SMART ACCESS USER MANUAL

DATE:
August 31, 2020

USER INTERFACE VERSION:
TM v5.2.0

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.

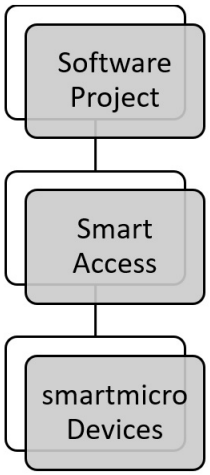


Figure 1: Smart Access context diagramm

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.

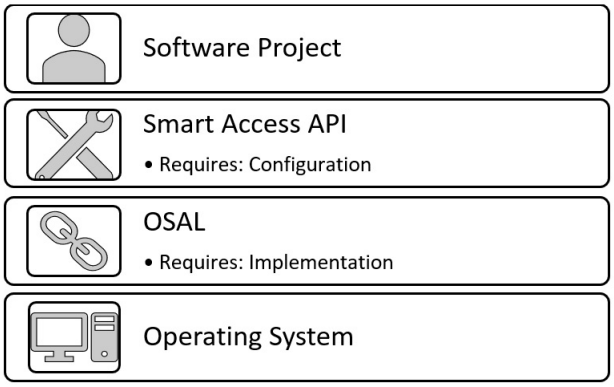


Figure 2: Smart Access diagramm

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 ADVANCED PROGRAMMERS INTERFACE (API)

The Advanced Programmers 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 is required.

3.2 DELIVERY PACKAGE

The Smart Access product contains the following items:

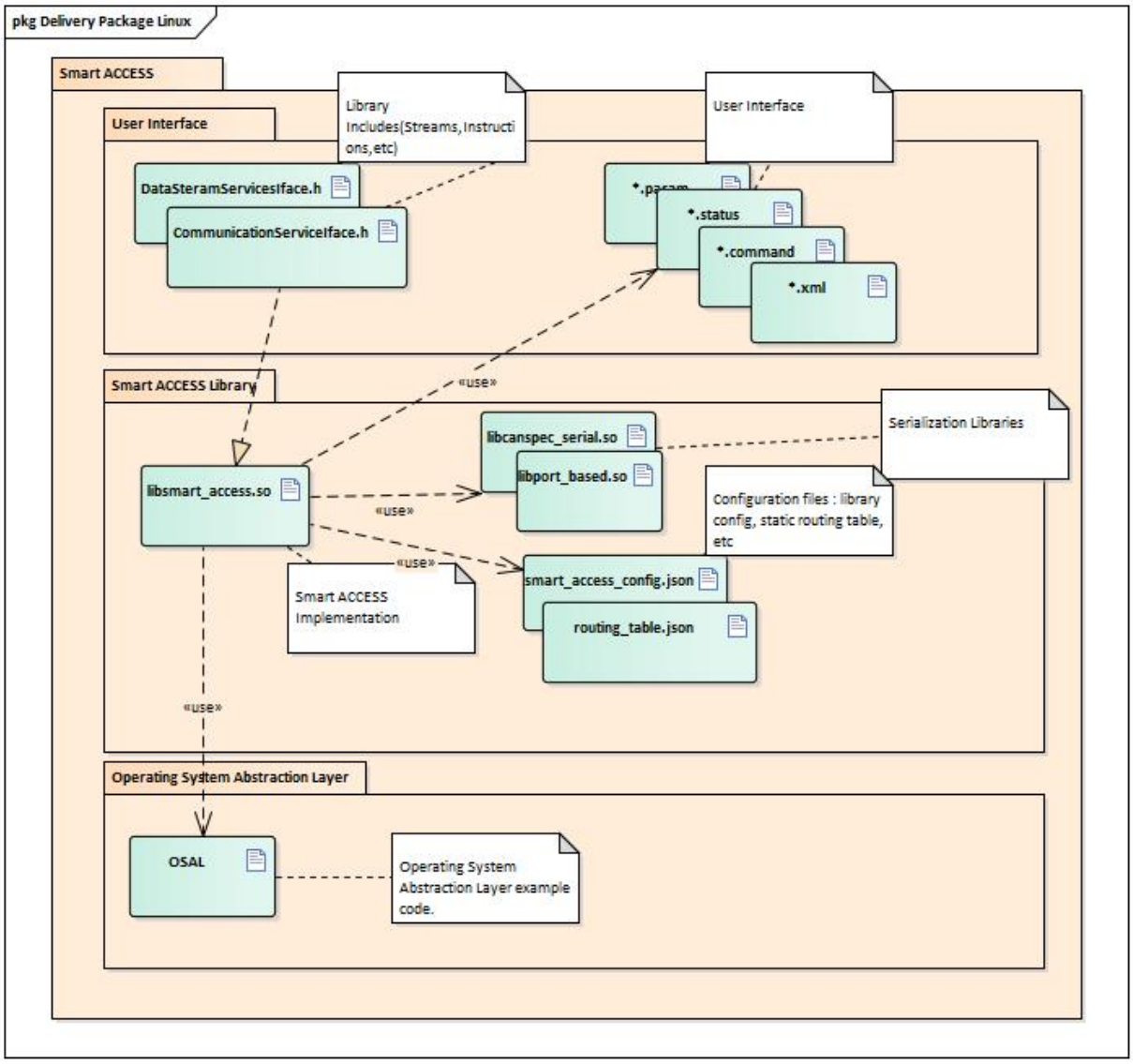


Figure 3: Deployment Diagramm

– **libsmart_access.so** – Main library responsible for all kinds of communication to smartmicro radar sensors and a corresponding set of header files.

- **libosal.so** – ”Dummy” OSAL implementation for the given operating system. It does not contain the real logic and allows the customer to compile his project. Together with the binary will be provided the source code of the dummy implementation, which can serve as a base for the customer OSAL. A reference OSAL code to illustrate, how it could be done, is provided as well.
- **libcan_format_serialization.so** - Serialization/deserialization library for the CAN-based communication.
- **libport_based_serialization.so** - Serialization/deserialization library for smartmicro’s port-based communication.
- **smart_access_cfg.json** - Reference configuration file.
- **routing_table.json** - Reference routing table for static routes.
- **hw_inventory.json** - Reference OSAL configuration .
- **user_interface** - Set of .json files (param, status and commands) to describe an interface to a specific smartmicro sensor platform. All the files will be listed in ”[user_interface_name].interface” file.

