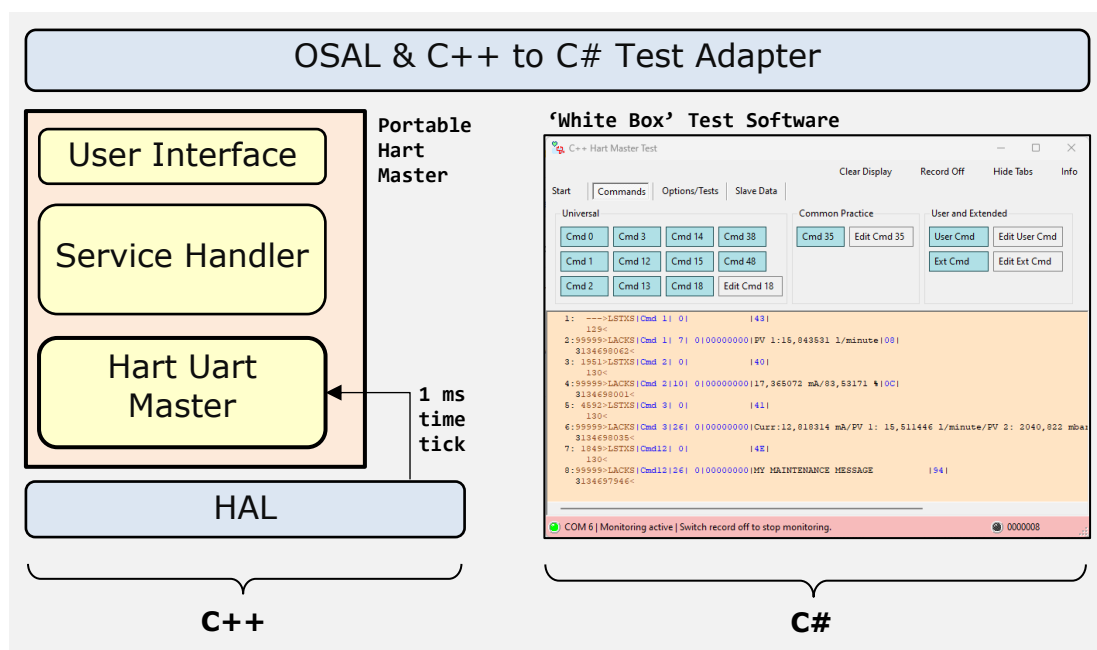


Hart Slave C++ 8.0E

Integration Example for Nrf52832 and GNU CC 4.7.4

To be completed ..

Introduction



The package Portable Hart Master includes all modules needed to represent the master part of the Hart protocol. The package is written in standard C++ and does not use any direct connection to a system environment. Data link layer, application layer and network management of the Hart protocol are implemented. The connection to the outside occurs via three interfaces: The User Interface, a Time Trigger and the HAL to the Uart interface. Special properties are:

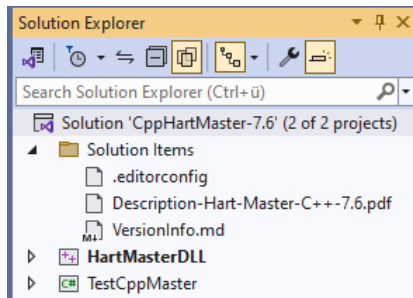
- No external dynamic memory management. The amount of reserved RAM remains constant.
- The number of objects is determined at compile time and startup.
- No operating system is required to integrate the software. Timers and serial interrupts are enough.
- The user interface is very close to the interface of the existing Hart DLL in HartTools 7.6.

In order to make the source code of the Hart Kernel visible, a simulation of an application is available. The simulation is carried out by integrating the Hart Kernel into a Windows library and controlling it via a 'Test Adapter'

Hart Slave C++ 7.6 Integration

Introduction.....	1
Visual Studio 2022	3
Prerequisites	3
Development Directory Structure	3
C++ Hart Master Code	4
Implementation Considerations	4
HAL (Hardware Abstraction Layer)	4
Architecture	5
Implementation Details	7
Directory Structure	7
Project Structure	7
List of Files.....	8
Public Functions.....	9
Embedded System Requirements.....	12
Windows Test Adapter	13
Overview	13
User Interface.....	13
Implementation Details	14
Directory Structure	14
Project Structure	15
List of Files.....	15
Code Walkthrough	17
Establishing a Connection	17
Executing a Command.....	18
Additional Information.....	19
Type Definitions	19
Coding Conventions	19
Appendix.....	20
Internet Links	20
Abbreviations.....	20
Download Location	20
Legal Issues.....	21
Conformity	21
Copyright	21
Warranty Disclaimer	21

Visual Studio 2022



There are only two projects in the solution. The C++ Hart Master is encapsulated in the HartMasterDLL project, while the test software (with C# and .NET) can be found in the TestCppMaster project.

The solution is in the path: `.\BorstAutomation\EmbHart\` depending on which directory you copied the package to.

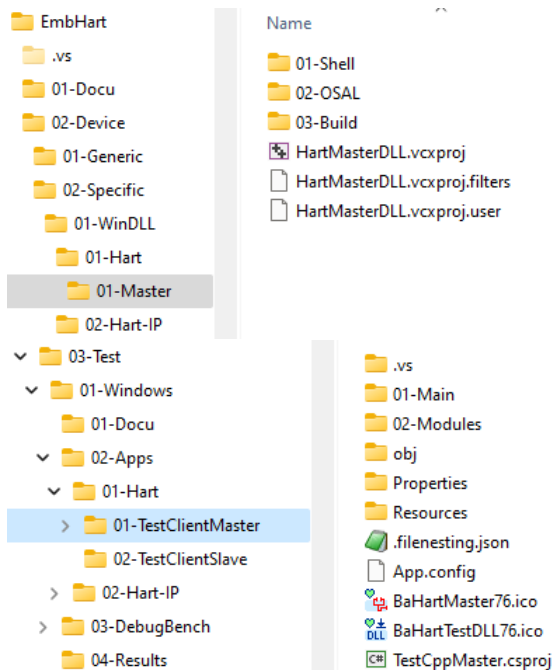
Prerequisites

Microsoft Visual Studio Community 2022 (64-bit)
Version 17.9.6
© 2022 Microsoft Corporation.
All rights reserved.

Microsoft .NET Framework
Version 4.8.09032
© 2022 Microsoft Corporation.
All rights reserved.

