED.

- **Parameters** : *Value (In)*: the pixel value to send, in RGB565 format
- **Return Value:** -

- **Function** : `void **LBF_OLED_DataEnd** (void)`
- **Description** : Deselects the OLED thus ending any sequence of data transfer
- **Parameters** : -
- **Return Value:** -

2.5 Data Flash

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** : `void LBF_FLASH_WriteBuffer(uint8_t* pBuffer, uint32_t WriteAddr, uint32_t NumByteToWrite)`
- **Description** : Writes into the Data Flash, starting at the adress specified by *WriteAddr*, the number of data bytes specified by *NumByteToWrite* located in a buffer pointer by *pBuffer*.
- **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

2.6 Sensors

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

> They can also be controlled using IC vendor-supplied API. These drivers are being integrated into the LimiFrog API (mostly a matter of mapping the generic bus interface of these drivers to I2C #2)
<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 >

Note : *on LimiFrog, I2C #2 is dedicated to interactions with the on-board sensors. I2C #1 can be made available on the extension port. I2C #3 is not used.*

- **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:** -

2.7 BlueTooth Low-Energy

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

The user simply sets a couple of directives in a 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 is easily done by connecting LimiFrog to PC through USB.

Initialization function *`LBF_Board_Selective_Inits()`* 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.

- **Function** : `void` **`LBF_UARTBLE_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.
- **Parameters** :

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

- **Function** : `uint8_t` **`LBF_UARTBLE_ReceiveByte`** `(void)`
- **Description** : Receives a data byte from UART3 which connects to the BLE module (with a time out set to tbd ms). 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

  <TO DO>: need to handle case of time out (no valid data received)

<TO DO – Improve this section – Support non-blocking accesses >

2.8 Extension Port Control

These functions are provided to use some of the interfaces provided over pins of the extension port in a simple way

I2C over extension port

- **Function** : `void` **`LBF_I2C1_WriteSingleReg`**`(uint8_t ChipID, uint16_t RegAdd, uint8_t RegVal)`
- **Description** : Writes the specified value at the specified address of the (off-board) chip (I2C slave) identified by I2C address *ChipID.* The transfer occurs over the pins of the extension port assigned to I2C interface #1.
- **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_I2C1_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 (off-board) chip (I2C slave) identified by I2C address *ChipID.* The transfer occurs over the pins of the extension port assigned to I2C interface #1.
- **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_I2C1_ReadSingleReg`** `(uint8_t ChipID, uint16_t RegAdd)`
- **Description** : Return value read from address *RegAdd* of the (off-board) chip (I2C slave) identified by I2C address *ChipID.* The transfer occurs over the pins of the extension port assigned to I2C interface #1.
  > *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_I2C1_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 (off-board) chip (I2C slave) identified by I2C address *ChipID.* The transfer occurs over the pins of the extension port assigned to I2C interface #1.
- **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_I2C1_RmodWSingleReg`** `(uint8_t ChipID, uint16_t RegAdd, uint8_t RegMask, uint8_t RegUpdateVal)`
- **Description** : Read value *RegVal* from address *RegAdd* of the (off-board) chip (I2C slave) 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*. The transfer occurs over the pins of the extension port assigned to I2C interface #1.
- **Parameters** : -
- **Return Value:** -

SPI over extension port

Function : `uint8_t` **`LBF_SPIoverUSART2_TransferByte`**`(uint8_t TxByte)`

- **Description** : Using USART2 as SPI interface, send a byte and simultaneously receive one as per SPI specfication. Chip Select is managed.
- **Parameters** :

  > *TxByte*: The 8-bit value to send over SPI (MOSI)
- **Return Value:** the 8-bit value received over SPI (MISO)

Function : `void` **`LBF_SPIoverUSART2_TransferMultiple`**`(tbd)`

- **Description** : Using USART2 as SPI interface, send and simultaneously receive a series of bytes as per SPI specfication. Chip Select is managed.
- **Parameters** :

  > tbd
- **Return Value:** -

  <TO DO – Improve this function – Interest of keeping previous in that case ? - Support non-blocking transfer >

UART2 / UART4 over extension port

- **Function** : `void` **`LBF_UART2_SendByte`**`(uint8_t TxByte)`

  Function : `void` **`LBF_UART4_SendByte`**`(uint8_t TxByte)`
- **Description** : Send a byte over the Tx pin of UART2 / UART4 on the extension port.

