Final Master's Project

University Master's Degree in Industrial Engineering

Hart slave emulator based on MicroPython

REPORT

Author: Pau Ferrer Almirall
Supervisor: Manuel Moreno Eguílaz

Date: June 2020

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Abstract

This project makes a general study of the HART communication protocol and develops a tool to simulate different types of field devices using Python. This will give the Department of Electronic Engineering at ETSEIB a cheap way to test any developed HART device.

The process to accomplish this goal is well described in this report. At first the focus is set on studying the protocol and then it is time for the development of the software. This software uses a wide range of modules and feeds the information for the slave definition from Json DDFiles. All developed functionalities for the emulation environment have been conceived with the intention of giving the tool maximum adaptability and expandability.

Once the software has been developed, a thorough test plan is applied in different phases. At first with a handmade and controlled environment, and then, pairing the slave environment with commercial software. The results are satisfactory since all functionalities are working properly.

Summary

1. Glossary

ACK:	*Acknowledge frame (Slave to Master frame)*
BACK:	*Burst Acknowledge frame*
CPU:	*Central Processor Unit*
EDD / DD:	*Electronic Device Description / Device Description*
FDT:	*Field Device Tool*
FSK:	*Frequency Shift Keying*
HART:	*Highway Addressable Remote Transducer*
HCF:	*Hart Communication Foundation*
HHT:	*HART Handheld Terminal*
ID:	*Identifier*
ISO:	*International Standards Organization*
LSB:	*Least Significant Bit*
MPI:	*Multi-Point Interface*
MS:	*Master Slave*
MSB:	*Most Significant Bit*
OSI:	*Open Systems Interconnection*
PLC:	*Programmable Logic Controller*
PV:	*Primary Variable*
RTS:	*Request-To-Send*
STX:	*Start of text (Master to Slave frame)*
UART:	*Universal Asynchronous Receiver/Transmitter*

2. Preface

2.1. Origin of the project

This project was proposed by the Electronics Department of ETSEIB in its repository under the name of ”Hart slave emulator based on MicroPython”. After working for three years in the Mechanical Engineering Department of a Baggage Handling Systems company, I felt the need to redirect my studies, and after having fully completed the subjects of the Mechanical branch of MUEI, this was the only chance to get a deeper knowledge of other fields of interest such as Industrial Automation and Communication Protocols.

2.2. Motivation

The main motivation of this project came with the need of the Department of Electronic Engineering at ETSEIB to develop a tool to properly test different Systems based on HART networks. HART sensors are an expensive hardware, which makes it difficult to set up a functional network with multiple slaves in it, hence the idea of a virtual simulation for such environment.

A network of simulated sensors offers a great flexibility at a much reduced expense, and requires almost no time to be set up and running once the tool has been developed. With this emulated network, the Department of Electronic Engineering will be able to test any kind of modem and software for HART networks management in a fully customizable way, allowing the emulation of any kind of sensor in the market with any given parameters.

2.3. Previous work

To fulfil the presented motive, a previous study of all the elements involved in the project has been necessary. First an introduction to serial port communication via MicroPython was required, as well as virtual serial port communication tools in order to develop and test the software. But most importantly, a deep study of the HART communication protocol has been done in order to fully comprehend its logic and procedures with the goal of emulating slaves capable of being recognized by HART commercial software.

Certain knowledge of python programming language is also required, but this has mostly been obtained during university years, and for this project it has not carried more study than a few specific modules and its methods.

3. Introduction

3.1. Objectives

The first objective of this project is to get a deep understanding of the HART communication protocol. To do this, Official HART protocol specification documents and other bibliography detailing the most important aspects of this protocol will be studied. With the gathered information in these documents the project will include a summary later in this section for the proper understanding of the later developed works.

After the protocol has been understood and assimilated, the project will focus on the practical scope and, according to the motivation, develop a tool for the Department of Electronics Engineering at ETSEIB that enables the testing of software and hardware with a simulated environment of HART sensors. This environment must be adaptable for each project specific requirements, and must also be easily modifiable, enabling the user to add and test more functionalities in a comfortable way without the need to deeply modify the core software.

As an additional requirement made by the Department of Electronics Engineering, the files containing the definition and information of the HART slaves should use the well-known Json format, for further standardization and the ability to request the slave information from suppliers in a standard easy format.

3.2. Scope

To fulfil the aforementioned objectives, the scope will include the development of the following elements:

- A process emulator that communicates with the master via virtual ports, and receives and sends the requested information both from the master and slaves.
- A slave emulator that contains the slave definition and all supported commands. This will interact with the process emulator and send the required information for each command.

The scope also includes the testing of all the developed elements and commands using a commercial tool to prove that the developed software can work in any environment.

3.3. State of the art

There are currently few offers of software in the market that supply services for HART protocol testing. In this project, there is a need for a commercial HART Master emulator and the developed Slave emulator. Below a list of products that are available in the market, both for the commercial and developed parts:

3.3.1. Master HART tools

These software packages offer an interface through which the user may send commands and check the returned response. For this project, they will use a virtual communications port and be executed from a PC. Some options available on the market are:

- HartTools 2.0: offers a wide range of commands and modes, even counts with a “Raw command” option that enables the user to manually determine each byte to be sent to the field devices.

- Hart Tools 7.4: more professional oriented software. Uneasy to run simple tests in order to check the software development.

Aside from these, other open-source software can be found in repositories like GitHub. These offer user oriented solutions developed for a wide range of purposes and detail. Companies like Borst-Automation, Utthunga or Control automation also offer non-free tools for Master emulation.

For its availability, comfort and functionality HartTools 2.0 is the selected software to be used during the test phase of this project.

3.3.2. Slave HART emulator

There are not many software emulators available on the market, even less open source ones. As well as with the master emulators, accessing GitHub or other software repositories may be an option in order to find a piece of software, but most likely it will require some tuning to align it with the user interests.

The most relevant professional options found are:

- HART Slave Simulator from e-FLOWNET, a software that allows the user to simulate up to 16 slaves and their sensor values. It only supports up to HART revision 5 and a small list of commands (0, 1, 3, 14 and 35).

- HART HYBIRD SLAVE SIMULATOR from FullChem-SZ. It offers a much more customizable slave than the above option and also supports burst mode. It is HART 7 compatible.

The mentioned options are non-free, although it has not been possible to determine their price. In any case, this project will cover the necessities of the Department of Electronic Engineering in a more suitable way, since it will include all the required functionalities and it will also be easily modifiable and expandable.

4. HART Background

The following section serves the very first objective of this project, the study and comprehension of HART protocol. In order to transmit the information gathered in the clearest way, some parts of this section are direct extracts from references listed in the bibliography. This has been done after a thorough study of different sources, and after deciding which pieces of information were most valuable and in which order. Since what follows gives an overall view of the protocol, the subsections that refer to matters applicable to the practical part of this work have been expanded so that to ease the understanding of the application sections. [10].

4.1. Introduction

The HART protocol was developed in the mid-1980s by Rosemount Inc. for use with a range of smart measuring instruments. The protocol was later published for free use by anyone, and in 1993, the registered trademark and all rights in the protocol were transferred to the HART Communication Foundation (HCF). However, the protocol remains open and free for all to use without royalties. The HART specifications have been improved and extended over the years, always under the control of its users through the HCF, with efforts to preserve full compatibility with existing products. [1].

The description “smart” for a field device was first used in the sense of “intelligent” to describe any device that included a microprocessor. Typically, this would imply extra functionality beyond that previously provided.

The majority of process automation equipment utilizes a milliampere (mA) analog signal for field communication. In most applications the signal varies within a range of 4-20 mA in proportion to the process variable being represented. The HART (Highway Addressable Remote Transducer) Protocol is an industry standard protocol for sending and receiving digital information across analog wires between field devices and control and monitoring systems. It preserves the traditional 4-20mA signal, and provides simultaneous transmission of digital communication signals on the same wiring, thus enabling a bi-directional communication with smart instruments without disturbing the 4-20 mA analog signal. In that way primary process variables and control signal information are carried by the 4-20 mA, while additional process measurements, device configuration and parameter information, calibration, and diagnostics information is accessible through the HART protocol.

4.2. HART devices and networks

HART is supported by basically two types of devices: master (host) and slave (field) devices. HART field devices include transmitters, sensors and various actuators that acknowledge commands from the primary or secondary master. This variety ranging from two wired or four wired devices to intrinsically safe versions that are to be operated in hazardous conditions Host (master) devices are usually a PLC, a handheld terminal, a computer based central control or monitoring system or a DCS.

Basically the HART data is superimposed on the 4-20 mA current loop making use of the frequency shift keying (FSK) principle, via a FSK modem integrated in field devices. This enables devices to communicate digitally using the HART protocol, while analog signal transmission takes place at the same time.

Hart devices have the capability to operate in one of two network configurations: point-to point connection or multidrop mode. According to the polling address structure of the HART protocol, the polling address of slave devices may range from 0 to 15. [2], [3], [4].

4.2.1. Point-to-point

This is the most common use of HART technology. In this mode, the HART master device is connected to exactly one HART field device. This communication capability allows devices to be configured and set-up for specific applications. The 4-20 mA signal carries one process (primary) variable, while additional data showing status, secondary variables, configuration parameters and preventive maintenance information are transferred digitally using the HART Protocol.

This connection alternative requires that the device address of the field device to be always set to zero since the HART master uses this address to establish communication. The analog 4-20 mA signal can be used for control in this point-to-point mode due to the digital signal is superimposed on the analog signal and does not interrupt or interfere with it. [2], [3].

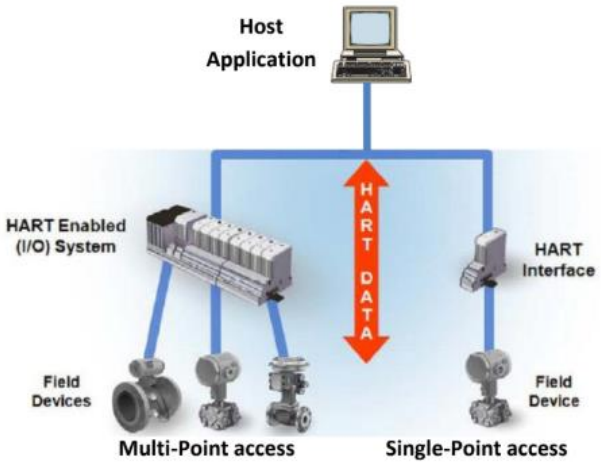

Figure 1: Point-to-point communication. Source: [2]

4.2.2. Multidrop

In multidrop operation all process values are transmitted entirely digital, i.e. only via the HART protocol. The communication protocol enables the capability to connect several two-wire measurement devices in a (typical) multidrop network configuration. Depending on the protocol revision, HART 5 or HART 7, can up to 15 or 62 devices be connected in parallel to a single wire pair. In addition, properly connected two-wire loop powered and four-wire active-source devices can be used in the same network.

All the devices are supplied from one voltage source and with a constant current consumption (usually 4 mA). Therefore, the analog current loop does not change, and it ceases to have any meaning relative to the process due to the analog signal that enters the master represents the sum of all the analog signals belonging to the devices in the network.

The host distinguishes the field devices by their pre-set polling addresses (or tag numbers) that must be unique in a range of 1-15 (potentially of 1-63). This address can be set by sending a special command to the devices. Standard HART commands are used to communicate with field instruments to determine process variables or device parameter information.

It can take several seconds before the HART signal, superimposed on the analog signal, follows the corresponding analog value. Process variables (normally transmitted on the 4-20 mA analog signal) are therefore much faster than that of the more accurate digital HART value.

The communication rate of the HART protocol in multidrop networks is perceived as too slow when it comes to monitoring and control of essential processes. Standard HART

commands are employed when it comes to communicate with field devices in order to determine process variables or device parameter information. Using HART protocol an average of 2-3 digital updates per second are expected, giving a typical cycle time of 500 ms to read information on a single variable from a HART device. Hence for a network of 15 devices, a total of approximately 7.5 seconds is needed to scan and read the primary variables from all devices. As the data field will usually contain values for up to four variables rather than just one, reading information from multivariable instruments may take longer. [2], [3], [4].

Significant reduction in the amount of wiring is among the advantages that multidrop network

brings. In addition, more than 15 devices are possible to connect in a multidrop network when the devices are individually powered and isolated.

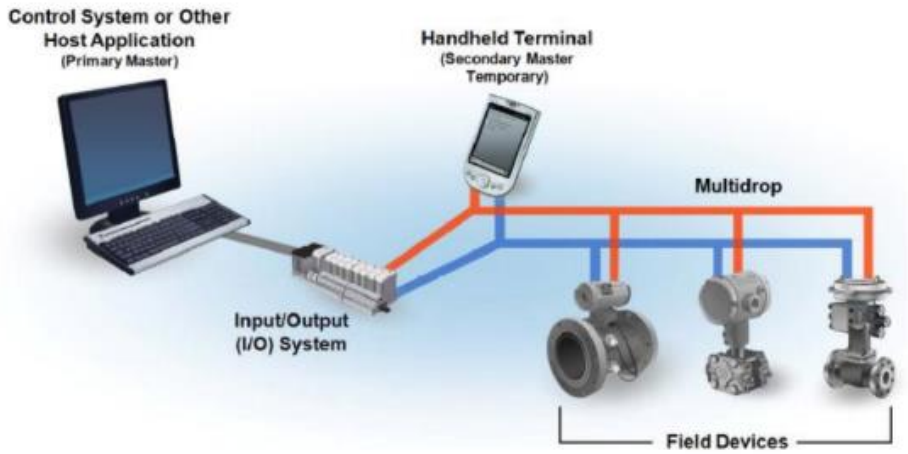

Figure 2: Multidrop connection. Source: [2]

4.3. HART Protocol and the OSI-model

The HART protocol is built on the Open Systems Interconnection (OSI) Model, developed by the International Standards Organization (ISO) [6].The OSI model describes the structure and elements needed for a communications system, sub-dividing it into layers where a collection of similar functions provides and request service from the layer above and below, respectively.

A simplified OSI-model is used by the HART Protocol. It implements layers 1,2,3,4 and 7 of the OSI 7-layer protocol model. The current HART Protocol is revision 7.5, where the "7" denotes the major revision level and the "5" denotes the minor revision level [5]. Table 1

describes the OSI model layers and their functions:

Layer		Function	HART
7	Application	Formatting of data	Commands and data formats
6	Presentation	Conversion of data format	
5	Session	Management of communication dialog	
4	Transport	End-to-end error recovery	
3	Network	Switching and routing for network connections	
2	Data link	Message format and media access	Message format and transaction rules
1	Physical	Physical connections and signals	FSK signal

Table 1: The OSI 7-layer protocol model. Source: [10].

The information forming a message passes down through the layers in one device, along the wire, then up through the corresponding layers in the other device, as depicted in Fig. 3. In case of a single local network, with master-slave operation based on single standalone transactions and without automatic retries for lost or corrupted messages, layers 3 to 6 are either not necessary or are much reduced. This leads to the outstanding simplicity of HART compared with other protocols. [1].

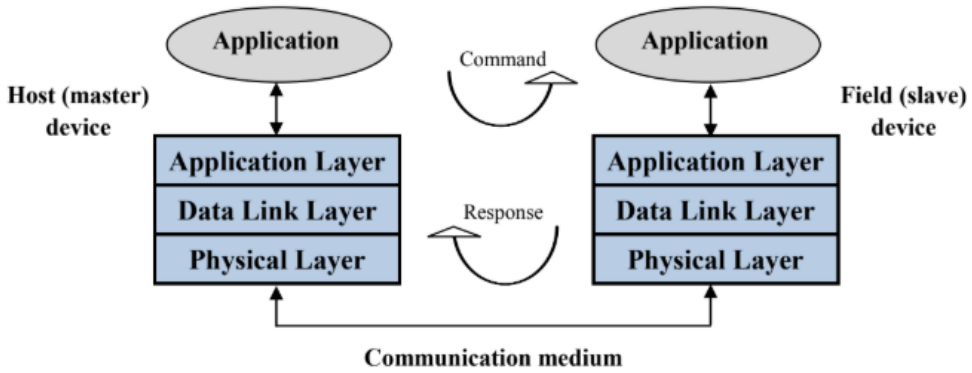

Figure 3: HART and the OSI model. Source: [1].

4.4. HART Communication Layers

Layer 1 is the HART Physical Layer which defines the physical and electrical specifications of the relationship between a HART device and the transmission medium (cable). Modulation is among the major functions and services performed by this layer. The HART Physical Layer makes use of the Bell 2022 Frequency Shift Keying (FSK) principle to superimpose digital communication signals at a low level on the top of the 4-20 mA signal, and communicates at 1200 bits per second. The signal frequencies representing bit values of 0 and 1 are 2200 and 1200Hz respectively. The pulse amplitude of ± 0.5 mA averages out to zero as not to falsify the measured signal. [5]

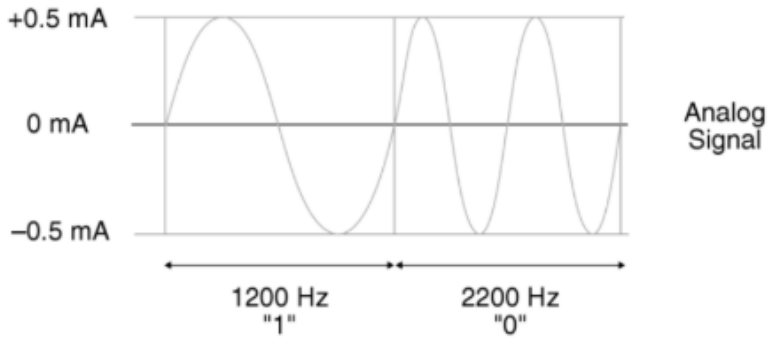

Figure 4: The HART FSK signal. Source [10].

