UM2005
User manual

General purpose GUI r2.0 for STLUX™ and STNRG digital controllers

Introduction

This user manual provides complete information for software developers about the complete STLUX and STNRG general purpose graphic user interface (GPGUI), its use and its integration into applications.

The STLUX/STNRG general purpose graphic user interface is a powerful tool that helps debugging applications making easier for the user to monitor peripherals and SMEDs configuration registers. It also makes more effective and simple interacting with applications tuning and modifying parameters during the testing activity.

The GPGUI can be used also as a powerful tool allowing to connect an STLUX and STNRG device to the STLUX SMED configurator making thus easy to program your device for a fast prototyping proof of the concept and application fine tuning.

The STLUX family of controllers is a part of the STMicroelectronics® digital devices tailored for lighting applications. The STLUX controllers have been successfully integrated in a wide range of architectures and applications, starting from simple buck converters for driving multiple LED strings, boost for power factor corrections, half-bridge resonant converters for high power dimmable LED strings and up to full-bridge controllers for HID lamp ballasts. The STLUX natively supports the DALI via the internal DALI communication module (DCM). The DALI is a serial communication standard used in the lighting industry.

The STNRG devices are a part of the STNRG family of STMicroelectronics digital devices designed for advanced power conversion applications. The STNRG improves the design of the STLUX family to support industrial power conversion applications such as the PFC + LLC, interleaved LC DC-DC, interleaved PFC for smart power supplies as well as the full-bridge for pilot line drivers for electric vehicles.

The heart of the STLUX (and consequently the STNRG where not differently specified) is the SMED (“State Machine, Event Driven”) technology which allows the device to operate several independently configurable PWM clocks with an up to 1.3 ns resolution. An SMED is a powerful autonomous state machine which is programmed to react to both external and internal events and may evolve without any software intervention. The SMED even reaction time can be as low as 10 ns, giving the STLUX the ability of operating in time critical applications.

The SMED devices are configured and programmed via the STLUX internal low power microcontroller (STM8). This user manual describes the whole design flow to easily use the STLUX385A technology.

Contents

List of tables

List of figures

1 Reference documents

- For hardware information on the STLUX and STNRG controllers and product specific SMED configuration, please refer to the STLUX and STNRG product datasheets and reference manual (RM0380).
- For information about the debug and SWIM (single-wire interface module) refer to the “STM8 SWIM communication protocol and debug module” user manual (UM0470).
- For information on the STM8 core and assembler instruction please refer to the “STM8 CPU programming manual” (PM0044).
- For information on the SMED configurator please refer to the UM1981 “SMED configurator v2.0 for STLUX™ and STNRG digital controllers” user manual.
- For information on the STLUX peripheral library please refer to the UM2001 “Standard peripheral library for STLUX™ and STNRG digital controllers” user manual.
- For information on the STEVAL-ILL075V1 or STEVAL-ISA164V1 evaluation boards please refer to the product datasheet.

2 Acronyms

In *Table 1* is a list of acronyms used in this document:

Table 1. List of acronyms

Acronym	Description
ACU	Analog comparator unit
ADC	Analog-to-digital converter
ATM	Auxiliary timer
AWU	Auto-wakeup unit
BL	Bootloader - used to load the user program without the emulator
CCO	Configurable clock output
CKC	Clock control unit
CKM	Clock master
CPU	Central processing unit
CSS	Clock security system
DAC	Digital-to-analog converter
DALI	Digital addressable lighting interface
ECC	Error Correction Code
FSM	Finite state machine
FW	Firmware loaded and running on the CPU
GPGUI	General purpose graphic user interface
GPIO	General purpose input output
GUI	Graphic user interface
HSE	High-speed external crystal - ceramic resonator
HSI	High-speed internal RC oscillator
I^2C	Inter-integrated circuit interface
IAP	In-application programming
ICP	In-circuit programming
ITC	Interrupt controller
IWDG	Independent watchdog
LIN	Local Interconnect Network
LSI	Low-speed internal RC oscillator
MCU	Microprocessor central unit
MSC	Miscellaneous
PM	Power management
RFU	Reserved for future use

Table 1. List of acronyms (continued)

Acronym	Description
ROP	Read-out protection
RST	Reset control unit
RTC	Real-time clock
SMED	State machine, event driven
STMR	System timer
SW	Software, is the firmware loaded and running on the CPU (synonymous of FW)
SWI	Clock switch interrupt
SWIM	Single-wire interface module
UART	Universal asynchronous receiver/transmitter
WWDG	Window watchdog

3 General purpose graphic user interface components

The STLUX general purpose graphic user interface is composed of two parts: the GPGUI software running on the PC host and the GPGUI firmware to be integrated in applications running on the STLUX target platform. The two parts communicate via a serial cable thanks to a proprietary communication protocol that will be specified later in *Section 6 on page 28*.

