Getting started with MotionXLF high-g low-g fusion library in X-CUBE-MEMS1 expansion for STM32Cube

Introduction

The MotionXLF is a library that runs on STM32 and high-g sensor (for example LSM6DSV80X or LSM6DSV320X) inertial measurement unit (IMU), which features a 3-axis digital high-g accelerometer, 3-axis digital low-g accelerometer, and finite state machine (FSM). It fuses data from the low-g accelerometer and high-g accelerometer.

The algorithm is provided in static library format and is designed to be used on STM32 microcontrollers based on the ARM® Cortex® -M0+, ARM® Cortex®-M3, ARM® Cortex®-M33, ARM® Cortex®-M4, or ARM® Cortex®-M7 architectures.

It is built on top of STM32Cube software technology to ease portability across different STM32 microcontrollers.

1 Acronyms and abbreviations

Table 1. List of acronyms

Acronym	Description
API	Application programming interface
BSP	Board support package
GUI	Graphical user interface
HAL	Hardware abstraction layer
IDE	Integrated development environment

2 MotionXLF middleware library in X-CUBE-MEMS1 software expansion for STM32Cube

2.1 MotionXLF overview

The MotionXLF library expands the functionality of the X-CUBE-MEMS1 software.

The MotionXLF (accelerometer fusion) library combines the complementary properties of the low-g accelerometer core (high resolution, poor dynamic range) and high-g accelerometer core (poor resolution, high dynamic range). The library decides when to activate the high-g accelerometer to conserve power.

Moreover, the library handles time-varying offset evolution, discontinuity, and noise profile inconsistency between the low-g and the high-g accelerometer cores, providing a smooth transition of data between the two sensor cores.

This library is intended to work with the high-g sensors only (for example LSM6DSV80X or LSM6DSV320X). Functionality and performance when using other MEMS sensors are not analyzed and can be significantly different from what is described in the document. Furthermore, the library requires the FSM feature to be available on the selected sensor.

A sample implementation is available on X-NUCLEO-IKS02A1, X-NUCLEO-IKS5A1 and X-NUCLEO-IKS4A1 expansion boards on a NUCLEO-F401RE, NUCLEO-L073RZ, NUCLEO-L152RE, or NUCLEO-U575ZI-Q development board.

2.2 MotionXLF library

Technical information fully describing the functions and parameters of the MotionXLF APIs can be found in the MotionXLF_Package.chm compiled HTML file located in the Documentation folder.

2.2.1 MotionXLF library description

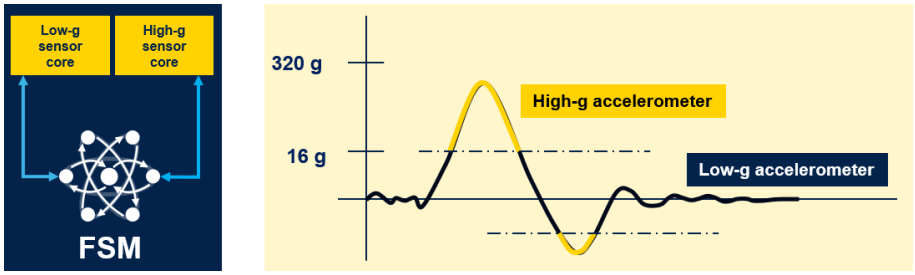
The MotionXLF library fuses data from high-g and low-g accelerometers; it features:

- Static offset compensation up to 500 mg, evolving offset compensation up to 50 mg.
- Noise density improvement of the high-g sensor up to 3.5 mg/√Hz.
- Automatic discontinuity handling when switching between low-g and high-g sensor.
- Sensor data sampling frequency from 480 Hz to 7.68 kHz.
- Resources requirements:
  - Cortex®-M0+: 6.0 kB of code and up to 0.2 kB of data memory.
  - Cortex®-M33: 5.0 kB of code and up to 0.2 kB of data memory.
  - Cortex®-M3: 5.9 kB of code and up to 0.2 kB of data memory.
  - Cortex®-M4: 5.5 kB of code and up to 0.2 kB of data memory.
  - Cortex®-M7: 5.3 kB of code and up to 0.2 kB of data memory.
- Available for ARM® Cortex® -M0+, Cortex®-M3, Cortex®-M33, Cortex®-M4, or Cortex®-M7 architectures.