Both frequencies form sine waves which are superimposed on the direct current (dc) analog signal cables in order to provide simultaneous analog and digital communications. Because the average value of the FSK signal is always zero, the 4-20 mA analog signal is not affected. [2].

While the Physical Layer sends the current of bits, the HART Data Link Layer (Layer 2) provides the functional and procedural means that will ensure reliable sending of data. It is in this layer where the HART protocol technical format is specified. The user can define here the communication Mode, either “Master-Slave” Mode, where a field device only replies when it is spoken to, or “Burst Mode”, which allows a single slave device to continuously broadcast a standard HART reply message. [2].

The Network Layer (Layer 3) provides routing, end-to-end security, and transport services, i.e. the operational and methodology means of transferring variable length data sequences from a source host to a destination host on separate networks. Furthermore, this layer manages "sessions" for end-to-end communication with corresponding devices.

The Transport Layer (Layer 4) provides and ensures successful direct transfer of data between end users. Typical examples of this layer are the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

Layer 7 is the HART Application Layer that defines the commands, responses, data types and status reporting supported by the Protocol. The commands make it possible for a system, connected to HART devices, to both collect and send data to the field devices. This can for example be collection of temperature measurements from a transmitter, or write in a name ("Tag") to the transmitter that can be used for device identification. Software applications implement a communication component that makes it possible to both the application layer and the user interacts directly with the application. [5], [6].

In the Application layer, the public commands of the protocol are divided into three major groups [2]:

1. Universal Commands - all HART compatible devices must recognize and support these commands.
2. Common Practice Commands - provide functions common to many, but not all HART communication devices.
3. Device Specific Commands - provide functions that are unique to each field device and are specified by the device manufacturer.

A fourth one called Device Family Commands may be defined. It provides a set of standardized functions for instruments with particular measurement types, allowing full generic access without using device-specific commands.

4.5. Physical Layer

The Physical Layer connects to a medium such as a cooper or optical cable, which serves all of the communicating systems. It describes how HART circuit/loops can be build up, which type of cable could be employed as well as its longitude limits, and which signal components take part of the HART circuit/loops. This includes layouts of pins, voltages, impedances, hubs, repeaters, network adapters, etc. The layer also describes the communication signal and the frequencies used for the digital communication.

In addition to interfacing to the network cable, the HART Physical Layer performs 4 basis functions [5]: modulating an outgoing message, demodulating an incoming message, turning on carrier for an outgoing message, detecting carrier for an incoming message.

4.5.1. Coding

HART is mainly a master/slave protocol. The data transmission between the masters and the field devices (slaves) is physically realized by superimposing an encoded digital signal on the 4 to 20 mA current loop. Since the coding has no mean values, an analog signal transmission taking place at the same time is not affected. This enables the HART protocol to include the existing simplex channel transmitting the current signal and an additional half-duplex channel for communication in both directions.

The bit transmission layer defines an asynchronous half-duplex interface which operates on the analog current signal line. To encode the bits, the FSK method is used and the two digital values, 1 and 0, have assigned frequencies of 1200 and 2200 Hz respectively.

HART uses FSK with a data rate of 1200 bit per second, which is a very low rate compared with field buses. In HART, both a logical 1 and a logical 0 operate in an equal time interval in order to obtain a correct data rate. Thus, a logical 1 is represented by a single cycle of 1200 Hz, while a logical 0 is represented by almost two cycles of 2200 Hz. [3], [4], [8].

4.5.2. Digital and Analogue frequency

The analogue signal, where the digital signal (information) is superimposed on, has usually a frequency between 0 and 20 Hz. Thus the frequency to the digital signal lies much higher than the analogue's. Ideally and for that reason none of the signals (digital and analogue) will affect each other. Filters (high- and low-pass) are used in order to separate both communication channels. The bandwidth occupied by the HART signal is conventionally stated as 950 Hz to 2500 Hz (see Fig. 5).

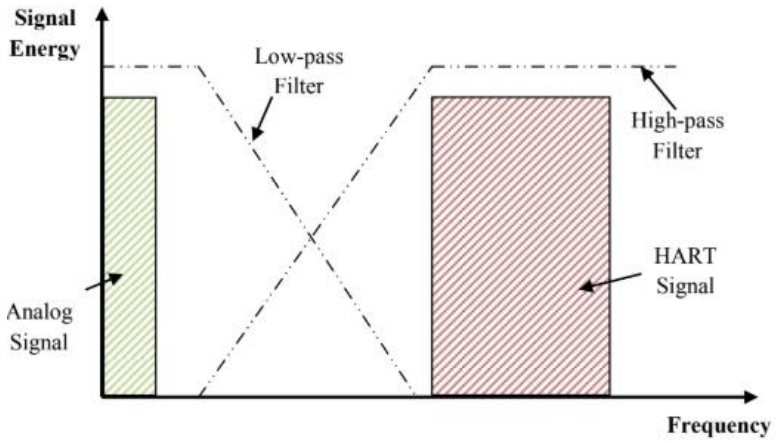

Figure 5: Analog and HART (digital) signal frequencies. Source: [1].

HART Communication between two or more devices can only function properly when all the

communication participants have the capacity to interpret the HART sine wave signals correctly. Field devices, with inputs and outputs, in the current loop, which are not specified only for the 4 to 20 mA technology, can impede or prevent the transmission of the data. Since the input and output resistances change with the signal frequency, such devices are likely to short-circuit the higher frequency HART signals (1200 to 2200 Hz).

4.6. Data Link Layer

This layer defines a request-response protocol. Any communication activity is initiated by the host communication device (master), which is either a control station or an operating device, and a (slave) field device only reply when it is spoken to. In other words the Data Link layer describes how the elements in a HART network communicate with each other. The layer gives a description of the messages' structure that are to be sent, and how they are identified by the right receiver (slave devices). It makes it possible to detect and correct errors that may occur in the Physical Layer. [2], [4], [8].

4.7. Master - Slave Protocol

HART is a protocol for Master/Slave communication that supports up to two master devices. The primary devices - generally a DCS, PLC - send HART messages with commands, communicating with the field devices which receive the command messages. The command specifies which information the master is looking for to get from the slave. Primary master modules connected with the slave devices, constitute a HART circuit/loop. The secondary master devices – for instance a PC/laptop or a HART Handheld Terminal (HHT) - can be hooked up to the HART circuit/loop or directly to the field device. The two masters would have different addresses, 1 and 0, respectively. The use of handheld terminals is quite common within maintenance personnel of HART field devices, in order to collect data and to set the instrument parameters. [3], [4].

HART field instruments- the slaves- respond when they have received a command message from the master. Since HART is a half-duplex master-slave protocol, the master and the slave cannot send a message at the same time, but one of them has to wait while the other sends. Communication is initiated every time the master makes a request, and there is no communication between the different slaves or masters. Hence, instant communication occurs just between one master and one slave.

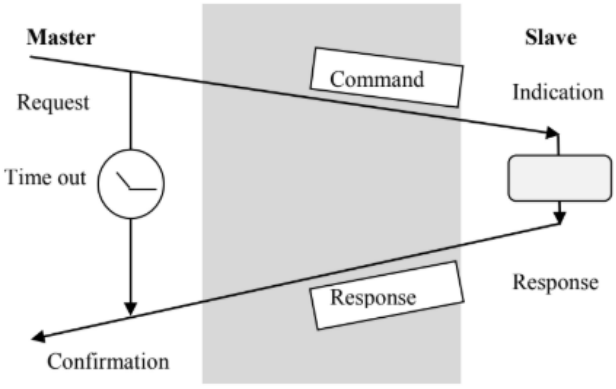

Figure 6: Master - Slave / Request – Response. Source: [3].

In case there are several devices in the network, only one of them can send a respond to the master at a given time. The emitted message by the master contains the address to the slave which the message is intended to. The slave sends right back a response to the message the master sent, showing that the slave received it, and the response contains the information the master requested.

The reply message contains also the address to the master it is aimed to. This will prevent an unintended master to get a response, in the event that two masters are active in a network. Once the data exchange between the control station and the field device is complete, the master will pause for a fixed time period before sending another command, allowing the other master to break in. Both masters observe a fixed time frame when taking turns communicating with the slave devices. [3], [4], [8].

4.8. Communication Modes

The HART protocol provides two modes for communicating information to/from smart field instruments and central control or monitoring equipment. The simplest and most common form is the Request-Response (master-slave) mode communication simultaneous with the 4-20 mA analog signal, where a master telegram is directly followed by a response or acknowledgement telegram from the slave. This mode allows digital information to be updated approximately twice or three times per second in the master, without interrupting the analog signal.

A HART message from a master and its corresponding response from a slave are called a transaction. The two bursts of carrier during a transaction are illustrated in Fig. 7. The master is responsible for controlling the message transaction. If it does not receive any reply from the slave after an expected time (RT1), the master will soon after send the same request again (RT2 – the time a master waits before sending a new request). The message

is dropped after a couple of attempts without getting a response from the slave device, due to a fault in the slave or in the communication loop has most likely arisen.

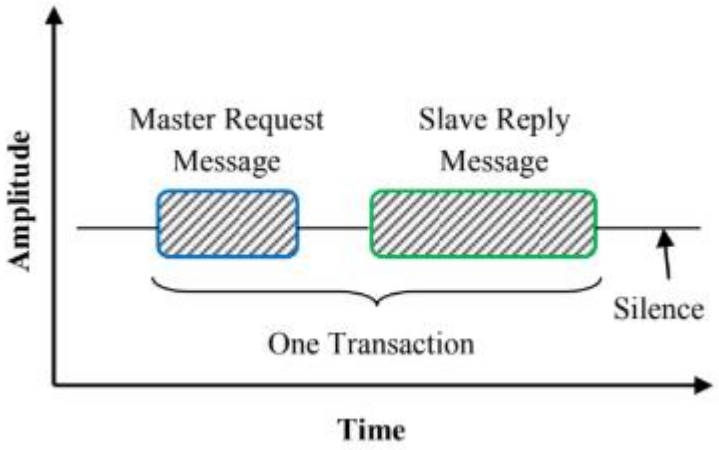

Figure 7: Carrier bursts during HART transaction. Source: [10].

Some HART devices support the optional Burst communication mode. It allows a single slave device to continuously broadcast a selected standard HART reply message such as a primary variable or other. In this way the master is relieved from having to send repeated command requests. A single field device cyclically sends message telegrams with short 75-ms breaks, which can alternately be read by the primary as well as the secondary master. While usually two transactions per second are possible, Burst mode enables faster communication up to 3-4 data updates per second (will vary with the chosen command). The host receives the message at the higher rate until it instructs the device to stop bursting.

In a network it is only one slave at a time that can be in "burst" mode. This communication mode is therefore more relevant for single slave device networks, where the user wishes fast and continuous updates of particular important equipment data. [2], [4], [7], [8].

4.9. HART Character Structure (Character Coding)

A HART message consists of a series of bytes, that go through an UART (universal asynchronous receiver/transmitter) before they are sent from the HART-modules. HART messages are coded as a series of 8-bit characters or “bytes”. These are transmitted serially using the UART function that serializes each byte, adding a start and stop bit (2 bits), and a parity bit (1 bit). Hence, it converts each transmitted byte into an 11 bit serial character.

The original byte that contains the message data becomes the data bit, the start and stop bits are used for synchronization and the parity bit is part of the HART error detection. Thus, these bits are used to identify every character in the receiver UART, and to check if an error in the character has occurred during the transportation from the sender to the receiver.

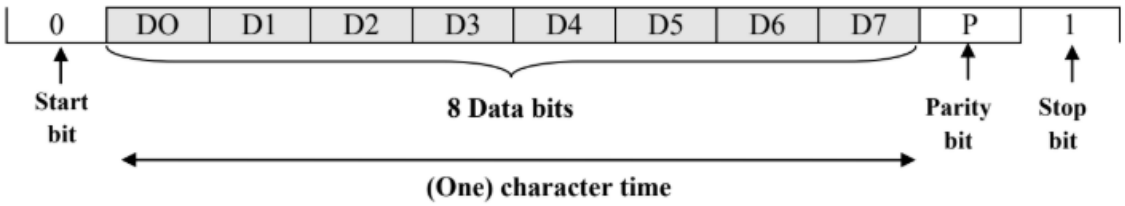

Figure 8: Bit structure of each HART character. Source: [12].

The start bit to each single character is a logical 0, while the stop bit is a logical 1. The other bits in the character vary between 0 and 1, depending on the character.

In HART communication, odd-numbers parity is used for error detection purposes. The 8 data bit determines the value of the parity bit. It is in the sender UART where the value (0 or 1) is set. Furthermore, the number of logical 1's in the 8 bits is summed up in order to check that the sum of them is an odd number. If this is the case, then a logical 0 is set to the parity bit. Otherwise it is set to 1, so that the sum of all 1's in the character becomes an odd number. Consequently, each single character sent from the UART consists of a number of logical 1's that constitute an odd number.

The parities to every character is again checked in the receiver UART, where an error in the transmission over the network is easily detected if the sum of 1's in the character is no longer an odd number. Noise and other disturbances could then be the cause of such errors. [7].

4.10. HART telegram structure and elements

Each command or reply is a message, varying in length from 10 to 12 bytes to typically 20 or 30 bytes. The structure of a HART telegram is depicted in Fig. 9, and it includes a preamble of between 5 to 20 bytes, which will be explained later. Each individual byte is sent as 11-bit UART character equipped with a start, parity and a stop bit. HART provides two telegram formats, long and short format, which use different forms of addressing. On the other hand, the HART message structure is equal for long and short format. [3], [4], [8].

Delimiter	Address	[Expansion Bytes]	Command	Byte Count	[Data]	Check Byte

Figure 9: Telegram structure[1]. Source: [1].

4.10.1. Preamble

This element consists of 5 to 20 characters. All bytes in these characters are set to the logical value 1, in such a way that each character represents the hexadecimal number FF. All the characters will therefore represent a constant sinus signal.

The preamble characters synchronize the signals of the participants, as well as it is employed to detect the start of a message. A message receiver often needs some time to be synchronized with the signal frequency, and the incoming character stream. As well as it could need some time to turn the message stream through the modem after it has sent a message.

The receiving processor expects a sequence of 3 contiguous bytes: preamble, preamble, start delimiter. Thus, at least two good preamble characters must be received and they must be those that immediately precede the start delimiter. Since HART requires a transmission of a minimum of 5 preamble characters, there can be a loss of up to 3 characters during the synchronization at the receiver or by other occurrences, without damaging the message. If a message is received and the receiver sees only one preamble character, then the message will be lost because the receiver module does not manage to detect the start of the incoming message. [1], [3], [4], [8].

Due to variation in number of preamble characters the different slaves need for being synchronized, the first message a master sends always contains 20 preamble characters. This is the maximum number allowed. A longer preamble means slower communication, therefore they should be avoided. To reduce the number of preamble characters needed for every message that is sent; a master can send either command 0 or command 11 to the slaves. The response from the slaves consists of the number of preamble characters the slave wants to receive. Given that the master module particularly demands a specific number of preamble characters, command 59 can be used to tell the slaves. Nowadays slave devices are designed so that they need only a 5 byte preamble (Five preamble characters to be used if no specification is given).

4.10.2. Delimiter Field

The delimiter is the first field in a HART message. It is used for message framing by

indicating the position of the Byte Count. It also indicates the Frame Type which is used for bus arbitration. The delimiter consists of four sub-fields (see Fig. 10):

- The most significant bit, bit number 7, indicates whether a polling (1 byte) or unique (5 byte) address is included in this frame.
- Bits 6 and 5 indicate the number of Expansion Bytes that are in this frame. Normally this field is set to zero.
- Bits 4 and 3 indicate the Physical Layer type. For the (asynchronous) FSK Physical Layer this field is set to 0.
- The least significant 3 bits (bits 2-0) indicate the frame type. These three bits plus the master address bit implies the passing of a token and together they are used for Media Access Control (MAC).

