Draft B

Work in progress

LimiFrog API

QUICK DOCUMENTATION

August 2015

Note: Some work is still required to clean-up API and align it with good software coding practices

TO DO :

> Implement and add return codes (success or error code) – Change existing return codes to align on usual conventions (0 : success, neg. value = error code)

> Document missing functions

> Implement better naming conventions

Library functions that pertain to LimiFrog are prefixed LF_ in this documentation. However, because LimiFrog (acronym LF) used to be called La BlueFrog (acronym LBF) before a recent name change, the prefix used in the actual code is still LBF_.

LimiFrog API consists of :

- board support functions :
  - > full **board initialization**
  - > **control of on-board hardware**
- a selection of **STM32 peripheral control functions** that work at higher level than the HAL library (building upon it)
- and functions to support ST-provided **middleware** on LimiFrog's hardware

1 FULL BOARD INITIALIZATION

These functions take care of configuring the STM32 low-level hardware for LimiFrog, as well as initializing other relevant on-board chips.

- **Function** : `void` **`LF_Board_Fixed_Inits`**`(void)`
- **Description** : Performs all « fixed » initializations, which do not depend on some user choices.
  Fixed initializations include :
  > configuring the SysTick time base and setting up the NVIC (Interrupt Controller)
  > configuring the clock generator
  > configuring all STM32 IO that have a fixed role in LimiFrog (example : I/Os used to support the I2C bus that goes to all sensors) – that's in fact all of them but those that are routed to the extension port (see next function below)
  > configuring all on-chip peripherals that play a fixed role in LimiFrog (example : I2C bus #2 is dedicated to interfacing with all on-board sensors)
- **Parameters** : -
- **Return values**: -

- **Function** : `void` **`LF_Board_Selective_Inits`**`(void)`
- **Description** : Performs all « selective » initializations, which depend on some user choices, expressed as directives in a configuration file (template provided).
  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
  The user typically provides his requirements through a file called *User_Configuration.h*
- **Parameters** : -
- **Return values:** -

- **Custom Type**: `typedef enum { FALSE = 0, TRUE = 1 }` **`boolean_t;`**

- **Description** : Boolean type, ued throughout the API

2 ON-BOARD HARDWARE CONTROL

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

2.1 Battery

- **Function** : `uint32_t` **`LF_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 values:** 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` **`LF_Led_ON`**`(void)`
- **Description** : Turns on the STM32-controlled LED located at the edge of the board
- **Parameters** : -
- **Return values:** -

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

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

2.3 Push-button switches

- **Function** : `boolean_t` **`LF_State_Switch1_IsOn`**`(void)`
- **Description** : Returns the state of push-button switch #1 located at edge of board
- **Parameters** : -
- **Return values:** TRUE if button being pushed, else FALSE

- **Function** : `boolean_t` **`LF_State_Switch2_IsOn`**`(void)`
- **Description** : Returns the state of push-button switch #2 located at edge of board
- **Parameters** : -
- **Return values:** 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` **`LF_OLED_Switch_ON`** `(void)`
- **Description** : Provides high voltage and turn display on with controlled latency as per display datasheet
- **Parameters** : -
- **Return values:** -

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

- **Function** : `void` **`LF_OLED_Clear`** `(void)`
- **Description** : Displays a uniform black screen
- **Parameters** : -
- **Return values:** -

- **Function** : `void` **`LF_OLED_Fill`**`(uint8_t x, uint8_t y, uint8_t width, uint8_t height, uint16_t color565)`
- **Description** : Fills the region starting at top left pixel of coordinates (*x,y*) with dimensions defined by *width* and *height* with RGB565 color defined by *color565.* Top left pixel of the screen area defines axis origin i.e. coordinates (0,0).
- **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
  > *color565 (In)*: code in RGB565 format of the color wth which to fill the region
- **Return values:** -

- **Function** : `void` **`LF_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 values:** -

******* 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` **`LF_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 values:** -

- **`Function :`** `void` **`LF_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 values:** -

- **Function** : `void` **`LF_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 values:** -

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

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

- **Function** : `void` **`LF_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 values:** -

- **Function** : `void` **`LF_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 values:** -

- **Function** : `void` **`LF_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 values:** -

- **Function** : `void **LF_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 values:** -

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

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

- **Function** : `void **LF_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 values:** -

- **Function** : `void` **`LF_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 values:** -

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

> 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.)

DOCUMENTATION TO WRITE

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 *`LF_Board_Selective_Inits()`* then takes care of downloading (over UART1, 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 #1 or over I2C bus #2.

