



# Prozessvisualisierung

Kommunikation zwischen MBED und C#

Dieses Dokument beschreibt die Prozessvisualisierung unter Verwendung eines „embedded systems“ unter MBED und einer Anwendersoftware in C#.

**HTL BULME Graz**

**HL/KS, 2021**

## Inhalt

<b>1</b>	<b>Prozessvisualisierung .....</b>	<b>2</b>
<b>2</b>	<b>Wie und wo werden Prozessgrößen angezeigt?.....</b>	<b>3</b>
<b>3</b>	<b>Die Datenübertragung .....</b>	<b>4</b>
<b>4</b>	<b>Aufgaben der Kommunikations-SW .....</b>	<b>5</b>
4.1	Übertragung verschiedener Datentypen.....	5
4.2	Mehrkanalige Übertragung .....	5
4.3	Fluss-Steuerung und Flusskontrolle .....	5
4.4	Zusammenspiel zwischen Kommunikations- und Steuer/Regel-SW .....	5
<b>5</b>	<b>Voraussetzungen auf dem µC.....</b>	<b>6</b>
<b>6</b>	<b>Konfiguration und Bedienung SvVis3 .....</b>	<b>7</b>
6.1	Control-Menü .....	8
6.1.1	Acq. On/Off .....	8
6.1.2	EmptyReceiveBuffer .....	8
6.1.3	Clear Messages .....	8
6.1.4	AcqPoints On/Off .....	8
6.1.5	Recording On/Off .....	8
6.2	Window-Menü .....	8
6.2.1	CurveWin On/Off.....	8
6.2.2	Keyboard On/Off .....	9
6.2.3	BarWin On/Off.....	9
6.2.4	PIDWin On/Off .....	9
6.2.5	Command On/Off.....	10
<b>7</b>	<b>Das Übertragungs-Protokoll .....</b>	<b>11</b>
7.1	Systemvariablen µC -> PC .....	11
7.2	Kommandos PC -> µC.....	11
<b>8</b>	<b>Serialize in C .....</b>	<b>11</b>
8.1	Funktionsweise .....	11
8.1.1	Das Schreiben auf einen Byte-Buffer .....	11
8.1.2	Das Lesen von einem Byte-Buffer .....	12
8.2	StreamRW .....	13
8.3	Writel16 (Int 16) .....	14
8.4	Writel32 (Int 32) .....	14
8.5	WriteF (Float) .....	14
8.6	WriteS (String) .....	15
8.7	Readl16 (Int16) .....	15
8.8	Readl32 (Int32) .....	15
8.9	ReadS (String) .....	16
<b>9</b>	<b>Code-Beispiele.....</b>	<b>16</b>
9.1	MBed : Simple Uart Klasse.....	16
9.1.1	MBed : SvTest.cpp .....	17
9.1.2	Der BinaryWriter .....	17
9.1.3	Der BinaryReader .....	18
9.1.4	Serialport Initialisierung: OnLoad/OnFormClose .....	19
9.1.5	Kommunikation mit µC Aufbauen: OnUCSendChk .....	19
9.1.6	Empfangspuffer leeren: OnEmptyReceiveBuffer.....	19
9.1.7	Protokolldekodierung/ Anzeige: OnTimer .....	20
9.1.8	Kommandos und Werte Senden (Textbox) : OnSendEditKeyDown .....	21
9.1.9	Kommandos und Werte Senden (Button) : OnCommand3 .....	21
9.1.10	MBed: Coordinate_Transfer_Demo.cpp.....	21
9.1.11	Der BinaryWriter (Array Extension) .....	22
9.1.12	Array senden: OnCommand 3.....	22

# 1 Prozessvisualisierung

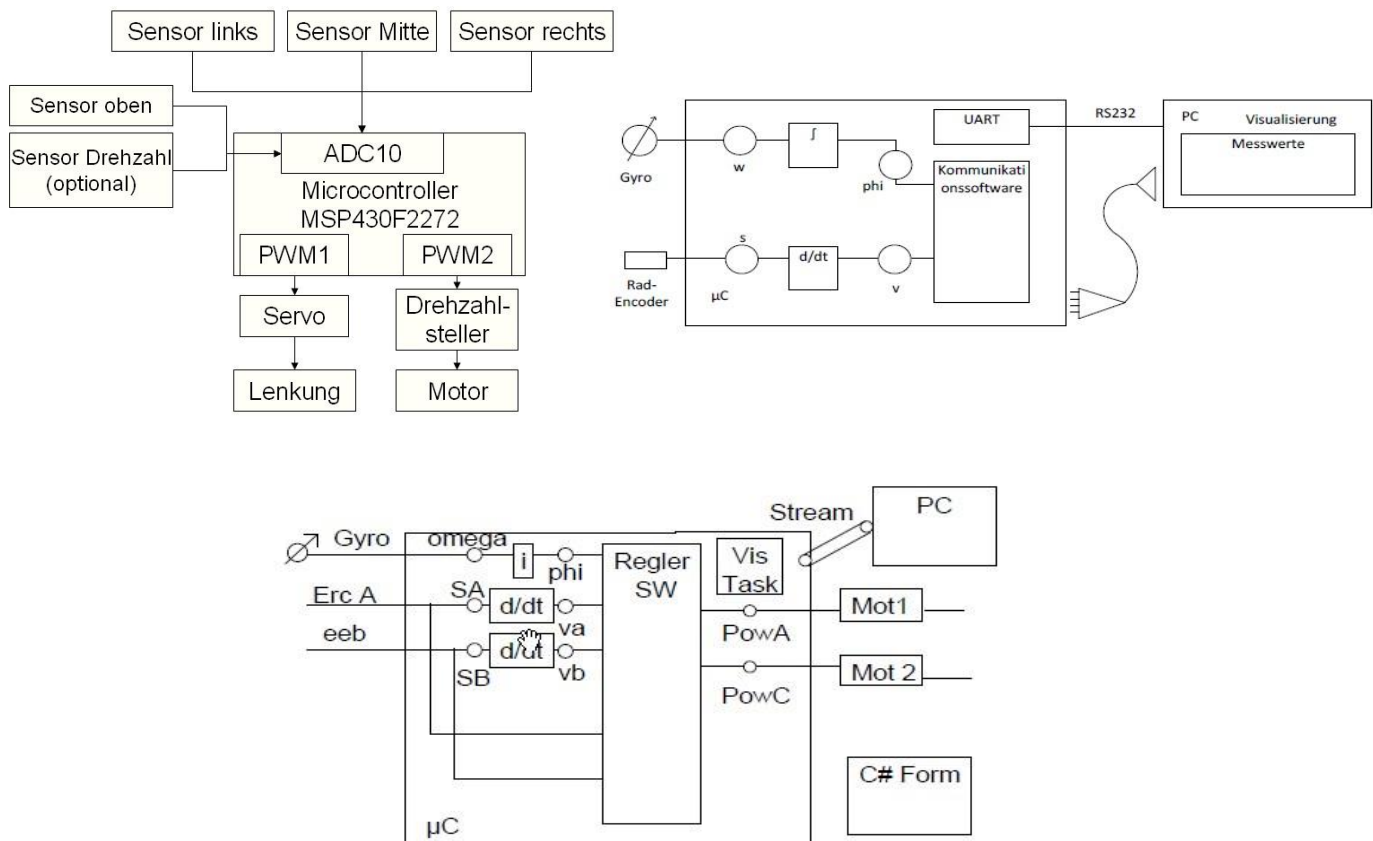
Die Steuerung, Beobachtung und Überwachung komplexer technischer Systeme erfolgt heutzutage mit Microcontrollern und Computern in sogenannten „embedded systems“.