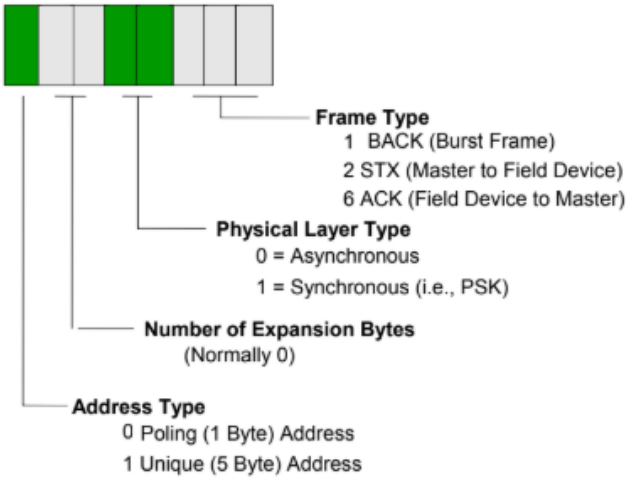

Figure 10: The Delimiter Field. Source: [12].

Frame Types

Three frame types are supported by the HART Data Link Layer:

- An STX (0x02) indicates a Master to Field Device (i.e., a Slave or Burst-Mode Device) frame. An STX is the start of a transaction and is normally followed by an ACK.
- An ACK (0x06) is the Slave's or Burst Mode Device's response to an STX.
- An BACK (0x01) is a Burst Acknowledge frame periodically transmitted by a Burst-Mode Device. These frames are transmitted without a corresponding STX.

4.10.3. Address

The HART Protocol supports both five (5) byte "unique" addresses and one (1) byte polling

addresses. The length of the Address field (1 or 5 bytes) is indicated by the Delimiter. The Address field always includes both the master address, the slave address, and (for ACK and BACK frames) whether the slave is in burst mode.

4.10.3.1. Master Addresses and the Burst Mode Flag

HART Protocol includes the source and destination addresses in each frame. For both unique and polling addresses, the most significant bit of the address field indicates the master associated with this frame. A primary master uses the value "1" for this bit, a secondary master uses the value "0".Slave devices must echo back this field unchanged. The next bit indicates whether the slave device is in Burst-Mode. Slave devices must set this bit to a "1" to indicate that the device is in Burst-Mode or to a "0" if the device is not in Burst-Mode. Master devices must set this bit to "0" in all requests to slaves.

The master address combined with the Frame Type imply a token that indicates to the MAC the next device that should begin a message transmission.

4.10.3.2. Unique Addresses

Except for Command 0, all HART frames consist of a 5 byte address based on the slave device Unique Identifier. A Unique Identifier is associated with each field instrument manufactured and consist of the Manufacturer Identifier, Expanded Device Type code and a unique Device Identifier. The Protocol normally uses the lower 38 bits of the Unique Identifier to address Data Link Layer requests to field devices on the link (see Fig. 11). As a result, the unique address must consist of:

- The master address and the burst mode bit as described previously.
- A 2 byte Expanded Device Type code. This code is allocated and controlled by the HCF. Further specifications regarding the use of Device Type codes can be found in the Command Summary Specification.

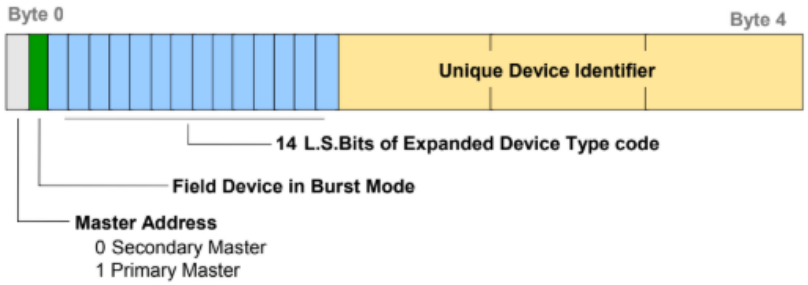

Figure 11: Long Frame Address. Source: [12].

- A 3 byte Device Identifier. This is similar to a serial number in that each device manufactured with the same Device Type Code must have a different Device Identifier.

4.10.3.3. Broadcast Address

In addition to unique addresses based on the device's Unique Identifier, the Protocol supports a Broadcast Address. A Broadcast Address is a 5 byte address with 38 bits of zeros in place of the Unique Identifier. Devices shall treat a frame with this address as though the frame was addressed directly to them. Frames with Broadcast Addresses shall only be used for services or commands that use other parameters in the broadcast message to ensure that zero or exactly one device generates a field device response message to the broadcast (e.g., see Command 11).

4.10.3.4. Polling Addresses

Polling addresses allow the Protocol to dynamically associate a short frame address with each Field Device on the link. The polling addresses may be used during a Master Device's network initialization to rapidly scan and automatically identify the field devices (see the Command Summary Specification). The Protocol provides the capability necessary to manipulate these addresses on initialization as well as during normal operations. It also provides the capability to obtain the Unique Identifier associated with a particular short frame address when the Master Device connects to the network or when necessary.

Fig. 12 shows the required format of the one byte short frame address consisting of:

- The master address and the burst mode bit as described in Section 5.3.
- The least significant 6 bits specify the polling address. Polling addresses 16-63 should not be assigned when multi-dropping with earlier Protocol revision (i.e., HART revision 3 through 5) field devices.

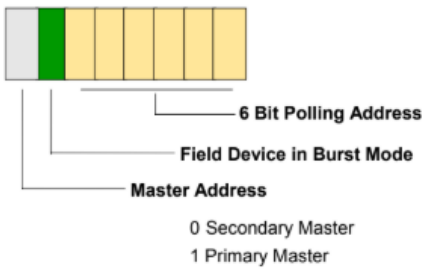

Figure 12: Short Frame Address. Source: [12].

4.10.4. Expansion Bytes

This field is 0-3 bytes long and its length is indicated in the Delimiter. The definition of the Expansion Bytes are controlled by the HCF. If a Field Device does not know the meaning of all Expansion Bytes contained in the frame then the Field Device must not answer.

4.10.5. Command

This field is one byte long. Command numbers are classified and allocated in the Command Summary Specification. Command byte values are echoed back unchanged in responses from Field Devices.

Note: Command number 254 is reserved and shall not be used in any implementation.

4.10.6. Byte Count

This field is one byte in length and represents the number of bytes of Application Layer data between the Byte Count and the Check Byte (both excluded from the count). All values between 0 and 255 (both inclusive) are legal in this field.

4.10.7. Data

The Data field is optional and consists of an integral number of bytes of Application Layer data. The Data field contains sub-fields as defined in the Command Summary Specification and contains the information transferred between the host application and the Field Device.

All BACK (Burst Frames) and ACK messages must contain at least two data bytes. If the most significant bit (i.e., bit 7) of first data byte is set then the byte contains Communication Error information (see the Command Summary Specification). Master Devices should send at least three retries (i.e., for a total of four STXs) when a Communication Error is encountered before indicating an error to the Application Layer.

No Communication Error shall be indicated in BACK frames as there is no corresponding STX about which to report a communication error. Aside from interpreting the communication error status byte (if it is present), Data Link Layer implementations shall not make any interpretation of the Data field. This means that frame recognition is disabled from the beginning of the Data field until the frame is complete as determined by a (correct) byte-count or by the physical layer signalling the end of a message (e.g., through absence of carrier detect).

4.10.8. Check Byte

This field is one byte long. The Check Byte value is determined by a bitwise exclusive OR of all bytes of a message including the leading delimiter. Along with the parity check performed in every byte received though the UART, the check byte is the second dimension to perform error detection.

The second dimension of error detection is generated by "Exclusive OR"-ing all bytes from the Delimiter up to and including the last Data byte. The result is transmitted as the last byte (i.e., the Check Byte) of a message. In other words, if all bytes in a frame are exclusive or'd together starting with the Delimiter and including the Check byte the result must be 0x00. The Vertical Parity on the Check Byte is odd as well.

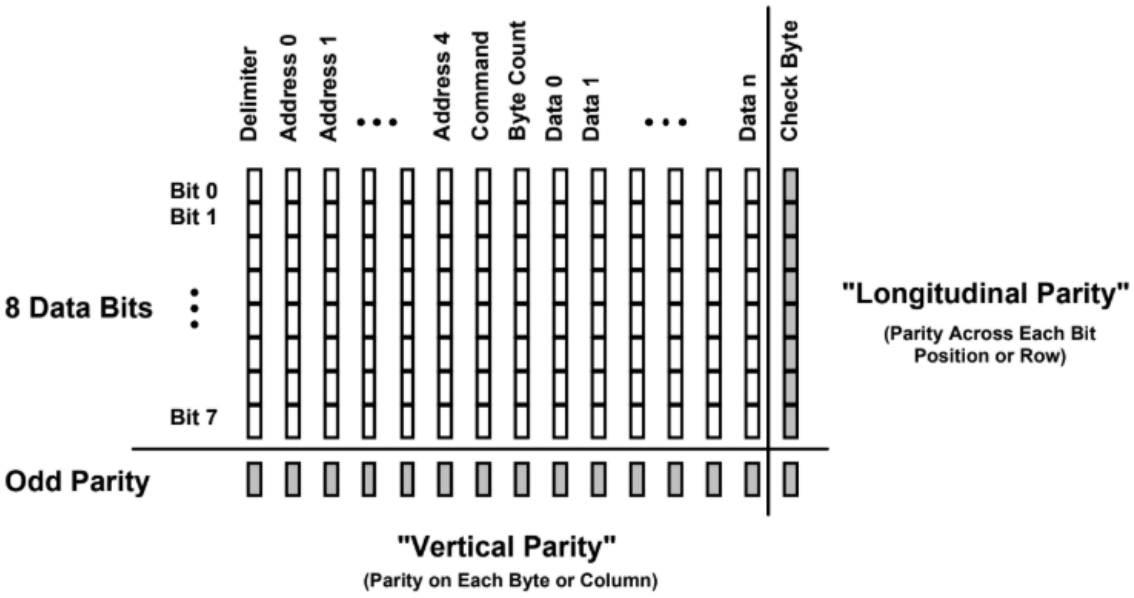

Figure 13: HART Error Detection Scheme. Source: [12].

4.11. Error Detection on different levels

On the lower levels, the UART and the longitudinal parity check reliably detect any single burst of up to three corrupted bits in the transmitted telegram (Hamming distance HD = 4), with an excellent chance of detecting longer or multiple bursts.

Errors occurring on higher levels, such as HART commands that cannot be recognized and device failures are indicated by the slave device upon each transaction using the status bytes. By polling at regular intervals, the master devices are able to know the state of all connected communication participants and to react as requested by the user or the

operating program. [1], [4].

4.12. Monitoring the HART network

Arbitration is employed to resolve access to the HART network. While Slaves access the network as quickly as attainable, Masters depend upon arbitration which is built on monitoring of network traffic and implementation of timers. Masters arbitrate by observing who sent the last transmission (a Slave or another Master) and by using timers to delay their own transmissions.

Any Master must be in condition to monitor the data transmission on a HART network, before it can start sending messages. To ensure communication between Masters and Slaves without disturbing any transaction, Masters monitor all network traffic.

In case there are two masters connected to the network, both will monitor any network activity in order to identify the moment they can send messages to the slaves. Reply messages sent by a slave will thus be read by both Masters, but only the one the response is addressed to will receive and process the response. Yet the other master will monitor the transaction to take notice when the network is free to be used.

Masters use their timers to make sure it is free to use the network, and to be able to share it to send messages. Moreover, timers are used to determine the number of times a message has been sent without the slave receiving a correct response. This is done to avoid a master sending the same message repeated times, when there is probably an error on the slave or the network that causes an incomplete transaction.

Timers are logical elements that count down from a specified time interval. When the time is counted down to 0, the timers send a notification. These elements are usually used for synchronization of signals. The timers constitute dead time when no device is communicating and therefore contribute to "overhead" in HART communication. [1], [4], [7], [8].

4.13. Monitoring the network transactions

In pursuance of determining when a communication transaction may be initiated, a Master needs to be aware of when a message starts, stops, or is present. A combination of carrier detect, UART status indications and monitoring message content is required to achieve this. Monitoring of the network transactions is done by both the Master and the Slave. In that way they are able to send and receive messages.

A carrier detect indicates whether the incoming signal has a bigger amplitude than the lowest threshold value of a HART signal. The receiver would know then that if a carrier of acceptable amplitude is present, a signal that could contain a message exists. A carrier detect that turns up in a HART device instructs the device to examine its UART output and status.

In the UART status, a receive buffer full (RBF) indication will occur once each received character. The presence of a message on the network is thus determined by the combination of carrier detect and the RBF indications. Ideally, these indications would occur at a constant rate of one every 9.17 ms and the last one would correspond to the checksum character. This is the time it takes to send a character in HART communication. However, the RBF's do not necessarily indicate the end of a message or the start of another.

The transition from one message to another can only be identified by monitoring message content. A 3-character start delimiter indicates the start of a message. The sequence of these characters has such a structure that makes it very improbable that such a sequence could show up other places in a message.

The sequence consists of 2 characters with "preamble" bytes, and a start character. A HART message concludes with a "checksum" byte. If this is observed by the HART device, then it knows the message is close to its end.

According to the HART protocol rules, gaps between characters in a message are not permitted. It is nevertheless impossible that gaps between characters in a message appear. They occur when a Slave is not capable to keep up with the 1200 bps data rate. In the time of a gap the carrier is present but no information is being sent.

Most of the HART devices have timers which have values higher than 18 ms, which is the least time it takes between 2 RBF indications assuming that a gap size on the order of 1 character time would occur. Provided that device timers are longer than 2 character times, gaps will have no effect except to slow down communication.

In case the HART indications no longer occur, a HART device can interpret this as the message is through. [7], [8].

4.14. Synchronization

A master device must be synchronized to be able to send messages. When first connected to a HART network, the device is "unsynchronized". It becomes "synchronized" when it has been monitoring bus activity and has recognized the type and end of a previous message. Loss of synchronization takes place if the device processor must briefly stop monitoring

network traffic in order to perform other tasks, even if the master device was already "synchronized".

The master (bus) will also become "unsynchronized" if there is no network traffic after a length of time called RT1, since the last performed message transaction (e.g. if there is no response to a command). Message errors would also prevent a master device from knowing what occurs, causing loss of synchronization.

In case a master is new to the network, it must first wait a time RT1 before it becomes synchronized and starts to use the network. Assuming that two masters are present in the network and both are synchronized, then they will make use of the network at intervals. If a given Master sees and recognizes a transaction of another Master with a Slave before RT1, then it is instantly synchronized. [7], [8].

4.15. Operational states

A Master device can be in three different states: monitoring, enabled and using. As mentioned earlier, during "monitoring" the master examines the communication on the network to determine the time it can make use of the network to send a message. This occurs while it waits to become synchronized.

When the master device is in the "enabled" state, it is able to send a message if intended. The master is in the "using" state when it has sent a message and waits for a response from that slave device the message was addressed to.

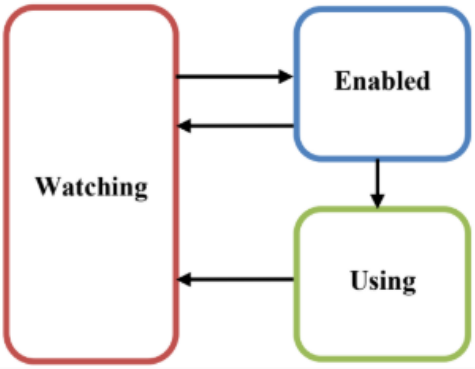

Figure 14: Operational states of a master device. Source: [8].

Master devices alternate between the named states while they are active and communicate on a HART network. The duration a master is on each state depends whether there is another master in the network and if the slave is sending messages to replies.

A Slave device in a request-response mode, in which the field device is polled to provide dynamic measured values or instrument data, would always be in "wait" state. A Slave device will only reply when it receives a command message addressed to it, otherwise the message is rejected / declined/ discarded. A reply from the slave device acknowledges that the command has been received and may contain data requested by the master. The response is sent out before a time TT0 is over.

If the slave device is switched into “burst mode” when it receives a new message, then the message will be processed and a response sent out before the device continues to broadcast “burst” responses. In case the address of the received message is incorrect, the timer BT used in burst mode is set to RT1(0), so that the master waiting for a response gets clocked out. This is done to prevent that a master interprets a “burst” response as a response to another message. [1], [7], [8].

4.16. Token passing

HART communication protocol is token passing with each message, implying the passing of token to another device. Token passing of multiple devices on a HART network can be seen as an implied token passing system where permission to transmit next is given by the communication carried in the start character and the master address bits of each message. Masters and Burst-Mode Slave take turns sharing the network. The token passing rules depend on whether a bursting slave device is present.

An important observation to make is that the slave addressed by a master may or may not be the same slave that is in burst mode.

The bursting slave gets an opportunity to send its burst message after every master-slave transaction. Two masters are supported using token passing to provide bus arbitration. Just like with normal master-slave transactions, the timing rules must be followed and if a device misses its turn, the token is considered as “lost”.

Token recovery is fast, if there is just one master present, it can determine that the other is absent and take over after only 75 ms. Likewise, if no master come up following a burst message a burst slave can broadcast its next burst message after only 75 ms. Note that loss of token is different as the bus becoming unsynchronized with a relatively long recovery time of 300 ms or more. [1], [8].

