# HART® Slave Stack

## Integration Manual for HART® Slave Stack

**MESCO Systems GmbH** | Berner Weg 7 | 79539 Lörrach | Tel. +49 7621 1575 475 | Fax +49 1575 175 | info@mesco-systems.com | www.mesco-systems.com

# Integration Manual for HART® Slave Stack

**MESCO Product Information**

| | |
|---|---|
| Product Name | HART® Slave Stack |
| Product Version | V7.9 Rev. A |

| | |
|---|---|
| **Document Identification** | HartStack_IntegrationManual |
| Status | Release |
| Issue Date | 2026-06-30 |
| Issued by | Elvys Melo/eme |

**Copyright**

Copyright 2026, MESCO Systems GmbH. All rights reserved. MESCO Systems reserves the right to make changes and improvements to its product without providing notice.

## Table of contents

## Table of figures

## Table of tables

# 1 Revision history

## 1.1 HART Slave Stack release notes

### V7.4 Rev. A

| Change | Category |
|---|---|
| --- start of change documentation --- | |

### V7.5 Rev. A

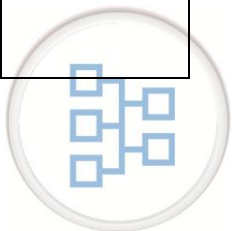
| Change | Category |
|---|---|
| **Command #524:**<br>Modification of mapping of bits 'more status available' and 'configuration changed' is not allowed (see HCF_SPEC-183 Rev 23.0 page 91). | Bugfix |
| **Loop Current Trimming:**<br>In Cmd #45 only the offset value (p_hvars_flash->current_trim_offset) must be adjusted. Gain must be kept! | Bugfix |
| **Condensed Status mapping:**<br>The default mapping of device specific status bits can be configured by the user now. Therefore the definition was moved to from the stack internal file hart_hap.c to the user specific file hart_config.h. | Enhancement |
| **Number of Dynamic Variables:**<br>The number of Dynamic Variables is now configurable to 1..4 (before it was only allowed to have 4 dynamic variables). | Enhancement |
| **Mapping of Dynamic Variables:**<br>The allowed mapping of the Dynamic Variables is now configurable. | Enhancement |
| **Alarm state/current handling:**<br>The Alarm state handling is done by the stack now (before it has to be done by the application). Therefore the API was changed: hart_setAlarmState() added; hart_setPvAlarmCode() changed to hart_checkPvAlarmCodeAndGetAlarmCurrent().<br>The PV alarm code is passed to the application. The application has to check it and return the alarm current to the stack. The alarm state is passed to the stack now and the stack sets the current to the alarm current. LOOP_CURRENT_FIXED and LOOP_CURRENT_SATURATED is also set correctly by the stack in alarm case. | Enhancement |
| **API for the loop current:**<br>The API function for the loop current hart_setLoopCurrent() was enhanced, so that the application gets also the 'real' loop current (not only the 'corrected' loop current). | Enhancement |

### V7.5 Rev. B

| Change | Category |
|---|---|
| No functional change, only documentation updated. | --- |

### V7.5 Rev. C

| Change | Category |
|---|---|
| **Command #48:**<br>The condensed status bits must be updated before the response of Cmd48 is built. Otherwise, the reaction of the condensed status bits was one command too late. | Bugfix |
| **Command #81:**<br>Copy/paste bug in access of trimpoint limits. The command should return min/max values of upper and lower trimpoint, but instead it returned 4 times the min value of the lower trimpoint. | Bugfix |
| **Command #523:**<br>- The command behavior was not correct when 255 entries were requested. In this case the command responded with 0 entries.<br>- Correction of checks for "start index" and "number of entries". Unequal start indexes were not handled correctly before. | Bugfix |



| | |
|---|---|
| **Command #524:**<br>- The mapping code of the bits "more status available" and "configuration changed" must always be “No effect”. It is not allowed to change it. The implementation of this check was erroneous. The command was always rejected if the request contains the mapping of these bits. Now it is allowed to write the mapping of this bit, but it is not allowed to write a mapping code different from “No effect”.<br>- The check for valid range of the mapping code was erroneous. The invalid code 2 (reserved) was not checked. Furthermore, the check was done within the loop, in which the values are adopted. So, if one mapping code was invalid, the data before this code was adopted anyway!<br>- Correction of checks for “start index” and “number of entries”. Unequal start indexes were not handled correctly before. | Bugfix |
| **Condensed Status / Undefined bits in Cmd48 status data:**<br>Now it is possible to mark the status bits in Cmd48 data as “not defined”. For the status bits, which are handled by the application, a corresponding interface was added. If a status bit is marked as "not defined", Cmd #524 only accepts mapping code “Not defined” and Cmd #523 always returns mapping code “Not defined” for this bit. | Enhancement |

## V7.5 Rev. D

| Change | Category |
|---|---|
| **Command #109:**<br>Command #109 must return response code “Invalid selection” (2) instead of “Invalid Burst message number” (9). Bug detected with FCG_KIT-192 V3.0 CAL037D. | Bugfix |

## V7.5 Rev. E

| Change | Category |
|---|---|
| **Actuator enhancement:**<br>Now the stack can also be used for HART actuator devices. With compiler switches it can be decided whether the stack implements a HART actuator or transmitter. For the actuator variant the data flow of the Primary Variable is inverted. Furthermore, it is possible to define multiple additional transmitter and actuator Device Variables. | Enhancement |

## V7.5 Rev. F

| Change | Category |
|---|---|
| **Alarm current / Loop current disabled:**<br>The alarm state handling has now a lower priority than the loop current mode. Previously it could be possible, that the device sets a high failure current even when loop signaling was disabled. | Bugfix |
| **Commad #53:**<br>Bugfix in Command 53. According to FCG Test and Spec_151 Response Code for RC ”Invalid unit code” must be RC12 instead of RC18. | Bugfix |
| **New DV parameters for “reported” LTL and UTL:**<br>Most Slave Device implementations have included a small buffer above the Upper Transducer Limit and below the Lower Transducer Limit to allow a slightly bigger process value span than reported by command #14 and command #54. In order to be able to use this kind of buffer, two new parameters have been created: “reported_upper_transducer_limit” and “reported_lower_transducer_limit” are used to define the specified transducer limits by the device. | Enhancement |
| **API function hart_updateTransmitterDvAndStatus():**<br>Function verifies now that it is only called for transmitter variables. Otherwise, it returns with an error. | Enhancement |

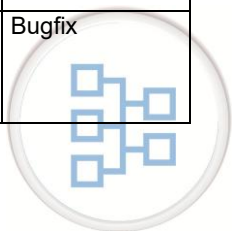
## V7.5 Rev. G

| Change | Category |
|---|---|
| **Command #9 Invalid timestamp**<br>Timestamp was wrong if requested DV in Slot0 was invalid. Now “Not-a-number” is returned for timestamp in this case. | Bugfix |
| **Command #9 Slot handling**<br>In case of normal STX/ACK communication the response was truncated in case of DV code “DEVICE_VARIABLE_NOT_USED” (250) was requested. This must only be done in case of burst. In normal STX/ACK communication NaN is returned in these slots. | Bugfix |

| | |
|---|---|
| **Command #107 Configure DV Slots**<br>All valid DV codes and DV code “DEVICE_VARIABLE_NOT_USED” was accepted for all slots. But slot0 must always contain a valid DV code. Therefore Command #107 has been adapted to accept only valid DV codes for slot0.<br>As a consequence, hart_resetHartIfaceToDefaults() now resets slot0 to DV code 0 instead of “DEVICE_VARIABLE_NOT_USED”. | Bugfix |
| **SI-Unit code check Commands #44, #53:**<br>The check for SI-Unit code has been added to Command #44 and Command #53. If the device is configured to accept SI-Unit codes only, all unit codes that are not the one specified in DEVICE_VARIABLE_ROM-> si_unit_code are rejected with “COMMAND_INVALID_SELECTION”. | Bugfix |
| **Command #100:**<br>In case command 100 requests the same alarm value as configured, the actually used alarm value is sent. | Bugfix |
| **SI-Unit code handling:**<br>The check for SI compliance was removed from the application interface. A new ROM variable was added to “DEVICE_VARIABLE_ROM” structure to specify the the SI-Unit code of every single device variable. | Enhancement |
| **Usage of user data in non-volatile data:**<br>Application interface has been enhanced to keep user defined data in non-volatile memory structure. | Enhancement |
| **Factory Commands:**<br>Factory Commands have been moved to the “hart_custom” folder in order to let the user specify the factory commands. | Enhancement |
| **Command #128 Usage of user data:**<br>Command #128 was enhanced to show an example of how to use the user data interface. | Enhancement |
| **Application interface enhancement:**<br>Function ‘hart_getRequestBytecount()’ added.<br>Function ‘hart_resetHartIfaceToDefaults()’ added. | Enhancement |

## V7.5 Rev. H

| Change | Category |
|---|---|
| **Handling of more status available bit in case of status simulation active**<br>When status simulation active is active, the more status available flag reacts on the “real” status data instead of the simulated status data. This is fixed now. Furthermore, the more status available bit handling is completely done within the stack now (see description of more_status_available_status_mask[] parameter). | Bugfix/ Enhancement |
| **Different limits for loop current for command #40**<br>Now it is possible to set different limits for the loop current in command #40. Before the limits were the same than the “normal” loop current range.<br>(see LOOP_CURRENT_MIN_CMD40, LOOP_CURRENT_MAX_CMD40) | Enhancement |
| **Handling of alarm selection code**<br>When alarm code “hold last value” is selected, the alarm value was not updated correctly. Now the interface is changed. The function hart_checkPvAlarmCodeAndGetAlarmValue() is split up into hart_setPvAlarmCode() and hart_getPvAlarmValue(). The function hart_getPvAlarmValue() is called once in each cycle if alarm state is active. So, the user application can implement user specific behavior. | Bugfix/ Enhancement |
| **Loop current trim algorithm optimized**<br>The loop current trim algorithm is optimized. Now it works like the device variable. Instead of adjusting gain (cmd #45) and offset (cmd #46) separately, now the lower trimpoint (cmd #45) and upper trimpoint (cmd #46) is stored and the gain/offset values are calculated from these trimpoints. With the old version some iteration was necessary to get an exact trimming. With the new method the trimming is exact already after the first loop. | Enhancement |
| **Status simulation must not be affected from “write protect” and “device lock”**<br>Now commands #526 and #527 are not affected from “write protect” and “device lock” anymore. | Bugfix |
| **Burst trigger selection when using special DV code in Slot 0**<br>The burst trigger could not be set to values different from “continuous” when using a special DV code (PV,SV,TV,etc.) in slot 0. | Bugfix |
| **Timestamp in cmd #9 response**<br>The timestamp value was wrong when requesting “Percent of range” or “Loop current” in slot 0.<br>If slot 0 DV code is invalid, NaN was returned as timestamp. Timestamp is not a float value, so now 0 is returned instead. | Bugfix |



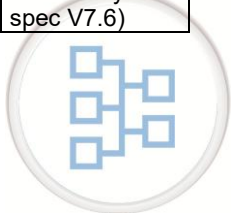
| | |
|---|---|
| **Bugfix in damping calculation**<br>In case of very small damping value the calculation has amplified the process value more than damped → this could lead to a NaN calculation of the process value. Now the damping tau is checked for being greater than delta time to avoid this. Furthermore, every device variable update sets the last damped timestamp to avoid damping is not executed after a long pause. | Bugfix |

## V7.5 Rev. I

| Change | Category |
|---|---|
| **Order of response codes Cmd #82**<br>Changed order of response codes "Applied process..." and "Passed_Parameter..." for Cmd #82. Specification does not define the order but with the change the stack also passes CAL080 with third party test system. | Enhancement |
| **Changed Cmd #45 / #46 (loop current trim mechanism)**<br>Split configuration LOOP_CURRENT_TRIM_MAX_DIFF into LOOP_TRIM_MAX_DIFF_SETPOINT and LOOP_TRIM_MAX_DIFF_ERROR to be more flexible. | Enhancement |
| **Defaults for LRV/URV**<br>Use "reported" lower/upper transducer limits for LRV/URV defaults instead of "real" lower/upper transducer limits. | Bugfix |
| **Magic pattern and parameter set version in NV data**<br>Implemented MAGIC_PATTERN, PARAMETER_SET_VERSION in hvars_flash (to detect empty or wrong NV memory content) | Enhancement |
| **API change for "Device ID", "Device revision", "HW revision", "Private label distributor code"**<br>- Device ID not via hart_init() parameter any more. Now it is a hvars_flash parameter<br>- Following parameters are moved from hvars_rom to hvars_flash: device revision, hw revision, private label distributor code; new API functions added to set these parameters (e.g. by factory cmds)<br>- hart_performDeviceReset() must ensure that all NV data is written to NV memory before performing device reset now! | Enhancement |
| **Bugfix in DLL state machine**<br>Transition to MAC_OFF is now done in MAC_BURST_CHECK_TRIGGER state instead of MAC_ANSWER_FINISH state. This is necessary to reach MAC_OFF if no communication is active. Otherwise reset will never be performed (e.g. by Cmd #42). | Bugfix |
| **Configuration of "Write protection" support**<br>Compiler switch WRITE_PROTECTION_ENABLED implemented | Enhancement |
| **New API function to read back real (trimmed) DV value and status**<br>Add API function hart_getRealDvValueAndStatus() | Enhancement |
| **Fixed handling of "More status available" (MSA) bit**<br>Old behavior (before the change):<br>- MSA is set at any change of an affected Cmd #48 status bit<br>- MSA is cleared by Cmd #48 call with correct request data.<br>New behavior (after the change):<br>- MSA is set if current Cmd #48 data is different than Cmd #48 acknowledged by the master (by sending Cmd #48 with corresponding request data).<br>- MSA is cleared if current Cmd #48 data is equal to Cmd #48 acknowledged by the master (by sending Cmd #48 with corresponding request data).<br>- MSA is cleared by Cmd #48 call with correct request data (no change here). | Bugfix |
| **Influence of Condensed status bits to "More status available" (MSA) bit**<br>All Condensed Status bits affect the MSA bit now independent from the setup in more_status_available_status_mask[]. (According to See HCF_SPEC-151, Rev. 10.0, page 32, footnote 25) | Bugfix |

## V7.6 Rev. A

| Change | Category |
|---|---|
| **Reject command #31**<br>Cmd #31 is not a real command, therefore it's rejected with COMMAND_NOT_IMPLEMENTED (instead of COMMAND_TOO_FEW_DATA_BYTES_RECEIVED). This occurs when Cmd #31 is requested in aggregated command (cmd #78). Also see HCF_SPEC-099, Rev. 10.0, chapter 7.2.5 | Bugfix (Behavior clarified by new spec V7.6) |



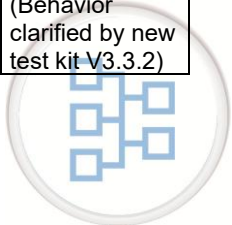
| | |
|---|---|
| **Status bits with special condensed status mapping behavior / restrictions**<br>Cmd #524:<br>- Update of status bits for which the CS mapping is fixed (not all of them were implemented before)<br>- Response code COMMAND_SET_TO_NEAREST_POSSIBLE_VALUE (warning) has to be used if status bit with fixed mapping or undefined status bit is attempt to be changed. Before we rejected the complete command with INVALID_SELECTION | Bugfix (Behavior clarified by new spec V7.6) |
| **Enable/disable of possibility to change unit code for special DVs**<br>Cmd #53: COMMAND_INVALID_DEVICE_VARIABLE_CODE has to be returned if DV only supports one unit code! (Before we accepted the correct unit code and rejected others with COMMAND_INVALID_UNITS_CODE). For realization new parameter 'multiple_unit_codes_allowed' is implemented in DVs ROM database (hart_parameters.c). | Bugfix (Behavior clarified by new spec V7.6) |
| **Write protection for Command #526 and #527 (Status simulation)**<br>Command #526 is affected by Write Protect and Locked. No comment for Command #527, so we assume that it's not affected by Write Protect and Locked. | Bugfix (Behavior clarified by new spec V7.6) |
| **Minimum request bytes for Cmd #527 (Simulate status bit)**<br>Minimum request bytes for Cmd #527 are 2 bytes, not 1 byte! | Bugfix |

## V7.6 Rev. B

| Change | Category |
|---|---|
| **Bugfix of burst period timing**<br>Before the bugfix, the period timer started when a burst message was triggered. But this point can be delayed (due to other bus activity). So, the delays were added up further and further so that the average period over long period was not correct. This leads to DLL045 failing in special cases. Now, the period timer is independent from the delay. The average period matches the configured time exactly now. | Bugfix |
| **Bugfix in delayed response mechanism**<br>DR commands were registered / started multiple times in hart_registerDelayedResponse(). | Bugfix |
| **Bugfix in status simulation**<br>When enabling status simulation with cmd #526 the status data is latched. In special cases it could happen that condensed status bits were not latched correctly. | Bugfix |
| **Special handling for non-public (factory) commands**<br>Non-public (factory) commands are rejected with response code “Command not implemented” if too few request data bytes are available. This way, the commands hidden from the end-user. | Enhancement |
| **Demo applications clock source changed**<br>Changed clock source for demo applications from 4.2MHz MSI to 8 MHz HSE (external clock from ST-Link controller; HSE oscillator bypassed); now clock is 8MHz / 2 (PLL) = 4MHz. Clock is more accurate now because it’s coming from crystal oscillator. | Enhancement (Demo appli only) |
| **Restructured demo application files**<br>Demo application source code files were restructured. Now a generic folder which belongs to transmitter and actuator demo is available. | Enhancement (Demo appli only) |
| **Added unit conversion module to demo application**<br>A unit conversion module was added to demo application which can be easily adapted to device specific unit codes. So the implementation of the DV unit code conversion is much easier now. | Enhancement (Demo appli only) |
| **Added ISO3166 module to demo application**<br>An ISO3166 module was added to the demo application which checks the country codes according to ISO3166. | Enhancement (Demo appli only) |
| **Updated EEPROM module in demo application**<br>EEPROM module is updated so that one call of eeprom_processor() executes only one erase/write operation. This way the function execution time is 1xTprog (~4ms) maximum and the application timing is not delayed for 2xTprog (~8ms) any more. | Enhancement (Demo appli only) |

## V7.6 Rev. C

| Change | Category |
|---|---|
| **Condensed status – Behavior of commands #523 and #524**<br>The modification of the requested range to the nearest possible value and the cases in which “invalid selection” or “too few data bytes received” has to be returned is adapted to the expectations of new FCG_KIT-192 V3.3.2. | Bugfix (Behavior clarified by new test kit V3.3.2) |



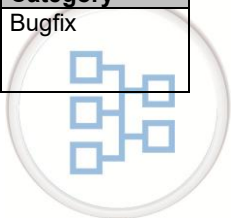
| | |
|---|---|
| **Status simulation – Behavior of command #527**<br>- It must be possible to simulate 'Device malfunction' bit (affected test cases: CAL523C, CAL524D, CAL526B, CAL526C, CAL526F)<br>- Bugfix in check of bit index range<br>- Rejection of cmd #527 in case of status simulation disabled must have a higher priority than the checks which cause "invalid selection" (affected test case: CAL526B)<br>- Simulation of bits in "operating mode" byte not allowed | Bugfix (Behavior clarified by new test kit V3.3.2) |
| **Classification of unsupported Dynamic Variables in Cmd #8 response**<br>Cmd #8 has to return DV Classification DV_CLASS_NOT_CLASSIFIED (0) instead of DEVICE_VARIABLE_NOT_USED (250) if Dynamic Variable is not supported. See HCF tracker #3775. | Bugfix |
| **Bugfix in aggregated commands behavior (Cmd #78)**<br>Before the bugfix the implemented behavior was, that all requests are rejected with RC 9 (Invalid Command requested) if any of the requested embedded commands is not implemented or doesn't have the FLAG_AGGREGATED flag. That's contradictory to specification of cmd #78.<br>New behavior:<br>1) If requested command is not implemented<br>=> Cmd #78 succeeds anyway, embedded response code is set to "Command not implemented"<br>2) If requested commands combination is not allowed (flag FLAG_AGGREGATED missing)<br>=> Cmd #78 rejected with RC 2 (Invalid selection)<br>3) If one of the requested commands is 31 or 78<br>=> Cmd #78 rejected with RC 9 (Invalid Command requested) | Bugfix |
| **Bugfix in Cmd #44 and cmd #53 behavior**<br>Commands must succeed if requested unit code is the same than the currently set unit code, otherwise CAL044 and CAL053 fail. | Bugfix |
| **Changed min. allowed extended command number from 512 to 256 (cmd #31)**<br>According to HCF_SPEC-085, Rev. 2.0, page 41, footnote 6 the former behavior is also ok but the change was necessary to pass test case DLL024C. | Bugfix (Workaround for test kit bug) |
| **Bugfix in allowed DV code ranges for cmd #9 and cmd #33**<br>DV codes 0-249 are accepted and slots are filled with NaN if DV is not available. DV codes 250-255 are rejected with 'invalid selection'. Before this bugfix all DV codes except of 255 were accepted.<br>(See HCF tracker #5680) | Bugfix |
| **Bugfix in cmd #103 (burst trigger periods)**<br>If the requested min. period time is higher than the max. period time, the stack has to adjust both values to the higher value instead of the lower value. | Bugfix |
| **Small code cleanup issues**<br>- Change type of parameter 'time_ms' of function hart_setRTC() from 'h_uint32_t' to 'h_int32_t' to be compatible with type 'HART_TIME_MS'<br>- Made inline functions static to avoid compiler warning "xxx is static but used in inline function yyy which is not static"<br>- bugfix: check_status_limited() was only active if ACTIVATE_CMD035 was active<br>- check_status_limited() only active if ACTIVATE_CMD036 or ACTIVATE_CMD037 | Code cleanup (no functional change) |
| **Bugfix in demo application EEPROM driver**<br>Padding mechanism did not work correctly. Problem was unrecognized because compiler makes natural alignment if possible and therefore we never had padding bytes. | Bugfix (Demo appli only) |

## V7.6 Rev. D

| Change | Category |
|---|---|
| **Bugfix in condensed status mapping for unimplemented bits**<br>All status bits which are defined in the specification must have a condensed status mapping although if the device does not support them. | Bugfix |
| **New status bit "Battery or Power Supply needs Maintenance"**<br>Added new status bit „Battery or Power Supply needs Maintenance" to standardized status 1. | Enhancement |

