iCOMOX Monitor
User Manual SDK

Table of Contents

1 Monitor

To modify the iCOMOX Monitor Python project, perform the following steps:

1. Download and install the latest python 32-bit version from https://www.python.org/
2. Download and install PyCharm IDE from https://www.jetbrains.com/pycharm/
1. Launch PyCharm.
2. In the Top Menu, click File / New Project….

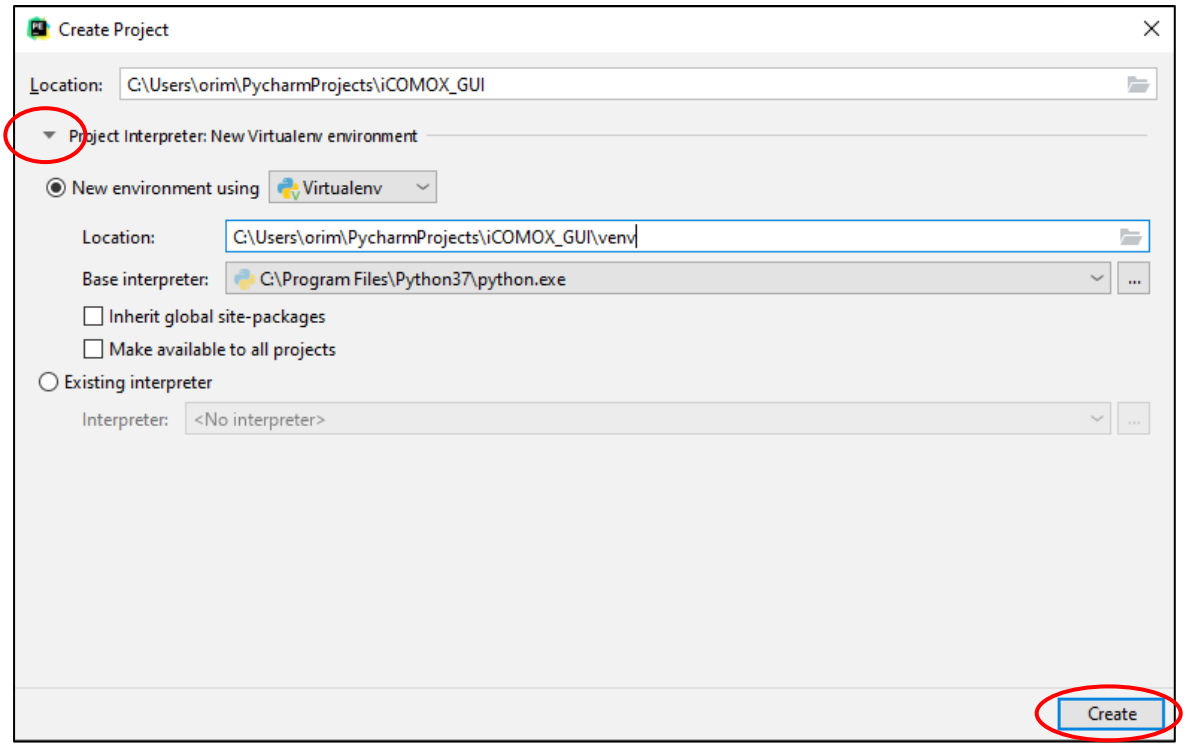

Figure 3: PyCharm create project menu.

3. Open the Project Interpreter menu, and select the "New environment using" radio button.
4. From the "New environment using" combo box, select "Virtualenv".
5. The location field path should be of the form "…/iCOMOX_GUI/venv".
6. The Base interpreter path should point to the python.exe file in the 32-bit python installation folder.
7. Click "Create".
8. Navigate to the settings menu through "File->Settings", select the "Project:iCOMOX_GUI" tab, and click on "Project Interpreter".

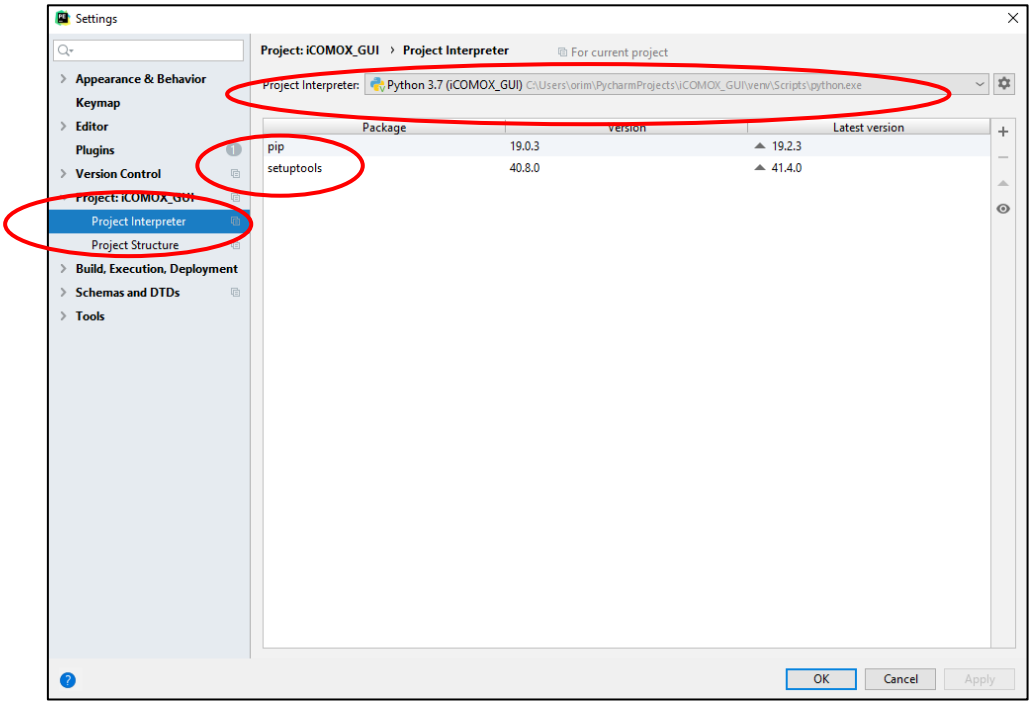

Figure 4: Project Interpreter tab.

9. Project Interpreter path should be of the form: "Python 3.8 (iCOMOX_GUI) {BaseFolder}\iCOMOX_GUI\venv\Scripts\python.exe".

Note: The following paragraphs (10-15) explain how to manually load Python libraries into the project's virtual environment. The project contains the requirements.txt file which allows the PyCharm IDE to automatically download and install the required libraries.

10. The package list should contain "pip" and "setuptools" packages. Double click each of them to launch the "Available Packages" menu.

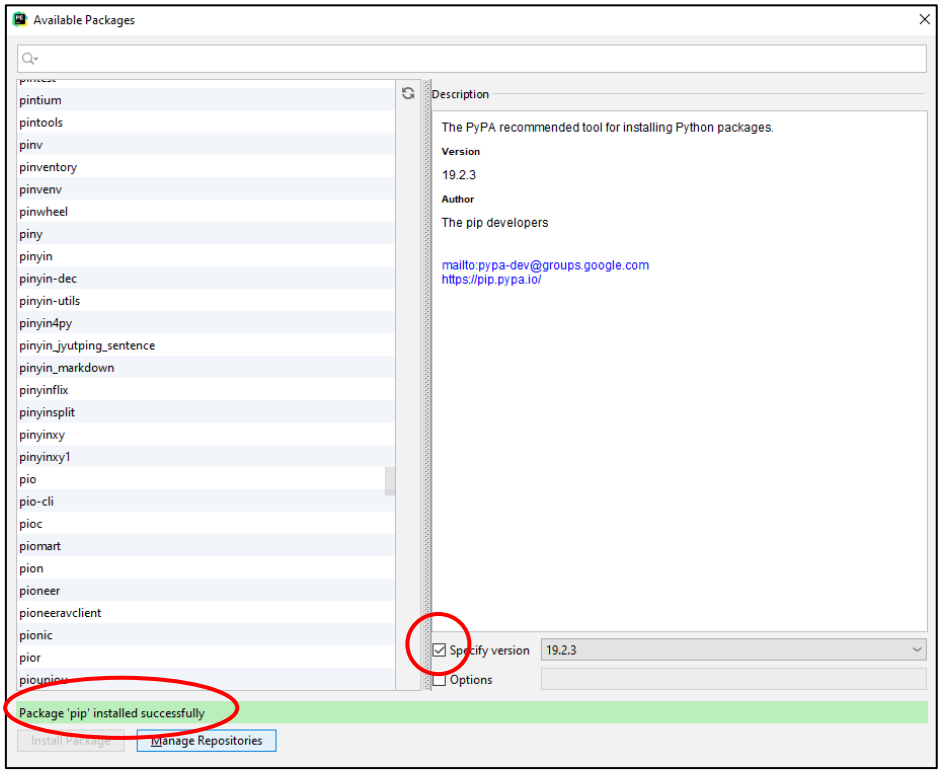

Figure 5: Available Packages menu.