3 STM32 PERIPHERAL CONTROL

NOTE :
ST provides the HAL (Hardware Abstraction Layer) library to manage access to all STM32 on-chip peripherals in a lot of different modes – it's very polyvalent but can therefore feel relatively complex to use at first.
LimiFrog simplifies usage of the HAL library by, first, taking care of initializing all the peripherals dedicated to a given role on LimiFrog (for example, I2C2 to communicate with the sensors) as well as configuring the IOs assigned to these peripherals – this is done as part of the `LF_Board_Init()` function formerly described.
In addition, the peripheral control functions described below allow to use commonly needed peripherals at a higher (simpler) level than with the HAL library (they build upon it).
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 not, 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 *LF_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** : `LF_GPIO_TOGGLE(GPIO_Port_Name , GPIO_Pin_Number)`
- **Description** : Toggles pin *GPIO_Pin_Number* of port *GPIO_Port_Name*.

- **Macro** : `LF_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.

- **Macro** : `LF_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.

3.2 External Interrupts :

- **Function** : `void` **`LF_Enable_ExtIT`**`(GPIO_TypeDef* GPIO_Port, uint16_t GPIO_Pin, boolean_t  Rising_nFalling_IT);`
- **Description** : Configures the specified GPIO as external edge-triggered interrupt source of specified polarity
- **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 values:** -
- **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` **`LF_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 values:** -
- **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 Timer and PWM :

- **Defined constants :**

```
#define LF_TIMER_UNIT_US   1
#define LF_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 {
  TIMER2 = 2,
  TIMER3 = 3,
  TIMER4 = 4,
  TIMER5 = 5,
  TIMER6 = 6,
  TIMER7 = 7,
  TIMER9 = 9,
  TIMER10 = 10,
  TIMER11 = 11
  }
  LF_TimerID_t;
  ```

- **Description** : Timer Identification, used by LimiFrog API Timer and PWM functions
  Only timers present in the STM32L151 are listed

TO DO : update for STM32L4

- **Custom Type**:

  ```
  typedef enum {
  CHANNEL1 = 1,
  CHANNEL2 = 2,
  CHANNEL3 = 3,
  CHANNEL4 = 4
  }
  LF_ChannelID_t;
  ```

- **Description** : Channel Identification within a given timer, used by LimiFrog API Timer and PWM functions

- **Function** : `void LF_Timer_Setup( 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 us. Can therefore be regarded as time unit for that timer. Aliases LF_TIMER_UNIT_US or LF_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 values:** -

- **Function** : `void` **`LF_Timer_Start_ITout`**`( 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 values:** -

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

- **Function** : `void` **`LF_PWMchannel_Setup`** `( TimerID_t TimerID, 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 values:** -

- **Function** : `void` **`LF_PWMChannel_Start`** `(TimerID_t TimerID, 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 values:*** -

- **Function** : `void` **`LF_PWMChannel_Stop`** `(TimerID_t TimerID, 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 values:*** -

- **Function** : `void` **`LF_PWMChannel_UpdatePulse`** `(TimerID_t TimerID, 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 values:** -

EXAMPLES :

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

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

`LF_Timer_Setup( TIMER3, TIMER_UNIT_MS, 150 );`

Timer #3 is initialized to operate with a 150 millisecond period.

> `LF_Timer_StartITout( TIMER4 );`

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

> `LF_PWMChannel_Setup (TIMER3, CHANNEL3, 10) ;`

On channel #3 of Timer #3, set-up a PWM generator with high pulse duration 10ms (as Timer #3 has been set-up to use milliseconds as base unit).

> `LF_PWMChannel_Start (TIMER3, CHANNEL3) ;`

Start generation of the PWM signal on channel #3 of timer #3

> `LF_PWMChannel_UpdatePulse (TIMER3, CHANNEL3, 20) ;`

Change pulse duration of PWM on channel3 of timer 3 to 20ms

3.4 Analog-to-Digital Converter

- **Custom Type**:

```
typedef enum {

 // ADC_CHANNEL_xx are defined in HAL ADC driver

ADC_CH_1 = ADC_CHANNEL_1,  // on PA5 = Pos 1 of extension port

ADC_CH_6 = ADC_CHANNEL_6,  // on PA5 = Pos 1 of extension port

ADC_CH_7 = ADC_CHANNEL_7,  // on PA5 = Pos 1 of extension port

ADC_CH_8 = ADC_CHANNEL_8,  // on PA5 = Pos 1 of extension port
ADC_CH_11 = ADC_CHANNEL_11 // on PA5 = Pos 1 of extension port
}
LF_ADC1_ChannelID_t;
```

