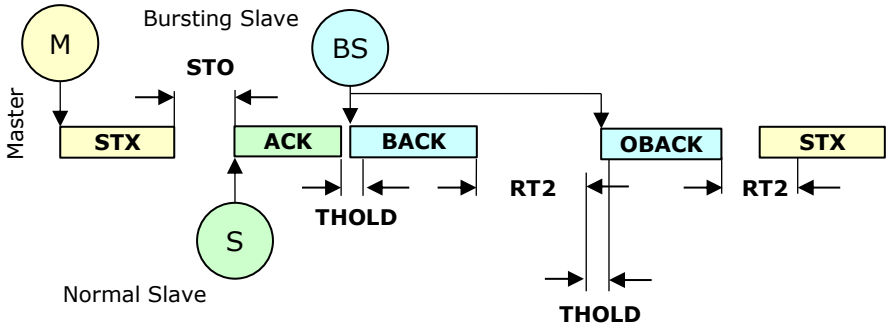
HartTools

HartSlave 7.5

Software/Firmware Documentation

Revision: 7.5.0
Date: 18.11.2017

Intelligent Firmware Solutions
for very small Systems



Borst Automation
Neue Reihe 33
DE-27472 Cuxhaven
GERMANY

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

http://borst-automation.de

info@borst-automation.de

Contents

Overview

Introduction

The introduction of microprocessors in the beginning of the 1980s changed the environment of the developers of devices for process control for the first time. Microcontrollers like the 8031 family at these times already had serial interfaces and soon the engineers started to use them for debugging and production purposes. Later the serial interface was introduced as digital communication to offer new functionality to the users of the devices.

"The HART Protocol was developed in the mid-1980s by Rosemount Inc. for use with a range of smart measuring instruments. Originally proprietary, the protocol was soon published for free use by anyone, and in 1990 the HART User Group was formed. In 1993, the registered trademark and all rights in the protocol were transferred to the HART Communication Foundation (HCF). The protocol remains open and free for all to use without royalties.", is the text published at the home page of the HCF (Hart Communication Foundation).

Today HART is the leading communication technology for field devices in process automation.

More details are available at the HCF:

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

The first slave software of Borst Automation was published in 1998 targeted to Hart Version 5.x. At this time a slave implementation was very easy to implement because the support of sending burst messages were not required at all. The support was optional and nobody was realizing this feature.

☞ Burst mode needed.

Today the support of burst messages is implemented in most of the field devices and HART 7.x is the version which is required, providing more services and functionality as the former Hart slave implementations.

That is the reason why Borst Automation developed a new slave software based on the needs of modern HART based field devices.

☞ Smart protocol management.

Hart Slave 7.5 is a solution which is supporting all Hart version since 5.3 by a smart Network management which is automatically adapting to the capabilities of the requesting Host device.

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.

☞ C++ versus C.

Most of the platforms are programmed in C++ but not all of them. Sometimes it is also desired by the developers to use standard C for embedded implementations. Therefore the most parts of the software are written in standard C language which can be easily be integrated into a C++ environment.

☞ 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 Slave was done as carefully as possible regarding the amount of flash memory and RAM.

☞ A low cpu clock makes it even more time critical.

HART devices are typically powered by two wires with a minimum current of 3.8 mA and a terminal voltage of about 12 Volts. This power is needed not only for the microcontroller but also for the measurements circuits and all the other required electronics. Thus the CPU clock is reduced to a frequency as low as possible. This makes the application even more time critical.

☞ The user needs source code.

The Hart Protocol of a slave requires a strict timing specially for burst mode support. 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 Slave 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 slave 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

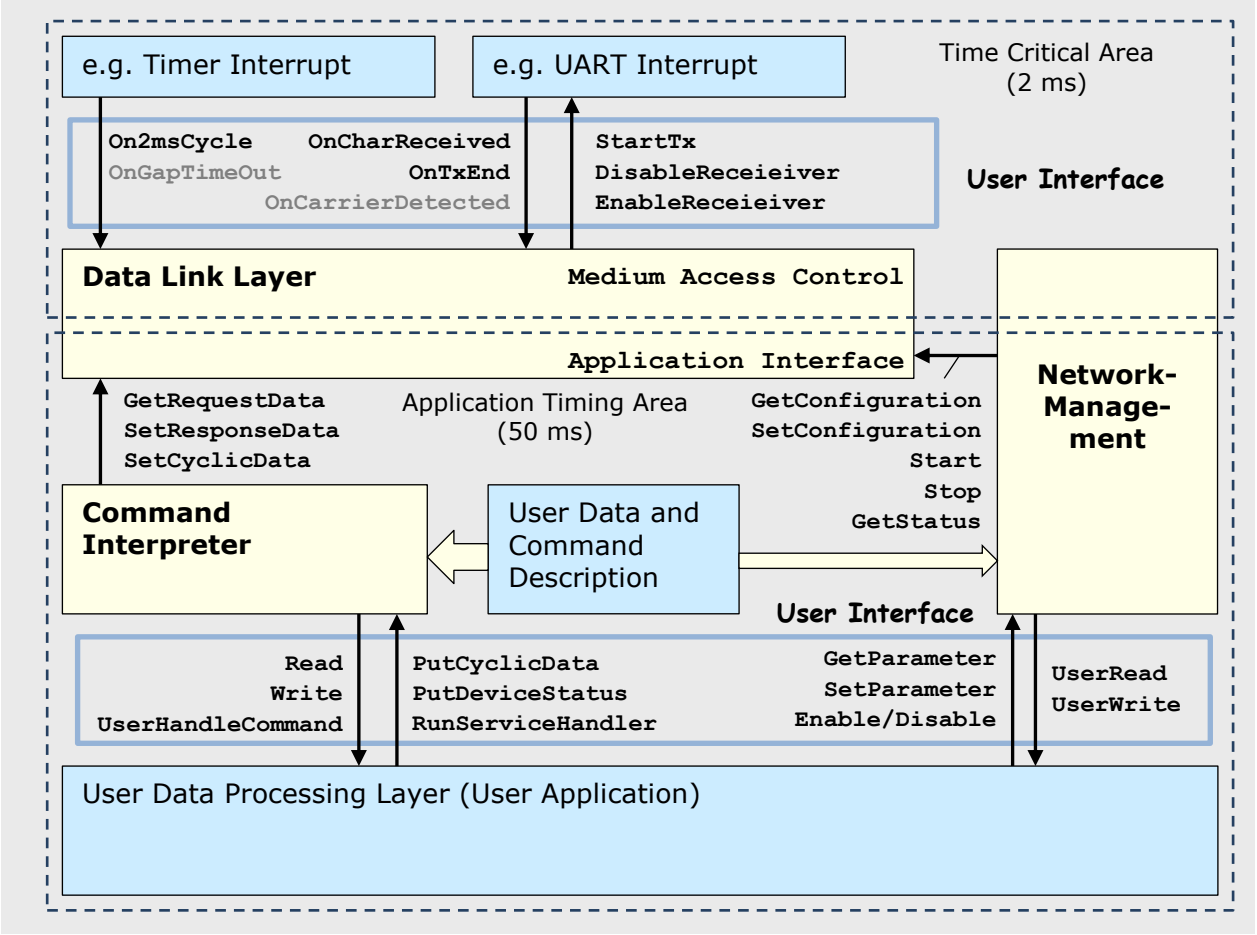


Figure 1: Structure of the HART Slave Software

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 were the application software is working, which is far less time critical.