11. Check the "Specify Version" check-box, then click on "Install Package" to update the package version.
12. A green indicator appears, indicating that the package was successfully installed.
13. Through the "Available Packages" menu, install the following packages (names are case sensitive): Pillow, tk-tools, pyserial, numpy, matplotlib, scipy, openpyxl, pywin32, iotc, requests, ifaddr.
14. Close the "Available Packages" menu, then click "Apply" and "Ok".
15. Download "iCOMOX_Monitor_vx.x.zip" from http://www.shiratech-solutions.com/products/icomox/ .

Note: Make sure you overwrite any files which previously existed in the destination directory.

16. Extract "iCOMOX_Monitor_v2.8.0.zip" and manually copy the extracted files into the project folder created in Step 5.
17. Open the "COMOX_GUI.py" file, then right click on the file title in the editor tab and select "Run iCOMOX_GUI.py".

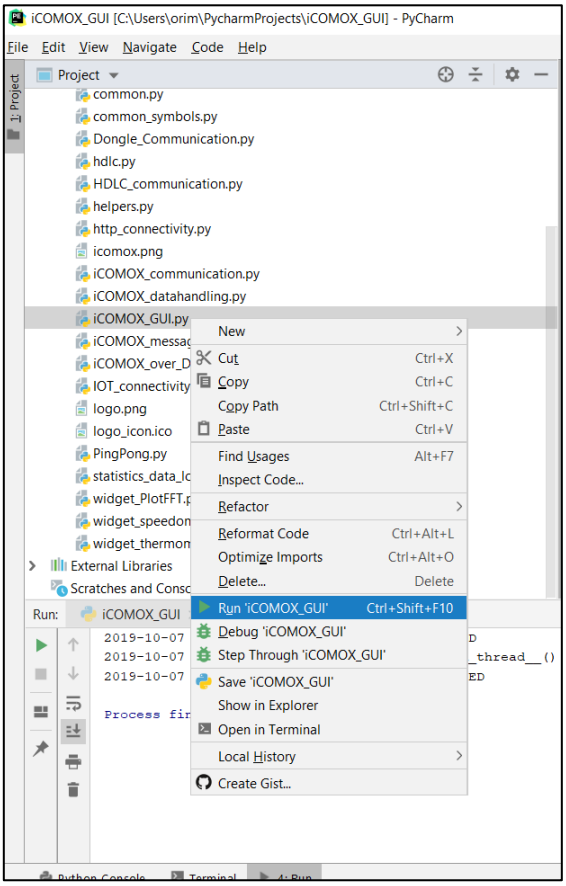

Figure 6: Running the project.

18. Wait a few moments until the Monitor window appear on the screen.

2 API

3.1 Introduction

The iCOMOX is the central component of the iCOMOX Data Acquisition (DA) kit. Its purpose is to acquire information through its sensors, for the kit's host.

The iCOMOX is controlled via an API which is described in this document. The API commands are described here as C structs, which are exchanged between the iCOMOX to the PC that controls it.

Different interfaces can add more complexity to the layers below the API (frame sync header, fragmentation control, etc.), in order to support its transfer over the respective interface. They are described elsewhere in the document.

All of the structs are packed so that there are no "memory holes". Structs size exactly equals the size of their respected fields.

Unless explicitly mentioned, all of the fields use a little-endian format.

The messages structs names use the prefix of sCOMOX_IN_MSG_xxx and sCOMOX_OUT_MSG_xxx. For all the interfaces, the "IN" messages mean that the direction is to user side while the "OUT" messages mean that the direction is to the iCOMOX side.

The iCOMOX always acknowledges any OUT message that it receives with a respective IN message; however, the opposite is not supported.

3.2 API Description

3.2.1 List of Message Codes

All messages begin with a “Code” field, which indicates the type of transmitted message. A pair of OUT and IN message always share the same value of their “Code” field.

The enum definition of the “Code” field type (eCOMOX_MSG_CODE) is displayed below:

```
typedef enum __attribute__((__packed__))
{
	cCOMOX_MSG_CODE_Hello,
	cCOMOX_MSG_CODE_Reset,
	cCOMOX_MSG_CODE_GetConfiguration,
	cCOMOX_MSG_CODE_SetConfiguration,
	cCOMOX_MSG_CODE_ReadEEPROM,
	cCOMOX_MSG_CODE_WriteEEPROM,
	cCOMOX_MSG_CODE_VerifyEEPROM,
	cCOMOX_MSG_CODE_COUNT,

	cCOMOX_MSG_CODE_Report = 0xFF
} eCOMOX_MSG_CODE;

typedef enum __attribute__((__packed__))
{
	cMODULE_RawData,
	cMODULE_AnomalyDetection,
	cMODULE_Maintenance,

	cMODULE_Debug
} eMODULE;
```

3.2.2 Hello Messages

a. sCOMOX_IN_MSG_Hello

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE                         Code;
        eCOMOX_BOARD_TYPE                       BoardType;
        sVERSION_MAJOR_MINOR                    BoardVersion;
        uCOMOX_SERIAL_NUMBER                    McuSerialNumber;
        sVERSION_MAJOR_MINOR_PATCH_BRANCH       FirmwareReleaseVersion;
        sDATE_TIME                              FirmwareBuildVersion;
        char                                    ProductPartNumber[32];
        char                                    ProductionSerialNumber[32];
        char                                    Name[32];
        eCOMOX_SENSOR_BITMASK                   BitStatus;