4 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:

4.1 OSAL INTERFACE

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

4.1.1 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;
}
```

4.1.2 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.

4.1.3 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:

4.1.3.1 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.

4.1.3.2 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.

4.1.3.3 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.

4.1.4 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:

4.1.4.1 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.

4.1.4.2 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.

4.1.4.3 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.

4.1.5 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.

4.1.6 UNREGISTER RECEIVER

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

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

4.1.7 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.

5 CONFIGURATION

5.0.1 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).

5.0.2 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:

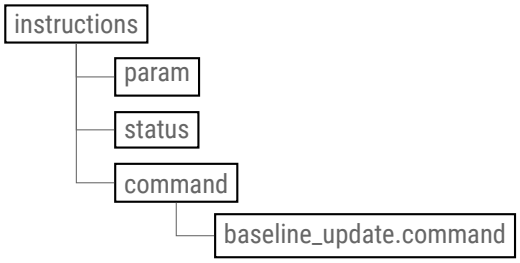
- **can_based** - Data will be split into CAN messages.
- **port_based** - Port data will be sent unchanged.

5.0.3 SMART ACCESS CONFIGURATION

The location of the configuration file needs to be set in the "SMART_ACCESS_CFG_FILE_PATH" environment variable. The format of the file should be as follows:

```
{
    "name": "Com Lib Config",
    "version" : "3.1.0",
    "client_id": 5,
    "role": "master",
    "shared_lib_path": "/usr/lib",
    "config_path": "/data/config",
    "user_interface_path": "/data/applet",
    "download_path": "/data/update",
    "instruction_serialization_type": "can_based",
    "data_serialization_type": "can_based"
}
```

- **name** - For informative purposes only.
- **version** - Indicating the Smart Access library version.
- **client_id** - The library's client identification number used for routing purposes.
- **shared_lib_path** - Absolute path to Smart Access libraries, such as can_format_serialization and port_based_serialization.
- **role** - Defined as "master" or "slave". Usually, a management system is determined as master and smartmicro sensors are determined as slaves.
- **config_path** - Absolute path to the folder, which contains configuration files.
- **user_interface_path** - Absolute path to the user interface folder, which should have a sub-folder "instructions" with the following structure:



- **download_path** - Path to the download folder, which is only relevant for slave devices.
- **instruction_serialization_type** - Serialization type of the instruction service, which can be either CAN-based (using "can_based") or port-based (using "port_based").
- **datai_serialization_type** - Serialization type of the data stream service, which can be either CAN-based (using "can_based") or port-based (using "port_based").

5.0.4 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.

5.0.4.1 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" :
    [
    ]
}
```

5.0.4.2 STATIC ROUTING

Static routes need to be added to the "routing_table.json" file having the following format:

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

- **name** - For informative purposes only.
- **version** - Indicating the Smart Access library version.
- **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.

Exemplary CAN configruation:

```
    "clients" :
    [
        {
            "client_id" : 1 ,
            "link_type": "can",
            "can_network_id": 0,
            "dev_id": 1
        },
        ......
    ]
```

- **clienti_id** - Client identifier with 32bit, which needs to be unique.
- **link_type** - Channel type, set to CAN "can".

- **can_network_id** - If several clients are using the same CAN bus, each client additionally needs a unique identifier in the range of 0 to14.
- **dev_id** - ID of the CAN device required by the OSAL layer for identifying the adapter to be used.

Exemplary RS485 configuration:

```
"clients" :
[
    {
        "client_id" : 1 ,
        "link_type": "rs485",
        "dev_id": 1,
        "instructuion_serialization_type" : "can_based",
        "data_serialization_type" : "can_based"
    },
    ......
]
```

- **client_id** - Client identifier with 32bit, which needs to be unique.
- **link_type** - Channel type, set to "rs485".
- **dev_id** - ID of the RS485 device required by the OSAL layer for identifying the serial device to be used.
- **instructuion_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").
- **data_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").

Exemplary Ethernet configuration:

```
"clients" :
[
    {
        "client_id" : 1 ,
        "link_type": "eth",
        "ip": "127.0.0.1",
        "port": 5555,
        "instructuion_serialization_type" : "port_based",
        "data_serialization_type" : "port_based"
    },
    ......
]
```

- **client_id** - Client identifier with 32bit, which needs to be unique.
- **link_type** - Channel type, set to "eth".
- **ip** - The client's IP address in text format.
- **port** - The client's UDP port number.
- **instructuion_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").
- **data_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").

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. 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 =
                            instServ->AllocateInstructionBatch();
```

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.

```
    batch->AddRequest(param1);
    batch->AddRequest(param2);
    batch->AddRequest(status);
    batch->AddRequest(cmd);
```

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> reponse)
     {
            // it could be responses with the same key but with different dimensions
            std::vector<std::shared_ptr<Response<uint8_t>>> myResp;
            if(reponse->GetResponse<uint8_t>("some_section","some_parameter",myResp))
            {
                for(auto &resp : myResp)
                {
                    uint8_t value = GetValue();

                }
            }
            std::vector<std::shared_ptr<Response<float>>> statuses;
            if(reponse->GetResponse<float>("some_section","some_status",statuses))
            {
                for(auto &resp : statuses)
                {
                    float value = GetValue();

                }
            }

     };


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

9. Please define a list of clients, that need to be configured.

```
        std::set<ClientId>  clients;
        clients.insert(2) // add client id 2
```

10. Send the instructions to the clients.