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

The figure above is clearly showing also two user interfaces. There is a user interface which is connecting the Hart Slave 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 communication independent.

The interface to the user's application is located on top of the User Data Processing Layer (User Application). Functions with names starting with 'User' are function which are expected by the communication stack to be provided by the user. Another set of functions are called by the user's software on demand. There is no restriction when these functions may be called. The functions for the user are neutral and does not show that they are used for HART communications.

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 SetParameter.

In the above figure the parts of the HartSlave 7.5 are shown in yellow color while the user parts are marked with blue. A major part is the block called User Data and Command Description. This block is binding the user data parts to the Hart commands and or function provided by the user. This is finally a set of tables stored in the flash memory of the device.

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

The figure is also showing a set of functions between the command interpreter 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 interpreter and network management.

Timing Solutions (Examples)

Usually a Hart slave is a very simple device. It waits for a request and has 256 ms of time to respond. That's quite easy to implement.

The problems are starting if the device has to support the burst mode for sending cyclic data. A device which is in burst mode has to synchronize to the dual master time controlled Hart protocol.

Sending a Burst Frame in Time

When the bursting device is the only one in the network (no master present) it is the token holder at an time. However, to allow eventual masters appearing in the network to synchronize to the token passing the distance of sending bursts is strictly regulated.

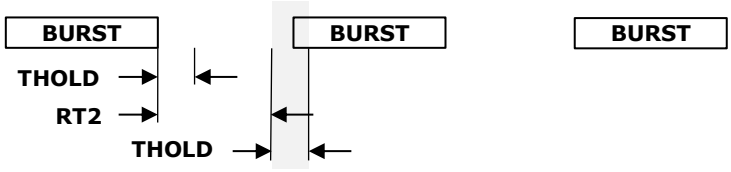


Figure 2: The Burst Messages are Part of Token Passing

After a burst message the token is passed to the master the burst was addressed to. If this master is present it may send a message within the token hold time. After this time has passed the other master holds the token and is allowed to send a message. If that doesn't happen the token goes back to the bursting slave allowing to send the next burst primitive.

Appending a Burst to a Response

If a device is in burst mode and is sending a response to a requesting master the token to the other (possible) master is only passed via a burst message. Therefore a bursting device has to append the burst directly to the response even if it is the response to a data link layer command which is enabling the burst mode (e.g. command 109).

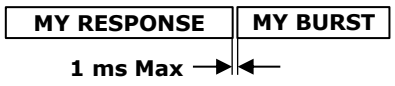


Figure 3: Sending Bursts Right Away

☞ The real time requirements are hidden from the user.

This is implemented in the stack of the data link layer. Network management is providing a default burst frame with real or default values at any time it is required by the data link layer.

Synchronizing Bursts to the Masters

Page 39 in [DLLSpec] is clearly showing the token passing mechanism of the HART protocol. The diagram shows that any acknowledgement of any other (none bursting) slave device is passing the token to the bursting device. Therefore the bursting device has to watch the bus traffic carefully and has to append the burst message in the right time window to pass the token to the appropriate master.

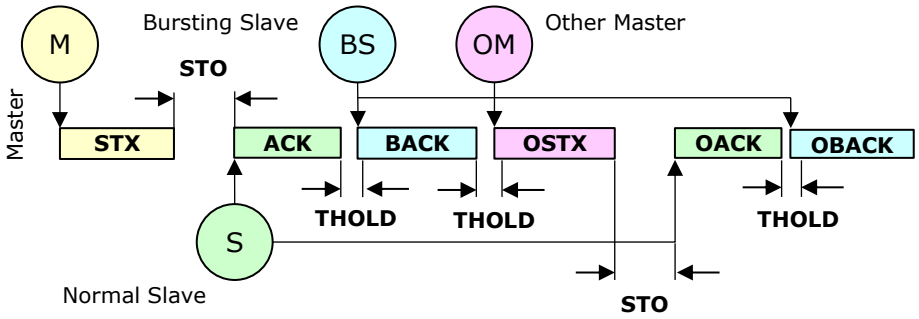


Figure 4: Timing with two active Masters

Figure 4 is showing the relevant timing if two masters are active. The task of a bursting slave is to send the burst messages within the token hold time after a response by the other slave was received.

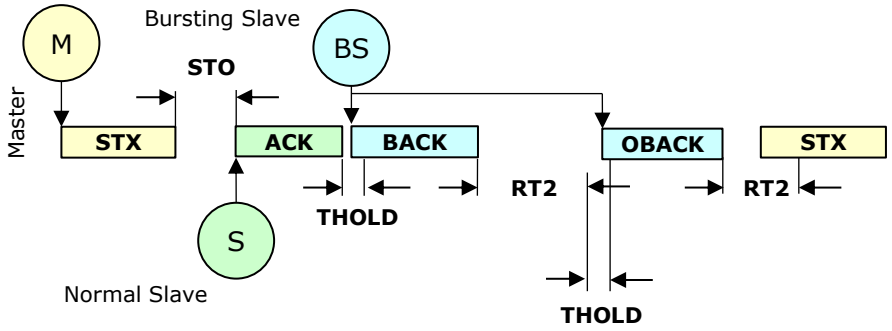


Figure 5: Timing with one active Master

Figure 5 shows that the gap for the other is kept open even if only one master is active.

Handling of Errors in a Bursting Slave

The handling of receiver errors in a slave is a complex part of the HART protocol and can be understood much better when studying the applied tests which are published by the HCF.

Here only one example is given to show one of the various details in special situations.

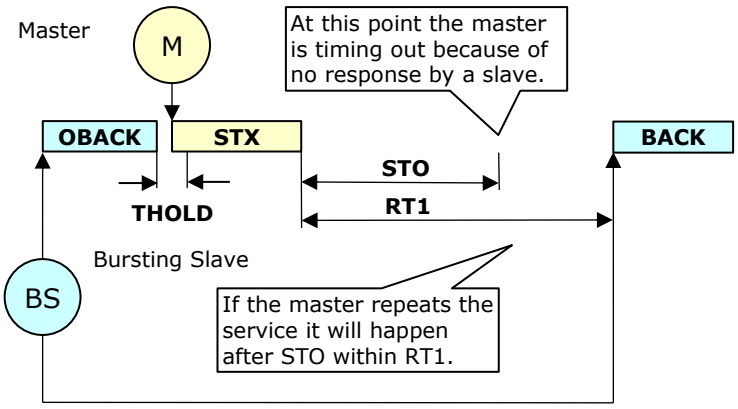
Use Case

A master and a bursting slave are connected at the communications.