2.2.2 MotionXLF library operation

The MotionXLF (accelerometer fusion) library combines the complementary properties of the low-g accelerometer core (high resolution, poor dynamic range) and high-g accelerometer core (poor resolution, high dynamic range). The library decides when to activate the high-g accelerometer to conserve power.

Figure 1. MotionXLF library functionality



The FSMs (Finite State Machines) are used to detect when the low-g and high-g sensor crosses or fall below a certain threshold. This information is used by the MotionXLF library to decide whether to pass low-g sensor output as is, whether to use low-g and high-g output, and whether to pass high-g sensor after noise and offset compensation.

2.2.3 MotionXLF library parameters

The following shows the types that are defined in the header file of the library.

```
typedef struct XLF_algo_settings
{
  int32_t ACCEL_FUSION_CALIB_WINDOW_SIZE;
  int32_t ACCEL_FUSION_DIGITAL_OUTPUT;
  int32_t ACCEL_FUSION_ENABLE_NOISE_REMOVAL;
  int32_t ACCEL_FUSION_ENABLE_DISCONTINUITY_REMOVAL;
  int32_t ACCEL_FUSION_ENABLE_OFFSET_CALCULATOR;
  int32_t ACCEL_FUSION_CONTINUOUS_TRACKING;
  int32_t ACCEL_FUSION_LOWER_THRESHOLD;
} XLF_algo_settings;
```

- `ACCEL_FUSION_CALIB_WINDOW_SIZE` - Number of samples to use for static offset compensation between the high-g and low-g accelerometer when the library first starts (default 50 samples).
- `ACCEL_FUSION_DIGITAL_OUTPUT` - Enable or disable digital output (20-bit) from the library (default 0, possible values: 1 or 0).
- `ACCEL_FUSION_ENABLE_NOISE_REMOVAL` - Enable or disable noise profile inconsistency handling between high-g and low-g accelerometer (default 1, possible values: 1 or 0).
- `ACCEL_FUSION_ENABLE_DISCONTINUITY_REMOVAL` - Enable or disable discontinuity handling between high-g and low-g accelerometer (default 1, possible values: 1 or 0)
- `ACCEL_FUSION_ENABLE_OFFSET_CALCULATOR` - Enable or disable time-evolving offset handling between high-g and low-g accelerometer (default 1, possible values: 1 or 0).
- `ACCEL_FUSION_CONTINUOUS_TRACKING` - If 1, then do not turn off the high-g accelerometer when not needed (default 0, possible values: 1 or 0).
- `ACCEL_FUSION_LOWER_THRESHOLD` - Full scale of low-g accelerometer (in mg) above which high-g accelerometer should ideally take over (default: 16000).

```
typedef void *XLF_Instance_t;
```

- pointer to the library instance loaded in data memory

```
typedef enum
{
  MOTION_XLF_MCU_STM32 = 0,
  MOTION_XLF_MCU_BLUE_NRG1,
  MOTION_XLF_MCU_BLUE_NRG2,
  MOTION_XLF_MCU_BLUE_NRG_LP,
} XLF_mcu_type_t;
```

- used MCU type

```
typedef enum
{
  MOTION_XLF_ALGO_OK = 0x00,         /* No error */
  MOTION_XLF_ALGO_ERROR = 0x01,      /* Algorithm error */
  MOTION_XLF_ALGO_READY = 0x02,      /* Algorithm is ready */
  MOTION_XLF_ALGO_NOT_READY = 0x03, /* Algorithm is not ready */
} XLF_return_t;
```

- library status – error code returned by API functions

```
typedef struct
{
  float x; /* Units: acc: [mg], gyr: [dps], mag: [micro tesla] */
  float y; /* Units: acc: [mg], gyr: [dps], mag: [micro tesla] */
  float z; /* Units: acc: [mg], gyr: [dps], mag: [micro tesla] */
} XLF_imu_data_t;
```

- Inertial measurement unit data in float format structure

```
typedef struct
{
  int x[20]; /* binary array 20-bit */
  int y[20]; /* binary array 20-bit */
  int z[20]; /* binary array 20-bit */
} XLF_imu_data_digital_t;
```