        union __attribute__((__packed__))
        {
                sSMIP_SOFTWARE_VERSION          SmipSoftwareVersion;
                sBG96_INFO                      BG96;
        };
} sCOMOX_IN_MSG_Hello;
```

Field name	Field Size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_Hello message type code.
BoardType	1	cCOMOX_BOARD_TYPE constant, to indicate the iCOMOX flavor: SmartMesh IP, NB-IoT or PoE.
BoardVersion	2	Struct which contains the board hardware assembly version. Currently for the SMIP iCOMOX it is: Major = 0, Minor = 0, which means the version is indeterminate. In the NB-IoT and PoE iCOMOX, a meaningful version is programmed during the production.
McuSerialNumber	16	A unique iCOMOX serial number. It is the unique ID of the microcontroller.
FirmwareReleaseVersion	4	The firmware version. Currently it is: Major = 2 Minor = 7 Patch = 0 Branch = cFIRMWARE_BRANCH_KIT If the major and the minor firmware versions are identical to that of the host, then it means their API is compatible.

FirmwareBuildVersion	7	The exact date and time in which the firmware was compiled.
BitStatus	1	Bitmask of the builtin test made on the sensors. For each sensor: “0” means OK, “1” means FAIL. Bit 0: ADXL362 sensor Bit 1: ADXL356B sensor BIT 2: BMM150 sensor BIT 3: ADT7410 sensor BIT 4: IM69D130 sensor
ProductPartNumber	32	This is a UTF-8 string which is programmed during the iCOMOX production. If it is shorter than 32 bytes, then it should be padded with bytes of 0xFF (preferred) or 0x00. It indicates the part number of the iCOMOX as identified by Shiratech. Note: This field is empty for the iCOMOX SMIP.
ProductionSerialNumber	32	This is a UTF-8 string which is programmed during the iCOMOX production. If it is shorter than 32 bytes, then it should be padded with bytes of 0xFF (preferred) or 0x00. It indicates the serial number of the specific iCOMOX as identified by Shiratech. Note: This field is empty for the iCOMOX SMIP.
Name	32	This is a UTF-8 string which is programmed by the user. If it is shorter than 32 bytes, then it should be padded with bytes of 0xFF (preferred) or 0x00. It contains a user name for the iCOMOX (e.g. “motor 1 of oil pump”). Note: This field is empty for the iCOMOX SMIP.
SmipSoftwareVersion	5	It contains the firmware version of the SmartMesh IP transceiver. Currently it is: Major = 1 Minor = 4 Patch = 1 Build = 9
BG96	1	It is a bitfield which contains the builtin test of the BG96 cellular transceiver of the iCOMOX-NB-IoT. It contains a combination of the cBG96_ACTIVITY_BITMASK_xxxx enum values. “1” means OK, “0” means failed or not tested. It indicates the UART communication to the BG96.

		If there is a SIM card it indicates if the SIM card was properly detected. If SIM card works properly then it indicates if registration to cell has been accomplished.

The sCOMOX_IN_MSG_Hello message is automatically sent by the iCOMOX after detecting a connection. It provides the host with all of the relevant version information that is needed to determine the validity of the API.

The definition of the non-standard types, that are used inside, are displayed below:

```
typedef struct __attribute__((__packed__))
{
	uint8_t			Major;
	uint8_t			Minor;
} sVERSION_MAJOR_MINOR;

typedef enum __attribute__((__packed__))
{
	cCOMOX_BOARD_TYPE_SMIP = 0,
	cCOMOX_BOARD_TYPE_NB_IOT,
	cCOMOX_BOARD_TYPE_POE,
} eCOMOX_BOARD_TYPE;

typedef enum __attribute__((__packed__))
{
	cFIRMWARE_BRANCH_KIT,
	cFIRMWARE_BRANCH_SUITCASE,
} eFIRMWARE_BRANCH;

typedef struct __attribute__((__packed__))
{
	uint8_t			Major;
	uint8_t			Minor;
	uint8_t			Patch;
	eFIRMWARE_BRANCH	Branch;
} sVERSION_MAJOR_MINOR_PATCH_BRANCH;

typedef struct __attribute__((__packed__))
{
	uint8_t			Major;
	uint8_t			Minor;
	uint8_t			Patch;
	uint16_t		Build;
} sSMIP_SOFTWARE_VERSION;

typedef enum __attribute__((__packed__))
{
	cBG96_ACTIVITY_BITMASK_UART		= 0x01,
	cBG96_ACTIVITY_BITMASK_SIM		= 0x02,
	cBG96_ACTIVITY_BITMASK_Registration	= 0x04,
} eBG96_ACTIVITY_BITMASK;

typedef struct __attribute__((__packed__))
{
	eBG96_ACTIVITY_BITMASK	Test;
} sBG96_INFO;
```

```
typedef union __attribute__((__packed__))
{
	uint8_t		bytes[16];
	uint16_t	words[8];
	uint32_t	dwords[4];
	uint64_t	qwords[2];
} uCOMOX_SERIAL_NUMBER;
```

b. sCOMOX_OUT_MSG_Hello

```
typedef struct __attribute__((__packed__))
{
	eCOMOX_MSG_CODE		Code;
} sCOMOX_OUT_MSG_Hello;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_Hello message type code.

The sCOMOX_OUT_MSG_Hello message contains only the "Code" field and triggers a response from the iCOMOX of the sCOMOX_IN_MSG_Hello.

Note: The iCOMOX-NB-IoT sends sCOMOX_IN_MSG_Hello messages every 10 seconds when it is active but have no messages to send, via its cellular interface. It serves as a keepalive mechanism.

3.2.3 Reset Messages

a. sCOMOX_IN_MSG_Reset

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE             Code;
} sCOMOX_IN_MSG_Reset;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_Reset message type code.

The sCOMOX_IN_MSG_Reset message is returned to the host after receiving the COMOX_OUT_MSG_Reset request message.

b. sCOMOX_OUT_MSG_Reset

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE             Code;
        eCOMOX_RESET_TYPE           ResetType;
} sCOMOX_OUT_MSG_Reset;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_Reset message type code.
ResetType	1	cCOMOX_RESET_TYPE constant: cCOMOX_RESET_TYPE_Software – forces the microcontroller to reset itself. cCOMOX_RESET_TYPE_Hardware – Disconnects the power to the iCOMOX components and thus forces a power reset. Currently it is only applicable to the iCOMOXNB-IoT and iCOMOX-PoE. cCOMOX_RESET_TYPE_FirmwareUpdate – unused

The sCOMOX_OUT_MSG_Reset message requests the iCOMOX to reset itself.

The definition of the eCOMOX_RESET_TYPE is displayed below:

```
typedef enum __attribute__((__packed__))
{
        cCOMOX_RESET_TYPE_Software,
        cCOMOX_RESET_TYPE_Hardware,
        cCOMOX_RESET_TYPE_FirmwareUpdate,
```

```
} eCOMOX_RESET_TYPE;
```

Note: After reset, the iCOMOX returns to its default configuration.

3.2.4 Get Configuration Messages

a. sCOMOX_IN_MSG_GetConfiguration

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE         Code;
        sCOMOX_CONFIGURATION    Configuration;
} sCOMOX_IN_MSG_GetConfiguration;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_GetConfiguration message type code.
Configuration	14	sCOMOX_CONFIGURATION structure.

The sCOMOX_IN_MSG_GetConfiguration message is returned to the host after receiving the sCOMOX_OUT_MSG_GetConfiguration request message. It returns the current configuration of the iCOMOX.

b. sCOMOX_OUT_MSG_GetConfiguration

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE         Code;
} sCOMOX_OUT_MSG_GetConfiguration;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_GetConfiguration message type code.

The sCOMOX_OUT_MSG_GetConfiguration message requests the iCOMOX to return its current configuration.

3.2.5 Set Configuration Messages

a. sCOMOX_IN_MSG_SetConfiguration

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE         Code;
        eCOMOX_RESULT           Result;
} sCOMOX_IN_MSG_SetConfiguration;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_SetConfiguration message type code.
Result	1	eCOMOX_RESULT result code of setting new configuration. Currently it can only report if it fails to create a file in the SD card.

The sCOMOX_IN_MSG_SetConfiguration message acknowledges receiving the sCOMOX_OUT_MSG_SetConfiguration request message.
If an error occurs, then a result code is reported in the Result field. The eCOMOX_RESULT type is defined as:

```
typedef enum __attribute__((__packed__))
{
        cCOMOX_RESULT_OK,
        cCOMOX_RESULT_UNKNOWN_ERROR,
        cCOMOX_RESULT_UNSUPPORTED_FEATURE,
        cCOMOX_RESULT_SD_CARD,
        cCOMOX_RESULT_INVALID_EEPROM_COUNT,
        cCOMOX_RESULT_INVALID_EEPROM_ADDRESS,
        cCOMOX_RESULT_INVALID_EEPROM_ADDRESS_AND_COUNT,
        cCOMOX_RESULT_EEPROM_WRITE_BOUNDARY_ERROR,
        cCOMOX_RESULT_EEPROM_VERIFY_FAILED,
        cCOMOX_RESULT_EEPROM_ACCESS_IS_NOT_ALLOWED,
        cCOMOX_RESULT_COUNT
} eCOMOX_RESULT;
```

