User manual

Getting started with the IO-Link demonstration kit firmware

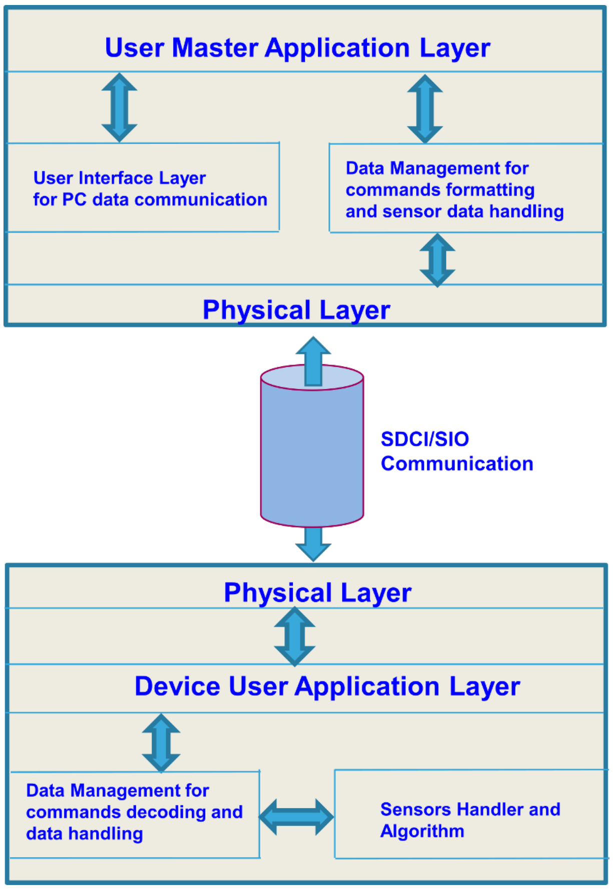
Introduction

This kit includes a complete firmware release for STEVAL-IDP004V1 and STEVAL-IDP003V1 to facilitate system management in terms of configuration, data exchange and processing. It is based on the STM32CubeHAL library and uses the powerful STM32CubeMX tool to configure the microcontroller and update the created workspace without data loss.

A simple command interface lets you manage PC and system configuration using the RS 485 interface and Master-Device communication between STEVAL-IDP004V1 and STEVAL-IDP003V1.

Data exchange with a sensor is implemented via on-board data acquisition through an I²C interface. Data processing (i.e., vibration monitoring with the IIS2DH sensor) is implemented through dedicated routines using the DSP_Lib to perform Fast Fourier Transform (FFT) calculations.

Figure 1. STSW-IO-LINK block diagram


User Master Application Layer
User Interface Layer for PC data communication
Data Management for commands formatting and sensor data handling
Physical Layer
SDCI/SIO Communication
Physical Layer
Device User Application Layer
Data Management for commands decoding and data handling
Sensors Handler and Algorithm

1 STM32Cube architecture

The STM32Cube firmware solution is built around three independent levels that can easily interact with one another, as described in the diagram below.