The solution must be opened with VS 2022. However, the community version is sufficient. There are no further requirements.

Development Directory Structure



The source code for the Hart Master in C++ can be found in the directory:
`.\02-Device\02-Specific\01-WinDLL\01-Hart\01-Master`

The test software is in the test area:
`.\02-Device\03-Test\01-Windows\02-Apps\01-Hart\01-TestClientMaster`.

The 03-DebugBech directory is also important in this context. The executable file `TestCppMaster.exe` and the simulation DLL `BaHartMaster-7.6.dll` are both located here.

C++ Hart Master Code

Details for the Hart Protocol are provided via the following link:

<https://www.fieldcommgroup.org/technologies/hart>.

Implementation Considerations

Microcontrollers which are used today for HART devices are at least 16 Bit microcontrollers. Otherwise the complexity of the measurement and number of parameters could not be managed.

☞ Low amount of memory.

The amount of memory is always critical because software kind of behaves like an ideal gas. It uses to fill the given space. Nevertheless, the coding of the Hart Master was done as carefully as possible regarding the amount of flash memory and RAM.

☞ The user needs source code.

The Hart Protocol requires a strict timing specially for burst mode support and the primary and secondary master time slots. To provide the optimum transparency to the user to allow all kinds of debugging and to give the opportunity to optimize code in critical sections, the Hart Master Software is not realized as a library but delivered as source code.

HAL (Hardware Abstraction Layer)

☞ OSAL is including the HAL.

A Hardware Abstraction Layer is needed to design the interface of a software component independent from the hardware platform. In this very small interface of the Hart master a distinction of HAL and OSAL was not made. Therefore only an Operating System Abstraction Layer is defined which is covering all the needs of an appropriate HAL.

Architecture

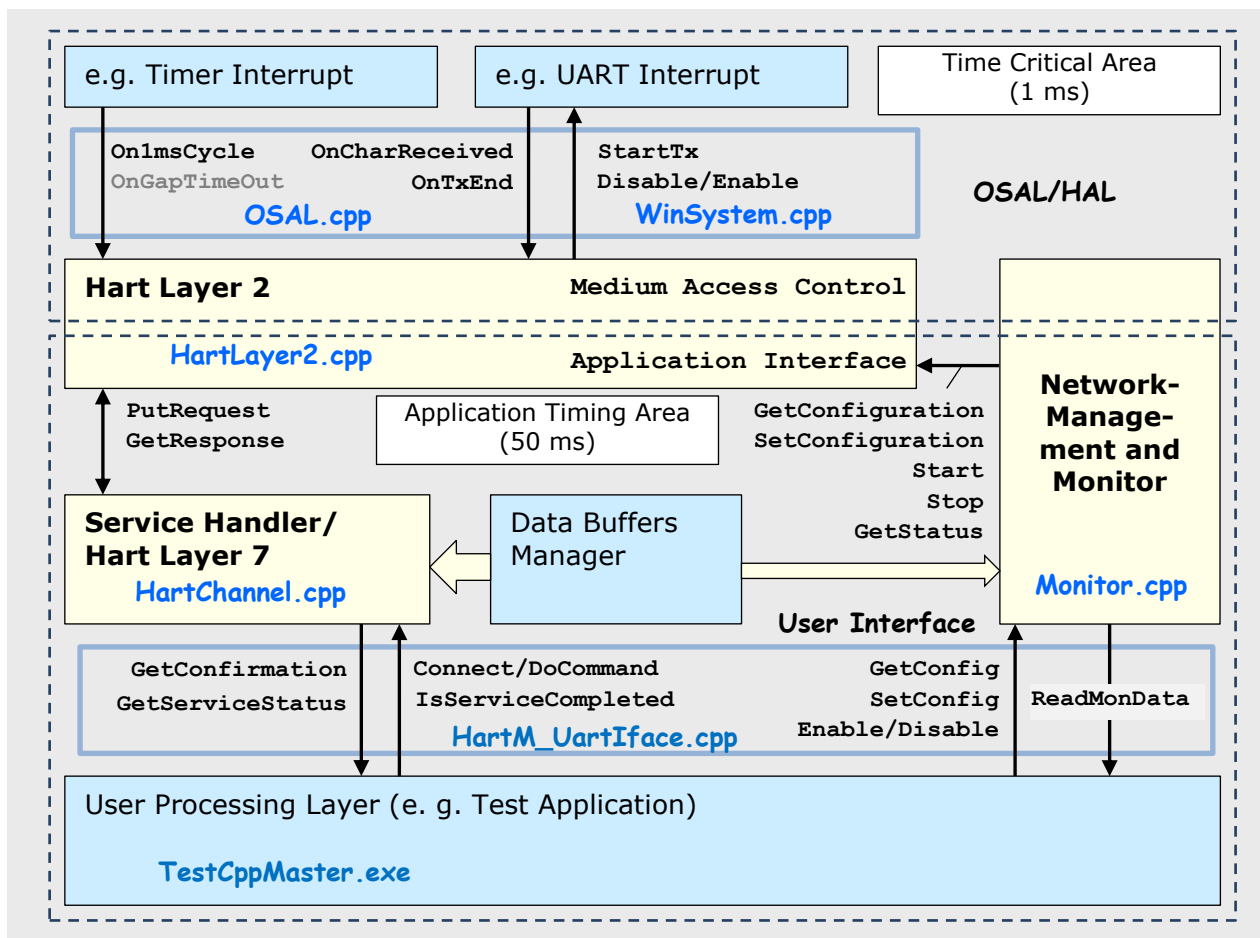


Figure 1: Architecture of the Implementation

The software is mainly divided into two areas. One is the time critical part, which is needed to meet the requirements of the time controlled dual master protocol of Hart. The other is the area where the application software is working, which is far less time critical.

☞ Find the source code in the figure above.

In the diagram above, for some modules I have clearly marked in which file you can find their implementation. This detail is particularly important to me so that you don't think that the diagram is pure theory. The file names are marked with blue color.

☞ The software architecture is optimized for systems with a few resources.

The figure above is clearly showing also two user interfaces. There is a user interface (OSAL/HAL) which is connecting the Hart Master software to a timer control interrupt and a UART interrupt which are used for the 'fast' service procedures. Most of the Hart protocol functionalities are solved in the timer part, which may run on interrupt level. There are arguments for and against this kind of implementation but you ever end up at a point that the incoming frame has to be processed as quickly as possible. So why not spending a few microseconds more once the program has already reached the interrupt level. The Hart

protocol is not very complex but it needs to be processed fast enough to catch a precise timing.

