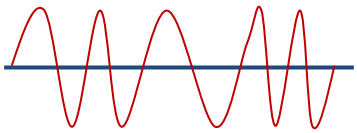
HartTools 7.5

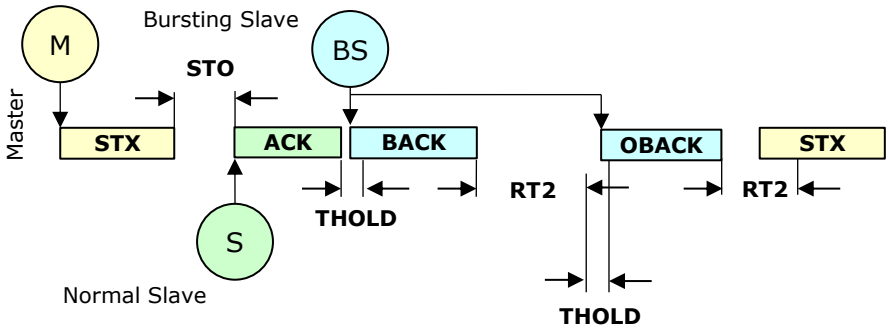
Software Documentation

Revision: 7.5.1
Date: 31.5.2018



Real Time
WINAPI

Software Solutions for
Hart Instrument Development
and Automation with Windows



Borst Automation
Neue Reihe 33
DE-27472 Cuxhaven
GERMANY

Fon: +49 (0)4721 6985100
Fax: +49 (0)6432 6985102

http://www.borst-automation.de

info@borst-automation.de

Contents

Overview

HartTools is a set of components used to provide applications based on Hart communication on a Windows computer.

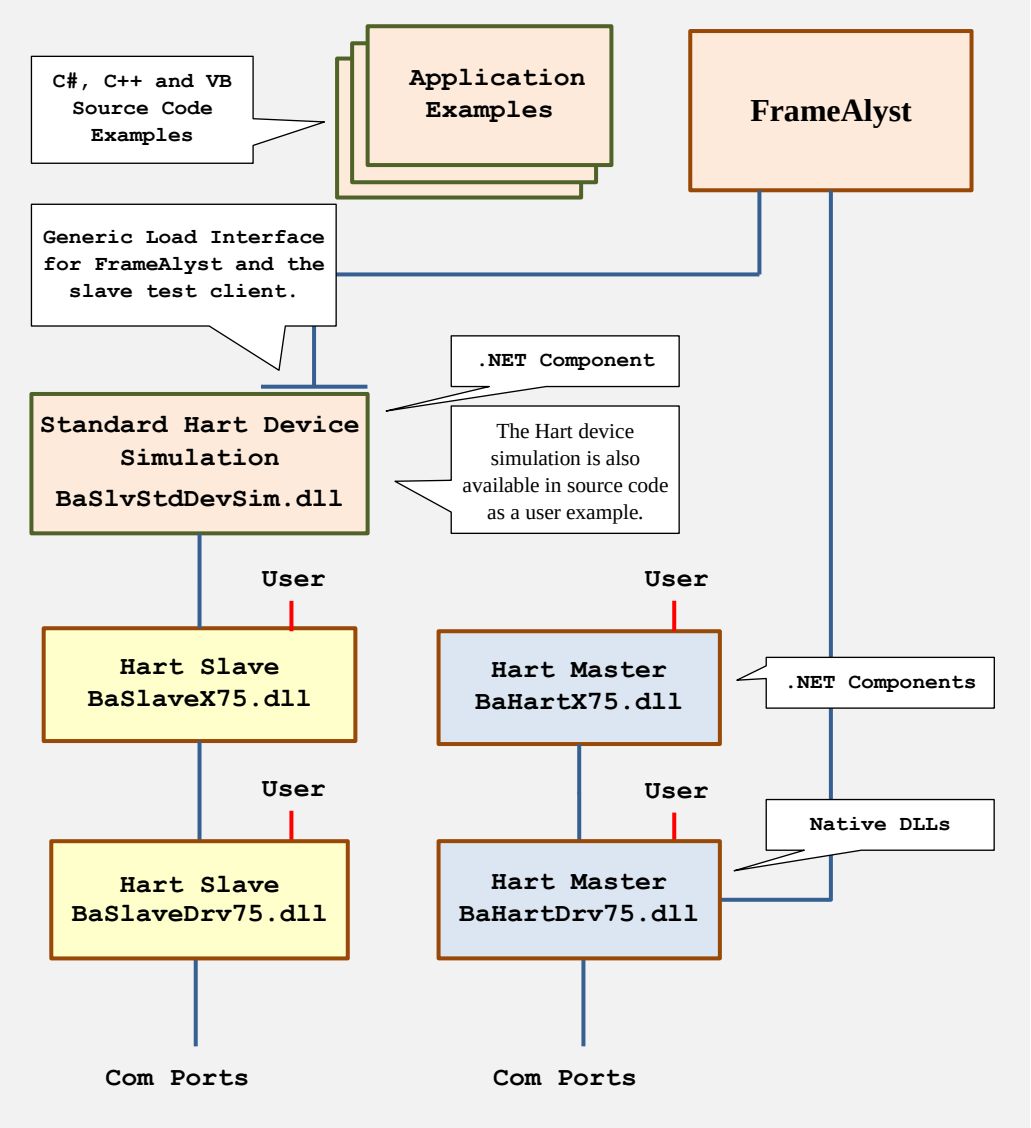


Figure 1: Components Architecture of Hart Tools 7.5

The Hart Tools are based on two native Windows DLLs. One for the master functionality and the other one for the slave services. For both a .NET component is provided.

The user may integrate the native DLLs or the .NET components into his application.

FrameAlyst is a standard application for monitoring and analyzing the communication streams. FrameAlyst is docking at the Hart Master DLL (BaHartDrv75.dll).

Because native DLLs can only be provided as 32 or 64 bit assemblies, both versions are available in the packet.

Component	Path	CPU	Description
BaHartDrv75.dll	.\UserDLLs\System\x86\ WindowsSystem(32 bit)	x86	The Hart master DLL is also providing a monitor interfaces for FrameAlyst and for the user.
	.\UserDLLs\System\x64\ WindowsSystem(64 bit)	x64	
BaSlaveDrv75.dll	.\UserDLLs\System\x86\ WindowsSystem(32 bit)	x86	The Hart slave DLL is providing function s which are needed by a Hart command interpreter.
	.\UserDLLs\System\x64\ WindowsSystem(64 bit)	x64	
BaHartX75.dll	.\UserDLLs\App\ .\Debug\ .\Debug(86)	Any	The .NET Hart master component is an additional shell to the master DLL.
BaSlaveX75.dll	.\UserDLLs\App\ .\Debug\ .\Debug(86)		The .NET Hart slave component is an additional shell to the slave DLL.
BaSlvStdDevSim.dll	.\ .\UserDLLs\App\ .\Debug\ .\Debug(86)		The standard Hart device simulation serves to purposes. One is to provide a slave simulation to FrameAlyst and to provide an example of a slave device simulation for the user.
BaHartFrameAlyst75.exe	.\ .\UserDLLs\App\ .\Debug\	Any	The FrameAlyst is the main application of the Hart Tools package.
	.\Debug(86)	x86	A 32 bit compilation of the application is provided to allow 32 bit debugging on a 64 bit machine.

Table 1: Components and Paths

Installation

The installation may be done into any directory. The solutions for the example applications are available at the path .\Examples\.

Note: The projects of the examples were generated with Visual Studio 2013. Trying the examples with an earlier Version of Visual Studio will not work.

On 64 bit platforms the installation provides the subdirectory .\Debug(x86) for debugging 32 bit applications on a 64 bit platform.

On 32 bit platforms the path .\Debug(x86) is not available because all applications and components which are compiled for Any CPU are automatically loaded as 32 bit modules.

Application Examples

Example	Subject	Description
HartDLL		
C#		
AppDeviceData.sln	Device Data Manager	This is a more complex example implementing the handling of data of various kinds.
ConnectAndRead.sln	Connection, Device Info	The Example demonstrates the usage of the connection information and the BHDrv_IsServiceCompleted method.
CsGetCyclicData.sln	Cyclic Data Callback	The example is showing how cyclic data is collected from the HartDLL (burst mode handling). The polling and the callback mechanisms are demonstrated.
GetUnIDbyTag.sln	Data Link Service	The example demonstrates the usage of the function BHDrv_ConnectByTagName of the HartDLL.
MultiThreadingDLL.sln	More than one Thread	The example demonstrates how to use several threads for Hart communication with the HartDLL. Two worker threads are used.
CsRdWrRangeAndTag.sln	Read and Write Data	In Hart commands usually more than one parameter is communicated. Here the handling is demonstrated.
SendExtCommand.sln	Hart 7, Service Callback	Sends a 16 bit command and demonstrates the use of the service callback for service completion.
C/C++		
UsingBaHartDrv.sln	BaHartDrv75.h	A little console application interfacing to the DLL.
Microsoft Office		
UsingHartDLL.xlsm	VBA Macros	Excel can be used to communicate through a Hart Network.
Visual Basic		
VbRdWrRangeAndTag.sln	VB Language	The example is showing the use of HartDLL is used in Visual Basic.
HartX		
C#		
CsUsingHartX.sln	.NET Objects	Demonstrates how to use Hart as a .NET object.
MultiThreadingX.sln	More than one Thread	Demonstrates how several instances of HartX are handled.
Microsoft Office		
ReadPVs.xlsm	Collecting Data	The example reads the dynamic values from a Hart slave.
Visual Basic		
VbUsingHartX.sln	Using .NET in VB	The example how the HartX is integrated into a VB application.
SlaveX		
UserDevSimSlave.sln	Salve Device Simulation	It is much easier to develop the logic of a device in a PC simulation using Visual Studio. The solution is containing two projects. One for a user slave simulation and another one for a simple test client.

Table 2: Examples for the HartDLL, HartX and SlaveX

Directory Structure

After installation the following directory structure is created.

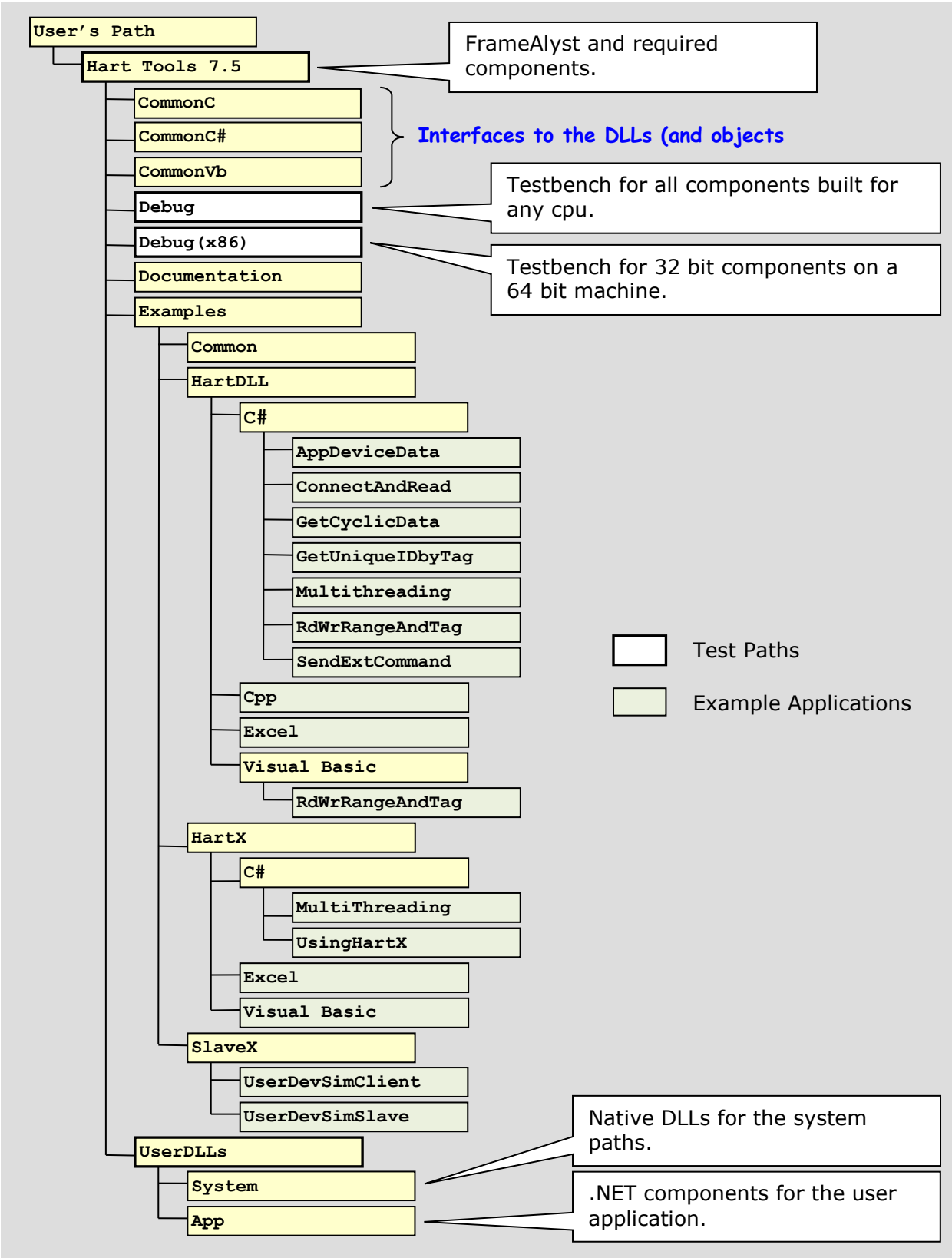


Figure 2: Directory Structure after Setup

Specifications

FrameAlyst

When the development of FrameAlyst was started it was mainly targeted to simply monitoring Hart frames to detect errors in the device implementation.

Later the tool was expanded to use the HartDLL for the emulation of a master function.

In the recent years also a slave emulations were introduced. While in the latest implementation either a slave or a master emulation was available today the new FrameAlyst is supporting both functionalities at a time.

Features

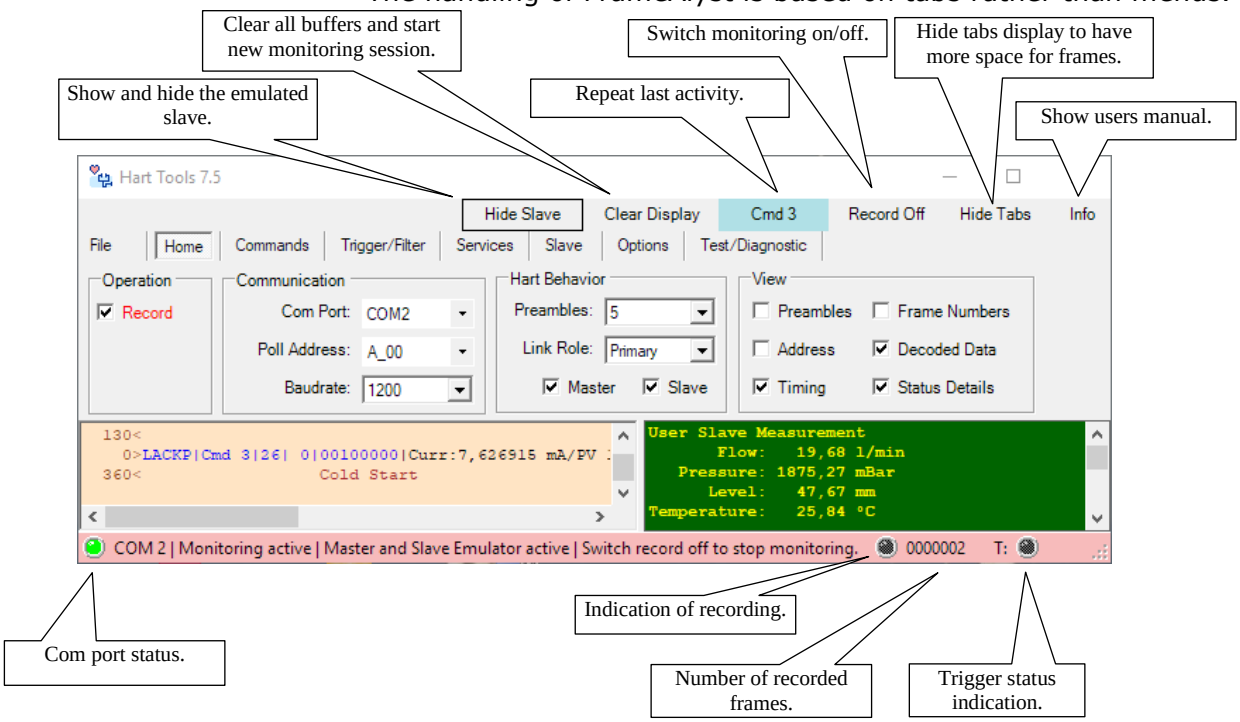
The main features which are supported by FrameAlyst are the following.

- Master emulation
- Slave emulation
- Slave DLL interface
- Trigger functions
- Filter functions
- Scripting
- Command data decoding
- Storing recorded data
- Test and diagnostic functions
- Integrated services
- Coding and Decoding
- Data syntax editor
- Data logging in xml-format

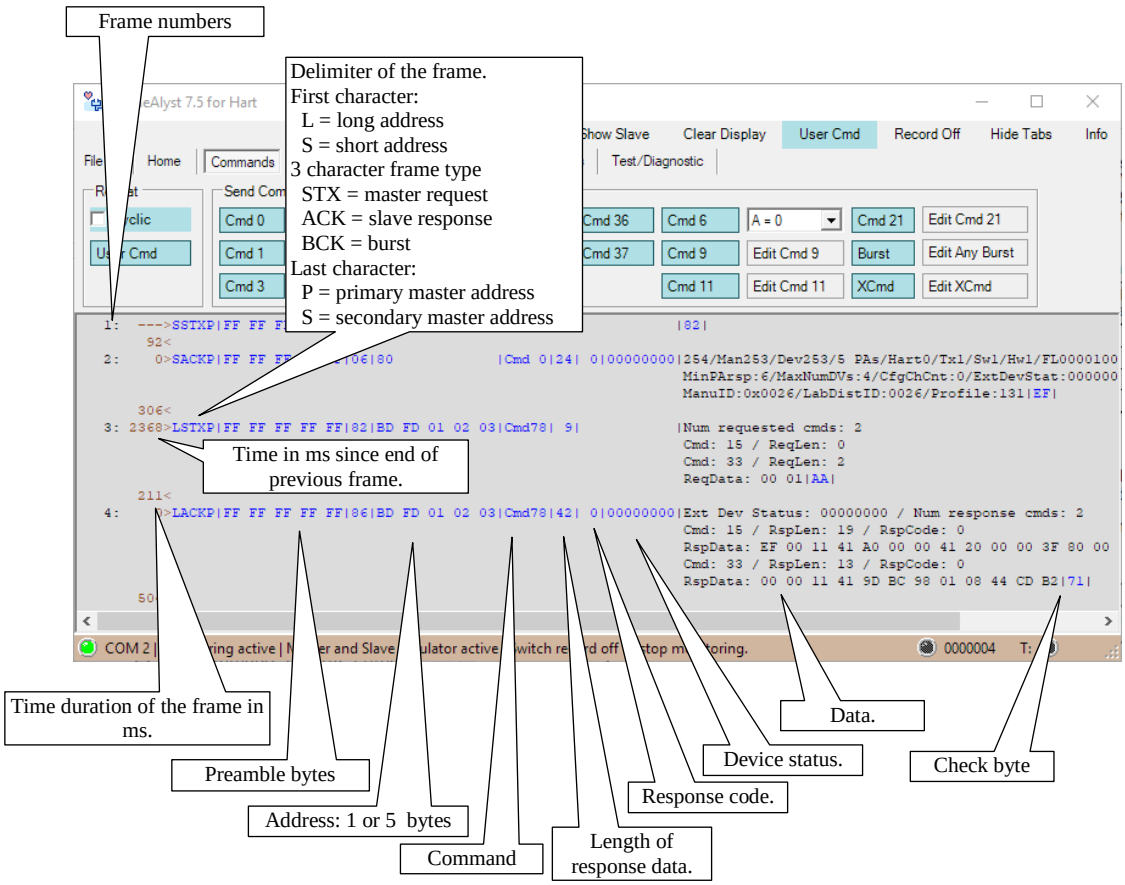
Functions and Menus

Common Elements

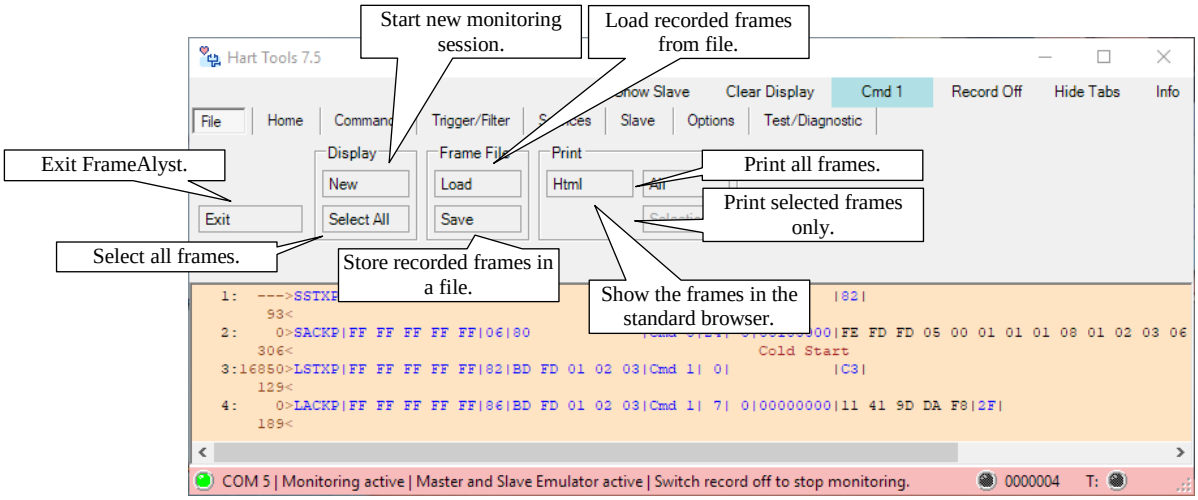
The handling of FrameAlyst is based on tabs rather than menus.