- Inertial measurement unit data in 20-bit format structure

```
typedef struct
{
   XLF_imu_data_t low_g_data_mg;  /* Data from low-g sensor in mg */
   XLF_imu_data_t high_g_data_mg; /* Data from high-g sensor in mg */
   char FSM_OUT1; /* Value in FSM_OUTS1 register (output of FSM1) */
   char FSM_OUT2; /* Value in FSM_OUTS2 register (output of FSM2) */
} XLF_in_t;
```

- Library input data structure

```
typedef struct
{
   XLF_imu_data_t fused_imu; /* Library output in mg (float) */
   XLF_imu_data_digital_t fused_imu_digital; /* Library output in 20-bit binary */
} XLF_out_t;
```

- Library output data structure

```
typedef void (*XLF_high_g_enable_disable_pointer_t)(void);
```

- Function pointer typedef to be used to pass functions that enable or disable the high-g sensor to the library.

2.2.4 MotionXLF APIs

The following shows the API functions that are defined in the header file of the library.

```
uint8_t MotionXLF_GetLibVersion(char *version)
```

- Retrieve the version of the library
- *version* – pointer to an array of 35 characters
- Return the number of characters in the version string

```
void MotionXLF_Initialize(XLF_mcu_type_t mcu_type)
```

- Perform MotionXLF library initialization and setup of the internal mechanism

Note: This function must be called before using the accelerometer fusion library and the CRC module in the STM32 microcontroller (in RCC peripheral clock enable register) has to be enabled.

- *mcu_type* – type of MCU used

```
XLF_return_t MotionXLF_Start(void)
```

- Start the MotionXLF engine
- Return an error code

```
XLF_return_t MotionXLF_Reset(void);
```

- Reset the state of the library including all buffers
- Return an error code

```
XLF_return_t MotionXLF_Update(XLF_in_t *data_in, XLF_out_t *data_out,
XLF_high_g_enable_disable_pointer_t enable_high_g,
XLF_high_g_enable_disable_pointer_t disable_high_g,
XLF_algo_settings *algo_set
);
```

- Execute one step of the algorithm
- *data_in* – pointer to the input data structure
- *data_out* – pointer to the output data structure
- *enable_high_g* – function pointer for enabling high-g sensor
- *disable_high_g* – function pointer for disabling high-g sensor
- *algo_set* – pointer to the algorithm settings
- Return an error code

2.2.5 Enabling or disabling the high-g sensor

The following functions have to be implemented specifically for each target platform to pass to the MotionXLF_Update() function using platform-independent drivers for the selected high-g sensor.

Enable high-g sensor

- An example for the LSM6DSV320X sensor that enables the high-g sensor and sets its ODR is shown below for the STM32U5 microcontroller using LSM6DSV320X platform-independent drivers

```
static void enable_high_g(void)
{
 LSM6DSV320X_Object_t *pObj = MotionCompObj[AccInstance];
 lsm6dsv320x_hg_xl_data_rate_t hg_data_rate = HgDataRate;
 (void)lsm6dsv320x_hg_xl_data_rate_set(&(pObj->Ctx), hg_data_rate, 1);
 HighGEnable = 1; // external flag
}
```

Disable high-g sensor

- An example that disables the high-g sensor and sets its ODR to OFF is shown below for the STM32U5 microcontroller using LSM6DSV320X platform-independent drivers:

```
static void disable_high_g (void)
{
 LSM6DSV320X_Object_t *pObj = MotionCompObj[AccInstance];
 lsm6dsv320x_hg_xl_data_rate_t hg_data_rate = LSM6DSV320X_HG_XL_ODR_OFF;
 (void)lsm6dsv320x_hg_xl_data_rate_set(&(pObj->Ctx), hg_data_rate, 1);
 HighGEnable = 0; //external flag
}
```

2.2.6 Finite state machine data types

In the following, the types that need to be defined in the fsm header file

```
#define MEMS_UCF_OP_READ 0
```

- Read the register at the location specified by the "address" field ("data" field is ignored)

```
#define MEMS_UCF_OP_WRITE 1
```