The load produced by the implementation is not very high. Because the communication runs with a speed of 1200 bit/s usually there is nothing to do in the 1 ms cycle than to keep track of the timing. Only every 10 ms - if a frame is coming in - a character has to be processed. The processing is done in an incremental way thus not implying the execution of too much instructions.

The split between the time critical area and the user application is done within the Data Link Layer and the so called Network Management. However, the user have not to take any special on these separations except the provision of a few OSAL services for Locking out other tasks. There is an 'atomic' lock out level which has to lock out the interrupts of the Data Link Layer as well as concurrent processes. The other level is 'critical section' which is locking out concurrent processes. More details are described in another chapter of this document.

☞ The top level user interface is at least platform independent.

The interface to the user's application is located on top of the User Data Processing Layer (User Processing Layer). The functions that are made available in this interface are implemented in the file HartM_UartIfc.cpp and are described in detail in the 'Public Functions' chapter.

There are a few data objects which are required for Hart protocol and which may be set by the user or an external Hart master. These are such as the tag name and the address. If e.g. the address of the Hart slave is changed through the network the Network-Management will call the user layer to store the data in the NV-memory. If the address is changed through the local HMI of the Hart device, the user layer calls Network-Management of Hart to advise the Data Link Layer protocol to work with the new address. The function used for this setting is SetConfiguration.

In the above figure the parts of the Hart Master are shown in yellow color while the user parts are marked with blue.

☞ The Data Link Layer is an independent piece of software.

The figure is also showing a set of functions between the command executor the network management and the data link layer. These functions may be used if the developer decides to use only the data link layer by providing its own command executor and network management.

Implementation Details

Directory Structure

The following table shows how the individual files are distributed among the directories.

01-Generic					
<ul style="list-style-type: none"> 02-Device <ul style="list-style-type: none"> 01-Generic <ul style="list-style-type: none"> 01-Hart <ul style="list-style-type: none"> 01-Common 02-Master 	<ul style="list-style-type: none"> HartCoding.cpp HartCoding.h HartFrame.cpp HartFrame.h HartLib.h 	<ul style="list-style-type: none"> 01-Generic <ul style="list-style-type: none"> 01-Hart <ul style="list-style-type: none"> 01-Common 02-Master 03-Slave 	<ul style="list-style-type: none"> Name 01-Interface 02-AppLayer 03-Layer7 04-Layer2 	<ul style="list-style-type: none"> 02-Master <ul style="list-style-type: none"> 01-Interface 02-AppLayer 03-Layer7 04-Layer2 	<ul style="list-style-type: none"> HartM_Uartiface.cpp HartM_Uartiface.h WbHartM_Structures.h WbHartM_Typedefs.h WbHartUser.h
<ul style="list-style-type: none"> 02-Master <ul style="list-style-type: none"> 01-Interface 02-AppLayer 03-Layer7 	<ul style="list-style-type: none"> HartChannel.cpp HartChannel.h 	<ul style="list-style-type: none"> 01-Interface 02-AppLayer 03-Layer7 04-Layer2 	<ul style="list-style-type: none"> HartService.cpp HartService.h 	<ul style="list-style-type: none"> 02-AppLayer 03-Layer7 04-Layer2 03-Slave 02-Hart-IP 02-Specific 	<ul style="list-style-type: none"> HartLayer2.cpp HartLayer2.h HMMacPort.h HMUartProtocol.cpp HMUartProtocol.h Monitor.h
02-Specific					
<ul style="list-style-type: none"> 02-Specific <ul style="list-style-type: none"> 01-WinDLL 01-Hart <ul style="list-style-type: none"> 01-Master 	<ul style="list-style-type: none"> 01-Shell 02-OSAL 03-Build HartMasterDLL.vcxproj 	<ul style="list-style-type: none"> 01-Master <ul style="list-style-type: none"> 01-Shell 02-OSAL 	<ul style="list-style-type: none"> Name BaHartMaster-7.6.c BaHartMaster-7.6.h 	<ul style="list-style-type: none"> 01-Master <ul style="list-style-type: none"> 01-Shell 02-OSAL 03-Build 02-Hart-IP 	<ul style="list-style-type: none"> HMMacPort.cpp Monitor.cpp OSAL.cpp WinSystem.cpp WinSystem.h

Project Structure

<ul style="list-style-type: none"> 01-Generic <ul style="list-style-type: none"> 01-Common 02-Master <ul style="list-style-type: none"> 01-Interface <ul style="list-style-type: none"> HartM_Uartiface.cpp HartM_Uartiface.h WbHartM_Structures.h WbHartM_Typedefs.h WbHartUser.h 02-AppLayer <ul style="list-style-type: none"> HartChannel.cpp HartChannel.h 03-Layer7 <ul style="list-style-type: none"> HartService.cpp HartService.h 04-Layer2 <ul style="list-style-type: none"> HartLayer2.cpp HartLayer2.h HMMacPort.h HMUartProtocol.cpp HMUartProtocol.h Monitor.h OSAL.h 02-Specific(Windows) <ul style="list-style-type: none"> 01-Shell 02-OSAL <ul style="list-style-type: none"> HMMacPort.cpp Monitor.cpp OSAL.cpp WinSystem.cpp WinSystem.h 	<p>The project structure is very similar to the directory structure. Here too there is a strict distinction between generic area and specific area. The specific contents of the files are described in more detail in the list below.</p>
--	--