This document describes the functionality of the GPGUI software and how to easily integrate the GPGUI firmware in your application. The GPGUI firmware technology has been developed basing on the STLUX peripherals library.

The purpose of this GUI is to provide a flexible standard interface for applications based on the STLUX and STNRG devices. Moreover the GUI represents a resource dedicated to application developers aiming to develop applications which firmware integrates a fast and effective interface based on serial port communication. Last but not least, the general purpose GUI has been conceived as a powerful tool to be mapped on the STEVAL-ILL075V1 or STEVAL-ISA164V1 evaluation boards to allow the fast prototyping, debug and parameters configuration of proof of concept applications.

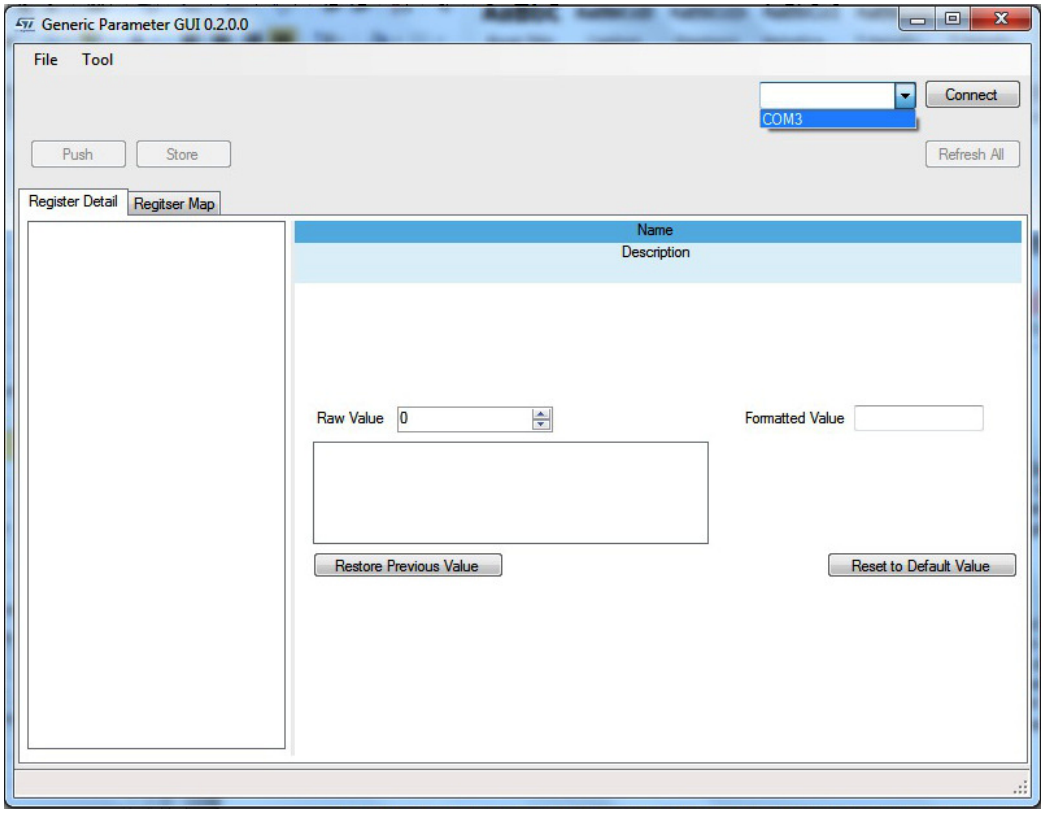
4 General purpose graphic user interface software

The aim of this paragraph is to give you a brief overview of the STLUX general purpose graphic user interface software. This interface is meant to run on a host PC connected via a serial cable to an STLUX target running the GPGUI firmware integrated in a generic application. The general purpose GUI tool suite is composed of two executable applications, the general purpose GUI application and the general purpose XML File Editor described here below in detail.

4.1 Graphic user interface

The general purpose graphic user interface is a helpful application independent and platform independent tool aimed first of all for runtime application debug and also for on the fly application parameters configuration and fine tuning. As a general purpose interface, it can blind connect via the serial connection to any STLUX based platform running the general purpose GUI firmware and automatically retrieve the basic information from it together with the generic application parameters made available by the application in “parameters configuration” mode which is the standard safe mode. Launching the GPGui.exe will open the GUI appearing as shown in *Figure 1*.

