



Measuring amplifier GSV-8 / BX8 Getting started into programming with C, C# or NI LabView®

Last changes: 05/02/2019



Table of contents

Scope of this document.....	2
MEGSV86xx.dll introduction.....	3
Chapter 1: Fundamental principles of a program using BX-8.....	3
Initialization (1).....	3
Main loop / task (2).....	3
Termination (3).....	3
Implementation of a simple program using MEGSV86xx.dll and Labview.....	5
Step 1: Initialization using LabView.....	5
Step 2: Reading measurement values using LabView.....	5
Step 3: Program termination using LabView.....	7
Implementation of a simple program using MEGSV86xx.dll and C language.....	8
Step 1: Initialization using C.....	8
Step 2: Reading measurement values using C.....	9
Step 3: Program termination using C.....	10
Simple program using MEGSV86xx.dll and C# language.....	11
Chapter 2: Additional information on MEGSV86xx.DLL.....	13
Driver for using USB port.....	14
Initialization step.....	14
Parametrization of general-purpose sensors.....	14
Input Type.....	15
GSV86setInType.....	15
Display scaling.....	16
GSV86getValObjectInfo.....	16
GSV86readUserScale.....	16
GSV86writeUserScale.....	16

Scope of this document

This manual is intended to describe the basic knowledge needed to develop simple programs interfacing the BX8 measuring amplifier at its USB or serial interface.

It is not intended to serve as an introduction to programming languages; however it gives some examples on how to use the BX8 Windows dynamic link library under the following languages:

- C / C++
- LabView©
- C#

To develop programs using operating systems other than Windows or hardware platforms other than 8086 PC, one has to communicate with the device using its interface commands directly. Therefore, another document describing the communication protocol is more suitable. Its available on the manufacturers or distributors website.

MEGSV86xx.dll introduction

The Windows Dynamic Link Library simplifies the communication with the measuring device by providing functions for particular reading or writing actions.

There are 32-Bit and 64-Bit versions (MEGSV86w32.dll and MEGSVx64.dll) which are identical in their functions and their calling parameter lists (but different calling conventions).

All functions accessing the device have a fundamental calling parameter "ComNo", with which the particular device with its serial interface is accessed and distinct. It must be known by the caller; informations on determining it can be found in the device manual.

Up to 256 different devices can be handled. All functions are described en detail in GSV86DLL_referenceManual.pdf

Chapter 1: Fundamental principles of a program using BX-8

If the purpose of an application program is to read and process measuring values only, the principle approach can be broken down in three parts:

Initialization (1)

This step is necessary and done only once for a particular device on a particular serial COMport No. / USB-CDC virtual COMport. It consists of:

- Loading the DLL (if not loaded statically or automatically, e.g. as in LabView)
- Opening the serial interface
- Allocating further resources
- Reading relevant parameters of the device and changing them, if necessary

Main loop / task (2)

Generally, in this step, measuring values are read and processed. The main task depends a lot on the application requirements. Here, we assume that its only purpose is to display measuring values. Other conditions are checked, e.g. request for program exit or device parameter change.

Termination (3)

This step is necessary and done only once for a particular device on a particular serial COMport No. / USB-CDC virtual COMport. The serial port must be closed and all resources allocated in step (1) must be freed.

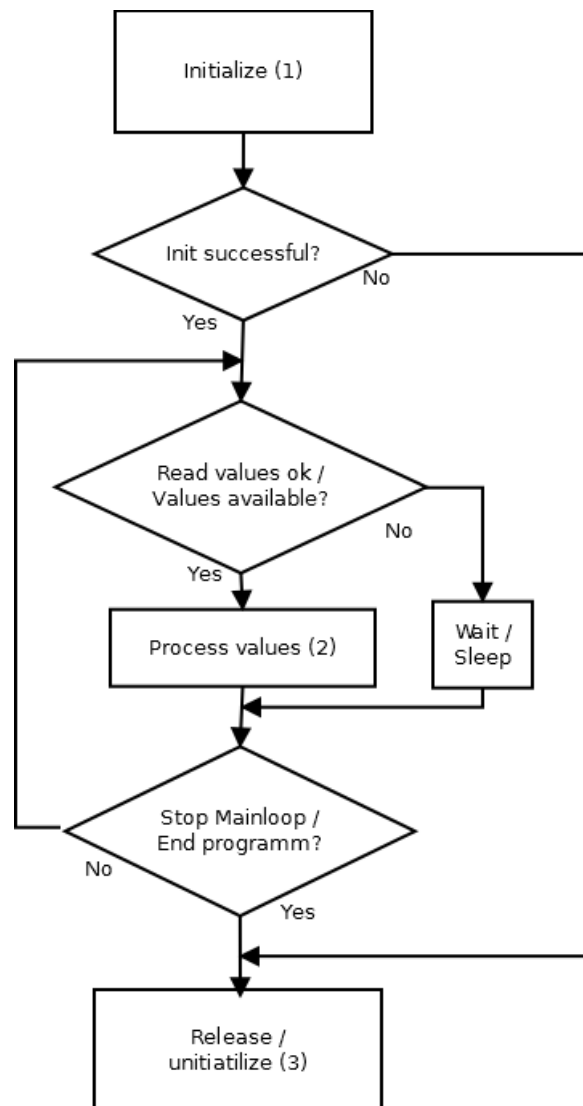


Figure 1: Flowchart of a simple program

Implementation of a simple program using MEGSV86xx.dll and Labview

For use with LabView, a library with wrapper-VIs is available in the directory \megsv\VIs.

