Programming Guide - **HAL API**

HAL-API for software development with **m:explore** ultra-wideband sensors

Ilmsens GmbH
Ehrenbergstraße 11
98693 Ilmenau
Germany

Tel.: +49 3677 76130-30
Fax: +49 3677 76130-39
Email: hal-api@ilmsens.com

Table of contents

1 HAL API for **m:explore** UWB sensors

1.1 Introduction

As a service to our customers who want to integrate Ilmsens UWB sensors into their software environment, Ilmsens offers a hardware abstraction layer (HAL) application programming interface (API) ("The software") for the **m:explore** ultra-wideband (UWB) sensors. The HAL API comes in form of a dynamic library working on top of the device drivers with corresponding C header files. This manual is a programming guide for the API. A software developer will find design and basic background information to ease application development. A reference manual with an up-to-date function description is provided separately by Ilmsens.

The HAL API is available for different operating systems and allows device management, sensor configuration, acquisition configuration, and performing measurements. It abstracts from device- or digital interface-specific details as much as possible to enable portability of the application software. Future generations of the HAL API and Ilmsens UWB sensors will be developed with backward compatibility in mind. Product-specific extensions will extend the API rather than changing existing functions.

By default, the HAL API is provided as software in binary (compiled) form for many popular operating systems. Should the provided functionality be insufficient for the customer's purposes, please contact Ilmsens for other options (see section 4.1). We provide limited software support for the HAL API. To report problems, ask for help, or suggest improvements, please contact Ilmsens (see section 4.1).

1.2 Copyright and Disclaimer

1.3 Platform support

The HAL API is currently available for the following operating system platforms:

- Windows™ Vista™ / Windows™ 7 / Windows™ 8.x / Windows™ 10
  - x86 (32 Bit) and x64 (64 Bit) versions
- Standard Linux distributions (currently those based on Debian package management)
  - i386 (32 Bit), amd64 (64 Bit), armel (ARM ports) versions
  - Ubuntu LTS 14.04 "Trusty Tahr" and up
  - Ubuntu LTS 16.04 "Xenial Xerus" and up
  - Ubuntu 16.10 "Yakkety Yak" and up
  - Debian LTS 7 "Wheezy" and up
  - Debian 8 "Jessie"

If you want to use the HAL API on other Linux distributions, please contact Ilmsens (see section 4.1). Support may be possible, if a sufficient development C/C++ tool chain is available for your distribution and Ilmsens has access to a reference installation.

Support for other platforms, such as MacOS™, Solaris™, FreeBSD, etc., may be added in the future.

1.4 Platform requirements

The HAL API works on top of the device drivers provided with your **m:explore** sensor and requires the sensor(s) to be connected to the computer and powered on for useful operation. Therefore, these platform requirements apply:

- Any of the supported operating systems listed under section 1.3
- USB2.0 host port
- Windows OS™:
  - USB device drivers provided with the Ilmsens **m:explore** evaluation kit
  - Microsoft Visual C++™ redistributable runtime in a matching version (included in the HAL package, usually already installed on Windows™ PCs)
- Linux OS:
  - LibUSB-1.0-0: V1.0.11 (or later) package installed
  - LibPocoFoundation9: V1.3.6 (or later) package installed
  - Device driver integrated into HAL API deb-package
- C/C++ development environment with dynamic library import
  - Free Microsoft™ Visual C++ Express 2012 and up supported
  - Linux: gcc 4.7.2 or later (5.x.x may work but was not tested)

1.5 HAL API setup

Please consult the separate "Ilmsens HAL API Setup Guide" for a description of the setup procedure. It also contains information on how to test a successful installation of the library.

1.6 HAL API function reference

This programming guide concentrates on HAL design and background information to help the developer building functional and efficient applications with the library. The contents of this guide will be valid for multiple revisions of the HAL API and is likely to change only with a new major release number. All functions mentioned throughout the document will be described in detail in the separate "Ilmsens HAL API Function Reference". The reference is updated and extended with every new revision and you will receive the corresponding version with your HAL API software.

2 Ilmsens HAL API architecture and usage

The basic view of the HAL on **m:explore** sensors is that of UWB measurement devices that feature similar (or identical) hardware properties and produce raw measured data in regular time intervals when a measurement run was started. It does not include any processing of the raw data (compare hints in chapter 3). The API is designed to support multiple devices measuring in parallel, but can also operate a single device, of course.