```
         instServ->SetInstructionBatch(clients,batch,callback);
```

11. 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(!reponse->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;
}
```

12. 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
```

6.2 COMMUNICATION DATA STREAM SERVICE

With the communication data stream service smartmicro ports can be received as C++ objects with simplified access functions, which are generated by the user interface. Smartmicro ports are data buffers which contain recorded by the radar data: objects, statistics, statuses of device etc. Each port contains a generic port header, with a port description : version, id, size etc. Sometimes contains a port also a dynamic list of objects. In order to receive a port, a callback needs to be registered with the service. The callback will be carried out periodically every sensor cycle time. Please note: This callback will be called in the context of a receiver thread, so the data needs to be copied and th thread must be released. Otherwise, the reception will be blocked. For more details please see the examples below. The following ports are supported.

6.2.1 TARGETLISTPORT PORT

Description:
To receive the port called "TargetListPort" from a specific client, please use the following registration interface:

```
void ReceiveTargetListPortClbk(IN std::shared_ptr<com::master::targetlistport::TargetListPort>
    targetListPort)
{

    // Getting members of  TargetListPort
    std::shared_ptr<targetlistport::GenericPortHeader> genericPortHeader =
                    targetListPort->GetGenericPortHeader();
    std::shared_ptr<targetlistport::StaticPortHeader> staticPortHeader =
                    targetListPort->GetStaticPortHeader();
    auto targetList = targetListPort->GetTargetList();

    // Getting members of  GenericPortHeader
    std::cout << "Variable PortId:"
```

```
            << genericPortHeader->GetPortId()
            << std::endl;
    std::cout << "Variable PortVersionMajor:"
            << genericPortHeader->GetPortVersionMajor()
            << std::endl;
    std::cout << "Variable PortVersionMinor:"
            << genericPortHeader->GetPortVersionMinor()
            << std::endl;
    std::cout << "Variable Timestamp:"
            << genericPortHeader->GetTimestamp()
            << std::endl;
    std::cout << "Variable PortSize:"
            << genericPortHeader->GetPortSize()
            << std::endl;
    std::cout << "Variable BodyEndianness:"
            << genericPortHeader->GetBodyEndianness()
            << std::endl;
    std::cout << "Variable PortIndex:"
            << genericPortHeader->GetPortIndex()
            << std::endl;
    std::cout << "Variable HeaderVersionMajor:"
            << genericPortHeader->GetHeaderVersionMajor()
            << std::endl;
    std::cout << "Variable HeaderVersionMinor:"
            << genericPortHeader->GetHeaderVersionMinor()
            << std::endl;
    // Getting members of  StaticPortHeader
    std::cout << "Variable NumberOfTargets:"
            << staticPortHeader->GetNumberOfTargets()
            << std::endl;
    // Getting members of  Target
    for(auto& target : targetList)
    {
        std::cout << "Variable Range1:"
                << target->GetRange1()
                << std::endl;
        std::cout << "Variable SigmaRange:"
                << target->GetSigmaRange()
                << std::endl;
        std::cout << "Variable AZAngleDeg:"
                << target->GetAZAngleDeg()
                << std::endl;
        std::cout << "Variable SigmaAngle:"
                << target->GetSigmaAngle()
                << std::endl;
        std::cout << "Variable SpeedRadial:"
                << target->GetSpeedRadial()
                << std::endl;
        std::cout << "Variable SigmaSpeed:"
                << target->GetSigmaSpeed()
                << std::endl;
        std::cout << "Variable RCS:"
                << target->GetRCS()
                << std::endl;
        std::cout << "Variable Amplitude:"
                << target->GetAmplitude()
                << std::endl;
        std::cout << "Variable TgtNoise:"
                << target->GetTgtNoise()
                << std::endl;
        std::cout << "Variable Elevation:"
                << target->GetElevation()
                << std::endl;
        std::cout << "Variable SigmaAngleEl:"
                << target->GetSigmaAngleEl()
                << std::endl;
    }
```

```
}



.....
    ClientId clientId = 1024; // id of the client
    ReceiveTargetListPortCallback callback =
                            std::bind(&ReceiveTargetListPortClbk,
                            std::placeholders::_1);
    if(ERROR_CODE_OK != comDataServ->RegisterTargetListPortReceiveCallback(clientId,callback)
    {
        std::cout << "Failed to register TargetListPort port callback" << std::endl;
    }
```

6.2.2 OBJECTLISTPORT PORT

Description:
To receive the port called "ObjectListPort" from a specific client, please use the following registration interface:

```
void ReceiveObjectListPortClbk(IN std::shared_ptr<com::master::objectlistport::ObjectListPort>
    objectListPort)
{

    // Getting members of  ObjectListPort
    std::shared_ptr<objectlistport::GenericPortHeader> genericPortHeader =
                        objectListPort->GetGenericPortHeader();
    std::shared_ptr<objectlistport::StaticPortHeader> staticPortHeader =
                        objectListPort->GetStaticPortHeader();
    auto objectList = objectListPort->GetObjectList();

    // Getting members of  GenericPortHeader
    std::cout << "Variable PortId:"
              << genericPortHeader->GetPortId()
              << std::endl;
    std::cout << "Variable PortVersionMajor:"
              << genericPortHeader->GetPortVersionMajor()
              << std::endl;
    std::cout << "Variable PortVersionMinor:"
              << genericPortHeader->GetPortVersionMinor()
              << std::endl;
    std::cout << "Variable Timestamp:"
              << genericPortHeader->GetTimestamp()
              << std::endl;
    std::cout << "Variable PortSize:"
              << genericPortHeader->GetPortSize()
              << std::endl;
    std::cout << "Variable BodyEndianness:"
              << genericPortHeader->GetBodyEndianness()
              << std::endl;
    std::cout << "Variable PortIndex:"
              << genericPortHeader->GetPortIndex()
              << std::endl;
    std::cout << "Variable HeaderVersionMajor:"
              << genericPortHeader->GetHeaderVersionMajor()
              << std::endl;
    std::cout << "Variable HeaderVersionMinor:"
              << genericPortHeader->GetHeaderVersionMinor()
              << std::endl;
    // Getting members of  StaticPortHeader
    std::cout << "Variable Rain:"
              << staticPortHeader->GetRain()
              << std::endl;
    std::cout << "Variable NumberOfObjects:"
```

```
                << staticPortHeader->GetNumberOfObjects()
                << std::endl;
        std::cout << "Variable NumberOfMessages:"
                << staticPortHeader->GetNumberOfMessages()
                << std::endl;
        std::cout << "Variable CycleDuration:"
                << staticPortHeader->GetCycleDuration()
                << std::endl;
        std::cout << "Variable CycleCount:"
                << staticPortHeader->GetCycleCount()
                << std::endl;
        std::cout << "Variable ObjectData0Format:"
                << staticPortHeader->GetObjectData0Format()
                << std::endl;
        std::cout << "Variable ObjectData1Format:"
                << staticPortHeader->GetObjectData1Format()
                << std::endl;
        // Getting members of  Object
        for(auto& object : objectList)
        {
            std::cout << "Variable XPoint1:"
                    << object->GetXPoint1()
                    << std::endl;
            std::cout << "Variable YPoint1:"
                    << object->GetYPoint1()
                    << std::endl;
            std::cout << "Variable HeadingDeg:"
                    << object->GetHeadingDeg()
                    << std::endl;
            std::cout << "Variable ObjectLen:"
                    << object->GetObjectLen()
                    << std::endl;
            std::cout << "Variable SpeedAbs:"
                    << object->GetSpeedAbs()
                    << std::endl;
            std::cout << "Variable ObjectID:"
                    << object->GetObjectID()
                    << std::endl;
        }
    }