Figure 1. General purpose GUI window



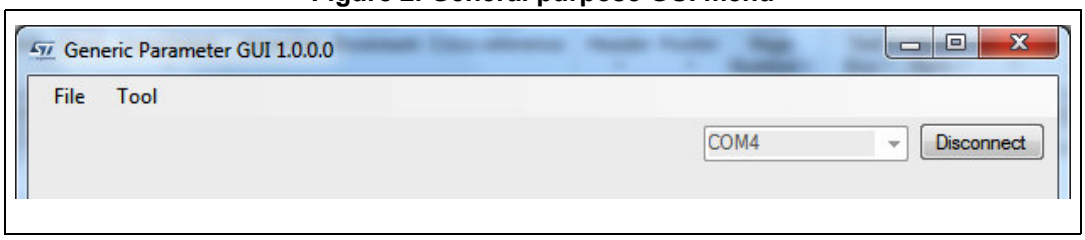
Once you connected your computer with a target platform via a serial cable, you are ready to connect the general purpose GUI with it. To do this you simply have to chose in the dialog box the right port and push the [connect] button. Once the connection is established, the GUI starts sending the information request to the target about the Product ID and Revision number to be able to look for XML files associated to the current application running on the target.

The GUI starts as the default mode in the parameters configuration mode asking for the number of parameters to be displayed and their name. In case a “Parameters Definition” XML file is associated, it shows all the information associated with the parameters and the fields format stored in the XML file. If a “Register Definition” XML file is found, the GUI switches to the debug mode also giving direct access to all the listed STLUX registers through their physical addresses.

4.1.1 General purpose GUI menu

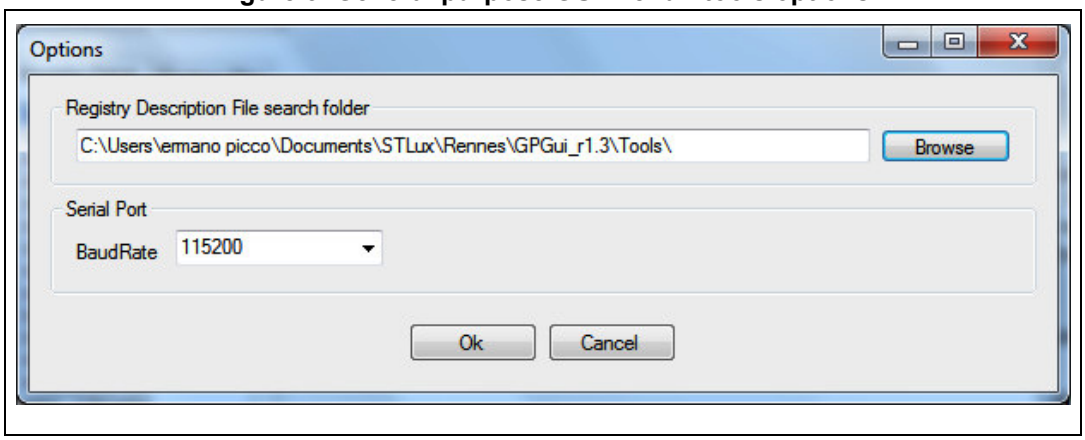
The general purpose GUI menu is composed of two options. The first option “File” allows to connect the GUI to a device by choosing an I/O port. The same operation can be easily done by pushing the Connect/Disconnect button on the upper right of the frame window.

Figure 2. General purpose GUI menu



The second choice available on the menu is the “Tool” option. This window allows to configure some general purpose GUI parameters like the source folder where to look for the *.xml configuration files. Here it is also possible to set the baud rate configuration for the serial port connection. Aside when specifically specified, the default general purpose GUI baud rate is set to 112500 bps.

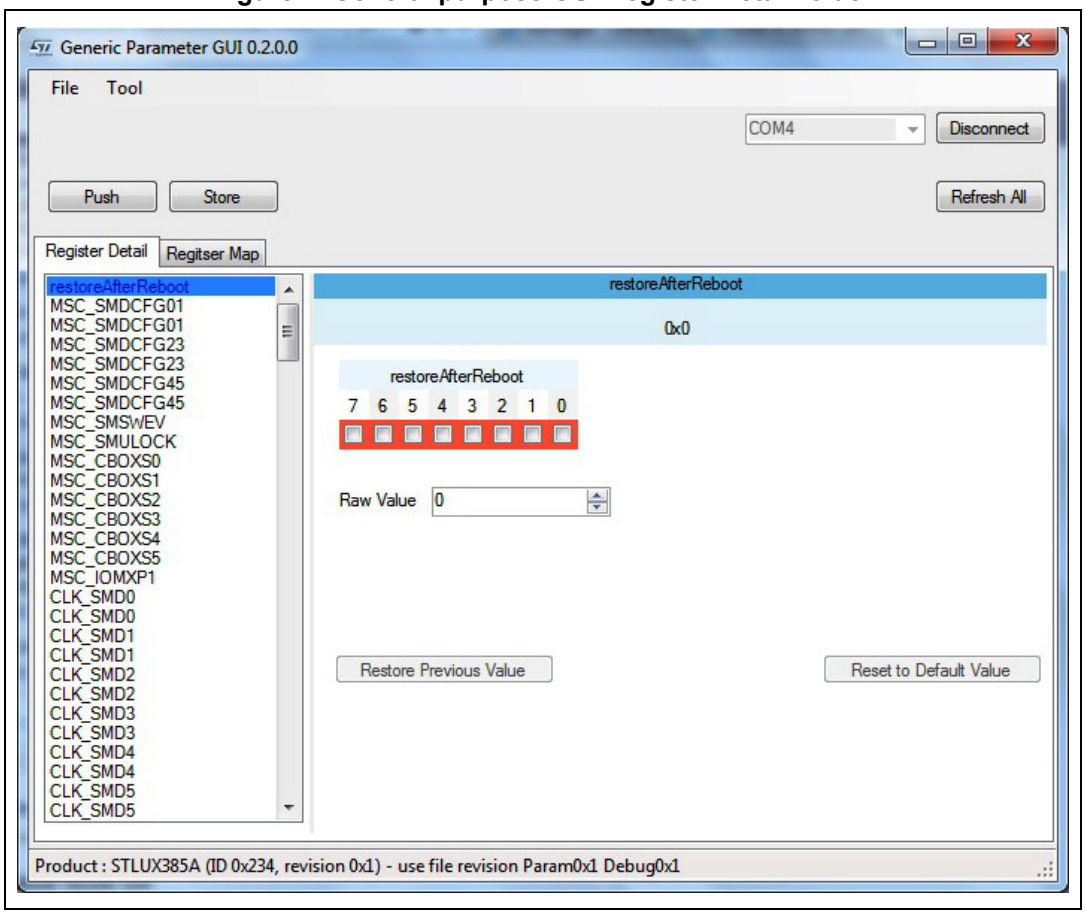
Figure 3. General purpose GUI menu - tools options