The master starts to search for other slave devices by sending a command 0 stx (request) to devices at the desired addresses. In this case not existing devices will not respond.

Note: The master has to synchronize to the burst messages of the bursting slave.

Bursting Slave Behavior



☞ Note: A bursting slave is a token holder like a master.

Figure 6: Bursting Slave Device Watching STO

Test DLL029 in [DLLTest] is defined as a test for the above mentioned use case. The diagram in Figure 6 makes clear what is happening if a master is getting no response from another slave device.

To keep synchronization of the network the bursting slave has to recognize stx frames addressed to other slaves and has to start it's timer for watching RT1 after the request was send. When STO elapses the master may or may not repeat the request.

If the master is repeating the service the bursting slave has to start RT1 anew.

If the master is not sending any further request within RT1 the bursting slave has to send the next burst message.

Functions Overview

Category	Name	Description
Data Link Layer		
DLL Initialization	**Init**	Initializes the protocol layer. This has to be the first call into the Hart Slave Data Link Layer.
DLL Operation	**GetConfiguration**	Returns the configuration from the protocol layer.
	SetConfiguration	Sets the configuration for the protocol layer.
	Start	Starts the Data Link Layer protocol, enables the receiver.
	Stop	Stops the Data Link Layer protocol. disables the receiver.
	GetStatus	Return the status of the Data Link Layer.
DLL MAC Access User Interface	**On2msCycle**	This routine has to be called on every two milliseconds. It run the HART protocol including mechanism for sending automated responses and the handling of burst frames. May be called by interrupt.
	OnCharReceived	Has to be called if a character was received. May be called from interrupt.
	OnTxEnd	Has to be called if transmitting octets is completed. This call has to be done after the stop bit of the last byte of a stream was sent. The function may be called from interrupt.
	StartTx	A request to the user part of the software for starting the transmission of a byte stream. This function is called from inside On2msCycle.
	DisableReceiver	A request to the user part of the software for switching off the receiver. This function is called from inside On2msCycle.
	EnableReceiver	A request to the user part of the software for switching on the receiver. This function is called from inside On2msCycle.
	OnGapTimeout	This function call is optional. If the user has enough timer resources he may implement his own more precise Gap Time Out. This would allow more jitter on the 2 ms cycle.
	OnCarrierDetected	This function call is optional. If the user's hardware is in able to detect the carrier this can be used as a better indication for frame start and frame end than the gap time out.
DLL APP Iface	**GetRequestData**	Returns the latest incoming request (usually a Hart command). This function is called if the user software has time.
	SetResponseData	Called from the user software (the command interpreter) to place the response to a command.
	SetCyclicData	Called from the user software. Used to set cyclic data for the burst frames. The Data Link Layer is managing to send the data if the protocol is allowing or requiring it.

Table 1: List of Data Link Layer Functions

User Data Processing Layer (User Application)		
Command Interpreter	**Read**	This function is called from the HartSlave software to read a parameter which is described in the User Data and Command Description.
	Write	This function is called from the HartSlave software to write a parameter which is described in the User Data and Command Description.
	HandleCommand	This function is called from the HartSlave if a command was received, which could not be handled by the HartSlave command interpreter.
	PutCyclicData	The user layer calls this function to update the cyclic data (e.g. measurement values).
	PutDeviceStatus	The user layer calls this function whenever the device status has changed.
	RunServiceHandler	This function is used to execute the service handler from application level. It has to be called in a cycle of about 50 ms .. 200 ms.
Network Management	**Read**	See description above.
	Write	See description above.
	SetParameter	This functions is provided for the user layer to provide settings like slave address or tag name.
	GetParameter	This function is provided for the user layer to get settings.
	Enable	Enables or disables the Hart Protocol in the device.
	Init	Initializes the slave stack including the command interpreter.

Table 2: List of User Layer Functions

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	**3k**	Depends very much on addressing structure of the controller and the used compiler and linker.
ROM (Flash)	**10k**	
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 3: Embedded System Requirements

Getting Started

The Project

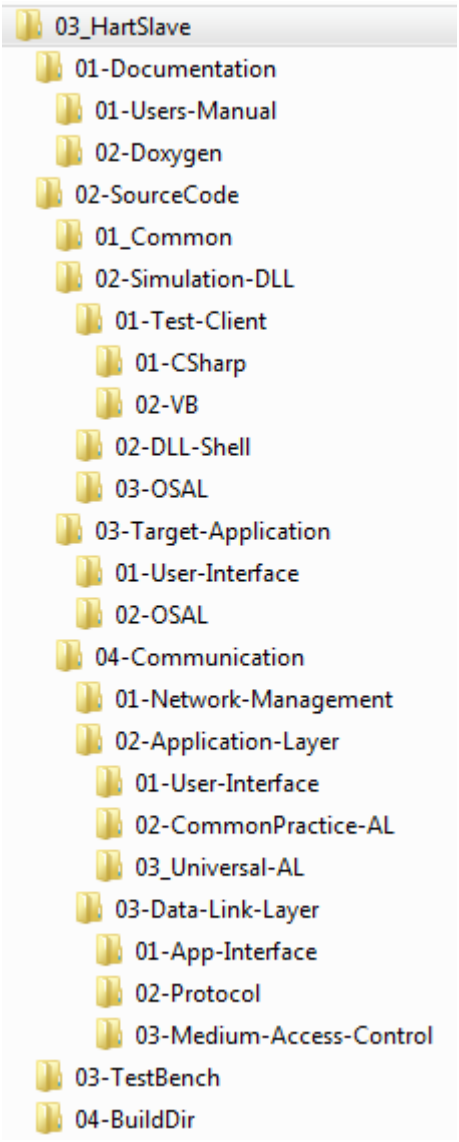


Figure 7: The Directory Structure

As shown in the graphic on the left there are four main areas in the project.

The documentation directory is containing the user's manual and a doxygen project.

The second holds all source codes. This implies a simulation shell for a Windows DLL as well as two test clients, one for C# and another one for Visual Basic. The Target-Application directory is empty as it is received for some modules of the final target microcontroller.

The TestBench is reserved for the PC-Simulation and is the place were all executable output files and DLLs are copied to.

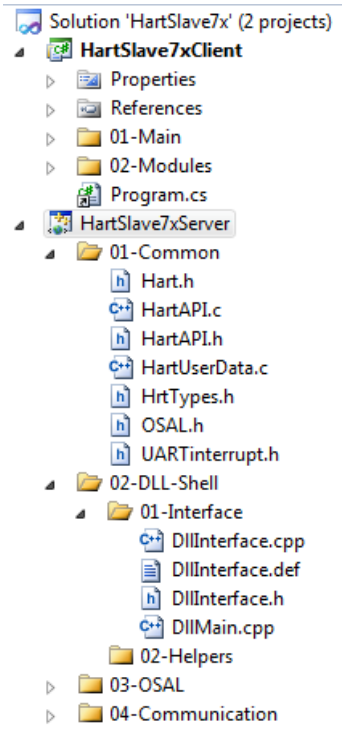
The BuildDir is only used for the intermediate files which are generated by the compilers.

