ST25RU3993

USB Protocol Documentation for Pc Applications

Confidential

Table of Contents

Revision History

Revision	Date	Originator	Description
0.7	24.01.2017		first Draft (uncontrolled)
0.8	31.01.2017		Reviewed
0.9	01.03.2018		Changes of review added
1.0	22.04.2018		Release V1.0

1 Introduction

USB is a serial bus system for connections between computer and external devices. HID is a device class of USB for PCs which is usually used for PC input devices (keyboard, mouse …). The ams framework is using HID for the communication between ams application boards and a PC. This document describes the USB communication StreamV2.

StreamV2 is a special implementation for USB communication between PCs and ams AG applications.

StreamV2 uses defined protocols. Chapter 2 gives a detailed description about the ams protocol architecture. The implementation for the PC-side is supported by the ams communication project which provides communication classes written in C++. The ams Communication Project will be described in Chapter 3. For the firmware side implementation, ams application boards use a microcontroller from Microchip. The used microcontroller supports USB and provides modules for the communication. ams has additional software modules for the adaption to the ams USB protocol and application firmware. Chapter 4 explains the firmware implementation in C. An example for setting up a communication for ams applications is given in Chapter 5.

For more information about the USB (Universal Serial Bus) Standard visit www.usb.org.

1.1 Purpose

This document describes the USB communication between a PC and ams application boards with a Microchip MCU using ams Streaming Classes.

1.2 Terms and Abbreviations

Term/ Abbreviation	Description
USB	Universal Serial Bus
HID	Human Interface Device
ASCII	American Standard Code for Information Interchange
HEX	Hexadecimal
GUI	Graphical User Interface
TID	Transaction ID
MCU	Microcontroller Unit
rx-prot	Buffer Size for the serialize Data
tx-prot	Buffer Size for the deserialize Data
LSB	Least Significant Byte
MSB	Most Significant Byte
LSb	Least Significant Bit
MSb	Most Significant Bit
IDE	Integrated Development Environment
0x	Values with 0x prefix must be interpreted as HEX-Values.
I2C	Inter-Integrated Circuit
SPI	Serial Peripheral Interface
PC	Personal Computer

2 Protocol Architecture

This chapter describes the format of the StreamV2 USB HID Protocol. Additionally, the assembling and disassembling of the protocol is described. A distinction is made between the transfer from host to device and from device to host.

2.1 Protocol from Host to Device

2.1.1 Protocol Summary

The following table describes the byte-stream of the protocol from host to device.

Table 2.1: Protocol Summary Host to Device

Byte	Content	Description
0	TID	Transaction ID, changes with every report send
1	Payload	Number of valid bytes in the current HID report (max. 64 bytes for HID)
2	Reserved	Reserved Byte
3	Protocol	Protocol Byte, defining the Command for the first protocol packet
4	tx-prot MSB	MSB for number of bytes to transmit in this protocol packet
5	tx-prot LSB	LSB for number of bytes to transmit in this protocol packet
6	rx-prot MSB	MSB for number of bytes expected to receive for this protocol packet
7	rx-prot LSB	LSB for number of bytes expected to receive for this protocol packet
8 …+tx-prot – 1	Data	Protocol Data to be sent to the Device
8 + tx-prot	Protocol B	Protocol Command for the second protocol packet B
9 + tx-prot	tx-prot B MSB	MSB for number of bytes to transmit in protocol packet B
10 + tx-prot	tx-prot B LSB	LSB for number of bytes to transmit in protocol packet B
11 + tx-prot	rx-prot B MSB	MSB for number of bytes expected as response to protocol packet B
12 + tx-prot	rx-prot B LSB	LSB for number of bytes expected as response to protocol packet B
13 + tx-prot B	Data	Protocol Data of packet B
…	…	…

Note:

- The tx-prot value consists of 2 bytes, the tx-prot MSB and the tx-prot LSB, but the maximum amount of data in a single packet is limited to 1024+64 bytes (defined in ams_stream.h). This is necessary as the device must handle a complete packet and the data memory on the MCU is limited.
- On the host side, the TID is generated as a 4-bit number counting from 0 to 0x0F, and then starting from 0 again.

The device side takes the TID received from the Host and moves it to the upper nibble, increments its own TID counter by one (range is again 0 to 0x0F) and inserts its own TID in the lower nibble.
At the device (txTID is the TID of the device, rxTID is the TID received from the host):
TID = (rxTID << 4) | (++txTID & 0xF)

2.1.2 Byte Stream Assembly

The generation of the byte stream at the PC is done in three steps.