Display Items (Frames)

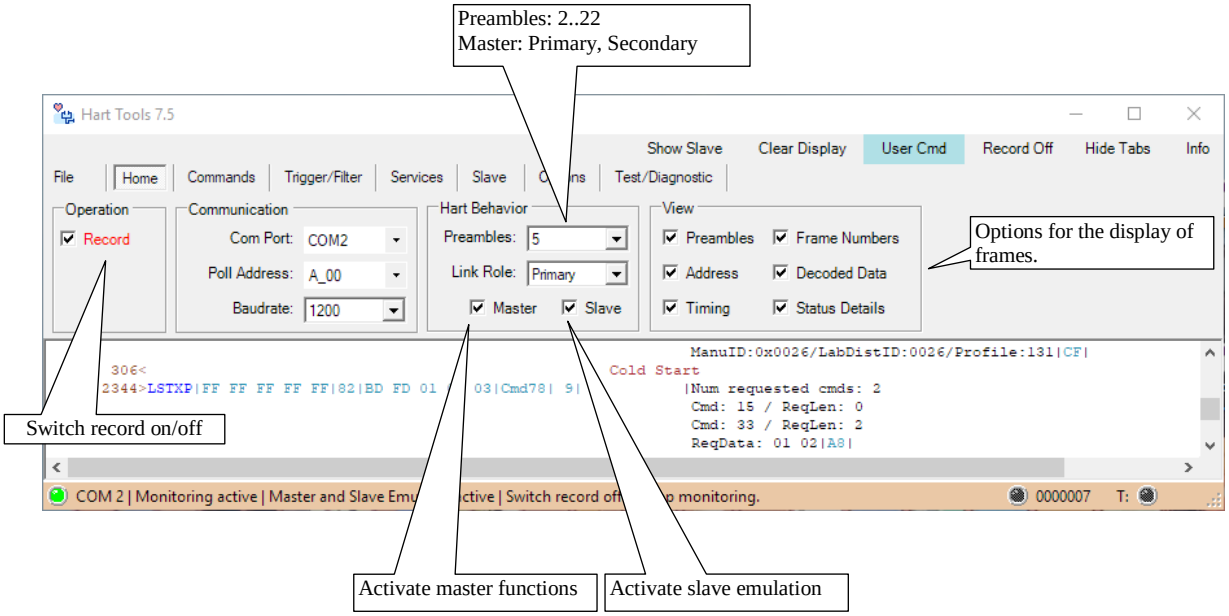


File Menu

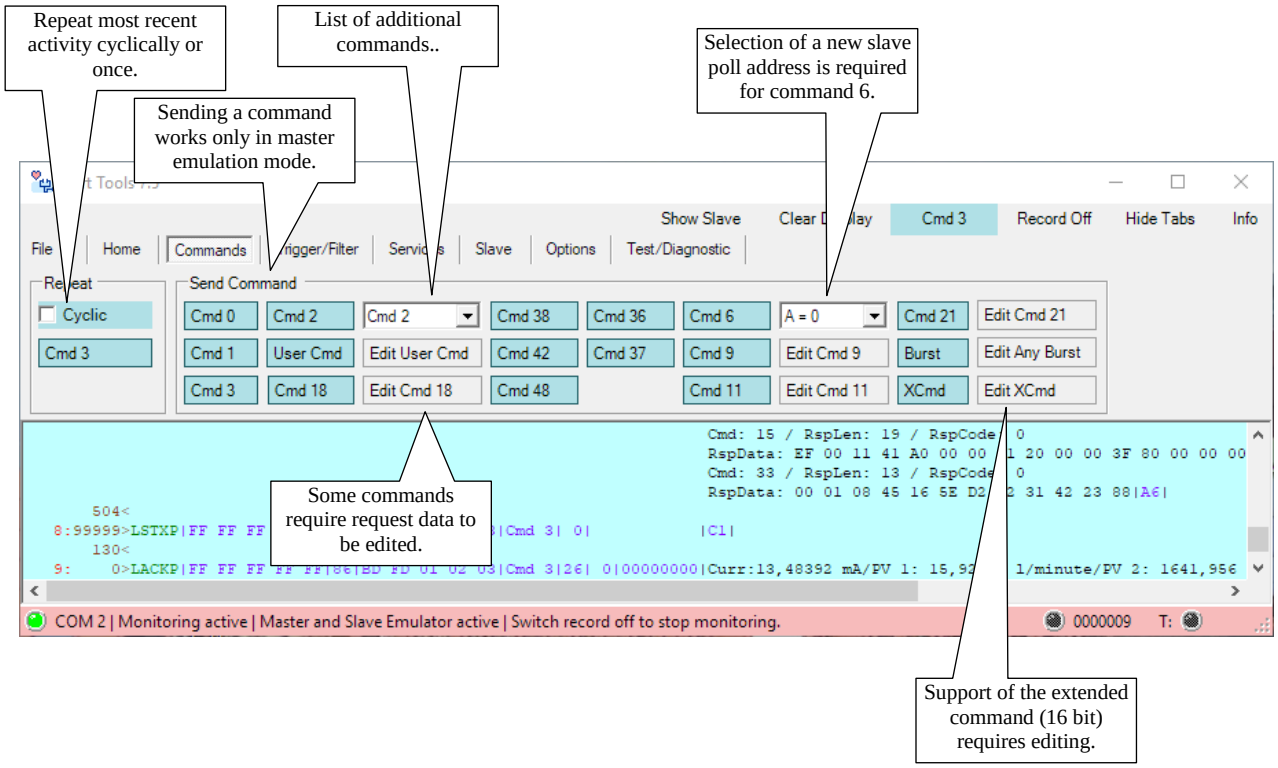


The frames are still stored in the format which was used in the past. However when saving the frame data you may also select an xml format or html format.

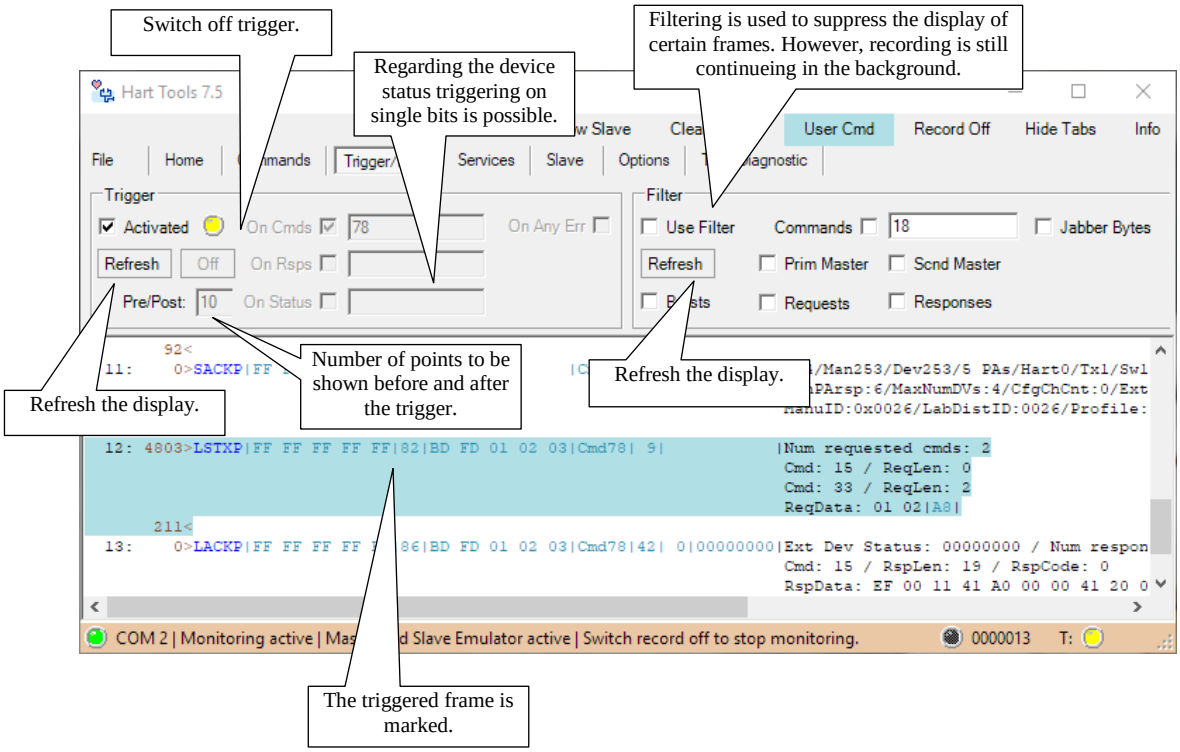
Home Menu



Hart Commands Menu

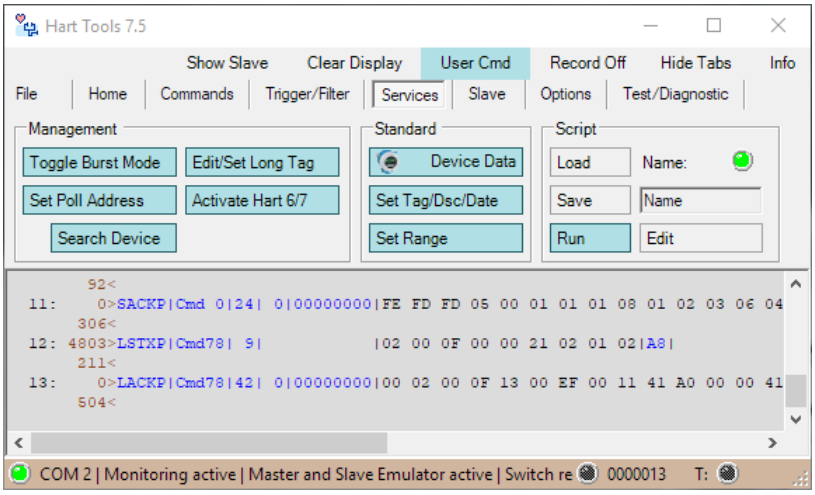


Trigger/Filter Menu



Services Menu

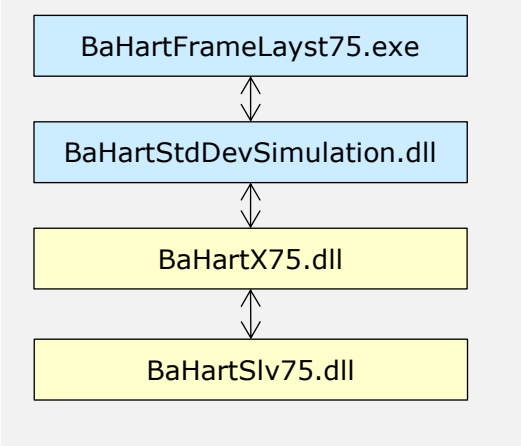
Services are some more complex functions as only sending a command.



The services are only working if the FrameAlyst is using the master emulation.

Details are explained in the chapter ‘Getting Started’.

Slave Menu

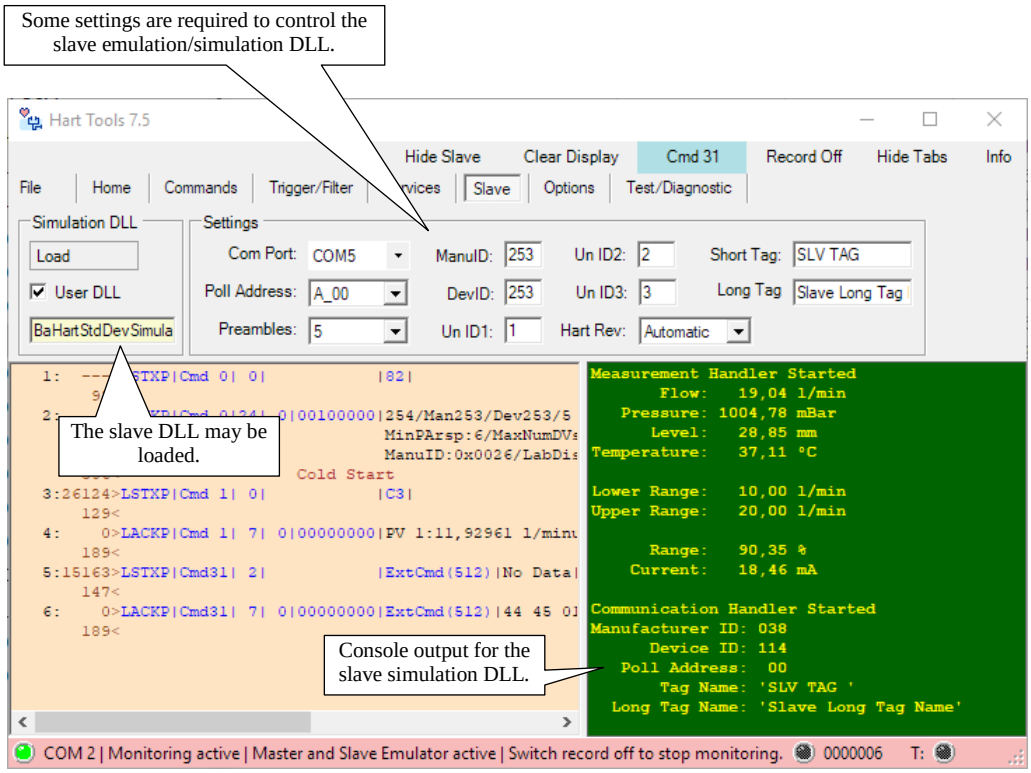


FrameAlyst able to load a DLL for the simulation of a slave device. This DLL is a class library written in C#. Thus it is also possible for the user to provide another slave device DLL written in C#.

The device simulation uses BaHartX75.dll which is a shell for the native library BaHartSlv75.dll.

Figure 3: Slave Emulation Architecture

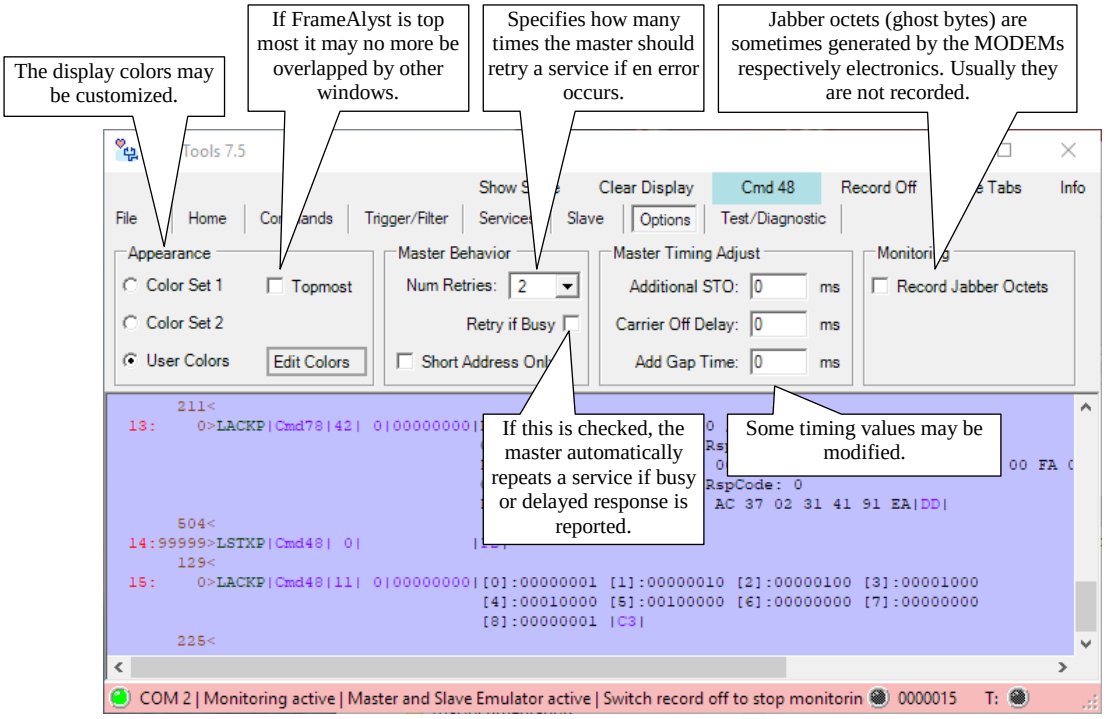
The slave may be configured through FrameAlyst.