Beispiele für solche Systeme:

- Quadrocopter, mobiler Roboter mit komplexen Sensoren
- Autonomes GPS-gesteuertes Modellauto
- ABS-System im Auto
- Engine Control Unit (ECU) im PKW-Motor (Stichwort „Chip Tuning“)
- Balancierender Roboter

Auf einem solchen eingebetteten System entstehen nun 3 Arten von Mess- und Regelgrößen, die mit einer für das System typischen Frequenz erfasst, verarbeitet und wieder ausgegeben werden:

- **Messgrößen** von Sensoren
- Durch Filter oder Regler **berechnete Größen**
- **Stellgrößen**, die an die Aktuatoren (Motoren, Servos ...) des Systems wieder ausgegeben werden



Beispiel: Mess- und Prozessgrößen eines balancierenden Roboters

Für die erwähnten Mess- und Prozessgrößen werden wir die Bezeichnung **Systemvariablen** (SV) verwenden.

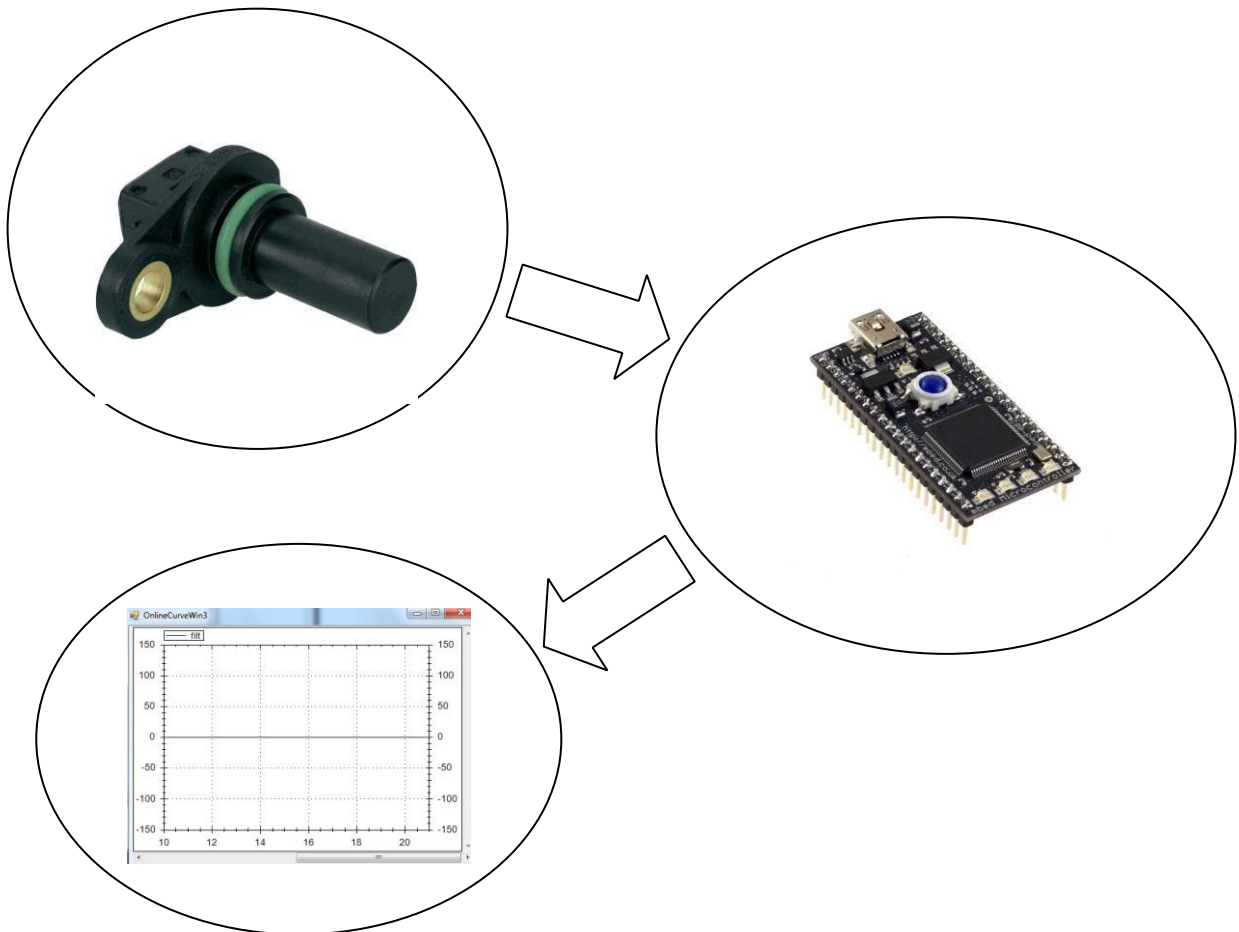
## 2 Wie und wo werden Prozessgrößen angezeigt?

Wir werden zur Anzeige von **Systemvariablen** einfache, selbstgeschriebene und gestaltete C#-Forms mit Standard-Kontrollelementen (TextBox, Slider, Buttons...) verwenden.

Zur Anzeige von Online-Grafiken (Kurven) wird ein fertiges Modul verwendet.

Sensoren erfassen physikalische Größen. Der  $\mu$ C digitalisiert die erfassten Größen und verarbeitet diese. Die erfassten Daten werden zum PC übertragen.

Die Daten werden empfangen und für die visuelle Darstellung aufbereitet.

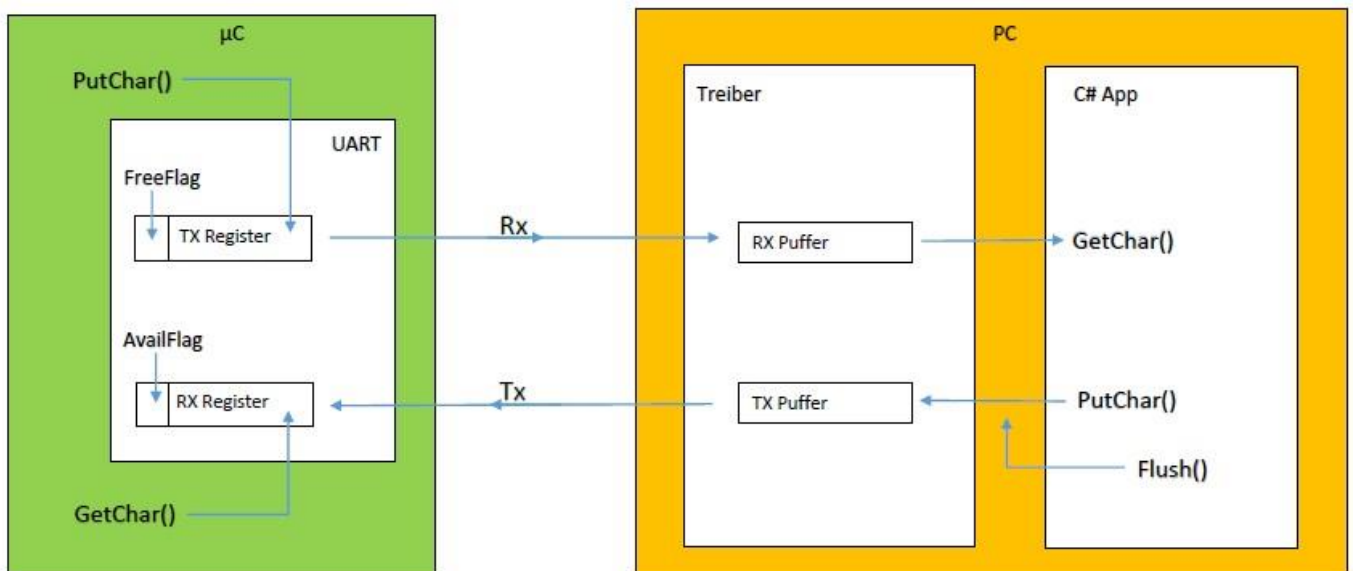


### 3 Die Datenübertragung

Embedded Systems müssen in der Lage sein, auch ohne angeschlossene Visualisierung zu funktionieren. Die Visualisierung wird nur angeschlossen, um Fehler zu analysieren oder z.B. Regler und Abläufe einzustellen und zu optimieren („tunen“). Der Visualisierungs-PC wird mit dem embedded System über einen sogenannten Stream verbunden. Der Stream hat die Eigenschaft, dass Bytes, die man auf der einen Seite hineinschiebt, in der gleichen Reihenfolge am anderen Ende wieder herauskommen. Der Stream zum Visualisierungs-PC ist typischerweise bidirektional d.h. es können auch Kommandos vom PC zum embedded System gesendet und µC-Systemvariablen auf dem PC verändert werden.