Note: The last line in the tables is the byte index in the packet.

Step1: AmsComObject

The AmsComObject class provides the data in a single packet.

Protocol Data			
Data			
0	1	…	tx-size – 1

Step2: Stream Driver

The stream driver on the host side takes a single data packet and adds the protocol header.

Protocol Header					Protocol Data		
Protocol ID	Tx-Protocol		Rx-Protocol		Data		
0	1	2	3	4	5	…	4 + tx-size

Step3: HID Driver

The HID driver provides a communication channel and transmits the data. For this, the HID driver on the host side cuts the data from the stream driver into packets that fit into a HID report. A HID report has a maximum size of 64 bytes. These 64 bytes include also the HID driver header (that has a size of 3 bytes). So the total payload in one HID packet is 61 bytes.

The following section describes the various scenarios that can occur (e.g. packet fits in 1 HID report, packet is more than 61 bytes – need more than one HID report, several small packets fit in 1 HID report, etc.).

Data buffer fits in one HID Report:

The report will be filled up to 64 Bytes if necessary.

HID Driver Header			Protocol Packet(s)				
TID	Payload	Reserved	Data			Padding Data	
0	1	2	3	…	2+payload	…	63

Protocol Packet(s) need more than one HID Report

The payload of all HID Reports except the last report is 61(64 bytes – 3 header bytes).

As many reports as needed for transmitting the packet(s) are generated.

The first HID Report (Payload is set to 61):

HID Driver Header			Protocol Packet(s)				
TID	Payload	Reserved	Data				
0	1	2	3	…			63

The next HID Reports get a new TID (Payload is still set to 61).

HID Driver Header			Protocol Packet(s)				
New TID	Payload	Reserved	Data				
0	1	2	3	…			63

The Last Report looks the same like in the case where the protocol packet fits in 1 HID Report).

HID Driver Header			Protocol Packet(s)				
New TID	Payload	Reserved	Data			Padding Data	
0	1	2	3	…	2+payload	…	63

Several Protocol Packets fit in HID Report

In this example 2 packets fit exactly in 1 HID report (payload is set to 61). If the sum of the 2 packets is smaller, padding bytes are added after the second packet.

HID Driver Header			Protocol Packet(s)				
New TID	Payload	Reserved	1st Packet Data			2nd Packet Data	
0	1	2	3	…	n	…	63

2.2 Protocol from Device to Host

2.2.1 Protocol Summary

The following table describes the byte-stream of the protocol from device to host.

Table 2.2: Protocol Summary Device to Host

Byte	Content	Description
0	TID	Transaction ID, generated from the received report and the internal TID
1	Payload	Number of valid bytes in the current HID report (max. 64 bytes)
2	HID status	HID Report Status Byte
3	Protocol	Protocol Byte defining the Command for this protocol response packet
4	Reserved	Reserved Byte
5	Protocol Status	Status Byte for this protocol response packet
6	tx-prot MSB	MSB for number of bytes to transmit to the host in this protocol response packet
7	tx-prot LSB	LSB for number of bytes to transmit to the host in this protocol response packet
8	Data	Protocol Data to be send to the Host
8+tx-prot	Protocol B	Protocol Byte defining the Command for the second protocol packet B
…	…	…

Note:

- If a protocol was not processed (because e.g. the protocol id was unknown) the next HID packet that is sent back will contain a HID status byte unequal 0 indicating that an error occurred.
- The protocol status byte contains the information whether the command was successful or not successful executed.
- There is also a flag indicating if the command shall produce always a response. This is the AMS_COM_WRITE_READ_NOT flag. If this flag is set the firmware always produces a response packet (even if there is no data to be sent to the host). This response packet contains at least the status of the protocol.

2.2.2 Byte Stream Assembly

The generation of the byte stream in the firmware application is done in three steps.

Step1: Data Packet

The firmware application provides a single data packet containing the result.

Protocol Data			
Data			
0	1	…	tx-size - 1

Step2: Process Received Packets

The function processReceivedPackets in the file stream_dispatcher.c adds the protocol header. This header contains the protocol byte, the reserved and the status byte as well as the tx-prot 16-bit word (from the information provided by the firmware application).

Protocol Header					Protocol Data		
Protocol ID	Reserved	Status	Tx-Prot		Data		
0	1	2	3	4	5	…	4+tx-size

Step3: HID Driver

The HID driver on the firmware side splits the buffer into packets with a maximum size of 64 bytes and adds for each HID Report a HID driver header. This is done in the following way.

Data buffer fits in one HID Report:

The report will be filled with padding up to 64 bytes if necessary.