- Write the value specified by the "data" field at the location specified by the "address" field

```
#define MEMS_UCF_OP_DELAY 2
```

- Wait the number of milliseconds specified by the "data" field ("address" field is ignored)

```
#define MEMS_UCF_OP_POLL_SET 3
```

- Poll the register at the location specified by the "address" field until all the bits identified by the mask specified by the "data" field are set to 1

```
#define MEMS_UCF_OP_POLL_RESET 3
```

- poll the register at the location specified by the "address" field until all the bits identified by the mask specified by the "data" field are reset to 0

```
typedef struct {
 uint8_t op;
 uint8_t address;
 uint8_t data;
} ucf_line_ext_t;
```

- Type definition for reading, writing, delaying, polling (set / reset) registers of the FSM.
  - Op – one of 5 FSM operations defined earlier in this section.
  - Address – register address within the FSM
  - Data – data to write

```
static const ucf_line_ext_t highglowg_fsm[]
```

- FSM configuration to be loaded onto the high-g sensor during initialization.

2.2.7 Finite state machine handler

The following functions have to be implemented by the user to handle communication with the FSM specifically for each target platform using platform independent drivers for the selected sensor:

```
static void FSM_Init(void)
```

- Loads the FSM configuration to the high-g sensor, the FSM configuration for LSM6DSV320X and ISM6HG256X is available in the application example in the X-CUBE-MEMS1 package
- Call before the main while loop in user application
- An example is shown below for STM32U5 microcontroller using LSM6DSV320X platform independent drivers:

```
void FSM_Init(void)
{
  int i;
  int length = 0;
  length = sizeof(highglowg_fsm)/sizeof(ucf_line_ext_t);

  for (i = 0; i < length; i++)
  {
  (void)BSP_MOTION_SENSOR_Write_Register(LSM6DSV320X_0, highglowg_fsm[i].address,  highglowg_
fsm[i].data);
  }
}
```

```
static void FSM_Handler(void)
```

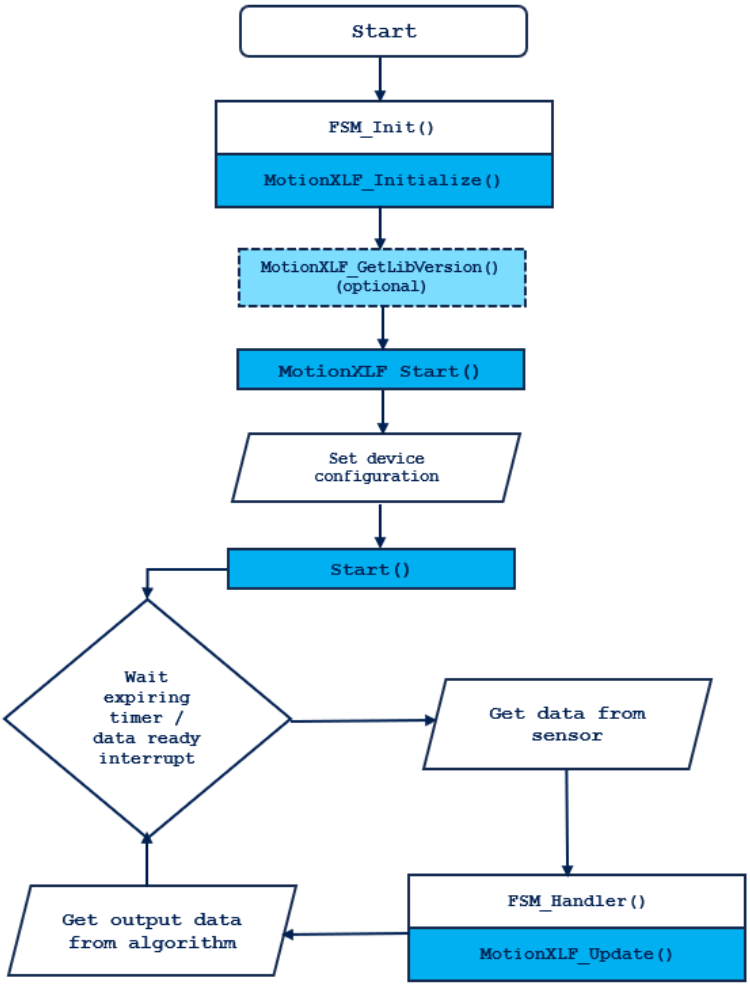
- Read the values in FSM_OUTS1 and FSM_OUTS2 registers