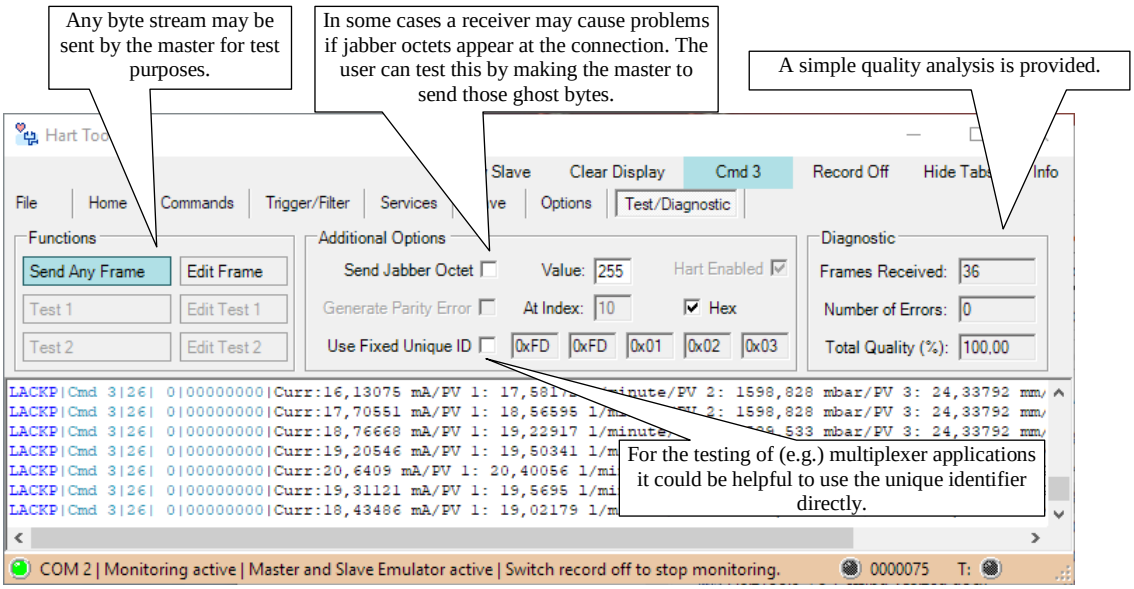
The slave interface of the HartDLL allows the developer of a Hart master device to simulate any slave functionality and any erroneous behavior of a Hart slave device.

Because the slave is running through a com port it can be be part of a multidrop environment.

Options Menu



Test/Diagnostic Menu



The above display was generated by using the filter for the suppression of requests.

HartDLL

The Hart Driver DLL is implementing the Hart communication protocol by resolving the real time requirements.

The DLL is not (!) using any framework like MFC. It does not use the Windows Registry and is not depending on any other DLL except the standard Windows system DLLs. The DLL itself is using standard Windows API calls and is therefore compatible to all Versions of Windows with the 32 bit and 64 bit API.

The implementation of the Hart Protocol does not contain any restriction to frame lengths like in Hart 5.x (e.g.). Therefore the all communication functions can be used for devices supporting Hart 5, Hart 6 or Hart 7.

Before using the communication the application software has to register for a com port of the PC. This can be any com port from 1 to 255 including virtual com ports as they are used for USB like the hart modems of Microflex.

Distribution of Applications

The only thing you have to provide with your application is a copy of the DLL (BaHartDrv74.dll). The best way is to provide a copy of the 32 bit DLL (x86) as well as a copy of the 64 bit DLL (x64). The files should be copied to the Windows system paths for 32 and 64 bit DLLs.

Note: Be sure that the first call of your application is a call of the validation function of the DLL (BHDrv_ValidateLicense) passing a valid license code and the correct user name to the DLL.

Functions

All functions of the DLL are thread safe. The interface for the functions calls is the same as the WINAPI functions. Thus the DLL may be used by all applications which support calls to the WINAPI functions.

Declaration	Description
Operation	
`void BHDrv_ValidateLicense (const char* pcUserName, const char* pcLicenseCode)`	The first call into the DLL should be a call to this function passing the correct license key and the user name to the software. The user name and the licensee code is provided by the User License Certificate.
`unsigned int BHDrv_OpenChannel (unsigned short usComPort)`	The function allocates the selected com port if possible and starts its own working thread for accessing Hart services. The value which is returned is a handle (hDrv) which has to be passed to all functions which are requesting a service. If it was not possible to open the com port the function is returning INVALD_DRV_HANDLE to indicate the error. The com port number is limited to the range of 1 .. 255.
`void BHDrv_CloseChannel (unsigned int hDrv)`	It is required to call this function at least when the application is terminating.
`void BHDrv_GetConfiguration (unsigned int hDrv, T_strConfiguration* pstrCfg)`	The function copies the configuration data to a data structure provided by the caller.
`void BHDrv_SetConfiguration (unsigned int hDrv, T_strConfiguration* pstrCfg)`	The function is setting all details required for the configuration. The data is passed in a structure provided by the caller.
`void BHDrv_GetRunTimeInfo (unsigned int hDrv, T_strRunTimeInfo * pstrRunTInf)`	Return some information about the communication channel (e.g. if the use of a FIFO at the UART was detected.
`void BHDrv_RegisterEventCallback (unsigned int hDrv, void (__stdcall* HandleServiceEvent) (unsigned int hDrv, unsigned short usEvent, unsigned int hService, unsigned int uiData))`	Register a function which is called when any requested service is completed. The service handle of the service is passed to the called CB function. HandleServiceEvent is the pointer to the handling function which is provided by the user. The parameter usEvent may have the values NONE, CONFIRMATION or BURST_INDICATION. The parameter hDrv is passed to the application to allow the support of more than one communication channel in one callback.
`void BHDrv_ClearEventCallback (unsigned int hDrv)`	Deletes a previously registered callback. After a call of this function no more callbacks to HandleServiceEvent will occur.
Connection Services	
`unsigned int BHDrv_ConnectByAddr (unsigned int hDrv, unsigned char ucAddr, unsigned char ucQOS, unsigned char ucNumRetries)`	Use command 0 with short address to get the connection information. hDrv: The handle which was returned by the OpenChannel function ucAddr: 0 .. 63 ucQOS: DRV_WAIT or DRV_NO_WAIT ucNumRetries: 0 .. 10 The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error.
`unsigned int BHDrv_ConnectByAddr (unsigned int hDrv, unsigned char * pucUniqueID, unsigned char ucQOS, unsigned char ucNumRetries)`	Use command 0 with short address to get the connection information. hDrv: The handle which was returned by the OpenChannel function pucUniqueID: Pointer to a five byte array with the unique identifier ucQOS: DRV_WAIT or DRV_NO_WAIT ucNumRetries: 0 .. 10 The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error.
`unsigned int BHDrv_ConnectByShortTag (unsigned int hDrv, unsigned char * pucTagName, unsigned char ucQOS, unsigned char ucNumRetries)`	Use command 0 with short address to get the connection information. hDrv: The handle which was returned by the OpenChannel function pucTagName: Pointer to the byte array of a length of 6 packed ASCII bytes ucQOS: DRV_WAIT or DRV_NO_WAIT ucNumRetries: 0 .. 10 The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error.

Declaration	Description	
`unsigned int BHDrv_ConnectByLongTag (unsigned int hDrv, unsigned char * pucTagName, unsigned char ucQOS, unsigned char ucNumRetries)`	Use command 0 with short address to get the connection information.	
	hDrv	The handle which was returned by the OpenChannel function
	pucTagName	Pointer to the 32 byte ISO Latin 1 string with the long tag name
	ucQOS	DRV_WAIT or DRV_NO_WAIT
	ucNumRetries	0 .. 10
	The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error.	
`void BHDrv_FetchConnection (unsigned int hSrv, T_strConnection * pstrConnection)`	Fills a structure provided by the caller with the connection information. hSrv is the service handle which was returned by one of the connection functions. Note: After a call of this function the driver is deleting the service. hSrv is no longer valid after calling FetchConnection once.	
Communication Services		
`unsigned char BHDrv_IsSendClear (unsigned int hDrv)`	The function returns B_TRUE, if no more service is pending.	
`unsigned int BHDrv_SendAnyData (unsigned int hDrv, unsigned char * pucData, unsigned char ucLen)`	Send any octet stream via the connected com port.	
	hDrv	The handle which was returned by the OpenChannel function
	pucData	Pointer to a native array of bytes
	ucLen	Number of bytes to be sent
	The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error. The function is provided for debugging purposes allowing to send any stream of data through the serial interface. Note: It is very important to acknowledge this service by calling the function FetchConfirmation after completion. Only with this call the service handle is deleted.	
`unsigned int BHDrv_DoCommand (unsigned int hDrv, unsigned char ucCommand, unsigned char ucQOS, unsigned char * pucReqData, unsigned char ucReqDataLen, unsigned long dwAppKey, unsigned char * pucUniqueID)`	Send a command in the range 0..255.	
	hDrv	The handle which was returned by the OpenChannel function
	ucCommand	Hart command (0..255) to be sent with the request
	ucQOS	DRV_WAIT or DRV_NO_WAIT
	pucReqData	Pointer to a native byte array which is sent as payload data
	ucReqDataLen	Length of the byte array
	dwAppKey	Any value. The value which the user is setting here is returned by the confirmation as is.
	pucUniqueID	Five byte unique identifier of the addressed device
	The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error. Do command can be used for the support of most of the Hart services including all user specific commands. Note: It is not(!) recommended to pass a function pointer through dwAppKey. This will cause problems with 64 bit applications!	
`unsigned int BHDrv_DoExtCmd (unsigned int hDrv, unsigned short ucCommand, unsigned char ucQOS, unsigned char * pucReqData, unsigned char ucReqDataLen, unsigned long dwAppKey, unsigned char * pucUniqueID)`	Send a command in the range 0..65535.	
	hDrv	The handle which was returned by the OpenChannel function
	usCommand	Extended Hart command (0..65535) to be sent with the request
	ucQOS	DRV_WAIT or DRV_NO_WAIT
	pucReqData	Pointer to a native byte array which is sent as payload data
	ucReqDataLen	Length of the byte array
	dwAppKey	Any value. The value which the user is setting here is returned by the confirmation as is.
	pucUniqueID	Five byte unique identifier of the addressed device
	The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error. The extended command in Hart 6/7 is an extension which is using the byte command 31 to carry a larger command within the data area. Therefore this function was introduced more or less for the convenience of the HartDLL user. The function is automatically taking care of the correct usage of command 31. Note: It is not(!) recommended to pass a function pointer through dwAppKey. This will cause problems with 64 bit applications!	

Declaration	Description	
`unsigned int BHDrv_DoBurstCommand (unsigned int hDrv, unsigned char ucCommand, unsigned char ucQOS, unsigned char * pucReqData, unsigned char ucReqDataLen, unsigned long dwAppKey, unsigned char * pucUniqueID)`	Send a burst command (cyclic service) in the range of 0..255.	
	hDrv	The handle which was returned by the OpenChannel function
	ucCommand	Hart command (0..255) to be sent with the request
	ucQOS	DRV_WAIT or DRV_NO_WAIT
	pucReqData	Pointer to a native byte array which is sent as payload data
	ucReqDataLen	Length of the byte array
	dwAppKey	Any value. The value which the user is setting here is returned by the confirmation as is.
	pucUniqueID	Five byte unique identifier of the addressed device
	The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error. To send a burst command may be helpful for device developers or for debugging a network. Note: Even if the burst command is only sent and no response is received, it is very important to acknowledge this service by calling the function FetchConfirmation after completion. Only with this call the service handle is deleted.	
`unsigned char BHDrv_IsServiceCompleted (unsigned int hSrv)`	Returns T_TRUE if the service (hSrv) was completed.	
`void BHDrv_FetchConfirmation (unsigned int hSrv, T_strConfirmation * pstrConfirmation)`	Fills a structure provided by the caller with the service results information such as the response codes and the response data (if any).	
Cyclic Data Services		
`void BHDrv_CycSrvStart (unsigned int hDrv)`	The function is enabling the reception of incoming burst messages. Note: If this function is called eventual existing messages in the drivers queue are deleted, thus the reception of Hart burst messages starts with an empty queue. However, before BHDrv_CycSrcStart is called incoming burst messages are discarded.	
`void BHDrv_CycSrvStop (unsigned int hDrv)`	After the call of this function the reception of burst messages is halted. Messages already in the queue may be read by BHDrv_CycSrvGetData.	
`unsigned char BHDrv_CycSrvGetData (unsigned int hDrv, T_strCyclicData * pstrCyclicData)`	Read cyclic data from the queue in the HartDLL. The returned value indicates if cyclic data was fetched from the queue or not: CYCDAT_OK or CYCDAT_NO_DATA.	
`void BHDrv_CycSrvRegisterCB (unsigned int hDrv, void (__stdcall * pfSubscribeCycData) (unsigned int hDrv, T_strCyclicData * pstrCyclicData))`	For asynchronous reading of cyclic data a callback function may be registered at the DLL. A pointer to a user function is passed, which is called when cyclic data was received. The user function accepts the channel handle and a pointer to a structure containing the received cyclic data.	
`void BHDrv_CycSrvUnregister (unsigned int hDrv)`	After this function was called no more callbacks will be done.	

Declaration	Description
Decoding	

`unsigned char BHDrv_PickInt8`
`(unsigned char ucOffset,`
`unsigned char * pucBytes)`
	Return the value of the byte in the byte array buffer pointed to by pucBytes at the position ucOffset.

`unsigned short BHDrv_PickInt16`
`(unsigned char ucOffset,`
`unsigned char * pucBytes,`
`unsigned char ucEndian)`
	Return the value of the integer 16 from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.

`unsigned long BHDrv_PickInt24`
`(unsigned char ucOffset,`
`unsigned char * pucBytes,`
`unsigned char ucEndian)`
	Return the value of the integer 24 from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.

`unsigned long BHDrv_PickInt32`
`(unsigned char ucOffset,`
`unsigned char * pucBytes,`
`unsigned char ucEndian)`
	Return the value of the integer 32 from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.

`float BHDrv_PickFloat`
`(unsigned char ucOffset,`
`unsigned char * pucBytes,`
`unsigned char ucEndian)`
	Return the value of the single precision IEEE754 number from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.

`double BHDrv_PickDouble`
`(unsigned char ucOffset,`
`unsigned char * pucBytes,`
`unsigned char ucEndian)`
	Return the value of the double precision IEEE754 number from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.

`void BHDrv_PickPackedASCII`
`(unsigned char * pucString,`
`unsigned char ucStringLen,`
`unsigned char ucOffset,`
`unsigned char * pucBytes)`
	Generate a string and copy it to the buffer pointed to by pucString. The final string should have the length ucLen. The packedASCII source is a set of bytes in the byte array buffer pointed to by pucBytes.
	Note: The string length has to by a multiple of 4 while the number of packedASCII bytes is a multiple of 3.

`void BHDrv_PickOctets`
`(unsigned char * pucDstBytes,`
`unsigned char ucNumberOfBytes,`
`unsigned char ucOffset,`
`unsigned char * pucSrcBytes)`
	Copy a number (ucNumberOfBytes) of bytes from the byte array buffer pointed to by pucSrcBytes to the user buffer pointed to by pucDstBytes.

`void BHDrv_PickString`
`(unsigned char * pucString,`
`unsigned char ucStringLen,`
`unsigned char ucOffset,`
`unsigned char * pucSrcBytes)`
	This function does the same as BHDrv_PickOctets.

Declaration	Description
Encoding	
`void BHDrv_PutInt8 (unsigned char ucData, unsigned char ucOffset, unsigned char * pucBytes)`	Insert an integer 8 into the byte array buffer pointed to by pucBytes starting at the position ucOffset.
`void BHDrv_PutInt16 (unsigned short usData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert an integer 16 into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHDrv_PutInt24 (unsigned long ulData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert an integer 24 into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHDrv_PutInt32 (unsigned long ulData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert an integer 32 into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHDrv_PutFloat (float fData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert a single precision IEEE 754 float value into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHDrv_PutDouble (double dData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert a double precision IEEE 754 float value into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHDrv_PutPackedASCII (unsigned char * pucString, unsigned char ucLen, unsigned char ucOffset, unsigned char * pucBytes)`	Insert a string of the length of ucLen in packed ASCII format into the byte array buffer pointed to by pucBytes starting at the position ucOffset.
`void BHDrv_PutPackedASCII (unsigned char * pucString, unsigned char ucLen, unsigned char ucOffset, unsigned char * pucBytes)`	Insert a string of the length of ucLen in packed ASCII format into the byte array buffer pointed to by pucBytes starting at the position ucOffset.
`void BHDrv_PutOctets (unsigned char * pucOctets, unsigned char ucLen, unsigned char ucOffset, unsigned char * pucBytes)`	Copy a number of ucLen bytes into the byte array buffer pointed to by pucBytes starting at the position ucOffset.
`void BHDrv_PutString (unsigned char * pucOctets, unsigned char ucLen, unsigned char ucOffset, unsigned char * pucBytes)`	This function does the same as BHDrv_PutOctets.

Table 3: HartDLL, List of Functions

HartX

The .NET Component HartX is implementing the Hart communication protocol by resolving all the real time requirements and coding as well as decoding issues.

The implementation of the Hart Protocol does not contain any restriction to frame lengths like in Hart 5.x (e.g.). Therefore the all communication functions can be used for devices supporting Hart 5, Hart 6 or Hart 7.

Before using the communication the component has to select a com port of the PC. This can be any com port from 1 to 254 including virtual com ports as they are used for USB modems like the hart modems of Microflex.

Distribution of Applications

The user has to provide a copy of the component DLL and the driver DLL (BaHartX.dll and BaHartDrv75.dll). The best way is to provide a copy of the 32 bit native DLLs (x86) as well as a copy of the 64 bit native DLLs (x64). The files should be copied to the Windows system paths for 32 and 64 bit DLLs.

Note: Be sure that the first call of your application is a call of the validation function of the DLL (HartX.ValidateLicense) passing a valid license code and the correct user name to the component DLL (the assembly).

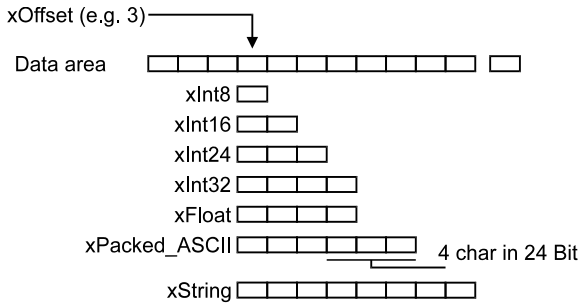
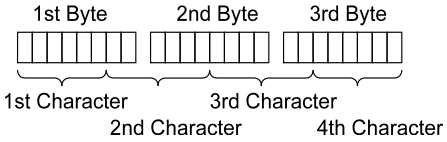
CHartX

Properties

Name	Type	Acc	Description
Operation/Control			
AddrMode	enum	R/W	AM_ShortAddress(0), AM_LongAddress(1) AM_ShortTag(2) -> packed ASCII(6), 8 characters AM_LongTag(3) -> string, 32 characters)
ComPort	byte		0: None 1-254: Com port number (com port in use when set) 255: Reserved, do not(!) use
AddrTagShort	string		Short tag name used for addressing. The string should have a length of 8 and should contain only capital letters.
AddrTagLong	string		Long tag name used for addressing. The string should have a length of 32.
ComState	enum		CS_OFF(0): No connection, CS_ON(1): Connection to device Note: If ComState is toggled from CS_OFF to CS_ON a command for retrieving the unique identifier is executed. This activity is not(!) generating an event.
BaudRate			BR_1200(0), BR_9600(1), BR_19200(2), BR_38400(3), BR_57600(4), BR_115200(5)
NoPreambles	byte		Number of preambles to be sent with a request (typically 5, range 5 .. 20)
PollAddress			Poll address used to get the unique ID (0..63)
NewPollAddress			Poll address to be set in the slave using action ACT_WrPollAddr.
NumRetries			Number of retries in case of error (0..255)
MasterRole	enum		The initial master role when starting communications MR_PrimaryMaster(0), MR_SecondaryMaster(1)
RetryIfBusy			Indicates if the control should retry as long as the device is responding with busy[1]: OPT_No(0), OPT_Yes(1).
LastError		RO	Most recent error: ERR_Success(0), ERR_NoComPortSelected(1), ERR_InvalidComPort(2), ERR_ComError(3), ERR_NoDeviceResponse(4), ERR_SlaveAddressError(5), ERR_UndefinedError(6), ERR_ServiceInvokationError(7), ERR_LicenseError(8)
LastErrorText	string		Text for the LastError value
UseUniqueID	bool	R/W	Indicates if the unique identifier shall be used directly as it was entered by the user.
UniqueID	byte[]		Array of 5 bytes for the unique identifier.
UniqueId0	byte		Long address byte 1
UniqueId1			Long address byte 2
UniqueId2			Long address byte 3
UniqueId3			Long address byte 4
UniqueId4			Long address byte 5
HandleOfChannel	int	RO	Handle of channel which was returned by the HartDLL. This is meant for debugging purposes.
DataLength	byte		Number of data bytes in the confirmation of a service. This can be used for debugging.
Response1			Response code for the command
CommandResponseText	string		Text for the response code 1.
Response2	byte		Device status
DeviceStatusText	string		Text for the response code 2
Information			
IsDeviceConnected	bool	RO	Indicates whether the unique identifier could be read from the device.
IsValidComPort	bool		Indicates whether the selected com port could be opened successfully.
BusyCount	int		Returns the number of currently active aynchronous services. These are services which had been started by DoCommand with the wait flag set to false.

