Getting started with InfraredPD presence and motion detection library in X-CUBE-MEMS1 expansion for STM32Cube

Introduction

The InfraredPD is a middleware library component of the X-CUBE-MEMS1 software and runs on STM32. It provides real-time information about the presence and the motion of a person in the field of view of the sensor. It also provides compensation for the object temperature data with respect to ambient temperature data changes.

This library is intended to work with the STHS34PF80 sensor only.

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.

The software comes with sample implementation running on X-NUCLEO-IKS01A3 and X-NUCLEO-IKS02A1 expansion board on a NUCLEO-F401RE, NUCLEO-L073RZ, NUCLEO-L152RE or NUCLEO-U575ZI-Q development board.

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 InfraredPD middleware library in X-CUBE-MEMS1 software expansion for STM32Cube

2.1 InfraredPD overview

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

The library acquires data from the infrared sensor and provides real-time information about the presence and the motion of a person in the field of view of the sensor.

This library is intended to work with the STHS34PF80 sensor only. Functionality and performance when using other sensors are not analyzed and can be significantly different from what described in the document.

A sample implementation is available on X-NUCLEO-IKS01A3 and X-NUCLEO-IKS02A1 expansion board on a NUCLEO-F401RE, NUCLEO-L073RZ, NUCLEO-L152RE or NUCLEO-U575ZI-Q development board.

2.2 InfraredPD library

Technical information fully describing the functions and parameters of the InfraredPD APIs can be found in the InfraredPD_Package.chm compiled HTML file located in the documentation folder.

2.2.1 InfraredPD library description

The InfraredPD sensor fusion library manages data acquired from the infrared sensor; it features:

- the compensation of ambient temperature changes in the object temperature
- the motion detection, and the presence detection
- recommended sensor data sampling frequency of 1 Hz to 30 Hz
- resources requirements:
- Cortex-M0+: 3.47 kB of code and up to 4 kB of data memory
- Cortex-M3: 3.28 kB of code and up to 4 kB of data memory
- Cortex-M33: 3.2 kB of code and up to 4 kB of data memory
- Cortex-M4: 3.2 kB of code and up to 4 kB of data memory
- Cortex-M7: 3.1 kB of code and up to 4 kB of data memory

Note: Size of dynamically allocated data memory is dependent on algorithm setup e.g.: type of compensation algorithm used (or not used at all), number of averaged samples in filter, ODR.

- available for ARM Cortex-M0+, Cortex-M3, Cortex-M33, Cortex-M4 and Cortex-M7 architecture