When multiple **m:explore** sensors are being used, they can either work independently (no physical synchronisation connection) or can work in a fully synchronous configuration, where the RF system clock is shared among all devices and a digital synchronisation connection is used. In the latter case, one of the sensors is declared as a master device while the remaining sensors are working as slaves. The master is responsible for triggering measurement runs and data counter resets for all slave modules, which ensures a fully synchronous and repeatable acquisition timing, i.e. data from each sensor is recorded at the same time. If independent sensors are used, all active modules must be declared as masters. If a measurement is started, the corresponding command is sent to the devices subsequently leading to a certain (usually small in the ms-range, but unknown) latency between data from different modules. Since the user is responsible for the hardware setup, the HAL API cannot know which mode - independent or synchronous operation - is being used. Consequently, the application software must take care of proper master/slave configuration and triggering of digital synchronisation.

The following sections introduce the abstract state machine of an **m:explore** sensor node as well as typical tasks an application may perform using the HAL API.

*2.1 State diagram of an **m:explore** sensor*

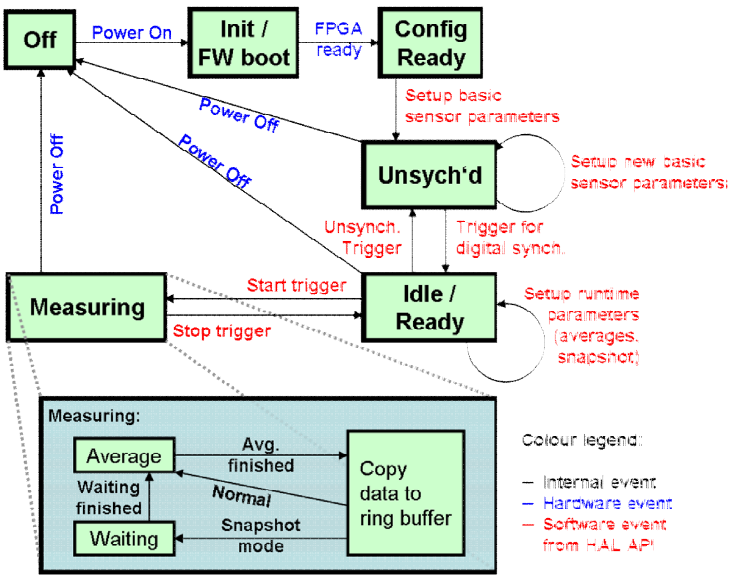

Fig. 2.1: Simplified state diagram of **m:explore** sensors

The state diagram in figure Fig. 2.1 shows the simplified processes in each sensor module. Some state transitions are caused by hardware (blue events), some are caused by internal timing (black events), and some are caused by the computer using the HAL API (red events). Correct parameter setup is required before a measurement can be started.

While a measurement is running, acquired data is stored in a device-internal ring buffer. The application is required to poll the buffer fill level and retrieve measured data, when one or more complete datasets are in the buffer. The HAL API provides corresponding functions. The application typically implements a state machine like in Fig. 2.2. If multiple sensors are measuring, the application may query and transfer data from the sensors subsequently. However, since the sensors produce data in fixed regular intervals, average retrieval speed of the application must be fast enough to avoid buffer overflows. The HAL API supports application timing in this regard by providing an (optional) threaded measurement mode where the data transfer to the computer is completely handled inside the HAL.

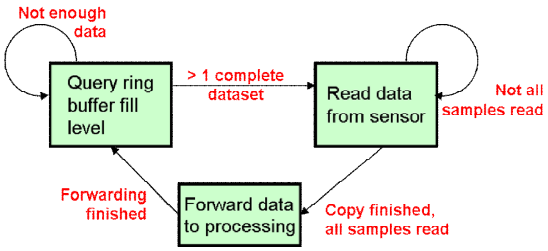

Fig. 2.2: Simplified data retrieval state machine for application

2.2 Basic application program flow for single measurements

The most basic application performing a single measurement with **m:explore** sensors must perform the following steps:

1. Initialise HAL API (this will detect sensors attached to the computer)
2. Open sensor connections to activate the modules
3. Setup basic parameters such as M-Sequence order and RF system clock rate
4. Setup Master/Slave mode for all modules
5. Trigger digital synchronisation
6. Reset M-sequence transmitters
7. Setup measurement parameters (such as number of averages and wait cycles)
8. Start a measurement
9. Retrieve measured data - repeat until enough data was read (compare Fig. 2.2)
10. Stop measurement
11. Close sensor connections to deactivate them in the HAL
12. de-initialise HAL API

2.3 How to perform typical tasks using the HAL API

The HAL provides functions to accomplish different tasks. These can be roughly grouped into the following topics