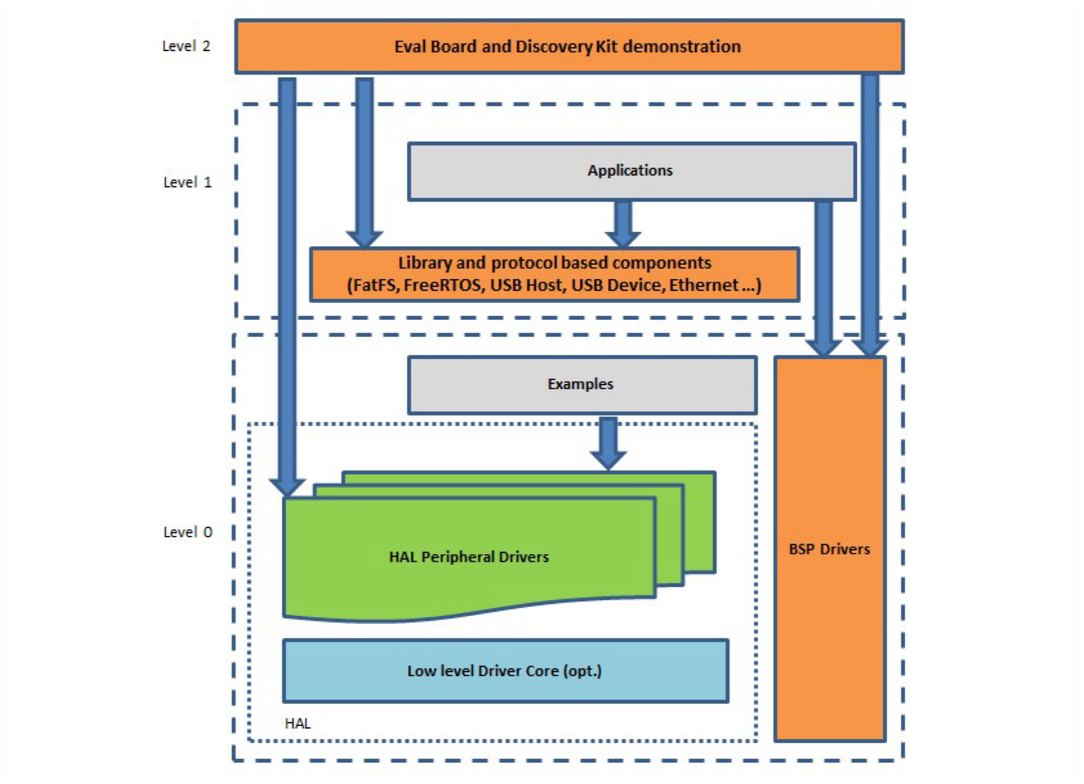

Level 2
Eval Board and Discovery Kit demonstration
Level 1
Applications
Library and protocol based components
(FatFS, FreeRTOS, USB Host, USB Device, Ethernet ...)
Examples
Level 0
HAL Peripheral Drivers
Low level Driver Core (opt.)
HAL
BSP Drivers

Figure 2. **Firmware architecture**

Level 0: This level is divided into three sub-layers:

- Board Support Package (BSP): this layer offers a set of APIs relative to the hardware components in the hardware boards (Audio codec, IO expander, Touchscreen, SRAM driver, LCD drivers. etc…); it is based on modular architecture allowing it to be easily ported on any hardware by just implementing the low level routines. It is composed of two parts:
  - Component: is the driver relative to the external device on the board and not related to the STM32, the component driver provides specific APIs to the external components of the BSP driver, and can be ported on any other board.
  - BSP driver: links the component driver to a specific board and provides a set of easy to use APIs. The API naming convention is BSP_FUNCT_Action(): e.g., BSP_LED_Init(), BSP_LED_On().
- Hardware Abstraction Layer (HAL): this layer provides the low level drivers and the hardware interfacing methods to interact with the upper layers (application, libraries and stacks). It provides generic, multi-instance and function-oriented APIs to help offload user application development time by providing ready to use processes. For example, for the communication peripherals (I²C, UART, etc.) it provides APIs for peripheral initialization and configuration, data transfer management based on polling, interrupt or DMA processes, and communication error management. The HAL Drivers APIs are split in two categories: generic APIs providing common, generic functions to all the STM32 series and extension APIs which provide special, customized functions for a specific family or a specific part number.
- Basic peripheral usage examples: this layer houses the examples built around the STM32 peripherals using the HAL and BSP resources only.

Level 1: This level is divided into two sub-layers:

- Middleware components: set of libraries covering USB Host and Device Libraries, STemWin, FreeRTOS, FatFS, LwIP, and PolarSSL. Horizontal interaction among the components in this layer is performed directly by calling the feature APIs, while vertical interaction with low-level drivers is managed by specific callbacks and static macros implemented in the library system call interface. For example, FatFs implements the disk I/O driver to access a microSD drive or USB Mass Storage Class.
- Examples based on the middleware components: each middleware component comes with one or more examples (or applications) showing how to use it. Integration examples that use several middleware components are provided as well.

Level 2: This level is a single layer with a global, real-time and graphical demonstration based on the middleware service layer, the low level abstraction layer and basic peripheral usage applications for board-based functions.