4.1.2 Register Detail folder

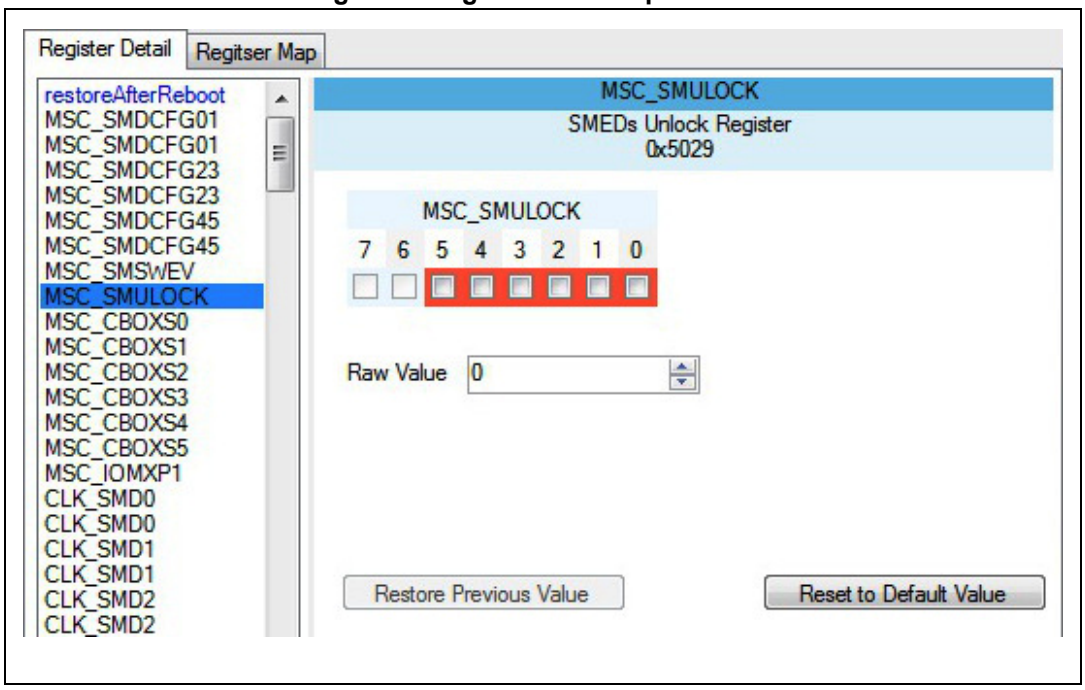
During this startup phase, all the parameters and registers values are loaded from the target and displayed in the “Register Detail” folder as shown in *Figure 4*.

Figure 4. General purpose GUI Register Detail folder



Digging deeper into the Register Detail folder it is possible to see how each application - the Parameter or STLUX Register is shown with a description on the top, its index/address and its bits field. Please note that for each field, it is possible to highlight the read-write access parts in red while keeping the read-only parts in cyan.