- **HAL API infrastructure**
  - Init/De-Init
  - Enumerate/detect sensors attached to the computer
  - Retrieve API information
  - Setup of debug level
- **Sensor device handling**
  - Open communication with sensor (activate)
  - Reading the unique sensor ID
  - Close communication with sensor (deactivate)
- **Setup, configuration, and query of sensor parameters**
  - Basic parameters (e.g. M-sequence order)
  - Query sensor configuration and status (e.g. device temperature)
  - Setup master/slave mode
- **Measurement management**
  - Digital synchronisation between multiple sensors
  - Control of sensor transmitter (e.g. power down)
  - Acquisition parameters (e.g. number of synchronous averages)
  - Start of measurement run
  - Stop of measurement run
- **Transfer of measured data to the user application**
  - Query data availability
  - Retrieve data from HAL
- **Memory-mapped direct access to the sensor(s)**
  - Register access
  - Memory access

2.3.1 HAL API infrastructure

When a user application starts or is reset, it must take care to load the HAL library into its context. Every development environment provides a different means to do so. After loading the library and before any HAL functions can be used, internal resources of the library must be prepared. This is done by a dedicated initialisation call. This first task is internally combined with sensor device enumeration, i.e. the library detects active sensors attached to the computer. The corresponding HAL function is:

- `ilmsens_hal_initHAL()`

The function allocates common HAL resources and returns the total number ***N*** of sensors that could be found during enumeration (or a negative error code). If that number is smaller than what was actually connected to the computer, check these issues:

- Power supply of sensors is on.
- USB connectors are safely attached and the cables are not broken.
- USB driver setup may be necessary in some OSs, if you connect a sensor to an USB port for the first time. When the installation process is not completely finished, the HAL will not detect such a sensor.
- Some computers use external USB ports, e.g. on an add-in card (PCI, PCMCIA, PCIe, etc.), or the sensor is attached via an USB hub. In some cases, enumeration of such sensors is not possible (especially if the USB bus power is not sufficient). You should try to use another USB port. Ports directly provided by the chipset of your computer tend to work well in most cases.

For most other HAL functions, a list of sensors that will be addressed by that function must be given. The list contains sensor numbers. The number of a specific sensor corresponds to the detection order during enumeration, i.e. the number can be ***1 .. N***. The user application never has to deal with interface handles or the like. While the HAL is initialised, the sensor number for a device does not change and sensors attached after the call to `ilm-sens_hal_initHAL()` will be ignored. However, when the HAL is closed (see below) and re-initialised it cannot be guaranteed, that the number for an individual sensor stays the same. Please also note that the order in which sensors are attached to the computer might or might not have influence on the enumeration process (which depends on the OS). Therefore, you should read out the sensor ID (see section 2.3.2) to identify a specific device every time you initialise the HAL.

At any time during HAL operation, you may query common information about the library. Especially, the specific version and build number of the library can be useful, if you are using different variants in your projects:

- `ilmsens_hal_getVersion(...)`

During usage of the HAL API, internal errors, warnings, and events are logged. The log messages can be useful for tracking down problems or bugs in the HAL or user application. You can influence the verbosity of logging by setting the debug level:

- `ilmsens_hal_setDEBLevel(...)`

It is recommended to use a high debug level during application development (which will degrade performance), but a reasonably low level during production use/real measurements.

By default, all HAL log messages are output to standard output, e.g. typically appearing at the console depending on your environment. You can redirect the standard output appropriately, if your application has no console to display the log messages.

When all operations with sensors are finished, the HAL API must be closed by calling the following function:

- `ilmsens_hal_deinitHAL()`

It releases all common HAL resources and closes still opened sensor connections. The enumeration order of sensors becomes invalid, too. The function will always succeed.

When the user application is closed, please ensure that the HAL API is closed before the library is unloaded. Otherwise, memory leaks or undefined behaviour may result. Especially if you detect any kind of error while using the HAL and shut down your application, you should still close the HAL correctly.

2.3.2 Sensor device handling

Enumerated sensors are identified by their ordering number when the HAL was initialised. Before a sensor can be used for any other operation, the communication connection must be opened, i.e. the sensor must be activated in the HAL

- `ilmsens_hal_openSensors(...)`

You can open multiple sensors at a time by listing their order numbers in the function call. The return value is either an error (negative number) or the success code. In case the sensors could not be opened successfully, it is ***not*** recommended to continue with any operation other than closing all sensors and closing the HAL.

If you are done using a sensor, call the following function to close the communication channel and deactivate the device in the HAL:

- `ilmsens_hal_closeSensors(...)`

Again, a list of sensor numbers must be provided.