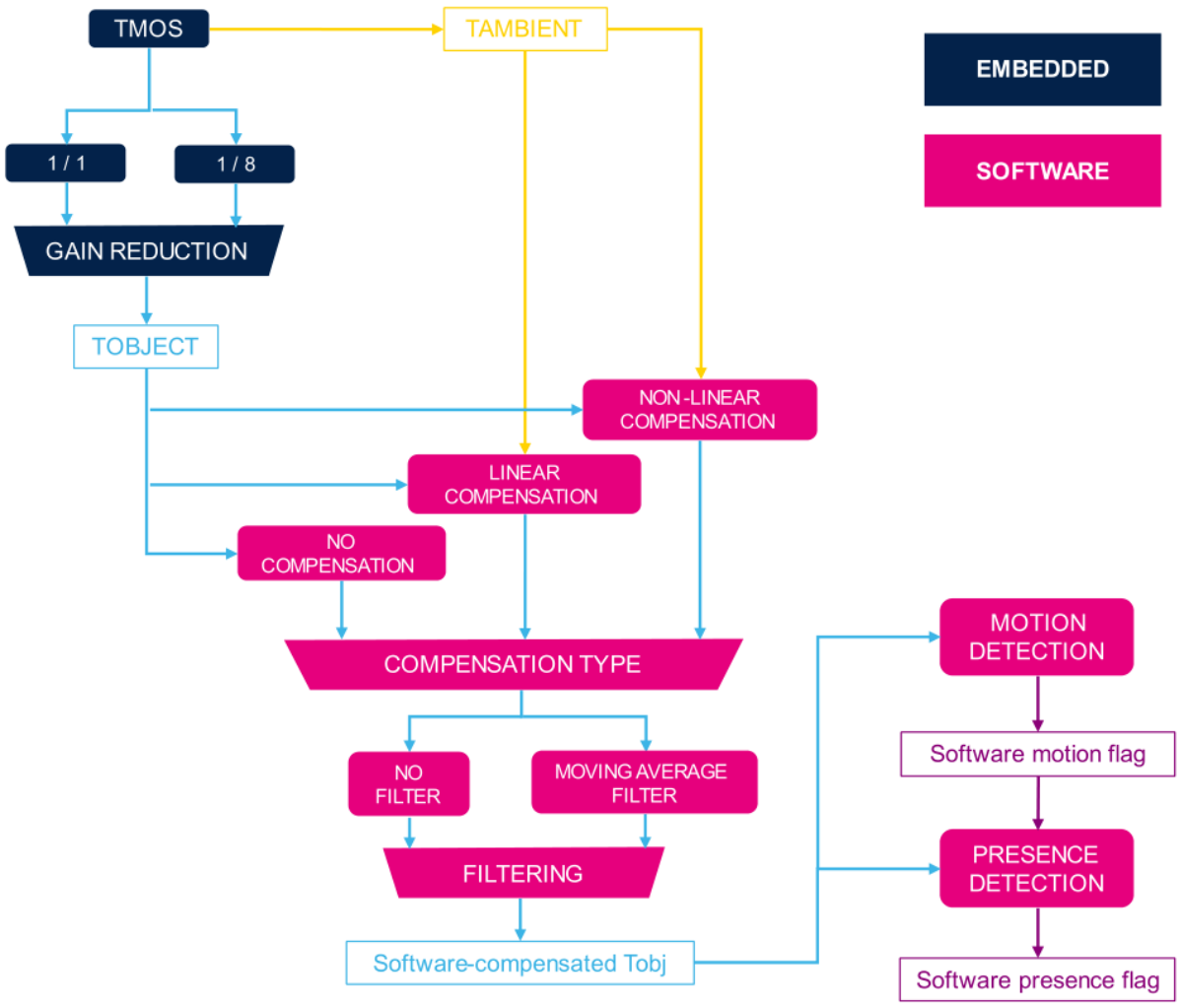
2.2.2 InfraredPD library operation

The InfraredPD library is comprised of three parts: the ambient compensation of object temperature, the motion detection, and the presence detection. They are implemented through three distinct algorithms run in succession at each library iteration.

The software library allows performing compensation and motion and presence detection even if the reduced gain mode (also called “wide mode”) of the sensor is enabled and so the embedded algorithms cannot be used.

The library is designed for the STHS34PF80 (also known as TMOS) sensor only. Functionality and performances when using other sensors are not analyzed and can be significantly different from what described in the document.

Figure 1. **Block diagram of InfraredPD library**



Compensation of ambient temperature changes in the object temperature

If during a measurement carried out with the STHS34PF80 sensor the ambient temperature changes significantly, the object temperature output might be affected. For this reason, a compensation algorithm based on ambient temperature (Tambient) data can be performed to improve the accuracy of the object temperature (Tobject) data.

Compensation of ambient temperature changes in the object temperature is effective only for compensating temperature changes that occur in the environment in thermal coupling with the sensor (for example, changes in the temperature of the PCB on which the sensor is mounted).

The library implements algorithms that acquire the T_{OBJECT} data and $T_{AMBIENT}$ data outputs from the STHS34PF80 sensor and provide a real-time compensated Tobject data with respect to changes in the Tambient data output that have occurred since the initialization/reinitialization of the algorithm (*t_obj_comp*).

The library implements two types of algorithms for compensation, and the user can choose which one to run (through the *comp_type* variable of the *IPD_algo_conf_t* struct):

- a non-linear algorithm (default setting), which is based on the Stefan-Boltzmann law;
- a linear algorithm, which replicates the embedded algorithm and is a linear approximation of the Stefan-Boltzmann law.

If the linear algorithm is chosen, the library can effectively replace the compensation algorithm embedded in the device in the case that the gain reduction of the sensor is enabled and therefore the embedded algorithm cannot be used. Please visit refer to AN5867 for more information on the embedded algorithm for the compensation of ambient temperature changes in the object temperature.

In the case of slow Tambient variations, the Tobject output can be effectively compensated with the linear algorithm. The formula of this algorithm is the following:

T_{OBJ_COMP}[LSB] = T_{OBJECT}[LSB] + SENSITIVITY[LSB/°C] x ($T_{AMBIENT}$ - $T_{AMBIENT\_ZERO}$) [°C] = T_{OBJECT}[LSB] + (SENSITIVITY/100)[-] x ($T_{AMBIENT}$ - $T_{AMBIENT\_ZERO}$) [LSB] ,

where T_{OBJ_COMP} is the output of the algorithm (the compensated value of object temperature), T_{OBJECT} is the current value of "raw" object temperature, SENSITIVITY is the sensitivity for object temperature data, $T_{AMBIENT}$ is the current value ambient temperature of the sensor, and $T_{AMBIENT\_ZERO}$ is the ambient temperature data in the first iteration of the algorithm after initialization / reinitialization.

In the case of fast Tambient variations, it is preferable to choose the non-linear algorithm, which strictly follows the physical law. Both algorithms work with respect to the Tambient data of the first iteration of the algorithm after initialization / reinitialization. Because of this, in the case that a thermal transient is detected at device power-on from the application it is preferable to wait for the initialization of the algorithm or to reinitialize it after the transient has concluded. The chosen compensation algorithm can be reinitialized at any time through the InfraredPD_ResetComp() API.

Note that the nonlinear algorithm, too, cannot compensate temperature changes that occur in an environment not in thermal coupling with the sensor (for example, effects caused by heaters or coolers distant from the sensor).