List of Files

Category	Name	Description
02-Device		
01-Generic	OSAL.h	The Operating System Abstraction Layer is the top header. This is where the central connection to the respective hardware or software platform takes place. The header OSAL.h can only exist once, while a special implementation (OSAL.cpp) exists for each specific hardware or software.
02-Device\01-Generic\01-Hart		
01-Common	HartCoding.cpp/h	This module combines functions that carry out the encoding and decoding of communication primitives and data objects.
	HartFrame.cpp/h	The hart frame is a construct used to collect all information which is needed to encode and decode data of so called service primitives like responses and requests, which are finally octet streams.
	HartLib.h	Some classes for the definition of HART constants.
02-Device\01-Generic\01-Hart\02-Master		
01-Interface	HartM_UartIface.cpp/h	This is where the actual interface of the master implementation is located, which would also have to be integrated into an embedded system. The version with the DLL is only intended for testing under Windows. You can find a detailed description of the provided functions in the 'Public Functions' chapter.
	WbHartM_Structures.h	This file contains structures which are accessed at the outer interface as well as in some modules in the master kernel.
	WbHartM_TypeDefs.h	This file contains type definitions which are used in all modules in the Hart master kernel.
	WbHartUser.cpp	Limits applied by the user of the hart master software.
02-AppLayer	HartChannel.cpp/h	The channel manages a communication interface and the associated properties. The channel also uses services to conduct Hart commands.
03-Layer7	HartService.cpp/h	In simple terms, a service executes a Hart command by passing a request to Layer2 of the Hart protocol. In doing so, it returns a handle to the caller, with which the calling program can check the status. A service is only considered completed when the caller has read the response (e.g. FetchConfirmation).
04-Layer2	HartLayer2.cpp/h	This module implements the entire state machine of the Hart communication protocol (CHartSM) including the state machines for sending (CTxSM) and receiving (CRxSM) bytes.
	HMMacPort.h	The interface to the MAC port is relatively narrow and can be defined generically. However, the implementation depends on the hardware and software environment. That's why there is only a header at this point, while the file HMMacPort.cpp can be found in the specific branch.
	HMUartProtocol.cpp/h	This protocol layer controls the UART interface on the lower level and calls the higher status machines when necessary (events). After this call, a ToDo Part occurs, which in turn affects the Uart interface.
	Monitor.h	The same applies to the Monitor function as to the MacPort. At this point only the interface can be defined. The implementation takes place in the specific part.
02-Device\02-Specific\01-WinDLL\01-Hart\01-Master		
01-Shell	BaHartMaster-7.6.cpp/h	The implementation for the calls to the Windows DLL is located here. In practice, it is just a shell through which the functions in the CUartMaster module are called. See also HartM_UartIface.cpp/h.
02-OSAL	HMMacPort.cpp	The Execute method is called directly by the fast cyclic handler. This basically drives all status machines in the Hart implementation. Here too, the method is divided into an Event handler and a ToDo handler.
	Monitor.cpp	On the one hand, there are methods that are mapped to the interface of the Windows DLL. In addition, there are a number of functions that are included with the kernel functions. Since this module is so small overall, the methods were not implemented in two different files.
	OSAL.cpp	The Operating System Abstraction Layer maps general functions to the operating system.
	WinSystem.cpp/h	The OSAL concept cannot be applied to all functions that are required. These functions were implemented in the code of this module.

Public Functions

The following functions are realized in the module HartM_UartIfc.cpp in the class CUartMaster. In the DLL interface for the test client the function names are preceded by BAHAMA_.

Declaration	Description
Operation	
<code>EN_Bool OpenChannel(TY_Word port_number_, EN_CommType type_);</code>	The function allocates the selected com port if possible and starts its own working thread for accessing the Hart services. The port_number_ is limited to the range of 1 .. 254. The selected communication type (type_) should be UART in this version of the paket. The function returns TRUE8 if successful. In the present implementation only a single channel is possible. Thus no channel handle is required.
<code>void CloseChannel();</code>	It is required to call this function at least when the application is terminating.
<code>void GetConfiguration(TY_Configuration* config_);</code>	The function copies the configuration data to a data structure provided by the caller.
<code>void SetConfiguration(TY_Configuration* config_);</code>	The function is setting all details required for the configuration. The data is passed in a structure provided by the caller.
Connection Services	
<code>SRV_Handle ConnectByAddr(TY_Byte address_, EN_Wait qos_, TY_Byte num_retries_);</code>	Use command 0 with short address to get the connection information.
	address_ 0 .. 63
	qos_ NO_WAIT(0), WAIT(1)
	num_retries_ 0 .. 10
The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error.	
<code>SRV_Handle ConnectByUniqueID(TY_Byte* data_ref_, EN_Wait qos_, TY_Byte num_retries_);</code>	Use command 0 with long address to get the connection information.
	data_ref_ Pointer to a five byte array with the unique identifier
	qos_ NO_WAIT(0), WAIT(1)
	num_retries_ 0 .. 10
The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error. Note: The function is not yet implemented.	
<code>SRV_Handle ConnectByShortTag(TY_Byte* data_ref_, EN_Wait qos_, TY_Byte num_retries_);</code>	Use command 11 with global address to get the connection information.
	data_ref_ Pointer to the byte array of a length of 6 packed ASCII bytes
	qos_ NO_WAIT(0), WAIT(1)
	numRetries 0 .. 10
The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error. Note: The function is not yet implemented.	
<code>SRV_Handle ConnectByLongTag(TY_Byte* data_ref_, EN_Wait qos_, TY_Byte num_retries_);</code>	Use command 21 with global address to get the connection information.
	data_ref_ Pointer to the 32 byte ISO Latin 1 string with the long tag name
	qos_ NO_WAIT(0), WAIT(1)
	num_retries_ 0 .. 10
The function returns a service handle if successful or INVALID_SRV_HANDLE if there was an error. Note: The function is not yet implemented.	
<code>void FetchConnection(SRV_Handle handle_, TY_Connection* connection_);</code>	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