4.17. Timing rules

Due to the fact that collisions between messages or other communication errors occur when a Master and Slave devices communicate, it is decisive that every element that transmit messages in a HART loop has a fixed time for when and how fast a message or response will be sent/routed. The various masters and slaves have thus prescribed rules for response times and when the (internal) timer shall start and stop. Thus, timers are employed in the logic in order to steer/rule the transmission of messages or responses.

To prevent that a Master stays and wait indefinitely for a response to a message, the Slaves have a maximum time it must answer within. This time is called TT0 (256 ms), and it is equal for all type of slaves. In case a slave does not manage to send a response within this period, the master will consider this as an unsuccessful transaction. TT0 is deliberately made quite large to accommodate less capable hardware and software that is likely to be found in a Slave. TT0 is set to that time it takes to send 28 characters on 11 bits in HART communication with a data rate of 1200 bits per second.

As specified previously, HART Master devices monitor the communication in the loop; detecting when a reply requested by a master is acquired. This is used by a primary and secondary master to be simultaneously active on the/a same network.

Two masters (if they are present) take turns at communicating with the slave devices. After each transaction is completed, one of the masters should pause for a short time RT2 (75 ms) before sending another command to allow a chance for the other master to break in if it attempts. I.e. after a master has received a message, it will start a "timer" with value RT2; which is equal for both the primary and secondary Master. Note that if three masters are connected and active, the timing rules fail and communication may be severely disrupted.

A slave in “burst” mode will repeatedly send a data message. Between each of these messages the slave will wait a time defined as BT in order to let the master send a message to the slave between every reply message the slave sends while in “burst” mode operation. BT’s value is the same as RT2. Under these circumstances BT is the time it takes to send 8 characters with 11 bits HART communication protocol. [1], [7], [8].

The different “timer” values are often expressed as the time it takes to send a certain number of characters in HART communication. Table 2 review the most important Data Link Timers for the HART protocol.

Data Link Timer		Characters	Meaning
STO Slave Time out	256 ms	28	TT0 28 The slave must begin its response within this time.
RT1 Link Quiet (primary) – RT1 (0) RT1 Link Quiet (secondary) – RT1 (1)	305 ms 380 ms	33 41	An unsynchronized master waits for this time before transmitting. This ensures that no ongoing transaction will be interrupted. Different values for the primary and secondary masters ensure that the primary master has first access if both are connected simultaneously.
RT2 Link Grant – RT2	75 ms	8	A master waits this time after a response to itself, to allow another master to take turn if it wishes.
Burst mode time - BT	75 ms	8	The time a slave in “burst” mode waits between each response it broadcasts.
HOLD	20 ms	2	A master (or a bursting slave) must start its next command within this time after its access entitlement begins.

Table 2: HART Timers. Source: [10].

4.18. Delayed response mechanism (DRM)

The DRM mechanism, introduced in HART 6, allows a slave device to indicate that it is unable to respond fully within the allowed time. The use of this added option to the HART protocol is however limited to bridge devices such as multiplexers or I/O systems; where there is often a speed difference and thus a delay in getting information from the field device.

Specific response codes report the initiation, continued existence or failure of the delay activity, so that a master can know how to proceed. Thus a slave implementing the DRM must provide a buffer to retain information regarding this operation. Bridging devices should rather maintain more than two buffers, so they can handle requests to each of its HART I/O channels.

Even while a DR is in progress for a write command, slave devices must always respond to read commands. In case a device runs out of buffers, it simply uses the normal “busy” status in response to the master. [1]

4.19. Performance data of HART transmission

The bit data rate and the number of bits per telegram define the time required to transmit a telegram. The length of the telegram varies depending on the message length and the message format.

HART protocol uses FSK with a data rate of 1200 bps. Not all HART commands or responses contain data. In earlier HART revisions, 25 data bytes or less were used by all Universal and Common Practice commands. However HART revision 6 specifies commands with up to 33 data bytes.

Regarding HART as an asynchronous half-duplex protocol with 11 bits per character, 9.167 ms would be the required time to transmit a single character. The following example gives a brief into the transmission time of a HART telegram.

Example

Consider a message using a short frame format and containing 25 characters. The following data applies to a HART transaction:

	Data
Time per bit	1 bits / 1200 s 0.83 ms
Bytes per telegram	25 message characters + 10 control characters
Telegram size	35 characters 11 bits 385 bits
Transaction time	35 characters 9.167 ms 0.32 s
User data rate	25 message characters (8 bits / 385 bits) 52 %
Time per user data byte	0.32 s / 25 bytes 13 ms

Table 3: HART transaction time. Source: [10].

An average of 500 ms is accounted for per HART transaction, i.e. to read information on a single variable from a HART device, including additional maintenance and synchronization times. Consequently 2-3 HART transactions are expected to be carried out per second, showing that HART communication is not suitable for transmitting time-critical data. [1], [4].

4.20. Application Layer

Layer 7 - the Application Layer- of the OSI protocol reference model, provides the user with network capable applications. The communication procedures of HART master devices and operating programs are based on HART commands defined in this layer. The HART command Set provides uniform and consistent communication for all field devices.

The public commands of the protocol are classified according to their function into commands for master devices and for field devices, defining four major groups: Universal commands, Common-Practice commands, Device-Specific commands, and Device-Family commands.

These predefined commands enable HART master devices (HMD) to give instructions to a field device or send messages. In that way, actual values and parameters can be transmitted as well as various services for start-up and diagnostics performed. An immediate response is sent by the field devices in terms of an acknowledgement telegram which contains the requested status reports and/or the data of the field device.

Reply messages will always include two Status bytes, reporting any outgoing communication errors, the status of the received command and the operational state of the slave device. The function of a HART command can be categorized as:

- Read: the field device responds with requested data; does not change the field device in any way.
- Write: sends a new value to be stored in the field device, e.g. measurement ranges and tag parameter.
- Command: instructs the field device to perform some action, which may involve writing to memory.

Data types used in these commands include integers, floats, alphanumeric, enumerated, bit and date formats. The command byte contains an integer in the range of 0 to 253 (0xFD), representing one of the HART commands. A command byte of 31 (0x1F) indicates the presence of an extended (device family) command. [1], [2].

4.20.1. Universal Commands

These commands provide functions which must be implemented in all field devices. They provide access to information useful in normal operations such as read PV and units. Universal Commands are in the range of 0 to 30. [1], [2], [3], [4].

4.20.2. Common-Practice Commands

"Common-Practice" commands are in the range of 32 to 121. They provide functions implemented by many, but not necessarily all, HART communication devices. Functions such as read measured variables, set parameters are among others. [1], [2], [3], [4].

Commands in the range 122 to 126 are defined as "non-public". They are commonly used by manufacturers to enter device-specific information during assembly, e.g. the device identification number, which will never be altered by users or for direct memory read and write commands. [1],

4.20.3. Device-specific Commands

These commands provide functions which are more or less unique to a particular field device. “Device-specific” commands are in the range 128 to 253. In HART revision 4 and earlier, device-specific commands always included the Device Type Code as the first byte of the data field to ensure that a command never reached an incompatible device. From HART revision 5 and on the use of Unique Identifiers guarantees that the host has fully identified the field device before any other command can be sent. [1], [2], [4].

4.20.4. Device family commands

Introduced in HART 6, it provides a set of standardized functions for instruments with particular measurement types (with specific transducer types), allowing full generic access without using device-specific commands. Furthermore, Device family commands offer improved interoperability between devices from different manufacturers, without using individual Device Descriptions (DD).

Device family commands are "extended commands" of 16 bits. These commands are in the range 1024 to 33791. The command set for each family, such as the "temperature device family" provides a consistent definition of data and configuration procedures.

The temperature device family describes any temperature measurement made using traditional copper, nickel or platinum resistance sensors (RTDs) or thermocouples.

Although the family specification includes special features, many are optional and may not be supported in a particular device. [1]

4.21. Data

A further part of layer 7 corresponds to the different kinds of information that HART field

devices contain, as well as the relationships between this data and beyond the devices. These are described as follows [1]:

- HART variables: include all stored HART-accessible data items which are capable of being changed, either by HART command or by changing process conditions. Some of these data items are accessible via specific HART commands.

- Device variables: are the set of floating point numerical data items chosen by the device manufacturer to represent the process measurements. These are readable in groups of up to four by HART commands #9 and #33.

- Dynamic variables: are the setup of up to four data items representing measurements: the primary, secondary, tertiary and quaternary variables, abbreviated as PV, SV, TV and QV. The PV is only read by HART command #1, while command #3 reads all four and the current output or input in mA.

- Configuration parameters: are a considerable quantity of configuration data to set up the field device for a particular application. Many are floating point variables, such as the end-of-range values for which the analog output is to be 4 mA and 20 mA. Others will be integers representing, e.g., sensor type or engineering units to be used. Some parameters are written and read by common practice HART commands; others may use device-specific commands.

- Device information: are the few data items which serve to identify the HART field device, such as manufacturer, device type and device revision levels. Others like alphanumeric tag or date are available for the user to set in relation to the particular application. The identification information forms parts of the response to HART command #0, while the user application-related information is accessible through universal commands #12, #13, #17, #18, #20 and #22.

- Data types: the HART protocol allows the following representations of data: Integer, Floating point, Alphanumeric, Enumerated, Bit and Date.

4.22. Establishing Communication with a HART Device

When first connecting to a field device, a HART master must access to the address of the field device in order to communicate successfully with it. To achieve this, a master can learn the address of a slave device by issuing Command #0 or Command #11. These and several other HART commands (e.g. #21, #73, #74) are available for initial communication, causing the slave device to respond with its address. [1], [2], [8].

- Command #0: “Read Unique Identifier” is the most widely used method in order to launch communication with a slave device. It enables a master to learn the address of each slave device without user interaction.
  Since HART revision 5, Command #0 is the only command which is recognized in the older short frame format, using the polling address 0 to 15 (0 to 63 in HART 6) as the device address. Therefore it can be used whether or not a field device has its tag set.
  When individual polling addresses have been configured before installation, command #0 can also be used to scan all possible polling addresses in multidrop applications. If the unique ID of a device is already known, command #0 can also be used in the long frame format to obtain further device information [1], [2]..

- Command #11: “Read Unique Identifier associated with tag” requires that the device has already been configured with a short tag. Applicable in the event there are more than 15 devices in the network or if the network devices were not configured with unique polling addresses.
  Command #11 puts the short tag in the command’s data field to specify which field device should respond. This is a long frame format command, thus the address field is set to the broadcast address of 38 zeros. Only the device with a matching tag will respond. [1], [2].

For bridge or I/O systems, Commands #74: “read I/O system capabilities” returns the structure (up to 3 levels) and (maximum) number of sub-devices attached. Command #75: “Poll sub-device” can then be used to poll individual sub-devices to find their unique IDs. Since it may be that not all possible devices are connected, response code 9 is used to indicate that no device has been found at a specified address. [1].

All the above commands (except #74) return the same identity data. Table 4 summarizes some of this information.

Byte	Content
1	Manufacturer identification code
2	Device type code
4	Universal command revision (same has HART major rev. nr. 7)
6	Software revision
7	Hardware rev. (5 bits) and physical signalling code (3 bits)

9-11	Device ID number
13*	Maximum number of device variables

Table 4 Device Information. Source: [10].

From the information returned in bytes 1, 2 and 9-11, the device's 38- bit Unique ID can be constructed. Once the Unique ID is known, the host will continue communication with the field device using the standard long frame address format. This applies only for devices of HART rev. 5 or above.

Knowing which HART revision is implemented in this field device, the host can anticipate which universal commands it should understand. [1].

5. Project development

This section will cover the entire practical works of this project, including all main functionalities of the developed software as well as initial setups for testing.

5.1. SOFTWARE HIERARCHY:

To settle the overall process concept being executed behind this project, the following graphic depicts the main classes and participating elements, and their interactions.

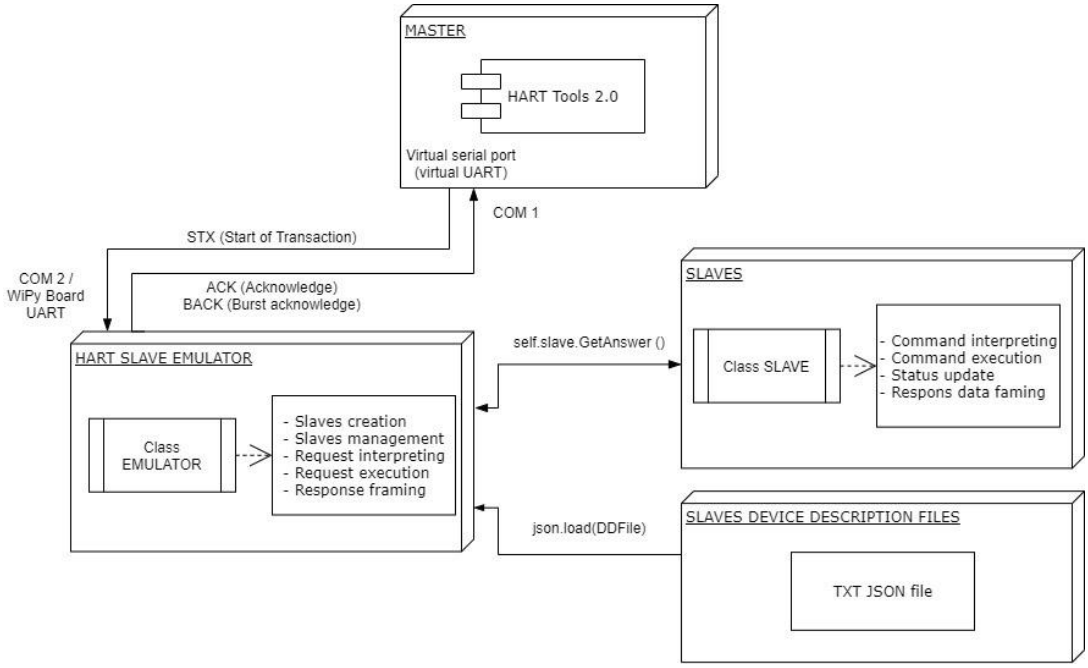

Figure 15: Slave emulator conceptual diagram. Source: own.

The diagram boxes in Fig. 15 represent the 4 main elements interacting, the Master which is a commercial software and need no development; the slave emulator, contains the class Emulator and is in charge of acknowledging the master requests and sending a response; the slaves, defined in the class Slave, that contain every supported command and update the slave status; and finally, the slaves Device Description Files, that contain all the information associated to a slave parsed in Json.

In order to run the developed programs, Python 3 has been installed and the terminal emulator Putty, that will be used during the project to compile the written code and to run the programs in its shell.

5.2. SETUP

Before writing the code for the core elements, the communication between them must be tested and all elements must be ready to function.

The first setup concerns the Pyboard. Before going into any testing the board must have its firmware updated. This process is extensively detailed in the Final Degree Project by Gerard Altés of ETSEIB entitled “Pyboard based HART-Wifi gateway for monitoring industrial sensors in a mobile app” [16]. The installed firmware revision used during this project is pybv10-dp-thread-20200306-v1.12-213-g8db5d2d1f. After that, a physical connection between the RX and TX of one of the available UARTs has been established in order to test the sending and receiving of information through the Board (pin Y9 to pin Y10).

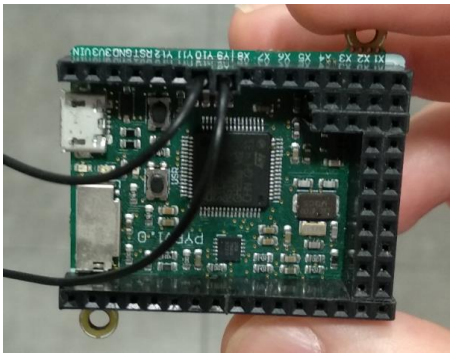

Figure 16: Pyboard with pins Y9 and Y10 connected. Source: own.

A first quick test of the Pyboard UART is done by sending some strings of bits through the TX and then reading the RX input. The results show that TX and RX are working properly since the transmitted message is equal to the received one. Fig. 17 shows the testing done using shell.

```
MicroPython v1.12-213-g8db5d2d1f on 2020-03-06; PYBv1.0 with STM32F405RG
Type "help()" for more information.
>>> from pyb import UART
>>> uart = UART(3, 9600)
>>> uart.init(9600, bits=8, parity=None, stop=1)
>>> uart.write('abc')
3
>>> uart.read(3)
b'abc'
>>> uart.write('abcde')
5
>>> uart.read(3)
b'abc'
>>> uart.read(3)
b'de'
>>>
```

Figure 17: TX and RX communication test. Source: own.

Once the Pyboard is ready, communication within the PC has been tested employing a small piece of software. A Putty shell runs a writer that can send a few given messages through COM 1 serial port. This port is then virtually connected using “Virtual Serial Port Tools” a COM 2 serial port, which will have a reader program also running on Putty ready to acknowledge the messages sent through COM 1. Fig. 18 shows the satisfactory results of this simple test and the entirety of the written code for this test can be found in the Annex.