[1] This could cause a very large delay, has to be handled with care.

Name	Type	Acc	Description
Simulation			
SimPvEnabled	bool	R/W	Sets or gets a flag indicating whether the simulation for the four PVs is active.
SimAmplitude	float		The simulation is running Pv values between 0.0 and 1.0. SimAmplitude is the factor to multiply the internal values with.
Parameter Properties			
These properties are used to get portions of data from the recently conducted command.			
Command 0 (Read Unique ID)			
Usually this command is automatically executed if the control is not yet 'connected' to the device (unique identifier unknown).			
p00Device	byte	RO	Device ID (8 bit)
p00DeviceNumber	uint		3 byte unique device ID
p00HardwRev	byte		Hardware revision
p00SoftwRev			Software revision
p00VendorID			Manufacturer/Vendor identifier
Command 1 (Read Primary Variable)			
p01Pv	float	RO	Value of process variable 1
p01PvUnit	byte		Unit code of process variable 1
p01PvUnitString	string		String for the unit of process variable 1
Command 2 (Read Current and Percentage)			
p02Current	float	RO	Value of the current output [mA]
p02Percent			Value of the percentage 0..100 %
Command 3 (Read dynamic Variables)			
p01Pv	float	RO	Value of process variable 1
p01PvUnit	byte		Unit code of process variable 1
p01PvUnitString	string		String for the unit of process variable 1
p02Pv	float		Value of process variable 2
p02PvUnit	byte		Unit code of process variable 2
p02PvUnitString	string		String for the unit of process variable 2
p03Pv	float		Value of process variable 3
p03PvUnit	byte		Unit code of process variable 3
p03PvUnitString	string		String for the unit of process variable 3
p04Pv	float		Value of process variable 4
p04PvUnit	byte		Unit code of process variable 4
p04PvUnitString	string		String for the unit of process variable 4
Command 12 (Read Message)			
p12Message	string	R/W	Hart message, the string should have a length of 32.
Command 13 (Read Tag, Descriptor, Date)			
p13DateDay	byte	R/W	Day of month 1..31
p13DataMonth			Month of the year 1..12
p13DateYear			Year as offset to 1900
p13Descriptor	string		String of 16 characters for the description
p13TagName	string		String of 8 characters for the short tag
Command 14 (Read Transducer Information)			
p14LoSensLimit	float	RO	Lower sensor limit
p14MinSpan			Minimum span
p14SensLimUnit	byte		Unit code for the sensor information (values)
p14SensSerNum	uint		24 bit sensor serial number
p14UpSensLimit	float		Upper sensor limit

Name	Type	Acc	Description
Command 15 (Read Device Information)			
p15AlmSelCode	byte	RO	Alarm selection code
p15LabDistCode			Label distributor code
p15LoRange	float		Lower range value
p15RangeUnit	byte		Unit code for the range values
p15UpRange	float		Upper range value
p15WrProtCode	byte		Write protection 0: None >0: Write protected
p15XferFuncCode			Transfer function code
Command 20 (Read Long Tag Name)			
p20TagNameLong	string	R/W	The long tag name, the string should have a length of 32
X-Properties (Any Command)			
xReqLen	byte	R/W	Defines the length of the request data buffer
xOffset			Defines the offset into the buffer for coding and decoding 
xStringLen			Defines the length of a string for coding and decoding
xPackedASCLen			Defines the length of a packed ascii string
xHexDataDump	string	RO	Returns a string with the hex dump of the buffer with a length of xReqLen
xInt8	byte	R/W	Sets or gets an 8 bit integer value in/from the buffer
xInt16	ushort		Sets or gets an 16 bit integer value in/from the buffer
xInt24	uint		Sets or gets an 24 bit integer value in/from the buffer
xInt32	uint		Sets or gets an 32 bit integer value in/from the buffer
xFloat	float		Sets or gets a float value in/from the buffer
xDouble	double		Sets or gets a double value in/from the buffer
xString	string		Sets or gets a string of xStringLen in/from the buffer
xPacked_ASCII	string		Sets or gets a packed ascii string of xPackedASCLen in/from the buffer. It very important to set the property xPackedASCIILen before accessing the property xPackedASCII. The format PackedASCII stores 4 characters in three octets (24 bits), using only 6 bits for each character. The xPackedASCIILen has to be set to the number of octets used to store the string. Possible values are 3,6,9.. etc.. For instance a xPackedASCIILen of 3 allows to access a string of a length of four characters. 

Methods

Declaration	Description	
`bool Lock()`	The method is trying to lock against the access by other threads. However the method is waiting for approximately 5 seconds. If the lock could not be placed in this time if will return false. Note: Each lock has to be followed by a call of the Unlock method. Otherwise the system may be blocked.	
`void Unlock()`	The method is removing a lock against concurrent access.	
`EN_LastError DoAction (EN_Action Action, bool wait)`	The DoAction method is mainly used to handle the parameter properties.	
	`EN_Action Action`	
	`ACT_None(0)`	Perform no action
	`ACT_RdPv(1)`	Read the primary process variable and the unit (Command 1). Update p01 properties.
	`ACT_RdCurrPerc(2)`	Read the value for the current (4..20 mA) and the pv in % (Command 2). Update p02 properties.
	`ACT_RdAllPv(3)`	Read all available process variables (Command 3). Update p03 properties.
	`ACT_RdMessage(4)`	Read the message (Command 12). Update p12 property.
	`ACT_RdTagDescrDate(5)`	Read Tag, Descriptor and Date (Command 13). Update p13 properties.
	`ACT_RdSensLimits(6)`	Read sensor limit data (Command 14). Update p14 properties.
	`ACT_RdRange(7)`	Read range data (Command 15). Update p15 properties.
	`ACT_WrMessage(8)`	Write message (Command 17). Use the p12 property.
	`ACT_WrTagDescrData(9)`	Write Tag, Descriptor and Date (Command 18). Use p13 properties.
	`ACT_WrPollAddr(10)`	Write a new poll address into the device. Use NewPollAddress for this action.
	`ACT_ResetStatus(11)`	Forces the control to forget the unique identifier of the most recently connected HART device.
`EN_LastError Connect()`	The method is retrieving the unique identifier (long address) from the Hart slave. Note: This method waits for a response and does not generate an event.	
`void Disconnect()`	The method deletes the internally stored unique identifier and discards all outstanding services.	
`EN_LastError DoCommand (byte command, bool wait)`	The method is performing a Hart command in the range 0 .. 255. For the data send with the request it is using xReqLen and the internal data buffer with the data bytes.	
`EN_LastError DoCommand (ushort command, bool wait)`	The method is performing a 16 bit Hart command. For the data send with the request it is using xReqLen and the internal data buffer with the data bytes.	
`void Close()`	Has to be called when the application terminates. Note: This method is simply setting the com port to 0 thus releasing the HartDLL.	
`string GetHartUnit (byte UnitCode)`	Returns the string associated with the 8 bit Hart unit code.	
`void FillBuffer (byte FillValue)`	Initialize all bytes in the internal buffer by the given FillValue.	
`void ValidateLicense (string UserName, string License)`	Call this function firstly after construction to activate all internal functions.	

If the parameter wait is set, the service will be completed if the function returns. Otherwise the event function CommResult will be called after completion.

Functions declared to return EN_LastError will return ERR_Success if the operation was successfully completed.

Events

Declaration	Description	
`void CommResult ( CommResultEventArgs CompletedService)`	The DoAction method is mainly used to handle the parameter properties.	
	`CommResultEventArgs CompletedService`	
	`Command`	Command used for the service
	`IsExtCommand`	True if extended command
	`LastError`	Code of last error
	`LastErrorText`	Text of last error
	`UsedAction`	Action triggered, if 0 no action was triggered.

SlaveDLL

Like the HARTDLL for the master the SlaveDLL is providing rudimentary services for the handling of the Hart protocol by a slave implementation.

However, there are also some differences in the implementation. In the following the term hDrv is missing. It was replaced by the term hChan.

Another issue is the connection. No connection services are provided because the slave does not have to handle any connection oriented details.

Functions

Declaration	Description
Control	
`void BHSlv_ValidateLicense (const char* pcUserName, const char* pcLicenseCode)`	The first call into the DLL should be a call to this function passing the correct license key and the user name to the software. The user name and the licensee code is provided by the User License Certificate.
`T_CHAN_HANDLE BHSlv_OpenChannel (unsigned short usComPort)`	The function allocates the selected com port if possible and starts its own working thread for accessing Hart services. The value which is returned is a handle (hChan) which has to be passed to all functions which are requesting a service. If it was not possible to open the com port the function is returning INVALID_SLV_HANDLE to indicate the error. The com port number is limited to the range of 1 .. 255.
`void BHSlv_CloseChannel (T_CHAN_HANDLE hChan)`	It is required to call this function at least when the application is terminating.
`void BHSlv_GetCommConfig (T_CHAN_HANDLE hChan, T_strSlvCommSettings * pstrCfg)`	The function copies the configuration data to a data structure provided by the caller.
`void BHSlv_SetCommConfig (T_CHAN_HANDLE hChan, T_strSlvCommSettings * pstrCfg)`	The function is setting all details required for the configuration. The data is passed in a structure provided by the caller.
`void BHDrv_RegisterEventCallback (T_CHAN_HANDLE hChan, void (__stdcall* HandleServiceEvent) (unsigned int hChan, unsigned short usEvent, unsigned int hService, unsigned int uiData))`	Register a function which is called when any requested service is completed. The service handle of the service is passed to the called CB function. HandleServiceEvent is the pointer to the handling function which is provided by the user. The parameter usEvent may have the values NONE, REUQEST_RECEIVED or BURST_REQUIRED. The parameter hChan is passed to the application to allow the support of more than one communication channel in one callback.
`BHSlv_SetEventFlags (T_CHAN_HANDLE hChan, T_USHR usEventFlags);`	Set the event flags mask.
`void BHDrv_ClearEventCallback (T_CHAN_HANDLE hChan)`	Deletes a previously registered callback. After a call of this function no more callbacks to HandleServiceEvent will occur.

Declaration	Description	
Operation		
`T_SERV_HANDLE BHSlv_GetRequest (T_CHAN_HANDLE hChan, T_USHR* pusCommand, T_USHR* pusIndInfo, T_UCHR* pucDataLen, T_UCHR* paucData);`	The function is used for polling to get an indication if a master request was received.	
	hChan	The handle which was returned by the OpenChannel function
	pusCommand	Return the command via this pointer.
	pusIndInfo	Get additional info about the request.
	pucDataLen	Returns the number of payload bytes.
	paucData	Returns the payload data.
	The function returns a service handle if successful or INVALID_SLV_HANDLE if there was an error.	
`void BHSlv_PutResponse (T_CHAN_HANDLE hChan, T_SERV_HANDLE hService, T_UCHR ucDataLen, T_UCHR* paucData, T_UCHR ucRspCode1, T_UCHR ucRspCode2);`	Provides all information to build the response for the recently received request.	
	hChan	The handle which was returned by the OpenChannel function
	hService	The handle returned by the GetRequest function.
	ucDataLen	Number of bytes for payload data
	paucData	Byte array for payload data
	ucRspCode1	Response code 1
	ucRspCode2	Response code 2
Decoding		
`unsigned char BHSlv_PickInt8 (unsigned char ucOffset, unsigned char * pucBytes)`	Return the value of the byte in the byte array buffer pointed to by pucBytes at the position ucOffset.	
`unsigned short BHSlv_PickInt16 (unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Return the value of the integer 16 from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.	
`unsigned long BHSlv_PickInt24 (unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Return the value of the integer 24 from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.	
`unsigned long BHSlv_PickInt32 (unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Return the value of the integer 32 from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.	
`float BHSlv_PickFloat (unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Return the value of the single precision IEEE754 number from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.	
`double BHSlv_PickDouble (unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Return the value of the double precision IEEE754 number from the byte array buffer pointed to by pucBytes at the position ucOffset. Assume that the most significant byte is the first if ucEndian is MSB_FIRST(0), which is the Hart standard.	
`void BHSlv_PickPackedASCII (unsigned char * pucString, unsigned char ucStringLen, unsigned char ucOffset, unsigned char * pucBytes)`	Generate a string and copy it to the buffer pointed to by pucString. The final string should have the length ucLen. The packedASCII source is a set of bytes in the byte array buffer pointed to by pucBytes. Note: The string length has to by a multiple of 4 while the number of packedASCII bytes is a multiple of 3.	
`void BHSlv_PickOctets (unsigned char * pucDstBytes, unsigned char ucNumberOfBytes, unsigned char ucOffset, unsigned char * pucSrcBytes)`	Copy a number (ucNumberOfBytes) of bytes from the byte array buffer pointed to by pucSrcBytes to the user buffer pointed to by pucDstBytes.	
`void BHSlv_PickString (unsigned char * pucString, unsigned char ucStringLen, unsigned char ucOffset, unsigned char * pucSrcBytes)`	This function does the same as BHDrv_PickOctets.	

Declaration	Description
Encoding	
`void BHSlv_PutInt8 (unsigned char ucData, unsigned char ucOffset, unsigned char * pucBytes)`	Insert an integer 8 into the byte array buffer pointed to by pucBytes starting at the position ucOffset.
`void BHSlv_PutInt16 (unsigned short usData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert an integer 16 into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHSlv_PutInt24 (unsigned long ulData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert an integer 24 into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHSlv_PutInt32 (unsigned long ulData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert an integer 32 into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHSlv_PutFloat (float fData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert a single precision IEEE 754 float value into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHSlv_PutDouble (double dData, unsigned char ucOffset, unsigned char * pucBytes, unsigned char ucEndian)`	Insert a double precision IEEE 754 float value into the byte array buffer pointed to by pucBytes starting at the position ucOffset. Start with the most significant byte if ucEndian is MSB_FIRST(0), which is the Hart standard.
`void BHSlv_PutPackedASCII (unsigned char * pucString, unsigned char ucLen, unsigned char ucOffset, unsigned char * pucBytes)`	Insert a string of the length of ucLen in packed ASCII format into the byte array buffer pointed to by pucBytes starting at the position ucOffset.
`void BHSlv_PutPackedASCII (unsigned char * pucString, unsigned char ucLen, unsigned char ucOffset, unsigned char * pucBytes)`	Insert a string of the length of ucLen in packed ASCII format into the byte array buffer pointed to by pucBytes starting at the position ucOffset.
`void BHSlv_PutOctets (unsigned char * pucOctets, unsigned char ucLen, unsigned char ucOffset, unsigned char * pucBytes)`	Copy a number of ucLen bytes into the byte array buffer pointed to by pucBytes starting at the position ucOffset.
`void BHSlv_PutString (unsigned char * pucOctets, unsigned char ucLen, unsigned char ucOffset, unsigned char * pucBytes)`	This function does the same as BHDrv_PutOctets.

Table 4: SlaveDLL, List of Functions

SlaveX

SlaveX is providing a small set of objects used to build a command interpreter easily and quickly.

A Hart slave is basically implementing a command interpreter for the Hart protocol. This is based on the use of the Hart communication services provided in the object HartSlave.

CSlaveX

Properties

Name	Type	Acc	Description
IsValidChannel	bool	RO	Returns true if there is a valid com port adressed by the channel.
ComPort	byte		Returns the comport number.
Status	EN_Status		Returns the status.

			`EN_Status : int`
			`{`
			`IDLE = 0,`
			`READY = 1,`
			`WAIT_RESPONSE = 2,`
			`DISABLED = 3,`
			`UNKNOWN = -1`
			`}`

PrintCallback	IntPtr	WO	Sets the pointer to a print callback function.
DataBase	CDataBase	RO	Returns a reference to the database of the component.

Methods

Declaration	Description
`void Start(int comPort)`	Starts the simulation at a defined com port.
`void Configure()`	Sets up internal data of the component using the static class CDataBase.
`void Enable()`	Enables the component.
`void Disable()`	Disables the component.
`CRequest GetRequest()`	Returns an instance of the CRequest class if a request was detected by the communication layers.
`void PutResponse(CResponse reponse, byte devstatus)`	Accepts the response to be sent and the HART device status.
`void Print(byte row, string text)`	Print a text on the debug output of the client if any is provided.

CRequest

The object is passed to the command interpreter when a Hart command was received by the communication DLL.

Properties

Name	Type	Acc	Description
Command	ushort	RO	The command that was passed with the request.
Len	byte		Number of bytes of productive data.
Data	byte[]		Returns an array of bytes with the payload data of the request.
Flags	ushort		Returns a bit stream which is not yet defined.

Methods

Declaration	Description
`byte GetByte(byte offset)`	Returns the value of a 8 bit unsigned integer at the position (offset) in the data of the request.
`ushort GetInt16(byte offset)`	Returns the value of a 16 bit unsigned integer at the position (offset) in the data of the request.
`ulong GetInt24(byte offset)`	Returns the value of a 24 bit unsigned integer at the position (offset) in the data of the request.
`float GetFloat(byte offset)`	Returns the value of a 32 bit float as IEEE754 at the position (offset) in the data of the request.
`string GetPackedASCII(byte offset, byte len)`	Returns the decoded string from a PackedASCII string at the position (offset) in the data of the request. len is the number of bytes of the PackedASCII coded string. Note: len has to be an integer multiple of 3, while the length of the resulting string is a multiple of 4.
`string GetString(byte offset, byte len)`	Returns the string with length (len) at the position (offset) in the data of the request.

CResponse

Properties

Name	Type	Acc	Description
`CmdResultCode`	`byte`	`R/W`	Gets or sets the cmd reponse code.
`DeviceStatus`	`byte`		Gets or sets the Hart device status.
`DataLength`	`byte`	`RO`	Gets the number of bytes of payload data in the response.
`Data`	`byte[]`		Gets an array of bytes with the payload data for the response.

Methods

Declaration	Description
`void SetByte(byte offset, byte value)`	Sets the value of an 8 bit unsigned integer at the position (offset) in the data of the response.
`void SetInt16(byte offset, ushort value)`	Sets the value of a 16 bit unsigned integer at the position (offset) in the data of the response.
`void SetInt24(byte offset, uint value)`	Sets the value of a 24 bit unsigned integer at the position (offset) in the data of the response.
`void SetInt32(byte offset, uint value)`	Sets the value of a 32 bit unsigned integer at the position (offset) in the data of the response.
`void SetFloat(byte offset, float value)`	Sets the value of a 32 bit float at the position (offset) in the data of the response.
`void SetPackedASCII(byte offset, string value, byte len)`	Convert the string (value) into PackedASCII-format and insert the resulting bytes at the position (offset) in the data of the response. len is the number of PackedASCII bytes to be inserted. It should be an integer multiple of 3. If this is not the case it is reduced to the next lower integer multiple of 3. The length of the string (val) should be an integer multiple of 4 following the formula: value.length = len / 3 * 4 if value.length is shorter than the required length the string is filled by ' '. If it is longer the string is truncated. Example: The Hart short tag name has to have 8 characters. Therefore len has to be 6.
`void SetString(byte offset, string value, byte len)`	Insert the bytes of a ISO Latin-1 string (val) with the length len at the position (offset) in the data of the response. If the string is shorter than len it is filled by char(0). If the string is longer than len it is truncated.