<pre>SRV_Handle LaunchCommand(TY_Byte command_, EN_Wait qos_, TY_Byte* data_ref_, TY_Byte data_len_, TY_Byte* bytes_of_unique_id_);</pre>	Send a command in the range 0..255.	
	command_	Hart command (0..255) to be sent with the request
	qos_	NO_WAIT(0), WAIT(1)
	data_ref_	Pointer to a native byte array which is sent as payload data
	data_len_	Length of the byte array
	bytes_of_unique_id_	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.	
<pre>SRV_Handle LaunchExtCommand(TY_Word command_, EN_Wait qos_, TY_Byte* data_ref_, TY_Byte data_len_, TY_Byte* bytes_of_unique_id_);</pre>	Send a command in the range 0..65535.	
	command_	Extended Hart command (0..65535) to be sent with the request
	qos_	NO_WAIT(0), WAIT(1)
	data_ref_	Pointer to a native byte array which is sent as payload data
	data_len_	Length of the byte array
	bytes_of_unique_id_	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: The function is not yet implemented.	
<pre>EN_Bool IsServiceCompleted(SRV_Handle service_);</pre>	Returns TRUE8 if the service (service_) was completed.	
<pre>void FetchConfirmation(SRV_Handle service_, TY_Confirmation* conf_data_);</pre>	Fills a structure provided by the caller with the service results information such as the response codes and the response data (if any).	
Encoding		
<pre>void PutInt8(TY_Byte data_, TY_Byte offset_, TY_Byte* data_ref_);</pre>	Insert an integer 8 into the byte array buffer pointed to by data_ref_ starting at the position offset_.	
<pre>void PutInt16(TY_Word data_, TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Insert an integer 16 into the byte array buffer pointed to by data_ref_ starting at the position offset_. Start with the most significant byte if endian is MSB_FIRST(0), which is the Hart standard.	
<pre>void PutInt24(TY_DWord data_, TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Insert an integer 24 into the byte array buffer pointed to by data_ref_ starting at the position offset_. Start with the most significant byte if endian is MSB_FIRST(0), which is the Hart standard.	
<pre>void PutInt32(TY_DWord data_, TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Insert an integer 32 into the byte array buffer pointed to by data_ref_ starting at the position offset_. Start with the most significant byte if endian is MSB_FIRST(0), which is the Hart standard.	
<pre>void PutInt64(TY_DWord data_, TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Insert an integer 64 into the byte array buffer pointed to by data_ref_ starting at the position offset_. Start with the most significant byte if endian is MSB_FIRST(0), which is the Hart standard.	
<pre>void PutFloat(TY_Float data_, TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Insert a single precision IEEE 754 float value into the byte array buffer pointed to by data_ref_ starting at the position offset. Start with the most significant byte if endian is MSB_FIRST(0), which is the Hart standard.	
<pre>void PutDFloat(TY_DFloat data_, TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Insert a double precision IEEE 754 float value into the byte array buffer pointed to by dataRef starting at the position offset. Start with the most significant byte if endian is MSB_FIRST(0), which is the Hart standard.	

Technical Documentation

<pre>void PutPackedASCII(TY_Byte* asc_string_ref_, TY_Byte asc_string_len_, TY_Byte offset_, TY_Byte* data_ref_);</pre>	Insert a string (asc_string_ref_) of the length of asc_string_len_ in packed ASCII format into the byte array buffer pointed to by data_ref_ starting at the position offset_. It is recommended that asc_string_len_ is an ordinary multiple of 4.
<pre>void PutOctets(TY_Byte* stream_ref_, TY_Byte stream_len_, TY_Byte offset_, TY_Byte* data_ref_);</pre>	Copy a number of stream_len_ bytes into the byte array buffer pointed to by data_ref_ starting at the position offset_.
<pre>void PutString(TY_Byte* string_ref_, TY_Byte string_max_len_, TY_Byte offset_, TY_Byte* data_ref_);</pre>	Copy a string from string_ref_ to data_ref_. The actual number of characters stored cannot be greater than string_max_len_. If the string contains a null, the last character saved is a null character if this does not exceed the string_max_len_ limit.
Decoding	
<pre>TY_Byte PickInt8(TY_Byte offset_, TY_Byte* data_ref_);</pre>	Return the value of the byte in the byte array buffer pointed to by data_ref_ from the position offset_.
<pre>TY_Word PickInt16(TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Return the value of the integer 16 from the byte array buffer pointed to by data_ref_ from the position offset_. Assume that the most significant byte is the first if endian is MSB_FIRST(0), which is the Hart standard.
<pre>TY_DWord PickInt24(TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Return the value of the integer 24 from the byte array buffer pointed to by data_ref_ from the position offset_. Assume that the most significant byte is the first if endian is MSB_FIRST(0), which is the Hart standard.
<pre>TY_DWord PickInt32(TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Return the value of the integer 32 from the byte array buffer pointed to by data_ref_ from the position offset_. Assume that the most significant byte is the first if endian is MSB_FIRST(0), which is the Hart standard.
<pre>TY_UInt64 PickInt64(TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Return the value of the integer 64 from the byte array buffer pointed to by data_ref_ from the position offset_. Assume that the most significant byte is the first if endian is MSB_FIRST(0), which is the Hart standard.
<pre>TY_Float PickFloat(TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Return the value of the single precision IEEE754 number from the byte array buffer pointed to by data_ref_ from the position offset_. Assume that the most significant byte is the first if endian is MSB_FIRST(0), which is the Hart standard.
<pre>TY_DFloat PickDFloat(TY_Byte offset_, TY_Byte* data_ref_, EN_Endian endian_);</pre>	Return the value of the double precision IEEE754 number from the byte array buffer pointed to by data_ref_ from the position offset_. Assume that the most significant byte is the first if endian is MSB_FIRST(0), which is the Hart standard.
<pre>void PickPackedASCII(TY_Byte* string_ref_, TY_Byte string_len_, TY_Byte offset_, TY_Byte* data_ref_);</pre>	Generate a string and copy it to the buffer pointed to by sb. The final string should have the length string_len_. The packedASCII source is a set of bytes in the byte array buffer pointed to by data_ref_, starting at index offset_. Note: The string length has to be a multiple of 4 while the number of packedASCII bytes is a multiple of 3.
<pre>void PickOctets(TY_Byte* stream_ref_, TY_Byte stream_len_, TY_Byte offset_, TY_Byte* data_ref_);</pre>	Copy a number (numOctets) of bytes from the byte array buffer pointed to by data_ref_ to the user buffer pointed to by dataDestination.
<pre>void PickString(TY_Byte* string_ref_, TY_Byte string_max_len_, TY_Byte offset_, TY_Byte* data_ref_);</pre>	The function reads a string from a buffer (data_ref_) starting at index offset_ and stores the characters in string_ref_. The string buffer is read from until a null character appears or string_max_len_ is reached. If possible, the null character is also saved.
Internal	
<pre>void FastCyclicHandler(TY_Word time_ms_);</pre>	Although this function is not accessible to the test client, it is required for the operation of the Hart protocol. The function must be called by a separate task approximately every millisecond to enable timing in the communication. The time_ms parameter indicates how many milliseconds have passed since the last call.

Embedded System Requirements

It is difficult to estimate the system requirements for targets based on different micro controllers and different development environments. The following is therefore giving a very rough scenario for the target system resources.