2 STEVAL-IDP004V1 firmware structure

The firmware structure is created with the STM32CubeMX tool, which lets you configure the microcontroller and generate the corresponding code repeatedly, without losing any programming code developed to handle the system.

The generated project solution has the following structure:

- **Application level**:
  - EWARM folder: containing the startup file
  - USER folder: containing main and interrupt routines
- **Driver level**:
  - CMSIS folder: containing source file library for the microcontroller core
  - STEVAL-IDP004V1 folder: customized folder based on the application
    - **PC_Communication_RS485.c**: source files to manage RS485 communication
    - **Master_Settings.c**: to configure L6360 IC
    - **Master_DeviceCOMM.c**: to manage data exchange with sensor node
  - STM32F2xx_HAL_DRIVERS: containing all HAL library source files for internal peripherals, memory and GPIO handling.

2.1 Application level

The application level contains the source files to configure the microcontroller and provide a front-end layer to manage the process flow.

From this level, you can recall the routines in the Driver Level to handle different parts of communication.

The source files used in this layer are:

- **Main.c**: groups all the routines used to initialize the peripherals and to manage the access to the data processing routines.
- **stm32f2xx_it.c**: manages the systick interrupt for PC communication time out and peripheral interrupts.
- **stm32f2xx_hal_msp.c**: source file for NVIC configuration and to enable peripherals.

2.2 Driver level

This level contains the HAL, CMSIS or user library routines used to handle microcontroller settings, data processing and user command management.

2.3 STEVAL-IDP004V1 user library

2.3.1 PC_Communication_RS485.c

This source file is used to manage communication with the PC using Tera Term (HyperTerminal interface) for STEVAL-IDP004V1 configuration and data exchange with the STEVAL-IDP003V1D board hosting the L6362A device (part of the STEVAL-IDP003V1 kit)

This source file has the commands and string messages to be exchanged during communication with PC and the sensor node, and the routines needed to manage the communication.

2.3.1.1 *String messages returned to PC from STEVAL-IDP004V1*

1. Presentation string message; it is sent back to the PC after the user enters the `START` command.

```
uint8_t SpaceString[]="";
uint8_t PresentationMessage0[]="*********************************\r\n";
uint8_t PresentationMessage1[]="*       STMicroelectronics      *\r\n";
uint8_t PresentationMessage2[]="* IO - Link MULTI-PORT SOLUTION *\r\n";
```

```
uint8_t PresentationMessage3[]="*********************************\r\n";
uint8_t StrRequest[]="\r\nSELECT MASTER OR DEVICE\r\n";
```

2.
```
uint8_t StrRequestIC []="\r\nINSERT ADDRESS IC :\r\n"
```

String message sent back to the PC after the user has specified the IC by writing `MASTER`, using a single digit from 0 to 3

3.
```
uint8_t StrRequestOPM[]="\r\nINSERT OPERATING MODE :\r\n"
```

String message sent back to the PC after the user has specified the IC address if the previously selected device is MASTER.
The user must specify one of the following user commands: WR_C, R_S, RD_C or RD_S.

4.
```
uint8_t StrRequestRADRS []="\r\nINSERT REGISTER ADDRESS :\r\n"
```

String message sent back to the PC after the user specifies the operating mode using the user commands (message available only for master programming). Insert:
- **1** if the user command is WR_S
- **no number** if the user command is RD_C
- a number **between 0 and 7** if the user command is RD_S
- a number **between 1 and 7** if the user command is WR_C register address.

5.
```
uint8_t StrRequestRVAL []="\r\nINSERT REGISTER VALUE :\r\n"
```

String message sent back to the PC (if writing commands were chosen), after the user specifies the address register value.
Insert three digit sequence for each value in the range 000 to 255, and separate each register value using the ‘`,`’ (comma) symbol. Complete the sequence with a ‘,’ even if the value is only one.