To identify a specific sensor among all active devices, a unique ID string can be retrieved using the following function, which takes a single order number only:

- `ilmsens_hal_getModId(...)`

2.3.3 Setup, configuration, and query of sensor parameters

The **m:explore** sensors require to be setup with correct basic parameters like the order of the employed M-sequence and the HAL needs to know additional device properties (e.g. the actual RF system clock rate) to ensure proper internal buffer management and data transfer timing. The setup of these basic parameters is required when a sensor is powered on and activated in the HAL for the first time in a session, i.e. when a sensor is subsequently activated and deactivated several times without de-initialising the HAL, an initial configuration is enough. The required parameters can be obtained from the hardware manual of the device and setup can be done in a single API call:

- `ilmsens_hal_setupSensors(...)`

As mentioned at the beginning of this chapter, multiple devices should have the same or similar parameters when they are used in parallel. The function takes a list of sensor order numbers and allows setting up multiple devices with the same parameters in one call. The following basic parameters must be defined:

- RF system clock rate of the sensor f_0 [GHz]
- M-Sequence order (currently 9,12, or 15)
- Oversampling factor (defaults to 1)
- Subsampling factor (defaults to 512)
- Number of transmitters in the device (defaults to 1)
- Number of Receivers in the device (defaults to 2)

It is possible to indicate to the HAL, that the default values shall be used for some of the parameters. *However, a call to the setup function is mandatory to ensure correct functioning of the device and the HAL.*

When the user application changes the basic parameters, it is required to re-trigger digital synchronisation between the sensors (compare section 2.3.4 and Fig. 2.1).

After parameter setup, the application can query the sensors' properties at any time during a HAL session. In addition to the basic parameters, further information on the state and properties of the device are reported. The corresponding function is:

- `ilmsens_hal_getModInfo(...)`

Like in the case of retrieving the devices' ID string, this function only takes a single order number. The following parameters are reported:

- RF system clock rate of the sensor f_0 [GHz]
- M-Sequence order
- Oversampling factor

- Subsampling factor
- Number of transmitters in the device
- Number of Receivers in the device
- Device-internal temperature
- Number of averages and wait cycles, including their limits (compare section 2.3.4)
- Number of samples per impulse response (useful for data management)
- ADC full scale range and resolution (in Volts)

As mentioned in the introduction of this chapter, **m:explore** sensors can be set to master or slave mode depending on whether multiple devices are physically connected for synchronisation (exactly one device must be master, all others slaves) or are working independently (all devices must be masters). If there is only a single device used, it must be set to master mode. *The setup of master/slave mode must always be performed* just like the call to the sensor setup described above. The following function set the mode:

- `ilmsens_hal_setMaster(...)`

When the basic parameters and master mode are setup for all activated sensors, the application may trigger digital synchronisation, setup measurement-related parameters, and start or stop measurements as described in the following section.

2.3.4 Measurement management

Before a measurement can be started, it is required to perform digital synchronisation. This is regardless whether a single sensor is being used, or multiple sensors are employed (independent and connected setups). When any property of the previous section 2.3.3 changes, digital synchronisation must be revoked and re-triggered again. Both tasks can be performed using the following function:

- `ilmsens_hal_synchMS(...)`

The function takes a list of sensor order numbers which contains all devices that should be synchronised. As a second parameter, the user application can request to revoke or trigger digital synchronisation.

After digital synchronisation, the M-sequence generator (transmitter) of each device should be reset to ensure repeatable alignment of the transmitters and receivers. A call to the following function also ensures a proper start-up of the transmitter electronics:

- `ilmsens_hal_setMLBS(...)`

The reset typically takes a few ms to complete and cannot be used to mute the transmitter(s). There is a specialised function that activates or de-activates a power-down feature on the transmitters of the requested sensors:

- `ilmsens_hal_setPD(...)`

Power down can only be changed, when no measurement is running. Besides the list of sensor order numbers, the function takes a flag to decide the state of the power down feature. If the user application controls an **m:explore** sensor network, it can decide to activate different transmitters between measurements.

Another set of parameters that should be setup by the user application and that can be changed only in between measurement runs, regards measurement speed and acquisition aperture. **m:explore** sensors use synchronous averaging of repeatedly recorded signal periods in order to improve signal-to-noise ratio (SNR) as well as lower the amount of data to be transferred to the computer. Optionally, the user can insert wait cycles. This process is realised in the sensor hardware and is illustrated in Fig. 2.1. Details regarding averaging and wait cycles are described in section 3.1. Both values are set with this function:

- `ilmsens_hal_setAvg(...)`

Because the data transfer timing is influenced by these settings, the function can only be used when no measurement in running.