Mögliche sinnvolle Stream-Verbindungen sind:

- Virtuelle serielle Schnittstelle (VCP) über USB
- Virtuelle serielle Schnittstelle über Bluetooth
- TCP/IP Verbindung



#### TxReg am µC:

Die SW am µC schreibt mit der Funktion `PutChar()` Bytes in das TxReg des UART. Diese Bytes werden dann mit der eingestellten Bitrate (9600KBit bis 10Mbit) seriell zum PC übertragen. Zur Synchronisierung zwischen Kommunikations-HW und der Kommunikations-SW wird das **Free-Flag** des **TxRegisters** verwendet. Das FreeFlag ist gesetzt, wenn das vorhergehende Datenbyte vom UART hinausgetaktet wurde.

#### RxReg am µC:

Datenbytes, welche vom PC zum µC gesendet werden, landen im RxReg des UART. Wenn ein neues Datenbyte angekommen ist wird das **Avail-Flag** im **RxRegister** gesetzt. Das AvailFlag wird zur Synchronisation zw. Kommunikations-HW und der Kommunikations-SW verwendet, es muss vom µC mit einer hinreichend hohen Frequenz abgefragt werden, damit keine Bytes verloren gehen.

## Serieller ( USB ) Treiber am PC

Das Gegenstück zu den UART-Registern am  $\mu\text{C}$  sind der **RxBuffer** und **TxBuffer** im Seriellen-USB Treiber am PC. Auf diese Puffer (Arrays, FIFO...) greift die Visualisierungs-SW am PC zu. Die Puffer-Schnittstelle am PC ist nicht so kritisch wie die Register- Schnittstelle am  $\mu\text{C}$ . Wird der RxBuffer eine Zeitlang nicht abgefragt so gehen keine Bytes verloren da die Bytes ja zwischengepuffert werden.

## 4 Aufgaben der Kommunikations-SW

### 4.1 Übertragung verschiedener Datentypen

Stream-Verbindungen können ohne zusätzliche Kommunikations-Libraries nur Bytes übertragen.

Es müssen zusätzliche Funktionen zur Übertragung der Datentypen **int, long, float, double, string** geschrieben werden.

### 4.2 Mehrkanalige Übertragung

Werden die Werte von mehreren Systemvariablen (SV) gleichzeitig am PC angezeigt müssen muss es eine Möglichkeit geben die Werte und Datentypen der einzelnen SV's im Datenstrom zu unterscheiden.

### 4.3 Fluss-Steuerung und Flusskontrolle

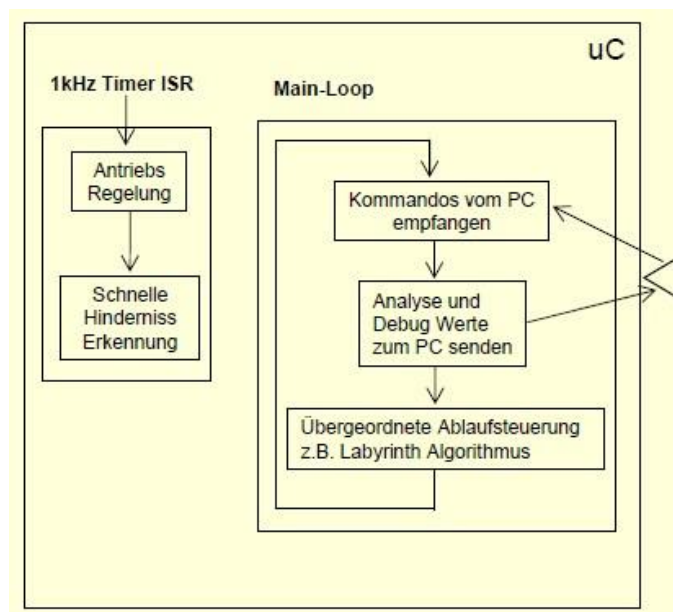
Mit einem Kommando vom PC aus muss die Datenübertragung für einzelne und auch für alle SV's ein und ausschaltbar sein. Es muss ein Mechanismus vorhanden sein der verhindert, dass die Visualisierung mit Daten überschwemmt wird.

Die Default Einstellung ist, das vom embedded System keine Visualisierungsdaten gesendet werden. Erst wenn ein Visualisierungs-PC zu dem System connected wird, werden Analysedaten gesendet.

### 4.4 Zusammenspiel zwischen Kommunikations- und Steuer/Regel-SW

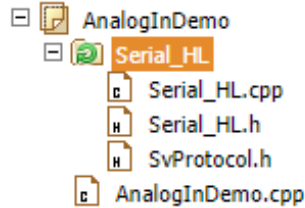
Die für die Prozessvisualisierung benötigte zusätzliche Prozessorleistung darf nur so gering sein, dass die eigentlichen Kontrollaufgaben des embedded Systems nicht gestört werden.

Durch geeignete Interrupt-Programmierung muss erreicht werden, dass z.B. ein Reglertask die Visualisierung jederzeit unterbrechen kann, wenn ein Abtast- und Regelungs-Schritt durchzuführen ist.



## 5 Voraussetzungen auf dem µC

Die Übertragung der Daten vom µC zum PC läuft am einfachsten über die Bibliothek „Serial\_HL“, welche aus den folgenden Dateien besteht:



Serial\_HL.cpp: Code für die Klasse „SerialBLK“ und die Klasse „SvProtocol“

Serial\_HL.h: Header-Datei für „SerialBLK“



SvProtocol.h: Header-Datei für „SvProtocol“

Beispiel-Anwendung:

```



1 #include "mbed.h"
2 #include "Serial_HL.h"
3 SerialBLK pc(USBTX, USBRX);
4 SvProtocol ua0(&pc);
5 void CommandHandler();
6 AnalogIn ana(A3);
7
8 int main(void)
9 {
10     pc.format(8, SerialBLK::None, 1);
11     pc.baud(9600);
12     ua0.SvMessage("AnalogInDemo"); // Meldung zum PC senden
13     Timer stw; stw.start();
14     while(1) {
15         CommandHandler();
16         if( (stw.read_ms() > 10) ) { // Abfrage ob Timer > 10ms => 100Hz
17             stw.reset();
18             if( ua0.acqON ) { // nur wenn vom PC aus das Senden eingeschaltet wurde
19                 ua0.WriteSvI16(1, ana.read_u16()); // Kanal1: int16 senden
20                 ua0.WriteSV(2, ana.read()); // Kanal2: float senden
21             }
22         }
23     }
24     return 1;
25 }
26 void CommandHandler()
27 {
28     uint8_t cmd;
29     int16_t idata1, idata2;
30
31     if( !pc.IsDataAvail() ) // falls keine Daten im RX-Register stehen->return
32         return;
33     cmd = ua0.GetCommand(); // Kommando vom PC lesen
34 }