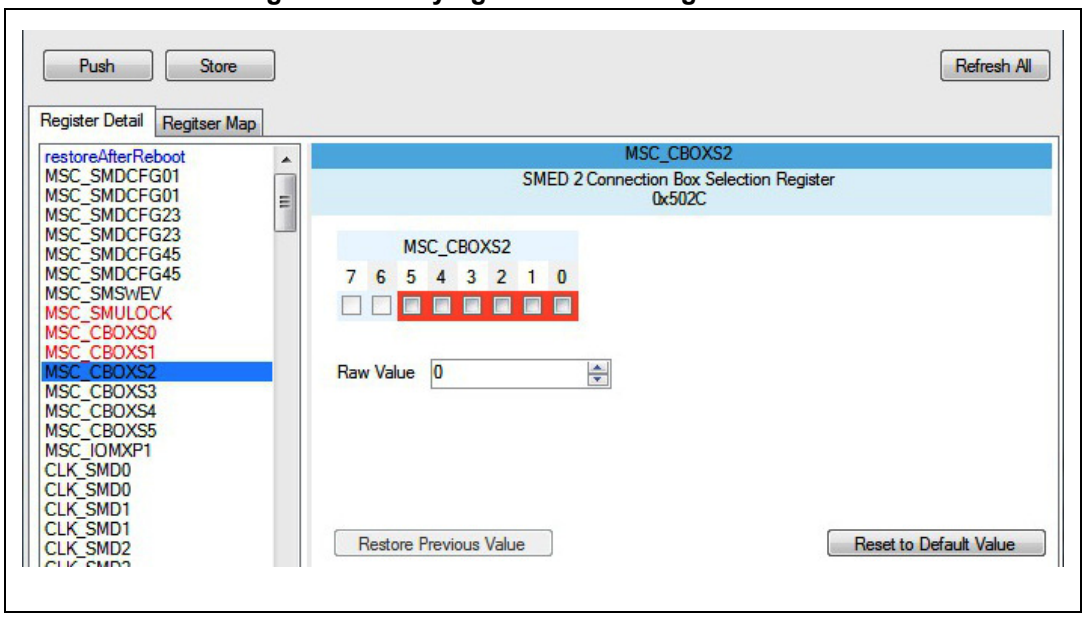
Figure 5. Register Detail input format



For example the STLUX SMEDs Unlock Register MSC_SMULOCK uses only 6 bits while the two MSB are read-only and the maximum valid input value is fixed consequently. All these rules of course can be specified in the XML Register definition file as explained in *Section 4.2*.

In the Register Detail folder you can set values for all the parameters and registers, restore them to their previous value or even reset them to their default value if defined. Every time you will modify a value, the correspondent field will turn to red on the left side list as shown in *Figure 6*.

Figure 6. Modifying Parameters/Registers value



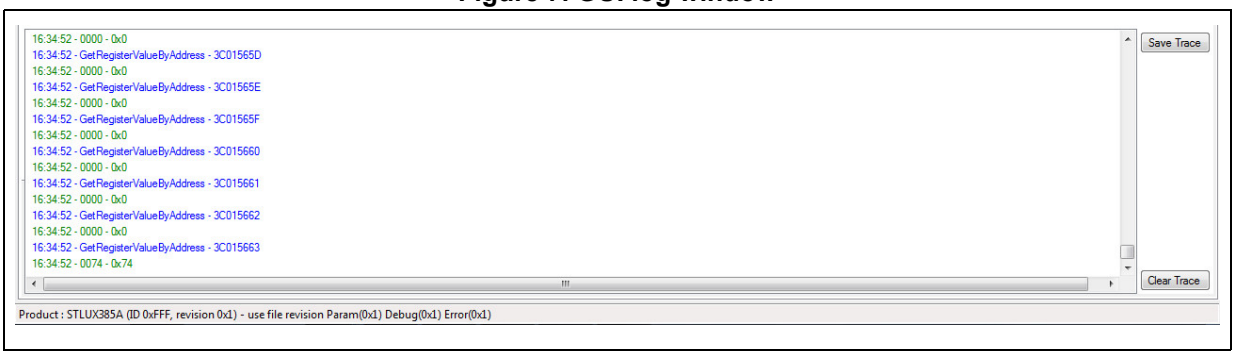
All the modified values though, won't be sent to the target application to be applied until the [Push] button is pressed. Clicking on the push button translates into a series of commands sent via a serial cable to apply all the modifications in the GUI to the target.

At the same way also a [Store] button is available. A store command sent to the target will enable the application to store all the meaningful Parameters/Registers into the EEPROM in order to be able to restore them at the next power-up if enabled. To do so, a RESTORE generic parameter has been added which takes the possible ENABLE/DISABLE entries.

4.1.3 General purpose GUI log window

At the end of a push or any other operation, you will be able to see in the lower part of the Register Map folder, the log history of all the commands followed by the result of the operations as you can see in *Figure 7*.