These VIs have almost the same name as the DLL's function names, only one or more white space characters may be included.

The wrapper VIs call the DLL-function with the same name, and most of them also call GSV86getLastErrorMessage in case of a failure and provide the error text as a part of the LabView-predefined error cluster. That text may be displayed by using the "Simple error handler" under "Application control". Example with GSV86act Ext.vi:

Step 1: Initialization using LabView

This is done by either calling GSV86actExt or GSV86activateExtended¹.

GSV86actExt is a simplified version with one parameter only, "ComNo" (see above).

Here, we use GSV86act Ext.vi. The return value of the Output error cluster should be evaluated, because it encodes the important information whether the device could be successfully accessed or not.

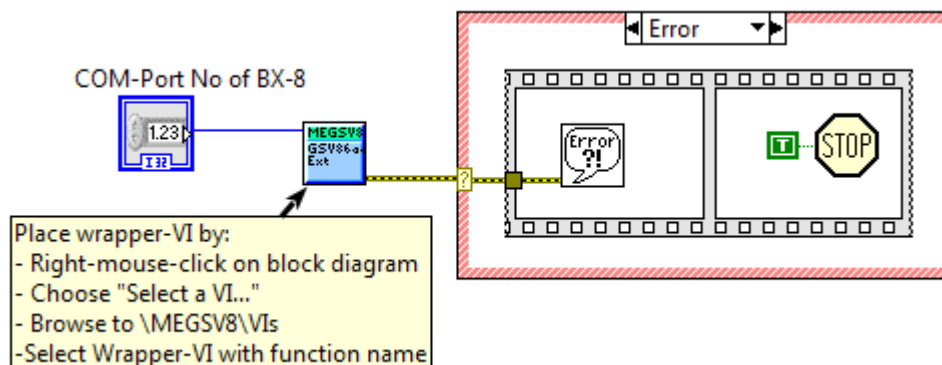


Figure 2: Simple initialization with LabView

Note: In case of an error or unsuccessful opening, the functions GSV86actExt / GSV86activateExtended initialize allocated resources for themselves, (exceptionally no need to call GSV86release).

Step 2: Reading measurement values using LabView

This can be done by calling one of the following

- DLL-functions: **GSVread** Read one measuring value of one analogue input channel
- GSV86readMultiple Read an array of various measuring values of one or all analogue input channels.

To call a dll-function, click on an empty area on the Block Diagram with the right mouse button. Then, choose "Select a VI..." and then, browse to the directory \MEGSV9\VIs (provided in the zip-archive of the GSV-8 API) and choose the VI with a similar name as the

¹ See Chapter 2

DLL-function – here GSV86read Multiple.vi.

By default, the GSV-8 device transmits measuring values permanently. The DLL receives these values with its own independent reading thread and stores them to a buffer. These read-functions copy values from that own buffer to the callers buffer (see chapter 2).

By default, the GSV-8 transmits measured values in a number format, that is scaled by a scaling factor. That factor should be adapted to the measuring task; e.g. to the sensor connected to the analogue input (see chapter 2).

The device stores its parameters (like this scaling factor) autonomously, so once configured correctly, the user doesn't have to care much about the parametrization (which itself is roughly described in chapter 2).

Here, for simplicity, it's assumed that the device is already configured correctly.