6. String message sent back to the PC if a Read operation was selected with the user command.

```
uint8_t Status[]="\r\nStatus register";
uint8_t Configuration[]="\r\nConfiguration register";
uint8_t Control1[]="\r\nControl 1";
uint8_t Control2[]="\r\nControl 2";
uint8_t LED_1H[]="\r\nLED 1 MSB";
uint8_t LED_1L[]="\r\nLED 1 LSB";
uint8_t LED_2H[]="\r\nLED 2 MSB";
uint8_t LED_2L[]="\r\nLED 2 LSB";
uint8_t Parity[]="\r\nParity Register";
```

Regarding the above list, if the user command is:
- `RD_C`, only the first string is sent
- `RD_R`, the whole string is listed based on the start address

7.
```
uint8_t MS_StrRequestADDR[]="\r\nINSERT SLAVE NODE :\r\n"
```

String message sent back to the PC after the user has specified the IC by writing `DEVICE` and entering the node address with a single number from 0 to 3. This number identifies the corresponding master node and the USART port selected for the communication.

8.
```
uint8_t MS_StrRequestCMD[]="\r\nINSERT SENSOR COMMAND :\r\n"
```

String message sent back to the PC after the slave node is inserted. Write one of the user commands listed in PC_Communication.c.

9.
```
uint8_t MS_StrFeedbackCMD[]="\r\nNO SENSOR :\r\n"
```

String message send back to the PC if the sensor is not connected.

10.
```
uint8_t MS_StrFeedbackMSG[]="\r\nPROGRAMM MASTER NODE BEFORE COMMUNICATION :\r
\n"
```

String message sent back to the PC if the node selected for the master device communication is not programmed.

11.
```
uint8_t MS_StrPRX_Var[]="\r\nPOSITION :";
uint8_t MS_StrTMP_Var[]="\r\nTEMPERATURE :";
uint8_t MS_StrACL_Var[]="\r\nACCELERATION :";
uint8_t MS_StrVIBR_Var[]=”\r\nVIBRATION :";
```
These string messages are sent back to the PC when you send commands to the device node to retrieve parameter values. Based on the sensor, one of these strings will be displayed before the number.

2.3.1.2 *Commands for STEVAL-IDP004V1 and STEVAL-IDP003V1 communication*

1.
```
uint8_t CmdSTART []="START"
```
PC communication Init

2.
```
uint8_t CmdWRC[]="WR_C"
```
Send request for L6360 Write Current procedure

3.
```
uint8_t CmdRDC[]="RD_C"
```
Send request for L6360 Read Current procedure

4.
```
uint8_t CmdWRS[]="WR_S"
```
Send request for L6360 Sequential Write

5.
```
uint8_t CmdRDS[]="RD_S"
```
Send request for L6360 Sequential Read

6.
```
uint8_t CmdICM[]="MASTER"
```
Select master board device

7.
```
uint8_t CmdICD[]="DEVICE"
```
Select device board

8.
```
uint8_t CmdNEW[]="COMMAND NEW"
```
Send request for new programming phase

9.
```
uint8_t CmdEND[]="COMMAND END"
```
Send request to close actual programming phase

10.
```
uint8_t MS_CmdID[9]="IDS"
```
Send request to get sensor identification (this command relates to master-device communication)

11.
```
uint8_t MS_CmdPRM[9]="PRM"
```
Send request to get the results of computation analysis on measured values (this command relates to master-device communication).

12.
```
uint8_t MS_CmdER[9]="SENSOR ERR"
```
Send request to retrieve sensor error code (this command relates to master-device communication, for the proximity sensor only).

2.3.1.3 *Routines used to manage the communication with PC*

1.
```
void User_Command (void)
```
Called at the main level when a frame has been received from the PC.

2.
```
void RS485_Decode(void)
```
Allows the decode command from the PC.

3.
```
void RS485_TX (uint8_t* RS485_data,uint8_t data_size)
void RS485_RX (uint8_t data_size)
```
These routines manage the RS485 communication.

4.
```
void Convert_ToAscii (SlaveNodeStruct_Typedef* Sdata,uint8_t n_byte,uint8_t no
de)
```
Converts the sensor data in ASCII format and send the result to the PC.

5.
```
void Reset_RS485_COM (void)
```
Resets PC communication if a command is wrong.

6.
```
void Drive_CommPin(uint8_t transm)
```
Manages the logic level on the DE pin of the RS 485 transceiver.

7.
```
void Print_Sensors_Info(void)
```
Transmits the sensor ID and parameter name for each sensor to the PC.

8.
```
void PrintRegister (uint8_t* string)
```
Transmits the L6360 register value to the PC after a read command.