```

Function **SerialBLK**  

Class: **mbed::SerialBLK**

SerialBLK::SerialBLK(PinName tx, PinName rx)

Class **SerialBLK**  

Namespace: **mbed**

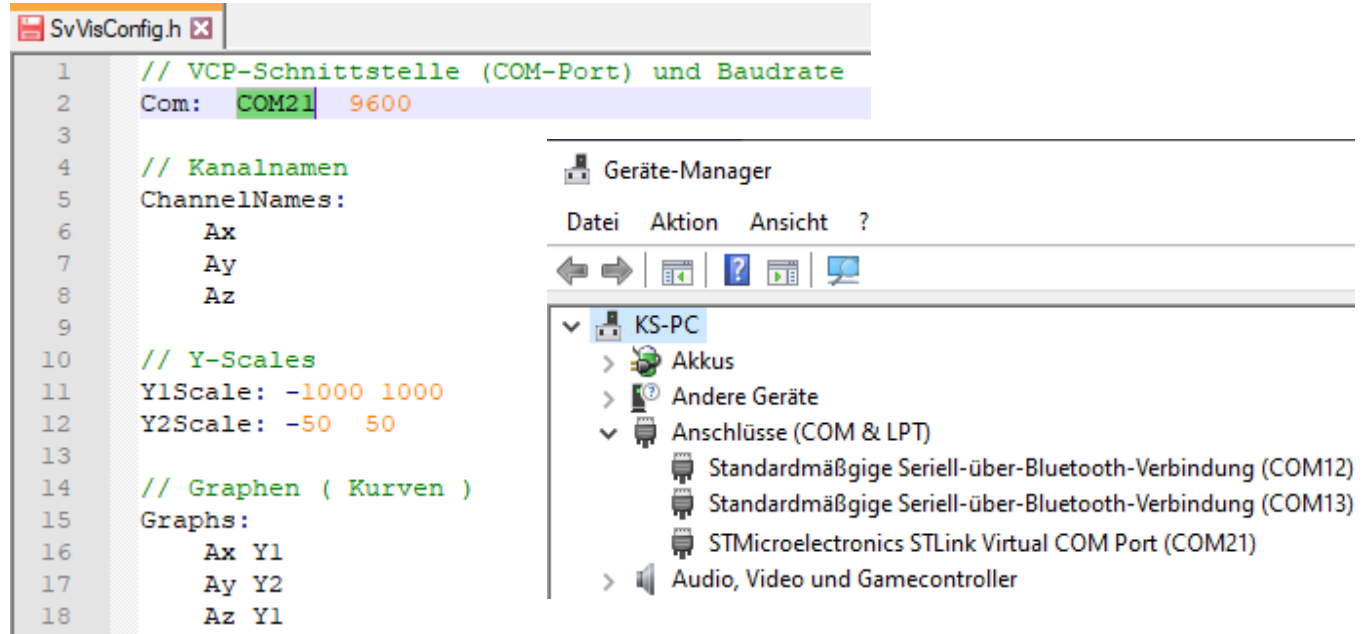
class SerialBLK : public IStreamHL

## 6 Konfiguration und Bedienung SvVis3

SvVis3 besteht aus den folgenden Dateien:

- SvVis3.exe
- ZedGraphRel.dll
- ZedHLRel.dll
- CmdList.h
- SvVisConfig.h

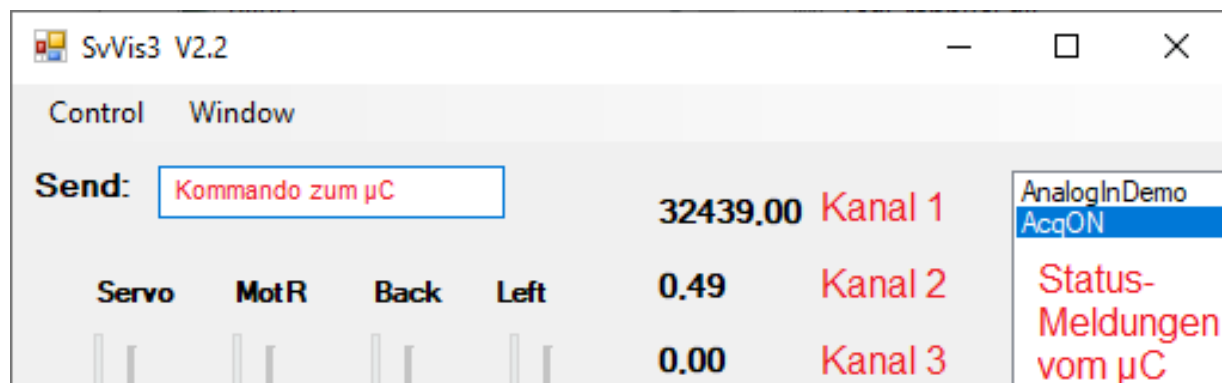
Vor dem Start der Anwendung (erfordert u.U. eine einmalige Sicherheits-Freigabe unter Windows) sollte der Inhalt der Datei „SvVisConfig.h“ geprüft und ggf. editiert werden:



„ChannelNames“ erlaubt die Benennung der zu übertragenden Kanäle (im Beispiel 3 Kanäle Ax, Ay und Ax)  
 „Y\_Scale“ definiert den Wertebereich für die Skala Y\_ (die y-Achse unterstützt 2 verschiedene Skalen)  
 „Graphs“ definiert für die angegebenen Kanäle die dazugehörige Skala (also entweder Y1 oder Y2)

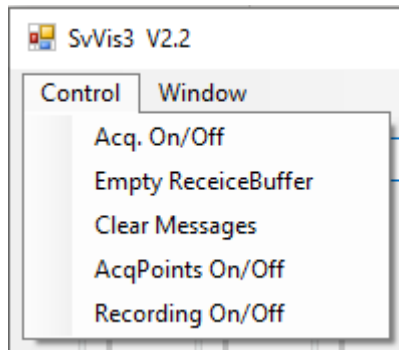
Die Kanäle werden im „OnlineCurveWin“-Fenster (Menü „Window->CurveWin On/Off“) grafisch dargestellt.

Die Datei „CmdList.h“ dient als Speicher für eine Sequenz von Kommandos (Befehle vom PC zum µC), damit diese ohne lästiges Eintippen in rascher Folge gesendet werden können. Das Kommando-Fenster kann über das Menü „Window->Commands On/Off“ ein- und ausgeschaltet werden.





## 6.1 Control-Menü



### 6.1.1 Acq. On/Off

Mit der Funktion „Acq. On/Off“ kann die Daten-Akquirierung auf dem  $\mu\text{C}$  ein- und ausgeschaltet werden. Dies setzt die Eigenschaft „ua0.acqON“ der Klasse SvProtocol.

Nur wenn die Akquirierung eingeschaltet ist (wird durch ein Häkchen im Menü und eine Statusmeldung angezeigt), werden Daten übertragen (und die Werte für die Kanäle aktualisiert).

### 6.1.2 EmptyReceiveBuffer

Mit dieser Funktion wird lediglich der Empfangspuffer geleert (erlaubt Synchronisation mit dem  $\mu\text{C}$  nach Reset)

### 6.1.3 Clear Messages

Löscht alle Statusmeldungen in der ListBox.