  For UART2 only : if hardware flow control has been enabled at init then UART2 CTS is taken into account to pause transmission over Tx.
- **Parameters** :

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

- **Function** : `uint8_t` **`LBF_UART2_ReceiveByte`** `(void)`

  Function : `uint8_t` **`LBF_UART4_ReceiveByte`** `(void)`
- **Description** : Receives a data byte from UART2 /UART4 on the extension port (with a default time out set to 100ms). For UART2 only : if hardware flow control has been enabled at init then UART2 RTS is managed to signal readyness to peer.
- **Parameters** : -

- **Return Value:** the data byte received

<TO DO – Improve this section – Support non-blocking accesses >

- **Function :** `void` **`LBF_UART2_SendString`**`(char* pString)`
  Function : `void` **`LBF_UART4_SendString`**`(char* pString)`
- **Description** : Send a string of characters over the Tx pin of UART2 / UART4 on the extension port. For UART2 only : if hardware flow control has been enabled at init then UART2 CTS is taken into account to pause transmission over Tx.
- **Parameters** :

  > *pString*: pointer to the first character of the string (\0 terminated).
- **Return Value:** -

- **Function** : `void` **`LBF_UART2_SendString_SwFlowControl`** `(char* pString)`
  Function : `void` **`LBF_UART4_SendString_SwFlowControl`** `(char* pString)`
- **Description** : Sends a string of characters over UART2/UART4 , with software flow control (suspend transmission if XOFF received, resume when XON received)
- **Parameters** :
  > *pString*: pointer to the first character of the string
- **Return Value:** -

PWM over extension port

- **Function** : `void` **`LBF_Timer_Setup`**`( LBF_TimerID_t TimerID, uint16_t Timer_TimeUnit_us, uint32_t Period_as_TimerUnits )`
- **Description** : For the timer identified by *TimerID,* defines the time unit (i.e. prescaled clock period) to use and the period of the timer.
- **Parameters** :
  > *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:** -

- **Function** : `void` **`LBF_PWMchannel_Setup`** `( LBF_TimerID_t TimerID, LBF_ChannelID_t ChannelID, uint32_t Pulse_as_TimerUnits )`
- **Description** : On channel number *ChannelID* of the timer identified by *TimerID*, set-up a PWM signal with high-pulse duration equal to *Pulse_as_TimerUnits*,
- **Parameters** :

  > *TimerID* : the identity (number) of the timer to which the PWM channel to setup belongs

  > ChannelID : the identity (number) of the channel to set up within the formerly specified timer
  > *Pulse_as_TimerUnits :* the duration of the high pulse of the PWM signal, expressed in the unit (i.e. prescaled clock period) defined when setting up this timer.
- **Return Value:** -

- **Function** : `void` **`LBF_PWMChannel_Start`** `(LBF_TimerID_t TimerID, LBF_ChannelID_t ChannelID)`
- **Description** : Start generation of the PWM signal previsouy setup on channel number *ChannelID* of timer number *TimerID*
- **Parameters** :
  > *TimerID* : the identity (number) of the timer to which the PWM channel to start belongs

  > ChannelID : the identity (number) of the channel to start within the formerly specified timer
- ***Return Value:*** -

- **Function** : `void` **`LBF_PWMChannel_Stop`** `(LBF_TimerID_t TimerID, LBF_ChannelID_t ChannelID)`
- **Description** : Stop generation of PWM signal on channel number *ChannelID* of timer number *TimerID*
- **Parameters** :
  > *TimerID* : the identity (number) of the timer to which the PWM channel to stop belongs

  > ChannelID : the identity (number) of the channel to stop within the formerly specified timer
- ***Return Value:*** -

- **Function** : `void` **`LBF_PWMChannel_UpdatePulse`** `(LBF_TimerID_t TimerID,`

```
LBF_ChannelID_t ChannelID, uint32_t  Pulse_as_TimerUnits)
```

