Prozessvisualisierung

Communication between mBed and C#

In diesem Schriftstück wird die Prozessvisualisierung unter verwendung eines eingebetteten Systems (mBed) und der Anwendersoftware (C#) beschriben.

28.01.2014

Inhalt

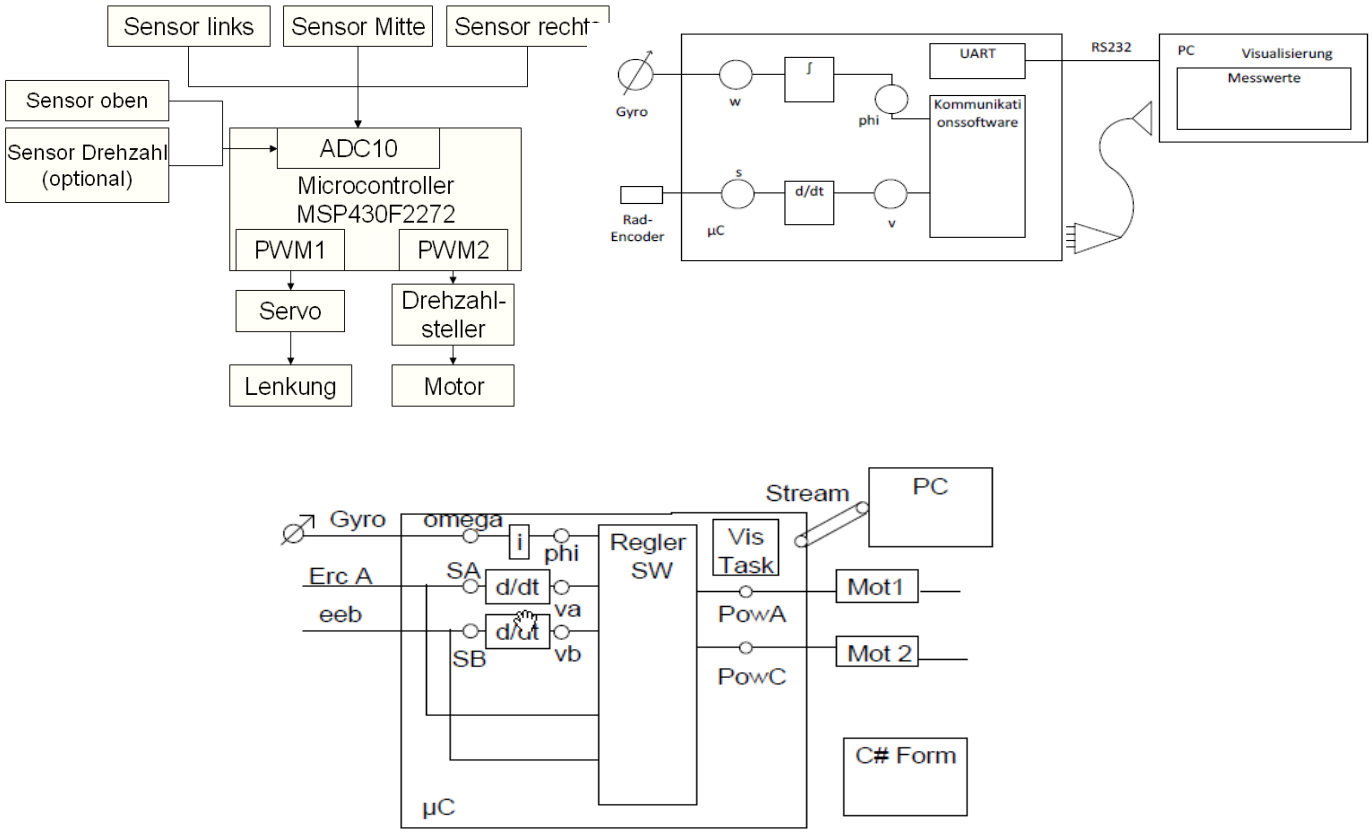
1 Prozessvisualisierung, Wozu ? und Was ist das?