9.
```
void I2C_Print_FB (void)
```
Transmits feedback related to an I²C operating mode.

Note: *In the USB version, the RS485_TX, RS485_RX, RS485_Decode routines, become USB_TX,USB_RX, RX_USB_Decode; the source file is PC_Communication_USB.c.*

2.3.2 Master_Setting.c

This source file groups all the routines used for master access mode (writing or reading) and parity calculation.

1.
```
void Master_Programming(void)
```
Used to address the L6360 programming procedure.

2.
```
int8_t Master_CurrentWrite(uint8_t M_address,DirectionMode_Typedef I2C_DIR, ui
nt8_t* master_reg, uint8_t register_pos )
```
Performs the Single writing access procedure to the master IC.

3.
```
int8_t Master_SequentialWrite(uint8_t M_address,DirectionMode_Typedef I2C_DIR,
 uint8_t* master_reg,uint8_t start_pos)
```
Performs the sequential writing access mode; in this case, the starting register address is one, corresponding to the Configuration register.

4.
```
uint8_t Get_MasterStatus(uint8_t M_address,uint8_t start_pos,uint8_t* get_reg)
```
Used for Status register reading, to get feedback regarding device and IO-Link bus failures.

5.
```
uint8_t Get_MasterConfig(uint8_t M_address,DirectionMode_Typedef I2C_DIR, uint
8_t* master_config, uint8_t start_pos)
```
Performs sequential or random register reading, to retrieve the programmed register.

6.
```
uint8_t WC_Parity_Calc(uint8_t in_data)
```
Calculates the partity byte to be sent when a Current Write operation is required.

7.
```
uint8_t WS_Parity_Calc(uint8_t* in_data)
```
Calculates the partity byte to be sent when a Sequential Write operation is required.

8.
```
void Master_Reset(uint8_t reset)
```
This routine is used to reset all master ICs.

9.
```
int8_t Master_EvaluateStatus(uint8_t node)
```
This routine is used to get feedback on master node programming status (programmed on not).

10.
```
void Master_DefaultStatus (void)
```
This routine is used to set the default L6360 configuration.

2.3.3 Master_DeviceCOMM.c

This source file has all the routines used to handle the IO-Link physical layer communication.

1.
```
int8_t IO_LINK_SEND (uint8_t address,uint8_t* data_byte,uint8_t txdata_lenght,
uint8_t rxdatalength)
```
Manages IO-Link physical layer communication.

2.
```
void Master_TxMode(uint8_t address)
```
Handles the ENCQ pin in transmission mode.

3.
```
void Master_RxMode(uint8_t address)
```
Handles the ENCQ pin in receive mode.

4.
```
int8_t Get_Slave_FB(uint8_t var,uint8_t length)
```
Decodes the information received by the device after a user command.

5.
```
int8_t Get_ReceiverFlag(void)
```
Get receiver flag value.

6.
```
void Master_LPlus_LineOn(uint8_t lplus)
```
Enables the supply line for the sensor.

7.
```
void Reset_ReceiverFlag(void)
```
Resets receiver flag.

8.
```
void Send_Waiting_For_Command(void)
```
Sends a frame to the slave, to signalize that a command will be sent.

9.
```
void Send_Command(void)
```
Sends the command requested to the slave.

3 STEVAL-IDP003V1 firmware structure

The Firmware structure has been created, using the STM32 CUBE-MX tool developed by ST, which permits to configure the microcontroller, and generate the corresponding code even several times without losing of any programming code developed by the user to handle the system.

The generated project solution shows the following structure:

- Application level:
  – EWARM folder: containing the startup file
  – USER folder: containing main routines and interrupt routines
- Driver level with:
  – BSP
    ◦ Components: this folder contains all the source files used for sensor programming and data processing
    ◦ STEVAL- IDP003V1D: this folder contains the routines used for IO-Link communication (the protocol stack is not implemented)
  – CMSIS folder: contains the source file library for the microcontroller core
  – STM32F2xx_HAL_DRIVERS: contains all the HAL library source files for internal peripherals, memory and GPIO handling.
- Middleware: contains all the routines used for Fast Fourier Transform (FFT) calculations

3.1 Application level description

This level contains the source files used to configure the microcontroller and provide a front-end layer to manage the process flow.