b. sCOMOX_OUT_MSG_SetConfiguration

```
typedef struct __attribute__((__packed__))
{
```

```
        eCOMOX_MSG_CODE             Code;
        eCOMOX_CONFIG_BITMASK       ConfigBitmask;
        eMODULE_BITMASK             ConfigModulesBitmask;
        uCOMOX_CONFIG_Common        Common
        int64_t                     LocalTimestamp;
        uint16_t                    TransmitIntervalInMinutes;
        uint8_t                     TransmitRepetition;

        eMODULE_BITMASK                     ActiveModules;
        sCOMOX_MODULE_CONFIG_RawData        RawData;
        struct __attribute__((__packed__))
        {
                eANOMALY_DETECTION_COMMAND      Command;
                uANOMALY_DETECTION_SENSORSSensors;
                uint8_t                         StateToTrain;
        }                                   AnomalyDetection;
        sCOMOX_MODULE_CONFIG_Maintenance    Maintenance;
} sCOMOX_OUT_MSG_SetConfiguration;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_SetConfiguration message type code.
ConfigBitmask	1	Bitfield which defines which fields in the message are going to be used (“1”) and which fields in the message are ignored (“0”). It allows to make a partial configuration, without modifying previously configured fields. Any combination of the cCOMOX_CONFIG_BITMASK_xxx enum values can be used to configure the following fields: - LocalTimestamp - Common - TransmitIntervalInMinutes & TransmitRepetition (together)
ConfigModulesBitmask	1	Bitfield which defines which modules in the message are going to be used (“1”) and which modules in the

		message are ignored (“0”). It allows to configure some modules without modifying others. Any combination of the cMODULE_BITMASK enum values can be used to configure the currently 2 supported modules: - Raw data streaming module - Anomaly detection module
Common	1	Struct with bit fields which defines various functionalities of the iCOMOX: - CommChannel – to which channel the report messages should be redirected: USB (“0”) or Auxiliary (SMIP, NB-IoT or Ethernet) interface (“1”). - Vibrator – whether to activate the vibration motor (“1”) or to disable it (“0”). - Transmit – whether to transmit the report messages to the selected channel (“1”) or not (“0”). - SaveToFile – whether to save the report messages to a file in the SD card (“1”) or not (“0”). This field is used only if bit 1 of ConfigBitmask is “1”.
LocalTimestamp	8	Timestamp describes the number of seconds since Midnight that started 1.1.1970 in LOCAL TIME (this is why it is NOT UNIX epoch). It is used in order to provide the iCOMOX an absolute time reference – so the iCOMOX can create proper timestamps for the files it creates on the SD card. The iCOMOX keeps tracking of the time using its 32.768KHz crystal oscillator. This field is used only if bit 0 of ConfigBitmask is “1”.
TransmitIntervalInMinutes	2	Part of the schedule reporting mechanism: The number of minutes in which the iCOMOX waits before it triggers new schedule cycles. It is the user responsibility to ensure that the time it takes the iCOMOX to acquire, transmit and/or store all the active modules data is shorter than this interval. In the time between the last cycle of the current schedule and the first cycles of the next schedule, the iCOMOX neither transmits nor saves to the SD card any non-alert report messages. The iCOMOX-NB-IoT also puts the BG96 into sleep mode within 1 minutes after the completion of the last cycles

		of the current schedule, and awakes it once the new schedule begins. This field is used only if bit 2 of ConfigBitmask is "1".
TransmitRepetition	1	Part of the schedule reporting mechanism: The number of cycles (beyond 1) within a single schedule. "Cycle" means a single round in which every active module runs once (in series). 0 means a single cycle. 1 means 2 cycles, etc… This field is used only if bit 2 of ConfigBitmask is "1".
ActiveModules	1	Bitmasks field which determines which module is enabled ("1") or disabled ("0"). Any combination of cMODULE_BITMASK_xxx enum values can be used. Only the bits whose corresponding bits in the ConfigModulesBitmask is "1" are used.
RawData	1	A single field struct which contains the sensors which are sampled and possibly transmitted and/or saved to the SD card ("1) or not ("0"), for the raw data streaming module. Any combination of the cCOMOX_SENSOR_BITMASK_xxx can be used. This field is used only if bit 0 of ConfigModulesBitmask is "1".
AnomalyDetection	3	A struct which allows to send a reset or train preset command. For reset command (which erases all previously trained data, and prepares the anomaly detection module for action) one needs to assign the following values: - Command = cANOMALY_DETECTION_COMMAND_Reset - Sensors = bitmask of the sensors that needs to be used. - StateToTrain = don't care This command results in start getting anomaly detection inference report messages. For train command (which saves the current sensors data into the selected preset) one needs to assign the following values: - Command = cANOMALY_DETECTION_COMMAND_Train - Sensors = don't care

		- StateToTrain = the preset to train (0, 1, 2, or 3) This command results in 5 anomaly detection train reports messages. This field is used only if bit 1 of ConfigModulesBitmask is "1".
Maintenance	4	Currently it is not used. This field is used only if bit 2 of ConfigModulesBitmask is "1".

The sCOMOX_OUT_MSG_SetConfiguration message requests the iCOMOX to change its current configuration with the one that is provided in its various fields.

By proper use of the ConfigBitmask and ConfigModulesBitmask fields – partial configuration changes can be done to the iCOMOX.

3.2.6 Read EEPROM Messages

a. sCOMOX_IN_MSG_ReadEEPROM

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE         Code;
        uint8_t                 Count;
        uint16_t                Address;
        eCOMOX_RESULT           Result;
        uint8_t                 Data[32];
} sCOMOX_IN_MSG_ReadEEPROM;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_ReadEEPROM message type code.
Count	1	Number of bytes than have been actually read. It is always less than or equal to 32.
Address	2	The address of the first read byte.
Result	1	One of the cCOMOX_RESULT_xxx enum value to indicate if error occurs.
Data	32	The first Count bytes contains the data that has been read from the EEPROM, in case Result equals cCOMOX_RESULT_OK.

The sCOMOX_IN_MSG_ReadEEPROM message is returned to the host after receiving the sCOMOX_OUT_MSG_ReadEEPROM request message. It returns the requested bytes that are read from the iCOMOX internal EEPROM.

b. sCOMOX_OUT_MSG_ReadEEPROM

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE         Code;
        uint8_t                 Count;
        uint16_t                Address;
} sCOMOX_OUT_MSG_GetConfiguration;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_ReadEEPROM message type code.
Count	1	Number of bytes to read. Must always be less than or equal 32.

Address	2	The address in the iCOMOX's EEPROM from which to read the data. Currently this number is 0 to 8191.

The sCOMOX_OUT_MSG_ReadEEPROM message requests the iCOMOX to read bytes from its internal EEPROM and return them via the sCOMOX_IN_MSG_ReadEEPROM message.

Note: Reading from the current iCOMOX-SMIP always results in cCOMOX_RESULT_UNSUPPORTED_FEATURE.

3.2.7 Write EEPROM Messages

a. sCOMOX_IN_MSG_WriteEEPROM

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE         Code;
        uint8_t                 Count;
        uint16_t                Address;
        eCOMOX_RESULT           Result;
} sCOMOX_OUT_MSG_WriteEEPROM;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_WriteEEPROM message type code.
Count	1	Number of bytes to read. Must always be less than or equal 32.
Address	2	The address in the iCOMOX's EEPROM to which the data was written. Currently this number is 0 to 8191. Both the first written byte and the last written byte must be in the same page (page size is 32 bytes).
Result	1	One of the cCOMOX_RESULT_xxx enum value to indicate if error occurs.

The sCOMOX_IN_MSG_WriteEEPROM message is returned to the host after receiving the sCOMOX_OUT_MSG_WriteEEPROM request message. It returns the requested address and bytes that were written by the iCOMOX to its internal EEPROM.

b. sCOMOX_OUT_MSG_WriteEEPROM

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE         Code;
        uint8_t                 Count;
        uint16_t                Address;
        uint8_t                 Data[32];
} sCOMOX_OUT_MSG_WriteEEPROM;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_WriteEEPROM message type code.
Count	1	Number of bytes than have been actually read. It is always less than or equal to 32.

Address	2	The address in the iCOMOX's EEPROM to which the data was written. Currently this number is 0 to 8191. Both the first written byte and the last written byte must be in the same page (page size is 32 bytes).
Data	32	The first Count bytes that contains the data to be written to the EEPROM.

The sCOMOX_OUT_MSG_WriteEEPROM message requests the iCOMOX to write bytes into its internal EEPROM and return the write result via the sCOMOX_IN_MSG_WriteEEPROM message.

Note: Writing to the current iCOMOX-SMIP always results in cCOMOX_RESULT_UNSUPPORTED_FEATURE.

3.2.8 Verify EEPROM Messages

a. sCOMOX_IN_MSG_VerifyEEPROM

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE         Code;
        uint8_t                 Count;
        uint16_t                Address;
        eCOMOX_RESULT           Result;
} sCOMOX_IN_MSG_VerifyEEPROM;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_VerifyEEPROM message type code.
Count	1	Number of bytes to verify. Must always be less than or equal 32.
Address	2	The address in the iCOMOX's EEPROM from which the data was verified. Currently this number is 0 to 8191.
Result	1	One of the cCOMOX_RESULT_xxx enum value to indicate if error occurs. If Result is cCOMOX_RESULT_OK then the verification of all the bytes that were provided in sCOMOX_OUT_MSG_VerifyEEPROM succeeded. If Result is cCOMOX_RESULT_EEPROM_VERIFY_FAILED then at least a single byte in the EEPROM is different from its corresponding provided byte in the sCOMOX_OUT_MSG_VerifyEEPROM message.

The sCOMOX_IN_MSG_VerifyEEPROM message is returned to the host after receiving the sCOMOX_OUT_MSG_VerifyEEPROM request message. It returns indication if the comparison between the bytes to verify and the actual EEPROM content succeeded (all bytes are identical) or failed (verification failure).

b. sCOMOX_OUT_MSG_VerifyEEPROM

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE                 Code;
        uint8_t                         Count;
        uint16_t                        Address;
        uint8_t                         Data[32];
} sCOMOX_OUT_MSG_VerifyEEPROM;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_VerifyEEPROM message type code.
Count	1	Number of bytes to verify. It is always less than or equal to 32.
Address	2	The address in the iCOMOX's EEPROM from which the verification should start. Currently this number is 0 to 8191.
Data	32	The first Count bytes that contains the data to be verified with the actual EEPROM content.

The sCOMOX_OUT_MSG_VerifyEEPROM message requests the iCOMOX to verify the contents of the internal EEPROM and returns the verification result via the sCOMOX_IN_MSG_VerifyEEPROM message.

Note: Verifying to the current iCOMOX-SMIP always results in cCOMOX_RESULT_UNSUPPORTED_FEATURE.

3.2.8 Debug Messages

a. sCOMOX_IN_MSG_Debug

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE             Code;
        eCOMOX_DEBUG_NBIOT_CMD      Cmd;
        eCOMOX_RESULT               Result;
} sCOMOX_IN_MSG_Debug;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_Debug message type code.
Cmd	1	eCOMOX_DEBUG_NBIOT_CMD command type code. It is equal to the Cmd field in sCOMOX_OUT_MSG_Debug.
Result	1	One of the cCOMOX_RESULT_xxx enum value to indicate if error occurs. If Result is cCOMOX_RESULT_OK then the command was successfully received. In case that Cmd = cCOMOX_DEBUG_NBIOT_CMD_Reset_BG96 it also indicates that the reset procedure of the BG96 completed (it happens after about 20 seconds). If the other valid commands were sent, then debug reports messages will be generated as well.

The sCOMOX_IN_MSG_Debug message is returned to the host after receiving the sCOMOX_OUT_MSG_Debug request message. It returns indication if the request message is a legal command, and puts the iCOMOX-NB-IoT in a debug mode which can be left only by sending a reset command to the iCOMOX. Commands of manual sending of AT commands and test connectivity results in sending sCOMOX_REPORT_Debug reports.

b. sCOMOX_OUT_MSG_Debug

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE                 Code;
        union __attribute__((__packed__))
        {
                struct __attribute__((__packed__))
                {
                        eCOMOX_DEBUG_NBIOT_CMD    Cmd;
                        union __attribute__((__packed__))
                        {
                                struct __attribute__((__packed__))
                                {
```

```
                                uint16_t     TimeoutInMiliSeconds;
                                char         ATCommand[128];
                            } Send_AT_Command;
                            struct __attribute__((__packed__))
                            {
                            } Reset_BG96;
                            struct __attribute__((__packed__))
                            {
                            } TestConnectivity;
                        };
                    } NBIOT;
                };
} sCOMOX_OUT_MSG_Debug;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_Debug message type code.
Cmd	1	eCOMOX_DEBUG_NBIOT_CMD command type code.
TimeoutInMiliSeconds	2	The address in the iCOMOX's EEPROM from which the verification should start. Currently this number is 0 to 8191.
ATCommand	128	Used only in the case when Cmd = cCOMOX_DEBUG_NBIOT_CMD_Send_AT_Command. It servers as the AT command string that is manually sent to the BG96. If the length of the AT command is shorter than 128 bytes, then it should be padded with at least one NULL character.

The sCOMOX_OUT_MSG_Debug message requests the iCOMOX to send manual AT command to the BG96, to reset the BG96 or to test the connectivity via the BG96. Determining it is done by the Cmd field. Reset the BG96 is a blocking command and may take about 20 seconds until a response arrives back. The other commands are not blocking, and after getting a successful acknowledge – debug report messages are generated and sent to the host.

Note: Verify USB connectivity with the iCOMOX before sending the sCOMOX_OUT_MSG_Debug command. Also remember that only resetting the iCOMOX can force it to leave the debug mode.

3.2.9 Report Message

a. sCOMOX_IN_MSG_Report

The sCOMOX_IN_MSG_Report is the central message in the iCOMOX system. Its purpose is to provide the various modules reports – all of them are based on samples from the various sensors of the iCOMOX.

This is the only message that is unidirectional (from the iCOMOX to the host only; there is no explicit request message for it).

This message also has several different forms, depending on the kind of module, sensor and axis (iCOMOX-SMIP only) about which the message reports.

The message structure is displayed below:

```
typedef struct __attribute__((__packed__))
{
        eCOMOX_MSG_CODE                 Code;
        uCOMOX_REPORT_PAYLOAD_TYPE      PayloadType;
        int64_t                         Timestamp;
        union __attribute__((__packed__))
        {
                uint8_t                 payload[0];
                sPAYLOAD_ADXL362        ADXL362;
                struct __attribute__((__packed__))
                {
                        saPAYLOAD_ADXL356       ADXL356;
#if COMOX_BOARD_TYPE == 0       // SMIP
                        int16_t                 ADXL356_ExtraSamples[ADXL356_SAMPLES_NUM_PER_PIN/4];
#else   // NB-IOT & POE
                        int16_t
        ADXL356_ExtraSamples[ADXL356_SAMPLES_NUM_PER_PIN*3/4];
#endif
                };
                s16aPAYLOAD_BMM150              BMM150;
                sPAYLOAD_ADT7410                ADT7410;
                sPAYLOAD_IM69D130               IM69D130;
                struct __attribute__((__packed__))
                {
                        saPAYLOAD_ADXL1002      ADXL1002;
                        int16_t ADXL1002_ExtraSample[ADXL1002_SAMPLES_NUM_PER_PIN/4];
                };
                sPAYLOAD_AnomalyDetection       AnomalyDetection;
                sPAYLOAD_Maintenance            Maintenance;
        };
} sCOMOX_IN_MSG_ReportBuffer;
```

The sCOMOX_IN_MSG_Report constants and related defined types are displayed below:

```
typedef enum __attribute__((__packed__))
{
	cMODULE_RawData,
	cMODULE_AnomalyDetection,
	cMODULE_Maintenance,

	cMODULE_Debug
} eMODULE;


typedef enum __attribute__((__packed__))
{
	cCOMOX_SENSOR_BITMASK_ADXL362		= 1 << 0,
	cCOMOX_SENSOR_BITMASK_ADXL356		= 1 << 1,
	cCOMOX_SENSOR_BITMASK_BMM150		= 1 << 2,
	cCOMOX_SENSOR_BITMASK_ADT7410		= 1 << 3,
	cCOMOX_SENSOR_BITMASK_IM69D130		= 1 << 4,
	cCOMOX_SENSOR_BITMASK_ADXL1002		= 1 << 5,


	cCOMOX_SENSOR_BITMASK_TOP		= 1 << 5,
} eCOMOX_SENSOR_BITMASK;


typedef union __attribute__((__packed__))
{
	uint8_t	value;
	struct __attribute__((__packed__))
	{
		eCOMOX_SENSORSensor : 3;
		uint8_t		    : 1;
		eAXIS		Axis   : 2;
		eMODULE		Module : 2;
	};
	struct __attribute__((__packed__))
	{
		uint8_t			: 6;
		eMODULE		Module	: 2;
	} AnomalyDetectionModule;
	struct __attribute__((__packed__))
	{
		uint8_t		Command	: 2;
		uint8_t			: 3;
		bool		LastPacket	: 1;
		eMODULE		Module	: 2;
	} DebugModule;
} uCOMOX_REPORT_PAYLOAD_TYPE; #define ADXL362_SAMPLES_NUM 1024


typedef int16_t  int16_vector_t[3];
```

```
typedef struct __attribute__((__packed__))
{
	int16_vector_t	samples [ADXL362_SAMPLES_NUM];
} sPAYLOAD_ADXL362;

#if COMOX_BOARD_TYPE == 0	// SMIP
	#define ADXL356_SAMPLES_NUM_PER_PIN(8192)
#else	// NB-IOT & POE
	#define ADXL356_SAMPLES_NUM_PER_PIN(2048)
#endif

typedef struct __attribute__((__packed__))
{
#if COMOX_BOARD_TYPE == 0	// SMIP
	int16_t  Samples[ADXL356_SAMPLES_NUM_PER_PIN*3/4];   // Single axis report message, after
packing
#else	// NB-IOT & POE
	int16_vector_t	samples [ADXL356_SAMPLES_NUM_PER_PIN*3/4];// Combined 3 axis report
message, after packing
#endif
} saPAYLOAD_ADXL356;

#define BMM150_SAMPLES_NUM (512)

typedef struct __attribute__((__packed__))
{
	int16_vector_t	samples[BMM150_SAMPLES_NUM];
} s16aPAYLOAD_BMM150;

typedef struct __attribute__((__packed__))
{
	int16_t		Temp_Q9_7;
} sPAYLOAD_ADT7410;

#define IM69D130_LR_WAKEUP_SAMPLES_NUM	(420)		// This provides more than 20msec
delay, for sample rate of 20.3125KSPS
#define IM69D130_LR_SAMPLES_NUM		(2468-420)
```

```
#define IM69D130_SAMPLES_NUM                    (IM69D130_LR_SAMPLES_NUM/2)

typedef struct __attribute__((__packed__))
{
        uint16_t s16aPAYLOAD_IM69D130[IM69D130_SAMPLES_NUM];
} sPAYLOAD_IM69D130;

typedef union __attribute__((__packed__))
{
        int32_t         uint32;
        int16_t         int16[2];
} sMIC;

typedef struct __attribute__((__packed__))
{
        sMIC    Left,
                Right;
} sTDM2;

// ADXL1002
#define ADXL1002_SAMPLES_NUM_PER_PIN        (2048)

typedef struct __attribute__((__packed__))
{
        int16_t  Samples[ADXL1002_SAMPLES_NUM_PER_PIN*3/4]; // Single axis packed samples.
} saPAYLOAD_ADXL1002;

typedef enum __attribute__((__packed__))
{
        cANOMALY_DETECTION_REPORT_TYPE_Inference,
        cANOMALY_DETECTION_REPORT_TYPE_Train,
} eANOMALY_DETECTION_REPORT_TYPE;

typedef union __attribute__((__packed__))
{
        uint8_t         value;
        struct __attribute__((__packed__))
```

```
        {
                bool    ModelLearned   : 1;
                uint8_t                : 6;
//#if COMOX_BOARD_TYPE == 0   // SMIP
//              bool Sensor_ACC_X              : 1;
//              bool Sensor_ACC_Y              : 1;
//              bool Sensor_ACC_Z              : 1;
//              bool Sensor_MAG                : 1;
//              bool Sensor_MIC                : 1;
//              bool Sensor_TEMP               : 1;
//#else  // NBIOT & POE
//              bool Sensor_ACC                : 1;
//              bool Sensor_MAG                : 1;
//              bool Sensor_MIC                : 1;
//              bool Sensor_TEMP               : 1;
//              uint8_t                        : 2;
//#endif
                eANOMALY_DETECTION_REPORT_TYPE    ReportType : 1;
        };
} uANOMALY_DETECTION_REPORT_STATUS;

typedef struct __attribute__((__packed__))
{
        sEAI_RESULT                         AnomalyResult;
        uANOMALY_DETECTION_REPORT_STATUS    ReportStatus;
        uint8_t                             Sensors;
        eEAI_RET                            Result;
} sPAYLOAD_AnomalyDetection;