Die Steuerung, Beobachtung und Überwachung komplexer technischer Systeme erfolgt heutzutage mit Microcontrollern und Computern in sog. eingebetteten Systemen.
Beispiele für solche Systeme:

- Autonomes GPS-gesteuertes Modellauto
- ABS-System im Auto
- Motor Control Unit im PKW-Motor (Chip Tuning)
- Selfbalancing NXT-Roboter
- Quadrocopter, Maze-Solving-Robot

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.
Die Größen sind:

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



Mess- und Prozessgrößen eines „balancing Robot“

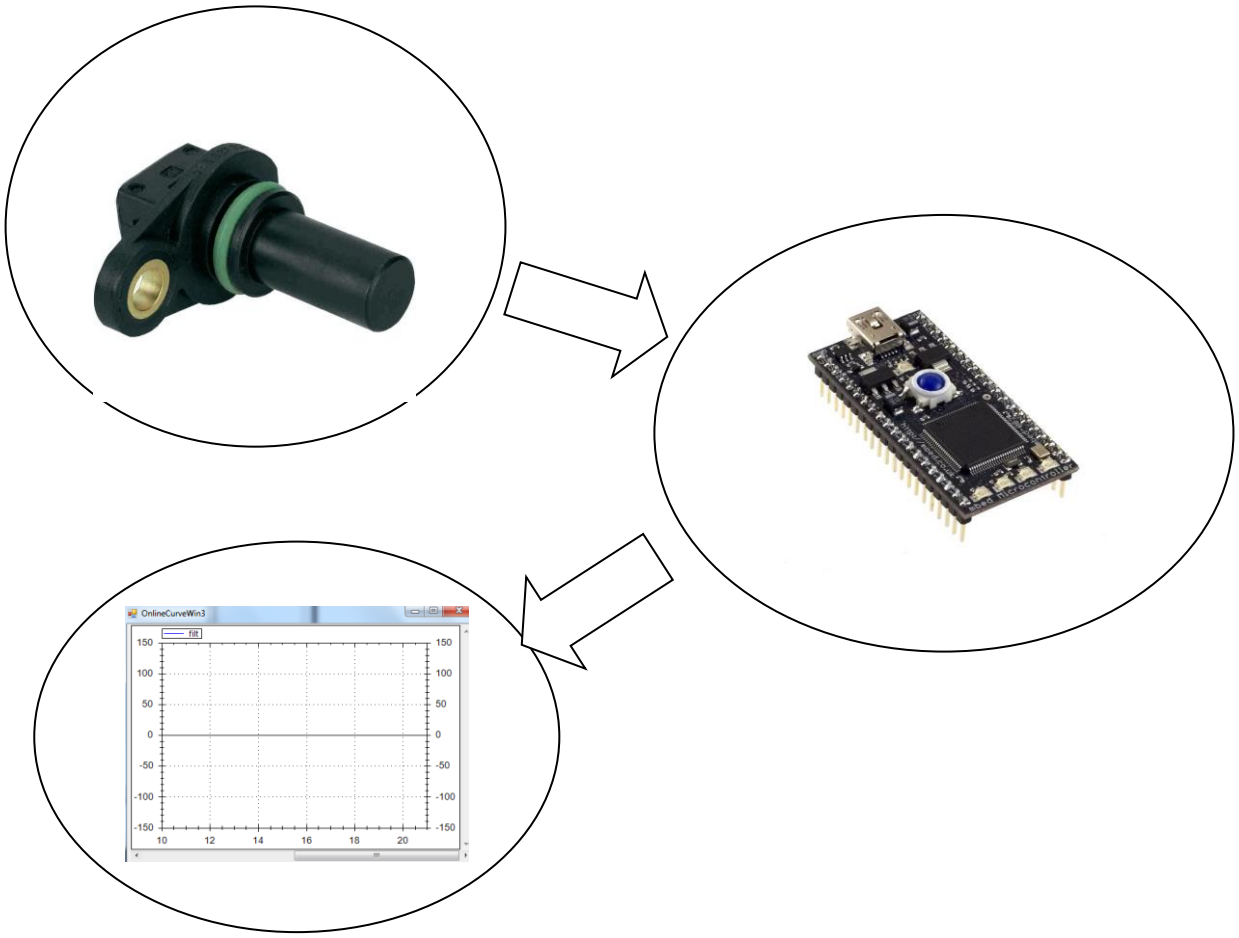
Für die erwähnten Mess und Prozessgrößen werden wir die Bezeichnung **Systemvariablen** verwenden.