    .....
        ClientId clientId = 1024; // id of the client
        ReceiveObjectListPortCallback callback =
                                std::bind(&ReceiveObjectListPortClbk,
                                std::placeholders::_1);
        if(ERROR_CODE_OK != comDataServ->RegisterObjectListPortReceiveCallback(clientId,callback)
        {
            std::cout << " Failed to register ObjectListPort port callback" << std::endl;
        }
```

6.2.3 TMTRIGGERLISTPORT PORT

Description:
To receive the port called "TmTriggerListPort" from a specific client, please use the following registration interface:

```
    void ReceiveTmTriggerListPortClbk(IN std::shared_ptr<com::master::tmtriggerlistport::
        TmTriggerListPort> tmTriggerListPort)
    {

        // Getting members of  TmTriggerListPort
```

```
    std::shared_ptr<tmtriggerlistport::GenericPortHeader> genericPortHeader =
                        tmTriggerListPort->GetGenericPortHeader();
    std::shared_ptr<tmtriggerlistport::StaticPortHeader> staticPortHeader =
                        tmTriggerListPort->GetStaticPortHeader();
    // Getting members of  GenericPortHeader
    std::cout << "Variable PortId:"
              << genericPortHeader->GetPortId()
              << std::endl;
    std::cout << "Variable PortVersionMajor:"
              << genericPortHeader->GetPortVersionMajor()
              << std::endl;
    std::cout << "Variable PortVersionMinor:"
              << genericPortHeader->GetPortVersionMinor()
              << std::endl;
    std::cout << "Variable Timestamp:"
              << genericPortHeader->GetTimestamp()
              << std::endl;
    std::cout << "Variable PortSize:"
              << genericPortHeader->GetPortSize()
              << std::endl;
    std::cout << "Variable BodyEndianness:"
              << genericPortHeader->GetBodyEndianness()
              << std::endl;
    std::cout << "Variable PortIndex:"
              << genericPortHeader->GetPortIndex()
              << std::endl;
    std::cout << "Variable HeaderVersionMajor:"
              << genericPortHeader->GetHeaderVersionMajor()
              << std::endl;
    std::cout << "Variable HeaderVersionMinor:"
              << genericPortHeader->GetHeaderVersionMinor()
              << std::endl;
    // Getting members of  StaticPortHeader
    std::cout << "Variable RelaysPt1:"
              << staticPortHeader->GetRelaysPt1()
              << std::endl;
    std::cout << "Variable RelaysPt2:"
              << staticPortHeader->GetRelaysPt2()
              << std::endl;
}



.....
    ClientId clientId = 1024; // id of the client
    ReceiveTmTriggerListPortCallback callback =
                        std::bind(&ReceiveTmTriggerListPortClbk,
                        std::placeholders::_1);
    if(ERROR_CODE_OK != comDataServ->RegisterTmTriggerListPortReceiveCallback(clientId,callback)
    {
        std::cout << " Failed to register TmTriggerListPort port callback" << std::endl;
    }
```

For a more detailed API description, please see Appendix A.

7 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.

A COMMUNICATION DATA SERVICE API

A.1 **TargetListPort**
Description: Provides a list of radar targets.
The object TargetListPort provides the following APIs:

```
std::shared_ptr<GenericPortHeader> GetGenericPortHeader() const;
```

Returns pointer to GenericPortHeader object, whose access functions are described below:
A.1.1 **GenericPortHeader**
Description: No description available
The object GenericPortHeader provides the following APIs:

```
uint32_t GetPortId() const;
```

Returns value of PortId of uint32_t data type.
PortId - Port identification number.

```
int16_t GetPortVersionMajor() const;
```

Returns value of PortVersionMajor of int16_t data type.
PortVersionMajor - Major version of the port API.

```
int16_t GetPortVersionMinor() const;
```

Returns value of PortVersionMinor of int16_t data type.
PortVersionMinor - Minor version of the port API.

```
uint64_t GetTimestamp() const;
```

Returns value of Timestamp of uint64_t data type.
Timestamp - Time of creation of the port.

```
uint32_t GetPortSize() const;
```

Returns value of PortSize of uint32_t data type.
PortSize - Size of the port including this header.

```
uint8_t GetBodyEndianness() const;
```

Returns value of BodyEndianness of uint8_t data type.
BodyEndianness - Endianness of the sensor system.Please notice that the data will be presented in the host endianness, and should not be reversed.

```
uint8_t GetPortIndex() const;
```

Returns value of PortIndex of uint8_t data type.
PortIndex - The port index is used for multiple occurance of ports with the same port identifier.

```
uint8_t GetHeaderVersionMajor() const;
```

Returns value of HeaderVersionMajor of uint8_t data type.
HeaderVersionMajor - Major version of the generic port API.

```
uint8_t GetHeaderVersionMinor() const;
```

Returns value of HeaderVersionMinor of uint8_t data type.
HeaderVersionMinor - Minor version of the generic port API.

```
std::shared_ptr<StaticPortHeader> GetStaticPortHeader() const;
```

Returns pointer to StaticPortHeader object, whose access functions are described below:
A.1.2 **StaticPortHeader**
Description: Static header of target list port
The object StaticPortHeader provides the following APIs:

```
uint16_t GetNumberOfTargets() const;
```

Returns value of NumberOfTargets of uint16_t data type.
NumberOfTargets - Provides the number of target in the port

```
const std::vector<std::shared_ptr<Target>>& GetTargetList() const;
```

Returns pointer to array of Target objects, whose access functions are described below:
A.1.3 **Target**
Description: Represents a single target
The object Target provides the following APIs:

```
float32_t GetRange1() const;
```

Returns value of Range1 of float32_t data type.
Range1 - Target range

```
float32_t GetSigmaRange() const;
```

Returns value of SigmaRange of float32_t data type.
SigmaRange - The standard deviation for range

```
float32_t GetAZAngleDeg() const;
```

Returns value of AZAngleDeg of float32_t data type.
AZAngleDeg - Azimuth angle from the target