HID Driver Header			Protocol Packet(s)				
TID	Payload	Status	Data			Padding Data	
0	1	2	3	…	2+payload	…	63

Protocol Packet(s) need more than one HID-report

The payload of all HID-reports except the last report is 61(64 Bytes – 3Header Bytes).

As many reports as needed for transmitting the packet(s) are generated.

First HID Report (payload is set to 61):

HID Driver Header			Protocol Packet(s)				
TID	Payload	Status	Data				
0	1	2	3	…			63

The next HID Reports (payload is still set to 61):

HID Driver Header			Protocol Packet(s)				
New TID	Payload	Status	Data				
0	1	2	3	…			63

The last Report looks similar to the case where the packet fitted into 1 HID Report.

HID Driver Header			Protocol Packet(s)				
TID	Payload	Status	Data			Padding Data	
0	1	2	3	…	2+payload	…	63

Several Protocol Packets fit in HID Report

In this example 2 packets fit exactly in 1 HID report (payload is set to 61). If the sum of the 2 packets is smaller, padding bytes are added after the second packet.

HID Driver Header			Protocol Packet(s)				
New TID	Payload	Reserved	1st Packet Data			2nd Packet Data	
0	1	2	3	…	n	…	63

2.3 Predefined Protocol IDs

The currently predefined commands shown in the following table can also be found in ams_stream.h.

Table 2.3: Protocol IDs

Value	Define
0x80	flag: AMS_COM_WRITE_READ_NOT
0x60	Protocol: AMS_COM_CONFIG
0x61	Protocol: AMS_COM_I2C
0x62	Protocol: AMS_COM_I2C_CONFIG
0x63	Protocol: AMS_COM_SPI
0x64	Protocol: AMS_COM_SPI_CONFIG
0x65	Protocol: AMS_COM_CTRL_CMD_RESET
0x66	Protocol: AMS_COM_CTRL_CMD_FW_INFORMATION
0x67	Protocol: AMS_COM_CTRL_CMD_FW_NUMBER
0x68	Protocol: AMS_COM_WRITE_REG
0x69	Protocol: AMS_COM_READ_REG
0x6B	Protocol: AMS_COM_CTRL_CMD_ENTER_BOOTLOADER
0x7F	Protocol: AMS_COM_FLUSH

Note:

– The following number range is special:
0x60 - 0x7F: reserved for generic commands (part of the ams common firmware)
0x60: is a configuration protocol for the firmware itself
0x6B: is reserved for the Bootloader

0x7F: is reserved for the flush
An application can use the numbers: 0x00 - 0x5F for its own commands.

- Protocol-Rules:
  The MSB of the protocol byte defines whether a response must be sent or not sent from the firmware. This is useful for write commands to which you want in some cases a status response.
- The Enter Bootloader become 0xEB because:
  AMS_COM_WRITE_READ_NOT | AMS_COM_CTRL_CMD_ENTER_BOOTLOADER == 0x80 | 0x6B = 0xEB
- Communication Error Responses:
  For the communication error handling communication responses listed in the following table are implemented.

Table 2.4: Communication Error Responses

Value	Define
0x00	AMS_STREAM_NO_ERROR
0x01	AMS_STREAM_UNHANDLED_PROTOCOL
0x02	AMS_STREAM_PROTOCOL_FAILED

3 Implementation for PC Host

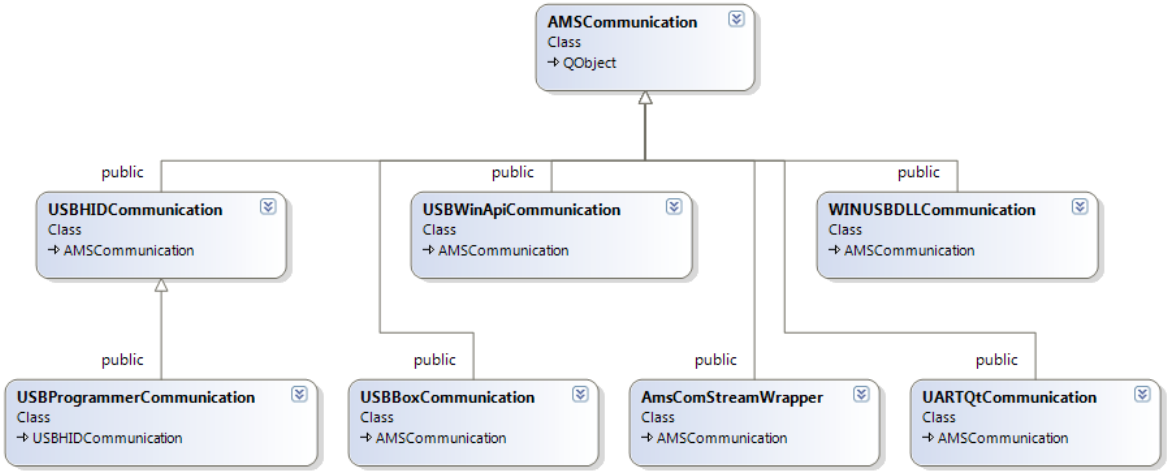
The StreamV2 communication on the PC-side is part of the ams Communication Project. This chapter gives an overview of the ams Communication project, the StreamV2 communication classes and the necessary steps to set up a StreamV2 communication. The implementation is done in C++.