Additional Information

Hart at a Glance

Frame Coding

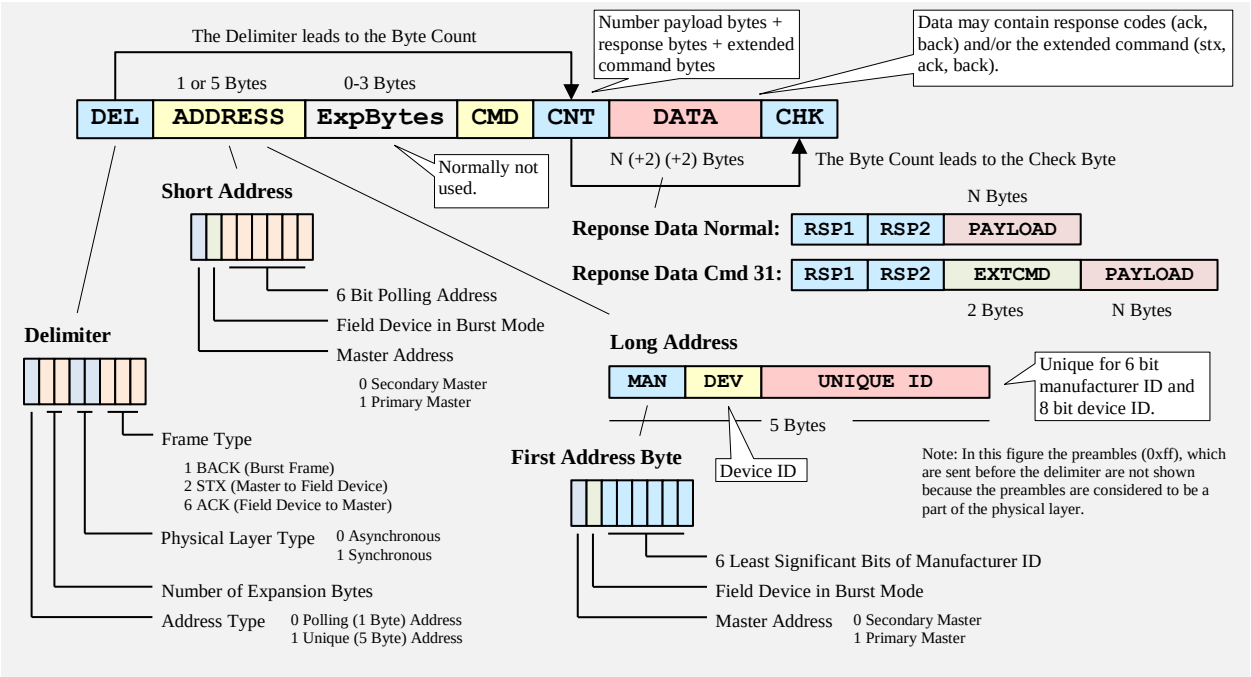


Figure 4: The Basic Coding of a Hart Frame

The figure above is giving an overview of the coding of a Hart frame. Usually Hart services are composed of a request (stx) by the master followed the response (ack) of a slave. Bursts (back) are frames looking like a response (including response codes) but sent by the slave without any request. The slave is sending these frames in burst mode within defined time slots following the rules of the protocol specification. In fact Hart is a token passing protocol which allows also the slave to be a token holder and send burst frames.

The following chapter is showing a list of Hart commands which are used very often. The list is showing the major differences between Hart 5.3, Hart 6 and Hart 7.4.

New items in Hart 6 are marked with yellow color while new items of Hart 7.4 are marked by blue color.

However, the following is not replacing any specification and is not showing the details which are needed for an implementation. The details has to be taken from the Hart specifications which are provided by the Hart Communication Foundation (http://de.hartcomm.org/).
That the listed commands are most commonly used is not the opinion of the HCF but the opinion of the author of this document.

Commonly Used Commands

No	Title	Request Data			Response Data		
Universal							
00	Read Unique Identifier	None			0	int8	254
					1		Manufacturer ID
					2		Short device ID
					3		Number preambles request
					4		Hart revision
					5		Device revision
					6		Software revision
					7		Hw rev and signaling code
					8		Flags
					9	int24	DevUniqueID
					12	int8	Number preambles response
					13		Maximum number device variables
					14	int16	Configuration change counter
					16	int8	Extended device status
					17	int16	Extended manufacturer code
					19		Extended label distributor code
					21	int8	Device profile
01	Read Primary Variable	None			0	int8	PV Units
					1	float	Primary variable
02	Read Current and Percent of Range	None			0	float	Current
					1	float	Percent of range
03	Read Current and Dyn. Variables	None			0	float	Current
					4	int8	PV1 units code
					5	float	PV1 value
					9	int8	PV2 units code
					10	float	PV2 value
					14	int8	PV3 units code
					15	float	PV3 value
					19	int8	PV4 units code
					20	float	PV4 value
06	Write Polling Address	0	int8	Polling Address	0	int8	PV Units
		1	int8	Loop current mode	1	int8	Loop current mode
07	Read Loop Configuration	None			0	int8	Polling address
					1		Loop current mode
08	Read Dyn. Vars Classification	None			0	int8	PV1 classification
					1		PV2 classification
					2		PV3 classification
					3		PV4 classification

No	Title	Request Data			Response Data		
Universal							
09	**Read Device Variables with Status**	0	int8	Slot0: Device variable code	0	int8	Extended device status
		1		Slot1: Device variable code	1	Slot0: Device variable properties	
		2		Slot2: Device variable code	1	int8	Device variable code
		3		Slot3: Device variable code	2		Device variable classification
		4	int8	Slot4: Device variable code	3		Device variable units code
		5		Slot5: Device variable code	4	float	Device variable value
		6		Slot6: Device variable code	8	int8	Device variable status
		7		Slot7: Device variable code	9	struct	Slot1: Device variable properties
					17		Slot2: Device variable properties
					25		Slot3: Device variable properties
					33	struct	Slot4: Device variable properties
					41		Slot5: Device variable properties
					49		Slot6: Device variable properties
					57		Slot7: Device variable properties
					65	time	Time stamp slot0
11	**Read Unique ID by Short Tag**	0	pac6	Tag name (packed ascii) 6 bytes = 8 characters	Same as command 0 read unique identifier		
12	**Read Message**	None			0	pac24	Message (packed ascii) 24 bytes = 32 characters
13	**Read Tag, Descriptor, Date**	None			0	pac6	Short tag (packed ascii) 6 bytes = 8 characters
					6	pac12	Descriptor (packed ascii) 12 bytes = 16 characters
					18	int8	Day
					19		Month
					20		Year (offset to 1900)
14	**Read Primary Variable Transducer Information**	None			0	int24	Transducer serial number
					3	int8	Units code
					4	float	Upper transducer limit
					8		Lower transducer limit
					12		Minimum span
15	**Read Device Information**	None			0	int8	Alarm selection code
					1		Transfer function code
					2		Units code
					3	float	PV upper range value (for 20 mA)
					7		PV lower range value (for 4 mA)
					11		PV damping value
					15	int8	Write protect code
					16		Reserved, must be set to 250
					17		PV analog channel flags
16	**Read Ass. Num**	None			0	int24	Final assembly number
17	**Write Message**	Same as response command 12			Same as response command 12		
18	**Write Tag, Descriptor, Date**	Same as response command 13			Same as response command 13		
19	**Write Ass. Num**	Same as response command 16			Same as response command 16		
20	**Read Long Tag**	None			0	str32	Long tag: 32 ISO Latin-1 characters
21	**Read Unique ID by Long Tag**	0	str32	Long tag: 32 ISO Latin-1 characters	Same as command 0 read unique identifier		
22	**Write Long Tag**	Same as response command 20			Same as response command 20		

No	Title	Request Data			Response Data		
Universal / Common Practice							
38	Reset Config Changed Flag	None			None		
		0	int16	Configuration change counter	0	int16	Configuration change counter
48	Read Additional Device Status	None					
		0	int8[5]	Transmitter specific status	0	int8[5]	Transmitter specific status
					6	int8[2]	Operating mode
		6	int8	Extended device status	6	int8	Extended device status
		7		Device operating mode	7		Device operating mode
					8	int8[3]	Analog output status
		8	int8	Standard status 0	8	int8	Standard status 0
		9		Standard status 1	9		Standard status 1
		10		Analog channel saturated	10		Analog channel saturated
					11	int8[3]	Analog output fixed
		11	int8	Standard status 2	11	int8	Standard status 2
		12		Standard status 3	12		Standard status 3
		13		Analog channel fixed	13		Analog channel fixed
					14	int8[3]	Transmitter specific status
		14	int8[10]	Transmitter specific status	14	int8[10]	Transmitter specific status
Common Practice							
33	Read Device Variables	0	int8	Slot0: Device variable code	0	Slot0: Device variable properties	
		1		Slot1: Device variable code	0	int8	Device variable code
		2		Slot2: Device variable code	1		Device variable units code
		3		Slot3: Device variable code	2	float	Device variable value
					6	struct	Slot1: Device variable properties
					12		Slot2: Device variable properties
					18		Slot3: Device variable properties
34	Write Prim. Var. Damping	0	float	PV 1 damping value	0	float	PV 1 damping value
35	Write Prim. Var. Range Values	0	int8	Units code	0	int8	Units code
		1	float	Upper range value	1	float	Upper range value
		5		Lower range value	5		Lower range value
36	Set Prim. Var. Upper Range	None			None		
37	Set Prim. Var. Lower Range	None			None		
40	Enter/Exit Fixed Current	0	float	Current value	0	float	Actual current value
42	Device Reset	None			None		
43	Set Primary Variable Zero	None			None		
44	Write Prim. Var. Units	0	int8	PV 1 units code	0	int8	PV 1 units code
45	Trim Prim. Var. Current Zero	0	float	Measured current value	0	float	Actual current value
46	Trim Prim. Var. Current Gain	0	float	Measured current value	0	float	Actual current value
50	Read Dynamic Variable Assignments	None			0	int8	PV 1 variable code
					1		PV 2 variable code
					2		PV 3 variable code
					3		PV 4 variable code

No	Title	Request Data			Response Data		
Common Practice							
51	Write Dynamic Variable Assignments	0	int8	PV 1 variable code	0	int8	PV 1 variable code
		1		PV 2 variable code	1		PV 2 variable code
		2		PV 3 variable code	2		PV 3 variable code
		3		PV 4 variable code	3		PV 4 variable code
54	Read Device Variable Information	0	int8	Device variable code	0	int8	Device variable code
					1	int24	Sensor serial number
					4	int8	Units code
					5	float	Variable upper limit
					9		Variable lower limit
					13		Variable damping
					17		Variable minimum span
					21	int8	Variable classification
					22		Variable family
					23	time	Acquisition period
					27	bin8	Variable properties
71	Lock Device	0	int8	Lock code	0	int8	Lock code
76	Read Lock State	None			0	int8	Lock status
78	Read Aggregated Commands	0	int8	Number of commands requested	0	int8	Extended device status
		1	str[]	Array of command requests struct { int16 command int8 byteCount int8[] requestData }	1	int8	Number of commands requested
					2	str[]	Array of command responses struct { int16 command int8 byteCount int8 responseCode int8[] responseData }
79[2]	Write Device Variable	0	int8	Device Variable Code	0	int8	Device Variable Code
		1		DV command code	1		DV command code
		2		DV units code	2		DV units code
		3	float	DV value	3	float	DV value
		7	int8	DV status	7	int8	DV status
103	Write Burst Period	0	int8	Burst message	0	int8	Burst message
		1	time	Update period	1	time	Update period
		5		Maximum update period	5		Maximum update period
104	Write Burst Trigger	0	int8	Burst message	0	int8	Burst message
		1		Trigger mode selection code	1		Trigger mode selection code
		2		Device variable classification for trigger level	2		Device variable classification for trigger level
		3		Units code	3		Units code
		4	float	Trigger level	4	float	Trigger level

[2] Used to simulate the value of a device variable

No	Title	Request Data			Response Data		
Common Practice							
105	**Read Burst Mode Configuration**	None			0	int8	Burst mode control code
					1	int8	Burst command number
					2	int8	Burst command slot 0
					3	int8	Burst command slot 1
					4	int8	Burst command slot 2
					5	int8	Burst command slot 3
		0	int8	Burst message	0	int8	Burst mode control code
					1		0x1f (31) command expansion
					2		DV code slot0
					3		DV code slot1
					4		DV code slot2
					5		DV code slot3
					6		DV code slot4
					7		DV code slot5
					8		DV code slot6
					9		DV code slot7
					10		Burst message
					11		Maximum number of burst messages
					12	int16	Extended command number
					14	time	Update time
					18		Maximum update time
					22	int8	Burst trigger mode code
					23		DV classification for trigger value
					24		Units code
					25	float	trigger value
106	**Flush Delayed Responses**	None			None		
107	**Write Burst Device Variables**	0	int8	DV code slot 0	0	int8	DV code slot 0
		1		DV code slot 1	1		DV code slot 1
		2		DV code slot 2	2		DV code slot 2
		3		DV code slot 3	3		DV code slot 3
		4	int8	DV code slot 4	4	int8	DV code slot 4
		5		DV code slot 5	5		DV code slot 5
		6		DV code slot 6	6		DV code slot 6
		7		DV code slot 7	7		DV code slot 7
		8		Burst message	8		Burst message
108	**Write Burst Mode Command**	0	int8	Command number for the burst response	0	int8	Command number of the burst response
109	**Burst Mode Control**	0	int8	Burst mode control code	0	int8	Burst mode control code
113	**Catch Device Variable**	0	int8	Destination DV code	0	int8	Destination DV code
		1		Capture mode code	1		Capture mode code
		2		Source slave manufacturer ID	2	int8[5]	Source slave address
		3		Source slave device type			
		2	int16	Source slave expanded device type			
		4	int8[3]	Source slave device ID			
		7	int8	Source command number	7	int8	Source command number
		8		Source slot number	8		Source slot number
		9	float	Shed time for this mapping	9	float	Shed time for this mapping
		7	int8	0x1f (31) command expansion	7	int8	0x1f (31) command expansion
		8		Source slot number	8		Source slot number
		9	float	Shed time for this mapping	9	float	Shed time for this mapping
		13	int16	Ext source command number	13	int16	Ext source command number

No	Title	Request Data			Response Data		
Common Practice							
114	**Read Caught Device Variable**	0	int8	Destination DV code	0	int8	Destination DV code
					1		Capture mode code
					2	int8[5]	Source slave address
					7	int8	Source command number
					8		Source slot number
					9	float	Shed time for this mapping
					7	int8	0x1f (31) command expansion
					8		Source slot number
					9	float	Shed time for this mapping
					13	int16	Ext source command number
523	**Read Condensed Status Mapping Array**	0	int8	Starting index status map	0	int8	Actual starting index
		1		Number of entries to read	1		Number of entries returned
					2	int4[]	Status map codes array
524	**Write Condensed Status Mapping Array**	0	int8	Starting index status map	0	int8	Actual starting index
		1		Number of entries to write	1		Number of entries returned
		2	int4[]	Status map codes array	2	int4[]	Status map codes array
525	**Reset Condensed Status Map**	None			None		
526	**Write Status Simulation Mode**	0	int8	Status simulation mode	0	int8	Status simulation mode
527	**Simulate Status Bit**	0	int8	Status bit index	0	int8	Status bit index
		1		Status bit value	1		Status bit value

Response Codes

As response code 1 is command specific it is documented together with the command specifications. However response code 2 is of general nature and contains 8 bit flags with the following meaning.

Flag Number / Meaning	Description
Bit #7 Field Device Malfunction	The device has detected a hardware error or failure. Further information may be available through the Read Additional Transmitter Status Command, #48.
Bit #6 Configuration Changed	A write or set command has been executed.
Bit #5 Cold Start	Power has been removed and reapplied resulting in the reinstallations of the setup information. The first command to recognize this condition will automatically reset this flag. This flag may also be set following a Master Reset or a Self Test.
Bit #4 More Status Available	More status information is available than can be returned in the Field Device Status. Command #48, Read Additional Status Information, will provide this additional status information.
Bit #3 Primary Variable Analog Output Fixed	The analog and digital analog outputs for the Primary Variable are held at the requested value. They will not respond to the applied process.
Bit #2 Primary Variable Analog Output Saturated	The analog and digital analog outputs for the Primary Variable are beyond their limits and no longer represent the true applied process.
Bit #1 Non Primary Variable Out of Limits	The process applied to a sensor, other than that of the Primary Variable, is beyond the operating limits of the device. The Read Additional Transmitter Status Command, #48, may be required to identify the variable.
Bit #0 Primary Variable Out of Limits	The process applied to the sensor for the Primary Variable is beyond the operating limits of the device.