## V7.6 Rev. E

| Change | Category |
|---|---|
| **Bugfix of Burst Timing**<br>Changed DLL state machine to optimize burst timing. Before it took to much hart_loop() calls until a BACK frame was started. This caused DLL028 (FCG_KIT-192, V3.5) to fail. | Bugfix |



| | |
|---|---|
| **Removed unnecessary parameters**<br>Removed DEFAULT_DEVICE_ID, DEFAULT_DEVICE_REVISION, DEFAULT_HARDWARE_REVISION and DEFAULT_PRIVATE_LABEL_DISTRIBUTOR_CODE from hart_config.h because these default values need not to be changeable by the user. The user has to use the API functions to change these values. | Enhancement |
| **Default Private Label Distributor code**<br>The Manufacturer ID is taken as default value for Private Label Distributor code now. | Enhancement |
| **Manufacturer ID for Demo Applications**<br>Changed Manufacturer ID of demo applications from 0 to 0x67FF. This way we have valid values for Manufacturer ID and Private Label Distributor code in default configuration. Otherwise DLL032 fails with failure point 346. | Bugfix<br>(Demo appli only) |

### V7.6 Rev. F

| Change | Category |
|---|---|
| Now commands can take up to 200ms. | Bugfix |
| DLL statemachine changed. HART commands are executed in the main context now. | Bugfix |
| Hart_loop() is executed in the SysTick_Handler() now. Must be called every 10ms. | Bugfix |
| Initialization of GPIOs before initialization of RTS pin now | Bugfix |

### V7.7 Rev. A

| Change | Category |
|---|---|
| HART commands can be performed in slices | Enhancement |

### V7.7 Rev. B

| Change | Category |
|---|---|
| Now GAP Error during receiving SOM is handled correctly | Bugfix |

### V7.9 Rev. A

| Change | Category |
|---|---|
| Now GAP Error during receiving SOM is handled correctly | Bugfix |
| CMD#42 disabled | Specification alignment |
| CMD#48, CMD#526, and CMD#527 set or clear the MSA in accordance with the specification. | Bugfix |
| With CMD#527, the state of the simulation-flag cannot be changed. | Bugfix |
| CMD#40, CMD#525 is write protected | Specification alignment |
| Trim points (CMD#43, CMD#52, CMD#82, CMD#83) are rounded. | Bugfix |
| Behavior of more status available (MSA) in simulation mode follows the specification | Bugfix |
| Reduction of array accesses | Enhancement |
| The stack only responds to specific commands sent to a broadcast address | Bugfix |
| Provide a configuration file to configure compiler macros for packed structs. | Enhancement |
| In the actuator variant, Commands #35, #36, and #37 are disabled because the current is controlled by the master. | Enhancement |
| In the actuator variant, Command #14 uses milliamps as the unit for the primary variable (PV) only. | Specification alignment |

## 1.2 Integration Manual changes documentation

*Status: M=Modified, A=Added, D=Deleted*

| Version | Chapter | Change | Status |
|---|---|---|---|
| V7.5 Rev. B | | --- start of change documentation --- | |
| V7.5 Rev. C | 6.2.2.4.2 | Table 4 updated. New responsibility type “unsupported” added.<br>Interface description added for unsupported Cmd48 status bits. | M |
| | 6.2.2.2.1 | Changed maximum number of device variable from 255 to 100 (DEVICE_VARIABLE_NUM). | M |
| V7.5 Rev. D | 6.2.2.1.4 | Added timing measurements for hart_loop function. | A |
| | 6.4 | Sequence diagram of transmission of HART message added. | A |
| | 6.2.2.3.2 | Bugfix: Hyperlink to chapter 6.2.2.2.3 was wrong. | M |
| V7.5 Rev. E | 2.3 | Added data flow diagrams as reference documents: /b/ and /c/ | M |
| | 5 | Description of HART_ACTUATOR / HART_TRANSMITTER added.<br>Description of demo applications in delivery package adapted. | A/M |
| | 6.2.1 | API functions modified according to actuator enhancement | M |
| | 7.1.4 | Removed “there are no interrupts in the system” because there are some.<br>Added comment about additional runtimes caused by UART interrupt service routines.<br>Table 5, Section 2: Changed formulation. Only reception, not transmission | M |
| | 7.2.1 | hvars_rom.device_variables_rom[].analog_channel_flags:<br>not restricted to ANALOG_CHANNEL_FLAG_OUTPUT anymore | M |
| | 7.2.2 | Formulation for transmitter variables modified.<br>Description for actuator variables added | M |
| | 7.2.3<br>7.2.4 | Splitted chapter into transmitter (existing) and actuator variant (new).<br>Added data flow diagram also for actuator variant. | M |
| | 7.2.6 | Updated description of DV Trimming according to actuator enhancement | M |
| | 7.3 | Renamed chapter “Current loop” to “Loop current” | M |
| | 7.3.1 | Added description for updating the loop current for actuator variant | M |
| | 7.3.2 | Added description for trimming the loop current for actuator variant | M |
| | 7.3.3 | Added separate chapter for loop current limiting (moved from chapter “Error states / PV Alarm Code”) | M |
| | 7.3.4 | Updated description of error states and PV alarm code according to actuator enhancement | M |
| | 7.4.2 | Bugfix in description of DEVICE_SPECIFIC_STATUS_UNDEFINED_MAP_2:<br>Number of elements must be adapted to<br>COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES instead of always 11. | M |
| | 7.3.3 | Bugfix in description of<br>- DEVICE_SPECIFIC_STATUS_MAPPING_FAILURE_2<br>- DEVICE_SPECIFIC_STATUS_MAPPING_FUNCTION_CHECK_2<br>- DEVICE_SPECIFIC_STATUS_MAPPING_OUT_OF_SPECIFICATION_2<br>- DEVICE_SPECIFIC_STATUS_MAPPING_MAINTENANCE_2<br>Number of elements must be adapted to<br>COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES instead of always 11. | M |
| | 8.2 | Sentence added about possibility to add additional factory commands. | M |
| | 9.2 | Bugfix: The following parameters are not in the substructure device_variables_rom[].<br>They are directly in the structure hvars_rom:<br>device_type, device_revision, software_revision, hardware_revision,<br>physical_signaling_code, flag_assignments, manufacturer_identification_code,<br>privat_label_distributor_code, device_profile | M |
| | 10.6 | Chapter added for description of assert macro | A |
| | 12 | New diagram and description of project organization. | M |
| | 13.2.1 | Updated description of module process.c/.h.<br>Module loopcurrent.c/.h added (only for actuator variant). | M |
| | 13.2.2 | Added definition of actuator demo variant of command 128 | M |
| V7.5 Rev. F | 7.2.1 | Description of new parameters reported_lower_transducer_limit and reported_upper_transducer_limit added. | A |

| | | | |
|---|---|---|---|
| V7.5<br>Rev. G | 6.2.1.1 | Added application interface functions:<br>- HART_USER_DATA_FLASH* hart_getUserDataPtr(void);<br>- void hart_markNonVolatileDataModified(void);<br>- h_uint8_t hart_getRequestBytecount(void);<br>- void hart_resetHartIfaceToDefaults(void); | M |
| | 6.2.1.2 | Removed function hart_checkUnitCodeSiCompliance() from stack interface.<br>Added function hart_resetUserData() to application interface | M |
| | 7.1.1 | Added hart_resetHartIfaceToDefaults() description added. | M |
| | 7.10.5 | Removed explanation of function to check SI compliance by application and added definition of hvars_rom.device_variables_rom[].si_unit_code'. | M |
| | 8.2 | Changed handling of special factory commands handling | M |
| | 8.3.3 | Added description of 'hart_getRequestBytecount()' | M |
| | 9.3 | Added chapter for user defined data. | A |
| | 13.2.2 | Adopted test command to include set of sample data via HART | M |
| V7.5<br>Rev. H | 6.2.1.1 | Removed parameter "more_status_available" from function hart_updateStatusData(). | M |
| | 6.2.1.2 | The function hart_checkPvAlarmCodeAndGetAlarmValue() is split up into hart_setPvAlarmCode() and hart_getPvAlarmValue(). | M |
| | 7.3.2 | Updated description of LOOP_CURRENT_TRIM_MAX_DIFF according to new trim algorithm. | M |
| | 7.3.3 | Add LOOP_CURRENT_MIN_CMD40 and LOOP_CURRENT_MAX_CMD40 definition and description. | M |
| | 7.3.4 | Updated description of new interface for PV alarm code and alarm value. | M |
| | 7.4.2 | Updated description of more status available flag handling and new parameter more_status_available_status_mask[]. | M |
| V7.5<br>Rev. I | 6.2.1.1 | Updated API function list | M |
| | 7.2.2 | Added description of new API function hart_getRealDvValueAndStatus() | A |
| | 7.3.2 | Description of new configuration parameters LOOP_TRIM_MAX_DIFF_SETPOINT and LOOP_TRIM_MAX_DIFF_ERROR. | M |
| | 7.4.1 | More status available bit is handled by stack (not by application any more) | M |
| | 7.4.2 | Updated description of More status available bit handling | M |
| | 7.7 | Add description of new configuration parameter WRITE_PROTECTION_ENABLED | M |
| | 7.10.2 | Add hint that application has to ensure that all NV data write actions are finished. | M |
| | 8.2 | Changed formulation for better understanding | M |
| | 8.2.1 | Changed response data structure: magic pattern is returned instead of dummy bytes | M |
| | 9.1<br>9.2 | Add description of handling of device identity parameters. New API functions described and therefore removed corresponding ROM parameters. | M |
| V7.6<br>Rev. A | 7.2.7 | Add description of new parameter hvars_rom.device_variables_rom[].multiple_unit_codes_allowed | A |
| V7.6<br>Rev. B | 1.1 | Corrected stack release notes of V7.6 Rev. A: Formulation in item "Reject command #31" was not correct. | M |
| | 3.2 | Changed HART Version from 7.5 to 7.6 | M |
| | 8.2 | Described exact match of request byte count for non-public commands. | M |
| | 8.3.4 | Corrected description: Only one DR command callback function is called every 20ms. | M |
| | 12 | Corrected project organization (files and folders) | M |
| | 13.2.1 | Corrected C module overview, added new modules iso3166 and unit_convert | M |
| | 13.2.2 | Response of DR test command (cmd #129) contains 2 bytes (not only 1 byte). | M |
| V7.6<br>Rev. C | 2.2 | Updated references to the HART V7.6 specification | M |
| V7.6<br>Rev. D | 7.4.2 | - Added status bit "Battery or Power Supply needs Maintenance"<br>- Marked bits which are not implemented but defined in the specification with "not implemented" instead of "undefined"<br>- Removed obsolete parameters EXT_DEVICE_STATUS_APPLI_UNDEFINED_MAP and STANDARDIZED_STATUS_0_APPLI_UNDEFINED_MAP | M |
| V7.6<br>Rev. E | --- | No changes in integration manual. | --- |
| V7.6<br>Rev. E | 7.1.1 | Adopted description of using hart_loop() | M |
| | 7.1.4 | Adopting the maximum processing time of HART command functions | M |
| | 8.3.1 | Adopted description of adding commands | M |
| V7.7<br>Rev. A | 8.3.1.1 | Description of performing commands in slices | A |
| | 15.1 | Added Command #534 in Table 13 | M |
| Ver 7.9<br>Rev. A | | Abbreviation HCF replaced by FCG (wherever needed) | M |
| | 1.2 | Reference link updated | M |

| | | | |
|---|---|---|---|
| | 2.1 | Reference link updated | M |
| | 2.2 | This manual is based on the current version of the protocol specification. | M |
| | 3.2 | The HART stack supports HART Version 7.9 | M |
| | 6.1 | Reference link updated | M |
| | 6.8.2 | Add information about packed struct | M |
| | 6.11 | Write a note that digital write protection is currently not supported | M |
| | 6.14.2 | Add information that command 42 is disabled due device reset must not to be used in new designs | M |
| | 11 | Replace figure (STM32CubeIDE replaces KEIL-IDE) | M |
| | 12.1 | Replace figure of development environment | M |
| | 12.2.2 | Add note that device specific test commands 129, 130 and 64768 are typically disabled | M |
| | 14.2 | Add abbreviation FCG | M |

# 2 Scope

## 2.1 Purpose of document

This integration manual provides all necessary information to integrate the MESCO HART Slave Stack into existing microcontroller applications. Full documentation of the API is provided within /a/. Furthermore, the stack architecture is described and the extension possibility for device specific commands is described.

## 2.2 HART Communication Protocol Specification

| Ref | Doc-Identification | Document Title | Date yyyy-mm-dd | Version / Release |
|---|---|---|---|---|
| /1/ | HCF_SPEC-13 | HART Communication Protocol Specification | 2023-01-06 | **7.9** |
| /2/ | HCF_SPEC-54 | FSK Physical Layer Specification | 2022-10-06 | 9.1.1 |
| /3/ | HCF_SPEC-81 | Token Passing Data Link Layer Specification | 2019-12-03 | 9.1 |
| /4/ | HCF_SPEC-99 | Command Summary Specification | 2023-01-05 | 11.0 |
| /5/ | HCF_SPEC-127 | Universal Command Specification | 2020-06-22 | 7.2 |
| /6/ | HCF_SPEC-151 | Common Practice Command Specification | 2023-01-05 | 13.0 |
| /7/ | HCF_SPEC-183 | Common Tables | 2023-04-05 | 27.0 |
| /8/ | HCF_SPEC-307 | Command Response Code Specification | 2023-02-13 | 7.0 |

**Table 1: HART Communication Protocol Specification**

## 2.3 Referenced documents

| Ref | Doc-Identification | Document Title |
|---|---|---|
| /a/ | HartStack_Doxygen_API_Reference.pdf | Doxygen API Reference Documentation |
| /b/ | HartStack_DataFlowDiagram_Transmitter.pdf | Detailed data flow diagram for transmitter variables<br>(only available in full delivery package of the HART stack) |
| /c/ | HartStack_DataFlowDiagram_Actuator.pdf | Detailed data flow diagram for actuator variables<br>(only available in full delivery package of the HART stack) |

**Table 2: Referenced documents**

# 3 Technical data summary

## 3.1 System requirements

| | |
|---|---|
| Data memory (RAM) | 1KB – 2KB (depending on enabled functionality) |
| Program memory (FLASH) | 20KB – 25KB (depending on enabled functionality) |
| Microcontroller | For HW modem variant: 8-bit / 16-bit / 32-bit controller >= 3MHz<br>For soft modem variant: 32-bit controller >=4MHz |
| Hardware peripherals | 1 timer used as system time counter<br>For HW modem variant: 1 UART with parity generator / checker and framing error checker + 2 GPIOs |

**Table 3: System requirements**

## 3.2 Supported functionality

- FCG compliant (various devices realized)
- HART Version 7.9
- Backwards compatible to HART 5 and HART 6 masters
- Expanded manufacturer codes
- Time stamped data
- Multi-drop mode
- Additional Device Status (Cmd #48)
- Delayed Response commands possible
- All universal commands implemented
- Common practice commands implemented
    - Extended burst mode
        - Multiple individual configurable burst messages
        - Individual trigger conditions
    - Command aggregation
        - For normal commands and for burst messages
    - Dynamic variable mapping
    - Trim support for loop current and device variables
    - Supports real-time clock (RTC) or emulates RTC
    - Supports Lock Device mechanism
    - Supports write protection mechanism
    - Communication statistics
- Easy integration
    - Tiny & efficient code: Runs on almost any controller (8/16/32 bit)
    - Supports various integration scenarios, with or without operating system (RTOS): Main Loop, Timer interrupt driven, or RTOS task handler
    - Supports external HART Modem or optional soft modem (Soft modem': HART modem realized by software and ADC / DAC)
    - Easy device specific parameterization
    - Well defined hardware interface for platform customization
    - API for device specific HART commands (user extensible)
    - Flexible API for non-volatile data

# 4 Stack architecture

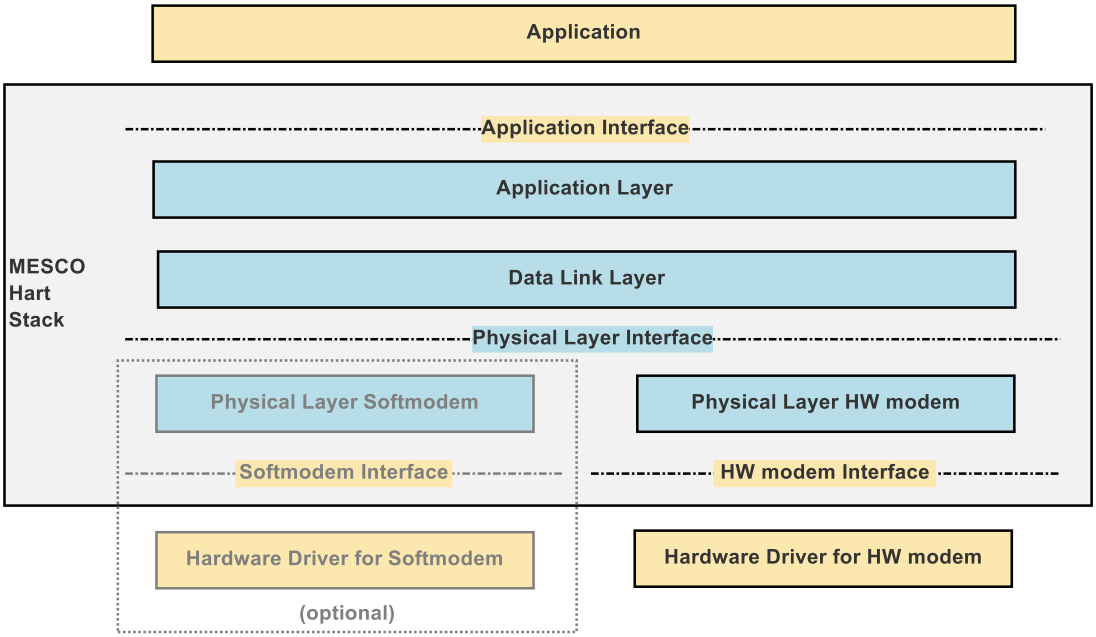



**Figure 1 Stack layer model**

The layers correspond to the layers of the OSI model:

- Application Layer (OSI layer 7)
- Data Link Layer (OSI layer 2)
- Physical Layer (OSI layer 1)

The Application is the system software that integrates the HART Slave Stack. This term is used throughout this document.

The Application Layer is the stack's top layer. The Application Layer implements control routines as well as the stack's command handlers and related high-level functionality, whereas the Data Link Layer is the connection between Application Layer and the Physical Layer. The generic part of MESCO HART Slave Stack consists of the Application Layer and the Data Link Layer.

The physical layer is responsible for writing the HART messages to the physical hardware and reading response messages from it. The HART standard defines various Physical Layers. MESCO provides implementation only for the FSK physical layer. The following two options are available:

- The HW modem variant works with an external FSK modem chip. If this option is used, the HW modem interface has to be served by the user.
- The soft modem variant implements an FSK modulator and demodulator in software using an ADC and a DAC. If this variant is used the soft modem interface has to be served by the user.

The hardware drivers are hardware specific and not part of the generic stack. MESCO provides support and engineering services for porting the existing sample-drivers to new hardware platforms.

# 5 Stack configuration

In order to customize the HART Slave Stack to the user's needs it is necessary to configure the stack according to the required functions for the target device. The configuration is done via customizable files that have to be adapted by the user. The configuration is done in definitions (#define) and read-only memory variables.

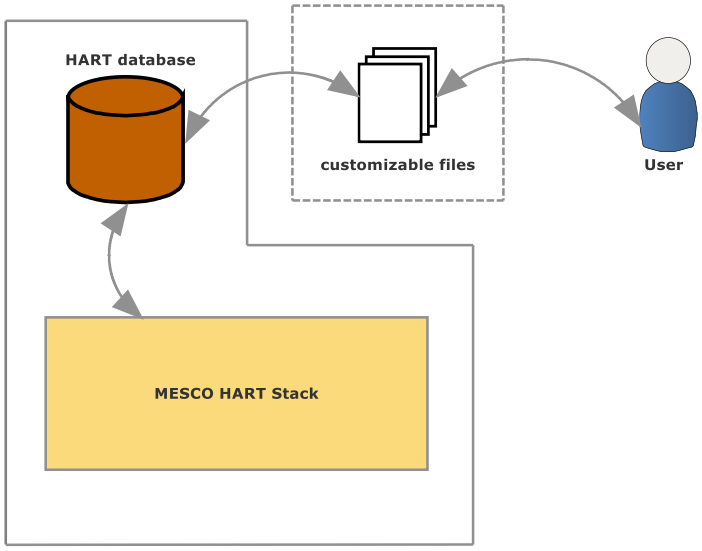



**Figure 2 Stack configuration user interaction**

The customizable parameters and definitions are divided into the following parts:

- Compiler switches and definitions (#define) in hart_config.h
- Standard type definitions in hart_config.h
- ROM Parameters in hart_parameters.c – const structure hvars_rom

The customization of the HART interface and the customization of the stack to the target hardware and compiler are described in the following chapters "8 HART device customization" and "9 Target/compiler customization".

There are two demo applications included in the delivery package: One for transmitter- and one for actuator-devices. The demo applications provide default values for all parameters and definitions to demonstrate a working HART slave device. The default values mentioned in this document are the values of the transmitter demo application.

| **Name** | **#define HART_ACTUATOR / #define HART_TRANSMITTER** |
|---|---|
| Item type | Definition in in hart_config.h |
| Data type | n/a |
| Default value | n/a |
| Valid range | n/a |
| Description | The two defines decide whether the stack implements a HART actuator or transmitter. Exactly one of the two defines must be active. The selection has influence on the application interface and on the data flow within the stack. |

# 6 Stack interfaces / API

## 6.1 Overview

The following figure shows the internal software structure of the HART Slave Stack. The corresponding interfaces are described in the following chapters.