Item	Requirement/Size	Comment
RAM	64k	Depends very much on addressing structure of the controller and the used compiler and linker.
ROM (Flash)	100k	
Timing	2 ms Timer interrupt	2 ms is the minimum requirement, 1 ms would be much better.
	50 ms cyclic all from task level	This is needed to run the command interpreter.
I/O	UART and Hart MODEM Rx and Tx functions	Carrier detection would be helpful but is not required.
System	Simple math +-*/ memcpy() memset() memcmp()	Only a few standard library functions are required. There is no special need for multi tasking, messaging or semaphores.
	1 ms timing resolution	

Table 1: Embedded System Requirements

Windows Test Adapter

Overview

The Windows test adapter is a software developed in C#. This test adapter uses a Windows DLL in which the Hart Master is embedded. The DLL implements the HART Protocol, whose firmware was written in C++ in real time.

The connection to the DLL is defined in the BaHartMaster-7.6.cs file. Here you can find the declarations for all functions, structures and constants that are required.

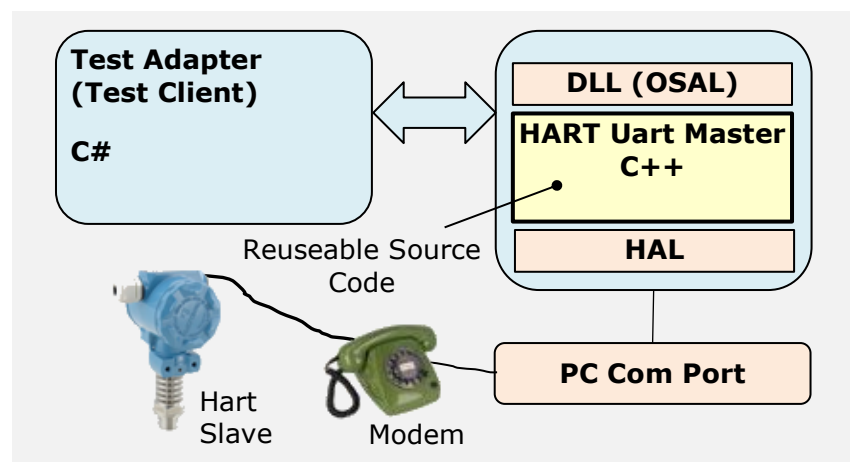


Figure 2: Architecture of the Test Environment

User Interface

The executable file for the test adapter is located at the following location:

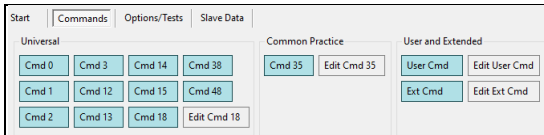
.\02-Device\03-Test\01-Windows\03-DebugBench\TestCppMaster.exe

The screenshot shows the 'Test Hart C++ Master, V7.6.1' application window. The 'Start' tab is active, displaying connection settings (COM Port: COM124, Poll Address: A_00, Baudrate: 1200), Hart Behavior (Preambles: 5, Master Role: Primary), and Hart Communication View (Frame Numbers, Address, Decoded Data, Timing, Status Details). A status bar at the bottom indicates 'COM 124 | Monitoring active | Switch record off to stop monitoring.' and a counter '0000000'.

When the executable file is started, the Simulations DLL for the master is automatically loaded. The work surface is divided into two halves. Settings are made or commands are given in the upper area, while the lower area is reserved for a monitor that shows the communication process. Some basic settings are possible in the Start tab and a connection can be established with the connected slave.

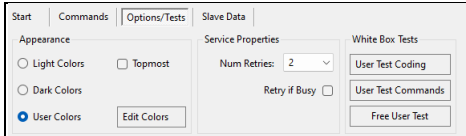
Screenshot 1: The Tab 'Start'

Technical Documentation



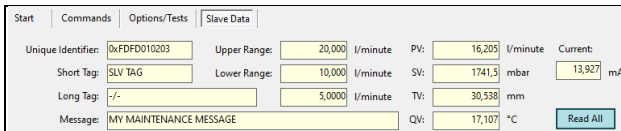
Hart commands are configured and executed in the Commands tab.

Screenshot 2: The Tab 'Commands'



The 'Options/Test' tab contains additional settings and allows the execution of simple tests. Since you have the source code, you can easily modify the tests or add new ones.

Screenshot 3: The Tab 'Options/Tests'



The 'Slave Data' tab is intended to read and display the data of a connected slave.

Screenshot 4: The Tab 'Slave Data'

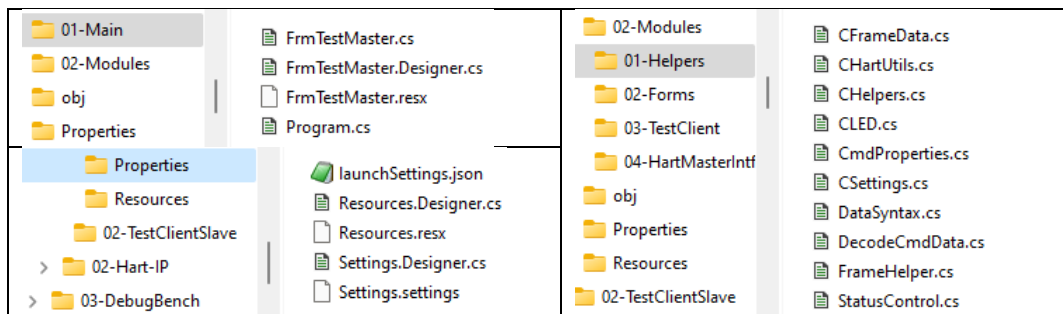
Implementation Details

The project file for the test adapter can be found in the following path:

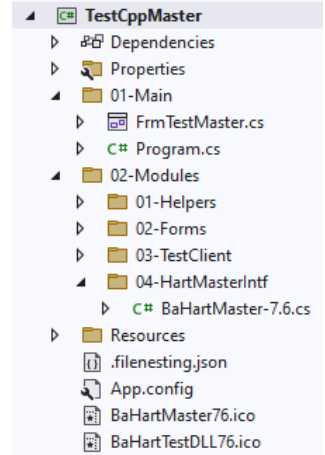
.\02-Device\03-Test\01-Windows\02-Apps\01-Hart\
01-TestClientMaster\TestCppMaster.csproj

Directory Structure

Note: The content of 02-Forms, 03-TestClient and 04-HartMasterIntf is not explicitly listed.