Two directories are named OSAL which stands for Operating System Abstraction Layer. However, this is pretty much the same as HAL for Hardware Abstraction Layer.

The Application Layer in the Communication Section contains the command interpreter for Common Practice Commands and Universal Commands. User specific commands are realized by the user implementation and are supported through the User-Interface of the Application Layer.

The Data Link Layer is also split into the parts. One is the interface to the Application Layer. Next is the protocol implementation. And finally there is the so called Medium-Access-Control

The PC-Simulation



The Visual Studio solution of the PC-Simulation consists of two projects. One project is the client used to drive the DLL which is realized through the other project building a dynamic link library of the device part.

The client is programmed using C# as development environment which allows very quick prototyping.

The Hart slave is embedded in a DLL. The shell for this purpose is realized by the modules in 02-DLL-Shell

Most of the sources used in the HartSlave7xServer are the same sources which have to be used by the target system.

The filter OSAL contains the specific parts for adapting to the Windows operating system.

However, the main part of the implementation regarding the run time environment is based on the header files which are listed under 01-Common. The headers here are generic and used by the PC-Simulation (the OSAL part) as well as by the final target implementation which has to be provided by the application software developer of the device.

Interrupts and tasks of the operating system environment are simulated by threads in the Windows environment.

Here is one example for the receiver interrupt. The implementation in UARTinterrupt.c provides the following code for the interrupt processing routine.

```
HTY_void g_UARTint_Interrupt(HTY_void)
{
  if ( ( m_u8_Event & HICTX_RX_CHAR ) != 0 )
  {
    if ( m_bRecvEnabled == HCO_true )
    {
      HartOnCharReceived(m_u8_RxReg, m_u8_ErrReg);
    }
    g_UARTint_ClearContext();
  }
```

Code Snippet 1: The Interrupt in the PC-Simulation

The interrupt directly calls the standard routine of the Hart salve implementation (HartOnCharReceived) as it would be also implemented in the target.

But in the PC-Simulation a real interrupt is not available this is implemented in an extra thread which runs cyclically to poll the UART interface of the computer.

Thus the call of the interrupt is simulated but in the behavior very close to the target. The thread responsible for that looks like it follows.

```
HTY_ulong __stdcall CComPort::UARTProc(HTY_void * pvData)
{
  OSAL_EnvSpecific_BeginTimeResolution(1);
  while ( m_u16_ThreadStatus == CThreadStatus::U16_Running )
  {
    if ( m_u16_PortStatus == CPortStatus::U16_Open )
    {
      if ( m_b_TransmitterActive )
      {
        HandleTx();
      }
      else
      {
        HandleRx();
      }
    }
    OSAL_Sleep(1);
  }
  OSAL_EnvSpecific_EndTimeResolution(1);
  return 1234;
}
```

Code Snippet 2: The thread for watching the com port

The thread which is shown in the above code snippet runs with high priority in a cycle of 1 ms. This is not as fast as the interrupt of the final target but fast enough for the Hart protocol which works only with 1200 bit/s. The routine HandleRx is finally calling the interrupt handler.

```
HTY_void CComPort::HandleRx(HTY_void)
{
  //...
  if (dwLength > 0)
  {
      HTY_u16 e;

      fReadStat = ReadFile(m_hActivePort,
                           au8_Data,
                           dwLength,
                          &dwLength,
                         &m_os_Read);
      if (!fReadStat)
      {
          // I/O error ???
          return;
      }
      for ( e = 0; e < dwLength; e++ )
      {
        //Simulate receiver interrupt
        g_UARTint_ClearContext();
        g_UARTint_SetContext(      au8_Data[e],
                                     u8_Err,
                             HICTX_RX_CHAR);
        g_UARTint_Interrupt();
      }
  }
}
```

Code Snippet 3: The thread for watching the com port

This is only one example to understand the philosophy behind the simulation implementation.

The DLL is exposing a set of functions which are used to simulate the behavior of the original device.

Functions Exported by the DLL

Category	Declaration	Description
Control	`int32 BASLV_Init ( iComPort, ucSlaveAddr )`	Initialization of the Hart slave. Com port 1..254, address 0..32.
	`int32 BASLV_Terminate()`	Has to be called to finish the simulation and release all Windows resources.
	`int32 BASLV_GetStatus()`	Returns the status of the simulation DLL
Operation	`int32 BASLV_BASLV_SetParameter ( usParamIdx, pvValue )`	Calls network management to set parameters like tag name, device id etc..
	`int32 BASLV_GetParameter ( usParamIdx, pvValue )`	Calls network management to get parameter values like for tag name, device id etc..
	`int32 BASLV_Enable()`	Enables the Hart communication.
	`int32 BASLV_Disable()`	Disables the Hart communication.
	`int32 BASLV_PutCyclicData ( pvData, ucLen)`	Delivers cyclic data to the Hart communications
	`int32 BASLV_PutDeviceStatus ( ucStatus )`	Delivers the device status to the Hart communications
	`int32 BASLV_RunServiceHandler()`	This function triggers the execution of the hart command interpreter in the Hart Slave.
Data Access	`int32_BASLV_RegUserRead ( pfncUserRead )`	Registers a pointer to the user read function
	`int32_BASLV_RegUserWrite ( pfncUserWrite )`	Registers a pointer to the user write function
	`int32_BASLV_RegUserCmdHandler ( pfncUserCmdHandler )`	Registers a pointer for the user cmd handler function.

Table 4: List of Exported Functions of the DLL

The above listed functions are directly mapped to the functions provided by the module HartAPI.c which is the main interface between the user software and the Hart slave.

In the client software these functions are directly loaded and called by the C# application.

```
[UnmanagedFunctionPointer(CallingConvention.StdCall)]
private delegate int                    delSLV_Init(int iComPort, byte bySlaveAddr);
```

Code Snippet 4: Delegate Declaration for BASLV_Init

```
hDLL = LoadLibrary(sDLLname);
if(hDLL != IntPtr.Zero)
{
  bResult = true;
  if ( bLoadFunction( ref bResult, ref pAddrSLV_Init, "BASLV_Init", lstFunctions) )
  {
    SLV_Init = (delSLV_Init)Marshal.GetDelegateForFunctionPointer(pAddrSLV_Init,
                                                                  typeof(delSLV_Init));
  }
```

Code Snippet 5: Loading the Function Reference

Delegates and pointers are used to store the references to the functions of the DLL.