2 Wie und wo werden Prozeßgrößen angezeigt?

Wie werden zur Anzeige von **Systemvariablen** einfache selbstgeschriebene und gestaltete C#-Forms mit Standard Kontrollelementen (TextBox, Slider, Button . . . .) verwenden.
Zur Anzeige von Online-Grafiken (Kurven) wird ein fertiges Modul zur Verfügung gestellt.

Sensoren erfassen Physikalische Größen. Der MBed digitalisiert die erfassten Größen und verarbeitet diese. Die Erfassten Daten warden zm PC übertragen.

Die Daten werden Empfangen und zur Visuellen Darstellung aufbereitet.

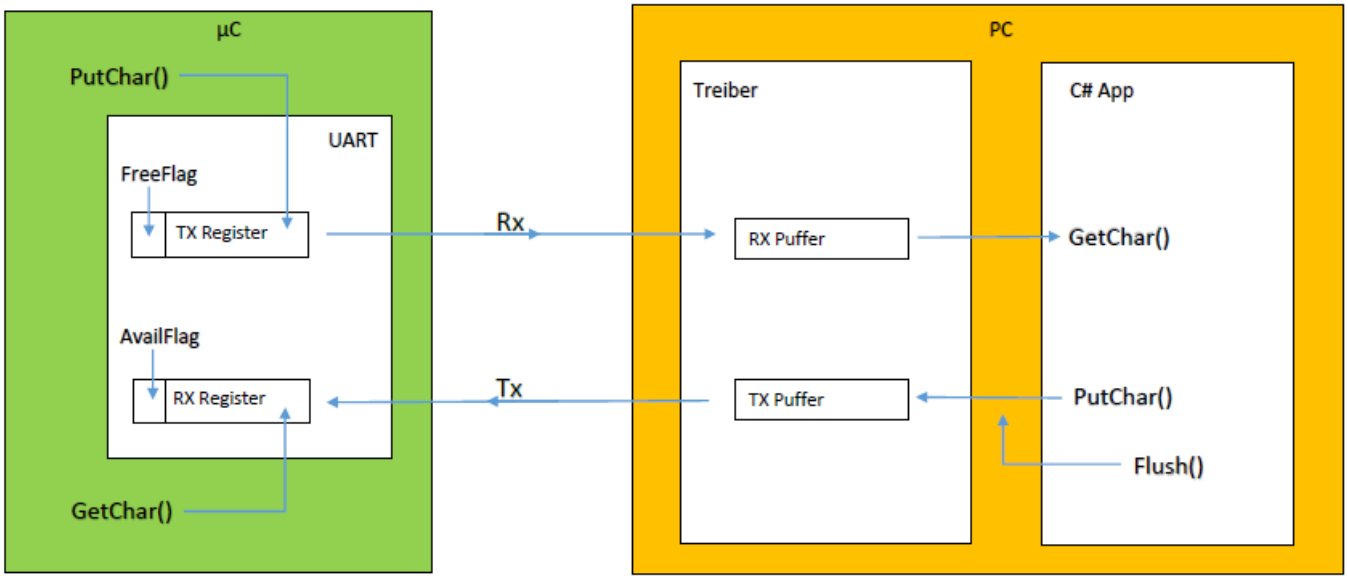


3 Die Datenübertragung

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 tunen. Der Visualisierung PC wird mit dem embedded System über einen sog. 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 Systemvariablen vom PC verändert werden.