Structures

Type	Name	Description
T_strConfiguration		
unsigned int	uiBaudRate	Baudrate as defined in winbase.h CBR_1200 CBR_9600 CBR_19200 CBR_38400 Default: CBR_1200
unsigned char	ucNumPreambles	Number of preambles used for a request (0..22) Default: 5
unsigned char	ucNumRetries	Number of retries if device response is erroneous (0..3) Default: 2
unsigned char	ucRetryIfBusy	0: Do not retry if device is responding with busy code 1..255: Retry the command if device is responding with busy code. The number of retries is reflected in the confirmation as ucUsedRetries. Default: 1
unsigned char	ucInitialMasterRole	0: Primary master 1: Secondary master Default: 0
unsigned char	ucReserved	Not used (former addressing mode)
unsigned char	ucDoNotUseRtsDtr	0: Use handshake signals 1: Do not use handshake signals Default: 0
unsigned short	usAddTimeOut	Additional time out to wait for a slave response in ms. Typical 100, 200 etc. Default: 0
unsigned short	usAddGapTime	Additional time for gap between characters in ms. Typical 5, 10 etc. Default: 0
unsigned short	usAddRtsOffDelay	Additional delay before Rts is switched off (carrier off) in ms. Typical 1, 2, 5, 10 etc. Default: 0
unsigned char	bSendJabberOctet	0: Normal sending 1: Append ucJabberOctet to each frame Default: 0
unsigned char	ucJabberOctet	Value of the jabber octet
unsigned char	bGenParityError	Generate a parity error on a particular position
unsigned char	ucParityErrorPos	Number of the byte at which the error should be injected
unsigned char	bHartEnabled	0: Hart not running 1: Hart protocol active
unsigned char	bRecJabberOctet	0: Ignore jabber octets 1: Report jabber octets to the monitor
T_strRunTimeInfo		
unsigned char	bActualMaster	0: Primary Master 1: Secondary Master
unsigned char	bFifoDetected	>0: More than 3 characters are received at once
unsigned char	ucBlockSize	Number of characters received at once
unsigned char	ucReserved	

Type	Name	Description	
T_strConnection			
unsigned char	ucManId	Manufacturer id as defined by the Hart Communication Foundation	
unsigned char	ucDevId	Vendor's device id	
unsigned char	ucNumPreambs	Number of preambles defined by the device	
unsigned char	ucCmdRevNum	Command set revision number as defined by Hart	
unsigned char	ucSpecRevCode	Device specific revision code	
unsigned char	ucSwRev	Software revision code (0..255)	
unsigned char	ucHwRev	Hardware revision code	
unsigned char	ucHartFlags	The flags as defined by Hart	
unsigned char	ucError	Service completion code	
		SRV_EMPTY(0)	Not active
		SRV_NO_DEV_RESP(1)	No device response
		SRV_COMM_ERR(2)	There was some error (too few data e.g.)
		SRV_INVALID_HANDLE(3)	Service handle is invalid
		SRV_IN_PROGRESS(4)	Service working
		SRV_SUCCESSFUL(5)	Service successfully completed
		SRV_RESOURCE_ERROR(6)	Out of memory
		SRV_TOO_FEW_DATA_BYTES(7)	Used for cmd 31
unsigned char	ucRespCode1	Response code 1 as defined by the Hart specification	
unsigned char	ucRespCode2	Response code 2 as defined by the Hart specification	
unsigned char	ucUsedRetries	Number of retries which were used for completion	
unsigned char	bDeviceInBurstMode	0: Normal mode 1: Device is in burst mode	
unsigned char	ucExtDevStatus	Extended device status	
unsigned short	usCfgChCount	Configuration changed counter	
unsigned char	ucMinNumPreambs	Minimum number of preambles	
unsigned char	ucMaxNumDVs	Maximum number of device variables	
unsigned short	usManuID	Extended manufacturer ID	
unsigned short	usLabDistID	Extended label distributor ID	
unsigned char	ucDevProfile	Device profile	
unsigned char	ucReserved	-/-	
unsigned char	aucUniqueID[5]	Unique identifier	
T_strCyclicData			
unsigned long	ulTimeStamp	Time in ms since recording of burst messages was started	
unsigned char	ucCmd	Command of the received frame	
unsigned char	ucRsp1	Device response code 1	
unsigned char	ucRsp2	Device response code 2	
unsigned char	ucDataLen	Number of bytes in productive data	
unsigned char	aucData[255]	Productive data of the burst message	

Type	Name	Description	
T_strConfirmation			
unsigned char	ucCmd	Command which was executed	
unsigned char	ucRespCode1	Response code 1 as defined by the Hart specification	
unsigned char	ucRespCode2	Response code 2 as defined by the Hart specification	
unsigned char	ucError	Service completion code	
		SRV_EMPTY(0)	Not active
		SRV_NO_DEV_RESP(1)	No device response
		SRV_COMM_ERR(2)	There was some error (too few data e.g.)
		SRV_INVALID_HANDLE(3)	Service handle is invalid
		SRV_IN_PROGRESS(4)	Service working
		SRV_SUCCESSFUL(5)	Service successfully completed
		SRV_RESOURCE_ERROR(6)	Out of memory
		SRV_TOO_FEW_DATA_BYTES(7)	Used for cmd 31
unsigned char	ucUsedRetries	Number of retries which were used for completion	
unsigned char	bDeviceInBurstMode	0: Normal mode 1: Device is in burst mode	
unsigned short	usDuration	Time for service execution in ms	
unsigned long	dwAppKey	Is returned by the FetchConfirmation function as it was passed to the DoCommand function.	
unsigned short	usExtCmd	Extended cmd number	
unsigned char	ucReserved	Reserved for future use	
unsigned char	ucLen	Number of response data bytes (octets)	
unsigned char	aucData [DATA_BUF_LEN]	Response data bytes (DATA_BUF_LEN = 255)	
T_strSlaveDynamicValues			
float	fPercent	Actual percent of range	
float	fCurrent	Actual current value as ma	
unsigned char	ucUnitCodePV1	Hart unit code for PV1	
unsigned char	ucUnitCodePV2	Hart unit code for PV2	
unsigned char	ucUnitCodePV3	Hart unit code for PV3	
unsigned char	ucUnitCodePV4	Hart unit code for PV4	
float	fPV1	Value of PV1	
float	fPV2	Value of PV2	
float	fPV3	Value of PV3	
float	fPV4	Value of PV4	
unsigned char	bDeviceMalfunction	Signals device mal function	
unsigned char	bCfgChangedPrimMaster	Configuration change flag for primary master	
unsigned char	bCfgChangedScndMaster	Configuration change flag for primary master	
unsigned char	bColdStartPrimMaster	Cold start flag for primary master	
unsigned char	bColdStartScndMaster	Cold start flag for secondary master	
unsigned char	bMoreStatusAvail	Flags more status available (see command 48)	
unsigned char	bLoopCurrentFixed	Signals fixed current mode active	
unsigned char	bLoopCurrentSaturated	Signals current output saturated	
unsigned char	bNonPrimVarOutLimits	Signals none primary variable out of limits	
unsigned char	bPrimVarOutLimits	Signals primary variable out of limits	
unsigned char	bUseExtValues	Indication to the slave simulation to use the values of this structure instead of its own.	
unsigned char	ucReserved1	Reserved for future use	

Type	Name	Description
T_strSlaveConfiguration		
unsigned char	ucManufacturerID	Manufacturer's identifier
unsigned char	ucDeviceID	Device identifier
unsigned char	ucNumPreambles	Number of preambles needed in a request (2..20, recommended: 2)
unsigned char	ucCmdSetRevision	Hart compatibility version (5..7, recommended: 5)
unsigned char	ucTransmSpecRev	Transmitter specific revision
unsigned char	ucSoftwareRevision	Software revision number
unsigned char	ucHardwareRevision	Hardware revision number
unsigned char	ucReserved1	Reserved for future use
unsigned char	ucDevNum1	Device number [LSB]
unsigned char	ucDevNum2	Device number [LSB+1]
unsigned char	ucDevNum3	Device number [LSB+2]
unsigned char	ucReserved2	Reserved for future use
unsigned char	aucShortTag[12]	Tag name, 8 characters (see 3.3.2.1 Packed ASCII Coding for possible characters)
unsigned char	aucLongTag[36]	Long tag name, 32 characters iso latin 1
unsigned char	ucPollAddress	Slave polling address
unsigned char	ucNumberOfPVs	Defines the number of variables to be sent with command 3
unsigned char	ucReserved3	Reserved for future use
unsigned char	ucReserved4	Reserved for future use
unsigned char	aucMessage[36]	Message, 32 characters coded in packed ASCII
unsigned char	aucDescription[20]	Description, 16 characters coded in packed ASCII
unsigned char	ucDay	Day of Hart date (1..31)
unsigned char	ucMonth	Month of Hart date (1..12)
unsigned short	usYear	Year of Hart date (1900..2155)

Constants

Name	Value	Description
Service Completion Codes		
SRV_EMPTY	0x00	Service not active
SRV_NO_DEV_RESP	0x01	Device did not respond
SRV_COMM_ERR	0x02	There was a communication error (too few data e.g.)
SRV_INVALID_HANDLE	0x03	Service handle not valid
SRV_IN_PROGRESS	0x04	Service not yet completed
SRV_SUCCESSFUL	0x05	Service successfully completed
SRV_RESOURCE_ERROR	0x06	Out of memory
SRV_TOO_FEW_DATA_BYTES	0x07	Used with cmd 31
Values of Handles		
INVALID_DRV_HANDLE	-1	Driver handle not valid
INVALID_SRV_HANDLE	-1	Service handle not valid
Endian		
MSB_FIRST	0x00	Big Endian (Hart standard): Most Significant Byte first
LSB_FIRST	0x01	Little Endian: Least Significant Byte first
Wait Options		
DRV_NO_WAIT	0x00	User will poll for the completion of service
DRV_WAIT	0x01	The function returns if service is completed
Slave Modes		
SLAVE_DISABLED	0x00	Slave emulation is not active
SLAVE_ENABLED	0x01	Slave emulation is active
Cyclic Data Handling		
CYCDAT_OK	0x00	Cyclic data available
CYCDAT_NO_DATA	0x01	Cyclic data not (yet) available
Boolean Values		
T_FALSE	0x00	True
T_TRUE	0x01	False

Data Types

Float IEEE 754

The following summarizes the IEEE 754 and recommends that standards are referred to for implementation.

The floating point values passed by the protocol are based on the IEEE 754 single precision floating point standard.

```
Data Byte   #0          #1          #2          #3

            SEEEEEEE    EMMMMMMM    MMMMMMMM    MMMMMMMM

S - Sign of the mantissa; 1 = negative
E - Exponent; Biased by 127 decimal in two's complement format
M - Mantissa; 23 least significant bits, fractional portion
```

The value of the floating point number described above is obtained by multiplying 2, raised to the power of the unbiased exponent, by the 24-bit mantissa. The 24-bit mantissa is composed of an assumed most significant bit of 1, a decimal point following the 1, and the 23 bits of the mantissa.

$$S1.M \cdot 2^{(E-127)}$$

The floating point parameters not used by a device will be filled with 7F A0 00 00: Not-a-Number.

Double IEEE 754

The following summarizes the IEEE 754 and recommends that standards are referred to for implementation.

The floating point values passed by the protocol are based on the IEEE 754 single precision floating point standard.

```
Data Byte   #0          #1          #2          #3

            SEEEEEEE    EEEEMMMM    MMMMMMMM    MMMMMMMM

Data Byte   #4          #5          #6          #7

            MMMMMMMM    MMMMMMMM    MMMMMMMM    MMMMMMMM

S - Sign of the mantissa; 1 = negative
E - Exponent; Biased by 1023 decimal in two's complement
format
M - Mantissa; 52 least significant bits, fractional portion
```

The value of the floating point number described above is obtained by multiplying 2, raised to the power of the unbiased exponent, by the 53-bit mantissa. The 53-bit mantissa is composed of an assumed most significant bit of 1, a decimal point following the 1, and the 52 bits of the mantissa.

$$S1.M \cdot 2^{(E-1023)}$$

Packed ASCII

The packed ASCII Format uses 6 Bit to encode a character. Therefore 4 characters in the original string require 3 octets in the resulting data. It is recommended to provide strings always as a multiple ordinal of 4 characters

Construction of Packed-ASCII characters:

a) Truncate Bit #6 and #7 of each ASCII character.
b) Pack four, 6 bit-ASCII characters into three bytes.

Reconstruction of ASCII characters:

a) Unpack the four, 6-bit ASCII characters.
b) Place the complement of Bit #5 of each unpacked, 6-bit ASCII character into Bit #6.
c) Set Bit #7 of each of the unpacked ASCII characters to zero.
d) The Packed ASCII code (hexadecimal) allows the representation of the following characters.

CHAR	CODE	CHAR	CODE	CHAR	CODE	CHAR	CODE
@	00	P	10	Space	20	0	30
A	01	Q	11	!	21	1	31
B	02	R	12	"	22	2	32
C	03	S	13	#	23	3	33
D	04	T	14	$	24	4	34
E	05	U	15	%	25	5	35
F	06	V	16	&	26	6	36
G	07	W	17	'	27	7	37
H	08	X	18	(	28	8	38
I	09	Y	19	)	29	9	39
J	0A	Z	1A	*	2A	:	3A
K	0B	[	1B	+	2B	;	3B
L	0C	\	1C	,	2C	<	3C
M	0D	]	1D	-	2D	=	3D
N	0E	^	1E	.	2E	>	3E
O	0F	_	1F	/	2F	?	3F

e) Note: The implementation of the function is assuming that the packed ascii string should be an ordinal multiple of 3. If the length of the passed string is not an ordinal multiple of 4 the missing packed ascii characters are replaced by spaces.

Getting Started

HartDLL

Service Processing Flow Diagram

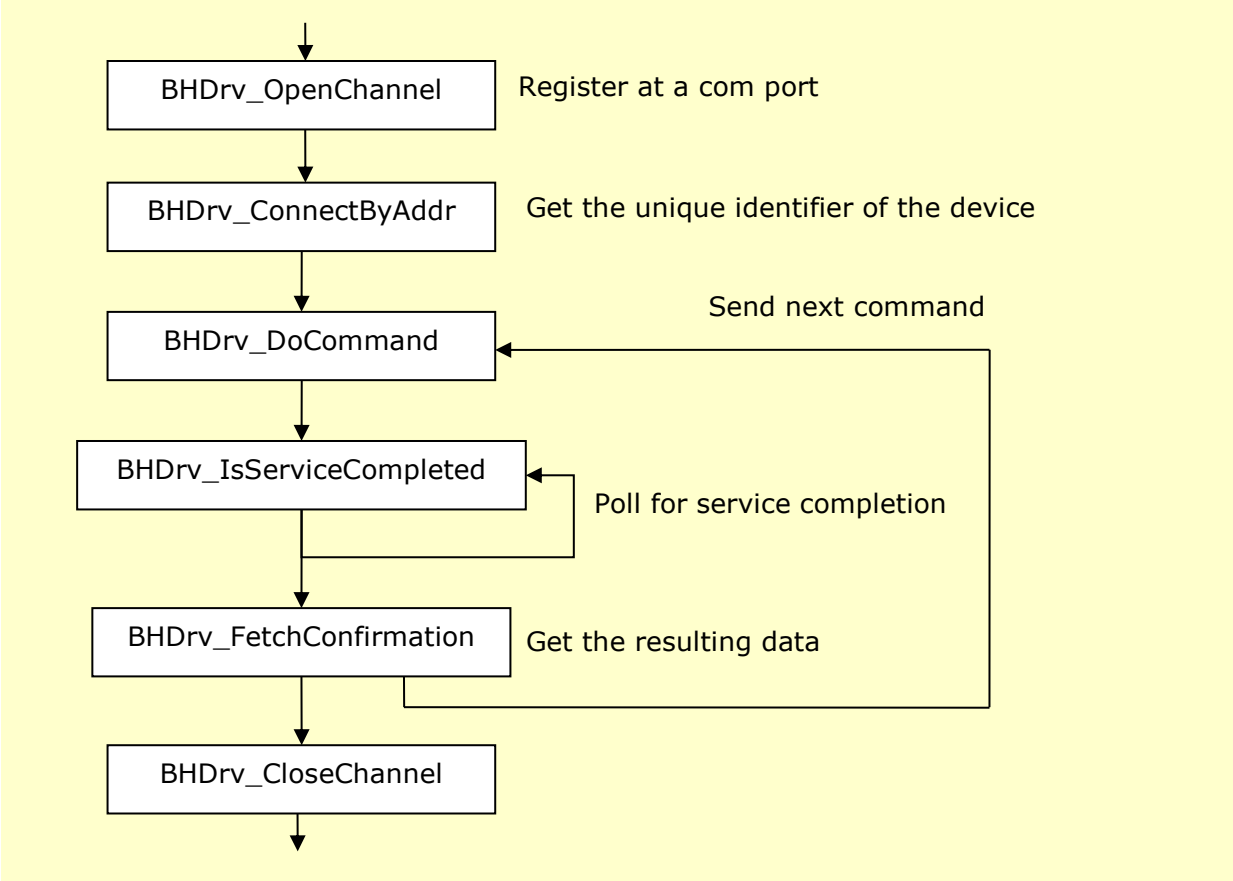


Figure 5: Polling for Service Completion

Because command 0 is the only command in Hart which is working with the short address (0..15/0..63) the unique identifier has to be fetched from the device to use it for the other commands. The unique identifier can be read by the commands 0, 11 and 21.

There are three ways to wait for the completion of a service. Picture 1 is showing the no wait mode. In the no wait mode the client program has to poll the DLL by calling BHDrv_IsServiceCompleted.

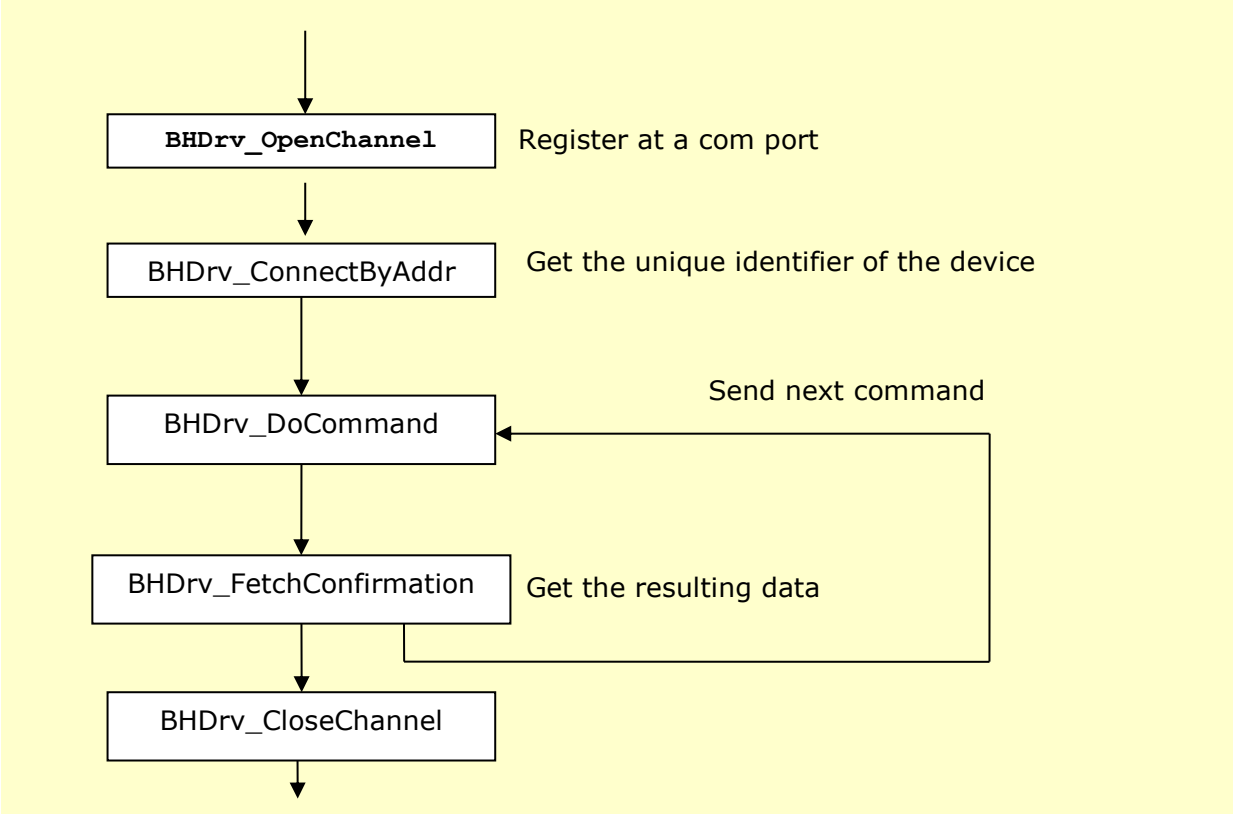


Figure 6: Using the Wait Mode of the DLL

When a service is processed using the function BHDrv_DoCommand with the option flag DRV_WAIT the program is returning when the service is totally completed even if there are errors or if the device is not responding.
Waiting for a service results in a small delay of approximately 250 ms.

Note: If a device is not responding, the function delay for a multiple of the number of retries which had been configured by the function BHDrv_SetConfiguration.

The third method is to register a callback function from the application software. In this case the DLL will call back as soon as any service of that application is completed.

