SMART ACCESS AUTOMOTIVE USER MANUAL

DATE:
January 22, 2024

LIBRARY VERSION:
TODO

s.m.s, smart microwave sensors GmbH
In den Waashainen 1
38108 Braunschweig
Germany

Phone: +49 531 39023-0
Fax: +49 531 39023-599
info@smartmicro.de
www.smartmicro.com

CONTENT

1 INTRODUCTION

Smart Access is a smartmicro software product that enables to get control over connected smartmicro devices. In the following, the integration and usage of Smart Access is described, including the description and examples of Advanced Programmers Interfaces (APIs), generated for the provided user interface.

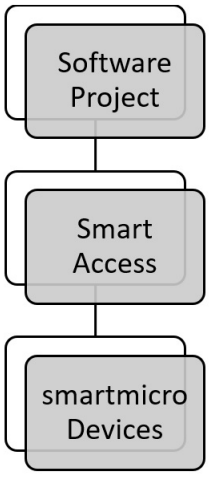

Figure 1: Smart Access context diagram

2 INTEGRATION

This chapter explains how to integrate Smart Access and describes additional tasks that need to be performed. Below, the software components for the integration of Smart Access are illustrated in a layered approach.

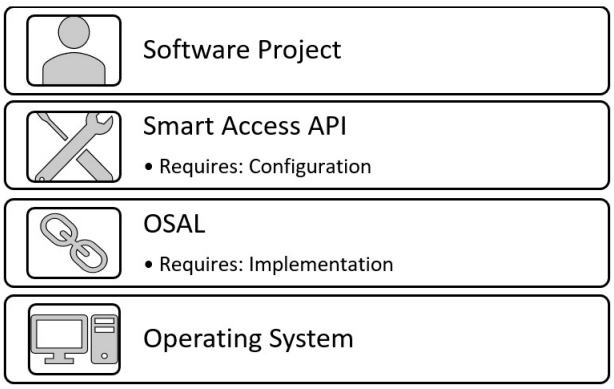

Figure 2: Smart Access diagram

2.1 OPERATING SYSTEM ADAPTION LAYER (OSAL)

smartmicro products are typically connected to CAN bus, Ethernet or RS485 interfaces. How to access those interfaces depends on the operating system and the hardware driver interfaces. Smart Access requires an Operating System Adaption Layer (OSAL) library, which realizes the OSAL interface for the individual target platform. There are example implementations of OSAL for the operating systems Windows and Linux, which can serve as a basis for customer-specific implementations after certain reconfiguration and adjustment efforts.

2.2 APPLICATION PROGRAMMING INTERFACE (API)

The Application Programming Interface (API) written in C++ provides features divided into communication services.

3 DEPLOYMENT

3.1 PREREQUSITES

For the deployment of Smart Access, a C++11 compatible compiler and experimental file system feature are required.

3.2 DELIVERY PACKAGE

The Smart Access product contains the following items:

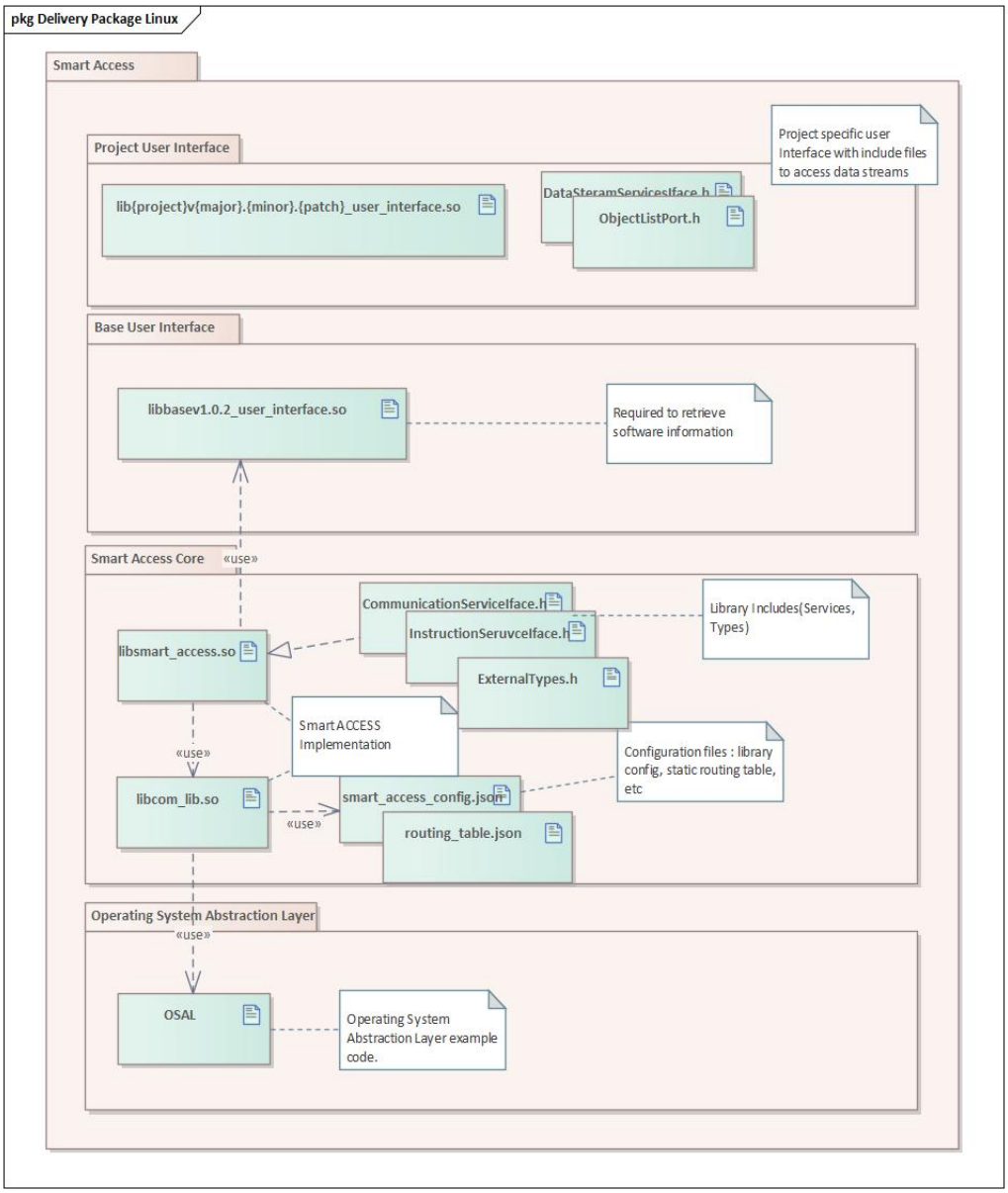

Figure 3: Deployment Diagram

3.3 Supported platforms

The Smart Access product supports the following platforms:

os	architecture	compiler
windows	x86_64	msvc 16
windows	x86_64	msvc 16 v14.1
linux	x86_64	gcc 6.5
linux	x86_64	gcc 7.5
linux	x86_64	gcc 9.3
linux	x86_64	gcc 12.3
linux	x86	gcc 6.5
linux	x86	gcc 7.5
linux	x86	gcc 9.3
linux	armv7hf	gcc 6.5
linux	armv7hf	gcc 7.5
linux	armv7hf	gcc 7.3 musl
linux	armv7hf	gcc 9.3
linux	armv7hf	gcc 12.3
linux	armv8	gcc 6.5
linux	armv8	gcc 7.5
linux	armv8	gcc 9.3
linux	armv8	gcc 12.3

Furthermore, a minimum version of libstdc++ is required under linux. The version of libstdc++ depends on gcc compiler version and can be found under https://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html.