3.1 The ams Communication Project

The ams Communication project contains all ams classes for communication between a PC host and external devices with certain interfaces. Implementations for USB and an implementation for Uart exist at the moment. All communication classes are derived from AMSCommunication class.

AMSCommunication is an abstract class used for abstraction of different communication interfaces between the PC and the ams demo boards.

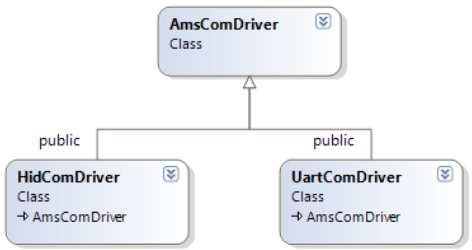
Figure 3.1: Class Diagram of AMSCommunication Classes



3.2 Communication Classes

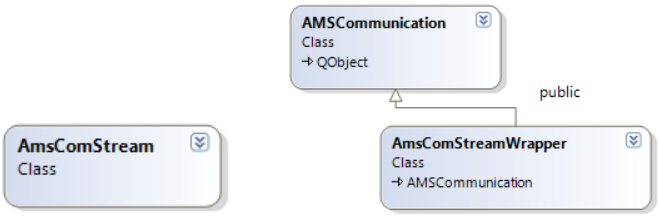
This section lists the driver classes, the stream communication classes and the Communication object classes.

Figure 3.2: Class Diagram of Driver Classes



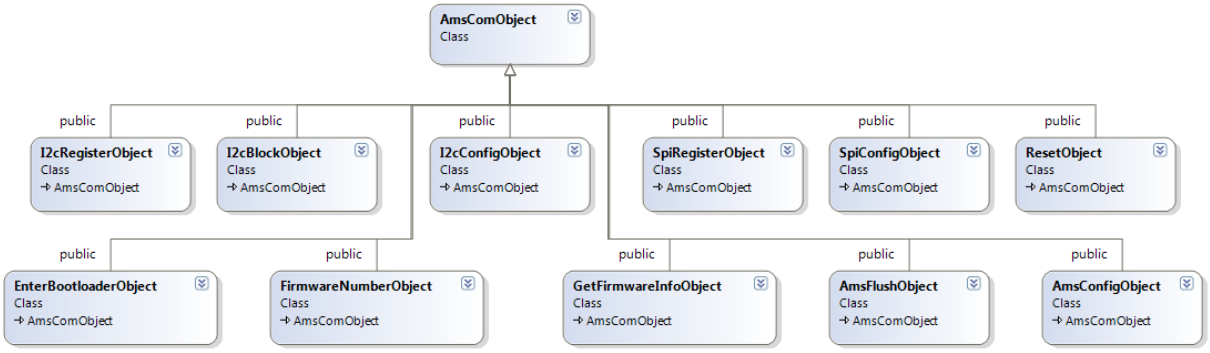
AmsComDriver	This is the base class of all stream communication drivers.
HidComDriver	This is the communication class for HID streaming communication. The HidComDriver is a derived class from AmsComDriver.
UartComDriver	Communication class for UART streaming communication.

Figure 3.3: Class Diagram of Stream Communication Classes



AmsComStream	This class provides a communication stream that can transmit and receive objects that are derived from class AmsComObject. The communication stream itself takes an instance of class AmsComDriver for the transport of the AmsComObjects.
AmsComStreamWrapper	The Stream Wrapper is a derived class of AMSCommunication, so that it fulfills the interface needed by the current version of the register map. The AmsComStreamWrapper class contains as member an instance of class AmsComStream. It uses this instance for communication.