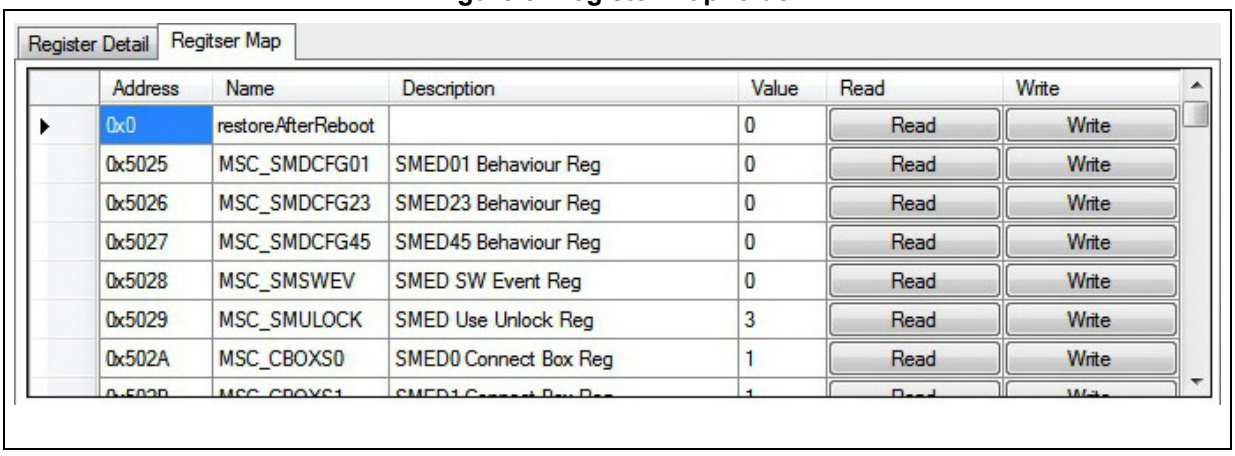
Figure 7. GUI log window



Of course the general purpose GUI log window keeps the track of all the commands sent via the serial port and all the information received by the application. In every moment is it possible also to clear the log window history pushing the “Clear Trace” button. As well it's possible to save all the history to an ASCII *.log file pushing the “Save Trace” button. This is useful to keep the track of the operations done during the debug and specific testing procedures.

4.1.4 Register Map folder

Figure 8. Register Map folder



The “Register Map” folder is divided into two sections. In the previous paragraph we already introduced the lower section which is the GUI log window, so now let's focus on the upper section that's specifically meant as the true Register Map. As shown in all the memory with both Parameters and Registers is represented here as a table. Each row represents a basic memory element and each column represents its properties. For Parameters the Address is associated with their numeric index while for STLUX Registers the Address is intended as the physical address. Also here for each element it is possible to access it through [Read] and [Write] buttons that will be enabled according to the previously defined register access properties. Please note that this window is meant to give a global representation of the application memory and checks and formats on input/output values don't apply here, therefore it is preferable to modify values in the Register Detail folder.

4.2 XML files structure

As an integral part of the general purpose GUI, a couple of XML format files must be provided to store information about both Registers and Parameters. They also contain information about the data meaning, data format and checks, so to make more effective and handy the data input and helping to keep simple the complexity of the firmware on the target side.

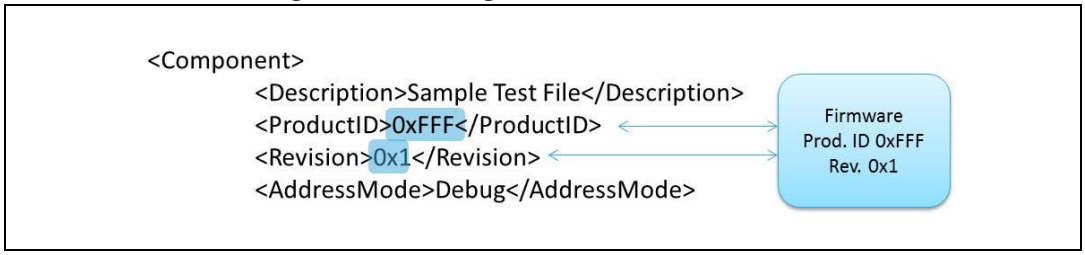
4.2.1 Register definition file

The Register definition file is mandatory only in the debug mode and makes accessible the STLUX internal registers through their physical address. If the Registers definition file is missing, the GUI skips the STLUX registers section and switches to “Generic Parameters Access” mode.

XML Register definition file header

The Registers definition XML file is divided into sections. The first section is the “Header” providing identification information about the specific application to which the XML file is related. In particular when opening a connection with the target device, the general purpose GUI sends a request for the Product ID and the Revision of the application firmware running so to be able to find, among several XML files, the correct ones suited for the current device. The specified address mode for the Registers is set to debug, that means addresses are specified as physical addresses. They can be expressed both in the hex and dec format.