4 GETTING STARTED

4.1 Content of Release Package

The release package contains the following content:

- **config** - Example configuration files for hardware interfaces, routing table and Smart Access configuration file.
- **doc** - User Manual and documentation of the user interfaces.
- **example** - Contains example app sources and build scripts.
- **include** - Includes for compilation.
- **lib-<os>-<architecture>-<compiler>** - Libraries of Smart Access for the given os, architecture and compiler version.
- **user_interfaces** - For each user interface there is a subfolder that contains information about instructions and data streams. Smart Access does not read these files at runtime, because the user interface information is already included in user interface lib.

4.2 How to build and run the app

4.2.1 Linux

To build the software the following software is required:

- **compiler** - For compiling and linking example APP/OSAL with Smart Access lib.
- **cmake** - Build tool.

4.2.1.1 How to Build

Go to the example directory and execute the following command.

```
./build.sh --build <x86 | x86_64 | armv7hf | armv8>
```

The build.sh will compile the example APP and link against Smart Access library.

4.2.1.2 How to Run

1. Export Environment variable : export SMART_ACCESS_CFG_FILE_PATH=<absolute path of smart_access_config.json>
2. Correctly specify all the paths in the smart_access_config.json (example in config directory)
3. Make sure the LD_LIBRARY_PATH is adapted or add configuration in the /etc/ld.so.conf.d/ and subsequently run ldconfig.
4. Steps to create configuration file in /etc/ld.so.conf.d/ for library loading before exampleAPP is run
   - sudo touch /etc/ld.so.conf.d/<anyname>.conf
   - edit it to specify the complete path of your lib directory
   - sudo ldconfig
5. Execute the Example app
   - For using the ldconfig of above step, this needs to be run with sudo. Which can be avoided if LD_LIBRARY_PATH is directly set instead of ldconfig.
   - For running the app with sudo, execute example_app also by specifiying the clientId (int value) of the slave as command line argument ./out/bin/example_app <ClientId>

4.2.2 Windows

To build the software the following software is required:

- **Visual Studio Build Tools 2019** - For compiling and linking example APP/OSAL with Smart Access lib.

  - C++ BuildTools
  - Windows 10 SDK
- **cmake** - Build tool.

4.2.2.1 How to Build

1. Open the command prompt
2. Activate your Visual Studio Developer Command for x64
3. Go to the example folder
4. Execute the build script: ”build.bat”

4.2.2.2 How to Run

1. Export Environment variable: set SMART_ACCESS_CFG_FILE_PATH=<absolute path of smart_access_config.json>
2. Correctly specify all the paths in the smart_access_config.json (example in config directory)
3. Execute example_app also by specifying the clientId (int value) of the slave as command line argument: ”example_app <ClientId>”

5 CONFIGURATION

5.1 CLIENT ID

Each network element needs a unique identification number in order to use smartmicro’s communication protocols. This number is called client ID and has the following format:

Client ID = 0xTTXXXXXX, where TT depends on the type.

Please use a type in the range of 192 to 255 (0xC0-0xFF) for the Smart Access configuration. The sensor client ID can not be changed. For a UMRR Sensor the type value is 00. The second part is the serial number of the sensor. For example:

Client ID 0x00066723 : Client type 0 (UMRR) with serial 66723

5.2 SERIALIZATION

Before the data is sent from one client to another, it needs to be serialized, depending on the protocol type being used and the underlying physical channel. Upon receiving, the data needs to be serialized to its previous form. There are two types of serialization:

- **can_based** - Data will be split into CAN messages.
- **port_based** - Port data will be sent unchanged.

5.3 SMART ACCESS CONFIGURATION

The location of the main JSON configuration file needs to be set in the ”SMART_ACCESS_CFG_FILE_PATH” environment variable.

You have to adapt at least these keys to your system:

- **shared_lib_path** - Absolute path to Smart Access libraries, e.g. ”/smart_access/lib-linux-x86_64-gcc_7”.
- **config_path** - Absolute path to the folder, which contains the configuration files.
  - smart_access_config.json

  - hw_inventory.json
  - routing_table.json
- **instruction_serialization_type** - Serialization type of the instruction service. This value will be used as default when the type is not specified in static route and cannot be determined via interview protocol. Allowed Values:
  - "can_based"
  - "port_based"
- **data_serialization_type** - Serialization type of the data stream service. This value will be used as default when the type is not specified in static route and cannot be determined via interview protocol. Allowed Values:
  - "can_based"
  - "port_based"
- **alive (optional)** - Activation of ALIVE protocol, can be either true or false. This argument is optional and the default value is true. If automotive radars are connected, the value must be set to false.

The following keys can be left untouched in the most cases:

- **name** - For informative purposes only.
- **version** - Data format version of this file.
- **client_id** - The client id of Smart Access, which needs to be unique. Recommendation: "0x1000001".
- **role** - Defined as "master" or "slave". Usually, a management system is determined as master and smartmicro sensors are determined as slaves.
- **download_path** - Path to the download folder, which is only relevant for slave devices.
- **user_interface_name** - Name of the user interface. The name **must** be set to "base".
- **user_interface_major_v** - Major version of the user interface. The value **must** be set to 1.
- **user_interface_minor_v** - Minor version of the user interface. The value **must** be set to 0.
- **user_interface_patch_v** - Patch version of the user interface. The value **must** be set to 2.
- **time_sync_cfg (optional)** - Configuration of the time sync service.
- **link_type (optional, required for time sync)** - Channel type: "can", "eth" or "rs485".
- **dev_id (optional, required for time sync)** - dev_id is the ethernet interface adapter as declared in hw_inventory.
- **role (optional, required for time sync)** - Defined the time sync service as "master" or "slave".
- **req_num_of_iter (optional, required for time sync as slave)** - The required number of iterations to synchronize the timebase
- **max_num_of_iter (optional, required for time sync as master)** - Maximum number of iteration to synchronize the timebase. It must be greater than the required number of iterations from the sensor. The maximum value is 255.