```
float32_t GetSigmaAngle() const;
```

Returns value of SigmaAngle of float32_t data type.
SigmaAngle - The standard deviation for azimuth angle

```
float32_t GetSpeedRadial() const;
```

Returns value of SpeedRadial of float32_t data type.
SpeedRadial - Target speed

```
float32_t GetSigmaSpeed() const;
```

Returns value of SigmaSpeed of float32_t data type.
SigmaSpeed - The standard deviation for speed

```
float32_t GetRCS() const;
```

Returns value of RCS of float32_t data type.
RCS - The Radar Cross Section from the target

```
float32_t GetAmplitude() const;
```

Returns value of Amplitude of float32_t data type.
Amplitude - Amplitude from the target

```
float32_t GetTgtNoise() const;
```

Returns value of TgtNoise of float32_t data type.
TgtNoise - Noise from the target

```
float32_t GetElevation() const;
```

Returns value of Elevation of float32_t data type.
Elevation - Elevation angle from the target

```
float32_t GetSigmaAngleEl() const;
```

Returns value of SigmaAngleEl of float32_t data type.
SigmaAngleEl - The standard deviation for elevation angle

A.1 **ObjectListPort**

Description: Provides a list of tracked radar objects.
The object ObjectListPort provides the following APIs:

```
std::shared_ptr<GenericPortHeader> GetGenericPortHeader() const;
```

Returns pointer to GenericPortHeader object, whose access functions are described below:

A.1.1 **GenericPortHeader**

Description: No description available
The object GenericPortHeader provides the following APIs:

```
uint32_t GetPortId() const;
```

Returns value of PortId of uint32_t data type.
PortId - Port identification number.

```
int16_t GetPortVersionMajor() const;
```

Returns value of PortVersionMajor of int16_t data type.
PortVersionMajor - Major version of the port API.

```
int16_t GetPortVersionMinor() const;
```

Returns value of PortVersionMinor of int16_t data type.
PortVersionMinor - Minor version of the port API.

```
uint64_t GetTimestamp() const;
```

Returns value of Timestamp of uint64_t data type.
Timestamp - Time of creation of the port.

```
uint32_t GetPortSize() const;
```

Returns value of PortSize of uint32_t data type.
PortSize - Size of the port including this header.

```
uint8_t GetBodyEndianness() const;
```

Returns value of BodyEndianness of uint8_t data type.
BodyEndianness - Endianness of the sensor system.Please notice that the data will be presented in the host endianness, and should not be reversed.

```
uint8_t GetPortIndex() const;
```

Returns value of PortIndex of uint8_t data type.
PortIndex - The port index is used for multiple occurance of ports with the same port identifier.

```
uint8_t GetHeaderVersionMajor() const;
```

Returns value of HeaderVersionMajor of uint8_t data type.
HeaderVersionMajor - Major version of the generic port API.

```
uint8_t GetHeaderVersionMinor() const;
```

Returns value of HeaderVersionMinor of uint8_t data type.
HeaderVersionMinor - Minor version of the generic port API.

```
std::shared_ptr<StaticPortHeader> GetStaticPortHeader() const;
```

Returns pointer to StaticPortHeader object, whose access functions are described below:
A.1.2 **StaticPortHeader**
Description: No description available
The object StaticPortHeader provides the following APIs:

```
int GetRain() const;
```

Returns value of Rain of int data type.
Rain - rain flag

```
uint16_t GetNumberOfObjects() const;
```

Returns value of NumberOfObjects of uint16_t data type.
NumberOfObjects - number of tracked objects

```
int GetNumberOfMessages() const;
```

Returns value of NumberOfMessages of int data type.
NumberOfMessages - number of messages

```
float32_t GetCycleDuration() const;
```

Returns value of CycleDuration of float32_t data type.
CycleDuration - cycle duration [s]

```
int GetCycleCount() const;
```

Returns value of CycleCount of int data type.
CycleCount - cycle counter

```
int GetObjectData0Format() const;
```

Returns value of ObjectData0Format of int data type.
ObjectData0Format - object data0 format

```
int GetObjectData1Format() const;
```

Returns value of ObjectData1Format of int data type.
ObjectData1Format - object data1 format

```
const std::vector<std::shared_ptr<Object>>& GetObjectList() const;
```

Returns pointer to array of Object objects, whose access functions are described below:
A.1.3 **Object**
Description: Represents a single tracked object
The object Object provides the following APIs:

```
float32_t GetXPoint1() const;
```

Returns value of XPoint1 of float32_t data type.
XPoint1 - x component of the position of the object, depends on selectedReferencePoint

```
float32_t GetYPoint1() const;
```

Returns value of YPoint1 of float32_t data type.
YPoint1 - y component of the position of the object, depends on selectedReferencePoint

```
float32_t GetHeadingDeg() const;
```

Returns value of HeadingDeg of float32_t data type.
HeadingDeg - heading of the object [-pi , pi]

```
float32_t GetObjectLen() const;
```

Returns value of ObjectLen of float32_t data type.
ObjectLen - length of the object

```
float32_t GetSpeedAbs() const;
```

Returns value of SpeedAbs of float32_t data type.
SpeedAbs - absolute object speed [m/s], in direction of the object heading

```
uint16_t GetObjectID() const;
```

Returns value of ObjectID of uint16_t data type.
ObjectID - unique tag number for identification of the object

A.1 **TmTriggerListPort**

Description: Represents the triggered virtual relays
The object TmTriggerListPort provides the following APIs:

```
std::shared_ptr<GenericPortHeader> GetGenericPortHeader() const;
```

Returns pointer to GenericPortHeader object, whose access functions are described below:

A.1.1 **GenericPortHeader**

Description: No description available
The object GenericPortHeader provides the following APIs:

```
uint32_t GetPortId() const;
```

Returns value of PortId of uint32_t data type.
PortId - Port identification number.

```
int16_t GetPortVersionMajor() const;
```

Returns value of PortVersionMajor of int16_t data type.
PortVersionMajor - Major version of the port API.

```
int16_t GetPortVersionMinor() const;
```

Returns value of PortVersionMinor of int16_t data type.
PortVersionMinor - Minor version of the port API.

```
uint64_t GetTimestamp() const;
```

Returns value of Timestamp of uint64_t data type.
Timestamp - Time of creation of the port.