- **Function** : `void` **`LF_ADC1_Init_Single_Shot`**`(void)`
- **Description** : Initializes ADC #1 for single shot conversion
- **Parameters** : –
- **Return values:** -

- **Function** : `void` **`LF_ADC1_Init_Single_Channel`**`(ADC1_ChannelID_t ChannelID )`
- **Description** : Sets up specified channel on ADC1 with default parameters : only 1 channel to sample, with sample time set to 16 cycles (could range from 4 to 384 cycles)
- **Parameters** :

  > ChannelID : one of the channels ADC1 that are actually available on LimiFrog – i.e. any channel supported by custom type LF_ADC1_ChannelID_t
- **Return values:** -

- **Function** : `uint32_t` **`LF_ADC1_Get_Value_Single_Shot`**`(void)`
- **Description** : Sample the analog value present on the previously configured channel of ADC1
- **Parameters** : -
- **Return values:** the sampled value (on 12 LSBs of 32-bit return argument)

- **Function** : `void` **`LF_ADC1_Deinit`**`(void)`
- **Description** : De-initializes ADC #1, incl. Stopping ADC clock
- **Parameters** : –
- **Return values:** -

- Note : HSI clock required by ADC not disabled in case it's being used by other ADC (not in L1, but possible in L4)

3.5 I2C Busses

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` **`LF_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 values:** -

- **Function** : `void` **`LF_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 values:** -

- **Function** : `uint8_t` **`LF_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 values:** the value read from the target register

- **Function** : `void` **`LF_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 values:** -

- **Function** : `void` **`LF_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 values:** -

3.6 UART

- **Function** : `void` **`LF_UART_SendByte`** `(UartID_t Uart_ID, uint8_t data)`

- **Description** : Send specified data byte over specified UART (with a default time out set to 100ms).
- **Parameters** :
  > *Uart_ID* : specifies target UART, can take value UART1 or UART3 (UART #2 not used in LimiFrog)

  > *data* : the data byte to send
- **Return values:** -

- **Function** : `void` **`LF_UART_ReceiveByte`** `(UartID_t Uart_ID, uint8_t data)`
- **Description** : Receives a data byte from specified UART (with a default time out set to 100ms).
- **Parameters** :
  > *Uart_ID* : specifies target UART, can take value UART1 or UART3 (UART #2 not used in LimiFrog)

  > *data* : the data byte received
- **Return values:** -

- **Function** : `void` **`LF_UART_SendString`** `(UartID_t Uart_ID, char* pString)`
- **Description** : Sends a string of characters over specified UART without any flow control (with a default time out set to 100ms on UART reception).
- **Parameters** :
  > *Uart_ID* : specifies target UART, can take value UART1 or UART3 (UART #2 not used in LimiFrog)

  > *pString*: pointer to the first character of the string
- **Return values:** -

- **Function** : `void` **`LF_UART_SendString_SwFlowControl`** `(UartID_t Uart_ID, char* pString)`
- **Description** : Sends a string of characters over specified UART (with a default time out set to 100ms), with software flow control (suspend transmission if XOFF received, resume

when XON received)

- **Parameters** :

  > *Uart_ID* : specifies target UART, can take value UART1 or UART3 (UART #2 not used in LimiFrog)

  > *pString*: pointer to the first character of the string

- **Return values:** -

***To be completed** with : hardware flow control support, with string reception, with data buffertransmission/reception*

3.7 SPI bus

…. to document

3.8 Services and Utilities

- Function : `void` **`LF_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 values:** -

- **Function** : `void` **`LF_Led_StopNBlinkOnFalse`**`( boolean_t Status )`
- **Description** : Enters infinite loop with flashing LED is *Status* is FALSE, else continue. Mostly for debug an diagnostic.
- **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 values:** -

+ *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** : `boolean_t` **`LF_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` **`LF_StopUSB_MassStorage`**`(void)`
- **Description** : Stops USB operation in Mass Storage mode.
- **Parameters** : -
- **Return values:** -

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** : `boolean_t` **`LF_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** : `boolean_t` **`LF_FatFS_DeInit`**`(void)`
- **Description** : De-initializes the FatFs file system, driver is unlinked.
- **Parameters** : -
- **Return values:** 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** : `boolean_t` **`LF_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_**.