Example:

```
{
    "name": "Smart_Access_Config",
    "version": "1.0.0",
    "client_id": "0x1000001",
    "role": "master",
    "shared_lib_path": "/usr/lib",
    "config_path": "/data/config",
    "user_interface_name": "base",
    "user_interface_major_v": 1,
    "user_interface_minor_v": 0,
    "user_interface_patch_v": 2,
    "download_path": "",
    "instruction_serialization_type": "can_based",
    "data_serialization_type": "can_based",
    "alive": false,
    "time_sync_cfg": {
        "link_type": "eth",
        "dev_id": 1,
        "role": "master",
        "max_num_of_iter": 50
    }
}
```

5.4 ROUTING TABLE

The routing concept of the Smart Access library is based on the client IDs. The devices are connected to each other via different physical interfaces (CAN, RS485 or Ethernet). They can be recognized dynamically using the ALIVE protocol or routes can be added statically. A client has to be reachable by another client via one physical interface only, redundant channels are not allowed. The routing table is defined in "routing_table.json" and has the following keys:

- **name** - For informative purposes only.
- **version** - Data format version of this file.
- **clients** - List of client's routes, containing one channel type each (CAN, RS485, or Ethernet). In the following, the configuration of all channel types are illustrated with an example below.

5.4.1 DYNAMIC ROUTING

For dynamic routes, the client's array in the "routing_table.json" file remains empty.

```
{
    "name": "Client Routing Table",
    "version" : "1.0.0",
    "clients" :
    [
    ]
}
```

The client_id will be the serial number of the sensor that is detected.

5.4.2 STATIC ROUTING

Static routes need to be added to the clients list and have the following:

- **link_type** - Channel type, "can", "eth" or "rs485".
- **client_id** - Sensor client identifier with 32bit, which needs to be unique over the complete system. In dynamic route, the alive service will use the sensor serial number for that.
- **user_interface_name (optional)** - Name of the sensor user interface.
- **user_interface_major_v (optional)** - Major version of the sensor user interface.
- **user_interface_minor_v (optional)** - Minor version of the sensor user interface.
- **user_interface_patch_v (optional)** - Patch version of the sensor user interface.
- **instruction_serialization_type** - The client's serialization type of the instruction service, which can be either CAN-based (using "can_based") or port-based (using "port_based"). This field is optional for traffic sensors and mandatory for automotive radars.
- **data_serialization_type**- The client's serialization type of the data stream service, which can be either CAN-based (using "can_based") or port-based (using "port_based"). This field is optional for traffic sensors and mandatory for automotive radars.

5.4.2.1 CAN bus Entry :

For connections with a CAN bus, the following keys are needed additionally:

- **can_network_id** - If several clients are using the same CAN bus, each client additionally needs a unique identifier in the range of 0 to 14. Use only the value 0. At the moment it is an experimental feature.
- **dev_id** - ID of the CAN device required by the OSAL layer for identifying the adapter to be used.

The keys **instruction_serialization_type** and **data_serialization_type** are not needed for CAN.

```
    "clients" :
```

```
    [
        {
            "link_type": "can",
            "can_network_id": 0,
            "dev_id": 1,
            "client_id" : "0x01",
            "user_interface_name": "tm",
            "user_interface_major_v": 7,
            "user_interface_minor_v": 0,
            "user_interface_patch_v": 1
        },
        ......
    ]
```

5.4.2.2 RS485 Entry :

For connections with a RS485 4wire connection, the following keys are needed additionally:

- **dev_id** - ID of the RS485 device required by the OSAL layer for identifying the serial device to be used.

```
    "clients" :
    [
        {
            "link_type": "rs485",
            "client_id" : "0x02" ,
            "dev_id": 1,
            "instruction_serialization_type" : "can_based",
            "data_serialization_type" : "can_based",
            "user_interface_name": "tm",
            "user_interface_major_v": 7,
            "user_interface_minor_v": 0,
            "user_interface_patch_v": 1
        },
        ......
    ]
```

5.4.2.3 Ethernet Entry :

For connections via Ethernet, the following keys are needed additionally:

- **ip** - The client's IP address (IPv4 only).
- **port** - The client's UDP port number.

Exemplary Ethernet configuration:

```
    "clients" :
    [
        {
            "client_id" : "0x03" ,
            "link_type": "eth",
            "ip": "192.168.11.11",
            "port": 55555,
            "instruction_serialization_type" : "port_based",
            "data_serialization_type" : "port_based",
            "user_interface_name": "tm",
            "user_interface_major_v": 7,
            "user_interface_minor_v": 0,
            "user_interface_patch_v": 1
        },
        ......
    ]
```

6 COMMUNICATION SERVICES

To initialize the communication services, please us the Init() function as follows:

```
bool initResult = com::master::CommunicationServicesIface::Get()->Init();
```

6.1 INSTRUCTION SERVICE

The instruction service allows for the user to configure smartmicro devices, to retrieve their status and to execute certain commands on the connected devices. Every instruction is identified by a name which is unique within the given user interface and consists of the pair section name and the instruction name. It contains a 32bit value (”int8_t”, ”uint8_t”, ”int16_t”, ”uint16_t”, ”int32_t”, ”uint32_t”, or ”float32_t”).

There are three types of instructions:

- Parameters represent the device configuration and can either be set or retrieved.
- Statuses describe the current state of the device.
- Commands execute certain functions on the device.

In those cases, where a configuration requires multiple instructions simultaneously, the instructions need to be grouped in batches. An instruction sent to the device is called ”request” and should be answered with a ”response”. An instruction batch will be checked for correctness upon sending. If the check fails, the sending will be refused and an error message will appear within the log file. This service has some limitations:

- Maximum number of instruction for can based serialization is 10.
- Maximum number of instruction for port based serialization is 255.
- A instruction batch can not contain multiple instructions with the same section name, instruction name, dimension 1 and dimension 2.

To send an instruction batch to one or several clients, please carry out the following steps:

1. Get instruction service.

```
std::shared_ptr<com::master::InstructionServiceIface> instServ =
                    com::master::CommunicationServicesIface::Get()-> GetInstructionService()
                        ;
```

2. Allocate an instruction batch.

```
std::shared_ptr<com::master::InstructionBatch> batch;
bool status = instServ->AllocateInstructionBatch(
                        123 /*sensor client id*/
                        batch);
```

3. Create a ”set parameter” request.

```
std::shared_ptr<SetParamRequest<uint8_t>> param1(
                    new SetParamRequest<uint8_t>(
                        "some_section"/*section name*/,
                        "some_parameter"/*parameter name*/,
                        1/*value*/,
                        0/*dimension 1 */,
                        0/*dimension 2*/));
```

4. Create a ”get parameter” request.

```
std::shared_ptr<GetParamRequest<uint8_t>> param2(
                    new GetParamRequest<uint8_t>(
```

```
                                "some_section"/*section name*/,
                                "some_parameter"/*parameter name*/,
                                0/*dimension 1 */,
                                0/*dimension 2*/));
```

5. Create a "get status" request.

```
    std::shared_ptr<GetStatusRequest<float>> status(
                            new GetStatusRequest<float>(
                                "some_section"/*section name*/,
                                "some_status"/*status name*/));
```