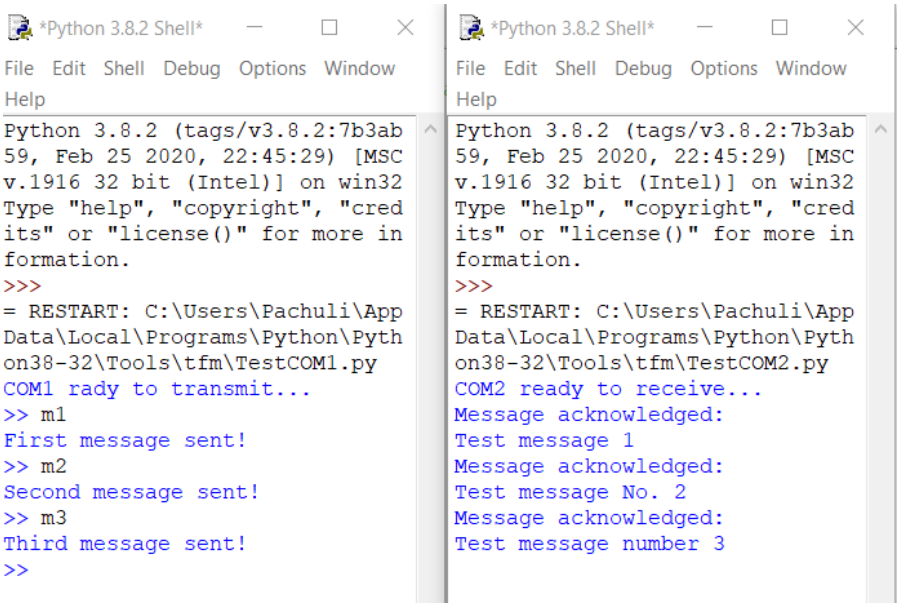

Figure 18: Read input through from the COM 1 and COM 2 Idle shell. Source: own.

5.3. HART SLAVE EMULATOR

This program contains the class Emulator that uses an infinite loop to keep reading any incoming bytes on the open COM serial port. It has a few differentiated parts each one covering a certain functionality of the program. In this section all parts and functionalities of this software will be detailed.

5.3.1. Imported modules

There are 8 modules imported, 2 of them are local and are only used to reference a certain code to its value, for example “HartUnits”, which contains a dictionary with a numerical code as key and the physical magnitude they represent as value; similarly to what “HartStatusErrors” does but with error values and their codes. Both modules coding is included in the Annex. Below there is a list of the remaining imported modules and how they are used:

- “time”: module that provides a wide range of time related functions. In this project this has been used to add time separators using function time.sleep() when swapping between TX or RX mode.

- “serial”: a module for encapsulating the access to serial ports. This module has been used in order to open the COM serial ports and set their parameters. Several functions of this module have been used, but mostly the serial.Serial() in order to define the port parameters, and the .setRTS(True or False) which if set true the port is set on TX mode and if False is on RX mode.
  The image below shows the parameters for the serial port for the Hart Slave Simulator.

```
self.ser = serial.Serial(
port = 'COM2',
baudrate = 1200,
parity = serial.PARITY_ODD,
stopbits = serial.STOPBITS_ONE,
bytesize = serial.EIGHTBITS
)
self.ser.setRTS(False) #Reception
```

Figure 19: Serial definition. Source: own.

- “taskrepetitivethreads”: this module contains the methods required for threading. Threading allows an activity to be run in a separately thread of control, thus allowing for multiple operations to be carried simultaneously. This module has been used both in the emulator and the slaves program. In the Emulator class case, threading is used to constantly run the method newdata() that will acquire the received data in case there is any.

- “HartSlave”: this module contains the class Slave. The method getanswer() is called from the emulator each time a full message from the master has been acknowledged in order for the specified slave to return the required data for the given command.

- “json”: a module designed for parsing data in json format. It is used during the creation of the slaves to read the DDFiles containing the slave information. It is very simple to use and only requires the method load() to translate an entire json file into the corresponding python object.

- “globe”: a module for pathname recognition. It is only accessed when loading the DDFiles, in order to specify the correct path. Only the method glob.(path) is used.

5.3.2. Emulator class _init_ constructor

Inside the _init_ constructor of the Emulator class, the first processes carried are the definition of the global variables, the initialization of the COM serial port, the creation of the slaves and the launch of the data acquisition thread.

Upon start, all the global variables are defined. Most of the variables are used during the processing of the master frame to store information that will later be given to the slave in order to perform the actions required by each command. Others are counters used throughout the process to keep tracking, these counter will have to be reset at the end of the process in order to no interfere with the following one. Finally, there is the self.slavesdef and self.slaves, which are lists used to store the information of every slave.

```
class Emulator:
    def __init__(self):
        self.state = 0                  'Process state counter'
        self.delimiter = 0              'Delimiter byte'
        self.preamble_counter = 0       'Number of preamble bytes acknowledged'
        self.address_counter = 0        'Number of address bytes processed'
        self.data_counter = 0           'Number of data bytes processed'
        self.dlc = 0                    'Data counter byte'
        self.command = 0                'Command byte'
        self.checksum = 0               'Checksum byte'
        self.address = [0,0,0,0,0]      'Address bytes'
        self.polladdress = 0            'Polling address byte'
        self.deviceID = [0,0,0]         'Device ID bytes'
        self.expdevtype = [0,0]         'Expanded Device Type bytes'
        self.burst = 0                  'Burst flag bit'
        self.master = 0                 'Master address bit'
        self.data = []                  'Data bytes'
        self.status0 = 0                'First status byte'
        self.status1 = 0                'Second status byte'
        self.slavesdef = []             'Array with original slave definition'
        self.slaves=[]                  'Array with live slave objects'
```

Figure 20: list of global variables and their use. Source: own.

After the global variables the serial port is started using the serial module as shown in the previous section. Once the COM port is open, the slaves are created by looping through all the files in the given path directory. For each file the json.load() method is called and the resulting dictionary is appended in the self.slavesdef list. This is then looped again and with every dictionary in the list the class slave is called, and a slave object is then added to the self.slaves list. There is an alternative way of creating the slaves in case there is not a single DDFile for every slave but a file with the definitions for all of then, in which case simply the commented lines are the ones to be executed.

```
#Create Hart Slaves:
#Option 1: from a single description file.
#with open('HartDD.txt', 'r') as file:
#    self.slavesdef = json.load(file)

#Option 2: from multiple description files.
path = 'C:/Users/Pachuli/Desktop/TFM/HartEmulator/*.txt'
files=glob.glob(path)
for file in files:
    with open(file , 'r') as DDfile:
        self.slavesdef.append(json.load(DDfile))
for i in self.slavesdef:
    self.slaves.append(HartSlave.Slave(i))
```

Figure 21: Slaves creation lines. Source: own.

To finalize the constructor, a thread is launched in order to begin the data acquisition in loop mode.

5.3.3. Class methods

This class contains three methods, newdata(), process() and sendframe(), each tasked with a certain functionality of the program.

5.3.3.1. newdata(self)

This method is called recursively using the thread. Its main task is to detect if there are any bytes of data received through the serial port in RX mode, and if so, read the first received byte and launch the process() method with that byte information. The received bytes are transformed into integers using ord() function.

```
#New data aquisition method:
def newdata(self):
    if self.ser.inWaiting() > 0:
        data = ord(self.ser.read(1))
        self.process(data)
```

Figure 22: newdata() method. Source: own.

5.3.3.2. process(self)

In the process() method the received bytes from the master are identified and stored in the global variables. The function is separated into several states, each according to the STX framing position the read byte is located, and the global variable self.state is used to

navigate them. For example, once the whole preamble has been read the state counter will increase and move the process to the next phase for the following byte to process. The following diagram describes the circulation of all states:

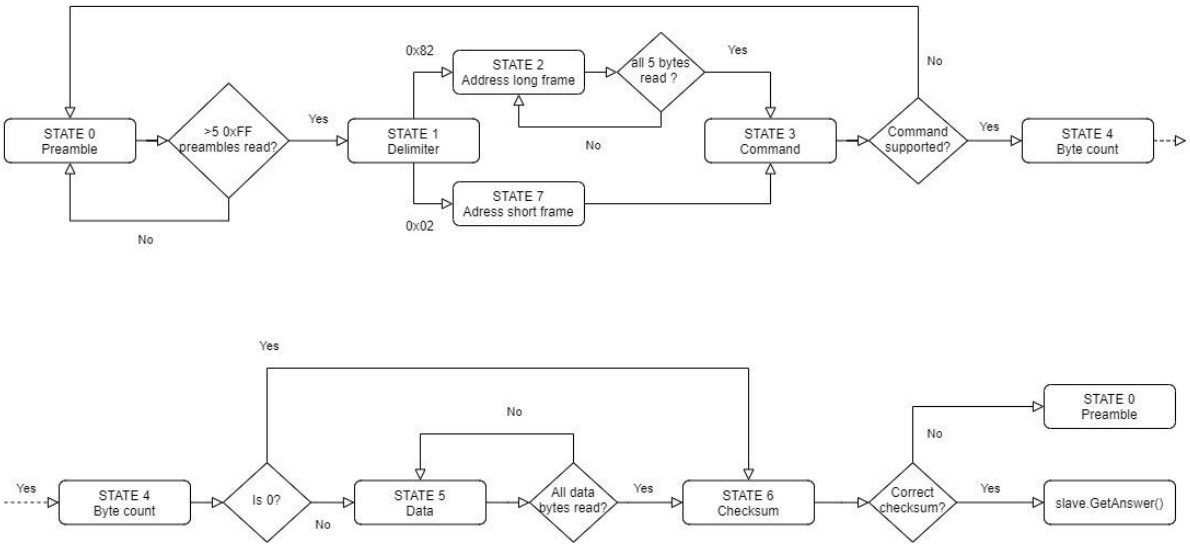

Figure 23: process() method flow diagram. Source: own.

As can be seen in the diagram, each state corresponds to a frame part of the master request. This architecture has proven to be robust, since in case the process is broken at some point, the program is able to discard the remaining bytes of that request and continue with the following one without causing an interruption error. For example, if a requested command is not supported, at state 3 this command will not be recognised and the self.state counter will be set to 0; then when new data is acquired, this new data will not correspond to 0xFF preamble bytes and thus the process() method will remain in state 0 until 5 or more 0xFF are acknowledged.

Also, since more than five 0xFF can be sent in a request (up to 20, though it is unusual) state 1 recognises 0xFF as a new preamble byte and does not advance to the following until it does not detect the expected value for the delimiter. For this revision of the program the expected values of the delimiter are 0x82 and 0x02, only supporting requests from the primary master in either short or long frame, but with no expansion bytes. This decision can be reverted in future works and allow the software to fully understand the delimiter byte, but for now this fully covers the requirements of the project.

Alternatively, according to HART revision 7, the process could be skipped from state 7 to state 6 directly, since the only command that can be called through short address frame is command 0, and does not include any data bytes in the request. However, since this is not the case for earlier HART revisions, to keep full compatibility, after processing the short

frame address the program will then move to determine which the requested command is.

Once the process is completed and the checksum is correct, the software will call the self.sendframe() method with slave.GetAnswer() method as parameter, that at it's turn uses the command and the received data as parameters. The class Slave will be revised later, but to summarize it for now, this will return the data bytes (status included) to be sent in the ACK frame. The full code written in Python is included in the Annex.

5.3.3.3. sendframe(self, data)

The method sendframe() is used to send the required bytes for the ACKnowledge framing. The data parameter must be a list containing the data bytes to be sent, this depends on the called command, but it will always contain the 2 status bytes at least.

At first the serial port is set on TX mode. Then, the preambles are sent, followed by the received delimiter XORed with a 4, since by definition the last byte must be a 6 in ACK frames. For BACK frames this second byte should be set to 1. After this, the address, command and data are sent, and finally the ACK checksum is determined and sent to close the frame. Once all bytes are sent, the serial port is set on RX mode again to keep monitoring the network for more STX frames.

5.4. SLAVE CLASS

This program contains the class Slave. This class is used to store all the supported commands for the implementation and to update the device status. In other words, it manages all the data bytes in the ACK frame to be sent back to the master.

5.4.1. Imported modules

There are 7 modules imported, 3 of which are local and are only used to reference a certain code to its value, exactly the same case as in the slave emulator, with the only difference that this class includes the "HartDeviceVariableCodes", that contains the codes for each variable supported by a slave device. All local modules coding is included in the Annex. Below there is a list of the remaining imported modules and how they are used:

- "taskrepetitivethreads": same module employed in the emulator class. In this case it is used to keep a thread that updates the primary value of the slave. The thread recursively calls the PVupdate() method, that updates the value for the primary variable following a certain function.

- "time": again the same module employed in the class emulator. In this class it is

used when defining the update function of the primary value. To make it a time dependent function, the time.process_time() gives a time variable that starts counting once the thread process is launched.

- “math”: this module provides access to the mathematical functions defined by the C standard. It is only used in the Primary value update function in order to give it a sinusoidal shape, with the math.sin() call and the value math.pi.

- “struct”: this module performs conversions between Python values and C structs represented as Python strings. It is used when converting a float number that the primary value or other variables may have to transmit into a string of bytes. In order to do this the method struct.pack(format, value) is called. The conversion must be done using the standard IEEE 754 binary32, that transforms a floating point value into 32 bits (single precision), and this is done by introducing the value ‘f’ in the format parameter. Once the packing has been done, the bytes must be reversed before being sent.

5.4.2. Slave class _init_ constructor

The _init_ constructor of the Slave class consists only of the global variables definition and the launch of the primary value update thread.

In this class only three global variables are defined. The self.slave, which contains the dictionary with all the slave information, this information is given as parameter when a slave object is created. The self.PrimaryVar, which is used to avoid modifying the original value of the Primary variable, and finally, the self.status variable, which stores status information to be returned in the answer.

```
class Slave:
    def __init__(self, data):
        self.slave = data       'Data contained in the STX frame.'
        self.PrimaryVar = 0     'Auxiliary variable to store new PV values.'
        self.Status = [0,0]     'Device status to be sent in the ACK frame.'
```

Figure 24: list of global variables and their use. Source: own.

After the variables are created, a thread is started to call recursively the method PVupdate(), as described before.

5.4.3. Class methods

5.4.3.1. getanswer(self, command, data)

This method contains all the supported commands and returns the data bytes to be sent in the ACK frame. As parameters it must be given the command as an integer type and the data included in the STX frame in a list type, data is optional since not all commands require any. Below all the supported commands, their specifications and their implementation:

5.4.3.1.1 Command 0: Read unique identifier

Returns identity information about the field device including: the Device Type, revision levels, and Device ID. This command is implemented by a field device in both Short and Long Frame Formats. Command 0 is the only command that may respond to a short frame address in HART revision 7, but for this implementation other commands can respond as well. This is a universal command so all field devices should recognize it.

Byte	Format	Description
0	Unsigned-8	"254"
1-2	Enum	Expanded Device Type (see Common Table 1, Device Type Codes and the *Command Summary Specification*, Section 6).
3	Unsigned-8	Minimum number of Preambles required for the request message from the Master to the Slave. This number includes the two preambles used in asynchronous Physical Layers (along with the Delimiter) to detect the start of message.
4	Unsigned-8	HART Protocol Major Revision Number implemented by this device. For HART Revision 7, this value must be the number 7.
5	Unsigned-8	Device Revision Level (refer to the *Command Summary Specification*)
6	Unsigned-8	Software Revision Level of this device. Levels 254 and 255 are reserved.
7	Unsigned-5	(Most Significant 5 Bits) Hardware Revision Level of the electronics in this particular device. Does Not Necessarily Trace Individual Component Changes. Level 31 is Reserved.
7	Enum	(Least Significant 3 Bits) Physical Signaling Code (see Common Table 10, Physical Signaling Codes)
8	Bits	Flags (see Common Table 11, Flag Assignments)
9-11	Unsigned-24	Device ID. This number must be different for every device manufactured with a given Device Type.
12	Unsigned-8	Minimum number of preambles to be sent with the response message from the slave to the master.
13	Unsigned-8	Maximum Number of Device Variables. This indicates the last Device Variable code that a host application should expect to be found in the field device (e.g., when identifying the Device Variables using Command 54).
14-15	Unsigned-16	Configuration Change Counter
16	Bits	Extended Field Device Status (refer to Common Table 17, Extended Field Device Status)
17-18	Enum	Manufacturer Identification Code (see Common Table 8, Manufacturer Identification Codes)
19-20	Enum	Private Label Distributor Code (see Common Table 8, Manufacturer Identification Codes)
21	Enum	Device Profile (see Common Table 57)

Figure 25: command 0 response data bytes. Source: [11].

For the implementation of the command, some bytes have been given random values. These bytes are of no interest towards the needs of the Department of Electronic Engineering, and thus, for simplification some parameters are not in the slaves definition. This will be easily modifiable though, since it is very simple to add new information in the slave definition and then include it in the command response. Command code below:

```
if command == 0 or command == 11: #Get Long Address
    self.Status = self.getstatus()
    return [self.Status[0], self.Status[1], 0xFE, self.slave['ExpDevType'][0],
self.slave['ExpDevType'][1], 0x05, 0x07, 0x00, 0x00, 0x00, 0x01,
self.slave['DeviceID'][0], self.slave['DeviceID'][1],
self.slave['DeviceID'][2], 0x05, 0x01, 0x00, 0x00, self.slave['ExpDevStatus'],
self.slave['ManufID'][0], self.slave['ManufID'][1], 0x00, 0x00, 0x01]
```

Figure 26: command 0 code implementation. Source: own.