Mögliche sinnvolle Streamverbindungen sind:

- Klassische serielle Schnittstelle
- Virtuelle serielle Schnittstelle über USB
- Virtuelle serielle Schnittstelle über Bluetooth
- TCP/IP Verbindung



TxReg am uC:
Die SW am uC 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 zw. 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 uC:
Datenbytes welche vom PC zum uC 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 uC mit einer hinreichend hohen Frequenz abgefragt werden, damit keine Bytes verloren gehen.

Serieller (USB) Treiber am PC

Das Gegenstück zu den UART-Registern am uC sind der **RxBuffer** und **TxBuffer** im Seriellen-USB Treiber am PC. Auf diese Puffer (Arrays, FiFos . . .) greift die Visualisierungs-SW am PC zu. Die Puffer-Schnittstelle am PC ist nicht so kritisch wie die Register- Schnittstelle am uC. 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

Streamverbindungen 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

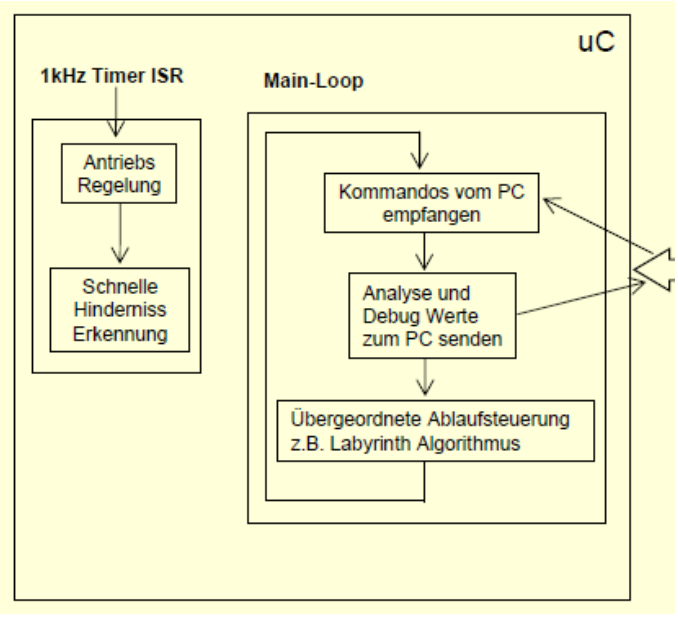
Werden die Werte von mehreren Systemvariablen gleichzeitig am PC angezeigt müssen muss es eine Möglichkeit geben die Werte und Datentypen der einzelnen SV's im Datenstrom zu unterscheiden.

4.3 Flusssteuereung 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 zw Kommunikations und Steuer/Regel SW

Die für die Prozessvisualisierung benötigte zusätzliche Prozessorleistung darf nur so gering sein, dass die eigentlichen Kontrollaufgaben des embedded System's 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 Das Übertragungs Protokoll

5.1 Systemvaribalen uC -> PC

SvId	**float**	**SvId**	**float**	**SvId**	**int**	**SvId**	**string**	**Sync**	**. . . . . .**

SvId . . . . 1,2,3 . . .
Der Datentyp kann in der **SvId** kodiert sein.
z.B. 1..50 int 51..100 float 101..150 string

5.2 Kommandos PC -> uC

Cmd1	**int**	
Cmd2	**int**	**float**
Cmd3	**float**	**float**

Jedes Kommando mit seinen Parametern entspricht einem Funktionsaufruf vom **PC** zum **uC**.