6. Create a "command" request.

```
    std::shared_ptr<CmdRequest> cmd(
                            new CmdRequest(
                                "some_section"/*section name*/,
                                "some_command"/*cmd name*/,
                                1/*value*/));
```

7. Add instructions to the batch.

```
        if(!batch->AddRequest(param1))
        {
            std::cout << "Failed to add parameter instruction param1 to batch"
                      << std::endl;
        }
        if(!batch->AddRequest(param2))
        {
            std::cout << "Failed to add parameter instruction param2 to batch"
                      << std::endl;
        }
        if(!batch->AddRequest(status))
        {
            std::cout << "Failed to add status request to batch"
                      << std::endl;
        }
        if(!batch->AddRequest(cmd))
        {
            std::cout << "Failed to add command instruction to batch"
                      << std::endl;
        }
        //Hint: If you use instructions which don't match with User-Interface
        //      you will get related error log here.
```

8. Send an instruction batch. In order to receive a response on the request, a callback needs to be created.

```
        com::master::ResponseCallback callback = [&](ClientId clientId,std::shared_ptr<
            ResponseBatch> response)
        {
            // It could be responses with the same key but with different dimensions
            std::vector<std::shared_ptr<Response<uint8_t>>> myResp;
            if(response->GetResponse<uint8_t>("some_section", "some_parameter", myResp))
            {
                for(auto &resp : myResp)
                {
                    uint16_t index1 = resp->GetFirstDimIdx();
                    uint16_t index2 = resp->GetSecondDimIdx();
                    uint8_t value  = resp->GetValue();
                    std::cout << "Value[" << index1 << "][" << index2 << "] = " << value
                              << std::endl;
                }
            }

            std::vector<std::shared_ptr<Response<uint32_t>>> statuses;
```

```
    if(response->GetResponse<uint32_t>("some_section", "some_status", statuses))
    {
        for(auto &resp : statuses)
        {
            uint8_t value  = resp->GetValue();
            std::cout << "Value = " << value << std::endl;
        }
    }
};

or

void MyFunc(IN ClientId clientId, IN std::shared_ptr<ResponseBatch> response)
{
   //my logic
}
...
com::master::ResponseCallback callback = std::bind(&MyFunc,
                                                   std::placeholders::_1,
                                                   std::placeholders::_2);
```

9. Send the instructions to the clients.

```
if(ERROR_CODE_OK != instServ->SendInstructionBatch(batch, callback))
{
   std::cout << "Error - sending of InstructionBatch failed"
             << std::endl;
}
```

10. Upon receiving a response from the device, the registered callback pointing to the response batch will be invoked. To extract the response for a certain instruction, please use the following call:

```
std::vector<std::shared_ptr<Response<uint8_t>>> myResp;
if(!response->GetResponse<uint8_t>("your_section_name","your_instruction_name",myResp))
{
   std::cout << "Response section [your_section_name] inst [our_instruction_nam] does not
       exist" << std::endl;
}
```

11. To check if a request was successful, please call the following function:

```
respType = myResp->GetResponseType();
```

These are the possible responses:

```
RESPONSE_NO_RESPONSE = 0U,          // No instruction Response
RESPONSE_SUCCESS = 1U,              // Instruction Response was processed successfully
RESPONSE_ERROR =  2U,               // General error
RESPONSE_ERROR_REQUEST = 3U,        // Invalid request
RESPONSE_ERROR_SECTION = 4U,        // Invalid section
RESPONSE_ERROR_ID = 5U,             // Invalid id
RESPONSE_ERROR_PROT = 6U,           // Invalid protection
RESPONSE_ERROR_MIN = 7U,            // Value out of minimal bounds
RESPONSE_ERROR_MAX = 8U,            // Value out of maximal bounds
RESPONSE_ERROR_NAN = 9U,            // Value is not a number
RESPONSE_ERROR_TYPE = 10U,          // Type of Instruction is not valid
RESPONSE_ERROR_DIM = 11U,           // Dim of Instruction is not valid
RESPONSE_ERROR_ELEMENT = 12U,       // Element of Instruction is not valid
RESPONSE_ERROR_SIGNATURE = 13U,     // Signature of Instruction is not valid
RESPONSE_ERROR_ACCESS_LVL = 14U     // Access level is not valid
```

12. Release allocated InstructionBatch.

```
if(ERROR_CODE_OK != instServ->ReleaseInstructionBatch(batch)) {
   std::cout << "Error: Could not release allocated batch" std::endl;
}
```

6.2 TIME SYNCHRONIZATION SERVICE

The time synchronization service allows to synchronize the radar clocks with the master clock by probing periodically the local time of the slave and adjust it, if there is a time drift.
The time synchronization service has the following roles(Configured in main configuration file):

- Master - serves as a clock source.
- Slave - synchronized by the master.

Please notice, that Smart Access can serve as both master and slave, but not simultaneously.

6.2.1 TIME SYNCHRONIZATION MASTER

As the first step a callback,which retrieves the reference clock, shall be implemented.

```
bool MyGetRefTimeCallback(uint64_t& time)
{
    // your logic where time of uin64_t type shall be set in milliseconds
}
```

To get time synchronization master service please use the following code lines.

```
auto comServ = CommunicationServicesIface::Get();
auto timeSyncMaster = comServ->GetTimeSyncServiceIface()->GetMasterService();
```

Now the described above callback shall be registered within the service.

```
if(ERROR_CODE_OK != timeSyncMaster->RegisterRefTimeCallback(
                    std::bind(&MyGetRefTimeCallback,std::placeholders::_1)))
{
    std::cout << "Failed to register callback" << std::endl;
}
```

Now we need to add the slaves which shall participate at time synchronization. Each slave has to be added on master side using his slave id in related API call. This id must match with the slave id which was configured on slave side for this slave.

```
// e.g add slave with uint8_t id = 0
uint8_t id = 0;
if(ERROR_CODE_OK != timeSyncMaster->AddSlave(id))
{
    std::cout << "Failed to add slave id = " << id << std::endl;
}
```

Additional remarks regarding the slave id used in API-Function 'AddSlave':

- This slave id is only used in the time synchronization service and has no dependency to the client id of the related slave.
- Each slave which was added to the time synchronization service must own an unique slave id. To configure multiple slaves with the same slave id causes errors in the time synchronization.