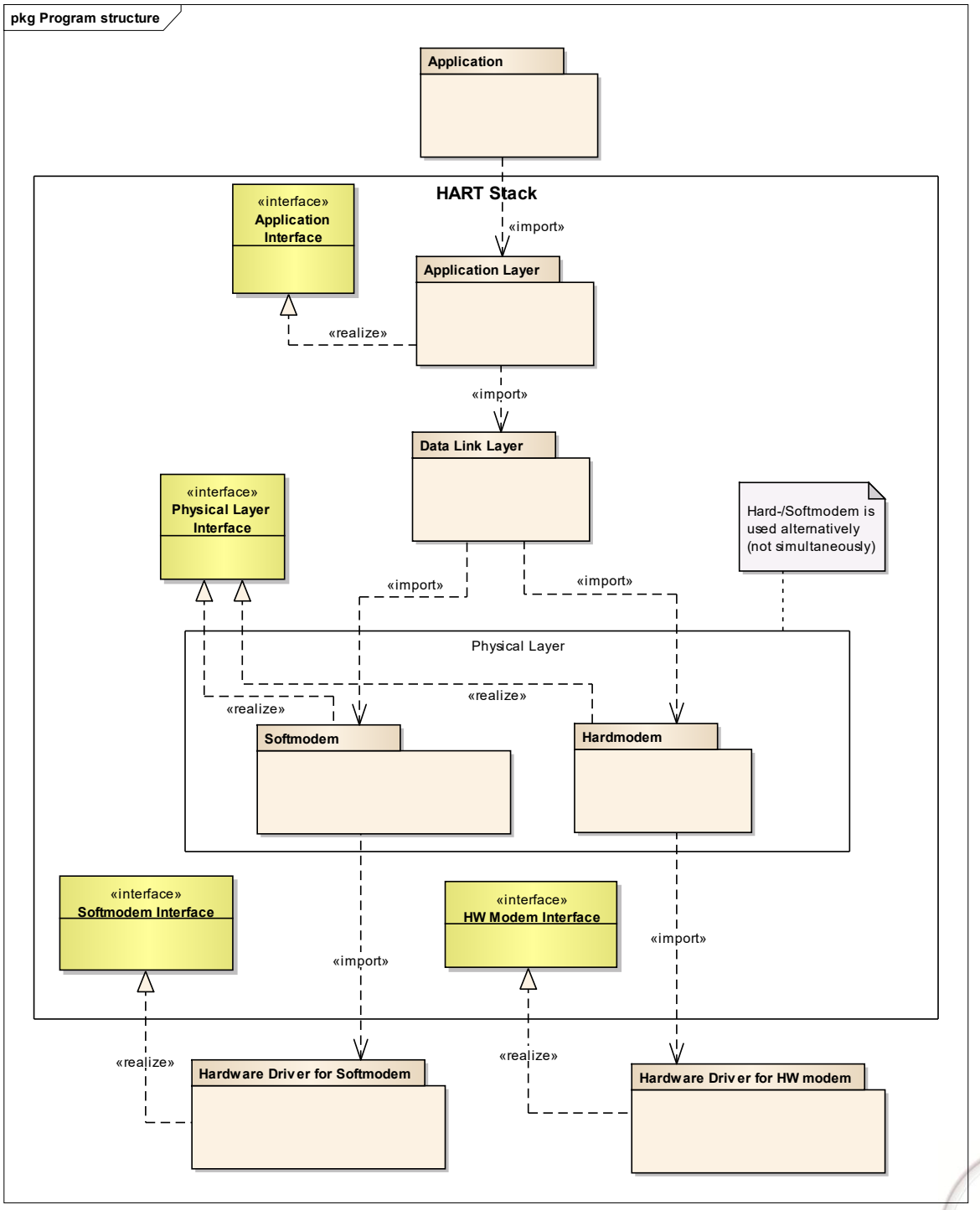



**Figure 3 Stack interfaces overview**

## 6.2 Application Interface

The Application interface is defined in the following 3 header files:

- al/hart_stack_iface.h API functions implemented by Application, called by Stack
- al/hart_appli_iface.h API functions implemented by Stack, called by Application
- al/hart_iface_types.h Generic definitions and types of the Application interface

### 6.2.1 API Functions…

#### 6.2.1.1 …implemented by the stack, called by the application

The following functions are implemented by the HART Slave Stack and called by the Application. Their prototypes are defined in the file hart_appli_iface.h and the functions are implemented in the file hart_hap.c. See /a/ for the detailed definitions of the functions (i.e. function parameters, return values, descriptions).

- void **hart_init**(void);
- h_uint8_t **hart_loop**(void);
- void **hart_setWriteProtectMode**(h_bool_t bWriteProtected);
- h_bool_t **hart_updateTransmitterDvAndStatus**(h_uint8_t dv_code, h_float32_t value, h_uint8_t status);
- h_bool_t **hart_updateActuatorDvStatus**(h_uint8_t dv_code, h_uint8_t status);
- void **hart_updateActuatorLoopCurrent**(h_float32_t loop_current_appli);
- h_bool_t **hart_getRealDvValueAndStatus**(h_uint8_t dv_code, h_float32_t *value, h_uint8_t *status);
- void **hart_setDeviceStatusMalfunction**(h_bool_t set);
- void **hart_setAlarmState**(h_bool_t set);
- void **hart_updateStatusData**(STATUS_DATA_T *p_appl_data);
- void **hart_getStatusData**(STATUS_DATA_T *p_status_data);
- DELAYED_RESPONSE_STATE **hart_checkDelayedResponse**(void);
- DELAYED_RESPONSE_STATE **hart_registerDelayedResponse**(DELAYED_RESPONSE_CALLBACK callback);
- GLOBAL void **hart_printResponseByte**(h_uint8_t data);
- void **hart_printResponseWord**(h_uint16_t data);
- void **hart_printResponseLongword**(h_uint32_t data);
- void **hart_printResponseFloat**(h_float32_t data);
- void **hart_printResponseArray**(h_uint8_t const * p_data, h_uint8_t cnt);
- void **hart_printResponseTime**(HART_TIME_MS time_ms);
- h_uint8_t **hart_readRequestByte**(h_uint8_t const idx);
- h_uint16_t **hart_readRequestWord**(h_uint8_t const idx);
- h_uint32_t **hart_readRequestLongword**(h_uint8_t const idx);
- h_float32_t **hart_readRequestFloat**(h_uint8_t const idx);
- void **hart_readRequestArray**(h_uint8_t idx, h_uint8_t *p_data, h_uint8_t cnt);
- HART_TIME_MS **hart_readRequestTime**(h_uint8_t const idx);
- HART_USER_DATA_FLASH* **hart_getUserDataPtr**(void);
- void **hart_markNonVolatileDataModified**(void);
- void **hart_setIdentityDeviceID**(h_uint8_t device_id[]);
- void **hart_setIdentityDeviceRevision**(h_uint8_t device_revision);
- void **hart_setIdentityHwRevision**(h_uint8_t hardware_revision);
- void **hart_setIdentityPrivateLabelDistributorCode**(h_uint16_t private_label_distributor_code);
- h_uint8_t **hart_getRequestBytecount**(void);
- void **hart_resetHartIfaceToDefaults**(void);

#### 6.2.1.2 …implemented by the application, called by the stack

The following functions have to be implemented by the application and are called by the HART slave stack. Their prototypes are defined in the file hart_stack_iface.h and the functions have to be implemented in any application specific file. See /a/ for the detailed definitions of the functions (i.e. function parameters, return values, descriptions).

- void **hart_performSelftest**(void);

- void **hart_performDeviceReset**(void);
- h_bool_t **hart_squawk**(h_uint8_t squawk_control);
- h_bool_t **hart_getFindDeviceArmedState**(void);
- void **hart_writeNvData**(h_uint8_t *p_nonvol_mirror_data, h_uint16_t nonvol_size);
- h_bool_t **hart_readNvData**(h_uint8_t *p_nonvol_mirror_data, h_uint16_t nonvol_size);
- HART_TIME_MS **hart_getCurrentTime**(void);
- void **hart_setRTC**(h_uint8_t day, h_uint8_t month, h_uint8_t year, h_uint32_t time_ms);
- void **hart_getRTC**(h_uint8_t *day, h_uint8_t *month, h_uint8_t *year, h_uint32_t *time_ms);
- void **hart_updateActuatorDvValue**(h_uint8_t dv_code, h_float32_t value);
- void **hart_updateTransmitterLoopCurrent**(h_float32_t loop_current_real, h_float32_t loop_current_corrected);
- h_bool_t **hart_setDampingTau**(h_uint8_t DevVarID, h_float32_t *damping_tau);
- h_bool_t **hart_setPvAlarmCode**(h_uint8_t alarm_code);
- h_float32_t **hart_getPvAlarmValue**(void);
- h_bool_t **hart_setCountryCode**(h_uint8_t country_code[COUNTRY_CODE_SIZE]);
- h_bool_t **hart_checkDvUnitBlocked**(h_uint8_t dv_code, h_uint8_t unit_code);
- h_bool_t **hart_checkDvUnitConvert**(h_uint8_t unit_code_from, h_uint8_t unit_code_to);
- h_float32_t **hart_convertDvValue**(h_float32_t value, h_uint8_t unit_code_from, h_uint8_t unit_code_to);
- void **hart_resetUserData**(void);

## 6.3 Softmodem Interface

*The Softmodem Physical Layer is currently under development.*

## 6.4 HW Modem Interface

This chapter describes the HW Modem Interface. The interface has to be implemented by the application to communicate with the HART modem (e.g. Analog Devices AD5700). The following figure shows the connection between the microcontroller and the HART modem.

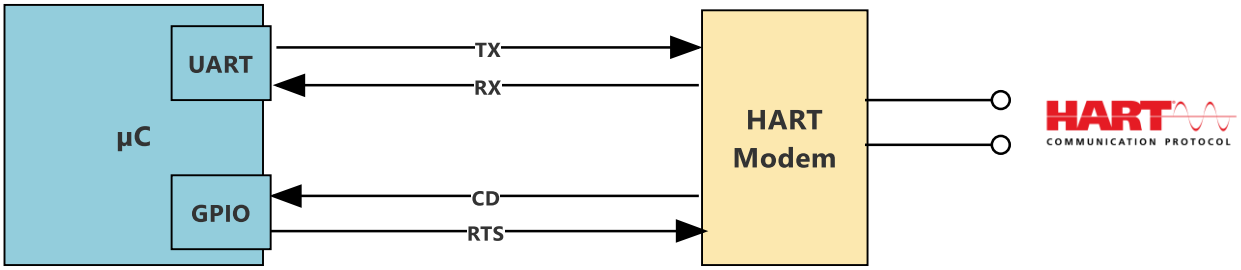



**Figure 4 HART modem HW interface**

| Name | Description |
|---|---|
| TX | Transmit line to send data (UART TX) |
| RX | Request line to receive data (UART RX) |
| CD | Carrier detect input for carrier active indication |
| RTS | Request to send line to activate the modem modulator |

The HW Modem Interface is defined in the file hart_hwmodem_driver.h. See /a/ for the detailed definitions of the functions (i.e. function parameters, return values, descriptions). The following figure shows the function calls between the HART Slave Stack and the HW modem driver which has to be implemented by the user.

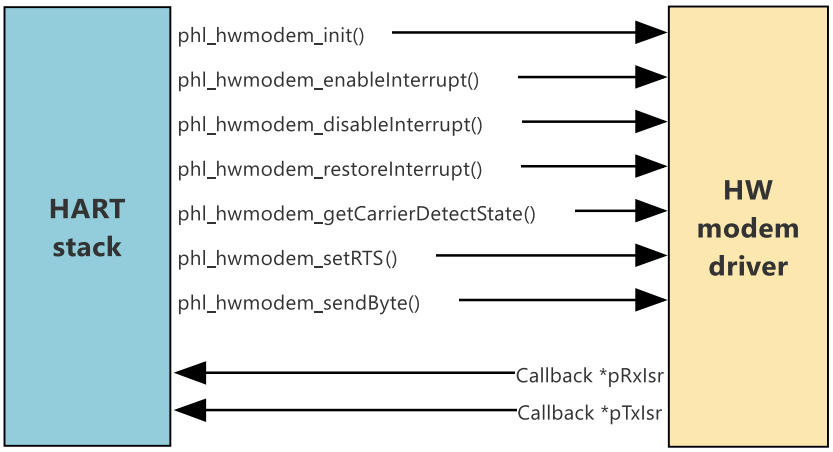



**Figure 5 HW modem interface**

The HW modem driver is initialized by the stack by calling phl_hwmodem_init().

The interface functions for the signals RTS and CD are simple GPIO access functions. The stack calls the function phl_hw_modem_setRTS() for setting the RTS signal to active or inactive state. The stack calls the function phl_hw_modem_getCarrierDetectState() for reading the CD signal state.

For communication with the HW modem a standard UART peripheral with the following 3 interrupts is necessary.

- Receive Data Ready Interrupt (RX)
- Transmit Data Register Empty Interrupt (TX ready)
- Transmission Complete Interrupt (TX complete)

For the activation control of the interrupts the stack calls the functions phl_hwmodem_enableInterrupt(), phl_hwmodem_disableInterrupt() and phl_hwmodem_restoreInterrupt().

For the execution of the Interrupt Service Routines the HART Slave Stack provides two function pointers at startup (parameters of function phl_hwmodem_init()). One for the RX-interrupt and one for the two TX-interrupts. If an active interrupt occurs, the HW modem driver has to call the corresponding callback function in the stack.

For sending a message the stack activates the RTS signal. After 5-bit times (but not before the next hart_loop() call) the stack enables the TX ready interrupt. At this moment the UART is idle and the transmit register is empty, so the interrupt occurs immediately. Now sending single bytes is done within the TX callback function by calling phl_hwmodem_sendByte().

When the last byte of the command is sent, the TX ready interrupt is disabled, and the TX complete interrupt is enabled. In the TX complete interrupt-routine the RTS signal is switched off again.

For receiving a message, the arriving bytes cause RX interrupts. So, the RX callback function is called by the HW modem driver and the bytes are stored into the receive buffer including the error information flags. When the message is received completely, it is processed and the answer is generated in the next hart_loop() call.

The UART must be configured as follows: 1200 baud, 1 start bit, 8 data bits, 1 odd parity bit, 1 stop bit.

The UART must be able to detect parity and framing errors (=stop bit errors).

The following figure shows the sequence diagram for a transmission of a complete HART message.

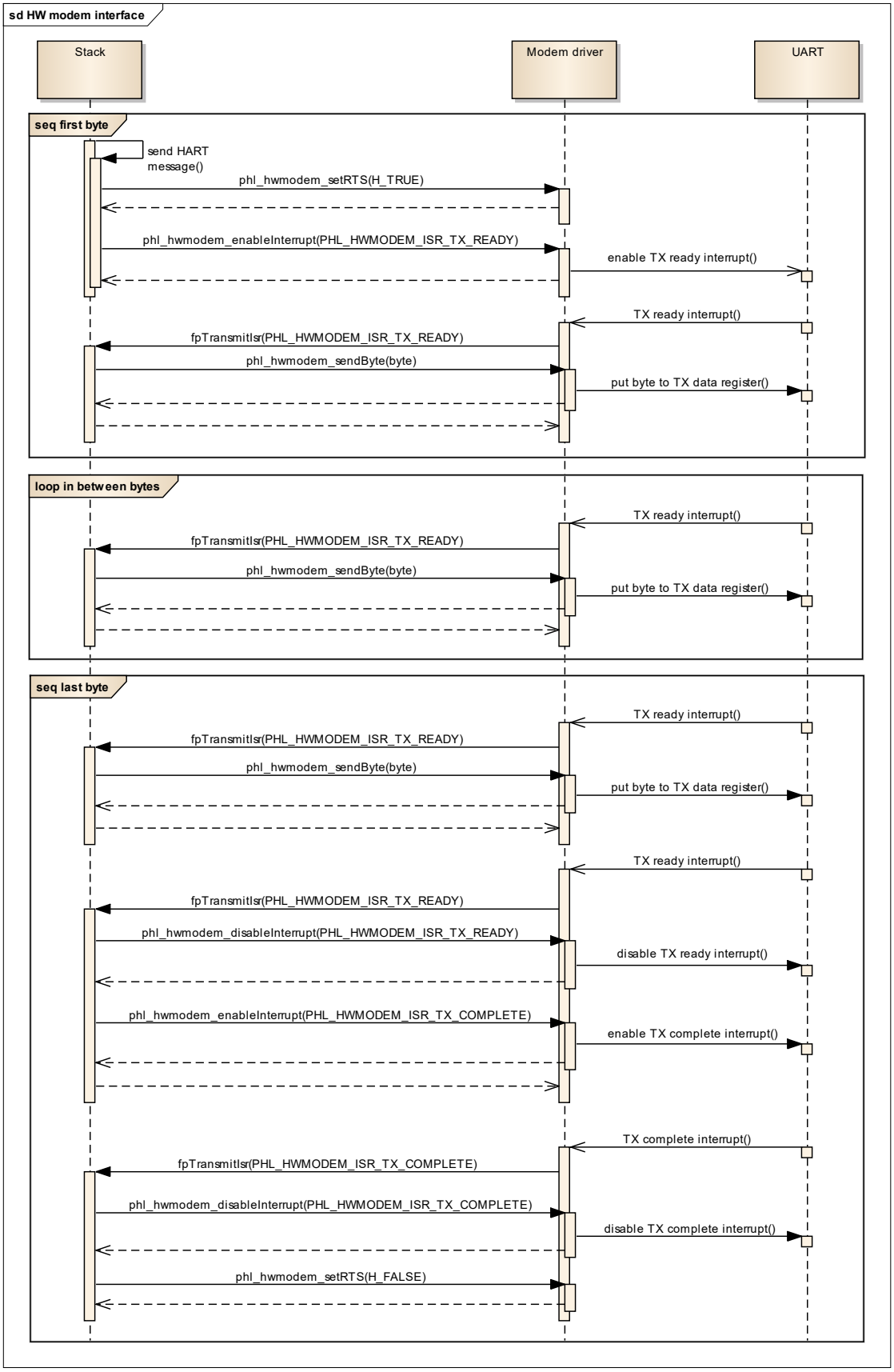



**Figure 6 Sequence diagram – transmission of HART message**

**Integration issues**

The following chapters point out integration issues related to the Application Interface.

## 6.5 Scheduling / timing

### 6.5.1 Operating system integration

The operating system has the following main entry points to call the HART Slave Stack.

**hart_init():**

This function shall be called at startup before any other routine of the stack is called. It initializes the stack.

**hart_loop():**

This function is the heartbeat function of the stack. It shall be called by the application on a periodic basis (period time <= 10 ms). It processes the received HART message and generates the response message.

If “HART_LOOP_ERROR” is returned, a buffer over-/underrun happened in the stack. This should never happen. A possible reason is that the call of the function hart_loop() was too late. It's up to the application to handle this fatal error. The stack tries to continue normal communication after this error.

**hart_resetHartIfaceToDefaults():**

In some situations, it may be useful for the application to reset all stack parameters to the default values. After the call of this function, all non-volatile data is set to default, and the device performs a reset (Refer 6.14.2).

### 6.5.2 System timer integration

The HART Slave Stack needs time information for internal timing calculations (e.g. measuring timeouts). It calls hart_getCurrentTime() to read the current system time from the application.

The system time value must be a 32-bit integer value which is incremented every millisecond and wraps around from 0xffffffff to 0x00000000. The absolute value has no meaning.

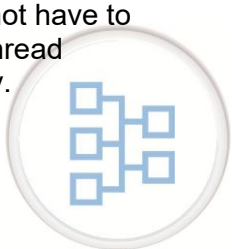
### 6.5.3 Multi-threading considerations

Since the HART Slave Stack runs in a single thread (hart_loop() thread) all API functions that are called from the HART Slave Stack are called from this thread.

hart_loop() can be integrated in different ways:

a) hart_loop() is called in the systems main loop. A round robin scheduler may be used to switch between hart_loop() and other tasks.

b) hart_loop() is called from a timer interrupt. Other tasks are running in the processor's main loop. This approach has the advantage that the design of the main loop does not have to consider the timing constraints of the HART Slave Stack. On the other hand, thread synchronization mechanisms may be necessary to guarantee data consistency.



### 6.5.4 Real-time considerations

The following figure shows the timing of the MESCO HART® slave stack during a STX/ACK communication between HART-Master and HART-Slave. RX / TX show the UART communication signals between the HART modem and the HART-Slave microcontroller. “Testpin” is a GPIO test output controlled by a special test variant of the HART Slave Stack firmware. The test pin is set when function hart_loop() is entered and reset when the function is left, so the signal can be used to measure the execution time of the function hart_loop().

There are several sections during command processing with different CPU load. Furthermore, execution time of section 3 depends on the executed command and is only a peak load which occurs once for each command processing.

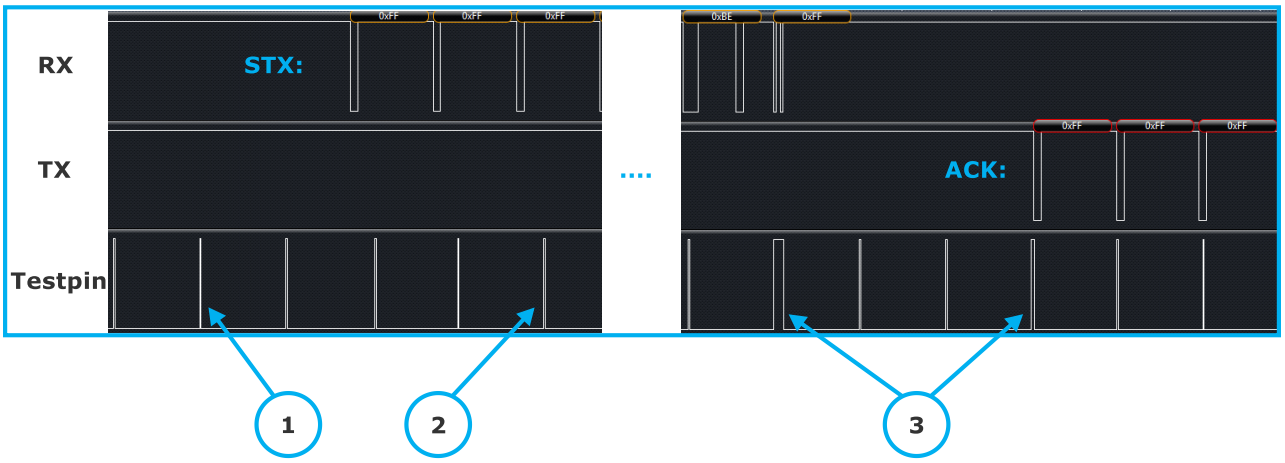



**Figure 7 Execution time of HART Slave Stack**

| Section | Executed Command |
|---|---|
| 1 | HART Slave Stack is idle |
| 2 | HART Slave Stack is receiving a byte |
| 3 | HART Slave Stack processes a command |

**Table 4: Execution time sections**

The following table shows the timing measurement results for the described sections. Some default and worst-case scenarios / commands are used to measure the execution time of the hart_loop(). The measured times depend on the used compiler and its settings. Of course, depending on the HW modem interface implementation there are additional runtimes caused by the UART interrupt service routines.