Integration into the Target

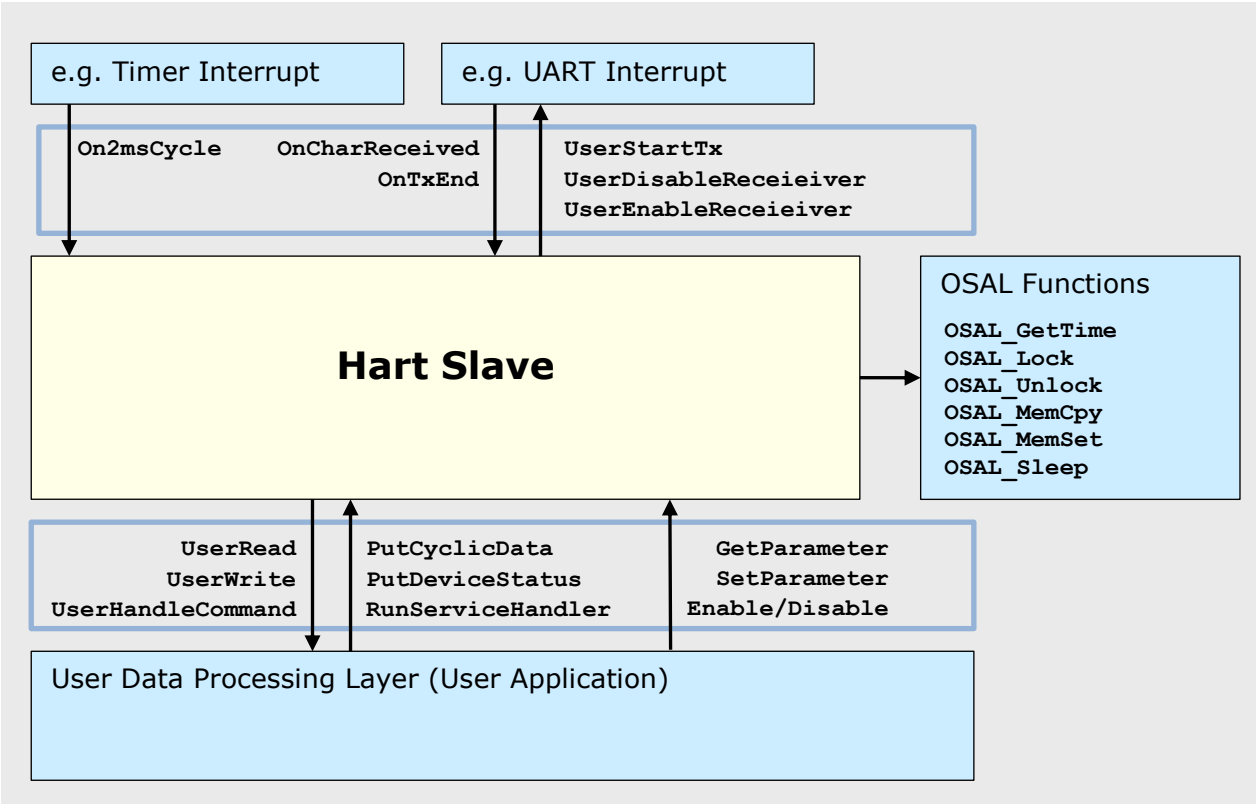


Figure 8: Required Interface Functions

The figure above is illustrating how simple the integration into the target environment can be achieved. Only a few functions have to be implemented by the user and another few set of functions have to be called.

Called by the User	Called by the Hart Slave	
Hart Slave	USER	OSAL

`On2msCycle`
`OnCharReceived`
`OnTxEnd`
`PutCyclicData`
`PutDeviceStatus`
`RunServiceHandler`
`GetParameter`
`SetParameter`
`Enable`
`Disable`
	`StartTx`
	`DisableReceiver`
	`EnableReceiver`
	`Read`
	`Write`
	`HandleCommand`
		`GetTime`
		`Lock`
		`Unlock`
		`MemCpy`
		`Sleep`
		`HartChanInit`
		`HartChanEnable`
		`HartChanDisable`

Table 5: Functions sorted by Usage

Preprocessor Definitions

Definition	Description
`BAHRT_LSB_FIRST`	The least significant byte is stored first in memory (little endian). Note: HART is coding MSB first, which is the default of the Hart Slave software.
`BAHRT_CD_SUPPORTED`	Use carrier for end of frame detection.

Functions

This section defines the functions provided in the Hart Slave software as they are implemented. Therefore the full function names are used including the module identifier prefix.

OSAL (Operating System Abstraction Layer)

A typical target embedded system for measurement applications is too small to distinguish strictly between the hardware and the operating system abstraction layer. Therefore the OSAL is servicing all the needs required for the implementation of the HART slave.

The OSAL functions have to be provided by the user developer.

Header: OSAL.h

Declaration	Description
Hardware Abstraction	
`void OSAL_HartChanInit(void);`	Initialize the Hart channel at startup.
`void OSAL_HartChanEnable (void);`	Enables UART communications.
`void OSAL_HartChanDisable(void);`	Disable UART communications.
General	
`void OSAL_Sleep(u16 u16_ms);`	Allow other tasks to be executed for the next u16_ms milliseconds.
`void OSAL_Lock(void);`	Do not allow other tasks to be executed after this call.
`void OSAL_Unlock(void);`	Allow other tasks with higher priority to be executed again after this call. Note: a lock must be followed by an unlock.
`u32 OSAL_GetTime(void);`	Returns the system time as milliseconds. The system time returned is running from 0 to 0xffffffff and scrolling over to 0.
`void OSAL_MemCpy ( u8* pu8_Dst, const u8* pu8_Src, u16 u16_Size);`	Copy a byte stream from pu8_Src to pu8_Dst. The number of bytes is given in u16_Size.
`void OSAL_MemSet ( u8* pu8_Dst, u8 u8_Val, u16 u16_Size);`	Set an array of bytes pointed to by pu8_Dst to the value u8_Val. The number of bytes is given in u16_Size.

Data Link Layer (DLL)

Initialization

Header: HartDL.h

Declaration	Description
`void HartDL_Init(void);`	Initializes the HART communication stack.

Operation

Header: HartDL.h

Declaration	Description	
`void HartDL_GetConfiguration (HTY_ST_DL_Config* pst_Config);`	Gets a copy of the configuration information from the data link layer (e.g. baud rate, num preambles).	
	pst_Config	Pointer to a buffer to copy the configuration to.
`u8 HartDL_SetConfiguration (HTY_ST_DL_Config* pst_Config);`	Sets the configuration of the data link layer. Note: Configuration only accepted if the protocol is not running.	
	pst_Config	Pointer to a buffer to copy the configuration from.
	Returns	HRT_ERR_NONE HRT_ERR_ACC_DENIED
`void HartDL_Start(void);`	Starts the communication.	
`void HartDL_Stop(void);`	Shut down communication.	
`u8 HartDL_GetStatus(void);`	Gets the status of the data link layer.	
	Returns	HRT_DL_STAT_IDLE HRT_DL_STAT_STARTING HRT_DL_STAT_RUNNING

MAC (Medium Access Control)