To start the services please call the following function.

```
if(ERROR_CODE_OK != timeSyncMaster->Start())
{
    std::cout << "Failed to start time sync master service" << std::endl;
}
```

To stop the services please call as follows

```
if(ERROR_CODE_OK != timeSyncMaster->Stop())
{
    std::cout << "Failed to stop time sync master service" << std::endl;
}
```

If a slave shall not be more synchronized , it could be removed from the list of the slaves.

```
// e.g remove slave with uint8_t id = 0
uint8_t id = 0;
if(ERROR_CODE_OK != timeSyncMaster->RemoveSlave(id))
{
    std::cout << "Failed to remove slave id = " << id << std::endl;
}
```

6.2.2 TIME SYNCHRONIZATION SLAVE

As the first step a callback,which retrieves the local clock, shall be implemented.

```
bool MyGetLocalTimeCallback(uint64_t& time)
{
    //your logic where time of uin64_t type shall be set in milliseconds
}
```

The slave service requires also a callback, which sets the offset to the master clock

```
bool MySetOffsetCallback(int64_t offsetTime)
{
    //your logic to adjust slaves local clock, where the offset to master is in milliseconds
}
```

To get time synchronization slave service please use the following code lines.

```
auto comServ = CommunicationServicesIface::Get();
auto timeSyncSlave = comServ->GetTimeSyncServiceIface()->GetSlaveService();
```

Now the described above callbacks shall be registered within the service.

```
if(ERROR_CODE_OK != timeSyncSlave->RegisterGetSyncTimeCallback(
                    std::bind(&MyGetLocalTimeCallback,std::placeholders::_1)))
{
    std::cout << "Failed to register get local time callback" << std::endl;
}
if(ERROR_CODE_OK != timeSyncSlave->RegisterSetSyncOffsetCallback(
                    std::bind(&MySetOffsetCallback,std::placeholders::_1)))
{
    std::cout << "Failed to register set local offset" << std::endl;
}
```

Now we need to set slave id, the same slave id shall be added at the master side.

```
if(ERROR_CODE_OK != timeSyncSlave->SetSlaveId(0))
{
    std::cout << "Failed to set slave id" << std::endl;
}
```

To enable the services please call the following function.

```
if(ERROR_CODE_OK != timeSyncSlave->Enable())
{
    std::cout << "Failed to enable time sync service" << std::endl;
}
```

To disable the services please call the following function.

```
if(ERROR_CODE_OK != timeSyncSlave->Disable())
{
    std::cout << "Failed to disable time sync service" << std::endl;
}
```

6.3 SOFTWARE UPDATE SERVICE

After successful initializing of the communication services (see call on top of this chapter) the software update service is ready to use. The following preconditions must be fulfilled before a new software update can be triggered:

1. Preparation of proper UpdateInfoService-Handler. Before triggering the software update service it is mandatory to define an adequate 'ServiceHandler' which must match with following function type:

```
typedef std::function<void(SWUpdateInfo&)> SWUpdateCallback;
```

```
//Example: Own UpdateInfoService-Handler
void MyServiceInfoHandler(SWUpdateInfo& info)
{
   //Own Code to use the info object
   //e.g.copy info into own created info-Object
   ownUpdateInfoObject = info;
   ...
}
```

This handler will be handed over as function object to the update service and is used as callback to inform about the current update status via the info argument from type 'SWUpdateInfo'. With this object the user can get information about the status of the update procedure. The following states are defined:

```
enum SWUpdateStatus
{
    RUNNING = 0,
    READY_SUCCESS,
    STOPPED_BY_MASTER,
    STOPPED_BY_SLAVE,
    STOPPED_BY_ERROR_TIMEOUT,
    STOPPED_BY_ERROR_BLOCK_REPEAT,
    STOPPED_BY_ERROR_IMAGE_INVALID,
    STOPPED_BY_ERROR_UNKNOWN,
    NOT_VALID
};
```

To get status information about the current upload the following Getter-Function can be used:

```
SWUpdateStatus GetUpdateStatus()
```

To get information about Path/Filename of the current uploaded image please use:

```
std::string GetCurrentImage()
```

To be informed which image segment is currently uploaded please use:

```
uint32_t GetCurrentSegmentNumber()
```

To get information about number of current uploaded image bytes please use:

```
uint64_t GetCurrentDownloadedBytes()
```

If the image has only one segment the call for getting downloaded bytes provides the number of downloaded bytes from current image. If the image has more than one segment this Getter-Function provides the number of downloaded bytes from current data segment. So with proper using of here described service handler it is possible for the user to follow the status of a started update procedure.

2. Preparation of string object related to type of update image.

   – Upload of images as binary files

   ```
   //Create a string with path to binary imagefile
   std::string updateImageBin = "/OwnImageDirectory/OwnBinaryImage.enc";
   ```

   In this case the target platform expects the download of a compressed binary imagefile which must be handed over as parameter to the update service (typical suffix is .enc).

   – Upload of images as XML-Files

   ```
   /Create a string with path to XML-Imagefile
   std::string updateImageXml = "/OwnImageDirectory/OwnXmlImage.xml";
   ```

   In this case an imagefile in XML-Format will be handed over as parameter to the update service (typical suffix is .xml). The update service recognizes the XML-Format of the imagefile and triggers a XML-Parser to prepare related datasegments which will be transferred to the target platform.

3. Get handler to Update Service.

   ```
   std::shared_ptr<com::master::UpdateServiceIface> updateServicePtr =
                        com::master::CommunicationServicesIface::Get()->GetUpdateService();
   ```

   With this updateServicePtr it is now possible to start a new software update and to stop (Master-Break) a running update. It is not possible to trigger several software update procedures in parallel. A second start of this service during a running update without breaking up the first update leads immediately to an error (false return of 'SoftwareUpdate-Call'). If the described preconditions are fulfilled the software update procedure can be started.

4. Start a new update/stop a running update: SoftwareUpdate()/AbortSoftwareUpdate()

   ```
   //Prototype of SoftwareUpdate():
   ErrorCode UpdateService::SoftwareUpdate(IN std::string& updateImage, IN ClientId clientID,
                                           IN SWUpdateCallback callback);

   //Code-Example: Parameter updateImage and MyServiceInfoHandler already prepared
   ClientId  clientID = specificSensorId;
   uint64_t  imageBytesDownloaded = 0;
   ownAbortFlag = false;
   if(updateServicePtr->SoftwareUpdate(updateImage, clientID, MyServiceInfoHandler) !=
       ERROR_CODE_OK)
   {
      std::cout << "Start of SW-Download failed - Exit Update-Service" << std::endl;
      return false;
   }
   else
   {
      //Update-Service is running - Observe the Callbacks from Update-Service
      ...
      //Check upload info about currently downloaded Bytes
      if(imageBytesDownloaded != ownUpdateInfoObject.GetCurrentDownloadedBytes())
      {
         imageBytesDownloaded = ownUpdateInfoObject.GetCurrentDownloadedBytes();
         std::cout << "Currently downloaded Bytes = " << imageBytesDownloaded
                   << std::endl;
      }
      //Check upload info about current upload status
      if(ownUpdateInfoObject.GetUpdateStatus() != RUNNING)
      {
         std::cout << "Current upload has stopped, status = " << ownUpdateInfoObject.
             GetUpdateStatus()
                   << std::endl;
      }
   ```

```
    ...
    ...
    //Check for Master-Abort of current running update
    if(ownAbortFlag)
    {
        updateServicePtr->AbortSoftwareUpdate();
    }
    ...
}
```

After successful starting the update procedure via call of SoftwareUpdate() all further calls of SoftwareUpdate() will be ignored (return with error) and the running procedure will be continued until the update has stopped by success, by error or by master break command. Thereby all elements of update service needed to execute a software update via Smart Access are known and can be used in an own user application.