- **Description** :   On the specified channel number of the specified timer, assumed to have previously been configued to generate a PWM signal, change duration of the high pulse (period is unchanged)
- **Parameters** :

  > *TimerID* : the identity (number) of the timer to which the PWM channel to update belongs

  > *ChannelID* : the identity (number) of the channel to update within the formerly specified timer

  > *Pulse_as_TimerUnits :* the new duration of the high pulse of the PWM signal, expressed in the unit (i.e. prescaled clock period) defined when setting up this timer.
- **Return Value:** -

EXAMPLES :

`> LBF_Timer_Setup( TIMER4, TIMER_UNIT_US, 150 );`

Timer #4 is initialized to operate with a 150 micro-second period.

`> LBF_PWMChannel_Setup (TIMER4, CHANNEL3, 30) ;`

On channel #3 of Timer #4, set-up a PWM generator with high pulse duration 30us (as Timer #3 has been set-up to use micro-seconds as base unit)

`> LBF_PWMChannel_Start (TIMER3, CHANNEL3) ;`

Start generation of the PWM signal on channel #3 of timer #3 – with a period set to 150ms and a duty cycle set to 30ms, the duty cycle is 30/150=20%

`> LBF_PWMChannel_UpdatePulse (TIMER4, CHANNEL3, 60) ;`

Change pulse duration of PWM on channel3 of timer 3 to 60us – duty cycle is now 40 %

`> LBF_PWMChannel_Stop (TIMER4, CHANNEL3) ;`

Stop generation of PWM signal on channel #3 of timer #4

<TO DO – Some of this still to port from L1 >

CAN bus over extension port

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

  <TO DO >

A/D conversion of data from extension port

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

  <TO DO – port from L1 >

D/A conversion of data to extension port

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

  <TO DO >

3 OTHER HIGH-LEVEL CONTROL FUNCTIONS

NOTE :
The peripheral control functions described below allow to use commonly needed peripherals in a simple way. They only address a selection of STM32 on-chip peripherals though, in selected (simple) use cases. The intent is not to replace ST's HAL library, it is rather :
> (for beginners especially) to enable **writing simple code** without diving too deeply into the HAL library
> to provide some **examples that illustrate usage of the HAL library**, thus facilitating writing HAL-based code for other use cases
Programmers are free to use these functions or rely on their own code, of course.

3.1 General Purpose Input/Output (GPIO) :

TBD *: use functions rather than macros ?*

Prefix names.

- **Macro** :**`GPIO_HIGH`**`(GPIO_Port_Name , GPIO_Pin_Number)`
- **Description** : Sets to logic 1 pin *GPIO_Pin_Number* of port *GPIO_Port_Name*.
  GPIO_Port_Name can be GPIOA to GPIOC (or alias) and *GPIO_Pin_Number* can be 0 to 15 (or alias). File *LBF_Name_Aliases.h* proposes aliases for all STM32 GPIOs with meaningful names.

- **Macro** :**`GPIO_LOW`**`(GPIO_Port_Name , GPIO_Pin_Number)`
- **Description** : Sets to logic 0 pin *GPIO_Pin_Number* of port *GPIO_Port_Name*.

- **Macro** :**`GPIO_TOGGLE`**`(GPIO_Port_Name , GPIO_Pin_Number)`
- **Description** : Toggles pin *GPIO_Pin_Number* of port *GPIO_Port_Name*.

- **Macro** :**`IS_GPIO_SET`**`(GPIO_Port_Name , GPIO_Pin_Number)`
- **Description** : Returns true if pin *GPIO_Pin_Number* of port *GPIO_Port_Name* is set (Logic 1),

else returns false (result is cast to type *bool*)

- **Macro** :`IS_GPIO_RESET(GPIO_Port_Name , GPIO_Pin_Number)`
- **Description** : Returns true if pin *GPIO_Pin_Number* of port *GPIO_Port_Name* is reset (Logic 0), else returns false (result is cast to type *bool*)