typedef struct __attribute__((__packed__))
{
        char        Response[DEBUG_REPORT_SIZE];
} sPAYLOAD_Debug;
```

Field name	Field size (bytes)	Description
Code	1	cCOMOX_MSG_CODE_Report message type code.

PayloadType	1	It has uCOMOX_REPORT_PAYLOAD_TYPE type and contains the following bit fields: Module (2 bits, MSB) – of eMODULE. The value in this field determines which firmware module produced the report. The meaning of the other bits in PayloadType depends on this field value. For raw sensors data module, the following bit fields are available: Sensor (3 bits, LSB) – of eCOMOX_SENSOR enum type. It determines which sensor data is provided in the payload field (thus determines the message's actual size). Axis (2 bits, 4 bits offset) – it is eAXIS type. Defines which axis data is requested from the AXDL356B sensor. For anomaly detection module, no other fields are available. For debug module, the following bit fields are available: Command (2 bits, LSB) - determines which debug command was sent (send AT command manually, reset BG96, test connectivity) Last packet (1 bit, 5 bits offset) – determines if the debug report is the last in the sequence.
Timestamp	8	Number of ticks (1 tick=1/32768 second) since January 1st, 1970 in local time zone (if the local timestamp was configured before), else it is the number of ticks since the microcontroller application started.
	1. For streaming raw data module: it depends on the value of the "Sensor" field (and "Axis" field for the iCOMOX-SMIP): a. ADXL362 – 6144 b. ADXL356 – 12288 (for each axis in the iCOMOX-SMIP case) ADXL356 – 9216	This field is an anonymous field, which allows direct access to the modules: 1. Streaming raw data module: It can contain acceleration data of the ADXL362 accelerometer, acceleration data of a single axis (iCOMOX-SMIP) or all 3 axis (iCOMOX-NB-IoT & iCOMOX-PoE) from the ADXL356 accelerometer, magnetic fields from the BMM150 magnetometer, temperature from

	(all axis together, iCOMOX-NB-IoT and iCOMOX-PoE) c. BMM150 –3072 d. ADT7410 – 2 e. IM69D130 – 2048 f. ADXL1002 – 4096 2. Anomaly Detection module: a. Inference report – 23 b. Train report – 23 3. Debug module: it always contains payload of 2048 characters. In case the returned string is shorter than that, then a NULL character is used to mark the termination.	the ADT7410 and voice samples from the microphone IM69D130, and for some iCOMOXs also acceleration data of single axis from the ADXL1002 accelerometer.

Raw data streaming module reports:

The ADXL356 payload and the ADXL1002 payload - each contains samples of 12 bit (signed number) which are packed in a 16 bits array. The LSB MSB relations are preserved after the packing, so the first 3 elements of the 16 bits array (elements 0, 1 & 2) contain the following 4 (12 bit) samples (S0 to S3). In the image below we demonstrate how 4 samples of 12 bits (each sample has a different color) are packed into 3 elements of 16 bits:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	S1.3	S1.2	S1.1	S1.0	S0.11	S0.10	S0.9	S0.8	S0.7	S0.6	S0.5	S0.4	S0.3	S0.2	S0.1	S0.0
1	S2.7	S2.6	S2.5	S2.4	S2.3	S2.2	S2.1	S2.0	S1.11	S1.10	S1.9	S1.8	S1.7	S1.6	S1.5	S1.4
2	S3.11	S3.10	S3.9	S3.8	S3.7	S3.6	S3.5	S3.4	S3.3	S3.2	S3.1	S3.0	S2.11	S2.10	S2.9	S2.8

The ADXL356 is sampled with rate of about 48.3KSPS (for iCOMOX-SMIP) and with rate of about 3.611KSPS (for iCOMOX-NB-IoT & iCOMOX-PoE), ADC reference voltage of 1.8V, sensitivity of 0.08V/g, and 0g is at 0.9V. So, converting an unsigned sample to acceleration is performed using the following formula:

$$\frac{a}{g} = \frac{([unsigned\ sample] - 2048) * 1.8V}{4096 * 0.08V/g}$$

When g = 9.81 m/s^2

The BMM150 data is an array of 512 int16 elements, each contains the magnetic field in the X, Y, Z directions (please refer to the s16aPAYLOAD_BMM150 data struct). The units are of 1/16µT. The sample rate is about 333.3 SPS.

$$B[\mu T] = \frac{[signed\ sample]}{16}$$

The IM69D130 data is an array of 1024 int16 elements, each element contains the sound pressure measured by the microphone. The units are about 48.2532 SPL (calculated from AOP of 130 dB SPL represented as 65535). The sample rate is about 20.3125KSPS. Conversion of a signed data to air pressure is done using the following formula:

$$P[SPL] = [signed\ sample]\frac{10^{130/20}}{65535} \approx [signed\ sample] \cdot 48.2532 SPL$$

The ADT7410 data is an int16 scalar with fixed point format of Q9.7 in [°C], so converting the signed data to temperature is done using the following formula:

$$T[°C] = \frac{[signed\ sample]}{128}$$

The ADXL1002 (if assembled) is sampled with rate of 25KSPS. ADC reference of 1.8V and (after rescaling and biasing its output) its sensitivity is 0.018V/g and 0g is at 0.9V. So, converting an unsigned sample to acceleration is performed using the following formula:

$$\frac{a}{g} = \frac{([unsigned\ sample] - 2048) * 1.8V}{4096 * 0.018V/g}$$

When g = 9.81 m/s^2

Anomaly Detection Reports:

This module uses the same structure to transfer 2 different kind of reports. For both of these reports, the "Result" field must be 0 in order that the other fields may contain any meaningful information. The kind of the anomaly detection report is determined by bit 7 in the "ReportStatus" field. "0" means it is an "Inference report" and "1" means it is a "Train report".

1. Inference report – the payload of this report is the "AnomalyResult" field. This field is struct which contains 2 fields:
    a. valAnomaly – a float32 which contains a number between 0.0 to 100.0 which indicates how much the current system state is **not** compatible with any of the previously learned states. The higher this number, the system current state is less compatible. There is no definite threshold which is universal to all the systems.
    b. probState[4] – a float32 array with 4 elements. This array represents the distribution of the current system state over the previously learned presets (the number of learned presets can be 0, 1, 2, 3 or 4. Each preset can be trained with up to 32 different system states). The higher the value of each element – the more confident we are the current system state is similar to a state that was learned before and stored in the corresponding preset.

2. Train report – it contains no payload. Currently, after getting sCOMOX_OUT_MSG_SetConfiguration with a train request of one of the 4 presets, the host receives 5 train reports. After transferring all these 5 train reports, the host should assume that the current system state was learned, and added to the corresponding preset database.

Note: Once the sCOMOX_OUT_MSG_SetConfiguration is received with a command to reset the Anomaly Detection module, then all the presets' databases are erased.

3.3 Configuration Structure

The configuration is stored in a 10 bytes struct as follows:

```
typedef struct __attribute__((__packed__))
{
        uCOMOX_CONFIG_Common                    Common;
        uint8_t                                 TransmitRepetition;
        uint16_t                                TransmitIntervalInMinutes;
        eMODULE_BITMASK                         ActiveModules;
        sCOMOX_MODULE_CONFIG_RawData            RawData;
        sCOMOX_MODULE_CONFIG_AnomalyDetection   AnomalyDetection;
} sCOMOX_CONFIGURATION;
```

The meaning of all the fields except the “AnomalyDetection” is explained in the section of sCOMOX_OUT_MSG_SetConfiguration. The “AnomalyDetection” struct is currently irrelevant for the host and thus is beyond the scope of this document.

The sCOMOX_CONFIGURATION uses the following type defs:

```
typedef enum __attribute__((__packed__))
{
        cCOMOX_SENSOR_BITMASK_ADXL362           = 1 << 0,
        cCOMOX_SENSOR_BITMASK_ADXL356B          = 1 << 1,
        cCOMOX_SENSOR_BITMASK_BMM150            = 1 << 2,
        cCOMOX_SENSOR_BITMASK_ADT7410           = 1 << 3,
        cCOMOX_SENSOR_BITMASK_IM69D130          = 1 << 4,
        cCOMOX_SENSOR_BITMASK_ADXL1002          = 1 << 5,

        cCOMOX_SENSOR_BITMASK_TOP               = 1 << 5,
} eCOMOX_SENSOR_BITMASK;

typedef enum __attribute__((__packed__))
{
        cCOMOX_COMM_CHANNEL_USB,
        cCOMOX_COMM_CHANNEL_AUX,
} eCOMOX_COMM_CHANNEL;

typedef union __attribute__((__packed__))
{
```

```
        uint8_t         value;
        struct __attribute__((__packed__))
        {
                eCOMOX_COMM_CHANNEL     CommChannel   : 1;
                bool                    Vibrator      : 1;
                uint8_t                               : 4;
                bool                    Transmit      : 1;
                bool                    SaveToFile    : 1;
        };
} uCOMOX_CONFIG_Common;

typedef struct __attribute__((__packed__))
{
        eCOMOX_SENSOR_BITMASK   Sensors;
} sCOMOX_MODULE_CONFIG_RawData;