- Call during the main while loop in user application
- An example is shown below for STM32U5 microcontroller using LSM6DSV320X platform independent drivers:

```
static void FSM_Handler()
{
  (void)BSP_MOTION_SENSOR_Write_Register(LSM6DSV320X_0, LSM6DSV320X_FUNC_CFG_ACCESS, LSM6DSV320X_EMBED_FUNC_MEM_BANK << 7);
  (void)BSP_MOTION_SENSOR_Read_Register(LSM6DSV320X_0, LSM6DSV320X_FSM_OUTS1, &FSM_OUT1);
  (void)BSP_MOTION_SENSOR_Read_Register(LSM6DSV320X_0, LSM6DSV320X_FSM_OUTS2, &FSM_OUT2);
  (void)BSP_MOTION_SENSOR_Write_Register(LSM6DSV320X_0, LSM6DSV320X_FUNC_CFG_ACCESS, LSM6DSV320X_MAIN_MEM_BANK << 7);
}
```

2.2.8 API flow chart

Figure 2. MotionXLF API logic sequence



2.2.9 Demo code

```
/** includes **/
...
#include "motion_xlf.h"
...

/**defines, static variables, function defines**/
...
float_t low_g_accel[3];
float_t high_g_accel[3];
uint_8 fsm_out[2];
XLF_in_t input_struct;
XLF_out_t output_struct;
XLF_return_t return_value;
static void FSM_Init(void);
static void enable_high_g (void);
static void disable_high_g (void);
static void Time_Handler();
static void Accelero_Sensor_Handler(float_t* low_g_accel[3], float_t* high_g_accel[3]);
static void FSM_Handler(uint8_t* fsm_out[2]);
...

int main(void){
/**device configuration initialization functions**/

XLF_mcu_type_t mcu = MOTION_XLF_MCU_STM32;

/* Library API initialization function */
MotionXLF_Initialize(mcu);

/* Optional: Get version */
MotionXLF_GetLibVersion(lib_version);

....
HAL_Init();
SystemClock_Config();
GPIO_Init();
Sensor_Init();
....

FSM_Init();
enable_high_g();
return_value = MotionXLF_Start();

while(1){
  Time_Handler();
  FSM_Handler(&fsm_out);
  Accelero_Sensor_Handler(&low_g_accel, &high_g_accel);
  input_struct.low_g_data_mg.x = low_g_accel[0];
  input_struct.low_g_data_mg.y = low_g_accel[1];
  input_struct.low_g_data_mg.z = low_g_accel[2];
  input_struct.high_g_data_mg.x = high_g_accel[0];
  input_struct.high_g_data_mg.y = high_g_accel[1];
  input_struct.high_g_data_mg.z = high_g_accel[2];
  input_struct.FSM_OUT1 = fsm_out[0];
  input_struct.FSM_OUT2 = fsm_out[1];

  return_value = MotionXLF_Update(&input_struct,&output_struct, enable_high_g,
disable_high_g);

  printf("%f,%f,%f", output_struct.fused_imu.x,
      output_struct.fused_imu.y,
      output_struct.fused_imu.z;
  }
```

2.2.10 Algorithm performance

Table 2. Elapsed time (µs) algorithm

Description	Min.	Average	Max.
Cortex-M4 STM32F401RE at 84 MHz	1	1.88	3
Cortex-M3 STM32L152RE at 32 MHz	<1	<1	1
Cortex-M33 STM32U575ZI-Q at 160 MHz	<1	<1	1
Cortex-M0+ STM32L073RZ at 32 MHz	<1	<1	1