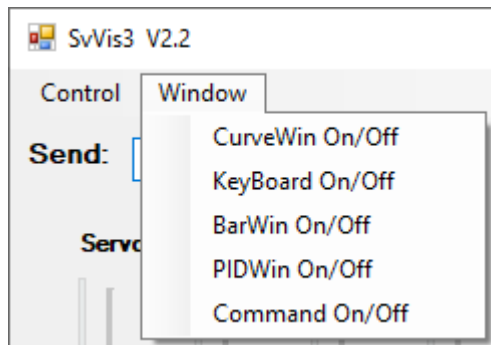
### 6.1.4 AcqPoints On/Off

Falls aktiv, werden auch die genauen Datenpunkte in den Graphen (im „CurveWin“) dargestellt/markiert

### 6.1.5 Recording On/Off

Falls aktiv, werden die empfangenen Daten in eine Datei „SvVis.dat“ geschrieben.

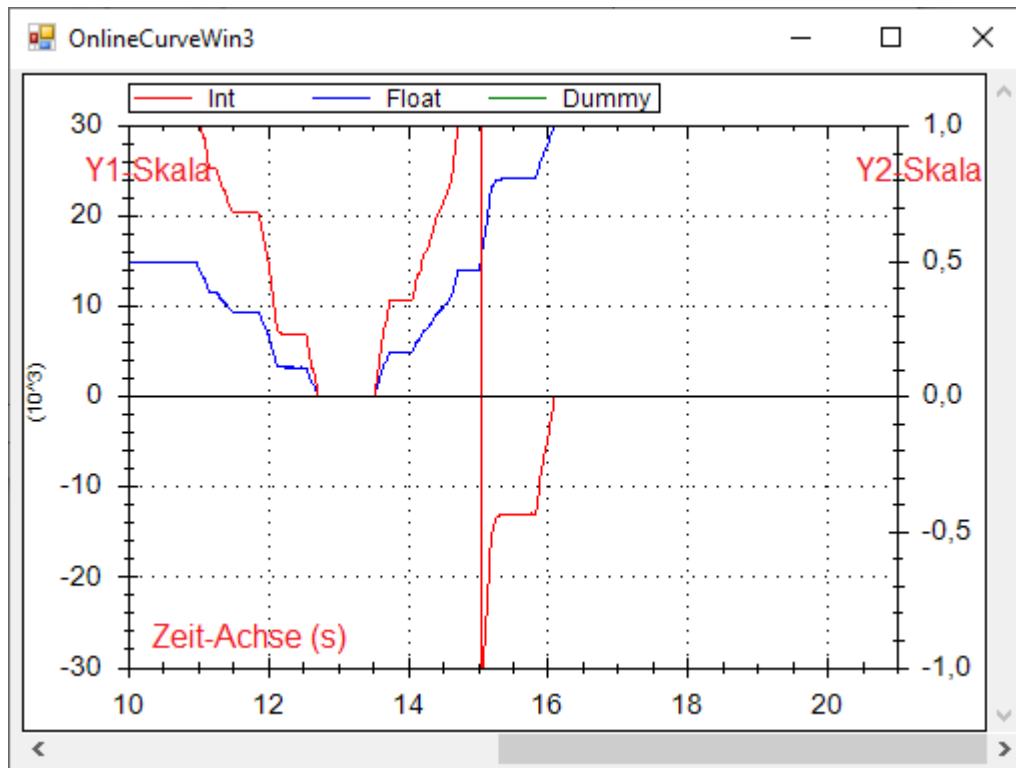
## 6.2 Window-Menü



### 6.2.1 CurveWin On/Off

Diese Funktion öffnet eine grafische Anzeige der empfangen Werte/Kanäle. Alle Kanäle, welche vom  $\mu\text{C}$  Werte gesendet bekommen, müssen dazu in der Datei „SvVisConfig.h“ mit einem Namen konfiguriert werden (Abschnitt „ChannelNames“).

Die Zuordnung der einzelnen Kanäle zu den 2 möglichen Y-Achsen müssen auch in „SvVisConfig.h“ definiert sein (Abschnitt „Graphs“).

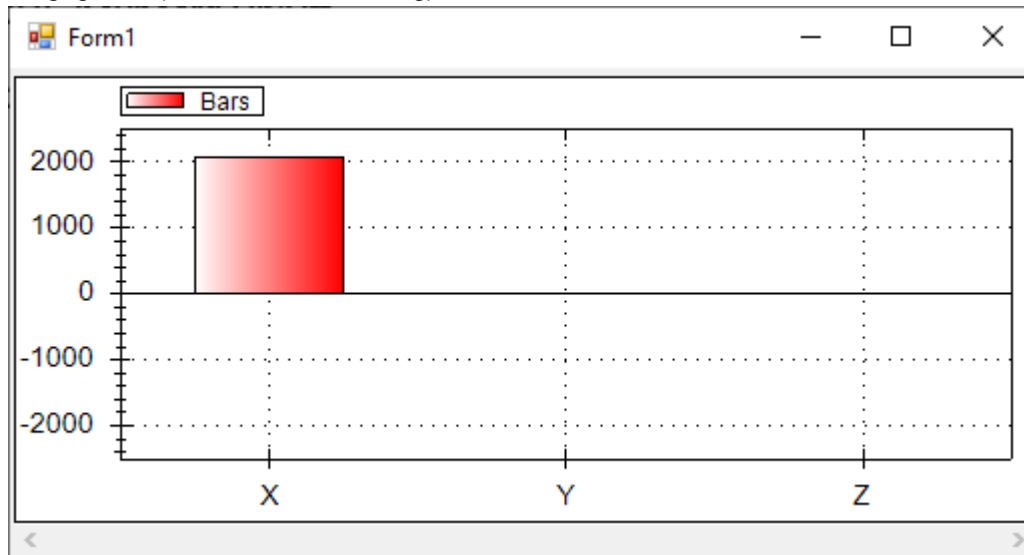


### 6.2.2 Keyboard On/Off

Öffnet einen Eingabe-Editor (PianoForm).

### 6.2.3 BarWin On/Off

Stellt die empfangen Werte als ein Säulendiagramm dar (gut geeignet für den Vergleich von mehreren identischen Sensorwerten, wie z.B. Liniensensoren eines mobilen Roboters). Die Skala ist fix mit  $\pm 2500$  vorgegeben (bis zu 11bit Auflösung).

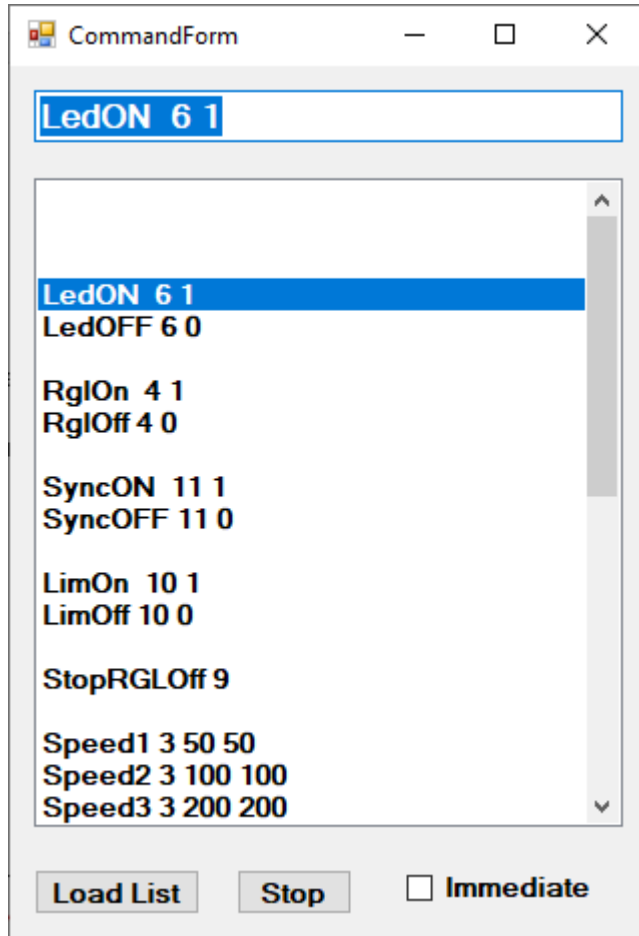


### 6.2.4 PIDWin On/Off

Öffnet einen Editor, um die Koeffizienten eines PID-Reglers zu setzen

### 6.2.5 Command On/Off

Öffnet ein Fenster zur Steuerung von Makro-Befehlen. Die einzelnen Befehle müssen in der Datei „CmdList.h“ definiert sein.