5.4.3.1.2 Command 1: Read primary variable.

Read the Primary Variable. The Primary Variable value is returned along with its Units Code. This is also a universal command.

Byte	Format	Description
0	Enum	Primary Variable Units (refer to *Common Tables Specification*)
1-4	Float	Primary Variable

Figure 27: command 1 response data bytes. Source: [11].

For the implementation of the command, the function struck.pack() is used to pack the floating point value into a four bytes string using standard IEEE 754 binary32.

```
elif command == 1: #Get Primary Variable
    ans = self.getstatus()
    ans.append(self.slave['Units'])
    b = struct.pack('f', self.PrimaryVar)
    for i in reversed(b): ans.append(i)
    return ans
```

Figure 28: command 1 code implementation. Source: own.

5.4.3.1.3 Command 2: Read Loop Current and Percent Of Range

Reads the Loop Current and its associated Percent of Range. Percent of Range always follows the Primary Variable value, even if Loop Current is in an alarm condition or set to a value. The Upper and Lower Range Values maps the Primary Variable value to the Percent

of Range. Percent of Range is not limited to values between 0% and 100%, but tracks the Primary Variable to the Transducer Limits when they are defined.

Byte	Format	Description
0-3	Float	Primary Variable Loop Current (units of milli-amperes)
4-7	Float	Primary Variable Percent of Range (units of percent)

Figure 29: command 2 response data bytes. Source: [11].

The implementation for this command is also randomized. First, a value is given to the primary value loop current (between 4-20 mA) and then, the percentage of this value according to the maximum loop current, which is 20 mA.

```
elif command == 2: #Read Loop Current And Percent Of Range
    ans = self.getstatus()
    b=struct.pack('f', self.slave['PrimaryVariableLoopCurrent'])
    for i in reversed(b): ans.append(i)
    b=struct.pack('f', (self.slave['PrimaryVariableLoopCurrent']-4)/16)
    for i in reversed(b): ans.append(i)
    return ans
```

Figure 30: command 2 code implementation. Source: own.

5.4.3.1.4 Command 3: Read Dynamic Variables and Loop Current

Reads the Loop Current for the primary variable and up to 4 predefined Dynamic Variables. The number of response data bytes depends on the amount of Dynamic variables supported by the field device. If only the primary variable is supported, the command will return 9 bytes of data, 4 referring to the loop current, 1 for the units code of the variable and 4 for the value of the variable. For every additional variable supported the device will respond with 5 additional bytes, up to a maximum of 24. Like the previous commands, this is a Universal command.

Byte	Format	Description
0-3	Float	Primary Variable Loop Current (units of milli-amperes)
4	Enum	Primary Variable Units Code (refer to *Common Tables Specification*)
5-8	Float	Primary Variable
9	Enum	Secondary Variable Units Code (refer to *Common Tables Specification*)
10-13	Float	Secondary Variable
14	Enum	Tertiary Variable Units Code (refer to *Common Tables Specification*)
15-18	Float	Tertiary Variable
19	Enum	Quaternary Variable Units Code (refer to *Common Tables Specification*)
20-23	Float	Quaternary Variable

Figure 31: command 3 response data bytes. Source: [11].

This command implementation only returns data for the primary variable. This could be modified by adding information in the slave definition about the number of dynamic variables available and these dynamic variables value and units, but for the current requirements for the project this was deemed unnecessary.

```
elif command == 3: #Read Dynamic Variables And Percent Of Range
(Demo for 1 variable)
    ans = self.getstatus()
    b=struct.pack('f', self.slave['PrimaryVariableLoopCurrent'])
    for i in reversed(b): ans.append(i)
    ans.append(self.slave['Units'])
    b=struct.pack('f', self.PrimaryVar)
    for i in reversed(b): ans.append(i)
    return ans
```

Figure 32: command 3 code implementation. Source: own.

5.4.3.1.5 Command 6: Write Polling Address

This is a Data Link Layer Management Command. Writes the polling address and the loop current mode to the field device. The polling address is used for automatic master identification of field devices. The loop current mode determines whether current signalling is being used by the field device. These commands includes data in the request, for the new polling address and loop current mode must be supplied. Both the response data bytes and request data bytes should be the same if the command is properly executed. Also, It is a Universal command.

Byte	Format	Description
0	Unsigned-8	Polling Address of Device (refer to the *Data Link Layer Specification*)
1	Enum	Loop Current Mode (refer to Common Table 16, Loop Current Modes)

Figure 33: command 6 request and response data bytes. Source: [11].

For the implementation of this command the loop current mode has been omitted and passed a 0x00 byte, but the polling address is properly modified in the slave definition and then returned.

```
elif command == 6: #write polling address
    ans = self.getstatus()
    self.slave['PollAddress'] = data[0]
    ans.append(self.slave['PollAddress'])
    ans.append(0x00)
    return ans
```

Figure 34: command 6 code implementation. Source: own.

5.4.3.1.6 Command 11: Read Unique Identifier Associated With Tag

This command may be issued using either the Device long frame address or the Broadcast Address. No response is made unless the Tag matches that of the device even when the device long frame address is used. This command returns identity information about the field device including: the Device Type, revision levels, and Device ID. The address in the Response Message is the same as the request. In order to check the device Tag, this must be supplied in the request data bytes. Also, it is a Universal command.

Byte	Format	Description
0-5	Packed	Tag

Figure 35: command 11 request data bytes. Source: [11].

The response data bytes and implementation are the same as in command 0.

5.4.3.1.7 Command 14: Read Primary Variable Transducer Information

Reads the Transducer Serial Number, Limits/Minimum Span Units Code, Upper Transducer limit, Lower Transducer Limit, and Minimum Span for the Primary Variable transducer. Another Universal command.

Byte	Format	Description
0-2	Unsigned-24	Transducer Serial Number
3	Enum	Transducer Limits and Minimum Span Units Code (refer to *Common Tables Specification*)
4-7	Float	Upper Transducer Limit
8-11	Float	Lower Transducer Limit
12-15	Float	Minimum Span

Figure 36: command 14 response data bytes. Source: [11].

The implementation of this command has been simplified by passing a constant floating point number for the last 3 requested values. The transducer serial number is also not supported, so instead of that value the device ID is returned and the units code for the Minimum span units code.

```
elif command == 14: #Get Primary Variable
    ans = self.getstatus()
    for i in self.slave['DeviceID']:
        ans.append(i)
    ans.append(self.slave['Units'])
    b = struct.pack('f', 0.1)
    for i in reversed(b): ans.append(i)
    b = struct.pack('f', 50.2)
```

```
    for i in reversed(b): ans.append(i)
    b = struct.pack('f', 10.0)
    for i in reversed(b): ans.append(i)
    return ans
```

Figure 37: command 14 code implementation. Source: own.

5.4.3.1.8 Command 33: Read Device Variables

This command allows a Master to request the value of up to four Device Variables. In other words, a Master may request only 1, 2, 3 or 4 Device Variables. When a Device Variable requested is not supported in the Field Device, then the corresponding Value must be set to "0x7F, 0xA0, 0x00, 0x00", and the Units Code must be set to "250", Not Used. This is a common practice command.

Byte	Format	Description
0	Unsigned-8	Slot 0: Device Variable Code (see Device Variable Codes Table in appropriate device-specific document)
1	Unsigned-8	Slot 1: Device Variable Code (see Device Variable Codes Table in appropriate device-specific document)
2	Unsigned-8	Slot 2: Device Variable Code (see Device Variable Codes Table in appropriate device-specific document)
3	Unsigned-8	Slot 3: Device Variable Code (see Device Variable Codes Table in appropriate device-specific document)

Figure 38: command 33 request data bytes. Source: [14].

Byte	Format	Description
0	Unsigned-8	Slot 0: Device Variable Code (see Device Variable Code Table in appropriate device-specific document)
1	Enum	Slot 0: Units Code (refer to *Common Tables Specification*)
2 - 5	Float	Slot 0: Device Variable Value
6	Unsigned-8	Slot 1: Device Variable Code (see Device Variable Code Table in appropriate device-specific document)
7	Enum	Slot 1: Units Code (refer to *Common Tables Specification*)
8 - 11	Float	Slot 1: Device Variable Value
12	Unsigned-8	Slot 2: Device Variable Code (see Device Variable Code Table in appropriate device-specific document)
13	Enum	Slot 2: Units Code (refer to *Common Tables Specification*)
14 - 17	Float	Slot 2: Device Variable Value
18	Unsigned-8	Slot 3: Device Variable Code (see Device Variable Code Table in appropriate device-specific document)
19	Enum	Slot 3: Units Code (refer to *Common Tables Specification*)
20 - 23	Float	Slot 3: Device Variable Value

Figure 39: command 33 response data bytes. Source: [14].

For now, this command has been implemented with limited functionalities. Since some of the variables that could be requested are not available in the slave definition, the battery charge left for example, the response data bytes passed will always correspond to the same variables disregarding those requested. But nonetheless, the returned number of variables will be equal to the total number of variables requested, with the only difference being that the first variable will always be the primary variable and the rest will be returned as specified in the command specification for not used variables case. This guarantees that if a master was to request command 33, the response given would not cause an error, but the received information could be unrelated to the requested and even misleading.

```
elif command == 33: #Read device variables
    ans = self.getstatus()
    num = len(data)
    ans.append(HDVC.PrimaryVariable)
    ans.append(self.slave['Units'])
    b = struct.pack('f', self.PrimaryVar)
    value = [0x7F, 0xA0, 0x00, 0x00]
    for i in reversed(b): ans.append(i)
    if num > 1:
        ans.append(HDVC.SecondaryVariable)
        ans.append(0xFA)
        for i in reversed(value): ans.append(i)
        if num > 2:
            ans.append(HDVC.TertiaryVariable)
            ans.append(0xFA)
            for i in reversed(value): ans.append(i)
            if num > 3:
                ans.append(HDVC.QuaternaryVariable)
                ans.append(0xFA)
                for i in reversed(value): ans.append(i)
                return ans
            else:
                return ans
        else:
            return ans
    else:
        return ans
```

Figure 40: command 33 code implementation. Source: own.

This can be redesigned by adding the variable code in the DDFiles and going through all the request data bytes identifying the variable requested and returning its value.

5.4.3.1.9 Command 48: Read Additional Device Status

Returns device status information not included in the Response Code or Device Status Byte. This Command also returns the results of Command 41, Perform Self-Test. Response

Bytes 0-5 and 14-24 may contain Device-Specific Status information. Extended Device Status and Device Operating Status contains commonly used status information. See the appropriate Common Tables for more information. This is a common practice command.

Byte	Format	Description
0 - 5	Bits or Enum Only	Device-Specific Status (refer to appropriate device-specific document for detailed information)
6	Bits	Extended Device Status (refer to Common Table 17, Extended Device Status Information)
7	Bits	Device Operating Mode (refer to Common Table 14, Operating Mode Codes)
8 - 10	Bits	Analog Channel Saturated (MSB-LSB)
11 - 13	Bits	Analog Channel Fixed (MSB-LSB)
14 - 24	Bits or Enum Only	Device-Specific Status (refer to appropriate device-specific document for detailed information)

Figure 41: command 48 response data bytes. Source: [14].

The only parameter included in the DDFiles is the Extended device status, the rest of values required are passed as 0x00 except Device operating mode, that is passed as 0x01.

```
elif command == 48: #Read Additional Device Status
    ans = self.getstatus()
    for i in range (6):
        ans.append(0x00)
    ans.append(self.slave['ExpDevStatus'])
    ans.append (0x01)
    for i in range (3):
        ans.append(0x00)
    for i in range (3):
        ans.append(0x00)
    for i in range (6):
        ans.append(0x00)
    return ans
```

Figure 42: command 48 code implementation. Source: own.

5.4.3.1.10 Command 140: Read Detailed Status

This command is used to read detailed status information (i.e., status information that provides detail beyond that of the response status byte and Common Command 48 (Read Additional Device Status See Table 7-2). This is a device specific command supported only by Liquid Ultrasonic Flow Meter of DANIEL MEASUREMENT AND CONTROL, INC. To ensure that this command is only called for the appropriate type of device, when command 140 is required the device id is checked and compared to the Liquid Ultrasonic Flow Meter. If they coincide the command is then executed, or the error command not supported is

passed otherwise.

The implementation for this command is irrelevant. The focus of this was to include a device specific command. For this reason, the full 17 bytes of data requested are all passed as 0x00. [15]

```
elif command == 140: #Device specific command:
status information
    ans = self.getstatus()
    for i in range (17):
        ans.append(0x00)
    return ans
```

Figure 43: command 140 code implementation. Source: own.

5.4.3.2. PVupdate(self)

This method is called recursively using a thread. It's main task is to update the primary value following a time dependent function. For this implementation, a sinusoidal unit has been used, with a variable amplitude of the defined primary value +- 10% and a period of 10 s, which is the same as 0,1 Hz.

```
def PVupdate(self):
    self.PrimaryVar=self.slave['PrimaryVariable']*
(1+0.1*math.sin(math.pi*time.process_time()/5))
```

Figure 44: PVupdate() method. Source: own.

5.4.3.3. getstatus(self)

This method is called during the implementation of every command. Therefore the two first bytes of data are always the status of the field device. If the definition does not include any error, the first byte is always 0 and the second byte is the 'ResponseCode', which will generally be a 0 as well. If there is an error and thus the 'CommsStatus' parameter is not 0, this parameter will make the first status byte and not the 'ResponseCode' but the 'FieldDevStatus' parameter will be the second status transmitted byte, and give further information on the error found.

```
def getstatus(self):
    Status = [0,0]
    if self.slave['CommsStatus'] == 0:
        Status[0] = self.slave['ResponseCode']
        return Status
    else:
```

```
        Status[0] = self.slave['CommsStatus']
        Status[1] = self.slave['FieldDevStatus']
        return Status
```

Figure 45: getstatus() method. Source: own.

5.5. DEVICE DESCRIPTION FILES

DDFiles contain the necessary information to create a slave object and execute the requested commands. They are plain text files written in Json format.

Json format uses different data types, numbers, strings, booleans, arrays and objects. Arrays are equivalent to a python list, since they are addressable and iterable and are represented by the same symbol. A Json object is in turn equivalent to a python dictionary, using keys to be accessed and every key having an associated value, that can be any data type including arrays and objects.

The preferred way of working for the Department of Electronic Engineering is to have every slave definition in a separate file. This way, a supplier can be asked to give a Json file including all the slave information required, and this slave will easily be recognized by the Slave emulator. To implement this, every DDFile will contain an object with the required slave parameters as keys and the information of the slave as values for these parameters. For example, one necessary parameter to define a slave its the device ID, this parameter will have a key named ‘DeviceID’ with an array of 3 values, each corresponding to the three bytes of the device ID. Fig. 46 shows an example of DDFile.

```
{
 "PrimaryVariable": 25,
 "Units": 32,
 "CommsStatus": 0,
 "ResponseCode": 0,
 "FiedDevStatus": 0,
 "ExpDevStatus": 0,
 "PollAddress": 3,
 "ExpDevType": [
  3,
  4
 ],
 "DeviceID": [
  1,
  1,
  1
 ],
 "ManufID": [
  0,
  3
 ],
 "Tag": [
  1,
  2,
  3,
  4,
  5,
  6
 ],
 "PrimaryVariableLoopCurrent": 11.5
}
```


Figure 46: Json DDFile example. Source: own.

This implementation offers a lot of flexibility in terms of slave definition. The Json txt files can easily be modified and created. Adding new parameters in future implementations will not even require declaring them on the slave class. Simply adding them in the DDFile will make the new parameters accessible inside the slave class.

Additionally, in order to optimize the process of slave creation for testing, a small piece of software to write Json files has been developed. It contains the parameters to be written and their values in a dictionary, that when the program is executed, is dumped into a json format txt file. This also gives the ability to specify visual characteristics like the level of indent and separations to make the DDFiles more user friendly. If not specified, the resulting txt file will have all the information in a single line, thus difficulting its understanding and modification. The Json writer program code is added in the Annex.

6. Testing and validation

After the programs have been developed it is the turn for testing and validation. The testing process begins by launching the programs and depurating syntax and other errors, basically debugging. This process will not be explained in detail because it would have very little interest for the reader, but it is worth mentioning since it can take a considerable amount of time and effort.

Once the programs are ready to be executed, the next step is to plan how tests are going to be run and the expected results. For this project, three phases of testing are prepared. The first will focus on virtual communication validation and checking that all commands are returning the expected data bytes. Next phase will focus on the interaction with commercial software, and determine if the developed code can respond well in a not tailored environment. And finally, the last phase will have the software loaded into a Pyboard to evaluate the Slave Simulator in a physical environment.