5. Simple example how SOFTWARE UPDATE SERVICE could be used

```
//Define MyServiceInfoHandler
void MyServiceInfoHandler(SWUpdateInfo& info)
{
    //Example-Code: Copy info into own application object
    ownUpdateInfoObject = info;
    ...
}
```

```
//Example-Code for using update service functions
bool ownUpdateServiceFunction()
{
    //Prepare Update-Parameter ------------------------------------------------
    //Initialize updateImage-Parameter with path to binary image
    std::string image = "/OwnImageDirectory/OwnBinaryImage.enc";
    //Example: Define own UpdateInfo-Object
    SWUpdateInfo ownUpdateInfoObject;
    ownUpdateInfoObject.SetUpdateStatus(RUNNING);
    ownUpdateInfoObject.SetCurrentImage(image);
    ownUpdateInfoObject.SetCurrentDownloadedBytes(0);
    ownUpdateInfoObject.SetCurrentSegmentNumber(0);
    //Initialize _clientID e.g. with ID = 0
    ClientId ownSensorClientID = 0;
    //Initialize ownAbortFlag
    ownAbortFlag = false;

    //Initialize Communication-Services ---------------------------------
    std::shared_ptr<com::master::CommunicationServicesIface> comServicesPtr =
                    com::master::CommunicationServicesIface::Get();
    if(comServicesPtr->Init() != true)
    {
        std::cout << "Initialization of Communication-Services failed" << std::endl;
        return false;
    }
    else
    {
        //Get Update-Service ------------------------------------------------
        std::shared_ptr<com::master::UpdateServiceIface> updateServicePtr =
                        comServicesPtr->GetUpdateService();
        while(1)
        {
            uint64_t imageBytesDownloaded = 0;
            //Start next SW-Update via UpdateService-Function -----------------
            if(updateServicePtr->SoftwareUpdate(image, ownSensorClientID, MyServiceInfoHandler)
                != ERROR_CODE_OK)
            {
                std::cout << "Start of new SW-Download failed" << std::endl;
            }
            ...
```

```
...
```

Continuance of function example:

```
            ...
            else
            {
                //Example to observe running update service ----------------------------
                //During running update further calls of SoftwareUpdate() will be ignored.
                //New start of software update is only possible after
                //update status is unequal to state RUNNING
                if(imageBytesDownloaded != ownUpdateInfoObject.GetCurrentDownloadedBytes())
                {
                    imageBytesDownloaded = ownUpdateInfoObject.GetCurrentDownloadedBytes();
                    std::cout << "Currently downloaded Bytes = " << imageBytesDownloaded
                              << std::endl;
                }
                //Check upload info about current upload status
                if(ownUpdateInfoObject.GetUpdateStatus() != RUNNING)
                {
                    //update service stopped - check the update result ------------
                    switch(ownUpdateInfoObject.GetUpdateStatus())
                    {
                        case READY_SUCCESS:
                        std::cout << "Software-Update successful finished" << std::endl;
                        break;
                        case STOPPED_BY_MASTER:
                        std::cout << "Software-Update stopped by Master" << std::endl;
                        break;
                        case STOPPED_BY_SLAVE:
                        std::cout << "Software-Update is stopped by Slave" << std::endl;
                        break;
                        ...
                        default:
                    }
                }
                ...
            }
        }
    }
}
```

6.4 DEVICEMONITOR SERVICE

After successful initializing of the communication services the device monitor service can be used. With this service it is possible to get information about all connected devices.

1. Get handler to device monitor service.

```
std::shared_ptr<com::master::DeviceMonitorServiceIface> deviceMonitorServicePtr =
                com::master::CommunicationServicesIface::Get()->GetDeviceMonitorService();
```

With this deviceMonitorServicePtr it is now possible to get information about connected devices and to register a notification callback to be informed about changes in the connected devices list.

2. Preparation and registration of proper DeviceMonitorService-Handler.

```
typedef std::function<void(ClientId id,bool clientConnectedStatus)> ConnectedClientCallback;
```

For getting related notifications about changes in the list of connected devices it is possible to define and register an adequate 'ServiceHandler' which must match with this function type "ConnectedClientCallback". Following register call is defined for the callback registration:

```
ErrorCode RegisterConnectedClientCallback(IN ConnectedClientCallback callback);
```

- First Argument in ConnectedClientCallback is "ClientId id" which represents the client ID of the device which is new connected or which is not connected anymore.
- Second Argument in ConnectedClientCallback is "bool clientConnectedStatus". If clientConnectedStatus is set to true, the device with related client ID is new connected or reconnected. If clientConnectedStatus is set to false, the device with related client ID is not connected anymore.
- Register-Call to register notification. With calling the register call for handing over an own ConnectedClient-Callback the devicemonitor service is able to inform the user of communication services about a change in the connected client list. The registration is not mandatory but without this registration the devicemonitor service is not able to send a related notification after a connection status change.

3. Get list of connected devices

```
bool GetConnectedClientInfo(OUT std::vector<ConnectedClientInfo>& clientInfo);
```

With this call it is possible to get the list of connected devices via class ConnectedClientInfo:

```
class ConnectedClientInfo
{
public:
    ConnectedClientInfo(ClientId id, ClientPhyLinkType phyType) :
        _clientId(id),
        _phyLinkType(phyType)
    {
    }

    virtual ~ConnectedClientInfo()
    {
    }

    inline ClientId GetConnectedClientId() const
    {
        return _clientId;
    }

    inline ClientPhyLinkType GetPhyLinkType() const
    {
        return _phyLinkType;
    }

private:
    ClientId            _clientId;
    ClientPhyLinkType   _phyLinkType;
};
```

- Get clientId of an element in the list of connected devices via call of GetConnectedClientId().
- Get Link-Type of an element in the list of connected devices via call of GetPhyLinkType().

```
enum ClientPhyLinkType
{
    CLIENT_PHY_TYPE_CAN = 0,
    CLIENT_PHY_TYPE_ETH,
    CLIENT_PHY_TYPE_RS485,
    CLIENT_PHY_TYPE_UNKNOWN
};
```

4. Check connection status via client-ID.

```
bool IsClientConnected(IN ClientId clientId);
```

To get the current connection status of a device with known client-ID this call can be used. If the return value is true, the device is connected. If the return value is false, the device is not connected.