Figure 3.4: Class Diagram of Communication Object Classes



AmsComObject	Base class of all classes that implement objects that can be transmitted and received using the AmsComStream class.
I2cRegisterObject	The class to read and write registers using I2C.
I2CBlockObject	Class to read or write a block of registers from a device using I2C.
I2CConfigObject	Configuration object used for I2C.
SpiRegisterObject	Class to read and write registers using SPI.

SpiConfigObject	Configuration object used for SPI.
ResetObject	Class to reset MCU or peripherals.
EnterBootloaderObject	The class to enter the Bootloader code at the MCU.
FirmwareNumberObject	The class to read out the firmware version number.
GetFirmwareInfoObject	The class to read out the firmware description string.
AmsFlushObject	The AmsFlushObject is used to trigger a flush in the AmsComStream class.
AmsConfigObject	Reserved for future use.

3.3 Set up a Communication

This section describes the necessary steps to set up a communication. An example is given in chapter 5.

- Set up the Streaming Communication

The VID (Vendor ID) and the PID (Product ID) of the device must be declared at first. In Windows it can be found in Devices and Printers of the Control Panel. The property “Hardware IDs” shows the VID and the PID of the corresponding device. The ams VID is 0x1325 and already defined in HidComDriver.h.

The AMSDeviceDetector is used in the GUI, to recognize that the right USB device is connected or disconnected. The device must be registered with the function “registerForHIDDevice”.

The HidComDriver class provides the communication channel for the stream communication. The VID and PID must be assigned to the driver.

The AmsComStream class provides the functionality for streaming data packets. For this purpose it uses an instance of class AmsComDriver (e.g. the HidComDriver).

The AmsComStreamWrapper combines the functionality of the HidComDriver and the AmsComStream classes and fulfills the AMSCommunication class interface.

- Creation of AmsComObjects

Several predefined objects like the EnterBootloaderObject can be used. If no existing object fulfills the required criteria’s, a new object has to be created. This new object must be derived from AmsComObject. Every new object needs its own unique protocol ID.

For a new object it is necessary to implement the functions “serialise”, “rxSerialise” and “deserialise”. These functions take care of reading out the data from the object and filling a protocol buffer which is passed to the function as parameter (or vice versa). The function “serialise” fills the buffer with the outgoing data. The function “deserialise” interprets the incoming buffer data. The function “rxSerialise” fills the buffer with the outgoing data necessary to receive data from the device (e.g. to read a register via I2C it is necessary to write an I2C command before reading back the value).

- Communication using AmsComObjects

To transmit an object, use either the function named “tx” or the operator “<<”. To receive an object, use either the function named “rx” or the operator “>>”. The default stream does an automatic flush of every object. The flush triggers an immediate transmission of the current data.

To improve the data throughput “late flush” can be used. In this case AMS_FLUSH has to be called explicitly. When using the “late flush” several small packets can be packed into 1 HID report, or 1 Uart packet.

AMS_FLUSH is the shortcut to instantiate “late flush”.

4 Implementation for Microchip Device Firmware

Application boards of ams frequently use a microcontroller of Microchip. MPLAB is an IDE for firmware development, provided by Microchip. The StreamV2 implementation on the firmware side is done with the programming language C.

4.1 USB and Stream Modules

Microchip provides modules for USB communication. Modules which must be included in ams firmware projects are usb.h, usb_ch9.h, usb_function_generic.h, usb_common.h, usb_config.h, usb_device.h, usb_function_hid.h, usb_hal.h, usb_hal_pic24.h. All these files can be found in the directory: common/firmware/microchip/include or in common/include.

The following additional modules from ams are available to support the access to USB.

Table 4.1: USB Stream Modules of ams

Modules	Description
ams_stream.h	Contains the constants which are shared between GUI and firmware concerning the ams streaming communication.
stream_dispatcher.h	Interface for stream packet handling.
stream_driver.h	Streaming driver interface declarations. The defines allow switching between different stream drivers. USB and UART are currently implemented.
usb_hid_stream_driver.h	USB HID streaming driver declarations.
weak_stream_functions.h	A weak implementation of the functions needed by the stream_dispatcher.c file. If you provide your own implementation yours will supersede the weak functions.

4.2 Streaming Implementation

In ams firmware programs the function ProcessIO (stream_dispatcher.c) is called to accomplish USB communication. This function must be called cyclically in a running program. The sequence of ProcessIO is illustrated in the following flow chart.