From this level, you can call the routines in the Driver Level to handle different parts of the communication.

3.1.1 Main.c

This source file contains the routines used to initialize the peripherals, grouped thus:

- IO-Link communication master-device communication
- Sensor configuration
- Interrupt for sensor data ready when MEMS is connected

3.1.2 stm32f2xx_it.c

This source file handles interrupt routines based on the connected sensor.

3.1.3 stm32f2xx_hal_msp.c

Source file for NVIC configuration and peripheral enable.

3.2 Driver level

This level contains the HAL library and user library routines used to handle the microcontroller settings, demonstration kit communication and sensor data acquisition and processing.

3.2.1 Components folder

This folder contains the source files for sensor configuration and data processing. In particular, the structure has a main source used to address all sensors called Sensor.c; is is based on the on-board sensors and points to different source file examples:

- VL6180X.c for proximity sensor management
- IIS2DH.c for vibration sensor management
- IIS328DQ.c for accelerometer management

- STTS751.c for temperature sensor management

3.2.1.1 *VL6180X.c description*

This source code contains the functions to configure and manage data processing for the proximity sensor:

1.
   ```
   int8_t VL6180x_WaitDeviceBooted(uint8_t dev)
   ```
   For device boot at startup.

2.
   ```
   HAL_StatusTypeDef VL6180X_DeInit(void)
   ```
   Sensor DeInit.

3.
   ```
   int8_t VL6180X_Calibration_Offset(uint8_t dev)
   ```
   Sensor calibration.

4.
   ```
   void VL6180X_StartStop_Acquisition(uint8_t value)
   ```
   Sensor data sampling.

5.
   ```
   int8_t VL6180X_Read_Register(uint8_t dev,uint16_t reg_addr,uint8_t* var,uint8_
   t nbyte)
   ```
   Used to perform a register access.

6.
   ```
   uint8_t VL6180X_GetStatus_Error(uint8_t* data)
   ```
   Store error code in the sensor structure.

7.
   ```
   int8_t VL6180X_GetRange
   ```
   Used to read the data register.

8.
   ```
   uint8_t VL6180X_GetStatus_Error (uint8_t* data)
   ```
   Used to get the status error code.

9.
   ```
   in8_t VL6180X_Read_Status_Data (void)
   ```
   Perform the IC read access to read error code.

3.2.1.2 *IIS328DQ.c description*

This source code contains the functions to configure and manage data processing for the accelerometer sensor:

1.
   ```
   int8_t IIS328DQ_Reboot(uint8_t Device)
   ```
   Sensor reboot.

2.
   ```
   int8_t IIS328DQ_Init(uint8_t Device,uint8_t* Reg_Data,uint8_t num_byte)
   ```
   Sensor Init.

3.
   ```
   uint8_t IIS328DQ_Get_Sensitivity (uint8_t Device)
   ```
   Sensor get sensitivity.

4.
   ```
   uint8_t IIS328DQ_Get_ODR(uint8_t Device)
   ```
   Sensor get ODR.

5.
   ```
   int8_t IIS328DQ_Get_Axes(uint8_t Device)
   ```
   Used to manage data acquisition.

6.
   ```
   int8_t IIS328DQ_Write_Data(uint8_t Device,uint8_t* Reg_Data,uint8_t num_byte)
   ```
   Called when a write sensor access is performed.

7.
   ```
   int8_t IIS328DQ_Read_Data (uint8_t Device,uint8_t* Reg_Data,uint8_t* data_out,
   uint8_t num_byte)
   ```

Called when a read sensor access performed.

8. `void IIS328DQ_Change_Axis(void)`

   Called to switch from one axis to another.

3.2.1.3 *IIS2DH.c description*

This source code contains the functions to configure and manage data processing for vibration sensor.