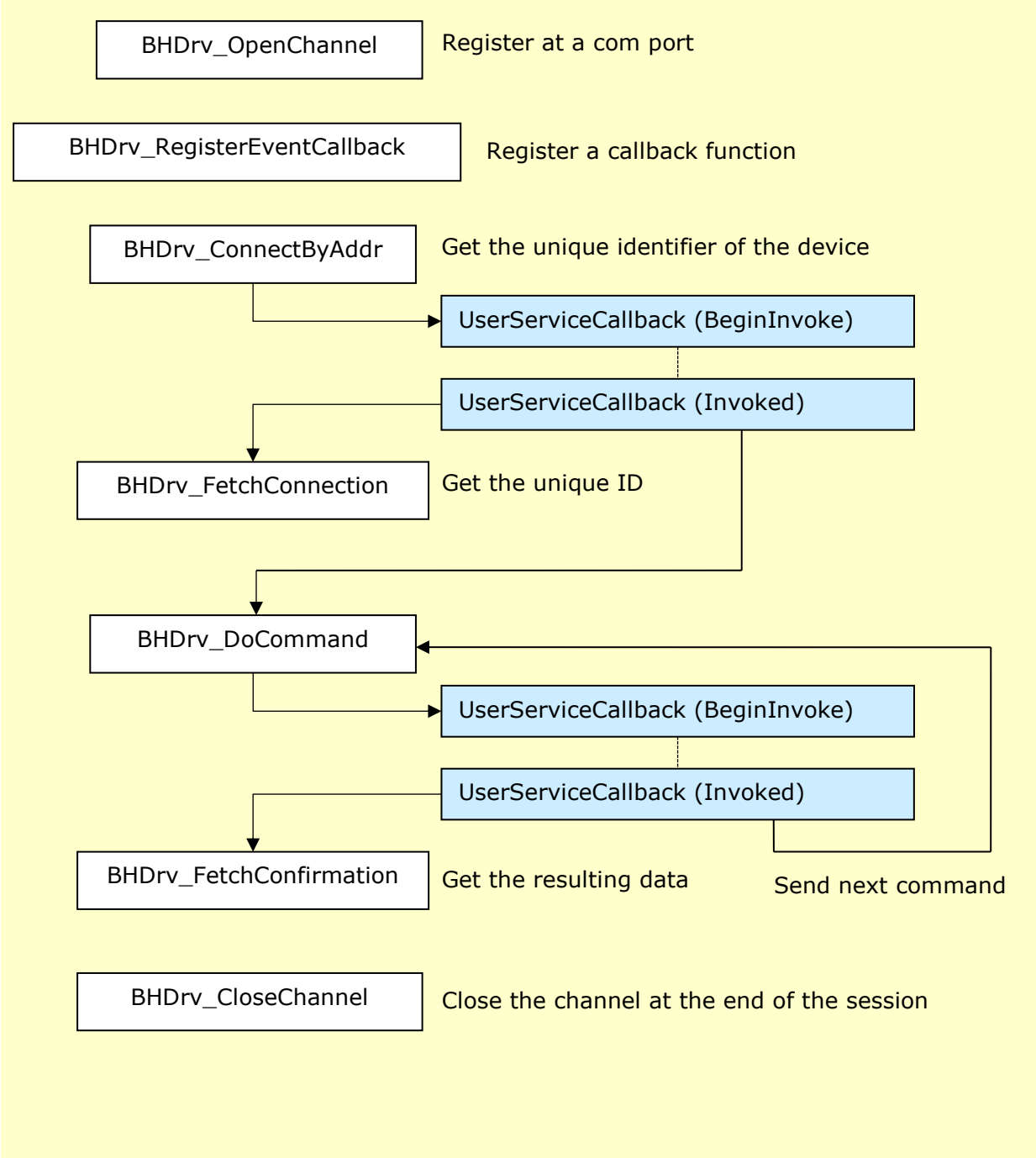


Figure 7: Using the a Callback Function for the DLL

The time between the call of the callback function and the execution of the invoked function is not determined because it is given by the Windows messaging system. But usually this time is short if the application is not busy in another event procedure.

Principle of Operation

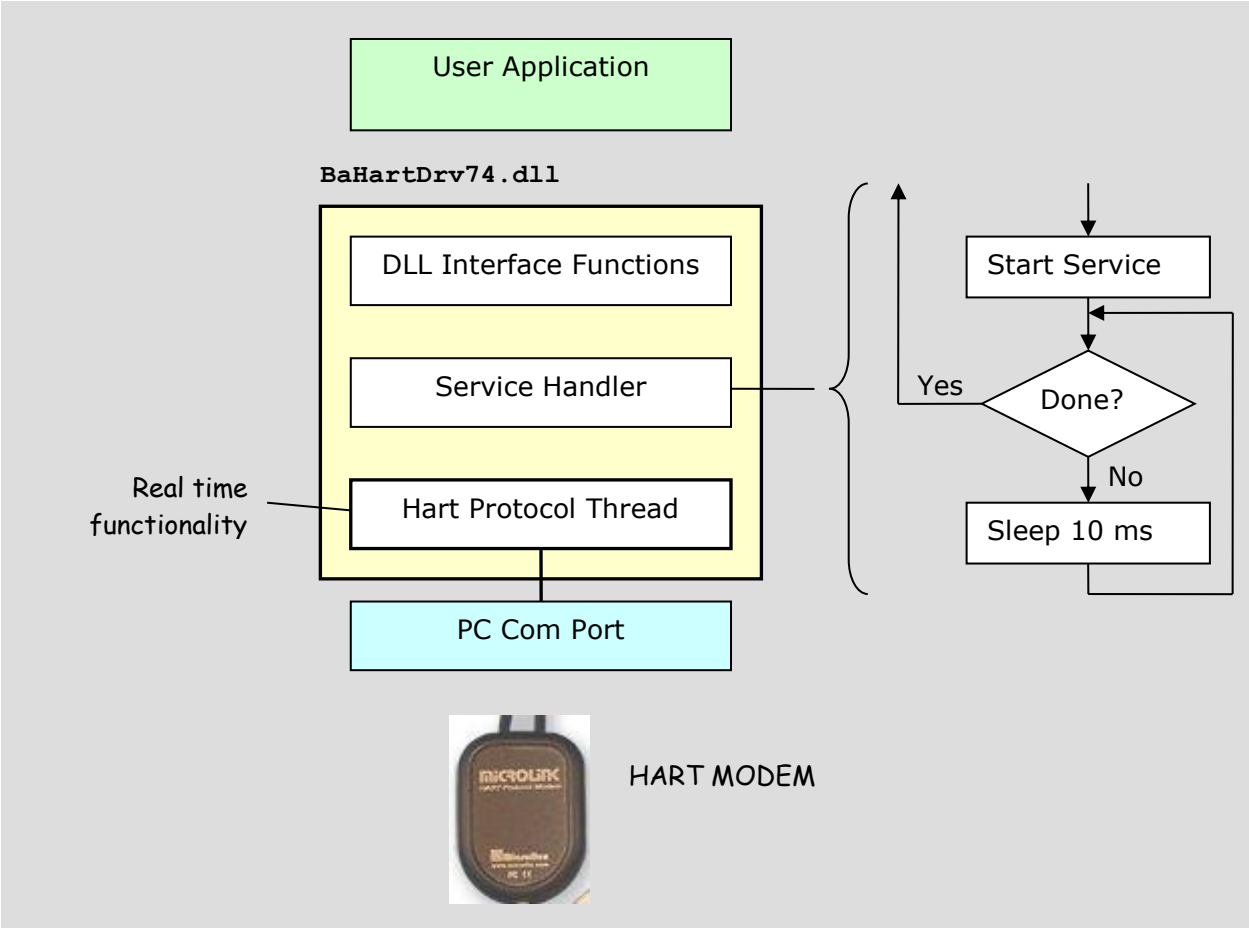


Figure 8: The Internal Structure of the DLL

The figure above shows that the DLL is using is using its own thread for the real time application. Thus the calling thread may be of any kind. Even if the DLL is waiting for the completion of the service it is taking the calling thread into sleep mode.

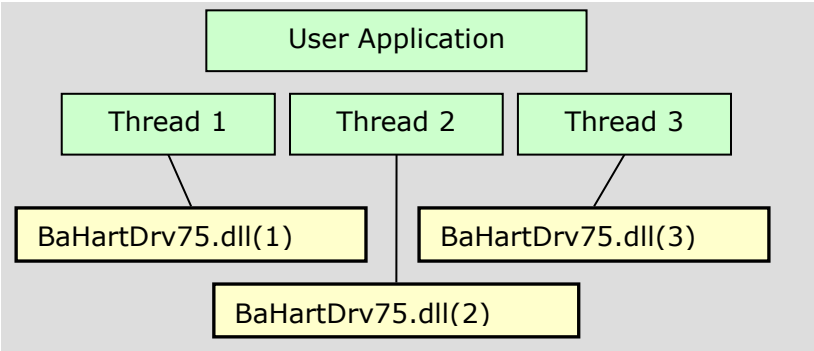


Figure 9: The DLL can be used by different Threads

The DLL may be called from several threads. The functions and communication services are thread safe. Each thread should register explicitly to get its own handle.

HartX

Service Processing Flow Diagram

If the wait flag is set in the call of DoCommand the following program flow is executed.

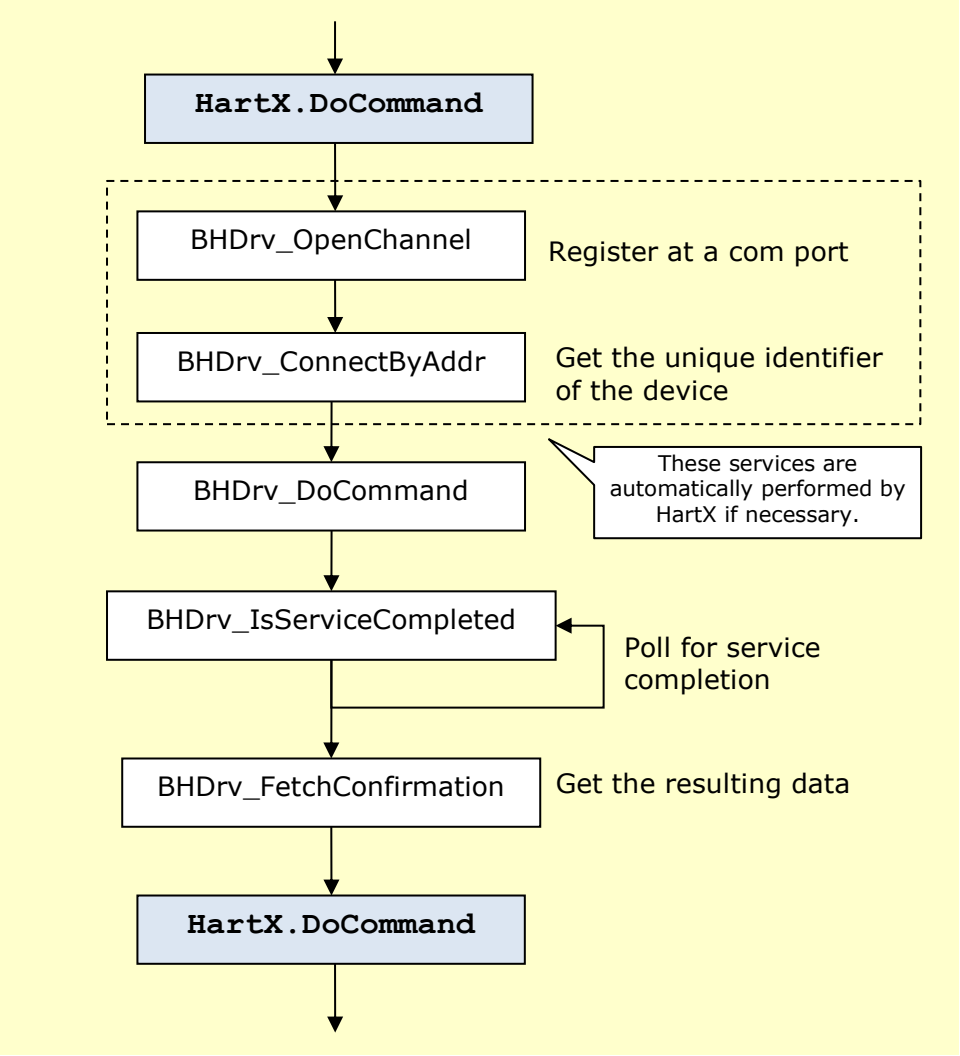


Figure 10: HartX Service Flow (waiting for service)

If the wait flag is cleared in the call of DoCommand will return immediately. After the service completion an event procedure will be called.

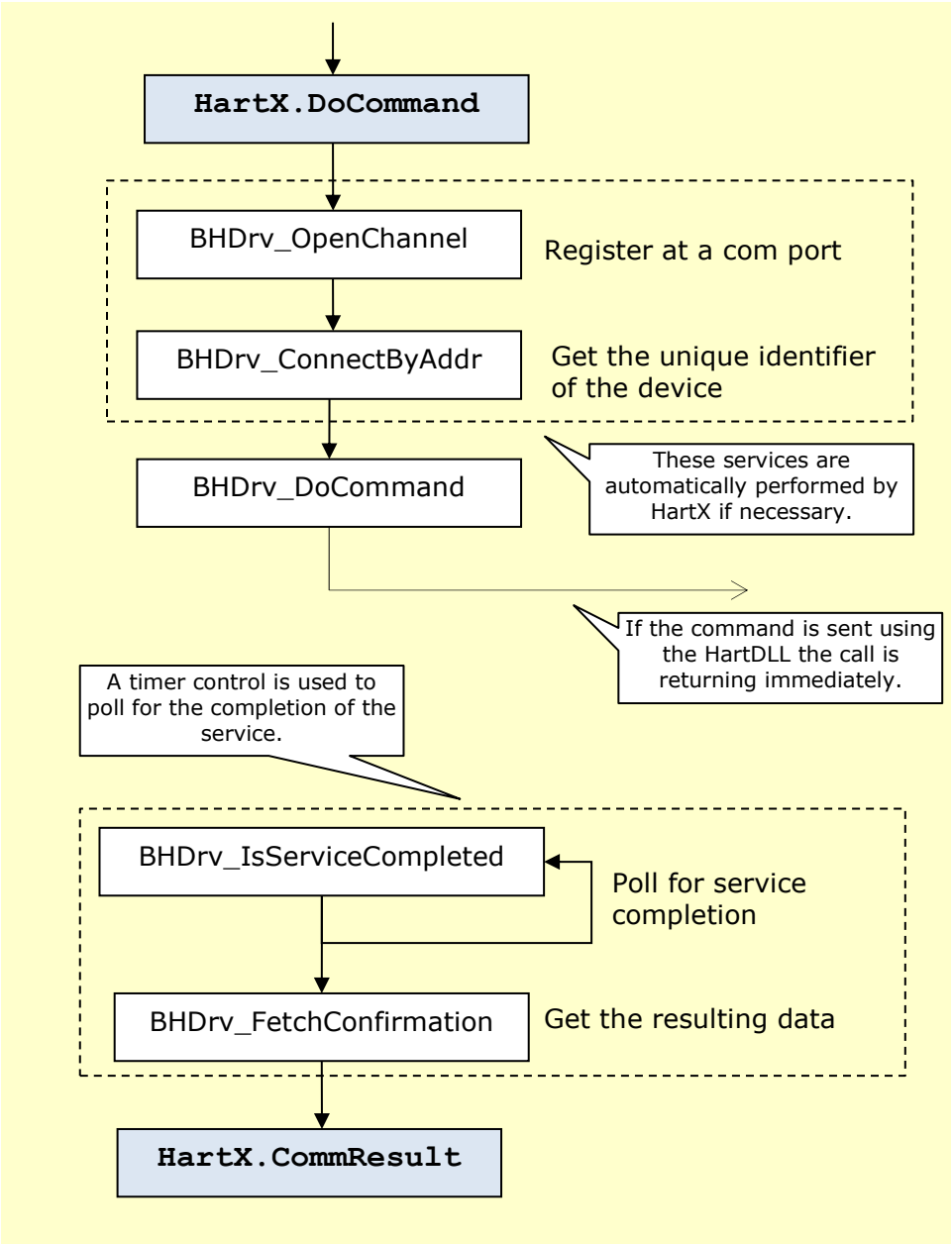


Figure 11: HartX Service Flow (not waiting for service)

Principle of Operation

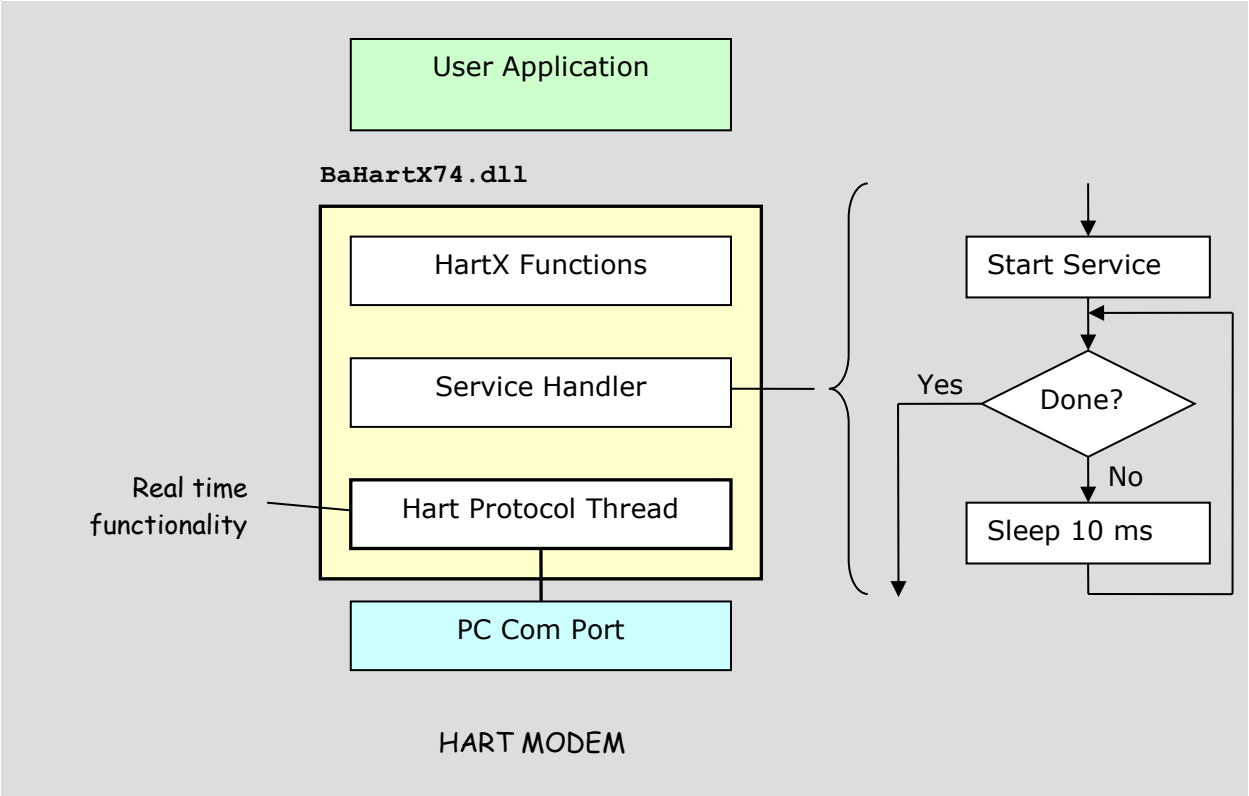


Figure 12: The Internal Structure of the DLL

The figure above shows that the HartX is using is using its own thread for the real time application. Thus the calling thread may be of any kind. Even if HartX is waiting for the completion of the service it is taking the calling thread into sleep mode.