In case an optical system is in use with the TMOS sensor, the actual sensitivity of the sensor will be lower than the default value. In this case, it is highly recommended to acquire info on the value of the transmittance of the optical system in the range of wavelength between 5 um and 20 um. This value should be passed to the library through the transmittance variable of the *IPD_algo_conf_t* struct in order for the compensation algorithm to work correctly. Note that the *sens_data* value of the *IPD_device_conf_t* struct should contain the default value of sensitivity for object temperature data (refer to AN5867).

The library also gives the possibility to filter the compensated Tobject data. This is useful as compensation introduces the noise of the Tambient output data in the compensated Tobject data. By default, the library filters with a moving average filter the compensated Tobject data so that it has the same signal-to-noise ratio of the "raw" Tobject data, introducing a latency that depends on the ODR and on the number of averaged samples for object temperature (AVG_TMOS) of the sensor. From this it follows that, if compensation is disabled, filtering is disabled. The moving average filter is enabled by default, but it can be disabled through the *comp_filter_flag* variable of the *IPD_algo_conf_t* struct.

In any case, it is highly recommended to keep the number of averaged samples for ambient temperature (AVG_T) at 8 (default value) when using the library, in order to not introduce too much noise in the compensated Tobject output.

Presence detection and motion detection are based on the compensated Tobject signal. If the user chooses to disable the compensation algorithm, the algorithms for presence and motion detection will be performed on the "raw" Tobject signal; in this case, their performances might be affected.

Motion detection

The algorithm for motion detection can provide an accurate output flag (*mot_flag*) which reports if a person is moving in the field of view of the sensor or not.

The algorithm run by the library works similarly to the algorithm for motion detection embedded in the device and replaces it in the case that the gain reduction of the sensor is enabled and therefore the embedded algorithm cannot be used. Please refer to AN5867 for more information on the embedded smart digital algorithm for motion detection.

The library gives the possibility of tuning the algorithm for motion detection by setting a specific threshold for detection (*mot_ths*). The smaller the threshold, the farther the distance of detection. However, by decreasing the threshold, the algorithm will be more sensitive to environmental noise. The default value is 150 LSB, which is fine-tuned for a detection range of 2-3 meters (without the use of lenses). An intermediate output signal which reports the level of change of the compensated Tobject output (*t_obj_change*) can be used to tune the threshold, just like the TMOTION signal of the embedded smart digital algorithm; the goal of the tuning procedure should be to make the algorithm sensitive to changes caused by actual movements of a person in the field of view of the sensor but not to changes caused by other sources of noise.

Presence detection

The algorithm for presence detection is the core of the library. It can provide an accurate and stable output flag (*pres_flag*) which reports if a person is present in the field of view of the sensor or not.

The algorithm run by the library improves upon the algorithm for presence detection embedded in the device and replaces it in the case that the gain reduction of the sensor is enabled and therefore the embedded algorithm cannot be used. Please refer to AN5867 for more information on the embedded smart digital algorithm for presence detection.

The library gives the possibility of tuning the algorithm for presence detection by setting a specific threshold for detection (*pres_ths*). The smaller the threshold, the farther the distance of detection. However, by decreasing the threshold, the algorithm will be more sensitive to environmental noise. The default value is 250 LSB, which is fine-tuned for a detection range of 2-3 meters (without the use of lenses). An intermediate output signal which reports the level of change of the compensated Tobject output (*t_obj_change*) can be used to tune the threshold, just like the TPRESENCE signal of the embedded smart digital algorithm; the goal of the tuning procedure should be to make the algorithm sensitive to changes caused by an actual entrance (or exit) of a person in (or from) the field of view of the sensor but not to changes caused by other sources of noise.

The trigger from absence state to presence state has no additional latency (except for the latency that is due to the compensation algorithm) thanks to the use of the output of the motion detection algorithm, while the trigger from presence state to absence state has an additional latency of a time interval up to 10 seconds in order to correctly evaluate the end of the presence condition.

When initializing the algorithms, the field of view of the sensor must be empty for the presence detection algorithm to work, as it assumes the absence state at start-up. When resetting the compensation algorithm, the state of the presence detection algorithm is always maintained (unlike the embedded algorithm, which loses the presence state). However, if the algorithm is in presence state when the reset is triggered, the presence condition must not change for at least 10 seconds: if the presence condition ends earlier than this interval, the algorithm might remain stuck in the presence state. Note that in any case *t_obj_change* and the motion detection flag are reset to 0 when reinitializing the compensation algorithm.

The software algorithm greatly improves upon the embedded algorithm particularly for the cases in which drifts in the Tobject signal which could not be rejected with the compensation algorithm are observed. Such drifts are usually related to temperature changes of the environment in the field of view of the sensor only, and not to temperature changes of the environment in thermal contact with the sensor. These drifts can cause errors in the results of the embedded algorithm, in particular:

- if drifts tend to decrease the Tobject signal (due to cooling events in the field of view), the presence flag switches from presence state to absence state even if a person is still inside the field of view;
- if drifts tend to increase the Tobject signal (due to heating events in the field of view), the presence flag never switches from presence state to absence state and remains indefinitely stuck in presence state, even when the field of view is left empty.