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

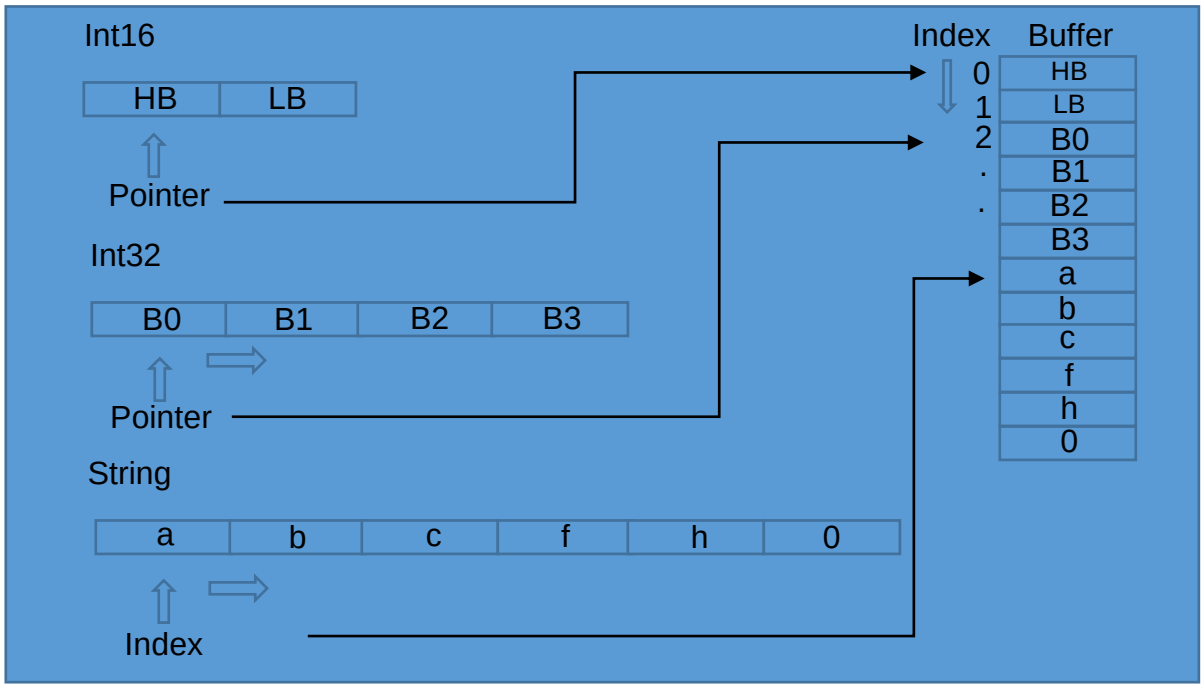
Dieser Vorgang wird Remote Procedure Call genannt da es sich um einen Funktionsaufruf von einem Rechner auf einen anderen handelt.

6 Serialize in C

Als Serialisieren bezeichnet man das Verpacken der Datentypen string, int, float in einem ByteStrom. Als Deserialisien bezeichnet man das Auspacken (Lesen) dieser Daten aus dem ByteStrom. Das Serialisieren und Deserialisien ist der wichtigste Vorgang beim Entwurf von Kommunikationsprotokollen über serielle ByteStrom-orientierte Verbindungen wie z.B. serielle Verbindung uC<->PC, TCP/IP Verbindung oder seriell über Bluetooth.

6.1 Funktionsweise

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).

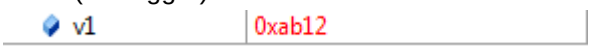


Das lesen von einem ByteBuffer
Im Wesentlichen ist es genau umgekehrt wie beim Schreiben auf einen Byte-Buffer. Es wird der niedrigste Bufferindex (0) auf das höchstwertige Byte (HB) der Variablen geschrieben.

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.Stelle weiterrücken (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):




```
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;
}
```

6.2 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;


}
```

6.3 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.Stelle weiterrücken (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
}
```

6.4 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
}
```

6.5 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
}
```

6.6 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;
  }
  }
```

6.7 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;
}
```

6.8 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
}
```

6.9 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;
  }
}
```

7 CodeBeispiele

7.1 MBed : Simple Uart Klasse

In der SimpleUart Klasse wird die Kommunikation über die. Div. UART's des MBed Initialisiert und durchgeführt.

- .Open()
  Dient zur initialisierung des UART zB. ua0.Open(0,115200), in diesem Beispiel wird der Uart 0 mit 115200 Baut Initialisiert.

- IsDataAvail()
  ```
  uint8_t IsDataAvail()
  { return (_ua->LSR & UART_LSR_RDR); }
  ```
  Es wird überprüft ob im Empfangsregister des UART's Daten vorhanden sind.