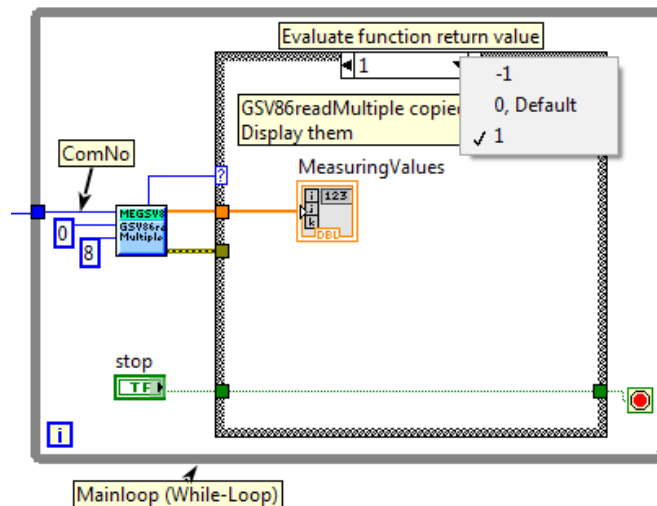


Figure 3: Simple main loop with LabView: Values to be displayed case

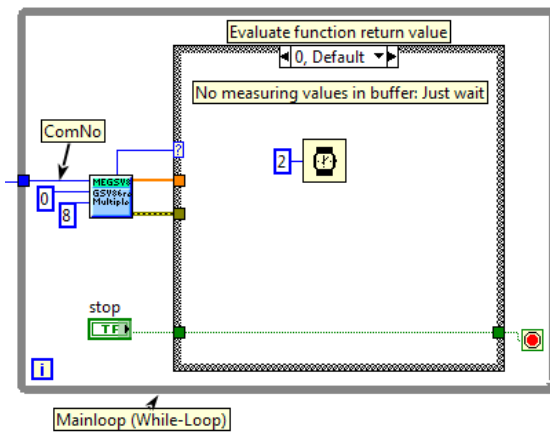


Figure 4: Simple main loop with LabView: No values received case

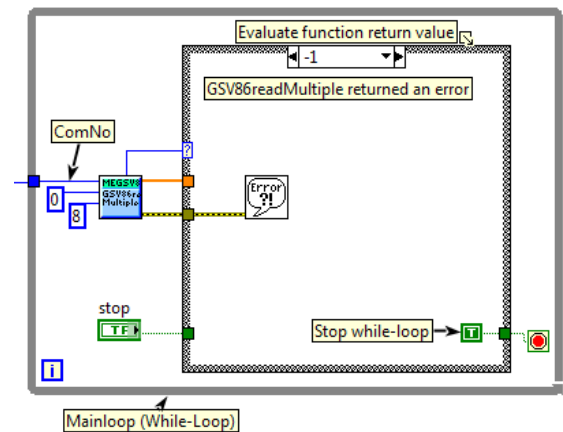


Figure 5: Simple main loop with LabView: Error case

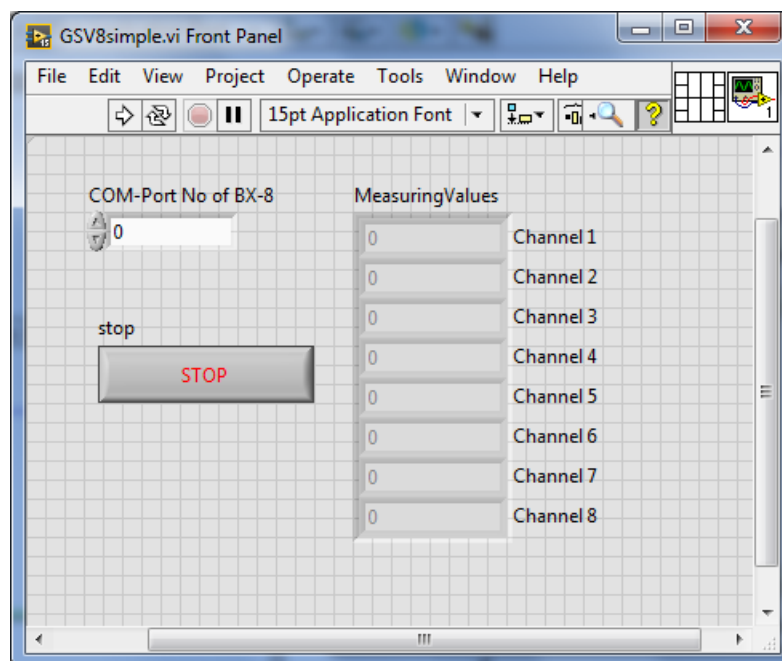


Figure 6: Simple program with LabView: Front panel

Step 3: Program termination using LabView

This must be done by using GSV86release.vi. See figure 7 depicting the complete simple program:

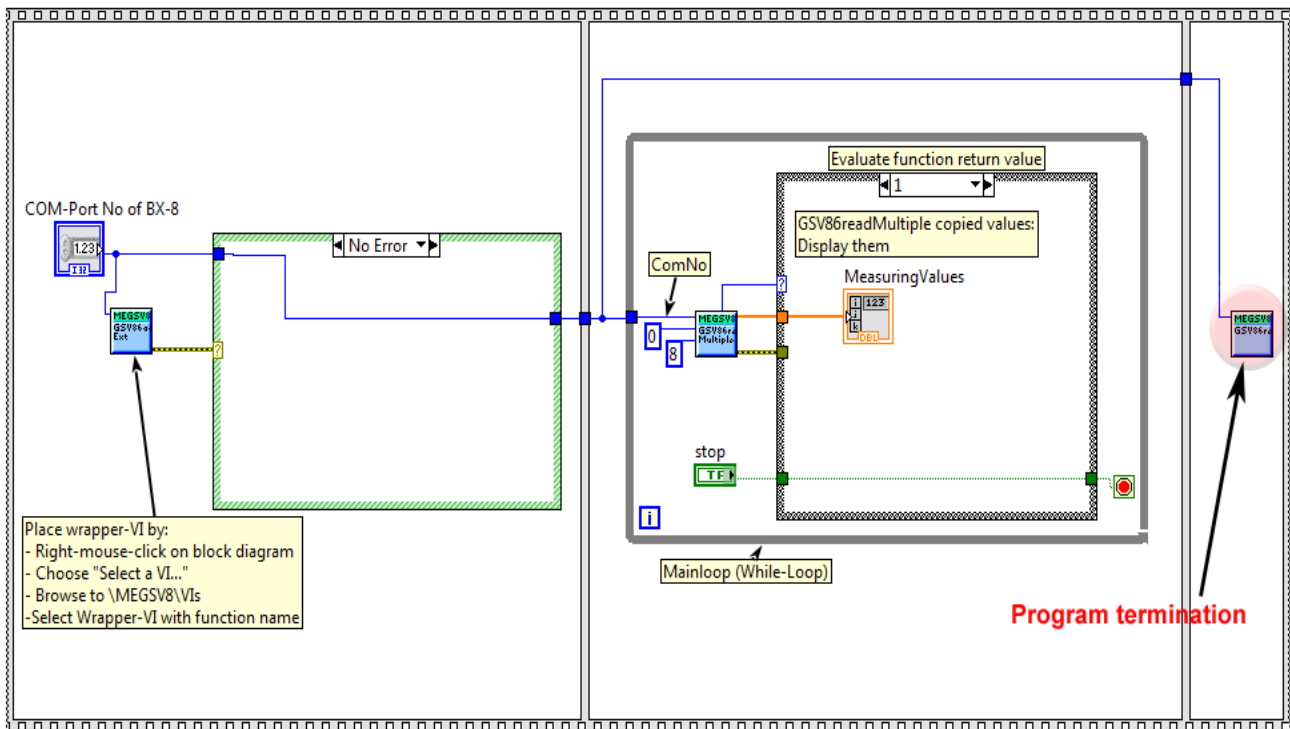


Figure 7: Simple Program, Variant 1

Implementation of a simple program using MEGSV86xx.dll and C language

With C, the DLL must be linked, so that in the build process the linker "knows" the entry points for the dll functions used. This can be done by two ways:

1. Static linking: To use this method, the file MEGSV86x32.lib (for 64-Bit version MEGSV86x64.lib), available in the DLLs zip-archive, must be included as an external resource file to the project in the IDE used. Also, the header MEGSV86w32.h (for 64-Bit version MEGSV86x64.h) must be included to the source files of the project in the IDE.
2. Dynamic linking: The DLL must first get loaded, e.g. by calling the Windows API function Load Library. On success, it returns a valid handle to the DLL. Then, for every function that is intended for use, the function entry point / procedure address must be retrieved; e.g. by calling the Windows API function GetProcAddress. The functions themselves are then used by their function pointer.

Step 1: Initialization using C

This is done by either calling GSV86actExt or GSV86activateExtended².

GSV86actExt is a simplified version with one parameter only, "ComNo" (see above).

The return value should be evaluated, because it encodes the important information whether the device could be successfully accessed or not.

² See Chapter 2

C example:

```
#include "MEGSV86w32.h" // "static" binding of dll; assumes that
                        // MEGSV86w32.lib is provided to the linker

int ComNo= 1; // must be set to correct value
char ErrTxt[ERRTEXT_SIZE]; // to hold explaining text in case of error
int retcode= GSV86actExt(ComNo);
if(retcode == GSV_ERROR)
{
    GSV86getLastErrorText(ComNo,ErrTxt); // get error reason info
    printf("Activation failed. Reason:\n%s",ErrTxt); // show error text
    return;
}
printf("Activation successful");
```

Note: In case of an error or unsuccessful opening, the functions GSV86actExt / GSV86activateExtended uninitialized allocated resources for themselves, (exceptionally no need to call GSV86release).

Step 2: Reading measurement values using C

This can be done by calling one of the following DLL-functions:

- **GSVread** Read one measuring value of one analogue input channel
- **GSV86readMultiple** Read an array of various measuring values of one or all analogue input channels.

By default, the GSV-8 device transmits measuring values permanently. The DLL receives these values with its own independent reading thread and stores them to a buffer. These read-functions copy values from that own buffer to the callers buffer (see chapter 2).

By default, the GSV-8 transmits measured values in a number format, that is scaled by a scaling factor. That factor should be adapted to the measuring task; e.g. to the sensor connected to the analogue input (see chapter 2).

The device stores its parameters (like this scaling factor) autonomously, so once configured correctly, the user doesn't have to care much about the parametrization (which itself is described in chapter 2).

Here, for simplicity, it's assumed that the device is already configured correctly.