Information about the HW setup used can be read in chapter “12.1 Development Environment.” The following compiler setup is used for the measurements:

- ARM none-eabi-gcc
  - No code optimization “-O0”

| Section | Executed Command | Scenario description | Time in ms @8.0 MHz SysCore clock | Approx. CPU cycles |
|---|---|---|---|---|
| 1 | No command | Idle state of the Hart Slave Stack. Executed at least every 10ms | 0.231ms | 1848 cycles |
| 2 | Don't care | Reception and processing of one single request byte. | 0.242ms | 1936 cycles |
| 3 | CMD 0 | This is normal read operation. Some parameters are read by the Master. | 0.328ms + 0.355ms | 5464 cycles |
| 3 | CMD 78 | Command 78 includes four commands (14, 15, 4, and 38). The byte count for all commands is 0, with no request data bytes for any of them. | 0.361ms + 0.389ms | 6000 cycles |
| 3 | Burst of Command 9 | A burst command produces also CPU load depending on the bursted command. In this case Cmd #9 with 4 slots is bursted. | 0.347ms + 0.363ms | 5680 cycles |
| 3 | Burst of Command 9 | A burst of Cmd #9 with 8 slots increases the CPU load. | 0.347ms + 0.356ms | 5624 cycles |

**Table 5: Process time of HART Slave Stack**

However, if time consuming device specific commands are implemented, it must be considered that the hart stack needs processing time for itself. Therefore, the execution time of the command handler functions for device specific commands must not be greater than approx. 200ms.

After a HART command was executed, the non-volatile memory possibly needs to be updated. Updating the non-volatile memory may be time consuming. The update algorithm can be device specific and is therefore not part of the HART Slave Stack. See chapter "6.9 Non-Volatile data handling".

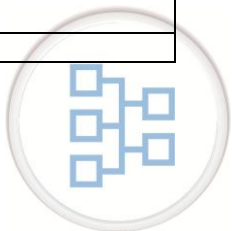
## 6.6 Process values

### 6.6.1 Device Variables definition and parameters

For the definition of a Device Variable (DV) the array hvars_rom.device_variable_rom[] must be extended by an additional item of the type DEVICE_VARIABLE_ROM. The following tables describe the fields of the array item. Furthermore the #define DEVICE_VARIABLE_NUM and DYNAMIC_VARIABLE_NUM must be updated correctly.

| Name | #define DEVICE_VARIABLE_NUM |
|---|---|
| Item type | Definition in in hart_config.h |
| Data type | h_uint8_t |
| Default value | 2 |
| Valid range | 1…8 tested (theoretically unlimited) |
| Description | Number of defined Device Variables |

| Name | #define DYNAMIC_VARIABLE_NUM |
|---|---|
| Item type | Definition in in hart_config.h |
| Data type | h_uint8_t |
| Default value | 3 |
| Valid range | 1…4<br>1=PV<br>2=PV,SV<br>3=PV,SV,TV<br>4=PV,SV,TV,QV |
| Description | Number of defined dynamic variables |



| Name | hvars_rom.device_variables_rom[].classification |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t (enum) |
| Default value | DV_CLASS_NOT_CLASSIFIED (0) |
| Valid range | see HCF_SPEC-183 /7/, common table 21 |
| Description | These codes indicate the function performed by the Device Variable. This allows masters and host applications to identify the type of process connection supported by the Device Variable and the Unit Code. |

| Name | hvars_rom.device_variables_rom[].device_family |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t (enum) |
| Default value | DV_FAMILY_NOT_USED (250) |
| Valid range | see HCF_SPEC-183 /7/, common table 20 |
| Description | This code indicates which family the Device Variable belongs to. If the Device Variable does not support Device Family Commands, then 250 “Not Used” must be returned. |

| Name | hvars_rom.device_variables_rom[].analog_channel_flags |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t (enum) |
| Default value | ANALOG_CHANNEL_FLAG_OUTPUT |
| Valid range | see HCF_SPEC-183 /7/, common table 26 |
| Description | These codes are used to clarify Analog Channel functions. The following values can be used:<br>If set, this Analog Channel is a Field Device analog input channel. If this bit is set Field Device has an ADC connected to this channel. When reset to zero the Analog Channel is an analog output (DAC). Input is only possible if the stack is configured in actuator mode (HART_ACTUATOR). |

| Name | hvars_rom.device_variables_rom[].minimum_span |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = internal unit of DV (see chapter ”6.6.7 Engineering unit conversion”) |
| Default value | 50.0f |
| Valid range | value must be greater than 0.0f and<br>lower than [upper transducer limit – lower transducer limit] |
| Description | The minimum span of a device variable is the minimum difference between Upper Range Value and Lower Range Value. |

The minimum span can be read out with cmds #14 and #54. According to the FCG specification the mentioned commands have to provide the minimum span value in the unit which is currently set for the Device Variable. As described in chapter “6.6.7 Engineering unit conversion” the HART Slave Stack internally always works with the same “internal unit”. Therefore, the minimum span value has to be converted before it is given out. Now the problem is that the minimum span is not absolute but a difference. The conversion is implemented as follows:

$unitA = internal\ unit$
$uintB = requested\ unit$

$$span[unitB] = convert(span[unitA], unitA, unitB) - convert\ (0, unitA, uintB)$$

This is only valid if the internal unit and the requested unit can be linearly converted into each other!

Most slave device implementations have included a small margin above the upper transducer limit and below the lower transducer limit

to prevent minor fluctuations beyond these limits from triggering status bits. Consequently, the stack includes two values for upper and two values for lower transducer limits. The parameter

“reported_upper_transducer_limit” is the one reported by the device and contains the value of the specification of the device. (These values are mainly reported by #54 and #14)

The parameter “upper_transducer_limit” contains the value used by the stack to limit the process values and calculate the status bits for PV out of limits or NON-PV out of limits. With this feature it is possible to allow a slightly bigger range than reported by the device.

| Name | hvars_rom.device_variables_rom[].reported_lower_transducer_limit |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = internal unit of DV (see chapter “6.6.7 Engineering unit conversion”) |
| Default value | -20.0f |
| Valid range | value must be greater or equal than hvars_rom.device_variables_rom[].lower_transducer |
| Description | Reported Lower Transducer limit of the Device Variable. |

| Name | hvars_rom.device_variables_rom[].reported_upper_transducer_limit |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = internal unit of DV (see chapter “6.6.7 Engineering unit conversion”) |
| Default value | 70.0f |
| Valid range | value must be lower or equal than hvars_rom.device_variables_rom[].upper_transducer |
| Description | Reported Upper Transducer limit of the Device Variable. |

| Name | hvars_rom.device_variables_rom[].lower_transducer_limit |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = internal unit of DV (see chapter “6.6.7 Engineering unit conversion”) |
| Default value | -20.9f |
| Valid range | value must be lower or equal than hvars_rom.device_variables_rom[].lower_transducer_trimpoint_min |
| Description | Lower Transducer limit of the Device Variable. |

| Name | hvars_rom.device_variables_rom[].upper_transducer_limit |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = internal unit of DV (see chapter “6.6.7 Engineering unit conversion”) |
| Default value | 70.9f |
| Valid range | value must be greater or equal than hvars_rom.device_variables_rom[].upper_transducer_trimpoint_max |
| Description | Upper Transducer limit of the Device Variable. |

### 6.6.2 Device Variables (DV) value and status updates

For transmitter variables the application has to call the function hart_updateTransmitterDvAndStatus() every time a new Transducer Value is available from the hardware. Note that the function must be called exactly one time after each measurement. Multiple calls between two measurements will cause malfunction of the update failure bit.

Parameters of the function are the new Device Variable value in the internal unit and the Device Variable Status. See HCF_SPEC-183 /7/ chapter A.3 for description of the Device Variable Status bits.

Only the ‘Process data status’ bits (D7…D6) are adopted.

The ‘Limit status’ bits (D5…D4) are handled and overwritten by the HART Slave Stack internally.

The ‘More DV status available’ flag (D3) and the ‘DV Family specific status’ bits (D2…D0) are not supported yet and set to 0.

The update function should be called at the same interval as the update time period (see following table). Sporadic faster update periods are allowed but slower update periods are not allowed.

| **Name** | **hvars_rom.device_variables_rom[].update_time_period** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | HART_TIME_MS; unit = milliseconds |
| Default value | HART_TIME_MS_50MS |
| Valid range | 0…0x7fffffff |
| Description | Acquisition Period indicates the maximum period between Device Variable was updated by the application. |

For actuator variables new values are passed to the application by the HART slave stack using the API function hart_updateActuatorDvValue().

For the DV status the application still has to update the status bits mentioned above using the function hart_updateActuatorDvStatus().

The application can read back the “real” (trimmed) DV value together with the complete DV status (including the bits which are handled by the stack) by calling the function hart_getRealDvValueAndStatus().

### 6.6.3 Data flow overview for transmitter variables

The following figure shows an overview of the data flow for a typical transmitter application. It contains some absolute values as examples for a better understanding of DV Trimming and loop current trimming.

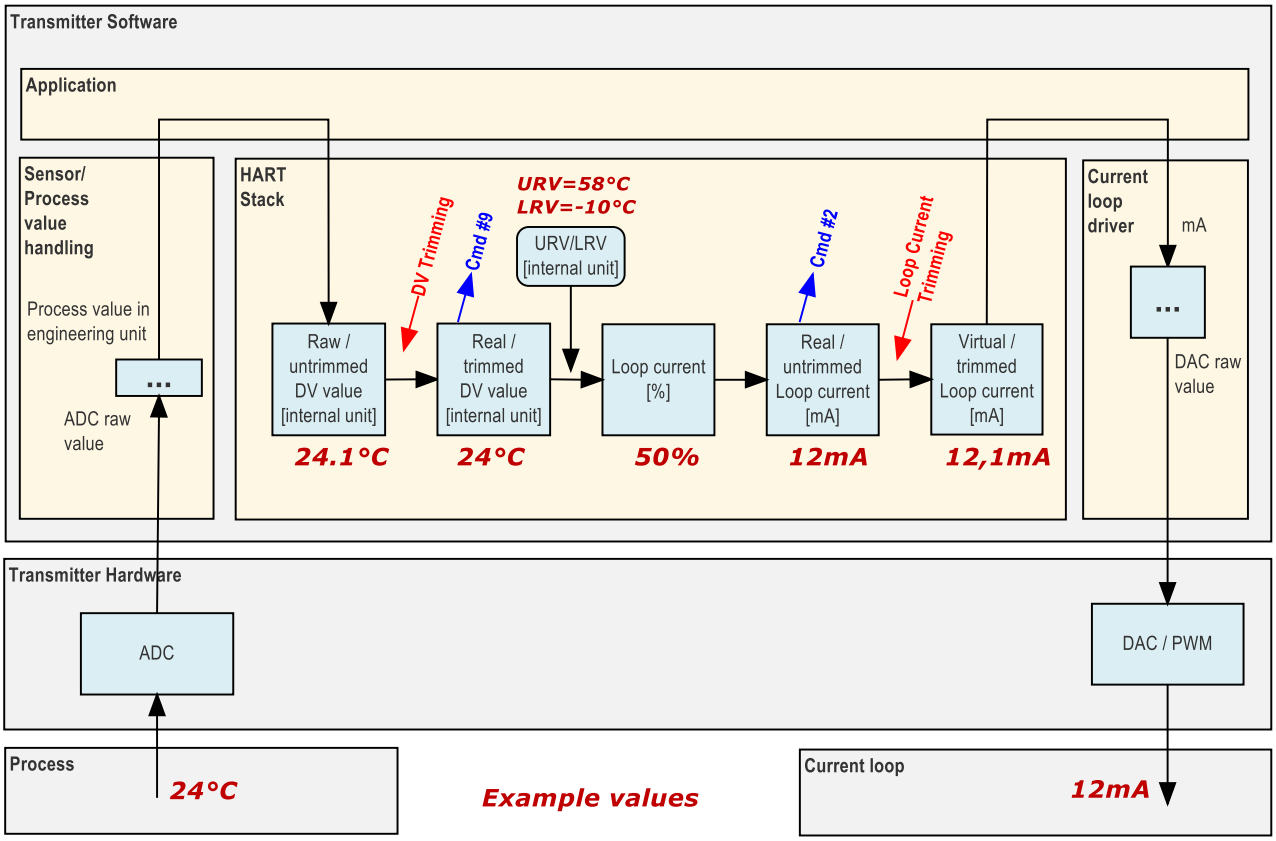



**Figure 8 Data flow overview transmitter**

See /b/ for a detailed data flow diagram of the transmitter Device Variables.

### 6.6.4 Data flow overview for actuator variables

The following figure shows an overview of the data flow for a typical actuator application. It contains some absolute values as examples for a better understanding of DV Trimming and Loop Current Trimming.

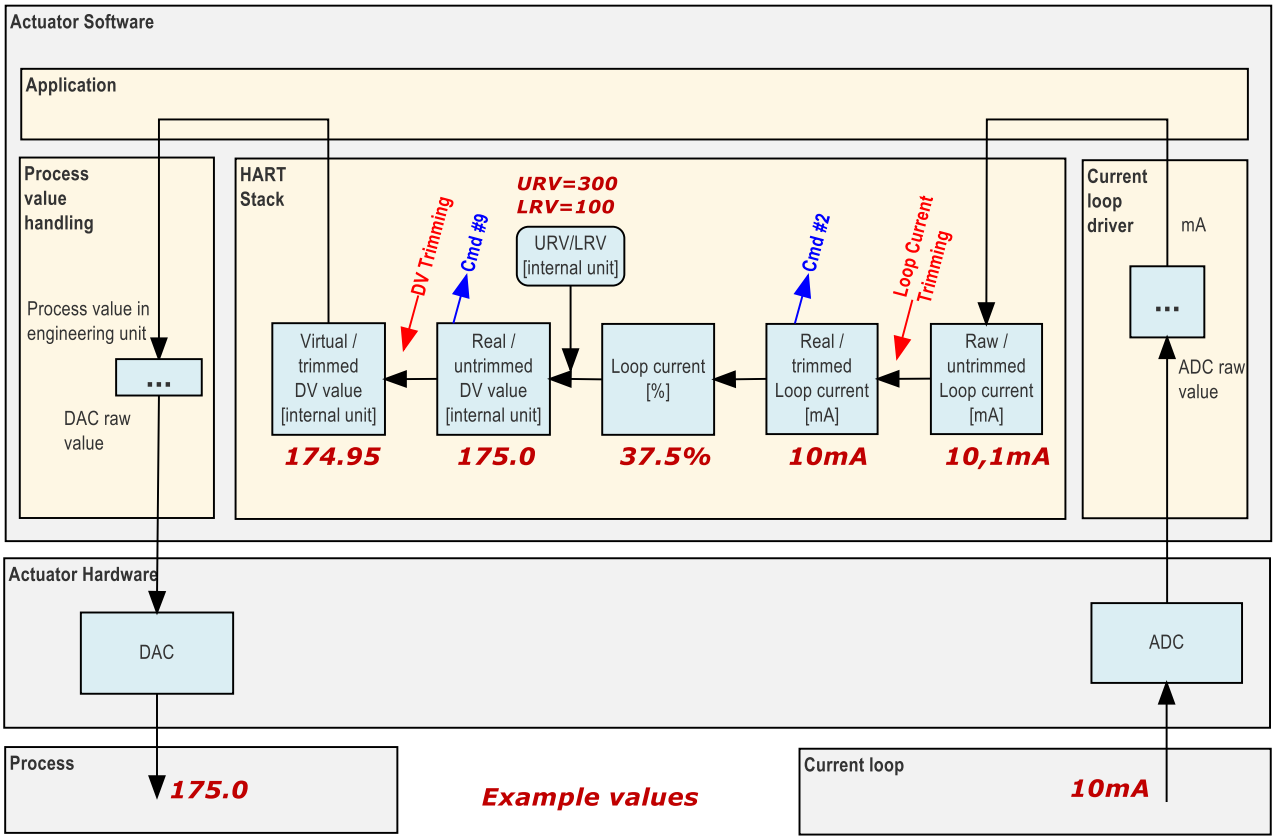



**Figure 9 Data flow overview actuator**

**See /b/ for a detailed data flow diagram of the actuator Device Variables.**

### 6.6.5 DV Damping

The HART Slave Stack has implemented a software damping function for each Device Variable.

The functionality can be enabled or disabled separately for each Device Variable (sw_damping_enable).

If the software damping is enabled the stack handles the damping itself. In this case the API function hart_setDampingTau() is never called.

If the software damping is disabled the API function hart_setDampingTau() is called once on startup, and whenever the damping value changes. In this case, it's up to the application to do the damping of the device variable with the given time constant. Check for upper and lower range of the time constant is still done by the Hart Slave Stack. Therefore, the maximum value for the time constant has to be defined (damping_max). The minimum time constant is always 0.0 (=damping disabled).

| Name | hvars_rom.device_variables_rom[].sw_damping_enable |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_bool_t |
| Default value | H_TRUE |
| Valid range | H_TRUE / H_FALSE |
| Description | Enables/disables the software damping of the HART Slave Stack. |

| Name | hvars_rom.device_variables_rom[].damping_max |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = seconds |
| Default value | 10.0f |
| Valid range | all positive values except of NaN |
| Description | Maximum damping factor to be set by command #34. All higher values are rejected. The minimum damping value is always zero. |

### 6.6.6 DV Trimming

This chapter describes the DV trimming block within the data flow (see chapters “6.6.3 Data flow overview for transmitter variables” and “6.6.4 Data flow overview for actuator variables”.

The HART Slave Stack has implemented a linear two-point trimming for all Device Variables. The following figure shows the mapping function which is implemented for the mapping between the untrimmed value and the trimmed value.

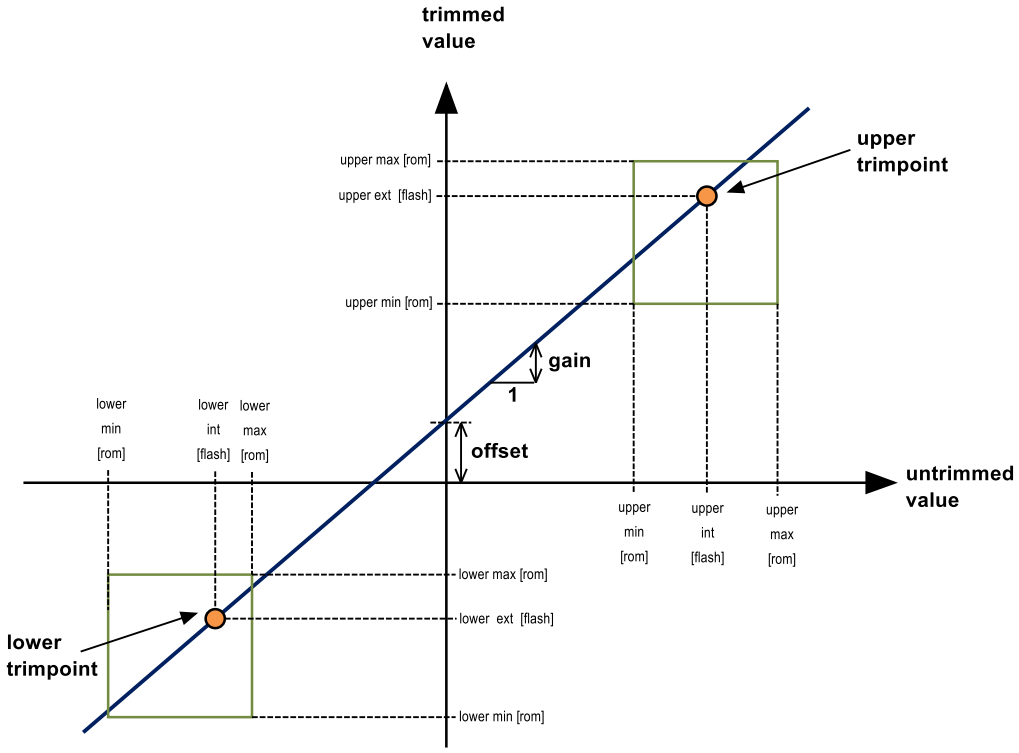



**Figure 10 Trim points**

The trimmed value is calculated as follows:

$$trimmed\ value = gain \cdot untrimmed\ value + offset$$

The trim points can be read out with command #80 and set with commands #43, #52 and #82 (see HCF_SPEC-151 /6/). They are non-volatile variables.

The trim point range values can be read out with command #81 (see HCF_SPEC-151 /6/). They are read-only stack configuration parameters and cannot be changed. See tables below.

The command #82 sets either the upper or the lower trim point. If the upper trim point is set, the lower trim point remains untouched. If the lower trim point is set, the upper trim point remains untouched. In both cases the offset and the gain value is changed. If the requested trim point is out of range, the command is rejected with the corresponding response code.

The commands #43 and #52 adjust only to the offset, so that the actual applied process value equals 0.0. As a result, both trim points are moved parallel to the vertical axis. Thus, both trim points and the offset value are adjusted, while the gain remains untouched. If this “ZERO” operation would cause one of the trim points exceeding the allowed range, the command is rejected with the corresponding response code.

| **Name** | **hvars_rom.device_variables_rom[].lower_transducer_trimpoint_min** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = internal unit of DV (see chapter “6.6.7 Engineering unit conversion”) |
| Default value | -20.0f |
| Valid range | value must be greater or equal than hvars_rom.device_variables_rom[].lower_transducer_limit;<br>value must be lower than hvars_rom.device_variables_rom[].lower_transducer_trimpoint_max |
| Description | Minimum value for lower transducer trim point. Defines the DV range in which the DV trimming can be done. See description of DV trimming above. |

| **Name** | **hvars_rom.device_variables_rom[].lower_transducer_trimpoint_max** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = internal unit of DV (see chapter ”6.6.7 Engineering unit conversion”) |
| Default value | -10.0f |
| Valid range | value must be greater than hvars_rom.device_variables_rom[].lower_transducer_trimpoint_min;<br>value must be lower than hvars_rom.device_variables_rom[].upper_transducer_trimpoint_min |
| Description | Maximum value for lower transducer trim point. Defines the DV range in which the DV trimming can be done. See description of DV trimming above. |

| **Name** | **hvars_rom.device_variables_rom[].upper_transducer_trimpoint_min** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = internal unit of DV (see chapter “6.6.7 Engineering unit conversion”) |
| Default value | 60.0f |
| Valid range | value must be greater than hvars_rom.device_variables_rom[].lower_transducer_trimpoint_max;<br>value must be lower than hvars_rom.device_variables_rom[].upper_transducer_trimpoint_max |
| Description | Minimum value for upper transducer trim point. Defines the DV range in which the DV trimming can be done. See description of DV trimming above. |

| **Name** | **hvars_rom.device_variables_rom[].upper_transducer_trimpoint_max** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_float32_t; unit = internal unit of DV (see chapter “6.6.7 Engineering unit conversion”) |
| Default value | 70.0f |
| Valid range | value must be greater than hvars_rom.device_variables_rom[].upper_transducer_trimpoint_min;<br>value must be lower or equal than hvars_rom.device_variables_rom[].upper_transducer_limit |
| Description | Maximum value for upper transducer trim point. Defines the DV range in which the DV trimming can be done. See description of DV trimming above. |

### 6.6.7 Engineering unit conversion

All internal calculations in the HART Slave Stack are done in the internal unit of the Device Variable. The internal unit code for each DV has to be set up as described in the following table.

| Name | hvars_rom.device_variables_rom[].internal_unit_code |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t (enum) |
| Default value | DV_UNIT_CODE_DEGREES_CELCIUS (32) |
| Valid range | see HCF_SPEC-183 /7/, common table 2 |
| Description | This unit code is used for internal calculation. This is not the unit which is communicated via HART. This unit must be convertible in all units which can be setup for the DV. |

When a value is passed to the stack by the application or when the stack reads a value from a request message, it always converts the value to the internal unit before using it.