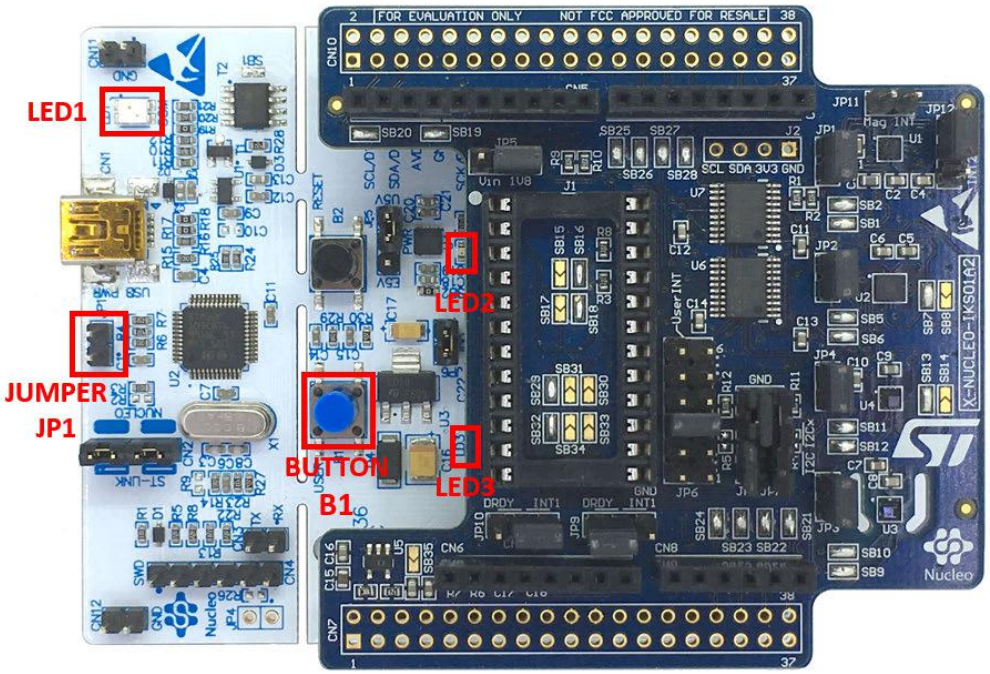
2.3 Sample application

The MotionXLF middleware can be easily manipulated to build user applications. A sample application is provided in the Application folder.

It is designed to run on X-NUCLEO-IKS02A1, X-NUCLEO-IKS5A1 and X-NUCLEO-IKS4A1 expansion board on a NUCLEO-F401RE, NUCLEO-L073RZ, NUCLEO-L152RE or NUCLEO-U575ZI-Q development board, with a high-g sensor DIL24 board mounted on the DIL24 socket.

The application fuses data from the low-g sensor and high-g sensor in real-time, showing the fused output, all while handling noise inconsistency, time-evolving offset, and discontinuities. The application also handles turning on or off the high-g sensor at the correct time to save power. The sample application uses the MEMS Studio application, which can be downloaded from www.st.com.

Figure 3. STM32 Nucleo: LEDs, button, jumper



The above figure shows the user button B1 and the three LEDs of the NUCLEO-F401RE board. Once the board is powered, LED LD3 (PWR) turns ON.

Note: After powering the board, LED LD2 blinks once indicating the application is ready.