5. Code-Example for using devicemonitor service

```
void MyConnectedInfoHandler(ClientId Id, bool clientStatus)
{
   //Own code to handle the notification info, e.g.
   if(clientStatus) {
       std::cout << "Device with id = " << Id << " is now connected" << std::endl;
   }
   else {
       std::cout << "Device with id = " << Id << " is not connected anymore" << std::endl;
   }
}
```

```
//Communication-Services must be already initialized before calling the following Example-
    Function:
bool ownDeviceMonitorServiceFunction()
{
    bool retVal = false;
    //Get DeviceMonitor-Service ------------------------------------------------
    std::shared_ptr<com::master::DeviceMonitorServiceIface> deviceMonitorServicePtr =
            com::master::CommunicationServicesIface::Get()->GetDeviceMonitorService();
    //RegisterClientCallback
    if(deviceMonitorServicePtr->RegisterConnectedClientCallback(MyConnectedInfoHandler)
                                == ERROR_CODE_OK)
    {
        //Create empty clientInfoVector
        std::vector<ConnectedClientInfo> clientInfoVector;
        //Get Client-List of current connected Devices
        if(deviceMonitorServicePtr->GetConnectedClientInfo(clientInfoVector))
        {
            //Iterator-Loop over ConnectedClientInfo-Elements to read connected clientIds
            std::vector<ConnectedClientInfo>::iterator it;
            for(it = clientInfoVector.begin(); it != clientInfoVector.end(); ++it)
            {
               std::cout << "ClientId = " << it->GetConnectedClientId() << " is connected"
                         << std::endl;
               //Set PhyLinkType of found connected element
               ClientPhyLinkType linkType = it->GetPhyLinkType();
               switch(linkType)
               {
                  case CLIENT_PHY_TYPE_CAN:
                  {
                     std::cout << "LinkType is PHY_CAN" << std::endl;
                     ret_val = true;
                  }
                  break;
                  case CLIENT_PHY_TYPE_ETH:
                  {
                     std::cout << "LinkType is PHY_ETH" << std::endl;
                     ret_val = true;
                  }
                  break;
                  case CLIENT_PHY_TYPE_RS485:
                  {
                     std::cout << "LinkType is PHY_RS485" << std::endl;
                     ret_val = true;
                  }
                  break;
               }
            }
        }
    }
    return retVal;
}
```

7 OPERATING SYSTEM ABSTRACTION LAYER(OSAL)

OSAL is a glue layer between the Smart Access library and the customers underlying operating system. It needs to support the customers CAN, RS485 and Ethernet drivers. The OSAL implementation should provide the following functionality:

7.1 OSAL INTERFACE

In this section is provided a platform independent interface description, which can be commonly implemented for Linux, Windows or other operating systems.

7.1.1 GETTING OSAL INSTANCE

The upper layer(Smart Access Library) shall be able to retrieve the OSAL instance by calling the function below:

```
static std::shared_ptr<OSALIface> GetOSAL();
```

It could be implemented in the following way:

```
static std::shared_ptr<OSALImpl> instance;

std::shared_ptr<OSALIface> OSALIface::GetOSAL()
{
    if(!instance)
    {
        instance.reset(new OSALImpl());
    }
    return instance;
}
```

7.1.2 INITIALIZATION

The upper layer(Smart Access Library) shall be able to initialize the OSAL layer:

```
virtual bool Init(IN const std::string& configPath) = 0;
```

Returns true, if the initialization was successful and false, if it failed. It gets a configuration path (please see configuration section of this manual) as argument.

7.1.3 GETTING LIST OF HW DEVICES

The upper layer(Smart Access Library) shall be able to retrieve from OSAL a list of all supported hardware devices :

```
virtual bool GetHwIfaces(OUT std::list<std::shared_ptr<HwIfaceDescriptor>>& ) = 0;
```

Where **HwIfaceDescriptor** is the base class, those descendants are:

7.1.3.1 HW CAN DESCRIPTOR

Class HwCanDescriptor has the following members:

```
private:
    CanDevId        _deviceId;
    std::string     _ifaceName;
    uint32_t        _baudrate;
```

- **_deviceId** – CAN device id. IMPORTANT: it must match with device id from the routing table.
- **_ifaceName** – CAN interface name.
- **_baudrate** – CAN baudrate .

For more information please see ExternalTypes.h file.

7.1.3.2 HW RS485 DESCRIPTOR

Class HwRS485Descriptor has the following members:

```
private:
    SerialDevId         _deviceId;
    std::string         _ifaceName;
    uint32_t            _baudrate;
```

- **_deviceId** – Serial Device Id. IMPORTANT: it must match with device id from the routing table.
- **_ifaceName** – Interface name e.g. “/dev/ttyS0” or “COM1”.
- **_baudrate** – RS485 baudrate.

For more information please see ExternalTypes.h file.

7.1.3.3 HW ETHERNET DESCRIPTOR

Class HwEthernetDescriptor has the following members:

```
private:
    uint16_t        _deviceId;
    std::string     _ifaceName;
    std::string     _ip;
    uint32_t        _port;
```

- **_deviceId** – Ethernet adapter device id.
- **_ifaceName** – Interface name e.g. “eth0”.
- **_ip** – IP Address.
- **_port** – UDP port.

For more information please see ExternalTypes.h file.

7.1.4 HW REGISTRY

The purpose of Hardware Registries is to describe a certain physical interface. As opposed to a hardware descriptor it describes both real physical hardware devices e.g. CAN and logical ones e.g. UDP sockets. **HwRegistry** , similar to **HwIfaceDescriptor** , is used in the interface as a base class and needs to be casted in OSAL implementation to its descendants:

7.1.4.1 CAN HW REGISTRY

Class CanHwRegistry has the following members:

```
private:
    std::set<CanId>             _ids;
    CanDevId                    _deviceId
```

- **_ids** – List of CAN Identifiers that shall be received by CAN device, the rest needs to be dropped.

– **_deviceId** – CAN device ID. The same id as in the hardware descriptor.

For more information please see ExternalTypes.h file.

7.1.4.2 RS485 HW REGISTRY

Class Rs485HwRegistry has the following members:

```
private:
    SerialDevId                 _deviceId
```

– **_deviceId** – Serial device ID. The same ID as in the hardware descriptor.

For more information please see ExternalTypes.h file.

7.1.4.3 ETHERNET HW REGISTRY

Class EthernetRegistry has the following members:

```
private:
    std::string                 _ip;
    uint32_t                    _port;
    EthTransportType            _transType;
```

– **_ip** – IP address.
– **_port** – UDP/TCP port.
– **_transType** – Transport type TCP/UDP/UDP MULTICAST (TCP is currently not supported).

For more information please see ExternalTypes.h file.

7.1.5 REGISTER RECEIVER

In order to receive incoming packets , the upper layer need to be able to register receiver callbacks, these callbacks shall be invoked upon receiving data on the described in hardware registry interface.

```
virtual ErrorCode RegisterReceiver(IN const HwRegistry& registry,
                                   IN ReceiverCallback callback) = 0;
```

The receiver callback has the following format:

```
typedef std::function<void(BufferDescriptor&)> ReceiverCallback;
```

Where the buffer descriptor is built as follows:

```
class BufferDescriptor
{
public:
    BufferDescriptor(uint8_t* bufferPtr, size_t size);
    BufferDescriptor();

    ~BufferDescriptor();

    uint8_t* GetBufferPtr() const;
    size_t GetSize() const;
    void SetBufferPtr(IN uint8_t* bufferPtr);
    void SetSize(IN size_t size);


private:
    uint8_t*            _bufferPtr;
    size_t              _size;
```

```
};
```

- **_bufferPtr** – Pointer to the allocated buffer. Please notice that the OSAL is responsible to allocate and to free this memory.
- **_size** – Size of the allocated buffer.