- GetCommand()
  Liest das Empfangsregister aus und überprüft ob das Kommando 1 gesandt wurde. Mit cmd == 1 wird die Kommunikation ein/ausgeschalten. 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)

7.1.1 MBed : SvTest.cpp

(Beschreibung, siehe Kommentare)

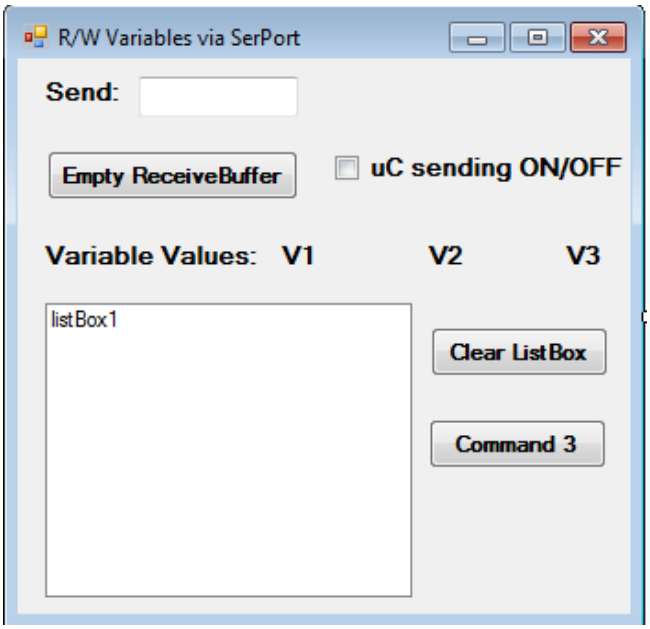
```
void CommandHandler()
{
  uint8_t cmd;
    int16_t idata1, idata2;
    float fdata;

    // 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));
    }
}
```

7.2 C# : SvVisLoesung.cs



7.2.1 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 SVVisxxxx Protokolls. Hier wird Byteweise auf das SendeRegister des Uarts geschrieben. (in diesem Fall WriteSV16, zuerst die ID. 1.Byte dannach das LB und das HB). (Extension zu Write())

7.2.2 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.

7.2.3 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.

7.2.4 Kommunikation mit uC 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/Ausgeschalten. (Dient dazu das die Kommunikation nur aktiv ist wenn Sie wirklich benötigt wird)

7.2.5 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.

7.2.6 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_DispAry[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 einder ListBox hinzugefügt, wenn die Id größer 10 ist wird Sie decodiert und als Text Angezigt (Array --> Label))

7.2.7 Kommandos und Werte Senden (Textbox) : OnSendEditKeyDowm

```
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 (zB.: 1,400 --> words [0] = ID = 1 , words [1] = 400)

7.2.8 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)10000);
  m_BinWr.Write((short)20000);
  m_BinWr.Flush();
}
```

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

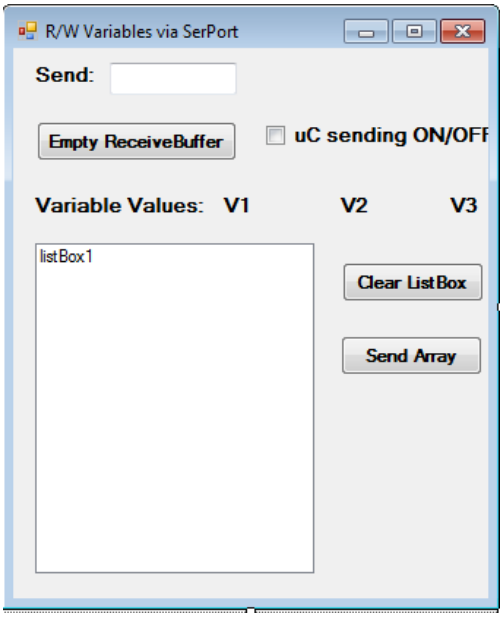
7.2.9 MBed: Coordinate_Transfer_Demo.cpp

```
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.

7.3 C# : SvVisArray.cs



7.3.1 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 dannach ein Array mit varriabler länge.

7.3.2 Array senden: OnCommad 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.