The algorithm of the library is based on a different logic than the embedded one, and it is aimed at dealing with these conditions.

If the issue of the presence flag stuck in presence state happens frequently even with the use of the software library algorithm, a logic of reset of the presence flag to the absence state can be enabled through the *abs_static_flag* variable of the *IPD_algo_conf_t* struct (it is disabled by default). This logic is based on the evaluation of the motion flag: continuous standstill for more than 30 seconds triggers the change of the presence flag from the presence state to the absence state; if a sudden movement is detected before 10 minutes from the trigger of the absence state in said conditions, the presence flag is restored to the presence state (otherwise, if no movement is detected for 10 continuous minutes, the presence state can no longer be restored).

2.2.3 InfraredPD library parameters

In the following, the types that are defined in the header file of the library.

```
typedef void *IPD_Instance_t;
```

- pointer to the library instance loaded in data memory

```
typedef enum
{
  IPD_MCU_STM32 = 0,
  IPD_MCU_BLUE_NRG1,
  IPD_MCU_BLUE_NRG2,
  IPD_MCU_BLUE_NRG_LP,
} IPD_mcu_type_t;
```

- used MCU type

```
typedef enum
{
  IPD_INIT_OK = 0,        /* No error */
  IPD_INIT_ERR_ODR,       /* Wrong ODR value */
  IPD_INIT_ERR_AVGTMOS,   /* Wrong number of average for Tobj */
  IPD_INIT_ERR_AVGT,      /* Wrong number of average for Tamb */
  IPD_INIT_ERR_GAIN,      /* Wrong gain reduction factor value */
  IPD_INIT_ERR_TRANS,     /* Wrong transmittance value */
  IPD_INIT_ERR_RES1,      /* Reserved error code */
  IPD_INIT_ERR_RES2       /* Reserved error code */
} IPD_init_err_t;
```

- library status – error code returned by InfraredPD_Start API function

```
typedef enum
{
  IPD_COMP_NONE,    /* No compensation */
  IPD_COMP_LIN,     /* Linear compensation */
  IPD_COMP_NONLIN   /* Non-linear compensation */
} IPD_comp_t;
```

- type of algorithm for compensation of ambient temperature changes in the object temperature

```
typedef struct
{
  uint8_t  odr;
  uint16_t avg_tmos;
  uint8_t  avg_t;
  uint8_t  gain_factor;
  uint16_t sens_data;
  float    transmittance;
} IPD_device_conf_t
```

- the parameters of the device, that must be configured and/or retrieved in the application code and passed to the algorithm during initialization:
  - *odr* – ODR in Hz, possible values are from 1 Hz to 30 Hz
  - *avg_tmos* – number of averaged samples for object temperature, up to 1024
  - *avg_t* – number of averaged samples for ambient temperature, it is strongly recommended to be kept at 8
  - *gain_factor* – gain reduction factor, either 1 or 8
  - *sens_data* – default sensitivity of the unit
  - *transmittance* – transmittance of the optical system in the range of wavelengths between 5 µm and 20 µm, ranging from 0 [0%] to 1 [100%]

```
typedef struct
{
  IPD_comp_t comp_type;
  uint8_t   comp_filter_flag;
  uint16_t  mot_ths;
  uint16_t  pres_ths;
  uint8_t   abs_static_flag;
} IPD_algo_conf_t;
```

- the parameters of the algorithm that can be configured from the application code:
  - *comp_type* – flag to choose the type of compensation algorithm
  - *comp_filter_flag* – flag to enable or disable moving average filter after compensation (default: enabled)
  - *mot_ths* – threshold for motion detection [LSB] (default: 150 LSB)
  - *pres_ths* – threshold for presence detection [LSB] (default: 250 LSB)
  - *abs_static_flag* – flag to enable or disable absence state trigger in static conditions (default: disabled)

```
typedef struct
{
  int16_t t_amb;
  int16_t t_obj;
} IPD_input_t;
```

- the inputs of the algorithms that must be provided to the algorithms at each iteration:
  - *t_amb* – ambient temperature data [LSB]
  - *t_obj* – raw object temperature data [LSB]

```
typedef struct
{
  int16_t t_obj_comp;    /* Compensated Tobject data [LSB] */
  int16_t t_obj_change;  /* Rate of change of compensated Tobject
                            data [LSB] */
  uint8_t mot_flag;      /* Motion detection flag [0: standstill, 1: movement] */
  uint8_t pres_flag;     /* Presence detection flag [0: absence, 1: presence] */
} IPD_output_t;
```

- the outputs of the algorithms that are provided by the algorithms at each iteration:
  - *t_obj_comp* – compensated object temperature data [LSB]
  - *t_obj_change* – level of change of Tobj_comp_sw, can be used to tune motion and presence detection algorithms [LSB]
  - *mot_flag* – motion detection flag [0: standstill, 1: movement]
  - *pres_flag* – presence detection flag [0: absence, 1: presence]