6.1. First test phase: virtual communication and commands

The objective in this phase is to prove that the developed software can send data properly and that each command provides the expected response. Since not all commands developed can be easily called through the commercial Master, a simple master demo program has been improvised. This master only sends the necessary frames to try the commands in different slaves, and then reads the exact number of data bytes that would be expected for that command response. If more bytes have been sent from the Slave emulator, then there will still be information in the Serial port RX buffer; and if less bytes than the expected are sent, then the master will not return any readings. The improvised master software is included in the Annex.

In order to lighten this section, only the testing for two commands will be included. The rest have been tested using the same process and all of them have successfully responded to the request command. In this section, command 33 and command 140 results will be presented. These commands are not tested in the commercial master implementation but both include functionalities that could be useful for future implementations. Command 33 has a response frame length dependent on the requested number of variables and command 140 should only be acknowledged if the Device Id of the requested address matches the type of device that supports the command.

6.1.1. Command 33

Three request tests will be launched in sequence to validate this command implementation. These requests will address two different slaves, and on slave will be requested a response to the command twice with a different number of variables requested each time. According to the implementation made for command 33, both devices should provide it's primary value and units code for the first requested variable, and return the specified value for not used variables for the rest.

The master implementation for this command is very simple. The command is called through the code 'com33a#', where # is the number of the slave to be addressed. For the testing environment 2 slaves are created, one with a primary value mean of 25 degrees Celsius and the second with a primary value of Bars. Once the command is called, it will require the amount of variables to be requested. With this information the STX framing is determined and then sent. After the slave emulator has acknowledged the command and sent back the response, the code 'read33' will read all data bytes received and present it in a user comprehensible way, rather than the raw byte string. Coding for the master can be found in the Annex.

The following figures show the results obtained on the Idle shell.

```
Enter your commands below.
Insert "exit" to leave the application.
>> com33a1
How many Device Variables should be consulted(0-4):
>> 4
>> read33
[65, 191, 252, 42]
[0, 0, 160, 127]
[0, 0, 160, 127]
[0, 0, 160, 127]
Address:
b'\x83\x04\x01\x01\x01'
Status:
b'\x00\x00'
First Variable:
Primary Variable
Units:
Celcius
Value:
(23.998126983642578,)
Second Variable:
Secondary variable
Units:
Not Used
Value:
(5.757515000371376e-41,)
Third Variable:
Tertiary Variable
Units:
Not Used
Value:
(5.757515000371376e-41,)
Forth Variable:
Quaternary Variable
Units:
Not Used
Value:
(5.757515000371376e-41,)
```

Figure 47: Command 33 first test Master terminal information. Source: own.

In this first test slave 1 has been addressed and 4 variables have been requested. Both the address and status correspond to the definition of Slave 1. The first variable requested is the primary variable, as implemented, and the response shows that units are Celsius and the primary value is in range of the mean value plus the introduced deviation. The rest of the variables requested are not present in the slave definition, and so the emulator returns the requested code plus the value and units specified in the command specifications. To double check this, the first lines of the response before the Address bytes are given correspond to the decimal number for the reversed bytes received as values for the variables, the first is equivalent to the primary value after the IEEE 754 transformation, and the rest are the specified bytes to be sent as expected.

```
>> com33a1
How many Device Variables should be consulted(0-4):
>> 1
>> read33
[65, 180, 192, 59]
Address:
b'\x83\x04\x01\x01\x01'
Status:
b'\x00\x00'
First Variable:
Primary Variable
Units:
Celcius
Value:
(22.593862533569336,)
```

Figure 48: Command 33 second test Master terminal information. Source: own.

The second test also addresses slave 1, but only requests one variable. Again the results are within the expected, and as can be seen the primary value bytes have changed due to the value being time dependent. On a small note, since the mean value for the primary variable is 25, and the amplitude of the oscillation is a 10% of this value, results between 27.5 and 22.5 are correct.

```
>> com33a2
How many Device Variables should be consulted(0-4):
>> 3
>> read33
[65, 23, 163, 217]
[0, 0, 160, 127]
[0, 0, 160, 127]
Address:
b'\xa0\xed\x02\x02\x02'
Status:
b'\x00\x00'
First Variable:
Primary Variable
Units:
Bars
Value:
(9.47750186920166,)
Second Variable:
Secondary variable
Units:
Not Used
Value:
(5.757515000371376e-41,)
Third Variable:
Tertiary Variable
Units:
Not Used
Value:
(5.757515000371376e-41,)
```

Figure 49: Command 33 third test Master terminal information. Source: own.

The third test addresses slave number 2 and requests 3 variables. As for slave 1, the address and status are correct, and the received values for all variables are the expected.

In addition to the displayed information, it can be assumed that the emulator is sending the right number of bytes. Since the tests have been launched in series without interruption, any additional or lacking bytes in the response would have caused the master shell to crash if not in the same request, certainly in the directly following.

6.1.2. Command 140

Two request tests will be launched in sequence to validate this command implementation. These requests will address two different slaves, slave 1 has a random Device Id and thus should not be able to respond to this device specific command. Slave 3 on the other hand is the right device type and should then acknowledge the command.

The master implementation for this command only shows raw bytes received because the content is irrelevant; the focus of this test is to determine if device specific commands are being handled appropriately. The command is called through the code ‘com140a#’, where # is the number of the slave to be addressed. As stated before, slave 3 is the one that supports this command. Once the response has been sent, the code ‘read140’ will print the raw bytes received. Coding for the master can be found in the Annex.

The following figures show the results obtained on the putty shell.

```
Enter your commands below.
Insert "exit" to leave the application.
>> com140a3
>> read140
[b'\x86', b'\x8d', b'(', b'\x03', b'\x03', b'\x03', b'
\x8c', b'\x13', b'\x00', b'\x00', b'\x00', b'\x00', b'
\x00', b'\x00', b'\x00', b'\x00', b'\x00', b'\x00', b'
\x00', b'\x00', b'\x00', b'\x00', b'\x00', b'\x00', b'
\x00', b'\x00', b'\x00', b'\xbf']
```

Figure 50: Command 140 first test Master terminal information. Source: own.

The first test addresses slave 3, which as can be seen has correctly acknowledged the command and delivered the expected response. Preamble values are not being printed but they are indeed being transferred. All the data had been set to 0 for this command ACK frame as can be seen; and the delimiter, command and bytecount are all correct. The address is not correctly displayed due to the fact that the print() function gives the ASCII representation of bytes, and for the second address byte its value has the “(” symbol representation. Keeping this display issue aside, the address bytes are the corresponding to slave 3.

```
>> com140a1
>> read140
```

Figure 51: Command 140 second test Master terminal information. Source: own.

```
Starting Slave Hart Simulator...
send a new response to master
Command not supported by device.
```

Figure 52: Command 140 second test Slave simulator terminal information. Source: own.

For the second test, a snap of both the Master and Slave simulator is shown. In the master terminal the process crashed (not responsive) due to the fact that there are no bytes in waiting in the virtual UART RX. In the simulator terminal, the message command not supported is returned because the addressed slave does not meet the necessary condition to acknowledge command 140, that being its Device ID. Although the simulator did not respond, it has not crashed and it is ready to receive another command, even after rejecting the last request.

These two tests prove both that slaves are not responding to device specific commands not specific to its Device ID, and that the correct information is sent when this command is acknowledged.

The conclusion to the First test phase is very satisfactory, all commands (not only the presented in the project) are working accordingly to what is expected. This allows for the

second test phase to start, where the simulator will be interacting with commercial master software.

6.2. Second test phase: Commercial Master HART Tools 2.0

The second phase of testing is oriented towards checking the compatibility of the developed emulator with software that could be used in a real industrial situation. While Hart Tools 2.0 may not be a professional tool used in industry, it does follow the same protocol and has some functions that give a great testing environment. The functionalities of this program that will be employed include a slave scan given a certain polling address, a full network scan (from polling address 1 - 15) and a raw command console, which sends the specified strings of bytes. Other services available are calibration commands, diagnosis tests, event notifications and more, but these fall out of the scope requiring much more complex slave simulations.

With this information, this testing phase has been split into 3 tests. The first will be recognizing and reading slave information through the polling address. The second will scan the entire network of slaves and send commands iterating between the found slaves in the network. And finally, the raw command input will be used to change the slave polling address, and determine if after writing information, Hart Tools 2.0 still recognizes it and in the right polling address.

6.2.1. First test: Device detection

For this first test 3 slave devices are simulated and running. Polling addresses range from 1 to 3, with the first two devices being set as pressure sensors with Bar as their unit and the third as a thermometer with ºC as unit. Once the Hart Tools 2.0 is open, the first thing to do is open the desired port, in this case COM 1. After that, In the first tab “Configure Device” theres is a polling address number that can be written, all polling addresses will be introduced as well as not supported polling addresses to see how the software responds. After the desired polling address is set, the button “Get Long Address” will launch a command 0 with short address frame to the selected polling address. Once the slave has acknowledged that command, the button “Read Device” will now be eligible, this will launch a set of commands in sequence (in order: 0, 12, 13, 14, 15, 16, 20, 7 and 1). Only commands 0, 14 and 1 are implemented, so for the rest of them the simulator is not supposed to respond.

After checking, all simulated devices and non populated polling addresses the following figures correspond to a first poll to an empty address directly followed by a request to polling address 3, with a simulated thermometer.

```
Starting Slave Hart Simulator...
No polling address match.          ——— Poll to address 4
send a new response to master      ——— Poll to address 3
send a new response to master      ——— Command 0
Command not supported by device.
Command not supported by device.
send a new response to master      ——— Command 14
Command not supported by device.
Command not supported by device.
Command not supported by device.
Command not supported by device.
send a new response to master      ——— Command 1
```

Figure 53: Device detection Slave simulator terminal information. Source: own.

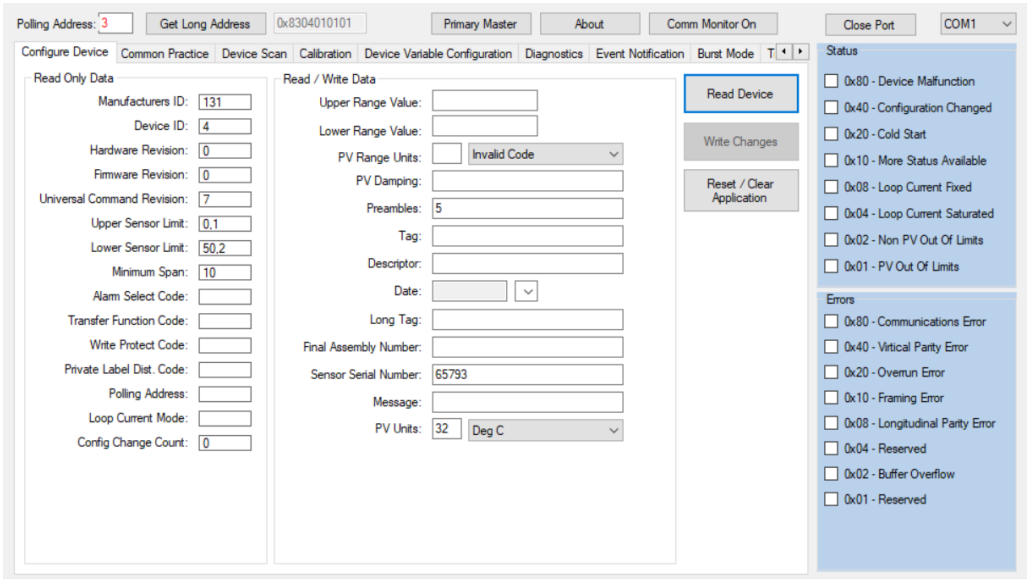

Figure 54: Device detection HART Tools 2.0 interface. Source: own.

As can be seen from the figures, the simulator environment has passed a polling address mismatch error, when at first the Polling address 4 was selected. As expected the simulator did not response and raised an error message, but avoided crashing and continued waiting for a STX frame. When polling address 3 is selected, the simulator environment now detects a slave with the same polling address and acknowledges the get long address command (Third line of Figure 53). After that, the Read Device function is executed, and the slave acknowledges only the implemented commands, and passes the “Command not supported by device.” error for the rest, see Fig. 53 for each corresponding command acknowledged. After this both the master and the slave simulator are ready for more requests.

This is a very satisfactory result, because it proves the viability of the developed software to work with real commercial applications. All the information displayed in Fig. 54, corresponds to the information given in the slave definition. In further implementations, all requested

commands could be implemented so that a Full read can be performed.

6.2.2. Second test: Network scan

The next test will use HART Tools 2.0 Network scan service. At first, on the “Device Scan” tab the button “Poll Network” will poll all the available polling addresses from 0 to 15 to recognise all available devices. In this step, if a device has polling address 0, the scan will stop and only acknowledge that device, for this reason polling address 0 is left unoccupied. After the full network has been scanned, the button “Start scan” will be available, this will launch a recurrent set of commands being requested iteratively among all detected devices.

This commercial software implementation has at least two issues that have so far been detected. The first seems to be a bug, where the “Poll network” function only detects devices if previously a single device has been acknowledged using the first test procedure, since after doing this HART Tools 2.0 seems to work fine. The second issue is due to the “Device scan” way of working. The cycling of commands is done by sending command 1 to the first recognised polling address, then command 2 to the second recognised device and finally command 3 to the next acknowledged device; at this point, since this service only uses the 3 mentioned commands, if there is a number of devices on the network multiple of 3, HART Tools will always request the same command to a certain slave which will cause that two third of the information is never requested. To solve this, 4 slaves will be on network, this way all slaves will be requested all commands after 4 cycles of requests.

With no further due, the following figures show the results of this test.

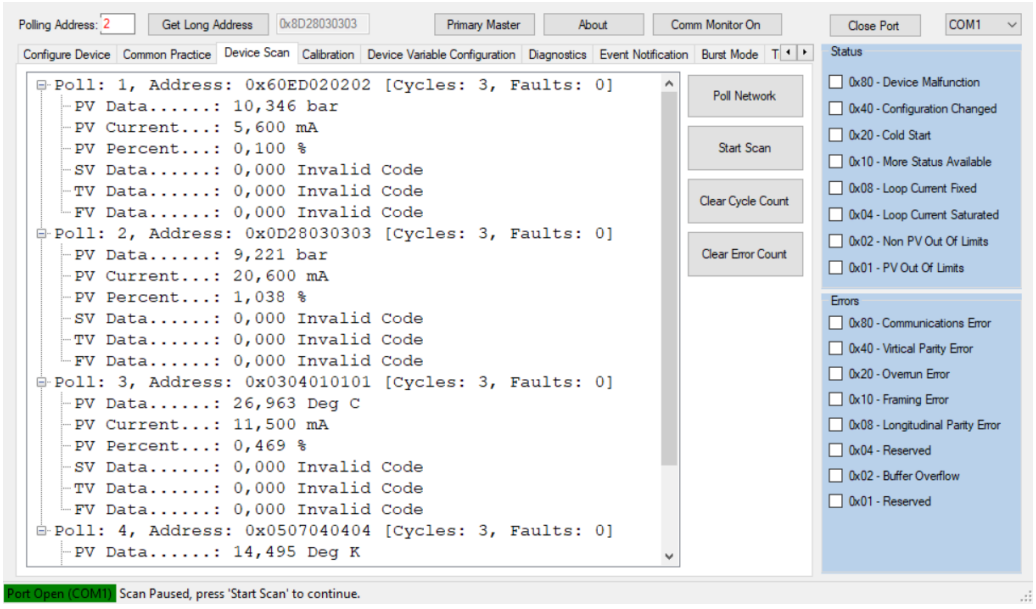

Figure 55: Network scan HART Tools 2.0 interface. Source: own.

Before the analysis of the obtained results, it is important to mention that two devices are set to Bar units, with a mean primary value of 10 and 9. Another slave uses ºC and is set at an average of 25. Finally, the last device is using Kelvin units and has a mean value of 15.

As can be seen in HART Tools 2.0 interface all 4 devices have been recognised and all the information included in commands 1, 2 and 3 is correctly received. All long addresses match those of the simulated slaves, as well as the primary value and the primary value units. This proves that the simulated environment can sustain a multidrop network of devices with all of them acknowledging all the requested commands simultaneously. Although it can not be perceived from the snap, the primary value is changing following the expected function, with an oscillation of a 10% around it's mean value.

6.2.3. Third test: Polling address write

One interesting functionality to test is the ability to write information on the simulated slaves. This has been tested using the improvised master, but will be confirmed using HART Tools 2.0. The most straightforward way to do this is to use the first test procedure to detect one device, then on the tab "Raw Command" write the bytes to change this device polling address to a new value, and finally determine if HART Tools does recognise the same slave in the new polling address. The byte string to be sent is the following [0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x82, 0x83, 0x04, 0x01, 0x01, 0x01, 0x06, 0x01, newpoll, check], where newpoll refers to the new polling address number to be given and check refers to the calculated checksum. This STX frame will address the slave initially located in polling address 3 and will try to move it to polling address 6.