C example: using GSVreadMultiple (continuing above example for step 1):

```
int i, count, ret;
double MeasuringValues[8];

while(1)
{
    ret= GSV86readMultiple(ComNo, 0, &MeasuringValues[0], 8, &count, NULL);
    if(ret == GSV_TRUE)
    {
        for(i=0; i<8;i++)
            printf("%.3lf\t",MeasuringValues[i]); //show values
        printf("\n"); //new line
    }
    else if(ret == GSV_OK)
    {
        Sleep(2); //no values available this moment: suspend to limit CPU load
    }
    else if(ret == GSV_ERROR)
    {
        GSV86getLastErrorText(ComNo,ErrTxt); //get error reason info
        printf("GSV86readMultiple failed. Reason:\n%s",ErrTxt); //show error
        GSV86release(ComNo); //Free ressources
        Exit(0);
    }
}
```

Step 3: Program termination using C

This must be done by calling the DLL-funtion GSV86release(int ComNo):\

```
GSV86release(ComNo);
```

Simple program using MEGSV86xx.dll and C# language

Using unmanaged dynamic-link libraries from managed code like C# requires entry point declarations of all the functions being called from C#.

To use DLL in your project, you will need to copy MEGSV86xx.dll into project directory ..\bin\x86\Debug\ or ..\bin\x64\Debug\ or ..\release, respectively and import the System.Runtime.InteropServices namespace. **Follow these steps:**

1. Identify required functions in the DLL.
2. Create a class to hold DLL functions.

You can use an existing class, create an individual class for each unmanaged function, or create one class that contains a set of related unmanaged functions.

3. Create prototypes in C# managed code.

Use the DllImport attribute to identify the DLL and function. Mark the method with the static and extern modifiers.

4. Call a DLL function.

Call the method on your managed class as you would with any other managed method.

For more information, see:

<https://docs.microsoft.com/en-us/dotnet/framework/interop/consuming-unmanaged-dll-functions>

C# Example:

```
using System.Runtime.InteropServices;
namespace megsv86_cs
{
    class Program
    {
// Use DllImport to import the Win32 GSV86actExt function.
        [DllImport("MEGSV86w32.dll", CharSet=CharSet.Unicode,
CallingConvention=CallingConvention.StdCall)]
        public static extern int GSV86actExt(int ComNo); // Use DllImport to
import the Win32 GSV86actExt function.
        [DllImport("MEGSV86w32.dll", CallingConvention =
CallingConvention.StdCall)]
        public static extern int GSV86getLastErrorText(int ComNo, StringBuilder
ErrText);
        [DllImport("MEGSV86w32.dll", CharSet = CharSet.Unicode,
CallingConvention = CallingConvention.StdCall)]
        public static extern int GSV86read(int ComNo, int Chan, ref double
outvar);
    }
}
```



```
[DllImport("MEGSV86w32.dll", CharSet = CharSet.Unicode,
CallingConvention = CallingConvention.StdCall)]
    public static extern int GSV86readMultiple(int ComNo, int Chan, ref
double outvar,int count,ref int valsread,ref int ErrFlags);
    [DllImport("MEGSV86w32.dll", CharSet = CharSet.Unicode,
CallingConvention = CallingConvention.StdCall)]
    public static extern int GSV86release(int ComNo);

    static void Main(string[] args)
    {
        int ComNo = 15; //The COM-port number of a serial device.
//allocate a char array of fixed size to hold the error text
        StringBuilder errText = new StringBuilder(new String(' ', 256));
//Open a GSV8 on COM-port, specified by parameters, with all necessary
resources.
        int retcode = GSV86actExt(ComNo);
        if (retcode == -1) //an error occurred within the request
        {
            GSV86getLastErrorText(ComNo, errText); //get error reason info
            Console.WriteLine("Activation failed. Reason: ");
            Console.WriteLine(errText);
            Console.ReadLine();
            return;
        }
        Console.WriteLine("Activation successful \n");

//Reading measurement values
        int count=0 , ret;
        int errFlags = 0;
        double[] MeasuringValues = new double[8];
        while (true)
        {
            ret = GSV86readMultiple(ComNo, 0, ref MeasuringValues[0], 8, ref
count, ref errFlags);
            if (ret == 0) //GSV_TRUE
            {
                for (i = 0; i < 8; i++)
                {
                    //show values
                    Console.WriteLine("{0:0.000}", MeasuringValues[i] );
                }
            }
        }
    }
}
```

```
    }
    Console.WriteLine("\n"); //new line
    Thread.Sleep(1000);
    Console.Clear();
}
else if (ret == 1) // GSV_OK
{
//no values available this moment: suspend to limit CPU load
    Thread.Sleep(20);
}
else if (ret == -1) // GSV_ERROR
{
    GSV86getLastErrorText(ComNo, errText); //get error reason
info
    Console.WriteLine("GSV86readMultiple failed. Reason:\n",
errText); //show error
    GSV86release(ComNo); //Close COMport and free all associated
dll-ressources
}
if (Console.KeyAvailable)
{
    break;
}
}
}
}
```

Chapter 2: Additional information on MEGSV86xx.DLL

If one of the initialization functions GSV86actExt or GSV86activateExtended succeeded, for each new COM-port number, they did the following:

1. Open a serial port. This can be a virtual port if using the device's USB port. The USB port requires a driver, see below.
2. Allocate memory on the heap, for using buffers and other variables.
3. Create an own thread, used for reading every data the device sends, which are measuring values and command answers. Measuring values and command answers are stored in different particular buffers.