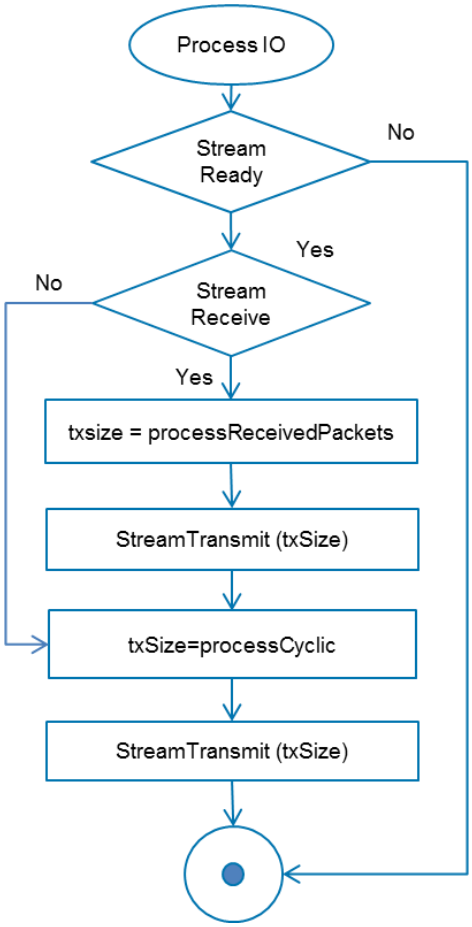
In the first step, the function checks if the microcontroller is ready for USB communication. If not, the function will return immediately.

When an USB HID report is received the function processReceivedPackets is called. This function does the de-/fragmentation into protocol packets. A single protocol packet is executed based on the protocol byte. E.g. for an “I2C Command” the function handleI2c is called, for a “Read Register Command” the function applReadReg is called, and so on. Each function takes care of packet data interpretation according to its own protocol structure. Self-defined protocol commands (the protocol byte is within the range 0x00 - 0x5F) will automatically call the function applProcessCmd. This function has to be implemented by the user.

After processReceivedPackets was executed, the StreamTransmit function is executed to transmit data from firmware to PC.

For applications that have data to send without receiving packets the function ProcessCyclic exists. This function is called periodically and checks if any data has to be sent. This is done by calling the function: applProcessCyclic. This function must be implemented by yourself if you need to transmit data without receiving first data. E.g. you want to send heartbeat information every other second. Data is again sent by using the StreamTransmit function.

Figure 4.1: Flow Chart Process IO



4.3 Implementation of Application Commands in firmware

– Register the Device for USB recognition

The PID and VID of the application have to be defined in the Device Descriptor Constants (usb_descriptors.c).

– Application Commands

The predefined commands listed in chapter 2.3 (or ams_stream.h) are already implemented. The calling of your own protocol commands has to be done in the function applProcessCmd or applProcessCyclic.

– Incoming and Outgoing Data

Access to incoming data is done by using the parameter rxData and access to outgoing data is done by using the parameter txData. The parameter rxSize holds the size of the incoming data, and txSize is set to the size of the outgoing buffer and must be set by the application to the size of the data to be transmitted.

5 Example

This example will show how to setup a StreamV2 communication in the GUI and in the Microchip firmware. Additionally, the implementation of the protocol command “readRegister” for a certain device will be explained. Subchapter 5.1 shows the byte stream handed to the low-level USB driver for this command. Subchapter 5.2 describes the GUI implementation and subchapter 5.3 the firmware implementation for the StreamV2 communication.

5.1 Byte Stream

The Microchip HID-protocol of a “Read Register Command” is described. All data are given in HEX encoding.

HID Protocol Transmitted from PC to Device: 05 06 00 02 00 01 00 01 03

HID Driver Header: 05 06 00

05	TID
06	Payload; the Protocol Packet contains 6 Bytes including the protocol header
00	Reserved

Protocol Header: 02 00 01 00 01

Protocol Data: 03

02	Protocol Id for Read Register
00	MSB tx-prot = Send Data Size MSB
01	LSB tx-prot = Send Data Size LSB (1 Byte for transmitting the Register Address)
00	MSB rx-prot = Read Data Size MSB
01	LSB rx-prot = Read Data Size LSB (1 Byte for receiving the Register Value)
03	Data itself = Register address 03

HID Protocol Transmitted from Device to PC: 53 06 00 02 00 00 00 01 5a

HID Driver Header:

53	TID
06	Payload
00	HID Status (AMS_STREAM_NO_ERROR)

Protocol Header: 02 00 00 00 01

Protocol Data: 5a

02	Protocol Id for Read Register
00	Reserved
00	Status (no error)
00	tx-prot MSB
01	tx-prot LSB
5a	Data itself = Value of Register 03 is 5a

5.2 GUI Implementation

This example shows how to create a HID streaming communication. Also a register read command will be implemented. For this, a communication class which is derived from the AmsComStreamWrapper will be generated.