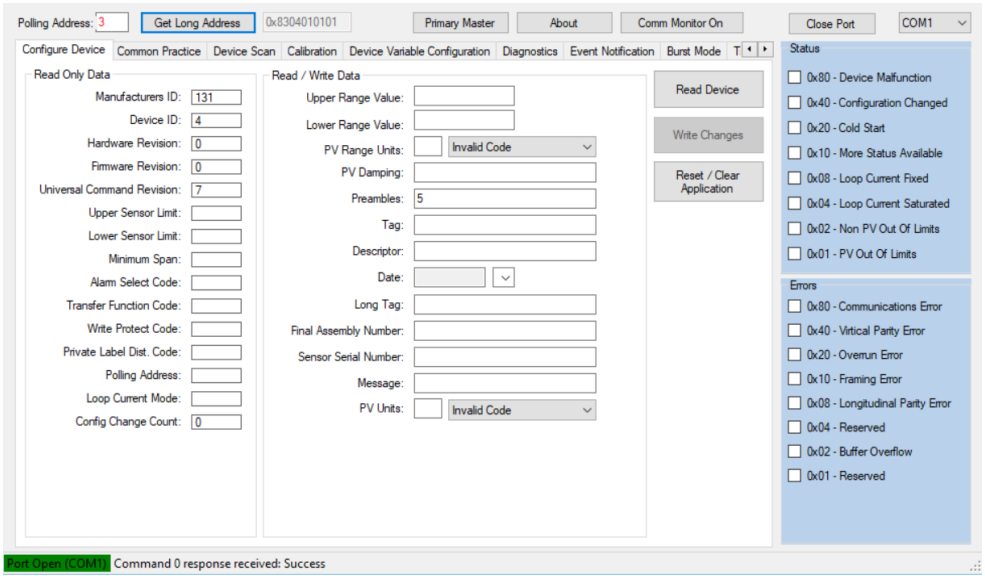

Figure 56: Polling address 3 device acknowledged, HART Tools 2.0 interface. Source: own.

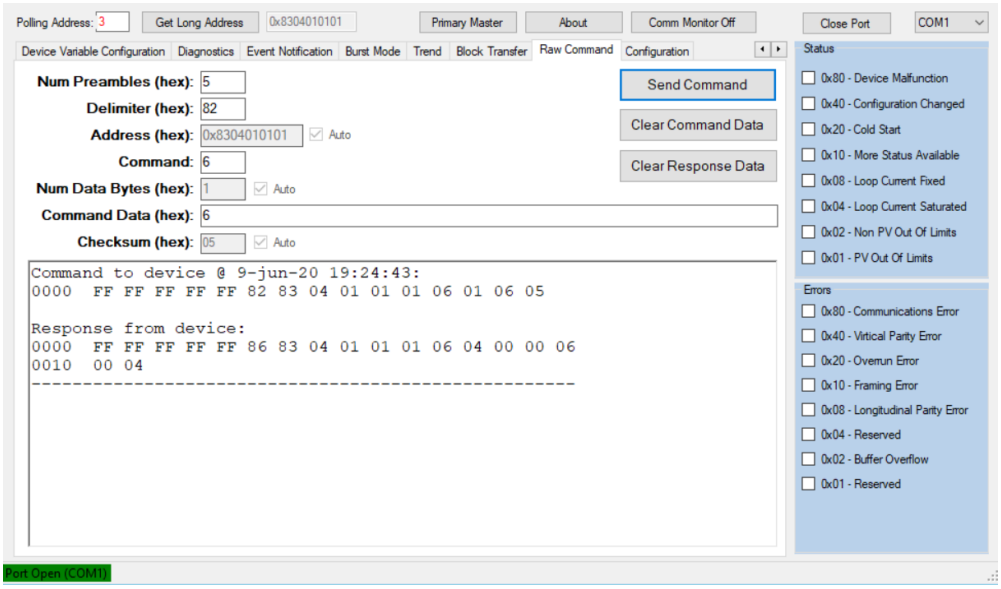

Figure 57: Raw command STX frame and ACK response, HART Tools 2.0 interface. Source: own.

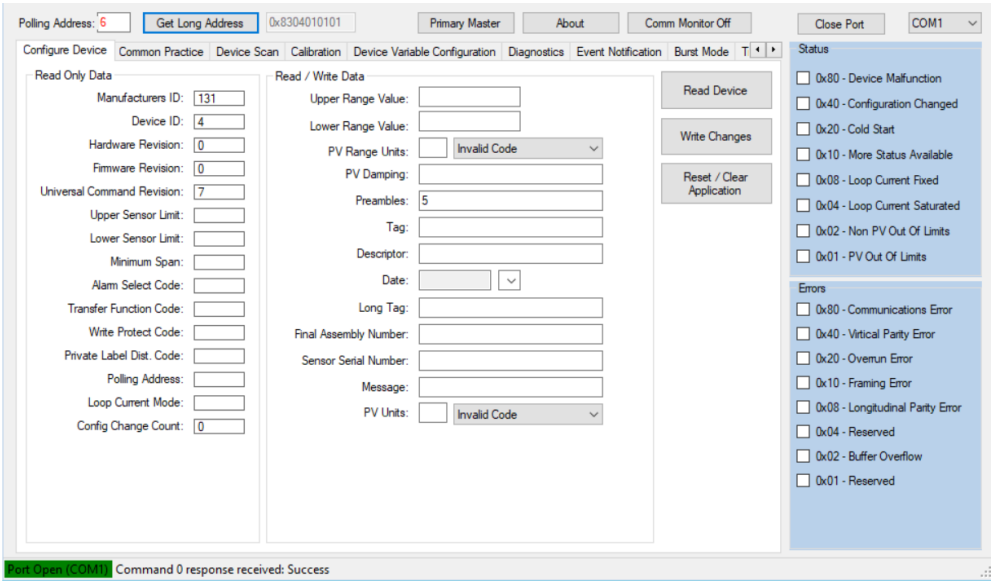

Figure 58: Polling address 6 device acknowledged, HART Tools 2.0 interface. Source: own.

The figures portrait the process followed in order. First the device is correctly detected on the polling address given in the DDFile. Figure 57 shows the bytes sent to the device and the response obtained from the device. And finally Fig. 58, shows that the device is now detected when polling address 6, and not 3.

This outcome shows that the simulated slaves are fully accessible and can store any written data for as long as the network is functioning. If the simulation is killed, newly written

information will be lost because it is not overwritten in the DDFiles. This could be reverted in future implementations but the Department of Electronic Engineering will have to determine which way is preferable.

6.3. Third test phase: Software on Pyboard and HART Tools 2.0

For practical reasons the software has been developed using Python 3.8. It would be useful though for the Department of Electronic Engineering to be able to install this software on a portable Pyboard. In order to achieve this, the Emulator code must be translated into MicroPython, a non-global python version without access to many modules.

During this translation, some modules have been replaced by others that MicroPyhton can support and some functionalities have been reworked in order to make the emulator responsive. To summarize the changes quickly: HartSlaveEmulator.py is now the main.py file; modules threading, glob and serial are not available, threading is now not available and this function have been reworked into an infinite loop, glob module have been replaced by pyb that has some similar methods, and the Pyboard does not use the serial communications port anymore but the USB virtual communications port, which is created through the pyb module also. All the modified code for this implementation can be found in the Annex.

After the translation is done the software is loaded into the Pyboard and the test phase begins. After pairing HART Tools to the USB COM port that contains Pyboard, the first device scan tests are done.

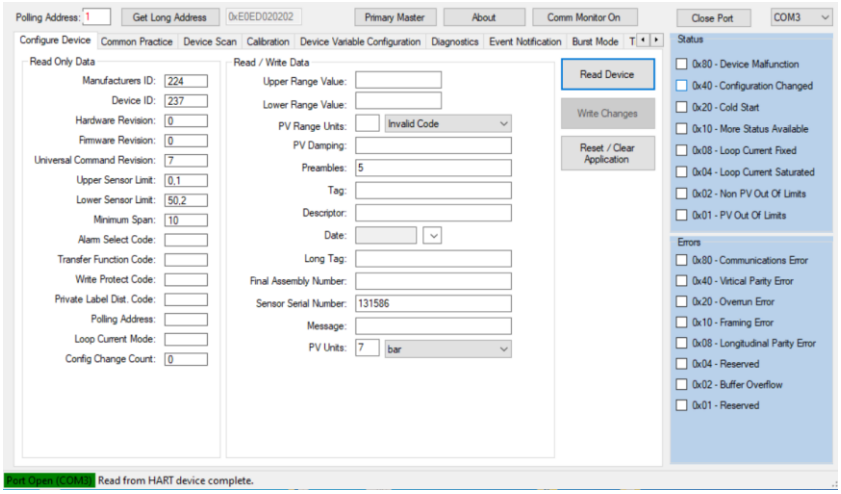

Figure 59: Device detection HART Tools 2.0 paired with Pyboard. Source: own.

The device scan is successfully achieved, and show the same results obtained in the virtual ports test. This shows that the simulation environment is responsive and executing.

The second test performed is the network scan. There will be no image attached to this test because this has not been successfully completed, for some reason HART Tool crashes during the scan. It is important to mention that during the scan, HART Tool addresses as the secondary master without any given reason, but this should not be a problem for the slave. To determine if this crashes are due to the simulated field device or HART Tool software, this test has been reoriented, and after determining the exact frame that the network scan sends, this very exact frames are being passed to the Pyboard using the Raw Command Input.

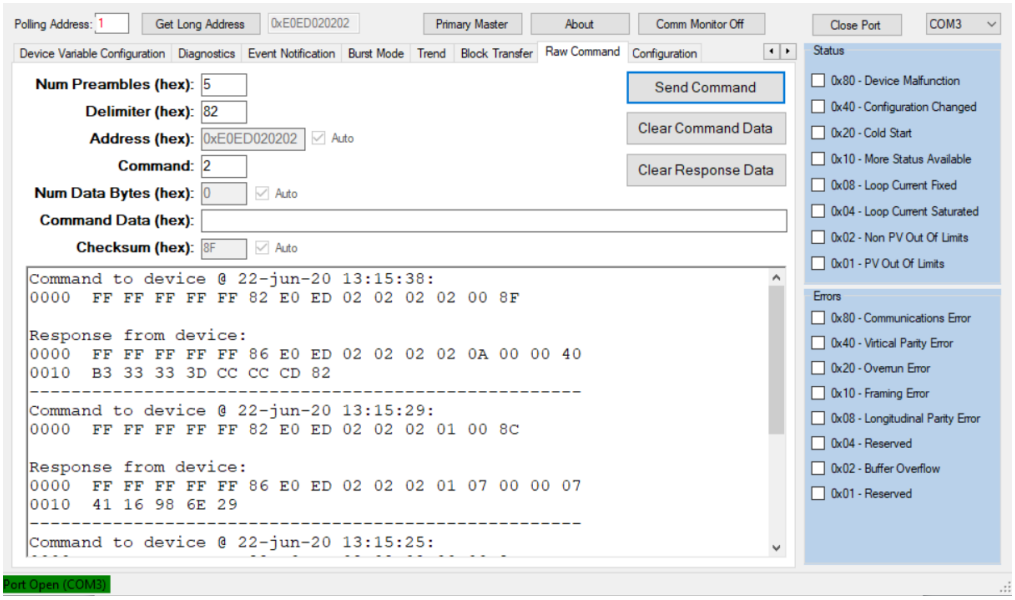

Figure 60: Raw command HART Tools 2.0 paired with Pyboard. Source: own.

Slave device is responsive to all commands executed during the network scan except command 3. For some reason this particular command causes HART Tool to crash. Code does not seem to be the reason for this, as command 3 is implemented using the same methods and functions as commands 1 and 2. Instability has been detected performing other tests, when sometimes the program crashes unexpectedly and after reset it performs the same process that caused the crash without any issue. This leads to a conclusion that probably physical data transmission is not working as it did when virtually supported. And particularly, command 3, may carry a longer processing time for the Pyboard.

7. Environmental impact

This Project main purpose is not that of tackling any environmental effort. That does not mean that through the use of the developed simulator, duplicate field devices will not have to be produced for testing. This is a common reward for simulation, both with a substantial decrease in the investment effort and resources required.

The carbon footprint for the production of a single field device is unavailable online, and giving a numerical estimate of the carbon reduction due to this software falls of this Project scope.

8. Budget

The main imputable charges to this Project are the engineering hours required, the PC amortization, software licensing (virtual ports communication software) and services (internet access). Table 5 summarizes all costs:

Concept	**Amount**	**Unitary value**	**Total cost**
Engineering	300 hours	40 €/h	**12000 €**
Pc amortization	0.25 years	750/5 = 150 €/y	**37.5 €**
Software licensing	1 -	120 €	**120 €**
Services	4 moths	30 €/month	**120 €**
TOTAL			**12775.5 €**

Table 5 Budget. Source: own.

The overall developing costs for this project are 12775.5 €. At this price, the project would be financially successful, since the money saved on acquiring actual field devices would compensate this expenditure in a few projects.

Conclusions and future work

The conclusions of this project are satisfactory. Both initial objectives have been accomplished. As a result from the first objective, this report offers a global overview of the protocol, extracted from various official sources and other HART oriented thesis and projects. All this information has been structured and presented so that any Reader can easily select which sections are of more interest and also get a broader view of how the protocol Works.

For the second objective, the results prove the stability and functionalities of the Emulated Slave network. All the development has been done bearing in mind that this software will be subjected to changes and modifications, thus the intention has been to make the code as easily modifiable and comprehensive as possible. All classes and methods have been constructed with clear divisions that will ease the modification of any specific functionality. One thing that would now have been changed during the development is the fact that the original python software was not designed for microcontrollers, and some modules are not available in MicroPython, thus when the implementation on the Pyboard was made some feature had to be modified or reimplemented with the available modules. Now, probably a parallel development would have given better results and equivalent software for all platforms. This changes and probably, the subjection of the code to the physical environment have caused instabilities when working alongside HART Tool, so until further development and testing is done, it is advised to employ the python 3.8 supported version of the simulated slaves.

For future works regarding this project, the implementation of the Burst mode would be one of the most interesting additions. This mode would allow the Department of Electronics Engineering to test on a continuous stream of communication. Other obvious additions are more command implementations and the expansion of the Device definition, these would give the opportunity to bring the simulated slaves closer to the real field devices in terms of functions.

Acknowledgements

Deep gratitude to Manuel, my project supervisor, for the suggested topic and for all the help received while writing this thesis, and specially for giving me the opportunity to stray from the master specialization I choose and focus my project on something entirely different, something I am now more interested in.

To all my dear friends and flatmates, who accompanied me through these tough years to fulfil my studies, especially the Master phase.

And finally above all I would like to thank my parents Norma and Toni, and my girlfriend Eva for all the love, understanding and support they provided me during all my years of study.

Bibliography

[1] Bowden, Romilly, 2007. “HART Field Communications Protocol - A technical Overview”. HCF_LIT-20 Rev. 3.0, HART Communication Foundation.

[2] HART Communication Foundation, 2010. “HART Communication - Application Guide”. HCF_LIT-039 Rev. 1.0.

[3] Zhang, Peng, 2008. “Industrial Control Technology: A Handbook for Engineers and Researchers”. William Andrew Inc., New York.

[4] SAMSON, 2005. “SAMSON Technical Information - HART communications”. http://www.samson.de/pdf_en/l452en.pdf

[5] HCF (HART Communication Foundation), 2009. HCF-Main pages, March 10th of 2020. http://www.hartcomm.org/

[6] OSI MODEL WIKI PAGE, March 20th of 2020: http://en.wikipedia.org/wiki/OSI_model

[7] http://www.analog.com/ April 5th of 2020

[8] Kingestad Bjørn, 2005. “HART - grensesnitt til feltutstyr”. Universitetet i Stavanger.

[9] Boyes, Walt, 2003. “Instrumentation Reference Book”, 3rd Ed., Butterworth-Heinemmann Publications, Massachusetts.

[10] Ochoa Hidalgo, Richard, 2011. “Sensor diagnostic HART overlay 4-20mA”, University of Stavanger, Faculty of Science and Technology.

[11] HART Communication Foundation, 2010. “Universal Command Specification”. HCF_SPEC-127_7.1.

[12] HART Communication Foundation, 2010. “Token-Passing Data Link Layer Specification”. HCF_SPEC-81_8.2.

[13] HART Communication Foundation, 2010. “Command Summary Specification”. HCF_SPEC-99_9.0.

[14] HART Communication Foundation, 2010. “Common Practice Command Specification”. HCF_SPEC-151_ 8.0.

[15] Daniel Measurement and Control, Inc. “HART Field Device Specification Guide: HART Field Device Specification for Daniel Liquid Ultrasonic Flow Meters revision 2”, 2011.

[16] Gerard Altés, April 2017. “Pyboard based HART-Wifi gateway for monitoring industrial sensors in a mobile app” UPC commons.

Complementary bibliography

This references have not been directly included in the project, but they have served as a source of information during the study phase and later on for development.

- JSON official Organization website. JSON-Main pages. https://www.json.org/json-en.html/
- Python official Organization website. Python-Module description pages. https://www.python.org/
- MicroPython official Organization website. MicroPython-Description pages. https://micropython.org/
- Bristol Babock, 2000. “HART Master Protocol Manual”. https://micropython.org/

.