2.2.4 InfraredPD APIs

In the following, the API functions that are defined in the header file of the library.

```
uint8_t InfraredPD_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 InfraredPD_Initialize(IPD_mcu_type_t mcu_type)
```

- Perform InfraredPD library initialization and setup of the internal mechanism

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

- *mcu_type* – type of MCU used

```
IPD_Instance_t InfraredPD_CreateInstance(IPD_algo_conf_t *algo_conf)
```

- Create instance of InfraredPD library algorithm:
  - Allocate the memory for a library instance
  - Fill the structure pointed by algo_conf with the default values of the parameters for the configuration of the algorithms
  - Return a pointer to its memory location

Note: After calling this function, the algorithms can be initialized with the InfraredPD_Start() API function

- *algo_conf* – configuration of the algorithm
- Return pointer to new instance of the algorithm

```
void InfraredPD_DeleteInstance(IPD_Instance_t instance)
```

- Delete instance of InfraredPD library algorithm:
  - De-initialize the algorithm
  - Free the memory allocated for the instance
- *instance* – pointer to instance of the algorithm to be deleted

```
IPD_init_err_t InfraredPD_Start(IPD_Instance_t instance, IPD_device_conf_t
                                        *device_conf, IPD_algo_conf_t *algo_conf)
```

- Start the InfraredPD engine:
  - Initialize (or re-initialize) the algorithm of the instance following the parameters set in the two structures pointed by device_conf and algo_conf
  - Return an initialization error code (e.g.: if invalid device parameters were set)
- *instance* – pointer to instance of the algorithm to be started
- *device_conf* – configuration of the device
- *algo_conf* – configuration of the algorithm
- Return an initialization error code

```
void InfraredPD_Update(IPD_Instance_t instance, IPD_input_t *data_in, IPD_output_t *data_out)
```

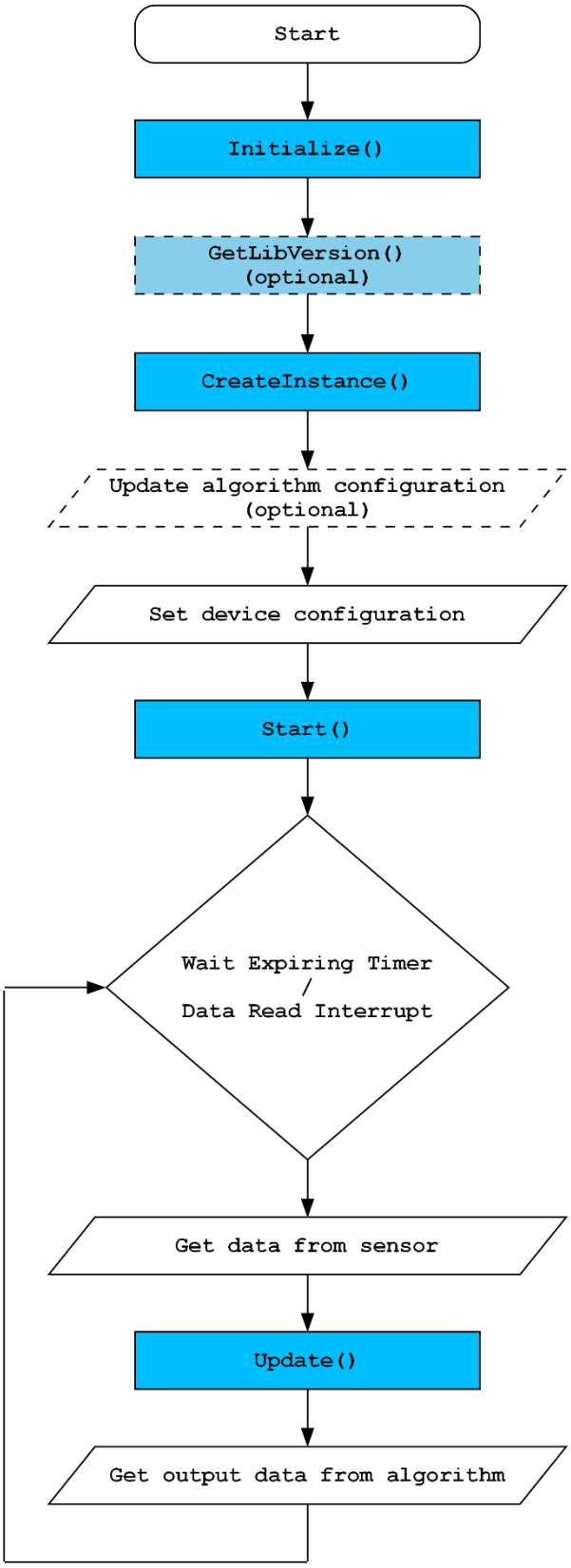
- Execute one step of the algorithm
- *instance* – pointer to instance of the algorithm
- *data_in* – input data
- *data_out* – output data

```
void InfraredPD_ResetComp(IPD_Instance_t instance)
```

- Reset the compensation algorithm
- *instance* – pointer to instance of the algorithm