Ein Befehl besteht aus einem Text (wird nicht gesendet, sondern dient nur zur einfacheren Erkennung), einem numerischen Code für den Befehl und (optional) aus einem oder mehreren Parameter.

Das markierte Beispiel „LedON 6 1“ sendet den Befehl „6“ mit dem Parameter „1“. Der  $\mu\text{C}$  muss regelmäßig über die Methode `pc.IsDataAvail()` (`pc` ist ein Objekt der Klasse „SerialBLK“) abfragen, ob ein Kommando vom PC erhalten wurde (siehe Code-Beispiel aus Abschnitt 5 „Voraussetzungen auf dem  $\mu\text{C}$ “).

Entsprechend des Formats des Befehls (also Abhängig von der Anzahl der Parameter) muss der  $\mu\text{C}$  mit der Methode `ua0.GetCommand()` (`ua0` ist ein Objekt der Klasse „SvProtocol“) den Befehl und die Daten der Parameter abfragen.

Durch Anklicken einer Zeile kann der Befehl in das Sende-Feld übernommen werden. Mit der Checkbox „Immediate“ kann ein Befehl durch Anklicken auch sofort gesendet werden.

## 7 Das Übertragungs-Protokoll

### 7.1 Systemvariablen $\mu\text{C}$ -> PC

Svld	float	Svld	float	Svld	int	Svld	string	Sync	.....
------	-------	------	-------	------	-----	------	--------	------	-------

Svld .... 1,2,3 ...

Der Datentyp kann in der Svld kodiert sein.

z.B. 1..50 int    51..100 float    101..150 string

### 7.2 Kommandos PC -> $\mu\text{C}$

Cmd1	int
------	-----

Cmd2	int	float
------	-----	-------

Cmd3	float	float
------	-------	-------

Jedes Kommando mit seinen Parametern entspricht einem Funktionsaufruf vom **PC** zum  **$\mu\text{C}$** .

```
Also: void Func1(int aPar1);          void
Func2(int aPar1, float aPar2);
      void Func3(float aPar1, float aPar2);
```

Dieser Vorgang wird Remote Procedure Call genannt da es sich um einen Funktionsaufruf von einem Rechner auf einen anderen handelt.

## 8 Serialize in C

Als Serialisieren bezeichnet man das Verpacken der Datentypen string, int, float in einem ByteStrom. Als De-Serialisieren bezeichnet man das Auspacken (Lesen) dieser Daten aus dem Byte-Strom. Das Serialisieren und De-Serialisieren ist der wichtigste Vorgang beim Entwurf von Kommunikationsprotokollen über serielle Byte-Stromorientierte Verbindungen wie z.B. serielle Verbindung  $\mu\text{C}$ ->PC, TCP/IP Verbindung oder seriell über Bluetooth.

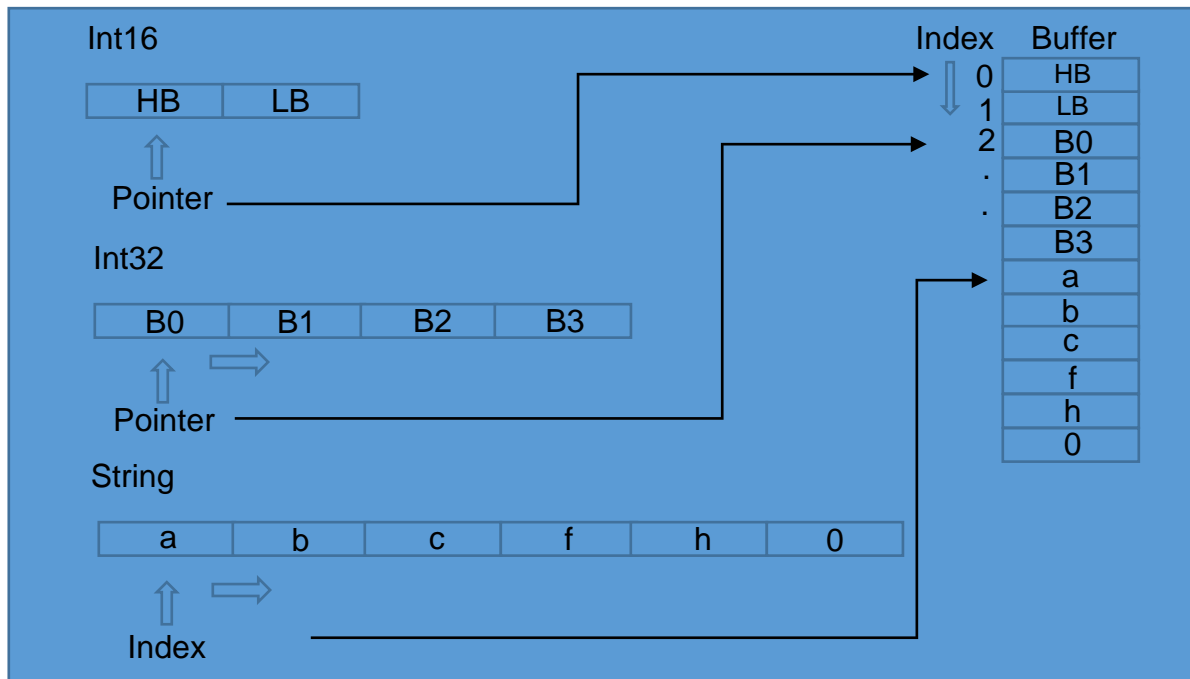
### 8.1 Funktionsweise

#### 8.1.1 Das Schreiben auf einen Byte-Buffer

Es wird mithilfe einer Indizierung auf einen Buffer (FIFO, Byte Array) geschrieben, die Einzelnen Bytes werden nacheinander übergeben, wobei je nach Datentyp, mehrere Speicherplätze im Buffer benötigt werden. Es muss im Übertragungsprotokoll klar definiert sein, welche Datentypen im Buffer gespeichert werden, z.B.: Int, short, int, char, string usw., damit der Buffer wieder ordnungsgemäß ausgelesen werden kann.

Im Falle einer Int16 (2 Byte) zeigt ein Pointer auf das High-Byte, welches dann auf die 1. Stelle im Buffer geschrieben wird, Bufferindex = 0, danach zeigt der Pointer auf das Low-Byte welches an die nachfolgende Stelle

im Buffer geschrieben wird, Bufferindex = 1. (siehe WriteI16 6.3). Diese Funktionsweise ist auf alle Datentypen übertragbar, es ändert sich lediglich die Anzahl der zu speichernden Bytes. Die Einzige Ausnahme, ist das Speichern von Strings. (String endet immer mit /0).



### 8.1.2 Das Lesen von einem Byte-Buffer

Im Wesentlichen ist es genau umgekehrt wie beim Schreiben auf einen Byte-Buffer. Es wird der niedrigste Bufferindex (0) auf das höchstwertige Byte (HB) der Variablen geschrieben.