```
uint32_t GetPortSize() const;
```

Returns value of PortSize of uint32_t data type.
PortSize - Size of the port including this header.

```
uint8_t GetBodyEndianness() const;
```

Returns value of BodyEndianness of uint8_t data type.
BodyEndianness - Endianness of the sensor system.Please notice that the data will be presented in the host endianness, and should not be reversed.

```
uint8_t GetPortIndex() const;
```

Returns value of PortIndex of uint8_t data type.
PortIndex - The port index is used for multiple occurance of ports with the same port identifier.

```
uint8_t GetHeaderVersionMajor() const;
```

Returns value of HeaderVersionMajor of uint8_t data type.
HeaderVersionMajor - Major version of the generic port API.

```
uint8_t GetHeaderVersionMinor() const;
```

Returns value of HeaderVersionMinor of uint8_t data type.
HeaderVersionMinor - Minor version of the generic port API.

```
std::shared_ptr<StaticPortHeader> GetStaticPortHeader() const;
```

Returns pointer to StaticPortHeader object, whose access functions are described below:
A.1.2 **StaticPortHeader**
Description: No description available
The object StaticPortHeader provides the following APIs:

```
uint32_t GetRelaysPt1() const;
```

Returns value of RelaysPt1 of uint32_t data type.
RelaysPt1 - Triggered relays 0...31