2.2.5 API flow chart

Figure 2. **InfraredPD API logic sequence**



2.2.6 Demo code

```
#define IPD_STR_LENG  35

[...]

/*** Initialization ***/

char lib_version[IPD_STR_LENG];
IPD_mcu_type_t mcu = IPD_MCU_STM32;
IPD_Instance_t IPD_Instance;
IPD_algo_conf_t algo_conf;
IPD_device_conf_t device_conf;
IPD_init_err_t status;

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

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

/* Create library algorithm instance */
IPD_Instance = InfraredPD_CreateInstance(&algo_conf);

/* Optional: Modify algorithm and device configuration */
algo_conf.comp_type = TMOS_PRES_COMP_NONLIN;
[...]

/* Setup device configuration */
device_conf.odr = 30;
[...]

/* Start the algorithm engine */
status = InfraredPD_Start(instance, &device_conf, &algo_conf);

/*** Using Presence and Motion Detection algorithm ***/
Timer_OR_DataRate_Interrupt_Handler()
{
  IPD_input_t data_in;
  IPD_output_t data_out;

  /* Get data from sensor */
  ReadSensor(&data_in.t_amb, &data_in.t_obj);

  /* Execute one step of the algorithms */
  InfraredPD_Update(instance, &data_in, &data_out)

  /* Get output data from algorithm */
  int16_t ObjectTempComp = data_out.t_obj_comp;
  int16_t ObjectTempCompChange = data_out.t_obj_change;
  uint8_t MotionDetected = data_out.mot_flag;
  uint8_t PresenceDetected = data_out.pres_flag;
}
```

2.2.7 Algorithm performance

Table 2. Elapsed time (µs) algorithm

	Min.	Average	Max.
Cortex-M4 STM32F401RE at 84 MHz	4	5	36
Cortex-M3 STM32L152RE at 32 MHz	70	75	636
Cortex-M33 STM32 U575ZI-Q at 160 MHz	2	2	17

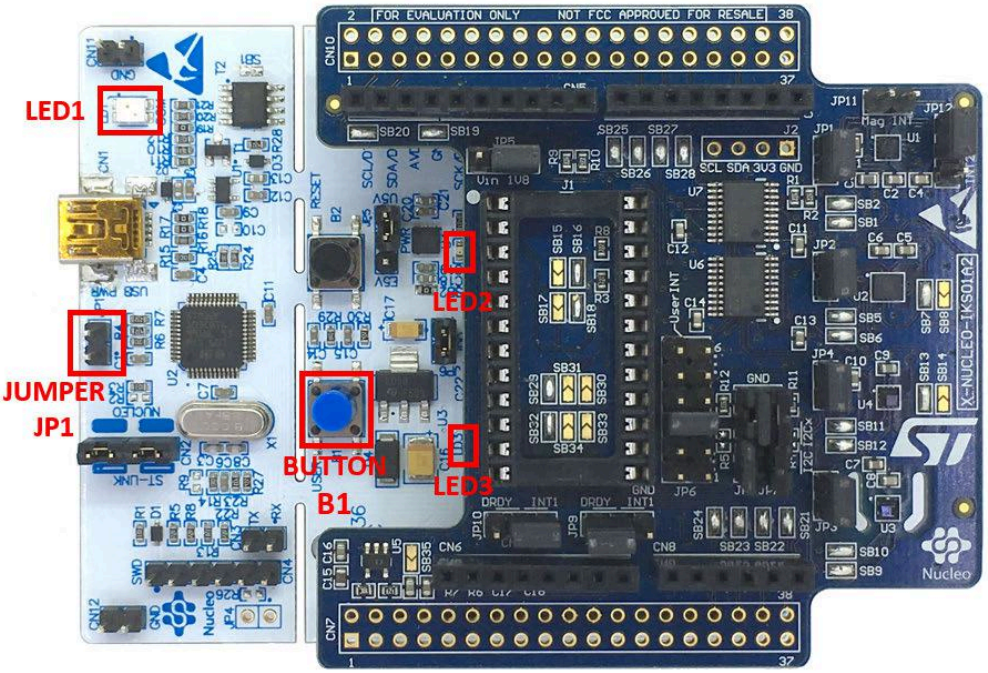
2.3 Sample application

The InfraredPD 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-IKS01A3 and X-NUCLEO-IKS02A1 expansion board on a NUCLEO-F401RE, NUCLEO-L073RZ, NUCLEO-L152RE or NUCLEO-U575ZI-Q development board.

The application provides real-time information about the presence and the motion of a person in the field of view of the sensor. It also provides compensation for the object temperature data with respect to ambient temperature data changes.

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.