Beispiel (Int16, 2Bytes):

```
void StreamRW::WriteI16(int16_t aVal) // int16_t zeiger besteht aus 2 Byte {
    // ptr zeigt auf das LB von aVal
    byte* ptr = (byte*)&aVal;
    // *ptr; // der Wert des LB
    _buffer[_idx] = *ptr; // Das LB von aVal wird auf den Stream geschrieben
    _idx++; // Schreibindex um 1.Stelleiterrücken (1Byte)
    ptr++; // ptr auf das NB weiterschalten
    _buffer[_idx] = *ptr; // Das HB von aVal wird auf den Stream geschrieben
    _idx++; // zeigt immer auf die nächste freie Stelle des Streams
}
```

Buffer (Debugger):

v1 0xab12

```

int16_t StreamRW::ReadI16()
{
    int16_t val;
    // auf aVal einen Pointer setzen um val Byte_weise beschreiben können.
    byte* ptr = (byte*)&val;
    //LB aus dem Buffer auf das LB von val schreiben
    *ptr = _buffer[_idx],
        _idx++, ptr++;
    //HB aus dem Buffer auf das HB von val schreiben
    *ptr = _buffer[_idx],
        _idx++, ptr++;
    return val;
}

```

## 8.2 StreamRW

```

class StreamRW {
public:
    StreamRW(byte aBuffer[]);

    void Reset() // Reset _idx to Start-Position
    {
        _idx = 0;
    }
    void WriteI16(int16_t aVal);
    void WriteI32(int32_t aVal);
    void WriteF(float aVal);    void
    WriteS(char* aTxt);

    int16_t ReadI16();
    int32_t ReadI32();

    float    ReadF();
    void     ReadS(char* aDest);

private:
    byte* _buffer; // ptr to ByteStream with
Data    int _idx; // current Read/Write Index in the
stream
}

StreamRW::StreamRW(byte aBuffer[])
{
    _buffer = aBuffer;
    _idx = 0;
}

```

### 8.3 WriteI16 (Int 16)

```
void StreamRW::WriteI16(int16_t aVal) // int16_t zeiger besteht aus 2 Byte {
    // ptr zeigt auf das LB von aVal
    byte* ptr = (byte*)&aVal;

    // *ptr; // der Wert des LB

    _buffer[_idx] = *ptr; // Das LB von aVal wird auf den Stream geschrieben
    _idx++; // Schreibindex um 1.Stelleiterrücken (1Byte)

    ptr++; // ptr auf das NB weiterschalten

    _buffer[_idx] = *ptr; // Das HB von aVal wird auf den Stream geschrieben
    _idx++; // zeigt immer auf die nächste freie Stelle des Streams }
```

### 8.4 WriteI32 (Int 32)

```
void StreamRW::WriteI32(int32_t aVal) // int32_t zeiger besteht aus 4 Byte
{
    byte* ptr = (byte*)&aVal;
    _buffer[_idx] = *ptr; // Byte0
    _idx++; ptr++; // Pointer incrementieren
    _buffer[_idx] = *ptr; // Byte1
    _idx++; ptr++; // Pointer incrementieren
    _buffer[_idx] = *ptr; // Byte2
    _idx++; ptr++; // Pointer incrementieren
    _buffer[_idx] = *ptr; // Byte3
    _idx++; ptr++; // Pointer incrementieren
}
```

### 8.5 WriteF (Float)

```
void StreamRW::WriteF(float aVal)
{
    byte* ptr = (byte*)&aVal;
    _buffer[_idx] = *ptr; // Byte0
    _idx++; ptr++; // Pointer incrementieren
    _buffer[_idx] = *ptr; // Byte1
    _idx++; ptr++; // Pointer incrementieren
    _buffer[_idx] = *ptr; // Byte2
    _idx++; ptr++; // Pointer incrementieren
    _buffer[_idx] = *ptr; // Byte3
    _idx++; ptr++; // Pointer incrementieren
}
```

## 8.6 WriteS (String)

```
void StreamRW::WriteS(char*
aTxt)
{
    int i = 0; // Index Text
while(1)
{
    _buffer[_idx] =aTxt[i];
i++,_idx++;
    if(aTxt[i]=='\0')
        break;
}
}
```

## 8.7 ReadI16 (Int16)

```
int16_t StreamRW::ReadI16()
{
    int16_t val;
    // auf aVal einen Pointer setzen um val Byte_weise beschreiben können.
    byte* ptr = (byte*)&val;

    //LB aus dem Buffer auf das LB von val schreiben
    *ptr = _buffer[_idx],
        _idx++, ptr++;

    //HB aus dem Buffer auf das HB von val schreiben
    *ptr = _buffer[_idx],
        _idx++, ptr++;

    return val;
}
```

## 8.8 ReadI32 (Int32)

```
void StreamRW::WriteI32(int32_t aVal) // int32_t zeiger besteht aus 4 Byte
{
    byte* ptr = (byte*)&aVal;
    _buffer[_idx] = *ptr; // Byte0
    _idx++; ptr++; // Pointer incrementieren
    _buffer[_idx] = *ptr; // Byte1
    _idx++; ptr++; // Pointer incrementieren
    _buffer[_idx] = *ptr; // Byte2
    _idx++; ptr++; // Pointer incrementieren
    _buffer[_idx] = *ptr; // Byte3
    _idx++; ptr++; // Pointer incrementieren
}
```



## 8.9 ReadS (String)

```
void StreamRW::ReadS(char*
aDest)
{
    int i = 0; // Index Text
    while(1) {
        aDest[i] = _buffer[_idx];
        i++, _idx++;
        if(aDest[i]=='\0')
            break;
    }
}
```

## 9 Code-Beispiele

### 9.1 MBed : Simple Uart Klasse

In der SimpleUart Klasse wird die Kommunikation über den. UART des µC initialisiert und durchgeführt.

- .Open()  
Dient zur Initialisierung des UART z.B. ua0.Open(0,115200), in diesem Beispiel wird der Uart 0 mit 115200 Baud Initialisiert.
- IsDataAvail()  
uint8\_t IsDataAvail()  
{ return (\_ua->LSR & UART\_LSR\_RDR); }  
Es wird überprüft, ob im Empfangsregister des UARTs Daten vorhanden sind.
- GetCommand()  
Liest das Empfangsregister aus und überprüft, ob das Kommando 1 gesandt wurde. Mit cmd == 1 wird die Kommunikation ein/ausgeschaltet. Wird ein anderes Kommando ausgelesen, wird dieses in der Variablen cmd übergeben.
- ReadI16()  
Mit ReadI16 werden 2 Byte gelesen. (siehe read() )
- ReadF()  
Mit ReadF werden 4 Byte gelesen. (siehe read() )
- SvPrintf()  
Mit SvPrintf() wird eine Zeichenkette gesandt. (ist eine Erweiterung zu Printf()), speziell für SvVisXXXX)

(für genauere Definitionen: siehe SimpleUart.cpp und SimpleUart.h)

### 9.1.1 MBed : SvTest.cpp

(Beschreibung: siehe Kommentare)

```
void CommandHandler()
{
    uint8_t cmd;
    int16_t idata1, idata2;
    float fdata;

    // Fragen ob überhaupt etwas im RX-Reg steht
    if( !ua0.IsDataAvail() )
        return;

    // wenn etwas im RX-Reg steht
    // Kommando lesen
    cmd = ua0.GetCommand();

    if( cmd==2 )
    {
        // cmd2 hat 2 int16 Parameter
        idata1=ua0.ReadI16(); idata2=ua0.ReadI16();
        ua0.SvPrintf("Command2 OK %d %d", idata1, idata2);
    }
    if( cmd==3 )
    {
        // cmd3 hat einen int16 und einen float Parameter
        idata1=ua0.ReadI16(); fdata=ua0.ReadF();
        ua0.SvPrintf("Command3 OK %d %d", idata1, (int16_t)(10*fdata));
    }
}
```