Driver for using USB port

Generally, the required driver (usbser.sys) is provided by Microsoft and is available by Windows. The method to load and activate it depends on the Windows version used:

Windows 10: No installation steps necessary - Just plug device and wait.

Windows 8,7,Vista,XP: Execute the driver setup package (download it from www.me-systeme.de). It will link the GSV-8 to the required driver, so that Windows will "know" the plugged device afterwards.

Initialization step

GSV86actExt is a simplified version of **GSV86activateExtended**. The latter has the following calling parameters:

Bitrate: Bits per second. Allowed values:

9600,19200,38400,57600,115200,230400,460800,921600

- Must match device setting with Ethernet-Xport, RS232 and RS422 only;
- but can be any of above values with USB-CDC (will be discarded).

BufSize:

- Size of the buffer, given in number of measuring values.
- In this buffer measuring values are stored by the reading thread of this DLL and from where values are read by GSV86read/GSV86readMultiple.
- Size is in Measuring values per channel-object; e.g. with MappedObjectNum=8 and BufSize=30000,
- 8 buffers are allocated, and each has a capacity of 30000 measuring values.

flags:

Opened with special settings, specified by this flags. Can be ORed together.

- ACTEX_FLAG_HANDSHAKE =0x0004: If set, opens the port with hardware-handshake (RTS-CTS), reserved.
- ACTEX_FLAG_WAIT_EXTENDED =0x0100: Waits longer for device-answer
- ACTEX_FLAG_STOP_TX =0x0200: Stops continuous data transmission.

GSV86actExt calls **GSV86activateExtended** with the following parameters:

- Bitrate=CONST_BAUDRATE = 115200d
- BufSize=CONST_BUFSIZE = 48000d
- flags=0

Parametrization of general-purpose sensors

In order to read measuring values that represent physical values correctly, some parameters have to be adapted **once** - the device stores and restores its parameters autonomously. So,

as a good programming practise, the user may read relevant parameters first. Then, if they do not suit to the desired application/sensor/measuring task, they are changes by the adequate writing functions.

Parameters for general-purpose sensors are:

Input Type

That is the type of the analogue input. There are 5 different ones available:

0: Wheatstone-Bridge input. Excitation voltage 8.75V, input range ± 2 mV/V for standard version

1: Wheatstone-Bridge input. Excitation voltage 5V, input range ± 3.5 mV/V for standard version (default setting)

2: Wheatstone-Bridge input. Excitation voltage 2.5V, input range ± 7 mV/V for standard version

3: Single-ended voltage input with full-scale value of 10 V

4: PT1000 Thermocouple sensor for temperature measurement only

The input type also defines the total measurement range, which is from:

input range (=full-scale value) * -1.05 to

input range (=full-scale value) * 1.05

That means that there's 5% range overhead.

DLL-Functions for reading:

`GSV86getInTypeRange`

Normally, that one is used. It returns the Input type as listed above, and also writes the nominal input range to out value passed by pointer (see `GSV86DLL_referenceManual.pdf`)

`GSV86getAllInTypesRanges`

The latter is for informational purpose, since possibly (for device varieties) the input ranges may differ for the types listed above.

DLL-Function for writing:

GSV86setInType

With this function, the Input Type, as listed above, can be set.

If the full measuring range of the sensor shall be used, the BX-8 input range should at least match the electrical full-scale output of the sensor. With BX-8, the input-range of bridge sensors (types 0..2) also sets the sensor excitation voltage. For light-weight sensors, the excitation voltage of 8.75V has the disadvantage that due to heating of the sensor, a slight drift of the sensor signal may result. With 5V excitation voltage and below, that doesn't



happen for almost all bridge sensors.

Example:

Electrical full-scale output of the bridge sensor connected to input channel 1 of device at ComNo 9: 1.9mV/V

Perform as recommended (pseudo-c-code)

```
        type= GSV86getInTypeRange(9,1,&range)
if(type > 1)
    ret= GSV86setInType(9,1,1);
    if(ret==GSV_ERROR)
        GSV86getLastErrorText
```

Display scaling

In order to yield values correctly scaled in physical units, the raw measuring values must be multiplied by a factor. If the measuring value data type is FLOAT, that is done by the device itself. Otherwise, the user application program should multiply the raw values obtained by GSV86read or GSV86readMultiple with these factors. The DLL doesn't do that.

DLL-Functions for reading:

GSV86getValObjectInfo

With that function, the scaling factors of all input channels can be read, independently of value data type, so this is the most convenient function if used at program initialization, because the user doesn't have to care about value data type: Just use these factors and always multiply the values from GSV86read or GSV86readMultiple with them.

GSV86readUserScale

Reads scaling factors, that are used internally used by device in case of value data type = FLOAT, or externally otherwise.

DLL-Function for writing:

GSV86writeUserScale

Writes scaling factors, that are used internally used by device in case of value data type = FLOAT, or externally otherwise.