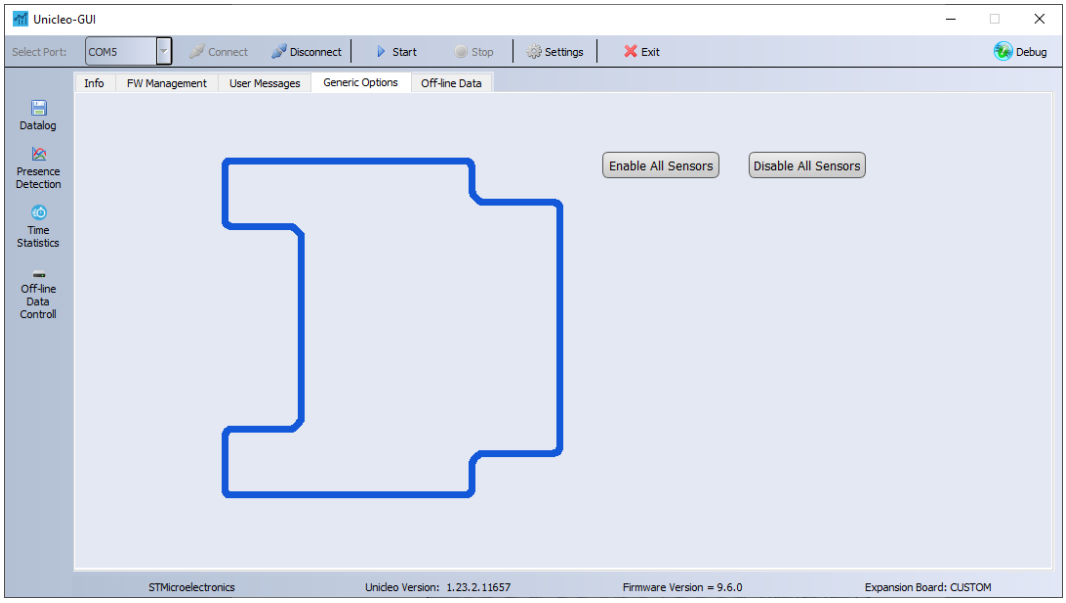
2.3.1 Unicleo-GUI application

The sample application uses the Windows Unicleo-GUI utility, 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.

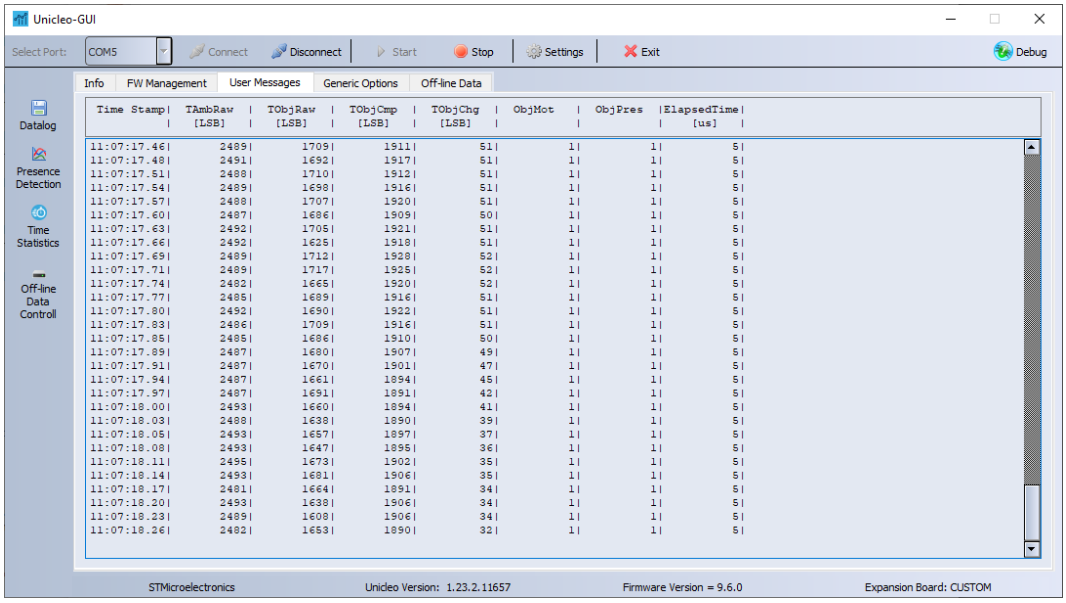
Step 2. Launch the Unicleo-GUI application to open the main application window. If an STM32 Nucleo board with supported firmware is connected to the PC, it is automatically detected and the appropriate COM port is opened.