```
uint32_t GetRelaysPt2() const;
```

Returns value of RelaysPt2 of uint32_t data type.
RelaysPt2 - Triggered relays 32...63

B USER INTERFACE INSTRUCTIONS TM VERSION 5.2.0

B.1 Parameter Section tm_interface_0dim

user interface parameters

Parameter Name	can_active
Description	[1] 0 = Interface Listen (RX only), 1 = Interface Active
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	1

Parameter Name	rs485_active
Description	[1] 0 = Interface Listen (RX only), 1 = Interface Active
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	eth_active
Description	[1] 0 = Interface Listen (RX only), 1 = Interface Active
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	posX
Description	[m] X position of the sensor in Cartesian coordinate system
Data Type	f32
Dimensions	None
Access	RW
Default	0.0
Min	-300.0
Max	300.0

Parameter Name	posY
Description	[m] Y position of the sensor in Cartesian coordinate system
Data Type	f32
Dimensions	None
Access	RW
Default	0.0
Min	-300.0
Max	300.0

Parameter Name	posZ
Description	[m] Z position of the sensor in Cartesian coordinate system
Data Type	f32
Dimensions	None
Access	RW
Default	5.0
Min	-20.0
Max	20.0

Parameter Name	xyOrientation
Description	[deg] Orientation of the sensor in Cartesian coordinate system in X-Y plane (azimuth angle), valid interval (-180.f, 180.f], 0.f for positions on the Z axis
Data Type	f32
Dimensions	None
Access	RW
Default	0.0
Min	-180.0
Max	180.0

Parameter Name	xzOrientation
Description	[deg] Orientation of the sensor in Cartesian coordinate system in X-Z plane (elevation angle), valid interval [-90.f, 90.f], 0.f for the origin of the coordinate system
Data Type	f32
Dimensions	None
Access	RW
Default	0.0
Min	-90.0
Max	90.0

Parameter Name	yzOrientation
Description	[deg] Orientation of the sensor in Cartesian coordinate system in Y-Z plane (roll angle) (0.f: not turned, 180.f: turned upside-down), valid interval (0.f, 180.f]
Data Type	f32
Dimensions	None
Access	RW
Default	0.0
Min	-180.0
Max	180.0

Parameter Name	waveform_selector
Description	[1] Select waveform index: WI0: WF0+FB1, WI1: WF0+FB2, WI2: WF0+FB3, WI3: WF1+FB1, WI4: WF2+FB1
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	3

Parameter Name	prf_set_selector
Description	[1] Select PRF set for auto switching
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	1

Parameter Name	prf_staggering_active
Description	[1] PRF switching: 0 = inactive, 1 = active
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	tx_power_selector
Description	[1] tx power selctor between low (=0) and high (=1) tx power
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	mute
Description	[1] 0 = normal mode, 1 = TX signal muted
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	1

Parameter Name	min_cycle_duration
Description	[ms] minimum cycle duration
Data Type	u16
Dimensions	None
Access	RW
Default	75
Min	50
Max	250

Parameter Name	interference_blanking_enable
Description	Enable/Disable the interference blanking. 0 = disabled, 1 = enabled.
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	sbd_enable
Description	Enable/Disable the sensor blind detection. 0 = disabled, 1 = enabled.
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	rain_detection_enable
Description	Enable/Disable the rain detection. 0 = disabled, 1 = enabled.
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	output_control_target_list
Description	[1] output data control, 0 = disabled, 1 = enabled
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	1

Parameter Name	output_control_object_list
Description	[1] output data control, 0 = disabled, 1 = enabled
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	output_control_triggers
Description	[1] output data control, 0 = disabled, 1 = enabled
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	output_control_statistics
Description	[1] output data control, 0 = disabled, 1 = enabled
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	output_control_pvr
Description	[1] output data control, 0 = disabled, 1 = enabled
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	output_control_queue_length
Description	[1] output data control, 0 = disabled, 1 = enabled
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	1

Parameter Name	max_nof_targets
Description	[1] maximum number of detected targets
Data Type	u8
Dimensions	None
Access	RW
Default	64
Min	0
Max	128

Parameter Name	max_nof_objects
Description	[1] maximum number of tracked objects
Data Type	u8
Dimensions	None
Access	RW
Default	32
Min	0
Max	64

Parameter Name	trigger_output_mode
Description	[1] defines trigger output mode, 0 = no output, 1= trigger header output
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	1
Max	1

Parameter Name	statistics_output_mode
Description	[1] defines statistics output mode, 1 = statistics header output, 2 = header + objects
Data Type	u8
Dimensions	None
Access	RW
Default	2
Min	1
Max	2

Parameter Name	pvr_output_mode
Description	[1] defines PVR output mode 1 = PVR header output, 2 = PVR header + objects
Data Type	u8
Dimensions	None
Access	RW
Default	2
Min	1
Max	2

Parameter Name	ethernet_target_client_id
Description	[1] client id of the ethernet target
Data Type	u32
Dimensions	None
Access	RW
Default	16777217
Min	1

Parameter Name	dhcp_enable
Description	enables dhcp
Data Type	u8
Dimensions	None
Access	RW
Default	0

Parameter Name	ip_source_address
Description	IP source address (32bit)
Data Type	u32
Dimensions	None
Access	RW
Default	3232238347

Parameter Name	control_port
Description	incoming port
Data Type	u16
Dimensions	None
Access	RW
Default	55555

Parameter Name	subnetmask_address
Description	Subnetmask address (32bit)
Data Type	u32
Dimensions	None
Access	RW
Default	4294967040

Parameter Name	gateway_address
Description	Gateway address (32bit)
Data Type	u32
Dimensions	None
Access	RW
Default	3232238337

Parameter Name	fault_handler_behaviour
Description	[1] What should be done in case of an error? stop sensor (=0) or restart sensor, w/o any report (=1)
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	1

Parameter Name	failsafe_mode
Description	activate failsafe mode. Bit0: rain failsafe
Data Type	u8
Dimensions	None
Access	RW
Default	17
Min	0
Max	255

Parameter Name	failsafe_relays_pt1
Description	defines which relays (0...31) should be active in failsafe mode
Data Type	u32
Dimensions	None
Access	RW
Default	4294967295
Min	0
Max	4294967295

Parameter Name	failsafe_relays_pt2
Description	defines which relays (32...63) should be active in failsafe mode
Data Type	u32
Dimensions	None
Access	RW
Default	4294967295
Min	0
Max	4294967295

Parameter Name	interference_threshold
Description	interference_threshold threshold
Data Type	u32
Dimensions	None
Access	RW
Default	300
Min	0
Max	4294967295

Parameter Name	failsafe_hysteresis
Description	Failsafe hysteresis timer [s]
Data Type	u16
Dimensions	None
Access	RW
Default	30
Min	0
Max	300

Parameter Name	interference_delay_cycles
Description	interference delay cycles for Failsafe activation[s]
Data Type	u16
Dimensions	None
Access	RW
Default	10
Min	0
Max	10000

Parameter Name	blind_delay_cycles
Description	hysteresis timer for Failsafe activation[s]
Data Type	u16
Dimensions	None
Access	RW
Default	100
Min	0
Max	10000

Parameter Name	simulation_mode
Description	[1] 0: no object simulation, 1: object simulation on lanes, 2: object simulation on splines
Data Type	u8
Dimensions	None
Access	RW
Default	0
Max	2

Parameter Name	sim_nof_lanes
Description	[1] number of lanes for lane simulation mode
Data Type	u8
Dimensions	None
Access	RW
Default	4
Max	64

Parameter Name	sim_objects_per_lane
Description	[1] number of simulated object per lane/spline
Data Type	u8
Dimensions	None
Access	RW
Default	5
Max	64

Parameter Name	target_simulation
Description	[1] target simulation mode. (simulation mode 1) 0=off; 1=on
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	1

Parameter Name	nof_lanes
Description	[1] Number of simulated lanes.
Data Type	u16
Dimensions	None
Access	RW
Default	2
Min	1
Max	6

Parameter Name	targets_per_lane
Description	[1] Number of targets per lane.
Data Type	u16
Dimensions	None
Access	RW
Default	2
Min	1
Max	10

Parameter Name	relay_simulation
Description	[1] relay simulation mode. (simulation mode 4) 0=off; 1=submode 1 (static, depending on active_relays parameters); 2=submode 2 (knight rider)
Data Type	u16
Dimensions	None
Access	RW
Default	0
Min	0
Max	2

Parameter Name	active_relays_part1
Description	[1] defines active relays (bit-coded, relay 1-32) for relay simulation submode 1
Data Type	u32
Dimensions	None
Access	RW
Default	1
Min	0
Max	4294967295

Parameter Name	active_relays_part2
Description	[1] defines active relays (bit-coded, relay 33-64) for relay simulation submode 1
Data Type	u32
Dimensions	None
Access	RW
Default	0
Min	0
Max	4294967295

Parameter Name	nof_simulated_relays
Description	[1] defines the number of simulated relays for relay simulation submode 2
Data Type	u8
Dimensions	None
Access	RW