When the stack passes a value to the application or into a response message, it converts the value from the internal unit to the correct unit first.

For the conversion between two units, the stack calls the API function hart_convertDvValue(). The conversion itself has to be implemented by the application.

Before calling the conversion function the stack checks, if the conversion is possible by calling hart_checkDvUnitConvert(). This function is also used to decide if commands which set up a new unit code are accepted or rejected.

With the function hart_checkDvUnitBlocked() the application has the possibility to block certain unit codes for certain device variables. When command #44 or #53 is received the HART Slave Stack checks if the requested unit can be adopted by checking the unit conversion possibilities of the application (hart_checkDvUnitConvert). If the conversion between the internal unit and the requested unit is not possible, the commands are rejected. If the conversions are possible the stack calls this function to check if the requested unit code is blocked anyway. If the unit is blocked, the commands are rejected too.

For some kind of Device variables, it makes no sense to allow changing the Unit Code. Therefore, the parameter ‘multiple_unit_codes_allowed’ is available as described in the table below.

| Name | hvars_rom.device_variables_rom[].multiple_unit_codes_allowed |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_bool_t |
| Default value | H_TRUE |
| Valid range | H_FALSE / H_TRUE |
| Description | This parameter decides whether unit code change is allowed for the corresponding DV or not. If unit code change is not allowed the commands #44 and #53 are rejected accordingly. |

### 6.6.8 Dynamic Variables mapping

As described in HCF_SPEC-099 /4/, chapter 8 “Dynamic and Device Variables”, the HART Slave Stack provides the possibility to map any of the Device Variables to any of the Dynamic Variables.

The mapping is handled completely inside the stack. The user has to define the default mapping which is adopted when the non-volatile parameters are reset to default.

| Name | hvars_rom.dynamic_variables_mapping_defaults[] |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | array of h_uint8_t; 4 elements (0=PV, 1=SV, 2=TV, 3=QV) |
| Default value | 0, 1, 1, 1 ( PV=DV0, SV=DV1, TV=DV1, QV=DV1 ) |
| Valid range | all values have to be valid DV codes (0 … DEVICE_VARIABLE_NUM-1) |
| Description | Default Dynamic / Device Variable mapping which is adopted when “reset to defaults” is executed. |

Furthermore, the user has the possibility to set up the permissions for the dynamic variables mapping. It is possible to allow or deny the mapping to each available dynamic variable for each device variable. Therefore the parameter dynamic_mapping_permission has to be set accordingly.

| Name | hvars_rom->device_variables_rom[].dynamic_mapping_permission[] |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | array of h_bool_t; DYNAMIC_VARIABLE_NUM elements (0=PV, 1=SV, 2=TV, 3=QV) |
| Default value | H_TRUE, H_TRUE, H_TRUE, H_TRUE |
| Valid range | all values have to be H_FALSE (mapping not allowed) or H_TRUE (mapping allowed) |
| Description | Permission setup for dynamic variables mapping.<br>There is one array for each DV. Each array has one value for each available dynamic variable.<br>If the value is H_TRUE, it is allowed to map the DV to the corresponding dynamic variable.<br>If the value is H_FALSE, it is not allowed to map the DV to the corresponding dynamic variable.<br>The permissions are checked in Command #51. |

## 6.7 Loop current (4..20mA)

### 6.7.1 Update of loop current

For transmitters the current loop value is calculated from the Device Variable value given by the application (hart_updateTransmitterDvAndStatus()). Therefore, the following steps are done within the stack:

1. DV Trimming
2. DV Damping
3. Limitation of value according to transducer limits
4. Dynamic / Device variable mapping (PV is selected for further steps)
5. Calculation of normed loop current % according to LRV/URV
6. Apply of Transfer function (see chapter “6.7.5 Transfer functions”)
7. Calculation of mA value (from normed % value)
8. Limitation according to min/max values (LOOP_CURRENT_MIN/LOOP_CURRENT_MAX)
9. Loop current trimming

Finally, the HART Slave Stack calls the API function hart_updateTransmitterLoopCurrent() to pass the resulting corrected / trimmed loop current value and the real loop current value to the application. The application has to implement the further processing of this value including the control of the loop current.

For actuators the measured loop current is passed to the stack by the application using the function hart_updateActuatorLoopCurrent().

### 6.7.2 Loop current trimming

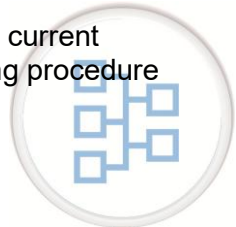
The HART slave stack implements linear two-point trimming for the loop current.

The trimming can be done with commands 45 and #46. See HCF_SPEC-151 /6/ for details.

The diagrams in chapters “6.6.3 Data flow overview for transmitter variables” and “6.6.4 Data flow overview for actuator variables” show the data flow including the loop current.

For transmitters the HART Slave Stack provides the application the virtual / trimmed loop current value in mA with the API function hart_updateTransmitterLoopCurrent(). With the trimming procedure the value is manipulated, so that the resulting “real” current is exact. In other words: The provided value is a virtual value. It represents not the “real” value which can be measured in the current loop. It represents a value which leads (together with the hardware inaccuracy) to the correct “real” value.

For actuators the application provides the HART slave stack the raw / untrimmed loop current value in mA with the API function hart_updateActuatorLoopCurrent(). With the trimming procedure the value is corrected, so that the resulting real / trimmed loop current value is exact.



With the following parameter the user has to determine the limit for the deviation between the trimmed and untrimmed loop current value. If cmd #45 or #46 would lead to a bigger value, the command is rejected.

| Name | #define LOOP_TRIM_MAX_DIFF_SETPOINT |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_float32_t; unit = milliampere |
| Default value | 0.4f (=> Allowed ranges: 3.6mA…4.4mA / 19.6mA…20.4mA) |
| Valid range | all positive values allowed |
| Description | Maximum difference from 4mA/20mA allowed for the setpoint when trimming the loop current (Cmds #45 and #46) |

| Name | #define LOOP_TRIM_MAX_DIFF_ERROR |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_float32_t; unit = milliampere |
| Default value | 1.0f |
| Valid range | all positive values allowed |
| Description | Maximum error difference allowed when trimming the loop current (cmds #45 and #46).<br>Or in other words: This is the maximum allowed difference between actual and target value in mA. |

### 6.7.3 Loop current limiting

The limit values for the loop current can be adapted with the definitions LOOP_CURRENT_MIN and LOOP_CURRENT_MAX (see tables below). The stack limits the current loop value to this range. In other words: If the primary variable value gets higher or lower, so that the loop current would exceed these limits, the loop current is set to the limit value and the “loop current saturated” bit in the device status is set.

| Name | #define LOOP_CURRENT_MIN |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_float32_t; unit = milliampere |
| Default value | 3.8f |
| Valid range | 0.0f…4.0f |
| Description | Minimum loop current value<br>This is the limit for the “real” value (see diagrams in chapters 6.6.3 and 6.6.4) |

| Name | #define LOOP_CURRENT_MAX |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_float32_t; unit = milliampere |
| Default value | 20.2f |
| Valid range | 20.0f…30.0f |
| Description | Maximum loop current value<br>This is the limit for the “real” value (see diagrams in chapters 6.6.3 and 6.6.4) |

It is possible to configure different limits for Cmd#40 (Enter/Exit Fixed Current Mode). Therefore, the definitions LOOP_CURRENT_MIN_CMD40 and LOOP_CURRENT_MAX_CMD40 (see tables below) have to be adapted accordingly.

| Name | #define LOOP_CURRENT_MIN_CMD40 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_float32_t; unit = milliampere |
| Default value | 3.2f |
| Valid range | 0.0f…4.0f |
| Description | Minimum loop current value for cmd #40. |

| Name | #define LOOP_CURRENT_MAX_CMD40 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_float32_t; unit = milliampere |
| Default value | 23.0f |
| Valid range | 20.0f…30.0f |
| Description | Maximum loop current value for cmd #40. |

### 6.7.4 Error states / PV Alarm Code

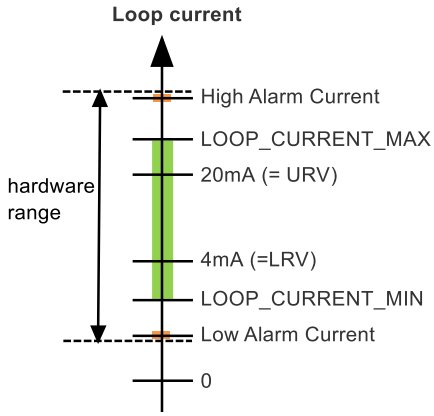



**Figure 11 Loop current range / alarm current**

For transmitters the loop current is set to a predefined alarm current value in case of some device specific errors. All device specific error cases must be handled by the application. The alarm state has to be passed to the stack with the function hart_setAlarmState(). The stack initializes the alarm state to H_FALSE (no alarm).

As described in HCF_SPEC-099 /4/ chapter 7.3.3, the alarm level set by the device (e.g. when the Device Malfunction bit is set) must be sufficient different from the Loop Current Saturated levels to allow differentiation by devices monitoring the Loop Current (see Figure 11).

For the calculation of the correct alarm current, the PV Alarm Code is provided from the HART Slave Stack to the application with the API function hart_setPvAlarmCode(). In the function the application has to check the alarm selection code. The function is called once at startup (during hart_init()) and whenever the PV Alarm Code is changed by command #100.

The application has to calculate the alarm value depending on the PV Alarm Code. The stack fetches the alarm value with the function hart_getPvAlarmValue() in each cycle if the alarm state is active.

The stack sets the alarm current if the alarm state is active. In alarm state the LOOP_CURRENT_FIXED bit is always set. The LOOP_CURRENT_SATURATED bit is set depending on the alarm current. If it is lower than LOOP_CURRENT_MIN or higher than LOOP_CURRENT_MAX the bit is set.

For actuators instead of the loop current the PV value is overwritten with the alarm value.

### 6.7.5 Transfer functions

The FCG specification defines a transfer function mechanism. The transfer function is applied to the loop current before the value is given out. The transfer function can be set with the HART command #47 (see HCF_SPEC-151 /6/).

The current revision of the HART Slave Stack does only support the transfer function ‘linear’. If Cmd #47 is called with a different transfer function, it is rejected with the response code ‘invalid selection’.

## 6.8 Status information handling

### 6.8.1 Field Device Status

The Field Device Status is transmitted in each response command from the device. It contains the following status information:

- Device Malfunction (Bit7)
- Configuration Changed (Bit6)
- Cold Start (Bit5)
- More Status Available (Bit4)
- Loop Current Fixed (Bit3)
- Loop Current Saturated (Bit2)
- Non-PV Out Of Limits (Bit1)
- PV Out Of Limits (Bit0)

Most of the Field Device Status bits are completely controlled by the stack. Only the ‘Device Malfunction’ bit must be controlled by the application.

For the ‘Device Malfunction’ bit the API function hart_setDeviceStatusMalfunction() has to be used by the application to set / clear the bit.

### 6.8.2 Cmd48 data

The number of device specific status bytes in the 2nd device specific part of command #48 data must be defined with the parameter COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES.

Due to differences in memory alignment across various microcontroller architectures, padding (gaps) may be introduced between 8-bit elements within a structure. To prevent this, the structure containing the status information must be defined as packed (see, for example, GCC packed structures).

Since different compilers use different keywords or macros to enforce structure packing, the user must define the appropriate macro in the file compilerDefines.h. This macro should match the syntax required by the specific compiler in use.

| **Name** | **#define COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES** |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_uint8_t |
| Default value | 11 |
| Valid range | 0…11 |
| Description | Number of device specific status bytes in 2nd device specific part of cmd #48 data. |

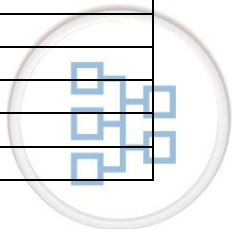
The function ‘hart_updateStatusData()' has to be used by the application to set or clear the application status bits of command #48 data (see table below, yellow marked bits). The function

expects a parameter of the type STATUS_DATA_T (type is defined in file 'hart_iface_types.h'). The structure contains all status bits but only the application bits are adopted. The other bits are controlled by the stack and therefore are overwritten by the stack.

To get a full and consistent copy of the complete status data the application has to use the API function hart_getStatusData().

The following table shows the Cmd #48 status data with a short description and the responsibilities (control by stack or application or unsupported). At startup (during hart_init()) the bits, which have to be controlled by the application, are initialized to 0 by the stack. The bits which are undefined are set to 0 by the stack.

| Byte | Bit | Field | Description | Responsibility | Remarks |
|---|---|---|---|---|---|
| 0…5 | | | Device specific status (1st part) | Application | |
| 6 | D7 | Ext. Device Status (Common table 17) | reserved | undefined | |
| | D6 | | reserved | undefined | |
| | D5 | | Condensed Status: Function Check | Stack | |
| | D4 | | Condensed Status: Out of Specification | Stack | |
| | D3 | | Condensed Status: Failure | Stack | |
| | D2 | | Critical Power Failure | Application | |
| | D1 | | DV Alert | Application | |
| | D0 | | Condensed Status: Maintenance required | Stack | |
| 7 | D7 | Device Operating Mode (Common table 14) | reserved | undefined | |
| | D6 | | reserved | undefined | |
| | D5 | | reserved | undefined | |
| | D4 | | reserved | undefined | |
| | D3 | | reserved | undefined | |
| | D2 | | reserved | undefined | |
| | D1 | | reserved | undefined | |
| | D0 | | reserved | undefined | |
| 8 | D7 | Standardized status 0 (Common table 29) | Device config locked | Stack | |
| | D6 | | Electronic defect | Application | |
| | D5 | | Environmental condition out of range | Application | |
| | D4 | | Power Supply out of range | Application | |
| | D3 | | Watchdog Reset Executed | Application | |
| | D2 | | Volatile Memory defect | Application | |
| | D1 | | Non-Volatile Memory Defect | Application | |
| | D0 | | DV simulation active | Stack | |
| 9 | D7 | Standardized status 1 (Common table 30) | reserved | undefined | |
| | D6 | | reserved | undefined | |
| | D5 | | reserved | undefined | |
| | D4 | | reserved | undefined | |
| | D3 | | Battery or Power Supply needs Maintenance | Application | |
| | D2 | | Event notification overflow | not implemented | |
| | D1 | | Discrete variable simulation active | not implemented | |
| | D0 | | Status simulation active | Stack | |
| 10 | D7 | Analog channel saturated (Common table 27) | reserved | undefined | |
| | D6 | | reserved | undefined | |
| | D5 | | reserved | undefined | |
| | D4 | | reserved | undefined | |
| | D3 | | Analog channel 4 saturated | not implemented | |
| | D2 | | Analog channel 3 saturated | not implemented | |
| | D1 | | Analog channel 2 saturated | not implemented | |
| | D0 | | Analog channel 1 saturated | not implemented | |
| 11 | D7 | Standardized | reserved | undefined | |



| | | | | | |
|---|---|---|---|---|---|
| | D6 | status 2 (Common table 31) | reserved | undefined | |
| | D5 | | reserved | undefined | |
| | D4 | | State Data Noticed | not implemented | |
| | D3 | | Sub-Devices with Duplicate IDs | not implemented | |
| | D2 | | Sub-Device mismatch | not implemented | |
| | D1 | | Duplicate Master detected | not implemented | |
| | D0 | | Sub-device List changed | not implemented | |
| 12 | D7 | Standardized status 3 (Common table 32) | reserved | undefined | |
| | D6 | | reserved | undefined | |
| | D5 | | reserved | undefined | |
| | D4 | | Radio Failure | not implemented | |
| | D3 | | Block Transfer Pending | not implemented | |
| | D2 | | Bandwith allocation pending | not implemented | |
| | D1 | | reserved | undefined | |
| | D0 | | Capacity denied | not implemented | |
| 13 | D7 | Analog channel fixed (Common table 28) | reserved | undefined | |
| | D6 | | reserved | undefined | |
| | D5 | | reserved | undefined | |
| | D4 | | reserved | undefined | |
| | D3 | | Analog channel 4 fixed | not implemented | |
| | D2 | | Analog channel 3 fixed | not implemented | |
| | D1 | | Analog channel 2 fixed | not implemented | |
| | D0 | | Analog channel 1 fixed | not implemented | |
| 14…X) | | Device specific status (2nd part) | | Application | |
| *X = 13 + COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES* | | | | | |

**Table 6: Cmd48 status bits**

The **More Status Available flag (MSA)** is automatically set by the stack. The stack user can define which Cmd #48 bits shall affect the more status available bit with the parameter more_status_available_status_mask[] in hart_parameters.c.

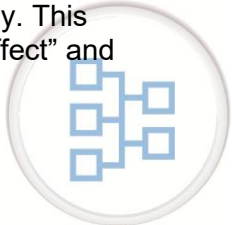
| **Name** | **hvars_rom.more_status_available_status_mask[]** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | array of h_uint8_t; same structure than STATUS_DATA_T (=cmd #48 data structure) |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Bit coded flags for Cmd48 status bits.<br>If flag is set, More Status Available bit is affected from the corresponding bit. |

MSA is cleared by the stack after the HART Master has collected the new status information by calling Cmd #48 with the current status data in the request.

MSA is set by the stack if current Cmd #48 data is different than Cmd #48 acknowledged by the master (by sending Cmd #48 with corresponding request data).

MSA is cleared by the stack if current Cmd #48 data is equal to Cmd #48 acknowledged by the master (by sending Cmd #48 with corresponding request data).

For the bits which must be handled by the Application the user can decide if the bits are supported or not. Therefore, the following definitions in hart_config.h must be adapted accordingly. This information is necessary for the condensed status mapping (difference between “no effect” and “undefined”).



| Name | #define DEVICE_SPECIFIC_STATUS_UNDEFINED_MAP_1 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; 6 elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for status bits of 1st device specific bytes in Cmd48 data<br>If flag is set (1), the corresponding status bit is not defined (not supported).<br>If flag is set (0), the corresponding status bit is active (supported). |

| Name | #define DEVICE_SPECIFIC_STATUS_UNDEFINED_MAP_2 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for status bits of 2nd device specific bytes in Cmd48 data<br>If flag is set (1), the corresponding status bit is not defined (not supported).<br>If flag is set (0), the corresponding status bit is active (supported).<br>Attention: Always define exactly COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements. It is possible to have an empty define here. |

### 6.8.3 Condensed status bits mapping

All bits of the Field Device Status and the Cmd48 Data can be mapped to one of the Condensed Status Bits. The mapping can be changed with command #524 and read out with command #523.

The mapping can be reset to defaults with command #525. The default values are implemented according to the FCG specification.

For the device specific status bits the user has to specify the default values. Therefore the following definitions in hart_config.h must be adapted accordingly.

| Name | #define DEVICE_SPECIFIC_STATUS_MAPPING_FAILURE_1 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; 6 elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for mapping of 1st device specific bytes in Cmd48 data<br>If flag is set, the corresponding bit is mapped to the condensed status bit “Failure” |

| Name | #define DEVICE_SPECIFIC_STATUS_MAPPING_FAILURE_2 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for mapping of 2nd device specific bytes in Cmd48 data<br>If flag is set, the corresponding bit is mapped to the condensed status bit “Failure”<br>Attention: Always define exactly COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements. It is possible to have an empty define here. |

| Name | #define DEVICE_SPECIFIC_STATUS_MAPPING_FUNCTION_CHECK_1 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; 6 elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for mapping of 1st device specific bytes in Cmd48 data<br>If flag is set, the corresponding bit is mapped to the condensed status bit “Function check” |

| Name | #define DEVICE_SPECIFIC_STATUS_MAPPING_FUNCTION_CHECK_2 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for mapping of 2nd device specific bytes in Cmd48 data<br>If flag is set, the corresponding bit is mapped to the condensed status bit “Function check”<br>Attention: Always define exactly COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements. It is possible to have an empty define here. |

| Name | #define DEVICE_SPECIFIC_STATUS_MAPPING_OUT_OF_SPECIFICATION_1 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; 6 elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for mapping of 1st device specific bytes in Cmd48 data<br>If flag is set, the corresponding bit is mapped to the condensed status bit “Out of specifcation” |

| Name | #define DEVICE_SPECIFIC_STATUS_MAPPING_OUT_OF_SPECIFICATION_2 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for mapping of 2nd device specific bytes in Cmd48 data<br>If flag is set, the corresponding bit is mapped to the condensed status bit “Out of specifcation”<br>Attention: Always define exactly COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements. It is possible to have an empty define here. |

| Name | #define DEVICE_SPECIFIC_STATUS_MAPPING_MAINTENANCE_1 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; 6 elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for mapping of 1st device specific bytes in Cmd48 data<br>If flag is set, the corresponding bit is mapped to the condensed status bit “Maintenance required” |

| Name | #define DEVICE_SPECIFIC_STATUS_MAPPING_MAINTENANCE_2 |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | array of h_uint8_t; COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements |
| Default value | 0 |
| Valid range | 0x00…0xFF |
| Description | Flags for mapping of 2nd device specific bytes in Cmd48 data<br>If flag is set, the corresponding bit is mapped to the condensed status bit “Maintenance required”<br>Attention: Always define exactly COMMAND_48_DEVICE_SPECIFIC_STATUS_BYTES elements. It is possible to have an empty define here. |

## 6.9 Non-Volatile data handling

The HART Slave Stack needs to save variables in non-volatile memory. The non-volatile memory must be provided and controlled by the application. The HART Slave Stack keeps a RAM mirror of the non-volatile data and operates on that mirror.