Project Structure

	<p>The project structure is very similar to the directory structure. Here too there is a strict distinction between generic area and specific area.</p> <p>The specific contents of the files are described in more detail in the list below.</p>
---	---

List of Files

Category	Name	Description
01-Main		
-/-	FrmTestMaster.cs	Includes the operation of the user interface and all functions necessary to coordinate the additional modules.
02-Modules		
01-Helpers	CFrameData.cs	The two most important functions of this class (CFrameData) are CatchFrame() and GetDisplayString(). CatchFrame() reads all information from a binary byte stream that can be interpreted (parsed), while GetDisplayString() formats a text from the information available, which is ultimately displayed in a control that understands RTX formatting.
	ChartUtils.cs	There are a few small functions here that read texts from numerical information in the hard protocol that indicate what the codes mean. An example of this is the engineering unit.
	Chelpers.cs	A number of small functions are implemented in the helpers that do nothing other than convert numbers into formatted text in a certain way. The functions generally have nothing to do with Hart.
	CLED.cs	The module provides the code that is needed to realize the graphical representation of an LED.
	CmdProperties.cs	The code provides texts for various elements of a command response.
	Csettings.cs	A .NET component is used to store and read the user settings. Nevertheless, the individual settings must be assigned to specific functionalities.
	DataSyntax.cs	DataSyntax is a simple construct that allows the user to define the structure of a data set. It's a bit similar to the definition of a structure in C++. Small example (data for command 18): pca6;TAG NAME;pca12;MESSAGE 16 CHARS; dec8;5;dec8;11;dec8;111
	DecodeCmdData.cs	The module provides functions to decode some commands. The input is a response as a byte array and the output consists of a string. The following commands are decoded: 0, 1, 2, 3, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 20, 21, 22, 34, 35, 38, 48, 78 and 109. Here is an example for command 0: In: FE 11 2B 05 07 01 01 08 00 9C 9F 4D 05 0B 00 94 00 00 11 00 11 01 65 Out: 254/Man17/Dev43/5 PAs/Hart7/Tx1/Sw1/Hw8/FL00000000/ID 0x9C 0x9F 0x4D/MinPArsp:5/MaxNumDVs:11/CfgChCnt:148/ExtDevStat:00000000/ManuID:0x0011/LabDistID:0011/Profile:1 65
	FrameHelpers.cs	A set of classes and methods needed to interact with the user. Mainly it's about decoding and representation.

Technical Documentation

	StatusControl.cs	This is a set of functions keeping track of configuration changes with the master test framework.
02-Forms	FrmAbout.cs	The form provides information about the current implementation. Here you can also access the documentation.
	FrmCmd18.cs	Configuration of command 18.
	FrmDataSyntax.cs	Data syntax is kind of a description language to define data sets. The form is an editor for this
	FrmExtCmd.cs	Configuration of an extended command.
	FrmSetColors.cs	Configuration of the coloring of the display.
	FrmUserCmd.cs	Configuration of user specific commands.
03-TestClient	TestClient.cs	TestClient is a very central module through which almost all communication processes are handled.
04-BaHartMasterIntf	BaHartMaster-7.6.cs	The module contains all declarations that are required to interface to the test DLL (BaHartMaster-7.6.dll).

Code Walkthrough

Establishing a Connection

N	Breakpoint	Comment
Startup		
1	HartM_UartIface.cpp, Line 29	● result = CChannel::Open(port_number_, CChannel::CType::UART);
2	Start debug	
3	Step into	
4	HartChannel.cpp, Line 56	● if (CHMMacPort::Open(m_port_number, m_baudrate, m_type) == EN_Bool::TRUE8)
5	Step into	
6	HMMacPort.cpp, Line 112	● if (CWinSys::CUart::Open((TY_Byte)port_, baudrate_) == EN_Bool::TRUE8)
7	Step into	The com port is getting initialized
8	WinSystem.cpp, Line 103	● CTask_uart_cyclic_task.Start(CUartMaster::FastCyclicHandler);
10	Step into	The start of a task (thread) is prepared
11	WinSystem.cpp, Line 319	● return CWinSys::CThread::Start(&uart_thread_control);
12	Step into	This is the start of the Windows thread
13	WinSystem.cpp, Line 232	● COSAL::CTimer::Init();
14	Step into	The central timer is initialized
15	WinSystem.cpp, Line 234	● return EN_Error::NONE;
16	HartChannel.cpp, Line 59	● return EN_Bool::TRUE8;
17	TestClient.cs, Line 125	● HartMasterDLL.BAHAMA_StartMonitor();
18	TestClient.cs, Line 133	● HartMasterDLL.BAHAMA_GetConfiguration(ref CTestClient.Configuration);
19	Continue	Startup completed
Establishing a connection		
Select tab 'Commands', connect a Hart slave		
1	Hart_UartIface.cpp, Line 90	● p_service = CChannel::GetServicePtr(h_service);
2	Start debug	
3	Click Cmd 0	The service will be initialized
4	Hart_UartIface.cpp, Line 104	● CChannel::SetServiceOwner(h_service, EN_Owner::PROTOCOL);
5		Finally the service is passed to the protocol handler.
6	HartLayer2.cpp, Line 642	● m_active_CService = CChannel::GetServicePtr(CChannel::GetRequestedService());
7		The service is picked by the protocol handler when it is in the state WATCHING
8	HartLayer2.cpp, Line 674	● *to_do_ = CHMUartProt::EN_ToDo::START_TRANSMIT;
9		This is starting the transmission of the request
10	HMUartProtocol.cpp, Line 152	● to_do = CHartSM::EventHandler(CHartSM::EN_Event::TX_DONE, NULL);
11		This is reached, when the sending of the request was completed. Now the receiver is enabled.
To get to the point where the answer was received you have to restart completely because the connected slave is not stopping when your machine is halted at a break. Therefore stop debugging		
12	HMUartProtocol.cpp, Line 73	● to_do = CHartSM::EventHandler(CHartSM::EN_Event::RX_COMPLETED_RSP, &m_response_frame);
13	Start debugging	
14	Click 'Cmd 0'	The breakpoint is indicating the successful reception of the response.
15	HartLayer2.cpp, Line 720	● CChannel::FireServiceEvent(CChannel::CServiceEvent::CONFIRMATION, m_active_CService->GetHandle(), 0);
16	HartChannel.cpp, Line 322	● GetServicePtr(handle_)->SetOwner(EN_Owner::USER);
17		After this call the user can access the service again.
18	HartChannel.cpp, Line 146	● EN_Bool CChannel::IsServiceCompleted(SRV_Handle handle_)
	HartChannel.cpp, Line 154	● return EN_Bool::TRUE8;

Executing a Command

Executing a command is very similar to executing a connection request. Therefore, in the following table I have limited myself to keywords and comments on interesting places in the code.