Header: HartDL.h, HartMA.h

Declaration	Description	
Called from Medium Access Provider		
`void HartMA_OnCharReceived (u8 u8_Char, u8 u8_Err);`	The function is typically called from the interrupt if a character was received and/or an error was detected. u8_Char: received character u8_Err: error flags as defined in HartAPI.h	
`void HartMA_On2msCycle(void);`	This function is typically called from the timer interrupt which triggers the 2 ms cycle. The function is controlling the timing of the Hart protocol.	
`void HartMA_OnTxEnd(void);`	The function has to be called if all octets had been sent and if the carrier was switched off. This call happens in the Tx part of the UART interrupt.	
`void HartMA_OnGapTimeOut(void);`	Optional function to be called if gap time detection is supported by medium access.	
`void HartMA_OnCarrChangeDet (u8 u8_State)`	Optional function to be called if a change of the carrier was detected.	
	u8_State	HRT_CARR_ON HRT_CARR_OFF
Called from Medium Access User		
`u8 HartMA_StartTx ( const u8* pu8_Data, u8 u8_Size);`	Starts the transmission of a frame. In case that a frame is already in transmission, an error is returned. The medium access provider has to prepare a local copy of the stream to be sent because the pointer pu8_Data becomes invalid after the call.	
	pu8_Data	Pointer to the stream to be sent.
	u8_Size	Size of the stream to be sent.
	Returns	HRT_ERR_NONE HRT_ERR_TX_REJECTED
`void HartMA_DisableReceiver(void)`	Disables the receiver.	
`void HartMA_EnableReceiver(void)`	Enables the receiver.	

Application Interface

Header: HartDL.h

Declaration	Description	
`b8 HartDL_GetRequestData (HRT_ST_DL_ReqRsp* pst_Req);`	Fills the buffer pointed to by pst_Req if any request was received.	
	Returns	HRT_B_FALSE HRT_B_TRUE
`void HartDL_SetReponseData (HRT_ST_DL_ReqRsp* pst_Rsp);`	Hands over the response to a previously received request.	
`void HartDL_SetCyclicData (HRT_ST_DL_Cyclic* pst_Cyc);`	Hands over the cyclic data (for bursts) to the data link layer. The data link layer is processing the bursts without any request to the application.	

User Data Processing Layer

Command Interpreter

Header: HartUL.h, HartCI.h

Declaration	Description	
Called from the Command Interpreter		
`u8 HartUL_Read ( u16 u16_ParIdx, u8* pu8_Val, u8* pu8_Size);`	Reads the value of a parameter identified by u16_ParIdx.	
	u16_ParIdx	Index into the parameter description table which is provided by the user.
	pu8_Val	Pointer to a buffer to write the value to.
	pu8_Size	Pointer to an 8 bit value. When called it contains the size of the available buffer, after return it contains the number of read bytes.
	Returns	HRT_ERR_NONE HRT_ERR_ID_ERROR HRT_ERR_SIZE_MISMATCH
`u8 HartUL_Write ( u16 u16_ParIdx, u8* pu8_Val, u8 u8_Size);`	Writes the value of a parameter identified by u16_ParIdx.	
	u16_ParIdx	Index into the parameter description table which is provided by the user.
	pu8_Val	Pointer to a buffer to get the value from.
	u8_Size	Size of the parameter to be written.
	Returns	HRT_ERR_NONE HRT_ERR_ID_ERROR HRT_ERR_SIZE_MISMATCH HRT_ERR_RANGE_ERROR
`u8 HartUL_HandleCommand (HRT_ST_DL_ReqRsp* pst_ReqRsp);`	If the command interpreter receives an unknown command it passes the command to the user by a call of this function.	
	pst_ReqRsp	In: Pointer to a structure to get the request information from.
		Out: Pointer to a structure to put the response information to.
	Returns	HRT_ERR_NONE HRT_ERR_NOT_IMPLEMENTED
Called from the Hart Slave User		
`void HartCI_PutCyclicData (HRT_ST_UL_Cyclic* pst_Cyc);`	Hands over the not encoded cyclic data to the Hart protocol command interpreter.	
	pst_Cyc	Pointer to the structure describing the cyclic data.
`void HartCI_PutDeviceStatus (u8 u8_DevStatus);`	Hands over the device status flags to the communication.	
`void HartCI_RunServiceHandler (void);`	This call is running the command interpreter and other stuff to drive the protocol state machines of the Hart Slave. It should be called every 50 ms at least.	

Network Management

The network management is implementing functionalities which are not strictly associated with one of the communication layers. This includes setting and getting of controlling data.

Declaration	Description	
Called from the Network Management		
`u8 HartUL_Read(...);`	See command interpreter for the definition.	
`u8 HartUL_Read(...);`		
Called from the Hart Slave User		
`u8 HartNM_GetParameter ( u16 u16_ParamID, u8* pu8_Data, u8* pu8_Size);`	Reads the value of a parameter identified by u16_ParamID.	
	pu8_Data	Pointer to a buffer to write the value to.
	pu8_Size	Pointer to an 8 bit value. When called it contains the size of the available buffer, after return it contains the number of read bytes.
	Returns	HRT_ERR_NONE HRT_ERR_ID_ERROR HRT_ERR_SIZE_MISMATCH
`u8 HartNM_SetParameter ( u16 u16_ParamID, const u8* pu8_Data, u8 u8_Size);`	Writes the value of a parameter identified by u16_ParamID.	
	pu8_Data	Pointer to a buffer to read the value from.
	pu8_Size	Pointer to an 8 bit value. When called it contains the size of the available buffer, after return it contains the number of read bytes.
	Returns	HRT_ERR_NONE HRT_ERR_ID_ERROR HRT_ERR_SIZE_MISMATCH HRT_ERR_RANGE_ERROR
`void HartNM_Enable(u8 u8_Mode);`	Sets the mode of the Hart communications. Note: If the mode is set to disabled the device will not respond any more.	
	u8_Mode	HRT_MOD_DISABLED HRT_MOD_ENABLED
`void HartNM_Init(void);`	Initializes the Hart Slave stack.	

Coding (Application Layer)

Header: HartCO.h

Encoding

Declaration	Description
`void HartCO_PutU8 ( u8 u8_Val, u16 u16_Pos, u8* pu8_Buf);`	Put the value u8_Val at position u16_Pos into the buffer pointed to by pu8_Buf.
`void HartCO_PutU16 ( u16 u16_Val, u16 u16_Pos, u8* pu8_Buf);`	Put the value u16_Val at position u16_Pos into the buffer pointed to by pu8_Buf.
`void HartCO_PutU24 ( u32 u32_Val, u16 u16_Pos, u8* pu8_Buf);`	Put the 24 bit value in u32_Val at position u16_Pos into the buffer pointed to by pu8_Buf.
`void HartCO_PutU32 ( u32 u32_Val, u16 u16_Pos, u8* pu8_Buf);`	Put the value u32_Val at position u16_Pos into the buffer pointed to by pu8_Buf.