Default	8
Min	0
Max	64

Parameter Name	interval_time
Description	used interval time [s] for calculating statistics
Data Type	f32
Dimensions	None
Access	RW
Default	300
Min	0
Max	3600

Parameter Name	statistics_flags
Description	used statistic flags. bit field: 0x0000001: speed_scale_unit
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	1

Parameter Name	exclusive_count
Description	defines if an object should be counted only on the first zone (value 1) or on all zones where the object was detected (value 0)
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	1

Parameter Name	occupancy_mode
Description	defines occupancy mode. Value 0 == without classification, 1 == with classification
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	8

Parameter Name	count_mode
Description	defines counting mode. Value 0 == without classification, 1 == with classification
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	8

Parameter Name	av_speed_mode
Description	defines av. speed mode. Value 0 == without classification, 1 == with classification
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	8

Parameter Name	85th_perc_speed_mode
Description	defines 85th perc. speed mode. Value 0 == without classification, 1 == with classification
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	8

Parameter Name	synchronize_statistics
Description	synchronized statistics output
Data Type	u8
Dimensions	None
Access	RW
Default	0
Min	0
Max	1

Parameter Name	polling_mode
Description	statistics polling mode. Value 0 == off, 1== send currently collected statistics.
Data Type	u8
Dimensions	None
Access	RW
Default	1
Min	0
Max	2

Parameter Name	85th_perc_classes
Description	number of used 85th perc speed classes
Data Type	u8
Dimensions	None
Access	RW
Default	9
Min	1
Max	9

Parameter Name	queueDistTwoObjects
Description	the maximum distance between two cars in a queue
Data Type	f32
Dimensions	None
Access	RW
Default	30.0
Min	0.0

Parameter Name	queueDistFirstObject
Description	the maximum distance for the first car in the queue from the beginning of the zone
Data Type	f32
Dimensions	None
Access	RW
Default	20.0
Min	0.0

B.2 Status Section tm_status

customer status section

Status Name	tm_interface_version_major
Description	TM interface version major. Increased, if new version is not totally backward compatible.
Data Type	i32
Dimensions	None
Access	R

Status Name	tm_interface_version_minor
Description	TM interface version minor. Increased, if parameters or statuses are changed or added. The new version is still backward compatible.
Data Type	i32
Dimensions	None
Access	R

Status Name	sw_generation
Description	TM Software Version generation
Data Type	u16
Dimensions	None
Access	R

Status Name	sw_version_major
Description	TM Software Version major
Data Type	u16
Dimensions	None
Access	R

Status Name	sw_version_minor
Description	TM Software Version minor
Data Type	u16
Dimensions	None
Access	R

Status Name	sw_version_patch
Description	TM Software Version patch
Data Type	u16
Dimensions	None
Access	R

Status Name	customer_id
Description	Customer Identifier
Data Type	u16
Dimensions	None
Access	R

Status Name	antenna_type
Description	antenna type
Data Type	u16
Dimensions	None
Access	R

Status Name	config_code
Description	configuration code
Data Type	u16
Dimensions	None
Access	R

Status Name	product_serial
Description	32Bit product id serial
Data Type	u32
Dimensions	None
Access	R

Status Name	product_gen
Description	product generation
Data Type	u32
Dimensions	None
Access	R

Status Name	product_mod_high
Description	product modification high
Data Type	u32
Dimensions	None
Access	R

Status Name	product_mod_low
Description	product modification low
Data Type	u32
Dimensions	None
Access	R

Status Name	product_rev
Description	product revision
Data Type	u32
Dimensions	None
Access	R

Status Name	v_major
Description	Major version of the app_tm module
Data Type	i32
Dimensions	None
Access	R

Status Name	v_minor
Description	Minor version of the app_tm module
Data Type	i32
Dimensions	None
Access	R

Status Name	v_patch
Description	Patch version of the app_tm module
Data Type	i32
Dimensions	None
Access	R

Status Name	build_stage
Description	Build stage of the app_tm module as ASCII code of the corresponding character: 85='U' (untested), 84='T' (test), 67='C' (release candidate), 82='R' (release)
Data Type	u8
Dimensions	None
Access	R

Status Name	variant
Description	Variant of the app_tm module
Data Type	u8
Dimensions	None
Access	R

Status Name	systime_low_dword
Description	time since the system timer was started in micro seconds (low dword)
Data Type	u32
Dimensions	None
Access	R

Status Name	systime_high_dword
Description	time since the system timer was started in micro seconds (high dword)
Data Type	u32
Dimensions	None
Access	R

Status Name	parameter_check_valid
Description	parameter settings for radar control are valid (=1) or not (=0)
Data Type	u8
Dimensions	None
Access	R

Status Name	acc_roll_angle_low_pass
Description	accelerometer angle of roll, 0 = accelerometer not calibrated
Data Type	f32
Dimensions	None
Access	R

Status Name	acc_pitch_angle_low_pass
Description	accelerometer angle of pitch, 0 = accelerometer not calibrated
Data Type	f32
Dimensions	None
Access	R

Status Name	acc_calib_status
Description	accelerometer indication if calibration is complete
Data Type	u32
Dimensions	None
Access	R

Status Name	rfe_temperature
Description	Radar Frontend temperature in deg C, 0 = temperature not available.
Data Type	u16
Dimensions	None
Access	R

Status Name	tx_power
Description	set tx power
Data Type	f32
Dimensions	None
Access	R

Status Name	nof_tracking_classes
Description	number of tracking classes
Data Type	u8
Dimensions	None
Access	R

Status Name	region_code
Description	app TM region code: 0=EU, 1=CH, 2=US
Data Type	u16
Dimensions	None
Access	R

Status Name	tm_application
Description	app TM : 0=IS, 1=HW, 2=SE, 3=RL
Data Type	u8
Dimensions	None
Access	R

B.3 Status Section software_info

Software information

Status Name	software_info_version
Description	Status software_info version. 0: Software info not available
Data Type	u16
Dimensions	None
Access	R

Status Name	project_id
Description	Project identifier
Data Type	u16
Dimensions	None
Access	R

Status Name	customer_id
Description	Customer identifier
Data Type	u16
Dimensions	None
Access	R

Status Name	applet_id
Description	Applet identifier
Data Type	u32
Dimensions	None
Access	R

Status Name	software_version_digits
Description	Number of used digits for software version
Data Type	u16
Dimensions	None
Access	R

Status Name	software_version_1
Description	project specific software version, digit 1
Data Type	u16
Dimensions	None
Access	R

Status Name	software_version_2
Description	project specific software version, digit 2
Data Type	u16
Dimensions	None
Access	R

Status Name	software_version_3
Description	project specific software version, digit 3
Data Type	u16
Dimensions	None
Access	R

Status Name	software_version_4
Description	project specific software version, digit 4
Data Type	u16
Dimensions	None
Access	R

Status Name	software_version_5
Description	project specific software version, digit 5
Data Type	u16
Dimensions	None
Access	R

Status Name	software_version_6
Description	project specific software version, digit 6
Data Type	u16
Dimensions	None
Access	R

Status Name	software_version_7
Description	project specific software version, digit 7
Data Type	u16
Dimensions	None
Access	R

Status Name	software_version_8
Description	project specific software version, digit 8
Data Type	u16
Dimensions	None
Access	R

Status Name	software_version_9
Description	project specific software version, digit 9
Data Type	u16
Dimensions	None
Access	R

Status Name	software_build_stage
Description	Build stage of the project software
Data Type	u16
Dimensions	None
Access	R

Status Name	baseline_v_major
Description	Baseline software major version
Data Type	u16
Dimensions	None
Access	R

Status Name	baseline_v_minor
Description	Baseline software minor version
Data Type	u16
Dimensions	None
Access	R

Status Name	baseline_v_patch
Description	Baseline software patch version
Data Type	u16
Dimensions	None
Access	R

Status Name	baseline_build_stage
Description	Baseline software build stage
Data Type	u16
Dimensions	None
Access	R

Status Name	base_interface_id
Description	Base interface identification number
Data Type	u32
Dimensions	None
Access	R

Status Name	base_interface_v_major
Description	Base interface major version
Data Type	u32
Dimensions	None
Access	R

Status Name	base_interface_v_minor
Description	Base interface minor version
Data Type	u32
Dimensions	None
Access	R

Status Name	user_interface_id
Description	User interface identification number
Data Type	u32
Dimensions	None
Access	R

Status Name	user_interface_v_major
Description	User interface major version
Data Type	u32
Dimensions	None
Access	R

Status Name	user_interface_v_minor
Description	User interface minor version
Data Type	u32
Dimensions	None
Access	R