Finally, the user application can start (and subsequently stop) a measurement run. A measurement can be started either with all activated sensors or with a subset of them. When using only a subset, some restrictions apply:

- Independent sensors must be masters
- Connected sensors can only be operated, when the master device is included in the measurement run. The slave modules to be used must be included as well.
- Slave modules cannot be used without their master module

The functions for starting and stopping a measurement are as follows:

- `ilmsens_hal_measRun(...)`
- `ilmsens_hal_measStop(...)`

While a measurement is running, the application must regularly poll the HAL for new data and retrieve it when available (compare section 2.3.5). Each **m:explore** features a ring buffer for relaxing the data transfer timing. However, the capacity of the buffer is limited and latencies occurring in the computer (e.g. caused by OS or user application) may cause a buffer overflow. In such a case, some measured data will be lost. However, the measurement and application can continue to run. The HAL provides different measurement modes to help avoid such situations and allow flexible user application design. The function `ilmsens_hal_measRun(...)` takes a mode flag, which selects one of the following options:

- RUN_RAW: polling of buffer fill state and data transfer is fully under control of the user application, i.e. no background operations are performed by the HAL
- RUN_BUF: the HAL uses a separate internal thread to poll and drain the devices' ring buffers. Data is copied into an internal HAL ring buffer on the computer, which can be much larger and consequently accommodate longer latencies of the application.

Buffered mode is the preferred mode of operation for the HAL. However, in this case further communication of the application with the devices by using HAL functions is prohibited (except `ilmsens_hal_measStop()` or `ilmsens_hal_getModInfo()`), since the internal measurement thread needs exclusive access to the sensors. In raw mode, the user application has exclusive access. Furthermore, buffered mode requires more resources (additional CPU thread and memory) than raw mode.

2.3.5 Transfer of measured data to the user application

For flexible user application design, the HAL provides blocking and non-blocking functions for measurement data transfer. The blocking case is the most simple way to retrieve data. Simply call the following function:

- `ilmsens_hal_measGet(...)`

Again, this function takes a list of sensor order numbers and blocks, until one dataset has been received from ***every*** sensor in the list. Only sensors, which were included when the measurement was started, can be queried. Please note that the application can specify a timeout, i.e. the call blocks as long as there is no new data up to the given timeout. The return value identifies, if new data was retrieved during the call.

The non-blocking data transfer functions allow to query, how many datasets are available in the buffers using the following call:

- `ilmsens_hal_measRdy(...)`

The call returns immediately after querying the devices' ring buffers and allows the application to decide, when data is actually received from the HAL. It returns the number of complete datasets that is available (i.e. the minimum number of new datasets that was available from all measuring sensors). If new data is available, the user application can subsequently call the corresponding read function to copy one complete dataset:

- `ilmsens_hal_measRead(...)`

The format of returned measurement data will be explained in chapter 3. As mentioned before, the user application must retrieve data from the sensors on average at least as fast as data is produced by them. The intermediate buffers in the devices and/or the HAL only serve to accommodate latencies and relax the real-time requirements on the application.

2.3.6 Memory-mapped direct access to the sensor(s)

Most of the HAL functions are internally translated into sequences of register or memory accesses. The HAL API provides functions for direct access, but it is the responsibility of the user application to avoid actions that put the sensor in an unknown state or break the internal functioning of the HAL. For example, if you set the M-sequence order directly by writing the corresponding sensor register, the HAL does not know about it and internal buffer management is no longer consistent with the sensor's setup – data transfer is most likely to fail in such a case. Furthermore, register and memory locations as well as the associated functions may change with new sensor generations. Direct access is provided for testing, development, and most notably debugging of issues occurring during measurements. A register reference and memory map can only be obtained from Ilmsens upon special request by the customer for specific tasks coordinated with Ilmsens and usually requires an NDA. One of the big advantages of a HAL is precisely to hide such product-specific details from a user application. Applications using direct register/memory access are likely to be bound to a specific HAL version and sensor generation and may not work on older/newer versions or devices.

The memory map of the sensors can be accessed as single locations (registers) or as memory blocks. The address space is 32 bit wide and increments in bytes. However, the memory granularity is 4 byte, i.e. you cannot access addresses that are not dividable by 4 (the two LSBs of all addresses must be *'00'*). Please note that not all addresses are mapped to physical memory. The following functions are included for single register access:

- `ilmsens_hal_readReg(...)`
- `ilmsens_hal_writeReg(...)`

As with most other HAL functions, multiple sensors can be queried in one call. The following two functions provide access to memory blocks:

- `ilmsens_hal_readBlk(...)`
- `ilmsens_hal_writeBlk(...)`

3 Timing, data format, and basic processing of UWB data

3.1 Details on measurement timing

When planning a measurement application, different competing issues must be consolidated. Since the device uses a periodic UWB signal, acquired data can be averaged coherently to improve signal-to-noise ratio (SNR). A high averaging number results in higher SNR, but it also means that the acquisition aperture (the time for recording a complete signal period incl. averages) becomes longer and consequently measurement speed is reduced. Please note that this aperture relates to the maximum amount of change in your device under test/scenario under test that can be correctly represented in the measured data and a compromise between SNR improvement and measurement speed reduction must often be made. Depending on the measurement task, a very short acquisition aperture may be required. When the number of averages is reduced, the measurement rate increases, i.e. the data stream between the computer and sensor also increases and the application must be able to handle and/or process that amount of data sufficiently fast.

The provide the user with some flexibility regarding the trade-off between acquisition aperture and measurement speed, the **m:explore** sensors can be operated in a continuous or a snapshot mode. In both cases, the number of averages defines the duration of the acquisition aperture. In continuous mode (the default), repetition rate is as high as possible, i.e. after completing an averaging cycle, the next cycle is immediately started. In snapshot mode, wait cycles are added between averaging cycles. During a wait cycle, acquired data is discarded and no new output is copied to the ring buffer. This way, a short acquisition aperture (low number of averages) can be combined with a low measurement rate (high number of wait cycles) to give the application enough time for data handling and processing. The rough sensor-internal process is shown in the state machine of Fig. 2.1.

The HAL API allows setting the number of averages and wait cycles via the `ilmsens_hal_setAvg()` call (see section 2.3.4). Please note, that a minimum amount of averaging (so-called 'hardware averages', HW averages) is always done by the sensors and cannot be changed. The values set by the HAL influence acquisition and waiting duration as multiples of HW average duration. The number of HW averages (***HWAvg***) can be obtained using the `ilmsens_hal_getModInfo()` call (see section 2.3.3 above).

The following rhythm is performed inside the sensor while a measurement is running:

1. Acquire a single period with hardware averages (***HWAvg***)
    a. Record all samples of a signal period
    b. Average the new samples with previous data
    c. If current HW averages are less than ***HWAvg***, go to a. otherwise to d.
    d. Copy averaged data to intermediate buffer and reset average counter
    e. Trigger step 2., then go to a.

2. Perform software averages (***SWAvg***) (triggered by step 1.)
    a. Wait for trigger from step 1.
    b. If current software averages are less than ***SWAvg*** go to c. otherwise go to 3.
    c. Average data in intermediate buffer with previous data, then go to a.
3. Perform wait cycles (***WC***)
    a. Wait for trigger from step 1.
    b. If current wait cycles are less than ***WC*** go to a. otherwise to step 4.
4. Reset software average and wait cycle counters go to step 1.

The following equations provide the relation between the systems parameters (M-sequence order *m*, RF system clock rate f_0, sampling clock pre-scaler *D*, *HWAvg*, *SWAvg*, *WC*, which can all be obtained via calling `ilmsens_hal_getModInfo(...)`):

$$N_{Samp} = 2^m - 1 \quad (3.1)$$

$$T_{IRF} = 1/f_0 \cdot D \cdot N_{Samp} \quad (3.2)$$

$$T_{HW\text{-}Avg} = (HWAvg + 1) \cdot T_{IRF} \quad (3.3)$$

$$T_{AP} = T_{SW\text{-}Avg} = SWAvg \cdot T_{HW\text{-}Avg} \quad (3.4)$$

$$T_{Wait} = WC \cdot T_{HW\text{-}Avg} \quad (3.5)$$

$$T_{Tot} = T_{SW\text{-}Avg} + T_{Wait} \quad (3.6)$$

$$f_{Meas} = 1 / T_{Tot} \quad (3.7)$$

with number of samples per period N_{Samp}, measurement aperture of a single period T_{IRF}, time interval for a single hardware averaging cycle $T_{HW\text{-}Avg}$, acquisition aperture T_{AP} (equals the time for completing software averages $T_{SW\text{-}Avg}$), time interval for wait cycles T_{Wait} (can be 0!), and finally total cycle time T_{Tot} or its inverse - the measurement rate f_{Meas}.

If wait cycles are set to 0 (the default), all hardware averaged data is processed and contributes to the final values sent to the computer after T_{Tot} - i.e the whole cycle time is used for averaging. When the number of software averages is low, a high measurement rate f_{Meas} results. Setting ***WC*** to a non-zero value (this is snapshot mode) and choosing a low value for ***SWAvg*** allows combining a short acquisition aperture T_{AP} with a low measurement rate f_{Meas}.