Figure 9. XML Register definition file header



XML Register definition file body

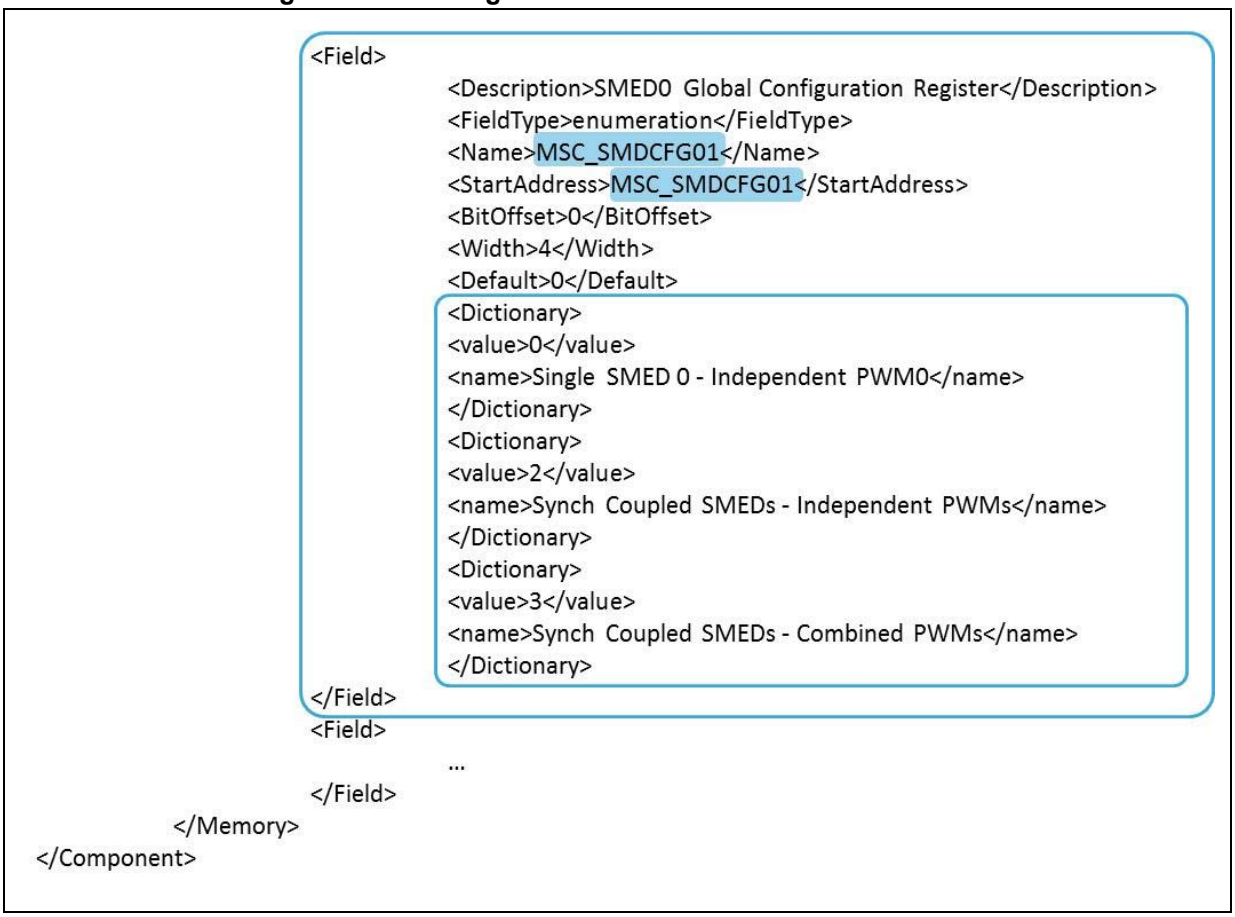
The second section of the Registers definition XML file is the body included within the <Memory> </Memory> tags. Inside this body there can be two kinds of declarations. First there are the registers size and address declaration which specifies the memory basic structure. Then it is followed by the registers declarations included in the <Register> </Register> stating for each register a description, the physical address, the name and the access properties.

Figure 10. XML Register definition file Memory structure and registers declaration

```
<Memory>
        <RegisterSize>size_8bits</RegisterSize>
        <AddressSize>size_16bits</AddressSize>
        <Register>
                <Description>SMED01 Behaviour Reg</Description>
                <Address>0x5025</Address>
                <Name>MSC_SMDCFG01</Name>
                <Access>ReadWrite</Access>
        </Register>
        <Register>
                ...
        </Register>
```

After having declared all the STLUX registers to be included in the debug mode, for each register can be defined a “Field”. A Field declaration specifies logical data that can involve one or more registers for which input/output rules can be defined such as the maximum, minimum and default value, input/output format, whether it is an integer or bitfield or enumeration type and in case of the latter, an entries dictionary can be defined as can be seen in *Figure 11*. Please note that in the field declaration the StartAddress can be specified as a Register name.