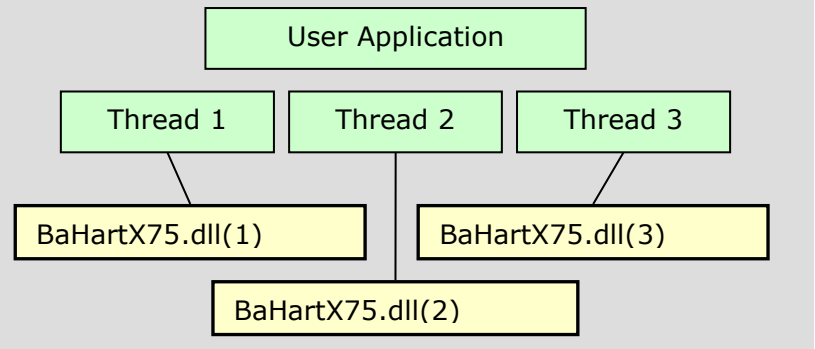


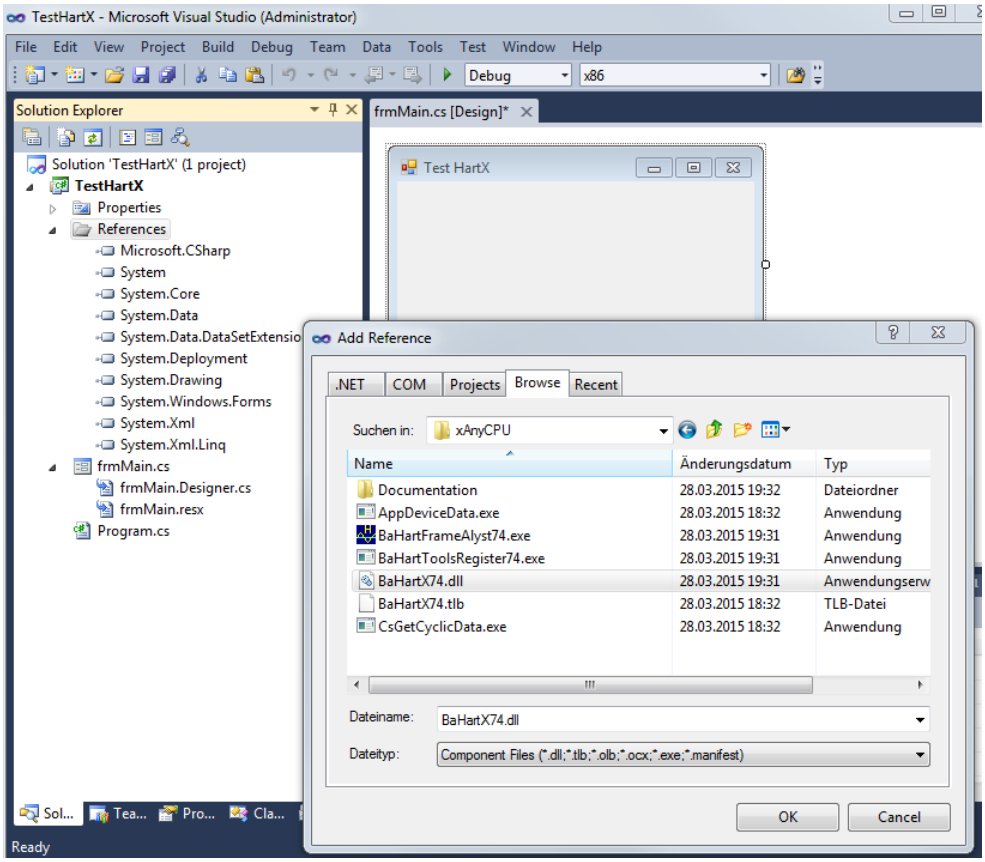
Figure 13: The DLL can be used by different Threads

HartX may be called from several threads. The functions and communication services are thread safe.

Visual Studio

Open Visual Studio and create a new project for a Windows Forms Application.

It is not necessary to install HartX74 on the toolbar. A simple reference to the library is enough.



The best way is to select the component from the path xAnyCPU because this library can be used in a 32 bit as well as in a 64 bit environment.

The next step is to set a reference in the namespace section.

```
namespace TestHartX
{
  using BaHartTools75.HartX;

  public partial class frmMain : Form
  {
    public frmMain()
    {
      InitializeComponent();
    }
  }
}
```

You should not forget to handle the licensing issue. Therefore a reference to the license module is set.

I recommend to include the module as a link to make sure that the module is shared and remains on its original place.

A variable is required to store a reference to the HartX.

```
  public partial class frmMain : Form
  {
    private CHartX hartX = null;

    public frmMain()
```

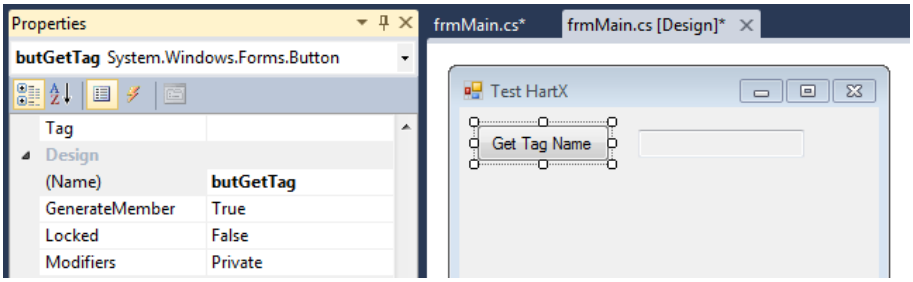
```
    {
```

The instance of HartX is inserted in the form load event handler. With setting the com port the HartDLL is loaded by the HartX and a channel for the communications is opened.

But before setting the com port the license has to be set in the HartX.

```
    private void frmMain_Load(object sender, EventArgs e)
    {
      this.hartX = new CHartX();
      this.hartX.ValidateLicense
        ("30-Days-Trial-User-License",
         " Ea58v60F-x3jk-wi9n-RrI3-7c072aA6ae0B");
      this.hartX.ComPort = 2;
    }
```

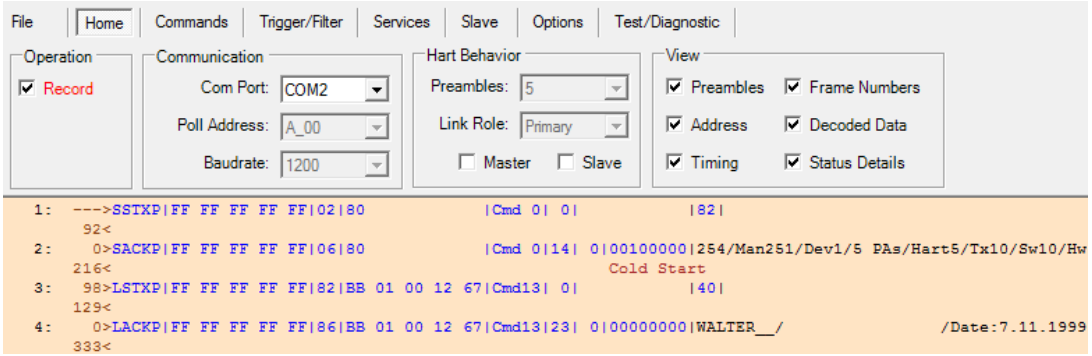
A button and a text box are used to perform some action.



The code required for reading the tag name is very short.

```
private void butGetTag_Click(object sender, EventArgs e)
{
  if (this.hartX.IsValidComPort)
  {
    // Read the tag name
    this.txtTagName.Text = "reading ...";
    this.hartX.XReqLen = 0;
    this.hartX.DoCommand(13, true);
    if (this.hartX.LastError == CHartX.EN_LastError.ERR_Success)
    {
      this.txtTagName.Text = this.hartX.P13TagName;
    }
    else
    {
      this.txtTagName.Text = "Error!";
    }
  }
}
```

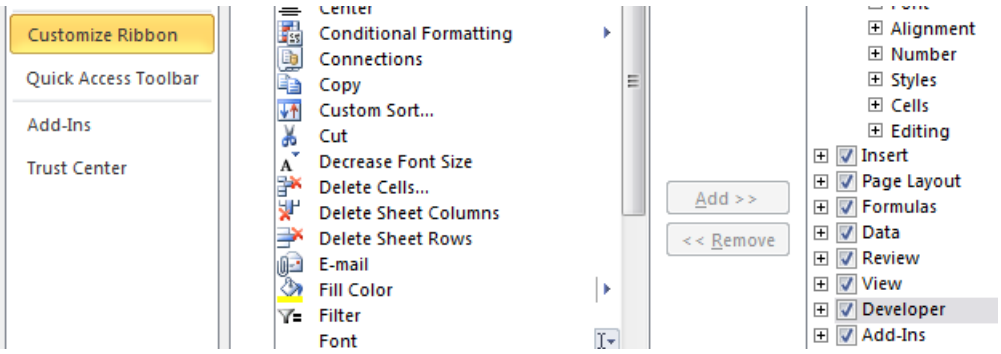
When clicking the button 'Get Tag Name' the following communication sequence is shown by FrameAlyst.



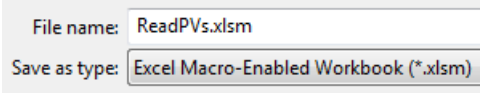
The HartX is firstly sending command 0 to get the unique identifier. Then the command 13 is used to get the Tag Name.

Excel

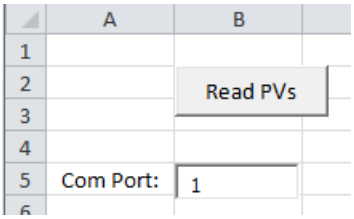
Before you can start to use VBA in Excel you have to activate the developer tabs in Options->Customize Ribbon.



To be sure that your macros (VBA program) are saved too you have to store the file as macro-enabled workbook.

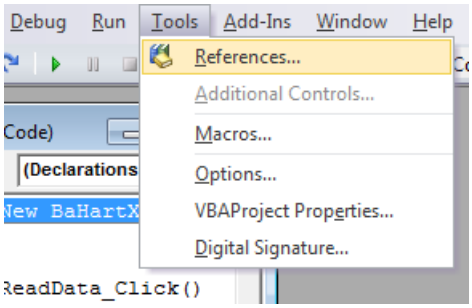


The example is using a button for starting and a textbox for the com port number.

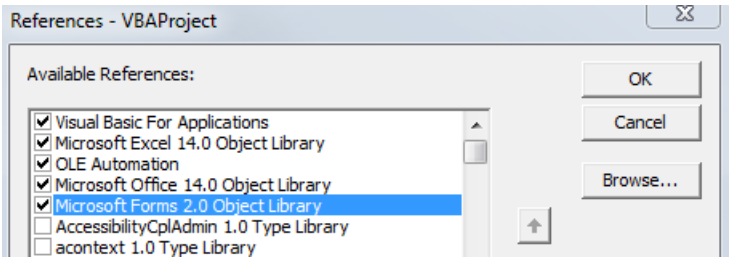


HartX is not a .net control but only a component. Therefore it has to be addressed by a reference. VBA does not accept a reference to the dll but to the type library (tlb) file.

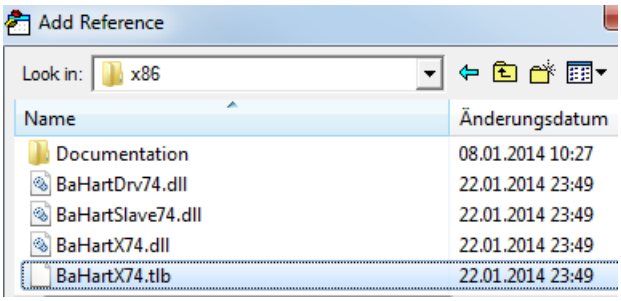
The reference has to be set in the code Window which is opened by the selection of 'View Code' in the Developer tab.

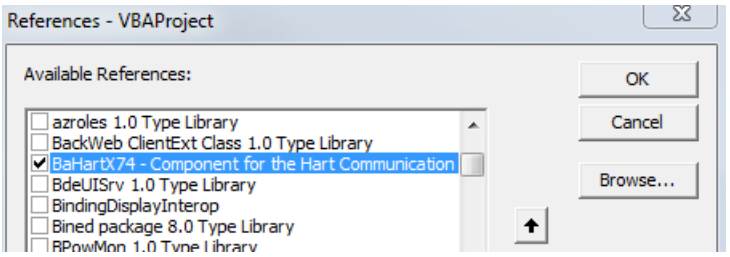


In the code window the menu Tools has the menu item References. After a click on this option the reference select Window opens.

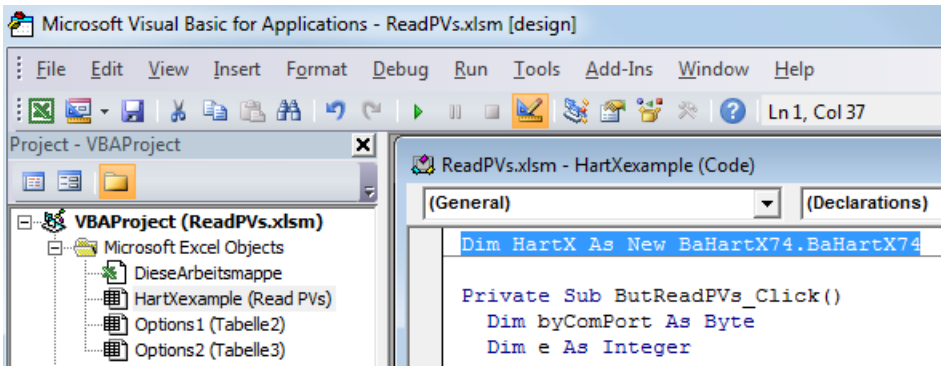


Click on browse and navigate to the tlb of the HartX.





Next is to declare an object using the HartX reference.



The example is coded in the event procedure of the button.

```
Private Sub ButReadPVs_Click()
  Dim byComPort As Byte
  Dim e As Integer
  Dim f As Single
  Dim sUser As String
  Dim sLicense As String

  'Set the user license code
  sUser = "30-Days-Trial-User-License"
  sLicense = "Ea58v60F-x3jk-wi9n-RrI3-7c072aA6ae0B"

  HartX.ValidateLicense sUser, sLicense

  byComPort = Val(TxtComPort.Text)
  If (byComPort > 0) And _
     (byComPort < 255) _
  Then
```

The first call of the HartX should be the call of the ValidateLicense method in order to set the HartDLL into a functional mode.

However the simulation of PVs also works without any License code.

```
' Set the com port
HartX.cComPort = byComPort
' Switch on simulation of the PVs
HartX.SimPvEnabled = True
' Set the amplitude to 10.0
HartX.SimAmplitude = 10#
' Initialize the cells
Cells(1, 4) = "Number"
Cells(1, 5) = "PV 1"
Cells(1, 6) = "PV 2"
For e = 0 To 19
  Cells(e + 2, 4) = ""
  Cells(e + 2, 5) = ""
  Cells(e + 2, 6) = ""
  'DoEvents
Next e
```

The only thing to do for the communications is to set the com port to which the Hart device is connected to.

The property SimPvEnabled is setting the simulation mode of the HartX. If this mode is set the PVs are simulate between values set by the SimulateAmplitude property.

The 'main program' of the example is a for loop reading two PVs from the device for 20 times and writing the results to the worksheet.

```
'Read 20 times PV 1 and PV 2
For e = 0 To 19
  HartX.DoAction 3
  f = e
  Cells(e + 2, 4) = Format(f, "0.0")
  f = HartX.p03Pv1
  Cells(e + 2, 5) = Format(f, "0.0")
  f = HartX.p03Pv2
  Cells(e + 2, 6) = Format(f, "0.0")
  'DoEvents
Next e
```

The call of DoAction is driving the simulation of the PVs and simulates a delay of 200 ms like the communication would do. In the case the simulation is switched of DoAction would run the Hart protocol activities. After running the example the worksheet will look as below.

	A	B	C	D	E	F
1				Number	PV 1	PV 2
2		Read PVs		0,0	0,0	-9,0
3				1,0	3,1	-8,0
4				2,0	5,9	-7,0
5	Com Port:	1		3,0	8,1	-6,0
6				4,0	9,5	-5,0
7				5,0	10,0	-4,0
8				6,0	9,5	-3,0
9				7,0	8,1	-2,0
10				8,0	5,9	-1,0
11				9,0	3,1	0,0
12				10,0	0,0	1,0
13				11,0	-3,1	2,0
14				12,0	-5,9	3,0
15				13,0	-8,1	4,0
16				14,0	-9,5	5,0
17				15,0	-10,0	6,0
18				16,0	-9,5	7,0
19				17,0	-8,1	8,0
20				18,0	-5,9	9,0
21				19,0	0,0	-10,0

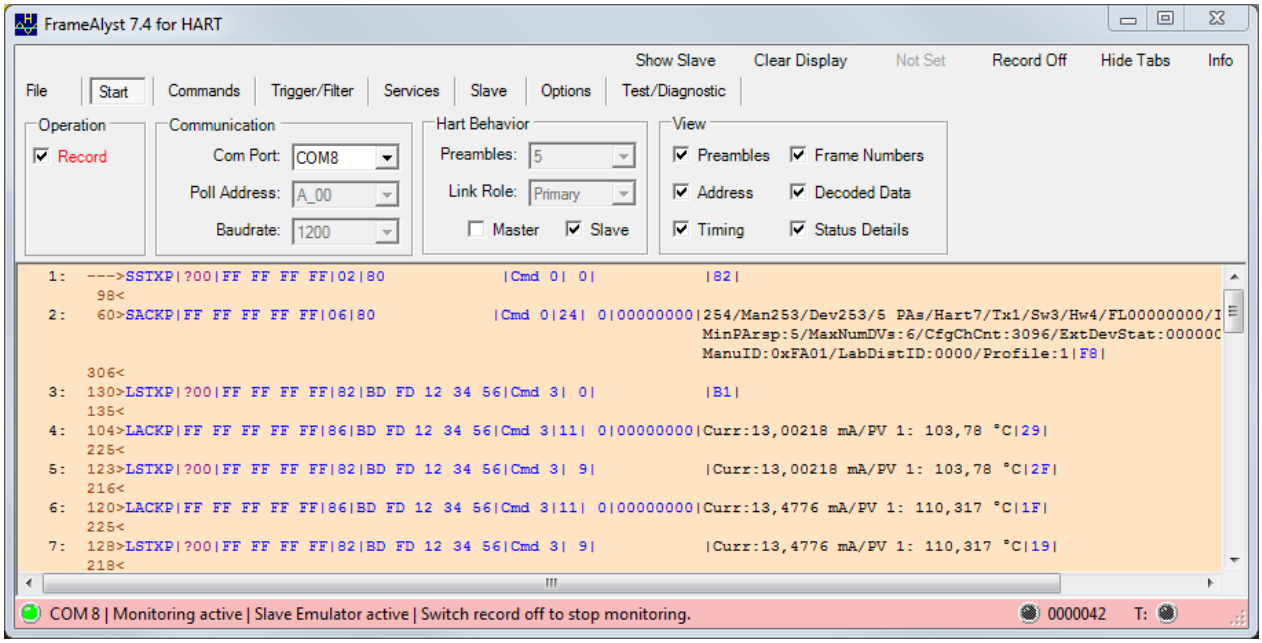
Running it with the simulation switched off, the example will communicate with the real device.

```
' Switch off simulation of the PVs
HartX.SimPvEnabled = False
```

The worksheet may look like it is shown below.

	A	B	C	D	E	F
1				Number	PV 1	PV 2
2		Read PVs		0,0	103,8	0,0
3				1,0	110,3	0,0
4				2,0	116,4	0,0
5	Com Port:	1		3,0	123,3	0,0
6				4,0	130,8	0,0
7				5,0	138,0	0,0
8				6,0	143,8	0,0
9				7,0	151,0	0,0
10				8,0	157,6	0,0
11				9,0	163,4	0,0
12				10,0	171,2	0,0
13				11,0	178,8	0,0
14				12,0	186,1	0,0
15				13,0	193,6	0,0
16				14,0	199,3	0,0
17				15,0	206,2	0,0
18				16,0	212,8	0,0
19				17,0	220,2	0,0
20				18,0	214,4	0,0
21				19,0	207,0	0,0

If you run FrameAlyst during the session you can see the communication activities.



Before starting to accept the command 3 requests HartX is automatically sending command 0 to retrieve the unique identifier from the device.

Appendix

Abbreviations

Abbreviation	Description
HCF	Hart Communication Foundation
DLL	Windows: Dynamic Link Library OSI-ISO: Data Link Layer
HAL	Hardware Abstraction Layer
HART	Highway Addressable Remote Transducer See also: http://en.wikipedia.org/wiki/Highway_Addressable_Remote_Transducer_Protocol
HMI	Human Machine Interface
ISO	International Standards Organisation
MODEM	MOdulator DEModulator
NV-memory	Non-Volatile memory
OSAL	Operating System Abstraction Layer
OSI	Open Systems Interconnection
UART	Universal Asynchronous Receiver Transmitter