`void HartCO_PutFloat ( f32 f32_Val, u16 u16_Pos, u8* pu8_Buf);`	Put the single precision float value f32_Val at position u16_Pos into the buffer pointed to by pu8_Buf.
`void HartCO_PutDouble ( f64 f64_Val, u16 u16_Pos, u8* pu8_Buf);`	Put the double precision float value f64_Val at position u16_Pos into the buffer pointed to by pu8_Buf.
`void HartCO_PutStream ( u8* pu8_Stream, u16 u16_Size, u16 u16_Pos, u8* pu8_Buf);`	Put the byte stream pointed to by pu8_Stream at position u16_Pos into the buffer pointed to by pu8_Buf. The number of bytes is contained in u16_Size.
`void HartCO_PutPA ( u8* pu8_String, u16 u16_Size, u16 u16_Pos, u8* pu8_Buf);`	Put the string of size u16_Size pointed to by pu8_String at position u16_Pos as packed ASCII into the buffer pointed to by pu8_Buf. Note: 4 characters in a string requires 3 bytes if they are coded as packed ASCII.

Decoding

Declaration	Description
`u8 HartCO_GetU8 ( u16 u16_Pos, const u8* pu8_Buf);`	Return the data at position u16_Pos from the buffer pointed to by pu8_Buf as 8 bit unsigned integer value.
`u16 HartCO_GetU16 ( u16 u16_Pos, const u8* pu8_Buf);`	Return the data at position u16_Pos from the buffer pointed to by pu8_Buf as 16 bit unsigned integer value.
`u32 HartCO_GetU24 ( u16 u16_Pos, const u8* pu8_Buf);`	Return the data at position u16_Pos from the buffer pointed to by pu8_Buf as 24 bit unsigned integer value.
`u32 HartCO_GetU32 ( u16 u16_Pos, const u8* pu8_Buf);`	Return the data at position u16_Pos from the buffer pointed to by pu8_Buf as 32 bit unsigned integer value.
`f32 HartCO_GetFloat ( u16 u16_Pos, const u8* pu8_Buf);`	Return the data at position u16_Pos from the buffer pointed to by pu8_Buf as single precision float value.
`f64 HartCO_GetDouble ( u16 u16_Pos, const u8* pu8_Buf);`	Return the data at position u16_Pos from the buffer pointed to by pu8_Buf as double precision float value.
`void HartCO_GetStream ( u8* pu8_Stream, u16 u16_Size, u16 u16_Pos, const u8* pu8_Buf);`	Copy the bytes at position u16_Pos from the buffer pointed to by pu8_Buf to a buffer pointed to by pu8_Stream. The number of bytes is given by u16_Size.
`void HartCO_GetPA ( u8* pu8_String, u16 u16_Size, u16 u16_Pos, const u8* pu8_Buf);`	Decode the packed ASCII data at position u16_Pos from the buffer pointed to by pu8_Buf to a string pointed to by pu8_String. The number of characters of the string is given by u16_Size. Note: 3 bytes of packed ASCII data expands to a 4 characters string.

Common

Structures

ST_DL_Config

Header: HartDL.h

Member	Description
u8_BaudRate;	The baudrate is also depending from the target. Therefore the selection of baudrates is optional (HRT_BR_1200, HRT_BR_9600, HRT_BR_19200, HRT_BR_38400, HRT_BR_57600).
b8_CarrDetSupported	Specifies that the carrier detection is supported (HRT_B_FALSE, HRT_B_TRUE).
u8_NumPreambles	Number of preambles to be sent (2..20).

ST_DL_Request

Header: HartDL.h

Member	Description
u16_ServiceID	This is provided by the Hart Slave and should be returned in the response passed back.
u16_Command;	Hart command.
u16_DataSize	Number of bytes in the payload stream.
au8_Data[HRT_MAX_NUM_BYTES]	Stream of octets sent with the request.

ST_DL_Response

Header: HartDL.h

Member	Description
u16_ServiceID	This is provided by the Hart Slave and should be returned in the response passed back.
u16_Command;	Hart command.
u16_DataSize	Number of bytes in the payload stream of the response.
u8_DevSpecRspCode	Command specific response code.
au8_Data[HRT_MAX_NUM_BYTES]	Stream of octets sent with the request.

ST_DL_Cyc

Header: HartDL.h

Member	Description
u8_Command	Burst command
u16_DataSize	Number of bytes in the payload stream of the burst.
u8_DevSpecRspCode	Command specific response code.
au8_Data[HRT_MAX_NUM_BYTES]	Stream of octets sent with the burst.

ST_UL_CycObject

Header: HartUL.h

Member	Description
u16_ParamIdx	Index into the user parameter table
pu8_Data	Pointer to the data of the object

ST_UL_Cyclic

Header: HartUL.h

Member	Description
u8_NumCycObjects	Number of cyclic objects
ST_UL_CycObject ast_CycObjets [HRT_MAX_NUM_CYC_OBJECTS]	Array of cyclic objects

ST_UL_ParamDescr

Header: HartUL.h

Member	Description
u8_ParamType	HRT_PTY_U8, HRT_PTY_U16, HRT_PTY_U24, HRT_PTY_U32, HRT_PTY_F32, HRT_PTY_F64, HRT_PTY_STR, HRT_PTY_PAC
u8_MemClass	HRT_MTY_ROM, HRT_MTY_NV, HRT_MTY_RAM
u8_Access	HRT_ACC_WRITE_PROT
u8_Size	Optional size of the parameter.
pu8_Data	Pointer to the data of the parameter. If this value is NULL the command interpreter will use the read function to get the value.

ST_UL_CmdItem

Header: HartUL.h

Member	Description
u16_ParamIdx	Index into the parameter description Table
u8_Pos	Position in the payload data stream

ST_UL_CmdDescr

Header: HartUL.h

Member	Description
u16_Command	Hart command
u8_Attrib	Command attribute HRT_CMDAT_READ, HRT_CMDAT_WRITE
u8_NumItems	Number of cmd items
apst_CmdItem[]	Array of pointers to cmd items. Last element of the array has to be NULL.

ST_UL_UserData

Header: HartUL.h

Member	Description
ST_UL_ConstantData	Pointer to Constant data as device ID, manufacturer ID etc.
	Command attribute HRT_CMDAT_READ, HRT_CMDAT_WRITE
u8_NumItems	Number of cmd items
apst_CmdItem[]	Array of pointers to cmd items. Last element of the array has to be NULL.

Constants

All constants are defined in HartAPI.h.