3.2 External Interrupts :

- **Function** : `void LBF_Enable_ExtIT(GPIO_TypeDef* GPIO_Port, uint16_t GPIO_Pin, bool Rising_nFalling_IT);`
- **Description** : Configures the specified GPIO as external edge-triggered interrupt source of specified polarity and enables it. Only handles GPIO that connect to valid interrupt sources on LimiFrog board : pins of the extension port and interrupt request outputs from on-board integrated circuits.
- **Parameters** :
  > *GPIO_Port* : the port to which the IO defined as external interrupt belongs – can be (alias of) GPIOA, GPIOB or GPIOC
  > *GPIO_Pin* : the pin number in the port (0 to 15) of the IO defined as external interrupt
  > *Rising_nFalling_IT :* specifies the polarityof the external interrupt : true to define rising edge trigerred external interrupt on the target IO, false to define a falling edge triggered interrupt.
- **Return Value:** -
- **Note :** *GPIO_TypeDef* is a type defined in the HAL library. Parameters of type *GPIO_TypeDef* can take values (aliased to) *GPIOA*, *GPIOB* or *GPIOC*.

- **Function** : `void LBF_Disable_ExtIT(GPIO_TypeDef* GPIO_Port, uint16_t GPIO_Pin);`
- **Description** : Disables the interrupt on the specified GPIO and sets this GPIO as regular input GPIO (without pull-up or pull-down)
- **Parameters** :
  > *GPIO_Port* : the port to which the IO defined as external interrupt belongs – can be (alias of) GPIOA, GPIOB or GPIOC
  > *GPIO_Pin* : the pin number in the port (0 to 15) of the IO defined as external interrupt
- **Return Value:** -
- **Note :** *GPIO_TypeDef* is a type defined in the HAL library. Parameters of type *GPIO_TypeDef* can take values (aliased to) *GPIOA*, *GPIOB* or *GPIOC*.

3.3 Timers :

- **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

<TO DO – Update timer numbering, this was for the L1 >

- **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 by LimiFrog API Timer and PWM functions

  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 by LimiFrog API Timer and PWM functions

- **Function** : `void **LBF_Timer_Setup**( LBF_TimerID_t TimerID, uint16_t Timer_TimeUnit_us, uint32_t Period_as_TimerUnits )`
- **Description** : For the timer identified by *TimerID,* defines the time unit (i.e. prescaled clock period) to use and the period of the timer.
- **Parameters** :
  > *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:** -

- **Function** : `void **LBF_Timer_Start_ITout**( LBF_TimerID_t TimerID )`
- **Description** : Starts the timer identified by *TimerID* with generation of an interrupt at the end of each period
- **Parameters** : *TimerID* : the identity (number) of the timer to set up
- **Return Value:** -

- **Function** : `void **LBF_Timer_Stop**( LBF_TimerID_t TimerID )`
- **Description** : Stops the timer identified by TimerID.
- **Parameters** : *TimerID* : the identity (number) of the timer to set up
- **Return Value:** -

EXAMPLES :

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

Timer #4 is initialized to operate with a 1.5 s period.

> `LBF_Timer_StartITout( TIMER4 );`

Start Timer #4 with generation of an IT each time the period elapses.

3.4 Services and Utilities

- Function : `void LBF_Delay_ms (volatile uint32_t nTime)`

  Description : Loops until the delay specified in milliseconds by *nTime* is elapsed.
- **Parameters** : *nTime* : the number of millisecond to wait
- **Return Value:** -

- **Function** : `void LBF_Led_StopNBlinkOnFalse( bool Status )`
- **Description** : Enters infinite loop with flashing LED is *Status* is false, else continue. Mostly for debug an 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:** -

+ *Embedded String Functions* :

`xprintf, etc.`

Compact string IO library provided by ChaN.

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

4 MIDDLEWARE

4.1 USB 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. LimiFrog API provides functions to start and stop USB in mass storgae mode with no further intervention by the application.

- **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:** -

4.2 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. LimiFrog API provides initialization 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.)

- **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

4.3 emWin Graphics Library

ST provides, under the name StemWin, an implementation of Segger's emWin professional graphics library ported to the STM32.

LimiFrog's software package includes the « glue » to interface this generic library with the OLED display controller. LimiFrog API provides an initialization function that allows to then use the emWin API to perform many graphics operation.

- **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_**.