typedef struct __attribute__((__packed__))
{
        uint8_t         target;
        uint8_t         selSensor;
        // actual number of learned
        uint8_t         count_learned;  // = NUMBER_LEARNED;
        uint8_t         model_learned;  // = 0;
} sCOMOX_MODULE_CONFIG_AnomalyDetection;
```

The configuration struct contains the iCOMOX state. Affecting the iCOMOX normal operation is done implicitly by changing the iCOMOX state.

3.4 Connecting and Configuring the iCOMOX

To connect and configure the iCOMOX, perform the following steps:

1. When a valid sCOMOX_IN_MSG_Hello message is received, go to step **4.**
2. Send sCOMOX_OUT_MSG_Hello message.
3. Receive incoming messages for a short time and go to step **1.**
4. Send sCOMOX_OUT_MSG_SetConfiguration message with the desired kind of reports.
5. Receive incoming messages for a short time.
6. When a valid sCOMOX_IN_MSG_Report message is received, then process it and go to step **5** again.
7. Go to step **4.**

3 Frame synchronization in different iCOMOX interfaces

3.1 USB interface

All the iCOMOX have a USB interface. The iCOMOX implements a UART over USB with the following UART parameters:

- 8 bits data word
- 2 stop bits
- 125K baud
- No parity
- No software and no hardware flow control

Every message sent from the iCOMOX to the host starts with the following 8 bits characters: “KOBI” (0x4B, 0x4F, 0x42, 0x49). Immediately after this prefix the API struct is sent too. If this prefix appears inside the API struct, then it is considered to be part of the API struct. If the host fails to receive this prefix, then it should wait until it succeeds to recognize it again, and then to assume that API struct arrives immediately after it.

Every message sent from the host to the iCOMOX should starts with a BREAK condition (few mili-seconds of BREAK condition is enough). After the removal of the BREAK condition, the API struct should be sent.

3.2 SMIP interface

The iCOMOX-SMIP has a wireless interface to mesh network created by Shiratech's SMIP dongle. This dongle is connected to the host via a USB interface. The dongle implements HDLC packets over UART over USB with the following UART parameters:

- 8 bits data word
- 1 stop bits
- 115.2K baud
- No parity
- No software and no hardware flow control

Explanation about the API between the dongle and the host is beyond the scope of this document.

Each iCOMOX-SMIP wireless interface has a unique MAC address, which allows it to have a unique identity in the mesh network. It is used to allow the iCOMOX-SMIP to transfer packets to/from dongle and/or other iCOMOX-SMIP devices.
The firmware supports communication only between an iCOMOX-SMIP to the host (via the dongle, and maybe also other iCOMOX-SMIP devices which serve as routing nodes in the mesh).
The communication is done via packets of 90 bytes that can be sent from the iCOMOX-SMIP to the dongle. Since most of the API-struct are much larger than 90 bytes, the iCOMOX decompose each API struct to groups of 89 bytes and sends them over the SMIP packet.
The first byte in each SMIP packet indicates the index of the SMIP packet:

- The 1st SMIP packet that contains the API-struct first group of 89 bytes, contains 0.
- The 2nd SMIP packet that contains the API-struct second group of 89 bytes, contains 1.
- The 3rd SMIP packet that contains the API-struct third group of 89 bytes, contains 2.

...

- The last SMIP packet may contains the API-struct last group of 89 (or less) bytes, contains:

$$\left\lfloor \frac{API\ struct\ size\ in\ bytes}{89} \right\rfloor$$

Once the host detects an SMIP packet with an unexpected index, it should wait until it gets a packet with packet index of 0, so it knows it is the beginning of a new API struct.
The iCOMOX-SMIP never transmits more than a single API struct on a single SMIP packet, even if the respected sizes allow it.

SMIP packet from the dongle to an iCOMOX-SMIP are limited to 82 bytes, but there, no packet index mechanism is applied. In addition, the host never transmits more than a single API struct on a single SMIP packet, if the respected sizes allow it.

3.3 TCP/IP interface

The iCOMOX-NB-IoT and the iCOMOX-PoE supports a TCP/IP communication. The iCOMOX behave as the client – which means that they initiate the connection procedure to the remote server. If the link is disconnected then the iCOMOX will try to initiate the connection procedure again and again – until a link is reestablished.
Since the TCP is a byte-oriented protocol which takes care on the frame boundaries, no special frames synchronization steps are done by iCOMOXs which use this interface, except of the following:
Once either the host or the iCOMOX detects an illegal API-struct, it gracefully disconnects the link. Then both sides wait until the iCOMOX will reestablish the link, as mentioned before.

Both sides apply TCP keepalive procedure in order to periodically checks connectivity.

Note: The iCOMOX-NB-IoT has no TCP keepalive, so it sends sCOMOX_IN_MSG_Hello to the remote host every 10 seconds, during periods of inactivity.

Note: The iCOMOX-NB-IoT puts its cellular IC into sleep mode, when it is configured to provide a schedule reporting. 60 seconds after the last message in the schedule completed, the IC is entered into sleep mode. Then the remote host lost its ability to control the iCOMOX-NB-IoT during the sleep time, until the next schedule arrives.

4 Files Descriptions

4.1 Python files description

File name	Description
common.py	Defines application global variables
common_symbols.py	Defines application global switches
helpers.py	Various helpers' functions (format strings, debug printing)
iCOMOX_communication.py	Defines class_iCOMOX_Communication which handles the read and write aspects of a generalized serial communication
HDLC_communication.py	Defines class_HDLC_Communication which adds HDLC support to the class_iCOMOX_Communication
hdlc.py	Helper file for HDLC_communication.py for checksum calculations
Dongle_Communication.py	Defines class class_Dongle_Communication which adds the specific dongle support to class_class_HDLC_Communication
iCOMOX_over_Dongle_communication.py	Defines class class_iCOMOX_over_Dongle_Communication which adds a layer of iCOMOX handling via the dongle to the class_Dongle_Communication
widget_PlotFFT.py	Defines class PlotFFT to display FFT graphs
widget_speedometer.py	Defines class Speedometer to display speed widget
widget_thermometer.py	Defines class Thermometer to display temperatures widget
BinFileConversion.py	Defines BinFileConversion() function to convert binary file recorded by the iCOMOX on SD card, to a text file readable by humans
iCOMOX_GUI.py	Main application
single_app_instance.py	Helper for checking if application instance already exists
statistics_data_logger.py	Defines class ClassStatisticalDataLogger for keeping/updating the statistics of the incoming sensors data: minimum value, maximum value, mean and standard deviation
PingPong.py	Defines class ClassPingPongArr that implements several double buffering handlers (for each sensor message)
iCOMOX_datahandling.py	Defines class class_DataHandling for handling all the aspects of receiving the incoming data and processing it
ADXL362.py	Defines class class_ADXL362 which centralizes all the aspects of analyzing the samples from the ADXL362 (accelerometer)
ADXL356.py	Defines class class_ADXL356 which centralizes all the aspects of analyzing the samples from the ADXL356 (accelerometer)

ADXL1002.py	Defines class class_ADXL1002 which centralizes all the aspects of analyzing the samples from the ADXL1002 (accelerometer)
BMM150.py	Defines class class_BMM150 which centralizes all the aspects of analyzing samples from the BMM150 (magnetic field sensor)
IM69D130.py	Defines class class_IM69D130 which centralizes all the aspects of analyzing samples from the IM69D130 (microphone)
iCOMOX_messages.py	Defines constants and functions to compose and decipher the messages to/from the iCOMOX
http_connectivity.py	Experimental REST reports
IOT_connectivity.py	Experimental IOTC reports

5 Document Revision History

Revision	Date	Author	Status and Description
2.0	29/07/2019	Ori Makover	Installation, Overview and Quick Start
2.0	28.08.2019	M Elias	Revision
3.0	12.09.2019	Ori Makover	Re-organization
3.0	15.09.2019	M Elias	Editing
3.1	7.10.2019	Kobi de Trenewan	Changing the Hello, Reset and Report messages. Adding information regarding the content of the reports' payloads.
3.1	7.10.2019	Ori Makover	New software version – changes in Monitor Python project and in API (Kobi)
3.2	27.10.2019	Kobi de Trenewan	Adding pywin32 library to the Python environment initialization. Updating aspects of the Hello and Reset (IN) messages. Mirroring the ADXL356B packing structure picture, and replace the term "compression" with "packing". Fixing and adding information regarding the reports' payloads (especially for the IM69D130). Added file descriptions tables.
3.3	17.11.2019	Kobi de Trenewan	Fixing the configuration structure, and the IN_MSG_SetConfiguration message. Modifying sections 4.1 & 4.2. Adding sections 4.3 & 4.4.
3.4	23.4.2020	Kobi de Trenewan	Fixing the API chapter in order to align it to version 2.7.0. Adding chapter 4 about frame synchronization for the USB, SMIP & TCP/IP. Replace any ADXL356B to ADXL356. Added ADXL1002 in the list of Python & C files. Added lwip 2.1.2 as a 3rd party library.
3.5	14.07.2020	Ori Makover	Removed C project references
3.5	14.07.2020	Kobi de Trenewan	Fixing the API chapter in order to align it to version 2.8.0

T. +972.3.943.5050 F. +972.3.943.5055 E. info@shiratech-solutions.com

58 Amal St, Kiryat Arie POB 3272, PetachTikva 4951358, Israel

www.shiratech-solutions.com