You should probably know something more about the implementation of state machines. The status machine is fed with events and, if necessary, data and carries out status changes if necessary. It also returns a to do variable to the caller, which specifies what action should take place next.

N	Breakpoint	Comment
1		Each command is mapped to the function LaunchComand() in HartM_UartIface.cpp. <code>SRV_Handle CUartMaster::LaunchCommand(TY_Byte command_, EN_Wait qos_, TY_Byte* data_ref_, TY_Byte data_len_, TY_Byte* bytes_of_unique_id_)</code>
2		After the service is prepared, it call it's own Launch method which is encoding the request frame.
3	Start debug and click Cmd 0	
4	HartService.cpp, Line 77	● <code>CCoding::EncodeFrame(&m_request);</code>
5	Click Cmd 18	
6	HartM_UartIface.cpp, Line 267	● <code>CChannel::SetServiceOwner(h_service, EN_Owner::PROTOCOL);</code>
7		After the request frame is encoded, the service is passed to the Hart protocol machine, which runs in its own task.
8	HartLayer2.cpp, Line 642	● <code>m_active_CService = CChannel::GetServicePtr(CChannel::GetRequestedService());</code>
9		As already shown in the connection, the service is taken up by the protocol machine on the other side.
10	The request is then sent and the response is expected. In this context, the question might arise as to where the incoming data is decoded. This happens in the so-called parser, which processes the frame.	
	HartFrame.cpp, Line 59	● <code>EN_Bool CFrame::TryParse(TY_Word* bytes_parsed_, TY_Byte* new_data_, TY_Byte* new_err_, TY_Word new_data_len_, EN_Bool gap_time_out_)</code>
		The next question might be: and where does the user data end up. The data is also known as a payload and is placed in a buffer at the following location.
	HartFrame.cpp, Line 111	● <code>m_status = GetPayload(new_data_[bytes_parsed], new_err_[bytes_parsed]); .. and further</code>
	HartFrame.cpp, Line 354	<code>CFrame::EN_Status CFrame::GettingPayload(TY_Byte data_, TY_Byte error_) { m_target_chk ^= data_; m_payload_data[m_payload_count] = data_; m_payload_count++;</code>

Additional Information

Type Definitions

Types	Enums
<pre>typedef unsigned char TY_Byte; typedef unsigned short TY_Word; typedef unsigned int TY_DWord; typedef int TY_Int32; typedef unsigned long long TY_UInt64; typedef float TY_Float; typedef double TY_DFloat; typedef TY_Word WRD_Handle; typedef TY_Word SRV_Handle;</pre>	<pre>enum class EN_SRV_Result : TY_Byte { EMPTY = 0, NO_DEV_RESP = 1, COMM_ERR = 2, INVALID_HANDLE = 3, IN_PROGRESS = 4, SUCCESSFUL = 5, RESOURCE_ERROR = 6, TOO_FEW_DATA_BYTES = 7, OBSOLETE = 8 }; enum class EN_Endian : TY_Byte { MSB_First = 0, // Big endian (Hart standard) LSB_First = 1 // Little endian }; enum class EN_Bool : TY_Byte { FALSE8 = 0, TRUE8 = 1 }; enum class EN_Bit : TY_Byte { CLEAR8 = 0, SET8 = 1 }; enum class EN_Error : TY_Byte { NONE = 0, ERR = 1 };</pre>
<p>I readily admit that the typedefinitions I use are not as precise as the original ones. But for me the code is more readable.</p>	
<p>For the enums, I consciously chose enum classes because they are the easiest way to associate the values of the enums with an integer type.</p>	
<pre>enum class EN_Wait : TY_Byte { NO_WAIT = 0, WAIT = 1 }; enum class EN_CommType : TY_Byte { NONE = 0, UART = 1, TCP_IP = 2 };</pre>	

Coding Conventions

Regarding this issue, I have only defined a format that makes the scope of a label clearer. It's just to make the code easier to read. This simple type of coding convention can be used in both C++ and C#.

Pascal case			
local_variable	function_param_	m_member_var	mo_member_object
Variable with local scope.	A function parameter has a trailing underscore.	Basic type private member variable	Complex object member
s_member_var			
Basic type static private member variable			
Camel case			
PublicVariable	PublicObject	AnyMethod	
Variable with public scope.	Object with public scope.	No difference between public and private.	

Appendix

Internet Links

Specification Documents	
HART Specifications	FieldComm Group
MODEMs	
RS 232 Modem	Microflex
USB Modem	Endress + Hauser
Viator USB Modem	Pepperl+Fuchs
Ethernet-APL	
Advanced Physical Layer	FieldComm Group
Ethernet - To the Field	Ethernet APL Organisation
HART-IP Developer Kit	FieldComm Group

Abbreviations

Abbreviation	Description
HCF	Hart Communication Foundation Integrated in FieldComm Group
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
HART-IP	Hart via Internet Protocol
HART APL	Hart Advanced Physical Layer
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

Download Location

The software package described in this document can be downloaded via the following link:

<https://github.com/BorstAutomation/Hart-Master-Slave-8.0E.git>

Legal Issues

Conformity

This software package was developed to the best of my knowledge and my belief. The basis is the specifications of the Hart Communication Foundation in version 7.9.

However, it cannot be guaranteed that the software included in this package meets the HCF specifications in all required respects.

It is only possible to prove the conformity of this software after the user has integrated the software into his device and commissions HCF or a certified company to carry out this test. Under no circumstances am I, Walter Borst, responsible for carrying out such tests. Nor am I responsible for correcting any deficiencies resulting from such a test.

Copyright

Copyright, Walter Borst, 2006-2024

Kapitaen-Alexander-Strasse 39, 27472 Cuxhaven, GERMANY

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

E-Mail: info@borst-automation.de

Home: <https://www.borst-automation.de/>

Warranty Disclaimer

This software/firmware is supplied with NO WARRANTIES. Walter Borst expressly disclaims any warranty for the software package. This software package and related documents are provided "AS IS"; without warranty of any kind, expressed or implied. This includes implied warranties of fitness for a particular purpose. All risk arising out of use of this package remains with the user. By using this software package, the user agrees that no event shall Borst Automation or Walter Borst make responsible or liable for damages whatsoever. This includes, without limitation, damages for loss of business profits, loss due to business interruption, loss of business information, or any other pecuniary loss, arising out of the use of or the inability to use this software package.