Error codes

```
#define HRT_ERR_NONE              0
#define HRT_ERR_ACC_DENIED        1 //Access denied
#define HRT_ERR_TX_REJECTED       2 //Tx not possible because already
                                  3 //  running or resource not available
#define HRT_ERR_ID_ERROR          4 //Identifier (parameter index) not supported
#define HRT_ERR_SIZE_MISMATCH     5 //One of the buffers has wrong size
#define HRT_ERR_RANGE_ERROR       6 //One of the values is out of range
#define HRT_ERR_NOT_IMPLEMENTED   7 //Command or function not implemented
```

NM Parameter IDs

```
#define HRT_PID_ADDR          0 //8 bit device poll address(0..63)
#define HRT_PID_STAG          1 //Short tag name 6 byte packed acsii
#define HRT_PID_LTAG          2 //Long tag name 24 byte packed ascii
#define HRT_PID_NUM_PA        3 //u8 Number of preambles to be sent
#define HRT_PID_BAUD          4 //8 bit enum for the baud rate (optional)
#define HRT_PID_CARR_SUPP     5 //Flag if carrier det is supported
#define HRT_PID_MANU_ID       6 //u8 manufacturer id
#define HRT_PID_DEV_TYPE      7 //u8 device type
#define HRT_PID_HART_REV      8 //u8 hart revision (5..7)
#define HRT_PID_MIN_PA_RQ     9 //u8 Minimum number of preambles in request (5)
#define HRT_PID_DEV_REV      10 //u8 Device revision level
#define HRT_PID_SW_REV       11 //u8 Software revision level
#define HRT_PID_HW_REV       12 //u5 Hardware revision level
#define HRT_PID_FLAGS        13 //u8 Hart flags
#define HRT_PID_UNDEVID      14 //u24 unique device id
#define HRT_PID_MIN_PA_RS    15 //u8 Minimum number of preambles in response (5)
#define HRT_PID_MAX_DEV_VAR  16 //u8 Maximum number of device variables
#define HRT_PID_CFG_CHN_CNT  17 //u16 Configuration changed counter
#define HRT_PID_EXT_DEV_ST   18 //u16 Extended field device status
#define HRT_PID_RUNIT        19 //u8 Range units code
#define HRT_PID_LRANGE       20 //f32 Lower range value
#define HRT_PID_URANGE       21 //f32 Upper range value
```

Protocol Machine Status

```
#define HRT_DL_STAT_IDLE      0 //No communication running
#define HRT_DL_STAT_STARTING  1 //Protocol starting, do not change mode
#define HRT_DL_STAT_RUNNING   2 //Protocol active, device is responding
```

Carrier State

```
#define HRT_CARR_OFF 0 //No carrier detected
#define HRT_CARR_ON  1 //Carrier present
```

Boolean Values

```
#define HRT_B_FALSE 0 //u8 boolean false
#define HRT_B_TRUE  1 //u8 boolean true
```

Parameter Types

```
#define HRT_PTY_U8   1 //8 bit integer
#define HRT_PTY_U16  2 //16 bit integer
#define HRT_PTY_U24  3 //24 bit integer
#define HRT_PTY_U32  4 //32 bit integer
#define HRT_PTY_F32  5 //float
#define HRT_PTY_F64  6 //double
#define HRT_PTY_STR  7 //byte stream
#define HRT_PTY_PAC  8 //packed ascii
```

Parameter Memory Class

```
#define HRT_MTY_ROM  0 //Read only
#define HRT_MTY_NV   1 //None volatile memory
#define HRT_MTY_RAM  2 //Free accessable
#define HRT_MTY_DYN  3 //Dynamic, read only
```

Global Description Data

The commands and the parameters of the user need some description by the user. This is handled by a few global data structures which are demonstrated in the following.

User Parameter Table Example

```
HTY_ST_UL_ParamDescr HartUL_ast_ParameterTable[] =
{
  /* HRT_ULPIDX_RNG_UNIT */ { HRT_PTY_U8,  HRT_MTY_NV, 0, 1, &m_u8_RangeUnit },
  /* HRT_ULPIDX_LO_RANGE */ { HRT_PTY_F32, HRT_MTY_NV, 0, 4, NULL            },
  /* HRT_ULPIDX_UP_RANGE */ { HRT_PTY_F32, HRT_MTY_NV, 0, 4, NULL            },
  /* END OF TABLE        */ { HRT_PTY_NONE,          0, 0, 0, NULL            }
};
```

Note: If the reference to the parameter's value is set to NULL the command interpreter will use the read function provided by the user to get the value.

This table has to list all user parameters which are used in the universal and common practice or user commands except those commands which are handled by the user's software modules.

Cmd Item Definition Example

```
static HTY_ST_UL_CmdItem mst_C35item_RangeUnit =
  { HRT_ULPIDX_RNG_UNIT, 0 };
static HTY_ST_UL_CmdItem mst_C35item_LoRange =
  { HRT_ULPIDX_LO_RANGE, 1 };
static HTY_ST_UL_CmdItem mst_C35item_UpRange =
  { HRT_ULPIDX_UP_RANGE, 5 };
```

Cmd 35 Descriptor Example

```
static HTY_ST_UL_CmdDescr Cmd35descriptor =
  { 35, HRT_CMDAT_WRITE, 3,
    { &mst_C35item_RangeUnit, &mst_C35item_UpRange, &mst_C35item_LoRange
    }
  };
```

User Cmd Descriptors List Example

```
HTY_ST_UL_CmdDescr* HartUL_apst_CmdDescriptors[] =
{
  //Other cmd descriptors...
  &Cmd35descriptor,
  NULL
};
```

All command descriptors are contained in one list, which is provided by the user developer.

Command Interpreter

Universal Commands

The following universal commands are handled directly by the command interpreter.

Command	Comments
Command 0	Read Unique Identifier
	The command interpreter is supporting the command without notifying the user. All required parameters are part of the network management.
Command 6	Write Polling Address
	The command is handled directly by the command interpreter. However, the application is notified by calling the Write function of the user's software modules to store the new address in the none volatile memory.
Command 11	Read Unique Identifier Associated With Tag
	Same handling as command 0.
Command 21	Read Unique Identifier Associated With Long Tag
	Same handling as command 0.

Table 6: Implicit Universal Commands

The other universal commands needs a descriptor in the user's cmd descriptors table to get the proper access of the required parameters.

Common Practice Commands

The common practice commands are predefined by Hart even if they are optional.

In the implementation of the Hart Slave some of these commands are used for the data link layer operation. As it is done with the above listed universal commands they are completely handled by the communication stack.

Command	Comments
Command 59	Write Number Of Response Preambles
	The command interpreter is supporting the command without notifying the user. All required parameters are part of the network management.
Command 109	Burst Mode Control
	The command is handled directly by the command interpreter. However, the application is notified by calling the Write function of the user's software modules to store the new burst mode in the none volatile memory.

Table 7: Implicit Common Practice Commands

User Specific Commands

One of the most important kind of commands for the device developer are the user specific commands.

User specific commands can be realized in two ways.

One way is to describe the commands in the user's cmd descriptors list, taking advantage of the Read and Write callbacks from the Hart communication stack.

The other way is to implement a user specific part of the command interpreter.

If the command interpreter of the Hart slave is receiving a command which is neither implicitly known nor described in the user's cmd description list it calls the user function HartUL_HandleCommand (). This function can be used by the user software developer to implement whatever is needed for the device realization.

However, for testing purposes two commands have been implemented in HartCI.c.

Test Commands

Command	Comments				
Command 150	Test the data type double				
	Request	#0	#8	#12	#20
		d1: double	f1: float	d2: double	f1: float
	Response	#0	#4	#12	#16
		f2: float	d2: double	f1: float	d1: double
Command 65000	The test has a similar behavior as command 150 but for the 16 bit command 65000				
	Request	#0	#8	#12	#20
		d1: double	f1: float	d2: double	f1: float
	Response	#0	#4	#12	#16
		f1: float	d1: double	f2: float	d2: double

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