1. `int8_t IIS2DH_Reboot(uint16_t Device)`

   Sensor reboot routine.
2. `int8_t IIS2DH_Init(uint16_t Device,uint8_t* Reg_Data,uint16_t num_byte)`

   Sensor Init.
3. `int8_t IIS2DH_AxisOnOff(uint16_t Device,uint8_t* Reg_Data,uint16_t num_byte)`

   Switch ON/OFF the axis before FFT calculation.
4. `uint8_t IIS2DH_Get_Sensitivity (uint16_t Device)`

   Read the sensitivity value of the sensor.
5. `uint8_t IIS2DH_Get_ODR(uint16_t Device)`

   Read ODR value of the sensor.
6. `int8_t IIS2DH_Evaluate_ODR(uint16_t value)`

   Store the calculated ODR value in the sensor structure.
7. `int8_t IIS2DH_Get_Axes(uint16_t Device)`

   Manage data acquisition.
8. `int8_t IIS2DH_Get_Axes_H(uint16_t Device)`

   Read the high part of the data register.
9. `int8_t IIS2DH_Write_Data(uint16_t Device,uint8_t* Register,uint16_t num_byte)`

   Called when a write sensor access is performed.

10.
    ```
    int8_t IIS2DH_Read_Data (uint16_t Device,uint8_t* Register,uint8_data_out,uint
    16_t num_byte)
    ```

    Called when a read sensor access is performed.
11. `void IIS2DH_FFT_Calling(float32_t* buffer)`

    Call the FFT calculation function and change the axis.
12. `uint8_t Get_FlagFFTdone (void)`

    Get FFT flag status.

3.3 STEVAL-IDP003V1D folder

3.3.1 Master_DeviceCOMM.c description

This source file contains all the routines used for PHY management and data packaging and decoding.

1. `void L6362A_IO_Init( uint8_t IO_Link_bus)`

   This routine allows GPIOs configuration for EN/DIAG and OL diagnostic pin of L6362A
2. `void BSP_GPIS_ENDIAG (uint8_t status)`

This routine allows EN/DIAG pin driving for TX/RX procedure

3.
```
uint8_t Frame_BUS_Decode(uint8_t* bus_data)
```
   This routine decodes the command frame type (i.e. GET_SENSOR_TYPE)
4.
```
void IO_Link_Data_Manager(void)
```
   This routine allows data packaging
5.
```
void IO_LINK_SEND(uint8_t * dataTX,uint8_t lenght)
```
   This routine is used to send frames
6.
```
void IO_LINK_RECEIVE(uint8_t * dataRX,uint8_t lenght)
```
   This routine is used to receive frames

3.4 Middleware

This level contains the routines implemented using the DSP library to perform the FFT calculations when the vibration sensor is connected to the STEVAL-IDP003V1D board. All the routines are grouped in the FFT.c source file.

3.4.1 FFT.c

The routines implemented in this source file are related to FFT calculation using the DSP library as well as the final vibration frequency calculation and its amplitude for each axis.

1.
```
void FFT_AclCalc (float32_t* dataIN_tmp,uint8_t axis_num)
```
   The main routine used to calculate the FFT output array.
2.
```
void Evaluate_MaxAmplitude_Range (FFT_Out_TypedefStruct* FFT_Output,uint16_t f
ft_psize,uint8_t axis_num)
```
   This routine performs a windowing process to find the component with a maximum value in a defined window during calibration phase and create a reference mask for noise.
3.
```
void Calculate_ABS_FFTprm(FFT_Out_TypedefStruct* FFT_Output,uint16_t fft_psize
)
```
   Performs a comparison in RUN between the real-time components and the mask obtained with the routine in point 2.
4.
```
void Print_FFT (FFT_Out_TypedefStruct* FFT_Output,uint16_t fft_psize,uint8_t a
xis_num)
```
   This routine calculates the real vibration frequency and its amplitude in m/s² for each axis and stores the value in the sensor structure.

Revision history

Table 1. **Document revision history**

Date	Version	Changes
07-Jun-2017	1	Initial release.
05-Sep-2017	2	Added Figure 1. STSW-IO-LINK block diagram.
26-Jun-2018	3	Updated Figure 1. STSW-IO-LINK block diagram, Section 2.3.1.2 Commands for STEVAL-IDP004V1 and STEVAL-IDP003V1 communication, Section 2.3.1.3 Routines used to manage the communication with PC and Section 2.3.3 Master_DeviceCOMM.c. Added Section 3.3.1 Master_DeviceCOMM.c description.

Contents

List of tables

List of figures