7.1.6 UNREGISTER RECEIVER

In order to remove a previously registered callback, the interface below will be called.

```
virtual ErrorCode UnRegisterReceiver(IN const HwRegistry& registry) = 0;
```

7.1.7 SEND DATA

The upper layer(Smart Access Library) shall be able to send data to a certain hardware registry.

```
virtual ErrorCode SendData(IN const HwRegistry& registry,
                           IN const BufferDescriptor& buffer) = 0;
```

Please do not free this memory it will be freed by the upper layer.

For more information about OSAL interface please see the OSALIface.h file.

7.2 CONFIGURATION

The Osal implementation is configured by the "hw_inventory". The configurations are described in the subsection Linux 7.2.1 and Windows 7.2.2.

7.2.1 Linux

Exemplary CAN configuration:

The baudrate and the interface must be set up before the API is executed. For this purpose the device must be registered as Socket CAN and the baudrate must be set to 500 kbit/s. As an example with a Peak Adapter:

sudo ip link set can_interface up type bitrate 500000.

When using LAWICEL adapter:

sudo slcand -o -c -f s6 /dev/ttyUSBX can_interface.

This command sets up a CAN interface (e.g., slcan0) with a baudrate 500 kbit/s and the device USBX, which can be viewed from the ls /dev command. After setting up a CAN interface with a baudrate, we need to bring the interface up with:

sudo ifconfig can_interface up.

Ideally, the user should verify if the setup has been successful, and this is possible with a candump of the interface.

```
"hwItems" :
[
    {
        "type"          : "can",
        "dev_id"        : 1,
```

```
            "iface_name" : "can0",
            "baudrate"   : 500000
        },
        ......
    ]
```

- **type** - Device type, set to "can".
- **dev_id** - ID of the CAN device required by the OSAL layer for identifying the adapter to be used.
- **iface_name** - Interface name of the used can socket.
- **baudrate** - OUTDATED. The baudrate must be set before execute Smart Access.

Exemplary RS485 configuration:

```
    "hwItems" :
    [
        {
            "type"    : "rs485",
            "dev_id"      : 1,
            "iface_name" : "/dev/ttyS0",
            "baudrate" : 115200
        },
        ......
    ]
```

- **type** - Device type, set to "rs485".
- **dev_id** - ID of the CAN device required by the OSAL layer for identifying the adapter to be used.
- **iface_name** - Interface name of the used rs485 socket.
- **baudrate** - Uart baudrate.

Exemplary Ethernet configuration:

```
    "clients" :
    [
        {
            "type"      : "eth",
            "dev_id"        : 1,
            "iface_name"    : "eth0",
            "port" : 55555
        },
        ......
    ]
```

- **type** - Device type, set to "eth".
- **dev_id** - ID of the CAN device required by the OSAL layer for identifying the adapter to be used.
- **iface_name** - Interface name of the used ethernet socket.
- **port** - Receive udp port.

7.2.2 Windows

There is no configuration for an CAN device, because the reference OSAL does not support CAN on Windows.

Exemplary RS485 configuration:

```
    "clients" :
    [
        {
            "type"      : "rs485",
            "deviceId"  : "2",
            "usedPort"  : "COM1",
            "linkSpeed" : "115200"
```

```
        },
        ......
    ]
```

- **type** - Device type, set to "rs485".
- **deviceId** - ID of the CAN device required by the OSAL layer for identifying the adapter to be used.
- **usedPort** - It is the name of the COM port.
- **linkSpeed** - Uart baudrate.

Exemplary Ethernet configuration:

```
    "clients" :
    [
        {
            "type"      : "eth",
            "deviceId"  : "1",
            "usedPort"  : "65000",
            "ipAddress" : "192.168.11.1"
        },
        ......
    ]
```

- **type** - Device type, set to "eth".
- **deviceId** - ID of the CAN device required by the OSAL layer for identifying the adapter to be used.
- **usedPort** - Receive udp port.
- **ipAddress** - Used pc ip address.

8 LEGAL DISCLAIMER NOTICE

All products, product specifications and data in this document may be subject to change without notice to improve reliability, function or otherwise. Not all products and/or product features may be available in all countries and regions. For legal reasons features may be deleted from products or smartmicro may refuse to offer products. Statements, technical information and recommendations contained herein are believed to be accurate as of the stated date. smartmicro disclaims any and all liability for any errors, inaccuracies or incompleteness contained in this document or in any other disclosure relating to the product.

To the extent permitted by applicable law, smartmicro disclaims (i) any and all liability arising out of the application or use of the product or the data contained herein, (ii) any and all liability of damages exceeding direct damages, including - without limitation - indirect, consequential or incidental damages, and (iii) any and all implied warranties, including warranties of the suitability of the product for particular purposes.

Statements regarding the suitability of products for certain types of applications are based on smartmicro's knowledge of typical requirements that are often placed on smartmicro products in generic/general applications. Statements about the suitability of products for a particular/specific application, however, are not binding. It is the customer's/user's responsibility to validate that the product with the specifications described is suitable for use in the particular/specific application. Parameters and the performance of products may deviate from statements made herein due to particular/specific applications and/or surroundings. Therefore, it is important that the customer/user has thoroughly tested the products and has understood the performance and limitations of the products before installing them for final applications or before their commercialization. Although products are well optimized to be used for the intended applications stated, it must also be understood by the customer/user that the detection probability may not be 100% and that the false alarm rate may not be zero.

The information provided, relates only to the specifically designated product and may not be applicable when the product is used in combination with other materials or in any process not defined herein. All operating parameters, including typical parameters, must be validated for each application by the customer's/user's technical experts. Customers using or selling smartmicro products for use in an application which is not expressly indicated do so at their own risk. This document does not expand or otherwise modify smartmicro's terms and conditions of purchase, including but not being limited to the warranty. Except as expressly indicated in writing by smartmicro, the products are not designed for use in medical, lifesaving or life-sustaining applications or for any other application in which the failure of the product could result in personal injury or death.

No license expressed or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document or by any conduct of smartmicro. Product names and markings noted herein may be trademarks of their respective owners.

Please note that the application of the product may be subject to standards or other regulations that may vary from country to country. smartmicro does not guarantee that the use of products in the applications described herein will comply with such regulations in any country. It is the customer's/user's responsibility to ensure that the use and incorporation of products comply with regulatory requirements of their markets.

If any provision of this disclaimer is, or is found to be, void or unenforceable under applicable law, it will not affect the validity or enforceability of the other provisions of this disclaimer.