If calibration data for a given sensor is available, the scaling factor can be calculated as follows:

UserScaling= (Physical full-scale value / Electrical full-scale value) * input range

The input range can be read by GSV86getInTypeRange, see above.

Example:

Electrical full-scale output of the bridge sensor connected: 1.9 mV/V

Physical full-scale value of the bridge sensor connected: 1000 N (force)

Input range read by GSV86getInTypeRange: 3.5mV/V

UserScaling= (1000N / 1.9mV/V) * 3.5mV/V = 1842.105263N

C example using InitSensor (shown below), continuing above example for step 1:

```
printf("Activation successful"); //continuing from step 1, chapter 1
if(InitSensor(ComNo, 1, 1.9, 1000.0) == GSV_OK) //example for input channel 1
only
    printf("InitSensor successful");
else {
    printf("InitSensor error");
    GSV86release(ComNo); //Free resources
    Exit(0); }
double ScaleFactors[8];
int NumObj; // number of channel-objects in value-frame. Default=8
NumObj= GSV86getValObjectInfo(9, &ScaleFactors[0], NULL, NULL); // fkt. returns
num. of input channels
/* main loop for reading measuring values scaled in physical units */

while(1)
{
    ret= GSV86readMultiple(ComNo, 0, &MeasuringValues[0], NumObj, &count,
NULL);

    if(ret == GSV_TRUE) {
        for(i=0; i<NumObj;i++)
            printf("%.3lf\t", MeasuringValues[i] * ScaleFactors[i]);
        printf("\n"); //new line
    }
    else if(ret == GSV_OK) {
        Sleep(2); //no values available this moment: suspend to limit CPU
load
    }
    else if(ret == GSV_ERROR) {
        GSV86getLastErrorMessage(ComNo,ErrTxt); //get error reason info
        printf("GSV86readMultiple failed. Reason:\n%s",ErrTxt); //show error
        GSV86release(ComNo); //Free resources
        Exit(0);
    }
}
```



C function InitSensor for sensor parametrization

```
/* @brief Check, compare and eventually change amp's input range and user
scaling
    according to bridge sensor parameters
@param[in] ComNo:    Number of Device Comport
@param[in]   Chan:  Number of input channel 1..8
@param[in] ElecFSval: Electrical full-scale output of the bridge sensor
connected
@param[in] PhysFSval: Physical full-scale value of the bridge sensor
connected
@return Simple errorcode: GSV_OK,if successful or GSV_ERROR if function
failed
    remark: Works for standard input ranges 2 / 3.5 / 7 mV/V only */
int InitSensor(int ComNo, int Chan, double ElecFSval, double PhysFSval)
{
    double InRange, ScaleDesired, ScaleAct;
    int typeDesired, typeAct, ret;
    char ErrTxt[256]="Emty";

    if(ElecFSval<=0 || PhysFSval<=0)
        return GSV_ERROR;
    if(ElecFSval > 3.5)
        typeDesired=2;
    else typeDesired=1;
    typeAct= GSV86getInTypeRange (ComNo,Chan,&InRange)
    if(typeAct==GSV_ERROR)
    {
        ret=GSV86getLastErrorText (ComNo,ErrTxt);
        printf("%s",ErrTxt);
        return ret;
    }
    if(typeAct != typeDesired)
    {
        ret= GSV86setInType (ComNo,Chan,typeDesired);
        if(ret==GSV_ERROR)
        {
            ret=GSV86getLastErrorText (ComNo,ErrTxt);
            printf("%s",ErrTxt);
            return ret;
        }
    }
}
```

```
typeAct= GSV86getInTypeRange (ComNo,Chan, &InRange)
}
ScaleDesired= (PhysFSval * InRange) / ElecFSval;
ret= GSV86readUserScale (ComNo,Chan, &ScaleAct);
if (ret==GSV_ERROR)
{
ret=GSV86getLastErrorText (ComNo,ErrTxt);
printf ("%s",ErrTxt);
return ret;
}
if ((ScaleAct+(ScaleAct/1E5)) > ScaleDesired && (ScaleAct-(ScaleAct/1E5)) <
ScaleDesired)
ret= GSV_OK; //scale inside 1E-5 interval of desired value: ok
else
{
ret= GSV86writeUserScale (ComNo,Chan, ScaleDesired);
if (ret==GSV_ERROR)
{
ret=GSV86getLastErrorText (ComNo,ErrTxt);
printf ("%s",ErrTxt);
}
}
return ret;
}
```