- **Registration of the ams Application**

The PID of the application has to be defined

```
#define AMS_EXAMPLE_PID   0xD003
```

An instance of AMSDeviceDetector is declared in MainWindow.hxx as private member.

```
private:
    AMSDeviceDetector    itsDeviceDetector;
```

The HID device must be registered.

```
itsDeviceDetector.registerForHIDDevice(AMS_VID, AMS_EXAMPLE_PID);
```

- **Implementation of the Communication Class**

The communication class is derived from AmsComStreamWrapper. The class contains the three functions getFirmwareNumber, writeRegister and readRegister.

```
class MyCommunication : public AmsComStreamWrapper
{
    Q_OBJECT

public:
    MyCommunication ( );
    ~MyCommunication ( ) { };

    QString getFirmwareNumber ( );

    void writeRegister ( unsigned int regAddress, unsigned int regValue );
    int readRegister ( unsigned int regAddress );
};
```

```
MyCommunication::MyCommunication (): AmsComStreamWrapper (AMS_EXAMPLE_PID, AMS_VID) {
}
```

The communication class can now be used in MainWindow. The communication is declared as private member in MainWindow.hxx.

```
private:
    MyCommunication      *itsCom;
```

The communication is generated in MainWindow.cpp.

```
    itsCom = new MyCommunication();
```

The communication must be connected before using it.

```
    itsCom->connect();
```

– **Implementation of getFirmwareNumber**

The implementation of the method getFirmwareNumber uses the object FirmwareNumberObject.

```
QString MyCommunication::getFirmwareNumber ( )
{
    QString fwNum = "err 43.43.43";
    FirmwareNumberObject fno;

    itsStream >> fno;
    AmsFirmwareCheck::convert( fno.get(), fwNum );

    return fwNum;
}
```

With this implementation the communication can be verified easily. The firmware number should be read correctly.

– **Implementation of the direct command “readRegister”**

A new class for a read register object will be implemented.

```
#define COM_ID_READ_REG 0x02 /* !< My own command to read a register.  */

class ReadRegisterObj : public AmsComObject
{
public:
    ReadRegisterObj ( unsigned int theRegAdress ) :
         AmsComObject (COM_ID_READ_REG, 1, 1, 1), itsRegAddress(theRegAdress) { };
    ReadRegisterObj ( const ReadRegisterObj & other ) : AmsComObject( other ) { };
    ~ReadRegisterObj ( ) { };

    bool serialise(unsigned char *buffer, int bufferSize, QXmlStreamWriter *xml);
    bool rxSerialise(unsigned char *buffer, int bufferSize, QXmlStreamWriter *xml);
    bool deserialise (unsigned char *buffer, int bufferSize, QXmlStreamWriter *xml);
    bool fill(QXmlStreamReader *xml) {return false;};
    unsigned int get( );

private:
    unsigned int itsRegAddress;
    unsigned int itsRegValue;

};
```

The functions serialise, rxSerialise, deserialise to read-out and filling the buffer must be implemented.

```
bool ReadRegisterObj::serialise ( unsigned char *buffer, int bufferSize, QXmlStreamWriter
*xml )
{
    if ( bufferSize > 0 )
    {
        buffer[ 0 ] = static_cast< unsigned char > (0xFF & itsRegAddress);
        return true;
    }
    return false;
}

bool ReadRegisterObj::rxSerialise ( unsigned char *buffer, int bufferSize,
QXmlStreamWriter *xml )
{
    return serialise( buffer, bufferSize, xml );
}

bool ReadRegisterObj::deserialise ( unsigned char *buffer, int bufferSize,
QXmlStreamWriter *xml )
{
    if ( bufferSize > 0 )
    {
        itsRegValue = buffer[ 0 ];
        return true;
    }
    return false;
}

unsigned int ReadRegisterObj::get ( )
{
    return itsRegValue;
}
```

The read register object can now be used. The flush-operator executes the “flush”.

```
int MyCommunication::readRegister ( unsigned int regAddress )
{
    unsigned int regValue;
    int streamError;

    ReadRegisterObj rrObj;
    stream() >> rrObj;

    regValue = rrObj.get();
    streamError = stream.lastError(); /* check if comm. is okay */
    if ( streamError != 0 )
    {
        /* some error occurred – do your error handling */
    }
    return regValue;
}
```

5.3 Firmware Implementation

In usb_descriptors.c Device Descriptor Constants define the PID and VID of the application.

```
    0x1325,                 // Vendor ID: ams AG
    0xD003,                 // Product ID
```

The calling of the implementation for the protocol command “readRegister” has to be done in the function applProcessCmd.