The HART Slave Stack calls ‘hart_writeNvData()' whenever it has updated a value in the RAM mirror. The application then should update the NV memory with the content of the mirror. Please note that ‘hart_writeNvData ()' can be called even if a variable was set to the same value. It is up to

the application to check for changes and to optimize the write algorithm (see demo application for example). If writing the mirrored data to the non-volatile memory leads to a time-consuming operation, the application should start a background process for this task and return immediately from function ‘hart_writeNvData ()'.

The initialization of the mirror during start-up works as follows: The stack creates the mirror as static memory space. During initialization the stack calls ‘hart_readNvData()'. This function must copy the data from non-volatile memory to the RAM mirror. If the non-volatile memory is blank or corrupted, the function must return 'H_FALSE'. In this case the stack initializes the RAM mirror with default values and calls hart_writeNvData() for writing them back again.

The application has to protect the non-volatile memory against data corruption. The parameters which are passed to the stack in the function ‘hart_readNvData()’ are taken without any further range or plausibility checks.

It is recommended to secure the non-volatile memory with a CRC. This allows detecting corruption during startup and run time.

## 6.10 Real-Time Clock (RTC)

Optionally HART devices can support a physical real-time clock (RTC), which is read and set with HART commands #89 and #90. The HART Slave Stack has two possibilities for supporting the RTC feature.

1) RTC Emulation

If no real-time clock is available, the stack can emulate the clock. This feature has to be enabled with the compiler switch EMULATE_RTC. If enabled, the stack uses the system’s timer value to count the time. The date / time is reset to 01.01.1900 00:00 if the microcontroller is reset. This option is normally used if the device has no ‘real’ RTC peripheral.

| **Name** | **#define EMULATE_RTC** |
|---|---|
| Item type | Compiler switch in hart_config.h |
| Default value | activated |
| Description | If no physical real-time clock (RTC) is available, the HART Slave Stack can emulate the RTC by software. In this case the clock is derived from the system timer value (mostly generated from CPU clock).<br>If a real RTC is used this define must be deactivated. |

2) RTC handled by application

If EMULATE_RTC is not defined the RTC has to be handled by the application. In this case the HART commands #89 and #90 lead to the calling of the API function ‘hart_setRTC()’ and ‘hart_getRTC()’. This option is normally used if the device has a ‘real’ RTC peripheral.

Of course, it is also possible to disable the RTC feature completely by deactivating the commands #89 and #90 (see chapter “7.1 Common practice commands”).

## 6.11 Write protection

The FCG specification defines a write protection mechanism.
Digital-write-protection is currently not supported.

If the Write Protection of a device is active the parameters of the device cannot be changed.

If a command which changes the parameterization of the device is called, while the Write Protection is active, the stack rejects the command with the response code ‘In Write Protect Mode’. The decision as to whether a command is affected by the write protection is done on the command

table. For this purpose, a dedicated flag or attribute for each command (see chapter “7.3.2 Command flags”).

Since the activation of the write protection is device specific (e.g. with a jumper, with a display/keyboard or with a device specific command), the application has to set the write protection state with the API function hart_setWriteProtectMode().

The write protection state can be read out with the universal command #15.

With the following compiler switch the user can configure whether the device supports the write protection feature or not. If the feature is disabled, the corresponding flag in the command table is ignored and Cmd #15 returns ‘None’ (251).

| **Name** | **#define WRITE_PROTECTION_ENABLED** |
|---|---|
| Item type | Compiler switch in hart_config.h |
| Default value | activated |
| Description | This switch decides whether the device supports write protection or not. |

## 6.12 Burst mode

The HART Slave Stack supports the burst-mode according to the FCG specification.

See HCF_SPEC-151 /6/ chapters 6.9 and command #103-#105, #107- #109 descriptions.

The number of supported independent burst messages can be set up with the definition NUM_OF_SUPPORTED_BURST_MESSAGES.

| **Name** | **#define NUM_OF_SUPPORTED_BURST_MESSAGES** |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_uint8_t |
| Default value | 3 |
| Valid range | Theoretical range: 3…255<br>But: It is strongly recommended to use 3 burst messages. Other values are untested. |
| Description | Number of supported burst messages.<br>Attention: If burst mode shall be disabled completely the corresponding commands have to be disabled (#define ACTIVATE_CMDxxx). For internal design reasons NUM_OF_SUPPORTED_BURST_MESSAGES must be set to 3 anyway (not 0!). |

With a dedicated flag / attribute in the command table it is possible to allow a command to be configured as burst command. See chapter “7.3.2 Command flags” for details. Device-specific commands can also be configured as burst commands, provided that the following restrictions are observed:

- Except for the Cmds #9 and #33, all burstable commands must have 0 request data bytes.
- All burstable commands must be read commands, which have no influence on the device configuration.

## 6.13 Aggregated commands

The HART Slave Stack supports command aggregation according to the FCG specification.

The feature is enabled automatically, if any of the following commands are activated (#define ACTIVATE_CMDxxx).

- #78 Read Aggregated Commands
- #103 Write Burst Period
- #104 Write Burst Trigger
- #105 Read Burst Mode Configuration
- #107 Write Burst Device Variables

- #108 Write Burst Mode Command Number
- #109 Burst Mode Control

Note that command aggregation is mandatory for burst devices.

With a dedicated flag / attribute in the command table it is possible to allow a command to be configured as aggregated command. See chapter “7.3.2 Command flags” for details. Device-specific commands can also be aggregated, provided that the following restriction is taken into account.

Aggregation is limited to read commands that do not modify the device configuration.

## 6.14 Special HART commands

### 6.14.1 Perform Self Test (Cmd #41)

See HCF_SPEC-151 /6/.

When the HART Slave Stack receives a command #41 “Perform Self Test”, it calls the API function hart_performSelftest(). The function has to trigger the start of a self-test. During the self-test the field device must continue to communicate. Therefore, the function has to return immediately.

The results of the self-test are transmitted via the status information (see chapter “6.8 Status information handling”). After the self-test has been finished the application has to update the status information accordingly.

It is also possible to do self-test in background continuously. In this case command #41 can be deactivated because no ‘start trigger’ is needed. If any error is detected, the corresponding error bit in the cmd48 status data must be set.

### 6.14.2 Perform Device Reset (Cmd #42)

See HCF_SPEC-151 Preface.

Is disabled due device reset must not to be used in new designs.

### 6.14.3 Squawk (Cmd #72)

See HCF_SPEC-151 /6/.

When the HART Slave Stack receives a command #72 “Squawk”, it calls the API function hart_squawk() with the received ‘Squawk control code’ (common table 66). The application has to decide if the requested squawk is possible or not. Depending on the return value of hart_squawk(), the HART Slave Stack responses to the command with the corresponding response code.

### 6.14.4 Find Device (Cmd #73)

See HCF_SPEC-151 /6/.

When the HART Slave Stack receives the command #73 “Find Device”, it calls the API function hart_getFindDeviceArmedState(). The application has to decide if the device is armed or not. If the device is armed, the HART Slave Stack responses to the command. If the device is not armed, no response is sent.

### 6.14.5 Country code (Cmds #512, #513)

See HCF_SPEC-151 /6/.

At startup (hart_init()) the country code is passed to the application with the API function hart_setCountryCode() once.

When a new country code is set with Cmd #513 the HART Slave Stack calls the API function hart_setCountryCode(). In the function the application has to check if the new country code is accepted or not. The command returns with the response code according to the return value of hart_setCountryCode(). In addition to the validation performed by the function, the application is notified when a new country code is detected. For example, this allows the display language to be updated accordingly.

In addition to the country code there is another parameter which is read / write by the commands #512 and #513: The 'SI Units Restriction' (see HCF_SPEC-183 /7/, Common Table 54).

When SI compliance restriction is enabled, the stack compares the configured unit code for each device variable with one stored in 'hvars_rom.device_variables_rom[].si_unit_code'. If one of the unit codes is not SI compliant the command is rejected with the response code "SI Units Restriction failed".

| **Name** | **#define DEFAULT_COUNTRY_CODE** |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | String |
| Default value | "DE" |
| Valid range | String with 2 standard ASCII characters |
| Description | Default value for country code which is set when "reset to defaults" is performed. |

| **Name** | **hvars_rom.device_variables_rom[].si_unit_code** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t (enum) |
| Default value | DV_UNIT_CODE_DEGREES_KELVIN (35) |
| Valid range | see HCF_SPEC-183 /7/, common table 2 |
| Description | This unit code is used for SI-compliance. In case the device is configured to accept SI-unit codes only, all other requested unit codes are rejected. |

# 7 HART commands tailoring

## 7.1 Common practice commands

The HART Slave Stack supports most of the common practice commands (see chapter "14.1 Supported HART commands"). They are implemented in 'hart_common_practice.c'.

The user can enable or disable each command individually. For this purpose, a dedicated compiler switch is provided for each command in 'hart_config.h'.

Some commands belong together and they must be enabled / disabled together.
Example: Burst commands must be all enabled or all disabled.

```
/*** Defines to activate Common Practice and Factory Commands ***/
// burst mode
#define ACTIVATE_CMD103 // Write Burst Period
#define ACTIVATE_CMD104 // Write Burst Trigger
#define ACTIVATE_CMD105 // Read Burst Mode Configuration
#define ACTIVATE_CMD107 // Write Burst Device Variables
#define ACTIVATE_CMD108 // Write Burst Mode Command Number
#define ACTIVATE_CMD109 // Burst Mode Control
```

If the user needs a common practice command which is not implemented, it is possible to implement it in the same way like a device specific command (see chapter "7.3 Device specific commands").

## 7.2 Special non-public / factory commands

| Name | #define USE_FACTORY_COMMANDS |
|---|---|
| Item type | Compiler switch in hart_config.h |
| Default value | activated |
| Description | If factory commands are used, this switch must be activated. |

The FCG specification defines a special set of commands (cmds #122 - #126) which is intended for factory-only use during the construction of a field device. These commands should not be used when servicing a device in the field (see HCF_SPEC-99 /4/).

The HART Slave Stack demo application has implemented the following factory command in hart_factory_cmds.c/.h. The factory command support can be enabled or disabled with the same mechanism like for Device specific command support (Refer 7.3).

Additional factory commands can be implemented in hart_factory_cmds.c if needed.

For non-public (factory) commands there is a special handling in the stack. The commands are rejected with response code “Command not implemented” instead of “Too few data bytes received” if the number of request data bytes is smaller than defined number in the command table. This way, the commands are hidden from the end-user. An additional magic pattern mechanism as shown in the example implementation in the demo application is implemented to be sure, that the commands cannot be executed by the end-user.

### 7.2.1 Command 122 “Reset non-volatile memory to factory defaults”

This command allows resetting the complete non-volatile memory to factory defaults. The write operation is done like “normal” update of any non-volatile variable (see chapter “6.9 Non-Volatile data handling”).

The request contains a ‘magic number’ to avoid resetting of parameters by mistake.

The command returns 20 dummy data bytes. After the response (ACK) has been sent completely, the device is reset. The dummy bytes are required because the writing to non-volatile memory is performed by the application in background. With the dummy bytes sending of the response takes at least 180ms, so the application has enough time to write the data to the memory before the reset occurs.

**Request Data Bytes**

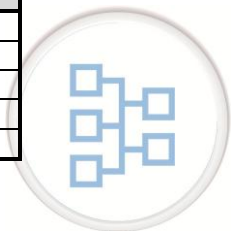
| Byte | Format | Descrption |
|---|---|---|
| 0 | Unsigned-8 | 0xAB (magic pattern) |
| 1 | Unsigned-8 | 0xCD (magic pattern) |

**Response Data Bytes**

| Byte | Format | Descrption |
|---|---|---|
| 0 | Unsigned-8 | 0xAB (magic pattern) |
| 1 | Unsigned-8 | 0xCD (magic pattern) |

**Command-Specific Response Codes**

| Code | Class | Descrption |
|---|---|---|
| 0 | Success | No Command-Specific Errors |
| 1 | | Undefined |
| 2 | Error | Invalid Selection (magic pattern error) |
| 3-4 | | Undefined |
| 5 | Error | Too Few Data Bytes Received |



## 7.3 Device specific commands

| Name | #define USE_DEVICE_SPECIFIC_COMMANDS |
|---|---|
| Item type | Compiler switch in hart_config.h |
| Default value | activated |
| Description | If device specific commands are used, this switch must be activated. |

| Name | #define COMMAND_DATA_SIZE_MAX |
|---|---|
| Item type | Definition in in hart_config.h |
| Data type | h_uint8_t |
| Default value | 69 |
| Valid range | 69…255 |
| Description | Maximum number of payload response bytes of all implemented commands which are allowed to be aggregated. Without any device specific command the relevant command is cmd #9 with 69 bytes of data. If device specific commands with bigger response are implemented and which are allowed to be aggregated, this value must be increased.<br>Attention: This will cause higher RAM usage because internal buffers will be increased. |

### 7.3.1 Adding commands

The HART Slave Stack can be extended with device specific commands. These commands are implemented in the file 'hart_device_specific.c' (header file 'hart_device_specific.h'). This feature must be enabled by defining the macro USE_DEVICE_SPECIFIC_COMMANDS in the file ‘hart_config.h’.

Every command is handled by a separate C function (command handler), which implements the interface 'HartCommandFunction'. Direct return statements are not permitted; the HART_RETURN macro must be used. See demo application for example implementation.

The table 'device_specific_command_table[]' needs to be extended by new commands. This table extends the stack internal available commands table (which contains the entries for the stack’s built-in commands).

A command table entry has the following structure:

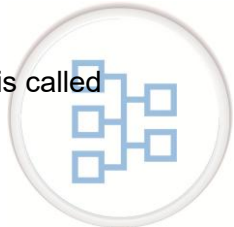
```
// HART command type
typedef struct HAP_COMMAND_TAG {
  COMMAND_NR_EXT command_nr;
  HartCommandFunction command_function;
  h_uint8_t request_data_cnt_min;
  h_uint8_t flags;
} HAP_COMMAND;
```

1. command_nr: 16-bit command number
2. command_function: Function pointer to command handler function
3. request_data_cnt_min: Minimum number of request data bytes:
   *If the request contains less data bytes the command is rejected by the stack with the response code “Too Few Data Bytes Received” without even calling the command handler function. It is also possible to set this value to 0 and do the check in the command handler function.*
4. flags: Command Flags (see chapter “7.3.2 Command flags”)

#### 7.3.1.1 Perform Commands in Slices

It is possible to perform a command function in slices. The related command function is called permanently at 10ms intervals as long as the return value is COMMAND_RUNNING.

See demo application for example implementation



### 7.3.2 Command flags

In the command table every command can have several of the following attributes/flags. Not all attribute/flag combinations are possible (e.g. FLAG_CONFIG_CHANGE is not possible together with FLAG_BURSTABLE because only read commands without any influence on the device configuration can be burstable). For new commands the user is responsible to setup a valid combination.

| | |
|---|---|
| FLAG_WRITE_PROTECT | command is rejected if device Write Protection is active<br>(see chapter “6.11 Write protection”) |
| FLAG_LOCK_DEVICE | command is rejected if device is locked<br>(see HCF_SPEC-151 /6/, page 83) |
| FLAG_CONFIG_CHANGE | Configuration Changed Counter is incremented and Configuaration Changed Bit in Device Status is set when command was executed |
| FLAG_AGGREGATED | command can be aggregated<br>(see chapter “6.13 Aggregated commands”) |
| FLAG_BURSTABLE | command can be used in a burst message<br>(see chapter “6.12 Burst mode”) |

Example:

```
HAP_COMMAND const device_specific_command_table[DEVICE_SPECIFIC_COMMANDS_NUM] = {
  //  cmd  cmd-function     min. request bytes    flags
  {   128, test_command,         1,              FLAG_AGGREGATED                 }, // test command
  {   129, dr_test_command,      0,              FLAG_AGGREGATED                 }, // test command for delayed response
  {   130, test_burst_command,   0,              FLAG_AGGREGATED | FLAG_BURSTABLE }, // test command for burst mode
  { 64768, high_nr_test_command, 0,              FLAG_AGGREGATED                 }, // test command for high command numbers
};
```

### 7.3.3 Request / Response buffer access functions

The HART slave stack provides the following API functions for accessing the request and response buffer. They can be used within the device specific command handler functions. For accessing the received request data (STX data) the read functions are used. For build the response (ACK data) the print functions are used.

| Print functions (access response buffer) | Read functions (access request buffer) |
|---|---|
| hart_printResponseByte() | hart_readRequestByte() |
| hart_printResponseWord() | hart_readRequestWord() |
| hart_printResponseLongword() | hart_readRequestLongword() |
| hart_printResponseFloat() | hart_readRequestFloat() |
| hart_printResponseArray() | hart_readRequestArray() |
| hart_printResponseTime() | hart_readRequestTime() |

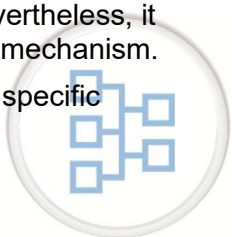
In order to get the number of bytes received in a HART frame, the function ‘hart_getRequestBytecount()’ can be used.

### 7.3.4 Delayed response commands

See HCF_SPEC-307 /8/, Chapter 7 “Delayed Slave Responses”.

The stack itself doesn’t use the Delayed Response mechanism for any command. Nevertheless, it is possible to implement device specific commands which use the Delayed Response mechanism.

Therefore the command has to be added to the command table like a “normal” device specific command as described in chapter “7.3 Device specific commands”.



Beside the command handler function an additional C function has to be implemented which is the “callback”-function for the command. The callback function is the “working” function for the delayed response command. It is cyclically called by the stack until the job is done.

Within the “normal” command handler function the API function hart_checkDelayedResponse() has to be called to get the actual DR state for this command from the stack. Depending on the states different actions have to be performed.

If the DR state is DELAYED_RESPONSE_UNUSED, no delayed response action is active for this command. In this case the callback function has to be registered with the API function hart_registerDelayedResponse(). If the registration succeeds a free DR buffer is available and the return code has to be set to “DR initiated”. Otherwise, no slot is free, and the command has to return the response code “DR conflict”. No data bytes are returned. The stack supports 4 DR buffers (2 for each master).

If the DR state is DELAYED_RESPONSE_ACTIVE, the command is already running, and the response code has to be set to “DR running”. No data bytes are returned.

If the DR state is DELAYED_RESPONSE_FINISHED, the command has been finished in background. In this case the response code and data bytes must be set according to the command execution result.

During the execution of a DR command the stack calls one callback function each 20ms (if multiple DR commands are running, the calls are less often). The callback function has to return DELAYED_RESPONSE_ACTIVE or DELAYED_RESPONSE_FINISHED.

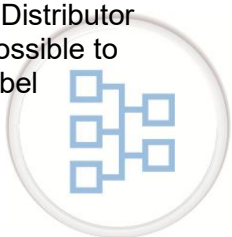
See ‘hart_device_specific.c’ in the demo application for DR command example.

# 8 HART device customization

## 8.1 Device Identity

The device identity parameters can be passed to the stack with the following API functions:

- hart_setIdentityDeviceID()
  - The ‘Device ID’ is a number unique to a particular field device. This number must be different for every device produced with a given Device Type.
    Data type: 24-bit value, range: 0x000000…0xFFFFFF
- hart_setIdentityDeviceRevision()
  - The ‘Device Revision’ indicates the version of the command set for the product (affects the DD!). The major number of the revision level (version number) of the device specific document must match this value (see HCF_SPEC-099 /4/ chapter 9.2).
    Data type: 8-bit value, range: 0…255
- hart_setIdentityHwRevision()
  - The ‘Hardware revision’ describes the current used Hardware revision number. It is a manufacturer specific number which doesn’t affect to the DD.
    Data type: 5-bit value, range: 0…31
- hart_setIdentityPrivateLabelDistributorCode()
  - The ‘Private Label Distributor Code’ and ‘Device Type’ allows the user to know which product is being communicated with. Normally the ‘Private Label Distributor Code’ is set to the same value than the ‘Manufacturer ID’ but it’s also possible to use a different value if the “real” manufacturer shall be hidden (brand label products).
    Data type: 16-bit value, range: 0x0000…0xFFFF



Setting the parameters with these API functions gives the most possible flexibility. So it's up to the application how the values are generated (e.g. read from EEPROM, read from hardware pins, set by non-public factory command).

The stack saves these parameters with the same mechanism together with all other non-volatile parameters.

The functions have only to be called when a parameter changes (which should normally only be the case during production).

Attention: After any change a device reset is executed by the stack (by calling hart_performDeviceReset).