2.3.1 MEMS Studio application

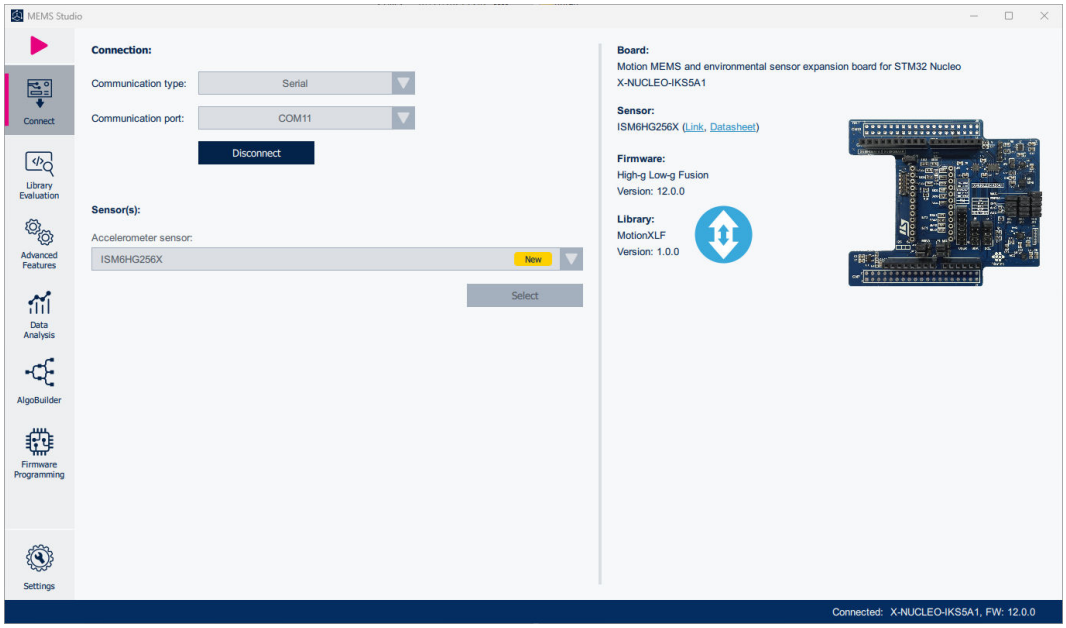
The sample application uses the MEMS Studio application, which can be downloaded from www.st.com.

Step 1. Ensure that the necessary drivers are installed and the STM32 Nucleo board with appropriate expansion board is connected to the PC.

Step 2. Launch the MEMS Studio application to open the main application window.

If an STM32 Nucleo board with supported firmware is connected to the PC, it is automatically detected. Press the [**Connect**] button to establish connection to the evaluation board.

Figure 4. MEMS-Studio - Connect

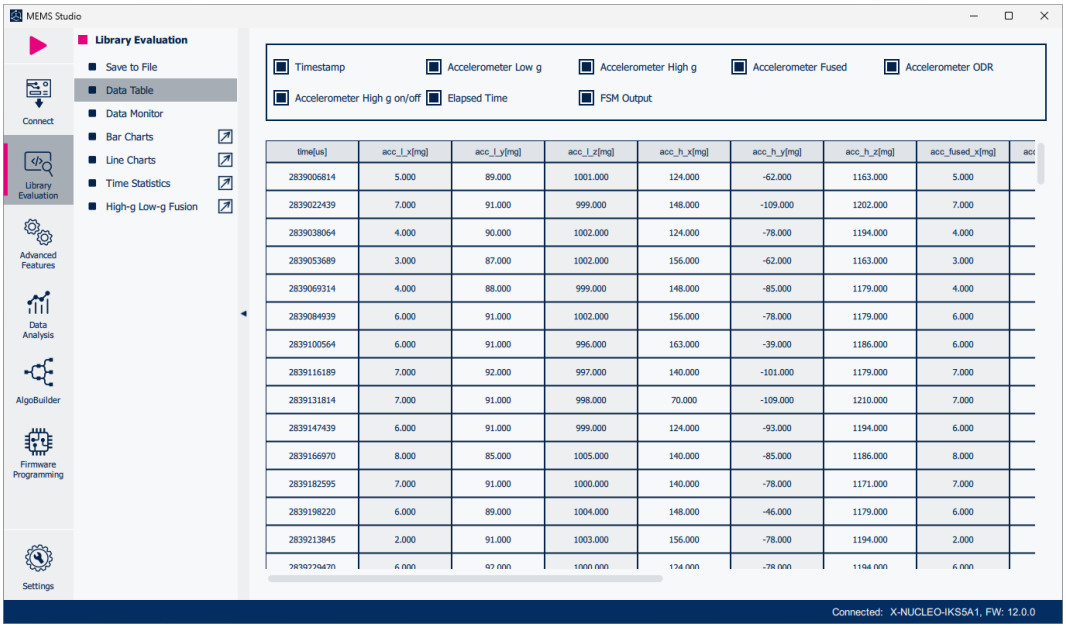


Step 3. When connected to a STM32 Nucleo board with supported firmware [**Library Evaluation**] tab is opened.

To start and stop data streaming, toggle the appropriate [**Start**] or [**Stop**] button on the outer vertical tool bar.

The data coming from the connected sensor can be viewed selecting the [**Data Table**] tab on the inner vertical tool bar.