### 9.1.2 Der BinaryWriter

```
class BinaryWriterEx : BinaryWriter
{
public:
    BinaryWriterEx(Stream input)
        : base(input)
    {
    }

    public void WriteSv16(byte aId, short aVal)
    {
        this.Write(aId);
        this.Write((byte)aVal); // LB
        this.Write((byte)(aVal >> 8)); // HB
    }
}
```

Der BinaryWriterEx ist eine Klasse speziell für die Kommunikation des SVisxxxx Protokolls. Hier wird Byteweise auf das Senderegister des Uarts geschrieben. ( in diesem Fall WriteSV16, zuerst die ID. 1.Byte danach das LB und das HB). (Extension zu Write())

### 9.1.3 Der BinaryReader

```

class BinaryReaderEx : BinaryReader
{
    byte[] m_CString = new byte[30];

    public BinaryReaderEx(Stream input)
        : base(input) {...}

    public string ReadCString()
    {
        int len = 0; byte ch;
        while (true)
        {
            ch = this.ReadByte();
            if (ch == 0)
                break;
            m_CString[len] = ch; len++;
        }
        string ret = Encoding.ASCII.GetString(m_CString, 0, len);
        return ret;
    }
}

```

Der Binary ReaderEx ist Eine Klasse speziell für die Kommunikation des SvVisxxx Protokolls. Hier werden die ankommenden Bytes in einem Array gespeichert.

### 9.1.4 Serialport Initialisierung: OnLoad/OnFormClose

```
protected override void OnLoad(EventArgs e)
{
    m_SerPort = new SerialPort("COM4", 115200, Parity.None, 8, StopBits.One);
    // m_SerPort.ReadBufferSize = 20 * 1024;
    m_SerPort.Open();
    m_BinWr = new BinaryWriter(m_SerPort.BaseStream);
    m_BinRd = new BinaryReaderEx(m_SerPort.BaseStream);
    m_Timer1.Enabled = true; m_Timer1.Interval = 100;
    base.OnLoad(e);
}
```

Hier werden der Serial Port, der Timer und die Binary Klassen zur Kommunikation Initialisiert.

```
protected override void OnFormClosing(FormClosingEventArgs e)
{
    m_Timer1.Enabled = false;
    m_BinRd.Close(); m_BinWr.Close(); m_SerPort.Close();
    base.OnFormClosing(e);
}
```

Hier wird der Timer gestoppt und die serielle Verbindung beendet.

### 9.1.5 Kommunikation mit µC Aufbauen: OnUCSendChk

```
private void OnUCSendChk(object sender, EventArgs e)
{
    m_BinWr.Write((byte)1); // cmd1
    if( m_uCSendChk.Checked )
        m_BinWr.Write((byte)1); // daten=1 für on
    else
        m_BinWr.Write((byte)0); // daten=0 für off
}
```

Dieser Code ist speziell für das SvVisxxxx Protokoll, hierbei wird durch senden einer 1 oder 0 die Kommunikation zum MBED Ein/Ausgeschaltet. (Dient dazu das die Kommunikation nur aktiv ist wenn Sie wirklich benötigt wird)

### 9.1.6 Empfangspuffer leeren: OnEmptyReceiveBuffer

```
private void OnEmptyReceiveBuffer(object sender, EventArgs e)
{
    int nBytes = m_SerPort.BytesToRead;
    for (int i = 1; i <= nBytes; i++)
        m_SerPort.ReadByte();
}
```

Hier wird lediglich der Empfangspuffer geleert.

### 9.1.7 Protokolldekodierung/ Anzeige: OnTimer

```
private void OnTimer(object sender, EventArgs e)
{
    int id, val;
    while (m_SerPort.BytesToRead >= 3)
    {
        id = m_SerPort.ReadByte();
        if (id == 10) // es ist die TextMeldung
        {
            string txt;
            txt = m_BinRd.ReadCString();
            listBox1.Items.Add(txt);
        }
        else // sonst int16 oder float
        {
            // - 10 ... Korrektur der Datentyp Codierung
            id = id - 10;
            val = m_BinRd.ReadInt16();
            // abhängig von id den Wert auf das richtige Textfeld schreiben
            m_Dispatch[id - 1].Text = val.ToString();
        }
    }
}
```

Dieser Code ist speziell für die Kommunikation über das SvVisProtokoll. Hier wird beim Interrupt des Timers der Empfangspuffer ausgelesen und die Daten werden decodiert. (in diesem speziellen Fall wird, wenn im 1.Byte die ID 10 gesandt wird, eine Textmeldung in der ListBox hinzugefügt, wenn die ID größer 10 ist wird Sie decodiert und als Text angezeigt (Array --> Label))

### 9.1.8 Kommandos und Werte Senden (Textbox) : OnSendEditKeyDown

```
void OnSendEditKeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyValue != 13) // CR
        return;

    short id, val;
    string[] words = m_SendEd.Text.Split(',');
    id = short.Parse(words[0]);
    val = short.Parse(words[1]);
    // m_BinWr.Flush();
}
```

Hier wird der String einer Textbox geparsert und der Inhalt gesendet (z.B.: 1,400 --> words [0] = ID = 1 , words [1] = 400)

### 9.1.9 Kommandos und Werte Senden (Button) : OnCommand3

```
void OnCommand3(object sender, EventArgs e)
{
    short val;
    m_BinWr.Write((byte)2); // cmd 2 senden
    m_BinWr.Write((short)1000);
    m_BinWr.Write((short)2000);
    m_BinWr.Flush();
}
```

Hier werden bei der Betätigung des Buttons „Command3“ eine vorgegebene ID (2) und die Werte (1000 und 2000) gesendet. (Protokoll SVVisxxxx)

### 9.1.10 MBed: Coordinate\_Transfer\_Demo.cpp

```
int16_t coordAry[20];
int16_t numCoord;
void ReadCoordinates()
{
    int nBytes = ua0.ReadI16();
    ua0.Read(coordAry, nBytes);
    numCoord = nBytes/2;
    ua0.SvMessage("ReadCoordinates");
    for(int i=0; i<numCoord; i++)
        ua0.SvPrintf("%d", coordAry[i]);
}

void CommandHandler()
{
    uint8_t cmd;
    if( !ua0.IsDataAvail() )
        return;
    cmd = ua0.GetCommand();
    if( cmd==2 )
    {
        ReadCoordinates();
    }
}
```

Hier werden wir im vorherigen Beispiel die Klasse SimpleUart() und der CommandHandler() verwendet. Allerdings wird hier mit ReadCoordinates() in ein Array eingelesen , gespeichert und wieder Übertragen.

### 9.1.11 Der BinaryWriter (Array Extension)

```
public void WriteAry16(byte aId, ref short[] aAry)
{
    short len = (short)aAry.Length;
    this.Write(aId);
    this.Write((short)(len * 2));
    for (int i = 0; i < len; i++)
        this.Write(aAry[i]);
}
```

Dies ist eine Erweiterung zum BinaryReader(), es wird zuerst die ID übertragen und danach ein Array mit variabler Länge.

### 9.1.12 Array senden: OnCommand 3

```
void OnCommand3(object sender, EventArgs e)
{
    short[] ary = new short[5];
    for (int i = 0; i < 5; i++)
        ary[i] = (short)((i + 1) * 10);
    m_BinWr.WriteAry16(2, ref ary);
}
```

Hier wird wie ein fest definiertes Array (Länge 5) mithilfe des WriteArray16 übertragen.