LabView example using InitSensor (shown below) and improving above example

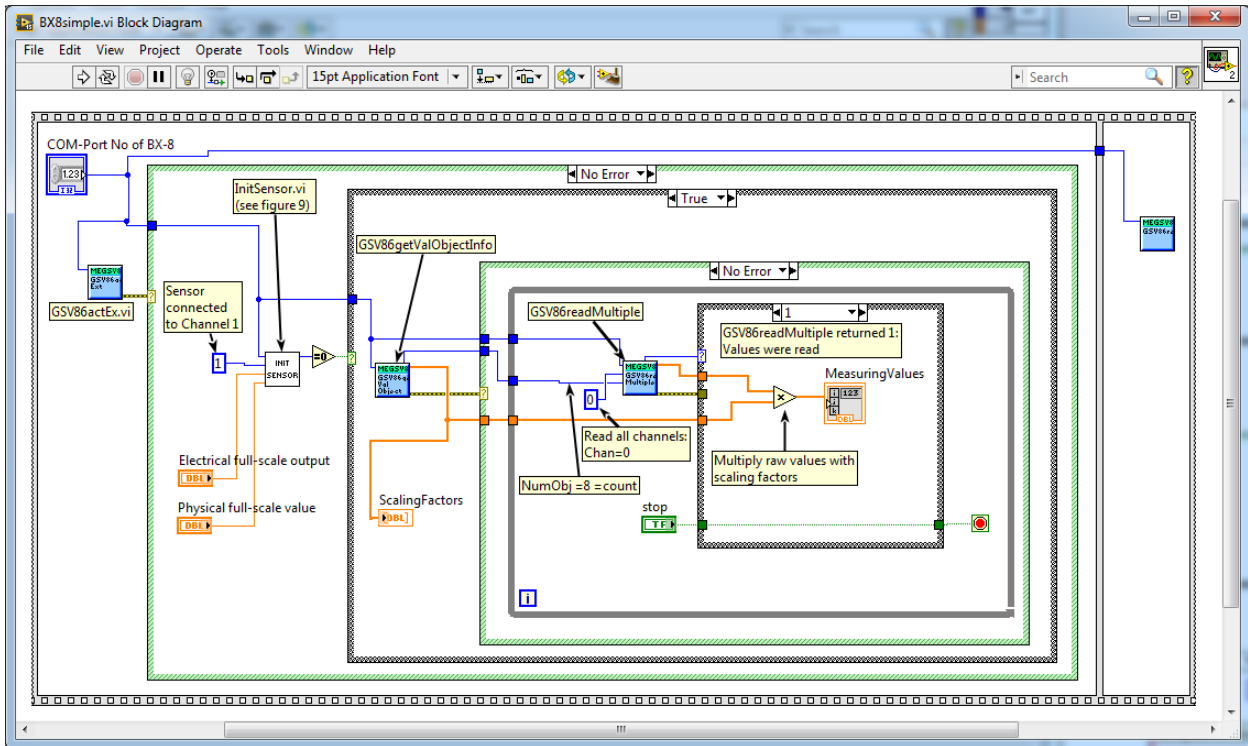


Figure 8: Simple program, improved

LabView VI InitSensor for sensor parametrization:

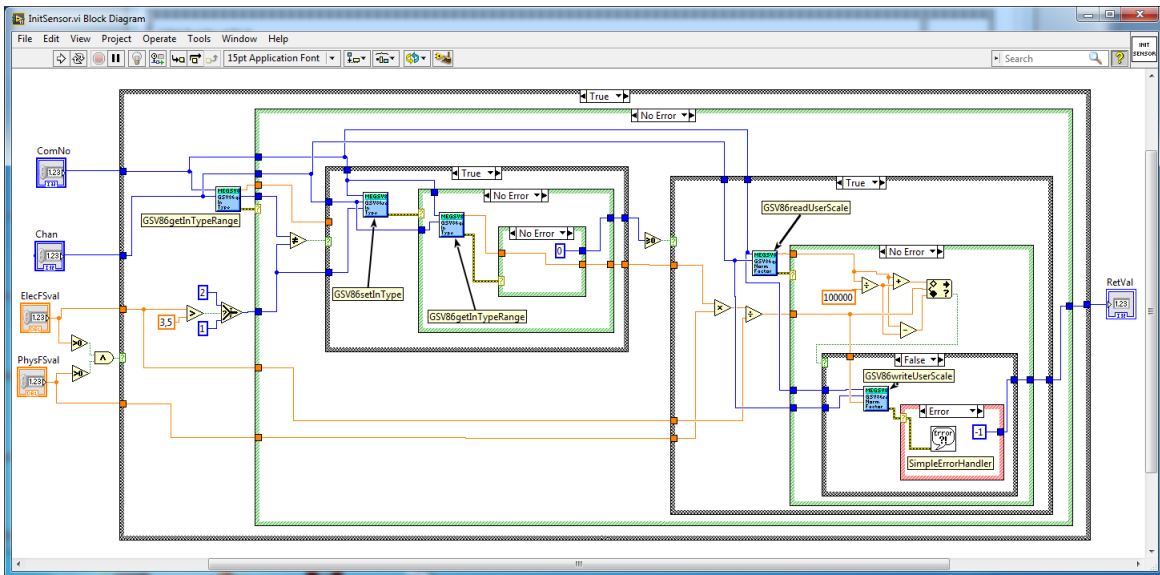


Figure 9: Example for General bridge sensor parametrization