Figure 5. MEMS-Studio - Library Evaluation - Data Table

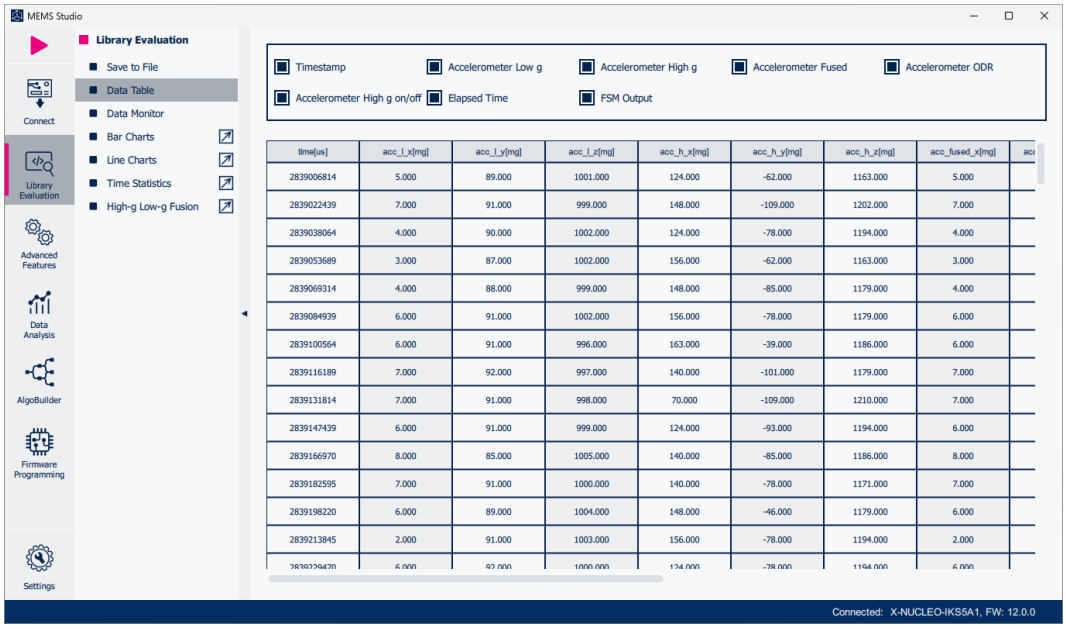


Step 4. Click on the **High-g Low-g fusion** icon in the vertical toolbar to open the dedicated application window. This window displays the low-g, high-g and fused value of the accelerometer, FSM output, and indicates whether the Accelerometer high-g is active or not. You can adjust [**Sensor ODR**], [**Report Rate**] which sets how fast the firmware sends data to the application, and enable or disable [**Continuous Tracking**].

Figure 6. MEMS-Studio - Library Evaluation - High-g Low-g Fusion

Step 5. Click on the **Save to file** icon in the vertical toolbar to open the datalog configuration window. You can select the data to be saved in the files. You can start or stop saving by clicking on the corresponding button.

Figure 7. MEMS-Studio - Library Evaluation - Save To File



2.4 References

All of the following resources are freely available on www.st.com.

1. UM1859: Getting started with the X-CUBE-MEMS1 motion MEMS and environmental sensor software expansion for STM32Cube
2. UM1724: STM32 Nucleo-64 board
3. UM3233: Getting started with MEMS Studio

Revision history

Table 3. **Document revision history**

Date	Revision	Changes
18-Apr-2025	1	Initial release.
16-Sep-2025	2	Updated Section 2.2.1: MotionXLF library description, Section 2.2.2: MotionXLF library operation, Section 2.2.3: MotionXLF library parameters, Section 2.2.4: MotionXLF APIs, Section 2.3.1: MEMS Studio application.

Contents

List of tables

List of figures

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice.

In the event of any conflict between the provisions of this document and the provisions of any contractual arrangement in force between the purchasers and ST, the provisions of such contractual arrangement shall prevail.

The purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgment.

The purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of the purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

If the purchasers identify an ST product that meets their functional and performance requirements but that is not designated for the purchasers' market segment, the purchasers shall contact ST for more information.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.