## 8.2 HART interface parameters

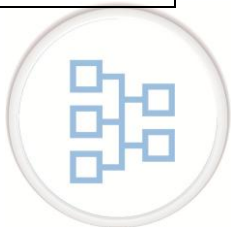
Setup of the HART interface parameters must be done via the following ROM parameters in hart_parameters.c in the constant structure 'hvars_rom'.

| **Name** | **hvars_rom.device_type** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t[2] |
| Default value | 0x00, 0x00 |
| Valid range | all values allowed |
| Description | The Expanded Device Type indicates a specific product. It is registered at the FCG and contained in HCF Common Table 1. Expanded Device Type and Device ID identify the device on the network. Expanded Device Type and Device Revision identify the command interface of the device. The high byte is usually identical with the Manufacturer ID. |

| **Name** | **hvars_rom.software_revision** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t |
| Default value | 0 |
| Valid range | all values allowed |
| Description | This value describes the current used Software revision number. It is a manufacturer specific number which doesn't affect to the DD. |

| **Name** | **hvars_rom.physical_signaling_code** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t |
| Default value | PHYSICAL_SIGNALING_BELL202_CURRENT (0) |
| Valid range | see HCF_SPEC-183 /7/, common table 10 |
| Description | see HCF_SPEC-183 /7/, common table 10 |

| **Name** | **hvars_rom.flag_assignments** |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t |
| Default value | 0x00 |
| Valid range | see HCF_SPEC-183 /7/, common table 11 |
| Description | This value describes the flag assignments of the device. Any bit not covered is undefined and must be set to zero. The bits are described in HCF_SPEC-183 /7/, common table 11. |



| Name | hvars_rom.manufacturer_indentification_code |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint16_t |
| Default value | 0 |
| Valid range | see HCF_SPEC-183 /7/, common table 8 |
| Description | This field must include the Manufacturer Identification Code given by FCG for each company that produces HART devices. |

| Name | hvars_rom.device_profile |
|---|---|
| Item type | Parameter in structure hvars_rom in hart_parameters.c |
| Data type | h_uint8_t |
| Default value | DEVICE_PROFILE_CODE_PROCESS_AUTOMATION (1) |
| Valid range | see HCF_SPEC-183 /7/, common table 57 |
| Description | see HCF_SPEC-183 /7/, common table 57 |

| Name | #define DEFAULT_POLLING_ADDRESS |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_uint8_t |
| Default value | 0 |
| Valid range | 0…63 |
| Description | Default polling address which is set when “reset to defaults” is executed. |

| Name | #define PRAEAMBEL_BYTES_DEFAULT |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_uint8_t |
| Default value | 5 |
| Valid range | 5…20 |
| Description | Default number of response preambles which is set when “reset to defaults” is executed. |

## 8.3 User data customization

All HART-related data requiring nonvolatile storage is organized within a single structure . It is possible to enhance this structure with device specific data (e.g. additional device information distributed using HART interface). Therefore, the type-definition ‘HART_USER_DATA_FLASH’ located in ‘hart_config.h’ can be adopted to include all user defined data. To activate the user data, it is necessary to enable the define ‘ACTIVATE_FLASH_USER_DATA’.

| Name | #define ACTIVATE_FLASH_USER_DATA |
|---|---|
| Item type | Definition in in hart_config.h |
| Data type | n/a |
| Default value | n/a |
| Valid range | n/a |
| Description | This define activates the handling for user defined data. It enables the interface functions to access the user defined data. |

As a result the structure handled in ‘hart_writeNvData ()' and ‘hart_readNvData ()' contains the data specified by the user (Refer chapter 6.9 Non-Volatile data handling). To access the data in application scope, the following functions can be used:

- HART_USER_DATA_FLASH* hart_getUserDataPtr(void)
- hart_markNonVolatileDataModified(void)

The function ‘hart_getUserDataPtr() returns a pointer to the user defined data. Using this pointer, the data can be read or written. In case the data has been modified, the function

‘hart_markNonVolatileDataModified()’ must be called. This will result in a call of ‘hart_writeNvData()’ to write the data to non-volatile memory. In some situation the stack resets all non-volatile data to default values (i.e at startup if non-volatile read-out fails and if hart_resetHartIfaceToDefaults() is called). Therefore, the function ‘hart_resetUserData()’ has to be implemented by the user application which reset all user non-volatile data defined in ‘HART_USER_DATA_FLASH’.

# 9 Target/compiler customization

## 9.1 Payload buffer size

If a microcontroller with very limited RAM is used, the RAM usage of the stack can be reduced by decreasing the payload buffer size for HART commands. Another possibility is to disable aggregated commands (see chapter 6.13 Aggregated commands) and burst mode (see chapter 6.12 Burst mode).
Of course, the buffer must be sufficiently large to hold the command with the largest number of data bytes (request or response).

| **Name** | **#define PAYLOAD_BUFFER_SIZE** |
|---|---|
| Item type | Definition in hart_config.h |
| Data type | h_uint8_t |
| Default value | 251 |
| Valid range | 71…251 |
| Description | Payload buffer size for incoming and outgoing HART messages |

The minimum value for PAYLOAD_BUFFER_SIZE is defined based on the build-in command with most data bytes: Cmd #9 with 8 slots = 69 data bytes + response code + device status = 71 bytes.

## 9.2 Debug output

During development and for debugging purposes the stack outputs strings, which describe internal states, events and other helpful information. Therefore, the stack calls the macro **H_DBGOUT()** which is defined in hart_config.h. The macro is invoked like the standard ‘printf()’ function from <stdio.h>. With the macro definition the user is free to define the action which shall be performed for displaying the message. For release version the macro should be defined as an empty statement.

***Attention:*** In the demo application the output is mapped to the debug interface module implemented for the evaluation board. The execution of the printf() function is very time-consuming. When using slow clock rates this may cause timing problems of the HART Slave Stack!

In the current revision of the HART Slave Stack the debug output is used only partially. It’s mainly a feature for future usage. It is recommended to define the macro as an empty statement (no operation).

## 9.3 Memory manipulating functions

The HART Slave Stack needs the functions memcpy(), memcmp(), memset() from <string.h>. For most possible flexibility the function calls are abstracted with the following macros in hart_config.h.

- H_MEMCPY(dst, src, len)
- H_MEMCMP(p1, p2, len)
- H_MEMSET(dst, val, len)

So the user is free to implement the functions himself or to include the <string.h> header and map the macros to the standard functions. A second possibility is shown in the demo application.

## 9.4 Inline functions

For timing optimization some functions of the HART Slave Stack are defined as “inline”-functions with the H_INLINE keyword defined in ‘hart_config.h’. If the used compiler supports this feature the definition should be adapted according to the compiler keyword usage (mostly ‘__inline’). If not supported the #define should be defined empty.

## 9.5 Standard data types

The following standard data types are used by the stack. They are defined in ‘hart_config.h’ and can be adapted to the used target / compiler.

| Name | Description |
|---|---|
| h_uint8_t | 8-bit unsigned integer (0…255) |
| h_bool_t | Boolean |
| h_int8_t | 8-bit signed integer (-128…127) |
| h_uint16_t | 16-bit unsigned integer (0…65’535) |
| h_int16_t | 16-bit signed integer (-32’768…32’767) |
| h_uint32_t | 32-bit unsigned integer (0…4’294’967’295) |
| h_int32_t | 32-bit signed integer (-2’147’483’648…2’147’483’647) |
| h_float32_t | 32-bit floating point value according to IEEE 754 |

There are two additional definitions for the Boolean type (h_bool_t): H_TRUE and H_FALSE. All variables from this type are only assigned to and checked against these values. The definitions can be adapted to the target / compiler.

## 9.6 Assert macro

The macro #define HART_ASSERT() is used within the stack to do some important checks. The assertions fail in case of fatal software errors or if the stack configuration is invalid. The assertion implementation should be active during development and testing of the device. After successful testing it can be disabled in the release version.

# 10 Stack internal design information

## 10.1 Source code documentation

The HART Slave Stack source code contains C comments with special keywords for Doxygen. From the source code detailed source code documentation can be generated with Doxygen. The documentation includes:

- File descriptions
- Function descriptions
- Function parameters descriptions
- Function return values descriptions
- Call graphs for each function
- Caller graphs for each function

### 10.1.1 Import / export mechanism

MESCO has established a mechanism for importing and exporting C functions and variables between different C modules. This mechanism is also used in the HART Slave Stack source files. A brief description is provided below:

- All functions and all module variables have one of the keywords LOCAL or GLOBAL in front of their declaration and definition.
- LOCAL is used for static variables / functions, which are only used within the C module in which they are defined. The keyword is always defined as ‘static’.
- GLOBAL is used for global variables / functions, which are exported by the C module in which they are defined and used by other C modules. The keyword is either defined to ‘extern’ or ‘’ (empty).
- The keywords themselves are defined in all source code header files (*.h). Therefore, each header file contains a generic “import / export mechanism” part at the beginning.
- Before the include-section each C file contains a #define [module name]_C. With this definition it is possible in the following included header files to decide if the header is included from the file which is actually compiled (export) or if the header is included from a different file (import).

With this mechanism, it is immediately clear whether a variable or function is local or global. In addition, global variables have to be declared only once (in the header).

For the user the mechanism has no impact. The stack header files required by the application can be included as usual. It is recommended to use this mechanism in modules that interface directly with the stack (see demo application), although it is not mandatory.

## 10.2 Internal interfaces

### 10.2.1 Physical Layer Interface

Implementation of the physical layer requires sophisticated knowledge about the specifications provided by the FCG. Implementation guidance is out of the scope of this manual.

The physical layer interface is common to both the soft modem and hardware modem variants of the stack. It consists of the following set of C functions. The prototypes of the functions are defined in the file hart_phl_interface.h. See /a/ for the detailed definitions of the functions (i.e. function parameters, return values, descriptions).

- GLOBAL void **hart_phl_Initialize**(void);

- GLOBAL void **hart_phl_GetCR**(PHL_CONST_REQ const * * cr);
- GLOBAL void **hart_phl_GetDR**(PHL_DYNAMIC_REQ * const dr);
- GLOBAL void **hart_phl_SetDR**(PHL_DYNAMIC_REQ const * const dr);
- GLOBAL void **hart_phl_GetReceiveState**(BF_PHLS_R * const r_state);
- GLOBAL void **hart_phl_GetTransmitState**(BF_PHLS_T * const t_state, HART_TIME_MS *last_byte_ms);
- GLOBAL H_RESULT **hart_phl_SetHartMode**(PHLM mode);
- GLOBAL H_RESULT **hart_phl_Receive**(PHL_RECEIVE_ITEM * const item, h_uint8_t som);
- GLOBAL H_RESULT **hart_phl_GetOutputBufferSpace**(h_uint16_t * const pBufSpace);
- GLOBAL H_RESULT **hart_phl_Transmit**(h_uint8_t const * const p_byte, h_uint16_t n_bytes, h_uint8_t complete);

# 11 Project organization

The HART Slave Stack source code is distributed among various files. For adoption of the stack for a specific device application and platform, various files have to be created. To simplify this process, the delivery package contains two complete demo projects: one for actuator and one for transmitter usage.

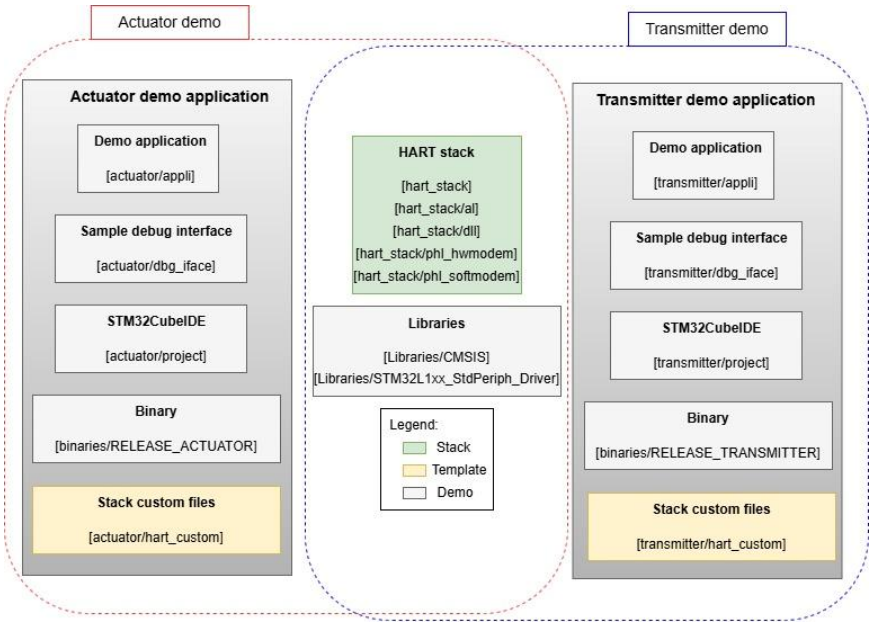



**Figure 12 Project organization**

To integrate the stack with the device specific needs, the different file types shall be used as follows:

- The HART stack files (marked in green color) shall be used without any modifications.
- The Stack custom files (marked in yellow color) shall be used as template. They shall be taken from the corresponding demo application (transmitter or application) and adapted to the device's specific needs.
- The other files (marked in grey color) shall be created from scratch. The demo files can be used as references.

The following tables describe the HART stack files and the Stack custom files (templates) with a short content description.

Global:

| Folder / File | Content description |
|---|---|
| hart_global.h | HART Slave Stack globals |
| hart_common_tables.h | Common Tables definitions (see HCF_SPEC-183 /7/) |
| hart_stack.h | Global HART Slave Stack include file |
| hart_parameters.h | Declarations for application specific parameters |
| hart_phl_interface.h | Physical Layer Interface definition |

Application Layer:

| Folder / File | Content description |
|---|---|
| al\hart_appl_interface.h | Application interface definitions (functions called by appli) |
| al\hart_common_practice.c / .h | Command handler functions for Common Practice Commands |
| al\hart_stack_iface.h | Application interface definitions (functions called by stack) |
| al\hart_iface_types.h | Generic application interface definitions and typedefs |
| al\hart_hap.c / .h | Application Layer implementation |
| al\hart_rtc.c / .h | Emulated RTC module |
| al\hart_universal.c / .h | Command handler functions for Universal Commands |

Data Link Layer:

| Folder / File | Content description |
|---|---|
| dll\hart_dll.c / .h | Data Link Layer implementation |

Physical Layer (for HW modem)

| Folder / File | Content description |
|---|---|
| phl_hwmodem\hart_hwmodem_common.c / .h | HW Modem (implements Physical Layer Interface) |
| phl_hwmodem\hart_hwmodem_driver.h | HW Modem Interface definition |
| phl_hwmodem\hart_hwmodem_rx.c / .h | HW Modem Receiver module |
| phl_hwmodem\hart_hwmodem_tx.c / .h | HW Modem Transmitter module |

Custom files (templates):

| Folder / File | Content description |
|---|---|
| hart_custom\hart_config.h | Stack configuration and platform dependent definitions |
| hart_custom\hart_device_specific.c / .h | Command handler functions for Device Specific Commands |
| hart_custom\hart_factory_cmds.c / .h | Command handler functions for special factory commands |
| hart_custom\hart_stack_iface.c | Application interface implementation (functions called by stack) |
| hart_custom\hart_hwmodem_driver.c | HW Modem Driver (implements HW Modem Interface) |
| hart_custom\hart_parameters.c | Application specific parameters |

**Table 7: Project folders and files**

## 11.1 Compiler parameters

A stack size of at least 256 bytes is necessary.

# 12 Demo application

## 12.1 Development Environment

For the development of the HART Slave Stack and the demo application MESCO uses the following environment.

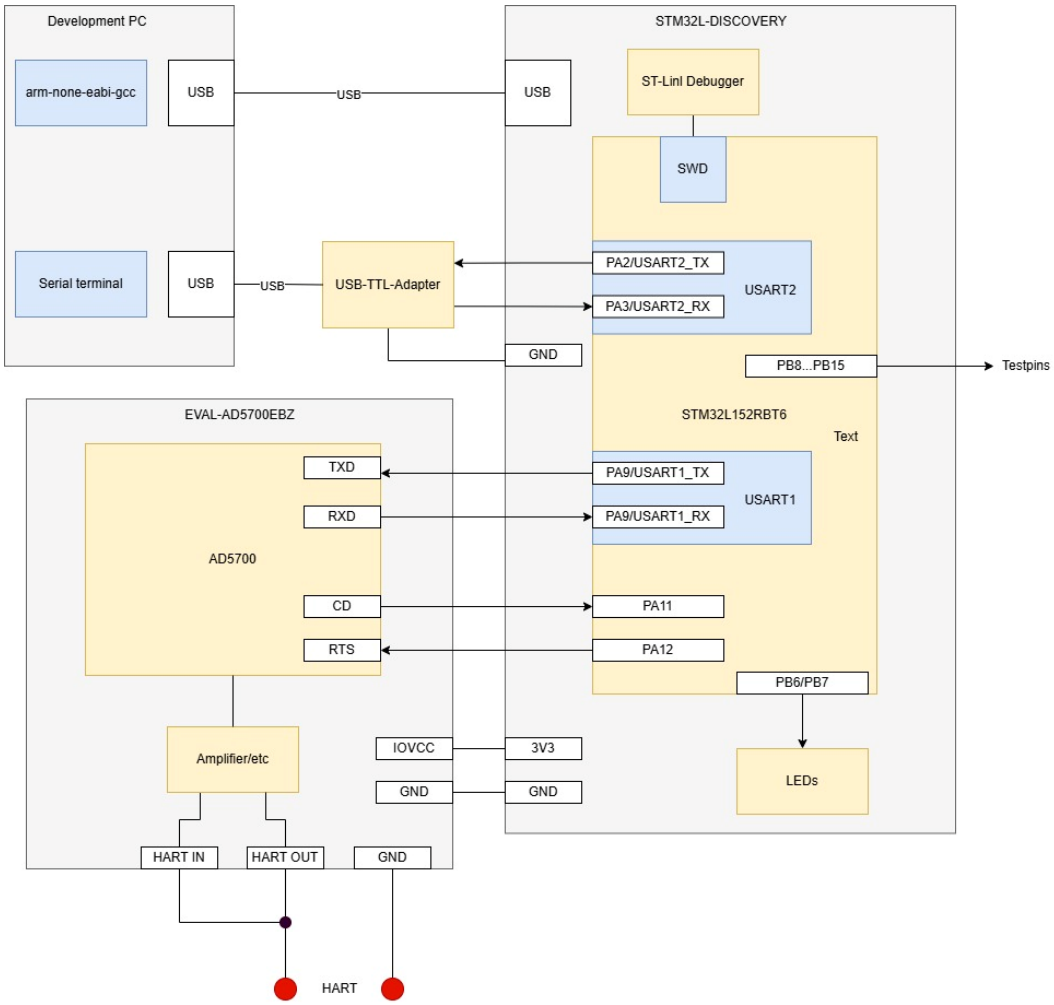



**Figure 13 Development environment - block diagram**

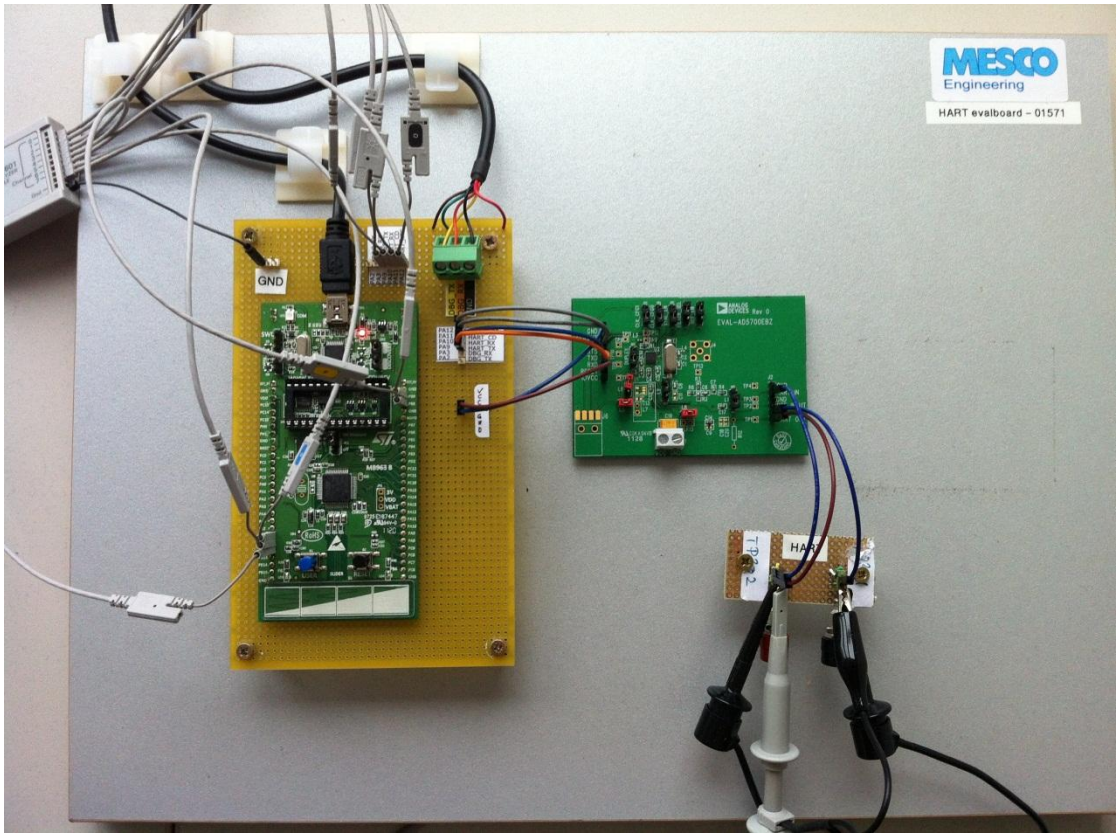



**Figure 14 Development environment - Picture**

The development environment consists of the following components:

- Microcontroller Evaluation Board (STM32L-DISCOVERY)
  - Overview page: https://www.st.com/en/evaluation-tools/32l152cdiscovery.html
  - User Manual STM32L1 discovery kits
    http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00027954.pdf
  - Getting Started document
    http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00035245.pdf
  - Microcontroller STM32L152RBT6
    Datasheet:
    http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00277537.pdf
    Manual:
    http://www.st.com/st-web-ui/static/active/en/resource/technical/document/reference_manual/CD00240193.pdf
- HART Modem Evaluation Board (EVAL-AD5700EBZ)
  - User Guide:
    http://www.analog.com/static/imported-files/user_guides/UG-382.pdf
- USB-TTL-Adapter
  - FTDI TTL Serial Adapter TTL-232R-3V3
  - Datasheet
    http://www.ftdichip.com/Support/Documents/DataSheets/Cables/DS_TTL-232R_CABLES.pdf
- Development PC
  - STM32CubeIDE
    https://www.st.com/en/development-tools/stm32cubeide.html

The following table shows the jumper settings for the EVAL-AD5700EBZ board.