Note it can also be implemented in this function. However if you have several commands to implement it is better to implement them in separate functions or files and just call them in this function.

Note: the types u8, u16, etc. are defined in the file: ams_types.h.

```
u8 applProcessCmd( u8 protocol, u16 rxSize, const u8 * rxData, u16 * txSize, u8 * txData
)
{
    u8 ret = AMS_STREAM_NO_ERROR;
    *txSize = 0; /* for most cases we do not want to send back some data */
    switch(protocol)
    {
        case COM_ID_WRITE_REG:   // write register
        {
            asxxxxWrite(rxData[0], rxData[1]);
            break;
        }
        case COM_ID_READ_REG 0x02:   // read register
        {
            u8 registerValue;
            asxxxxRead(rxData[0], &registerValue);
            txData[0] = registerValue;
            *txSize = 1; /* 1 byte to be sent back ->
                            protocol header is added by caller of this function */
            break;
        }
        default:
            ret = AMS_STREAM_UNHANDLED_PROTOCOL;
        break;
    }
    return ret;
}
```

6 Additional Information

6.1 Recording USB Stream Data

Using the Stream V2 communication gives the possibility for recording the data transferred over USB. The recorded data is stored in xml-Files. This functionality is available in the existing communication objects like for e.g. the I2cComObject. For new protocol command objects the implementation has to be done in the functions “fill”, “serialise”, “rxSerialise” and “deserialise”.

To have the communication recorded you have to add the following instruction in your main window:

```
 MyMainWindow::MyMainWindow :  AMSMainWindow(), …
{
    /* some code here to set up your communication etc. */

    AMSTrace::getInstace()->init(1); /* initialize to trace to file:
                C:\Users\<username>\AppData\Roaming\ams\<applicationName>\trace.txt */
}
```

A fragment of recorded data is shown by the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<session>
    <information guiversion="1.4.4.0" fwversion=""/>
<telegram pid="65" direction="write" trace="1">
    <reset>
        <objects>2</objects>
    </reset>
</telegram>
<telegram pid="7f" direction="write" trace="80000000"/>
<telegram pid="65" direction="write" trace="1">
    <reset>
        <objects>2</objects>
    </reset>
</telegram>
<telegram pid="7f" direction="write" trace="80000000"/>
```

6.2 XML Stream Player

Recorded data of USB stream communication as described in the previous chapter 6.1 can be played by the GUI with the class AmsComXmlPlayer. The implementation of AmsComXmlPlayer and AmsComXmlReader classes is done in the AmsCom.h and AmsCom.cpp files. A possible way to use the player is shown in the following example.

```
    QFile readIn( "C:\\temp\\trace.txt" );
    if ( readIn.open( QIODevice::ReadOnly ) )
    {
        QXmlStreamReader reader( &readIn );
        itsStream.setPolicy( AmsComStream::theLateFlushPolicy );
        itsStream.open( );
        AmsComXmlPlayer player( &reader, &itsStream );
        while ( player.playNext( ) ) ; /* play all commands from the XML file */
        itsStream.close( );
        readIn.close( );
    }
```

6.3 Compatibility to old Format

The StreamV2 firmware checks the format of the received USB command in the function usbStreamReceive. The function usbStreamOldFormatRequest handles commands in the old stream format.

The firmware response to an old format request is:

Byte	Content
0	0xDE = AMS_STREAM_COMPATIBILITY_TID
1	0x03= Payload in the old format
2	Protocol ID in the old format
3	0xFF = Status = failed , wrong protocol Version
4	0x00 = data length = No data

References

Nr.	Reference	Description
1	**USB Protocol Documentation 1v0**	**Description for the USB Communication with Stream V1**
2	**Ams_stream.h**	**Contains a description and constants for AMS Streaming Communication**
3	**www.usb.org**	**Universal Serial Bus Homepage**
4	**www.microchip.com**	**Microchip Homepage**
5	**C- Streaming Modules**	**C Source Code of the Microchip Firmware**
6	**AMS Communication class**	**C++ Source Code of the AMS Communication Project**
7	**de.wikipedia.org**	**Wikipedia**

Contact Information

WuXi Silicontrol Electronic Technology Co., Ltd.

WEB:
http://www.silicontrol.com
http://flagship.eleckits.com

E-Mail:support@eleckits.com

TEL:86-0510-83488567

Skype:eleckits2011