3.2 Format of measured data and required buffer size

The data from each sensor is read into an internal buffer and then copied to the user application buffer by the HAL functions `ilmsens_hal_measGet(...)` or `ilmsens_hal_measRead(...)` (see section 2.3.5). It is returned as an array of 32 bit integer values (data type ***int32_t***). The total size of the buffer *NumEl* can be calculated from the basic parameters of all sensors included in the call:

$$NumEl = \sum_{NumSen}^{i=1} Channelsize(i) \cdot NumberOfRx(i) = \sum_{NumSen}^{i=1} 2^{mOrder(i)} \cdot mOV(i) \cdot mRx(i) \quad (3.8)$$

where *NumSen* is the number of sensors in the order number list given to the data retrieval function calls (compare section 2.3.5) and the other parameters are obtained from the sensor information call `ilmsens_hal_getModInfo(...)` (see section 2.3.3 above). If all sensors have the same setup, eq. *(3.8)* reduces to:

$$NumEl = 2^{mOrder(i)} \cdot mOV(i) \cdot mRx(i) \cdot NumSen \qquad (3.9)$$

In any case, the application must make sure, that the buffer provided to the HAL is large enough to hold all returned values. Due to the properties of the UWB signal used in **m:explore** sensors (an M-sequence) the actual data vector per RX channel has only $(2^{mOrder}-1) \cdot mOV$ entries, e.g. a 9th order example sensor with the default oversampling of 1 delivers 511 samples per channel. The remaining entries in the return buffer (i.e. 1 additional ***int32_t*** per Rx channel of the sensor) are filled with status information regarding for the dataset.

Currently, only the first additional value for Rx 1 of each sensor in the result buffer (found at offset `buffer[(2^mOrder(1)·mOV(1))-1]` for the first sensor in the order number list, at `buffer[(2^mOrder(1)·mOV(1)·mRx(1))+(2^mOrder(2)·mOV(2))-1]` for the second one, etc.) is the sequence counter. It is reset to 0 at the start of a measurement and increased with every new measurement realisation (after the complete averaging and waiting process). The user application should check this value to detect lost datasets. For example, of the computer is too slow or too busy with other tasks, the sensors may not be read out fast enough. In such a case, the sequence counter will not be consecutive. Furthermore, when the multiple sensors are physically connected and hardware synchronised, the sequence counters of all sensors should be the same for every new dataset retrieved by the user application. Fig. 3.1 depicts a memory map for an example setup with two 9th order **m:explore** devices included in the measurement run.

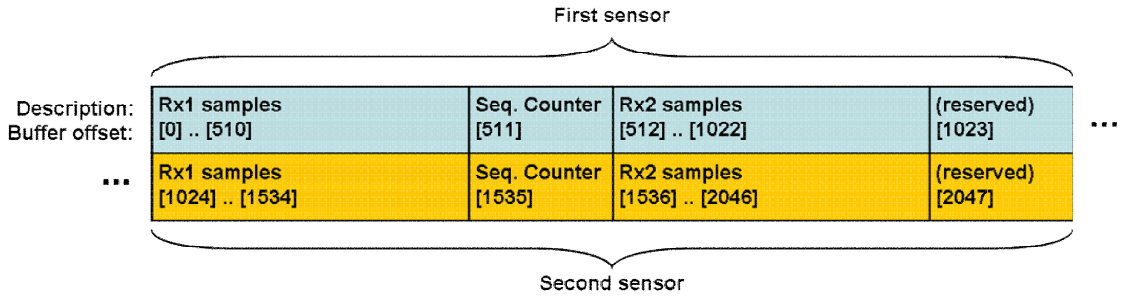

Fig. 3.1: Memory map of example setup with two measuring 9th order **m:explore** sensors

*3.3 Typical first processing steps for **m:explore** UWB data*

The HAL API does not do any calculations with the measured data. For example, if you want to obtain impulse responses from the raw samples values x_{raw}, you have to calculate proportional scaling and the correlation with the ideal M-sequence in the user application. These steps will be shortly explained here.

Sample data is delivered as 32 bit integers instead of floating point values to retain maximum quantisation accuracy as delivered by the ADCs in the sensors. For the same reason, the averaging process in the devices (compare section 3.1) actually only accumulates sample values x_{acc} according to eq. *(3.11)*, i.e. it does ***not*** perform the normalisation of eq. *(3.12)* by the number of total averages p to obtain the final averaged sample x_{avg}:

$$p = \boldsymbol{HWAvg} \cdot \boldsymbol{SWAvg} \qquad (3.10)$$

$$x_{raw} = x_{acc} = \sum_{p}^{i=1} x(i) \qquad (3.11)$$

$$x_{avg} = x_{acc} / p = \frac{1}{p} \sum_{p}^{i=1} x(i) \qquad (3.12)$$

Therefore, the first typical processing step is the normalisation of each sample by the number of total averages p, which can be obtained by eq. *(3.10)*. In order to avoid loss of accuracy and increase of quantisation noise, it is recommended that the application converts the sample values into double precision floating point format (**double**) before applying the division. This step must be applied to each sample in the signal period for each receive channel.