Figure 4. Unicleo main window



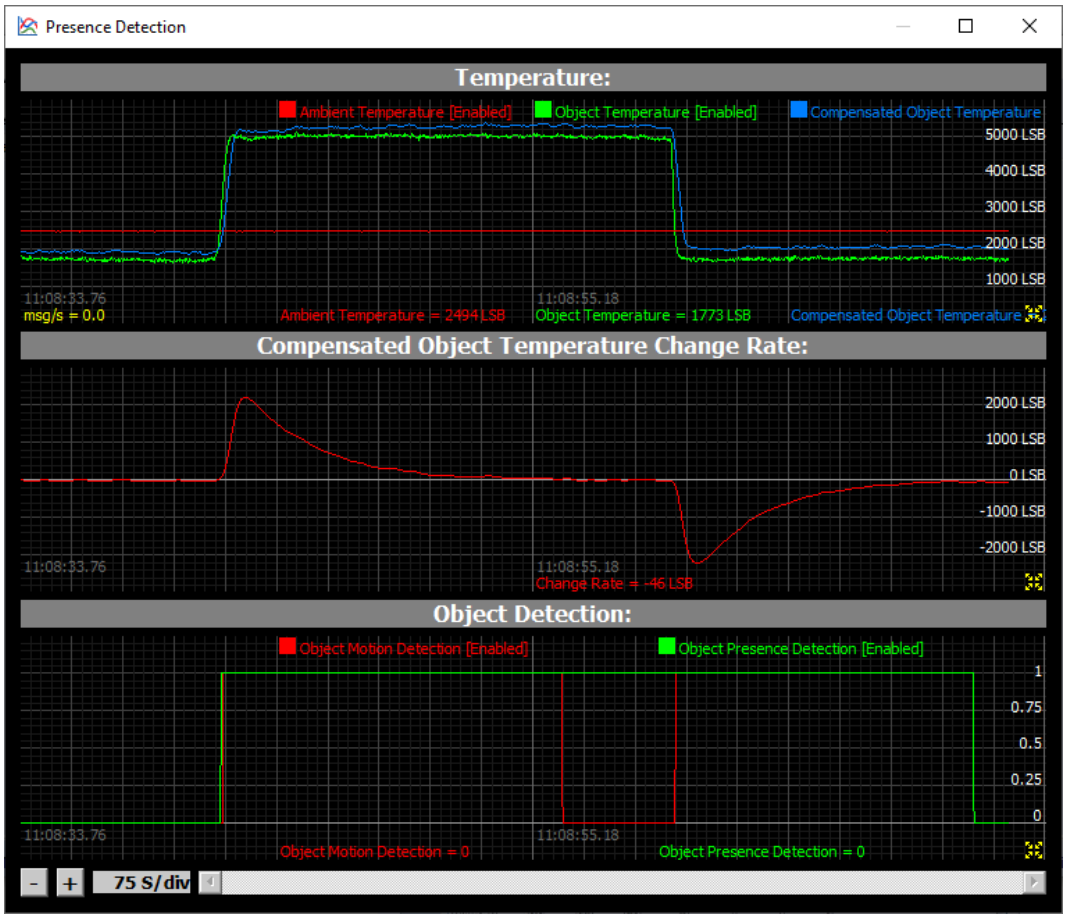
Step 3. Start and stop data streaming by using the appropriate buttons on the horizontal tool bar. The data coming from the connected sensor can be viewed in the User Messages tab.

Figure 5. User Messages tab



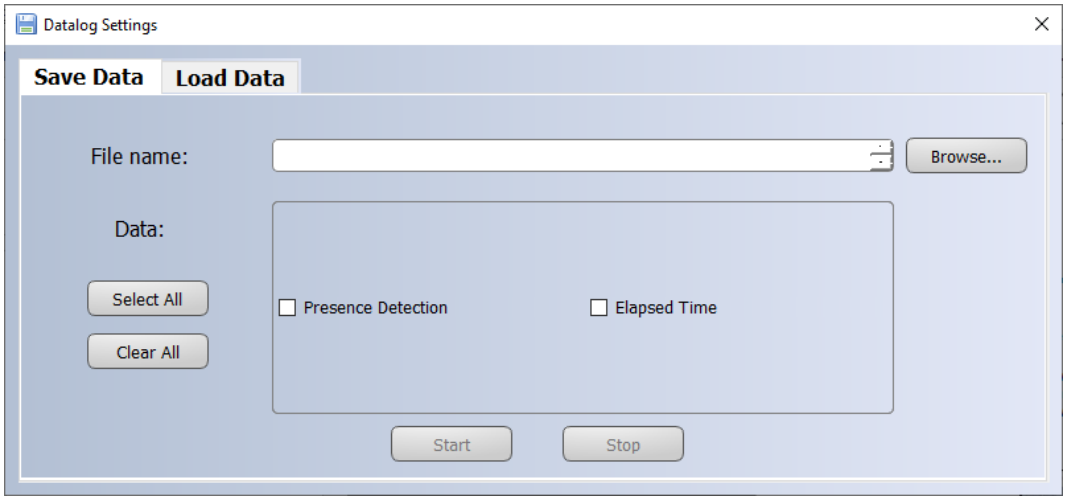
Step 4. Click on the **Presence Detection**icon in the vertical toolbar to open the dedicated application window.

Figure 6. Presence Detection window



Step 5. Click on the **Datalog** icon in the vertical toolbar to open the datalog configuration window: you can select the presence and the motion data to be saved in the files. You can start or stop saving by clicking on the corresponding button.

Figure 7. Datalog window



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. UM2128: Getting started with Unicleo-GUI for motion MEMS and environmental sensor software expansion for STM32Cube

Revision history

Table 3. **Document revision history**

Date	Revision	Changes
21-Jun-2023	1	Initial release.

Contents

List of tables

List of figures