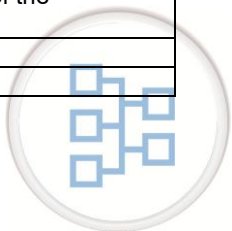
| Jumper | Setting | Description |
|---|---|---|
| CLK_CFG1 | A | No output on CLK_OUTPUT<br>Internal Crystal oscillator enabled |
| CLK_CFG0 | A | |
| XTAL_EN | A | |
| FILTER_SEL | B | Use the internal receive band-pass filter. |
| L1 | A | |
| L2 | A | |
| REF_EN | B | Enables the internal 1.5 V reference and buffer. |
| DUPLEX | A | Operates in half-duplex (controlling by RTS) |
| L5 | Connected | Connect Vcc to IOVcc |
| L6 | Not connected | - |
| L8 | Connected | - |

**Table 8: EVAL-AD5700EBZ jumper settings**

The following table shows the jumper and soldering bridge settings for the STM32L-DISCOVERY board. The settings marked in green color must be used. Except of SB17 (MCO) these are the default factory settings of the board.

| Jumper/Bridge | Setting | Description |
|---|---|---|
| SB18, SB20 (X3 crystal) | ON | PH0, PH1 are connected to P1 (X3, C21, C22, R30 must not be fitted). |
| | OFF | X3, C21, C22 and R30 provide a clock. PH0, PH1 are disconnected from P1. |
| SB7,9,11,13 (DEFAULT) | ON | Reserved, do not modify. |
| SB6,8,10,12 (RESERVED) | OFF | Reserved, do not modify. |
| SB1,2,14<br>(IDD_Measurement) | ON | PA0, PA4, PC13 are used by the IDD measurement. JP1 ON |
| | OFF | PA0, PA4, PC13 are available and IDD module cannot be used. JP1 OFF. |
| SB15,16<br>(X2 crystal) | ON | X2, C16, C17 and R28 deliver a 32 KHz clock.<br>PC14, PC15 are not connected to P1. |
| | OFF | PC14, PC15 are only connected to P1. Do not remove X2, C16, C17, R28. |
| SB5<br>(B2-RESET) | ON | B2 Pushbutton is connectd to the NRST pin of the STM32L152RBT6 MCU |
| | OFF | B2 Pushbutton is not connected the NRST pin of the STM32L152RBT6 MCU. |
| SB4<br>(B1-USER) | ON | B1 Pushbutton is connected to PA0. |
| | OFF | B1 Pushbutton is not connected to PA0. |
| SB21<br>(VDD powered from 3 V) | ON | VDD is powered from 3 V, SB22 must be OFF. |
| | OFF | VDD is not powered from 3 V, SB22 must be ON |
| SB22<br>(Battery enable) | ON | VDD is not powered by the CR2032 battery, SB21 must be ON |
| | OFF | VDD is powered by the CR2032 battery, SB21 must be OFF. |
| SB100 (NRST) | ON | The NRST signal of the CN2 connector is connected to the NRST pin of the STM32L152RBT6 MCU |
| | OFF | The NRST signal of the CN2 connector is not connected to the NRST pin of the STM32L152RBT6 MCU. |
| SB101 (SWO) | ON | The SWO signal of the CN2 connector is connected to PB3. |
| | OFF | The SWO signal is not connected. |
| SB102 (STM_RST) | ON | No incidence on STM32F103C8T6 NRST signal |
| | OFF | STM32F103C8T6 NRST signal is connected to GND |
| SB3 (BOOT0) | ON | The BOOT0 signal of the STM32L152RBT6 MCU is held low through a 510Ω pull-down resistor. |
| | OFF | The BOOT0 signal of the STM32L152RBT6 MCU is held high through a 10kΩ pull-up resistor. |
| SB19 (BOOT1) | ON | The BOOT1 signal of the STM32L152RBT6 MCU is held high through a 10kΩ pull-up resistor. |
| | OFF | The BOOT1 signal of the STM32L152RBT6 MCU is held low through a 510Ω pull-down resistor |
| SB17 (MCO) | ON | STM32F103C8T6 MCO clock signal is not used. |
| | OFF | STM32F103C8T6 MCO clock signal is connected to OSC_IN of the STM32L152RBT6 MCU |
| JP1 | ON | Enable or disable IDD measurement |
| ST-Link discovery | ON | Enable ST-Link Debugging |

**Table 9: STM32L-DISCOVERY jumper settings**



The following table shows the pin usage of the microcontroller in the demo application. It also shows the usage of the pins on the STM32L-DISCOVERY board. For running the demo application the LCD glass must be dismounted to have enough free pins.

| µC Pin | Usage | Eval. Board usage | Demo application usage |
|---|---|---|---|
| BOOT0 | BOOT0 | BOOT0 | BOOT0 |
| NRST | NRST | ST-Link (NRST) | Reset pin |
| PA0 | --- | User button | --- |
| PA1 | --- | (LCD glass) | --- |
| PA2 | USART2_TX | (LCD glass) | Debug USART TX |
| PA3 | USART2_RX | (LCD glass) | Debug USART RX |
| PA4 | --- | Idd measurement | --- |
| PA5 | --- | free/unused | --- |
| PA6 | --- | Touch sensor | --- |
| PA7 | --- | Touch sensor | --- |
| PA8 | --- | (LCD glass) | --- |
| PA9 | USART1_TX | (LCD glass) | HART USART RX |
| PA10 | USART1_RX | (LCD glass) | HART USART TX |
| PA11 | GPIO, input | free/unused | HART CD |
| PA12 | GPIO, output | free/unused | HART RTS |
| PA13 | SWDAT | ST-Link Debugger | ST-Link Debugger |
| PA14 | SWCLK | ST-Link Debugger | ST-Link Debugger |
| PA15 | --- | (LCD glass) | --- |
| PB0 | --- | Touch sensor | --- |
| PB1 | --- | Touch sensor | --- |
| PB2 | BOOT1 | free/unused | BOOT1 |
| PB3 | --- | (LCD glass) | --- |
| PB4 | --- | (LCD glass) | --- |
| PB5 | --- | (LCD glass) | --- |
| PB6 | GPIO, output | LED Blue | LED blue |
| PB7 | GPIO, output | LED Green | LED green |
| PB8 | GPIO, output | (LCD glass) | Test pin |
| PB9 | GPIO, output | (LCD glass) | Test pin |
| PB10 | GPIO, output | (LCD glass) | Test pin |
| PB11 | GPIO, output | (LCD glass) | Test pin |
| PB12 | GPIO, output | (LCD glass) | Test pin |
| PB13 | GPIO, output | (LCD glass) | Test pin |
| PB14 | GPIO, output | (LCD glass) | Test pin |
| PB15 | GPIO, output | (LCD glass) | Test pin |
| PC0 | --- | (LCD glass) | --- |
| PC1 | --- | (LCD glass) | --- |
| PC2 | --- | (LCD glass) | --- |
| PC3 | --- | (LCD glass) | --- |
| PC4 | --- | Touch sensor | --- |
| PC5 | --- | Touch sensor | --- |
| PC6 | --- | (LCD glass) | --- |
| PC7 | --- | (LCD glass) | --- |
| PC8 | --- | (LCD glass) | --- |
| PC9 | --- | (LCD glass) | --- |
| PC10 | --- | (LCD glass) | --- |
| PC11 | --- | (LCD glass) | --- |
| PC12 | --- | free/unused | --- |
| PC13 | --- | Idd measurement | --- |
| PC14 | OSC32_IN | 32.768kHz crystal | --- |
| PC15 | OSC32_OUT | 32.768kHz crystal | --- |
| PH0 | OSC_IN | crystal (unmounted) | Clock from ST-Link controller |
| PH1 | OSC_OUT | crystal (unmounted) | --- |

**Table 10: Demo application microcontroller pin usage**

## 12.2 Software

The purpose of the demo application is…

- … to be able to run the HART Slave Stack on a ‘real’ hardware during development phase.
- … to be able to do the FCG data link layer and application layer tests.
- … to give the user an example for the usage of the stack.

In the demo application the worst case is chosen for the call period time for ‘hart_loop()’.

### 12.2.1 C modules overview

| File(s) | Folder | Short description |
|---|---|---|
| main.c<br>main.h | appli | **Main module**<br>This module contains the main() function of the demo application. It implements a simple round robin scheduler with 3 Tasks: TaskFast, TaskMedium and TaskSlow.<br>The HART Slave Stack is initialized and the working function (hart_loop()) is called in the TastFast every 10ms. |
| eeprom.c<br>eeprom.h | generic | **EEPROM driver (non-volatile data handler)**<br>This module implements the non-volatile data handler for the demo application as described in chapter “6.9 Non-Volatile data handling”. The non-volatile data is stored in the internal EEPROM memory of the STM32 controller. An intelligent algorithm is used which compares the new RAM mirror content and the EEPROM content. Only if data is changed write operations are performed. |
| process.c<br>process.h | appli | **Process values module**<br>This module handles the device variables of the demo application. In the transmitter demo, to get realistic values, a mechanism is implemented which causes the process values to jitter a little bit. |
| loopcurrent.c<br>loopcurrent.h<br>(exist only in actuator demo) | appli | **Loop current module**<br>This module handles the loop current of the actuator demo application. To get realistic values, a mechanism is implemented which causes the loop current value to jitter a little bit. |
| systime.c<br>systime.h | generic | **System timer module**<br>This module handles the system timer. It initializes the Systick timer of the STM32 which is used as time base for the HART Slave Stack. |
| tp.c<br>tp.h | generic | **Testpin module**<br>This module contains some functions to set testpins. The testpins are used for timing measurements with the oscilloscope during development of the HART Slave Stack. |
| ui.c<br>ui.h | generic | **User interface module**<br>This module contains functions to access the user button and the two LEDs of the STM32 Discovery Board. |
| iso3166.c<br>iso3166.h | generic | **ISO3166 module**<br>This module contains functions to check the country codes according to ISO3166. |
| unit_convert.c<br>unit_convert.h | generic | **Unit conversion module**<br>This module contains functions to do the unit conversion of the DV values. |
| global.h | generic | **Global typdefs and #defines** |
| dbg_global.h<br>dbg_hal.c<br>dbg_hal.h<br>dbg_iface.c<br>dbg_iface.h<br>dbg_iostream.c<br>dbg_iostream.h<br>dbg_syslog.c<br>dbg_syslog.h<br>dbg_terminal.c<br>dbg_terminal.h | dbg_iface | **Debug interface**<br>Debug outputs from the HART Slave Stack (H_DBGOUT()) are mapped to this software module.<br>It contains a generic debug interface implementation with terminal menu and syslog functionality. The hardware abstraction layer is implemented in dbg_hal.c and can be adapted to any custom target. For the demo application, USART2 is used. Transmission is done with a DMA channel without any interrupt. Reception is done via interrupt without DMA. |

**Table 11: Demo application module overview**

### 12.2.2 Device specific test commands

The following device specific commands are implemented in the demo application.

<table>
<tr><th>Command</th><th>Short description</th></tr>
<tr><td>128<br>(transmitter demo)</td><td>Test command to simulate some device states which are necessary during the FCG conformance tests.<br>The following table defines the actions performed by this command.<br>The command has no response data bytes and the response code is always ‘success’.
<table>
<tr><th colspan="3">Request data</th><th rowspan="2">Description</th></tr>
<tr><th>Byte0</th><th>Byte 1</th><th>Byte 2…5</th></tr>
<tr><td rowspan="2">0</td><td>1</td><td>---</td><td>Set Write Protection = active (protected)</td></tr>
<tr><td>0</td><td>---</td><td>Set Write Protection = inactive (not protected)</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>[val]</td><td>Set DV0 to given float value [val]</td></tr>
<tr><td>1</td><td>[val]</td><td>Set DV1 to given float value [val]</td></tr>
<tr><td rowspan="2">2</td><td>1</td><td>---</td><td>Set device armed for cmd #73 "Find device"</td></tr>
<tr><td>0</td><td>---</td><td>Set device NOT armed for cmd #73 "Find device"</td></tr>
<tr><td rowspan="2">3</td><td>1</td><td>---</td><td>Set alarm state active</td></tr>
<tr><td>0</td><td>---</td><td>Set alarm state inactive</td></tr>
<tr><td>4</td><td>---</td><td>---</td><td>Change status bit and set more status available flag</td></tr>
<tr><td>5</td><td>val</td><td></td><td>Adopt value of val to user defined sample data.</td></tr>
</table></td></tr>
<tr><td>128<br>(actuator demo)</td><td>Test command to simulate some device states which are necessary during the FCG conformance tests.<br>The following table defines the actions performed by this command.<br>The command has no response data bytes and the response code is always ‘success’.
<table>
<tr><th colspan="5">Request data</th><th rowspan="2">Description</th></tr>
<tr><th>Byte0</th><th>Byte1</th><th>Byte2</th><th>Byte3</th><th>Byte4</th></tr>
<tr><td rowspan="2">0</td><td>1</td><td>---</td><td>---</td><td>---</td><td>Set Write Protection = active (protected)</td></tr>
<tr><td>0</td><td>---</td><td>---</td><td>---</td><td>Set Write Protection = inactive (not protected)</td></tr>
<tr><td>1</td><td colspan="4">[val]</td><td>Set Loop current to given float value in mA [val]</td></tr>
<tr><td rowspan="2">2</td><td>1</td><td>---</td><td>---</td><td>---</td><td>Set device armed for cmd #73 "Find device"</td></tr>
<tr><td>0</td><td>---</td><td>---</td><td>---</td><td>Set device NOT armed for cmd #73 "Find device"</td></tr>
<tr><td rowspan="2">3</td><td>1</td><td>---</td><td>---</td><td>---</td><td>Set alarm state active</td></tr>
<tr><td>0</td><td>---</td><td>---</td><td>---</td><td>Set alarm state inactive</td></tr>
<tr><td>4</td><td>---</td><td>---</td><td>---</td><td>---</td><td>Change status bit and set more status available flag</td></tr>
<tr><td>5</td><td>val</td><td>---</td><td>---</td><td>---</td><td>Adopt value of val to user defined sample data.</td></tr>
</table></td></tr>
<tr><td>129</td><td>Remark: This command is typically disabled.<br>Test command for delayed response mechanism.<br>The command needs approx 10 seconds to finish.<br>The command returns a 2-byte counter which counts the callback function calls.<br>Response code is always ‘success’.</td></tr>
<tr><td>130</td><td>Remark: This command is typically disabled.<br>Test command for burstable device specific command.<br>The command returns a free running 32-bit counter value.<br>Response code is always ‘success’.</td></tr>
<tr><td>64768</td><td>Remark: This command is typically disabled.<br>Test command for high number commands (16-bit).<br>The command has no response data bytes, and the response code is always ‘success’.</td></tr>
</table>

**Table 12: Demo application device specific commands**

# 13 Stack verification / certification process

The HART Slave Stack is tested with the FCG test kits. The data link layer and application layer tests are passed. Furthermore, MESCO developed additional tests to increase the test coverage.

The following figure shows the typical process until the FCG certification. MESCO Engineering performs the FCG certification tests with the official test kits from the FCG. Optionally MESCO Engineering can also do the porting of the stack and the application development.

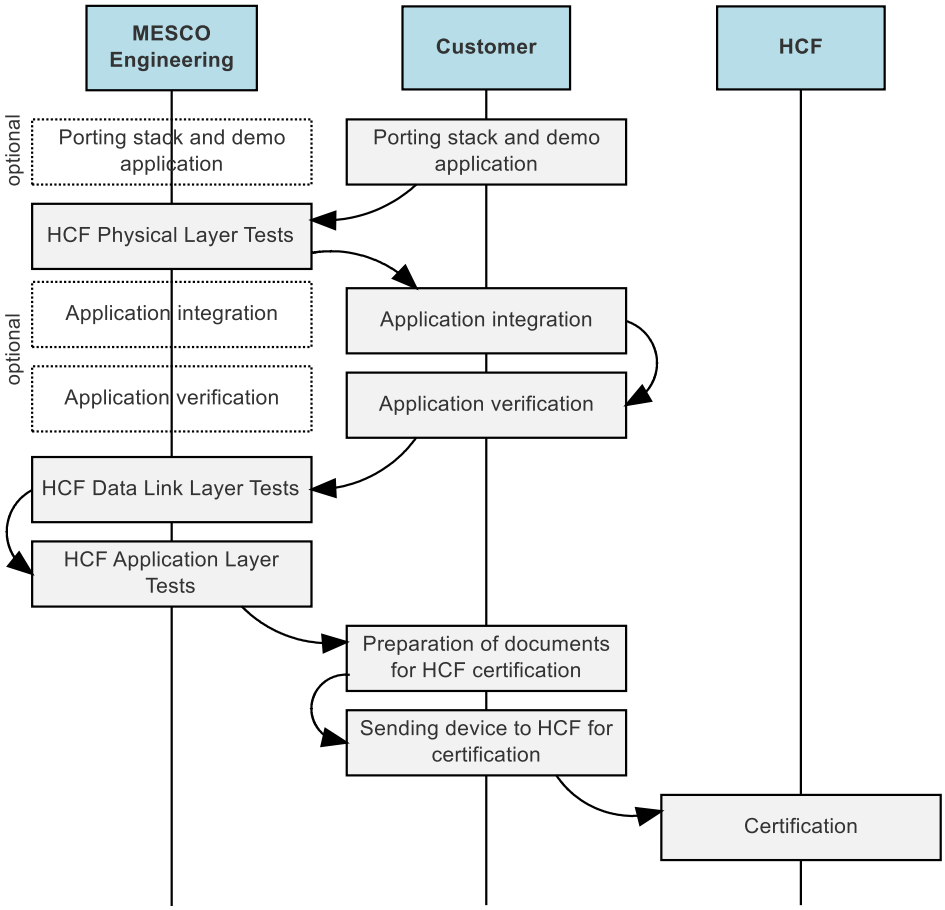



**Figure 15 Typical project process until FCG certification**

# 14 Appendixes

## 14.1 Supported HART commands

| No. | Type *) | Command Name | Feature / Command group |
|---|---|---|---|
| 0 | Uni | Read Unique Identifier | Identity |
| 1 | Uni | Read Primary Variable | Dynamic Variables |
| 2 | Uni | Read Loop Current And Percent Of Range | Dynamic Variables |
| 3 | Uni | Read Dynamic Variables And Loop Current | Dynamic Variables |
| 6 | Uni | Write Polling Address | Data Link Layer |
| 7 | Uni | Read Loop Configuration | Device Management |
| 8 | Uni | Read Dynamic Variable Classifications | Dynamic Variables |
| 9 | Uni | Read Device Variables with Status | Device Variable |
| 11 | Uni | Read Unique Identifier Associated With Tag | Identity |
| 12 | Uni | Read Message | Identity |
| 13 | Uni | Read Tag, Descriptor, Date | Identity |
| 14 | Uni | Read Primary Variable Transducer Information | Dynamic Variables |
| 15 | Uni | Read Device Information | Device Management |
| 16 | Uni | Read Final Assembly Number | Identity |
| 17 | Uni | Write Message | Identity |
| 18 | Uni | Write Tag, Descriptor, Date | Identity |
| 19 | Uni | Write Final Assembly Number | Identity |
| 20 | Uni | Read Long Tag | Identity |
| 21 | Uni | Read Unique Identifier Associated With Long Tag | Identity |
| 22 | Uni | Write Long Tag | Identity |
| 33 | Com | Read Device Variables | Device Variable |
| 34 | Com | Write Primary Variable Damping Value | Dynamic Variables |
| 35 | Com | Write Primary Variable Range Values | Dynamic Variables |
| 36 | Com | Set Primary Variable Upper Range Value | Dynamic Variables |
| 37 | Com | Set Primary Variable Lower Range Value | Dynamic Variables |
| 38 | Uni | Reset Configuration Changed Flag | Device Management |
| 40 | Com | Enter/Exit Fixed Current Mode | Device Management |
| 41 | Com | Perform Self Test | Special functions |
| 42 | Com | Perform Device Reset | Special functions |
| 43 | Com | Set Primary Variable Zero | Dynamic Variables |
| 44 | Com | Write Primary Variable Units | Dynamic Variables |
| 45 | Com | Trim Loop Current Zero | Trim loop current |
| 46 | Com | Trim Loop Current Gain | Trim loop current |
| 47 | Com | Write Primary Variable Transfer Function | Dynamic Variables |
| 48 | Uni | Read Additional Device Status | Diagnosis |
| 49 | Com | Write Primary Variable Transducer Serial Number | Dynamic Variables |
| 50 | Com | Read Dynamic Variable Assignments | Dynamic Variables |
| 51 | Com | Write Dynamic Variable Assignments | Dynamic Variables |
| 52 | Com | Set Device Variable Zero | Trim device variable |
| 53 | Com | Write Device Variable Units | Device Variable |
| 54 | Com | Read Device Variable Information | Device Variable |
| 55 | Com | Write Device Variable Damping Value | Device Variable |
| 56 | Com | Write Device Variable Transducer Serial No. | Device Variable |
| 59 | Com | Write Number Of Response Preambles | Data Link Layer |
| 71 | Com | Lock Device | Device Management |

| | | | |
|---|---|---|---|
| 72 | Com | Squawk | Special functions |
| 73 | Com | Find Device | Special functions |
| 76 | Com | Read Lock Device State | Device Management |
| 78 | Com | Read Aggregated Commands | Aggregated Commands |
| 79 | Com | Write Device Variable | Device Variable |
| 80 | Com | Read Device Variable Trim Points | Trim device variable |
| 81 | Com | Read Device Variable Trim Guidelines | Trim device variable |
| 82 | Com | Write Device Variable Trim Points | Trim device variable |
| 83 | Com | Reset Device Variable Trim | Trim device variable |
| 89 | Com | Set Real-Time Clock | Device Management |
| 90 | Com | Read Real-Time Clock | Device Management |
| 95 | Com | Read Device Communications Statistics | Device Management |
| 100 | Com | Write Primary Variable Alarm Code | Device Management |
| 103 | Com | Write Burst Period | Burst Mode |
| 104 | Com | Write Burst Trigger | Burst Mode |
| 105 | Com | Read Burst Mode Configuration | Burst Mode |
| 106 | Com | Flush Delayed Responses | Delayed Response |
| 107 | Com | Write Burst Device Variables | Burst Mode |
| 108 | Com | Write Burst Mode Command Number | Burst Mode |
| 109 | Com | Burst Mode Control | Burst Mode |
| 512 | Com | Read Country Code | Identity |
| 513 | Com | Write Country Code | Identity |
| 516 | Com | Read Device Location | Identity |
| 517 | Com | Write Device Location | Identity |
| 518 | Com | Read Location Description | Identity |
| 519 | Com | Write Location Description | Identity |
| 520 | Com | Read Process Unit Tag | Identity |
| 521 | Com | Write Process Unit Tag | Identity |
| 523 | Com | Read Condensed Status Mapping Array | Condensed Status |
| 524 | Com | Write Condensed Status Mapping | Condensed Status |
| 525 | Com | Reset Condensed Status Map | Condensed Status |
| 526 | Com | Write Status Simulation Mode | Status simulation |
| 527 | Com | Simulate Status Bit | Status simulation |
| 534 | Com | Read Device Variable Command code | Device Variable |

**Table 13: Supported Universal and Common Practice Commands**

*) Uni = Universal command; Com = Common Practice Command

## 14.2 Acronyms and abbreviations

| Abbreviation | Detailed / Complete |
|---|---|
| AL | Application Layer |
| API | Application programming interface |
| CPU | Central Process Unit |
| DLL | Data Link Layer |
| DUT | Device Under Test |
| FCG | FieldComm Group |
| FSK | Frequency shift keying |
| FW | Firmware = Embedded Software |
| HAL | Hardware abstraction layer |
| HCF | HART Communication Foundation |
| HW | Hardware |
| LRV | Lower Range Value |
| N.C. | Not Connected |
| PHL | Physical Layer |
| SW | Software |
| /TBD/ | To Be Defined |
| µC or uC | Microcontroller |
| URV | Upper Range Value |
| Ver. | Version |

**Table 14: Acronyms and abbreviations**

**MESCO Systems GmbH**
Berner Weg 7
79539 Lörrach
Germany

Tel. +49 7621 1575 475
Fax +49 7621 1575 175

info@mesco-systems.com