If your application requires the measured data to be expressed in physical units, the normalised data can be converted to voltages by using the LSB voltage of the ADCs in the sensors. This value can be obtained from the `ilmsens_hal_getModInfo(...)` call (see section 2.3.3 above). The application can use eq. *(3.13)* to do the conversion:

$$x_{volts} = x_{avg} \cdot mLSB\_Volt \qquad (3.13)$$

Of course, this step must be applied to each sample individually and can easily be combined with the normalisation of eq. *(3.12)*. However, if only relative amplitudes are required, the conversion to volts can be skipped.

Since **m:explore** sensors are using Ilmsens' unique M-sequence technology, measured data must be cross-correlated with the ideal M-sequence M_{ideal} in order to obtain the actual impulse response of the device/scenario under test. Please note, that all signals involved are periodic signals, i.e. the correlation must be performed as a cyclic correlation as defined by eq. *(3.14)*:

$$irf(\tau) = \oint_{T_{IRF}} x_{volts}(t) \cdot M_{ideal}(t+\tau) dt \qquad (3.14)$$

The ideal M-sequence M_{ideal} is a binary signal with the only values +1 and -1. It is not DC-free, since the number of samples N_{Samp} is odd (compare eq. *(3.1)*). The sequence matching your sensor is delivered as a text file (e.g. `'mlbs_9.txt'` for a 9th order device) in the software package as a reference for your application. Please note, that for most M-sequence orders, more than one generating polynomial exists, i.e. you must use the one implemented in the devices' transmitter.

A fast alternative to directly calculating the integral of eq. *(3.14)* is using the Fast Fourier Transform (FFT). An even faster option is the Fast Hadamard Transform (FHT). Very similar to the FFT, this transform can benefit from the butterfly algorithm and can be implemented using only additions and subtractions instead of complex multiplications (required for FFT). Details on the relation between M-sequence correlation and Hadamard Transform can be found in the literature, e.g.:

> M. Cohn and A. Lempel: "On fast m-sequence transforms", IEEE Transactions on Information Theory, vol. 23, no. 1, pp. 135-137, 1977

Further processing largely depends on the customers' measurement task and measurement setup. For example, if you are interested in the frequency response function rather than the impulse response function for your task, FFT can be used at any time to go to the frequency domain. Like with any other microwave measurement technology, the result of Fourier Transform only represent the real frequency content of the received analogue signal, if all stages in the receiving chain (e.g. LNAs, UWB-receivers in the **m:explore**, ADC analogue input) work in their linear region - receiver saturation or signal compression must be avoided.

Ilmsens has rich experience with a versatile pool of different UWB applications - including but not limited to: UWB ranging, localisation, through-wall radar, remote monitoring of vitality data (breathing/heartbeat), non-destructive testing of building materials, impedance spectroscopy of liquid and solid materials, etc. If you need assistance with data processing and handling for your application, Ilmsens provides consulting services. Please contact info@ilmsens.com for the options offered.

4 Further resources and revision history

4.1 Further resources

For further information please visit our website at www.ilmsens.com.

The following documents provide additional information for development:

- Measurement hardware and device drivers: "Ilmsens Hardware Manual m:explore" (provided with your **m:explore** Evaluation Kit)
- Drivers and Ilmsens application software: "Ilmsens Software Manual m:explore"
- HAL API setup info: "Ilmsens HAL API Setup Guide"
- HAL API function reference: "Ilmsens HAL API Function Reference"

If you need assistance with the HAL API feel free to contact us at:

Ilmsens GmbH
Ehrenbergstraße 11
98693 Ilmenau
Germany

Tel.: +49 3677 76130-30
Fax: +49 3677 76130-39
Email: hal-api@ilmsens.com

4.2 Document revision history

Rev.	Date	Author	Description
1.0	10/2016	Her	Initial revision for the **m:explore** HAL API manual
1.1	01/2017	Her	Major API improvements, update of supported platforms, added explanations for buffers/data format, minor corrections
1.2	01/2017	Her	Split the doc in programming guide (this) and separate installation guide as well as separate function reference guide produced by DoxyGen (PDF-version, HTML-version, etc.)