Figure 11. XML Registers definition file Fields declaration



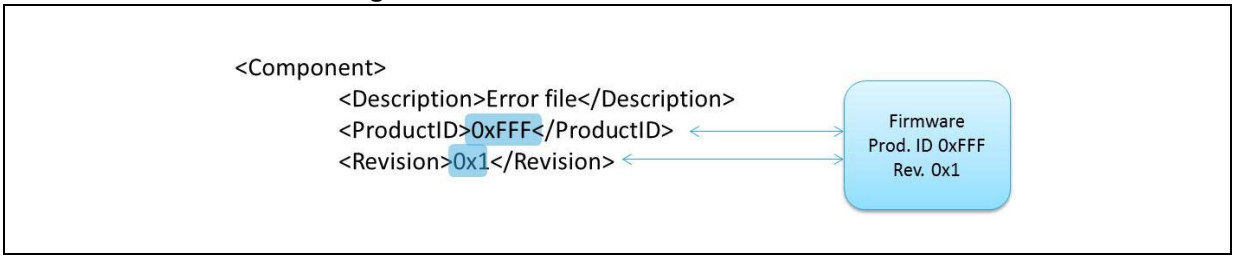
4.2.2 Errors definition file

The Errors definition file is an optional file allowing to an application to define the specific error codes of an application and their specific error messages. If the Errors definition file is missing, in case of error the GUI simply communicates this is occurred through a generic error message but there's no chance to have information about its details.

XML Errors definition file header

As for Registers and Parameters definitions, also the Errors definition XML file is divided into sections. The first section is the “Header” providing identification information about the specific application to which the XML file is related. As previously said, this is needed to identify, among several XML files, the correct ones suited for the current application.

Figure 12. XML Errors definition file header



XML Errors definition file body

The second section of the Errors definition XML file is the body included within the <Error code> </Error> tags. Inside this body all the error codes and error messages to be displayed are declared.

Figure 13. XML Errors definition file declaration

```
<Error code="1">Value out of Range</Error>
<Error code="2">Error Description for code 2</Error>
<Error code="3">Error Description for code 3</Error>
<Error code="4">Error Description for code 4</Error>
<Error code="5">Error Description for code 5</Error>
<Error code="6">Error Description for code 6</Error>
</Component>
```

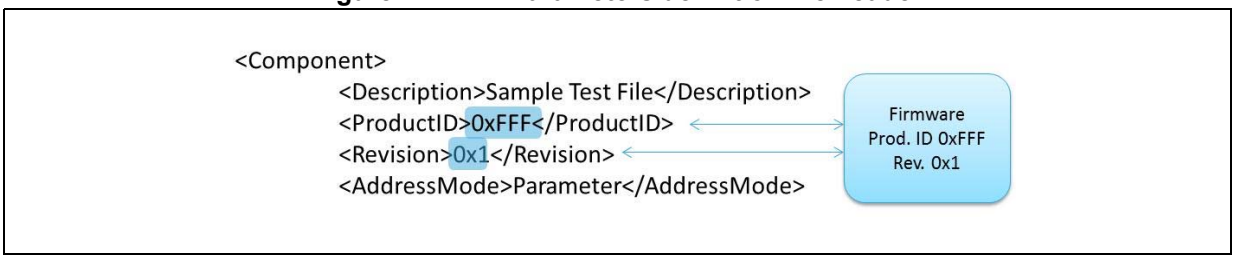
4.2.3 Parameters definition file

The Parameters definition file is an optional file allowing to an application to define the specific parameters of an application and their input/output format. If the Parameters definition file is missing, the GUI loads from the application the parameters number and names but there's no chance to have information about the input/output fields format.

XML Parameters definition file header

As for Registers definitions, also the Parameters definition XML file is divided into sections. The first section is the “Header” providing identification information about the specific application to which the XML file is related. As previously said, this is needed to identify, among several XML files, the correct ones suited for the current application. The specified address mode for parameters configuration is set to Parameter, that means addresses are specified as incremental numeric indexes. They can be expressed both in hex and in dec format.

Figure 14. XML Parameters definition file header



XML Parameters definition file body

The second section of the Parameters definition XML file is the body included within the <Memory> </Memory> tags. Inside this body there can be two kind of declarations. First there are the registers size and address declaration which specifies the memory basic structure. Then it is followed by the Parameters declarations included in the <Register> </Register> stating for each parameter a description, the index, the name and the access properties.

Figure 15. XML Parameters definition file Memory structure and parameters declaration

```
<Memory>
        <RegisterSize>size_8bits</RegisterSize>
        <AddressSize>size_16bits</AddressSize>
        <Register>
                <Description>Restore After Reboot</Description>
                <Address>0x0</Address>
                <Name>RESTORE</Name>
                <Access>ReadWrite</Access>
        </Register>
        <Register>
                ...
        </Register>
```

After having declared all the application parameters to be included in the Parameters mode (names must match the ones declared in the firmware), for each parameter a “Field” can be defined. A Field declaration specifies logical data that can involve one or more parameters for which input/output rules can be defined such as the maximum, minimum and default value, input/output format, whether it is an integer or bitfield or enumeration type and in case of the latters, an entries dictionary can be defined as can be seen in *Figure 16*. Please note that in the field declaration the StartAddress can be specified as a Parameter name.

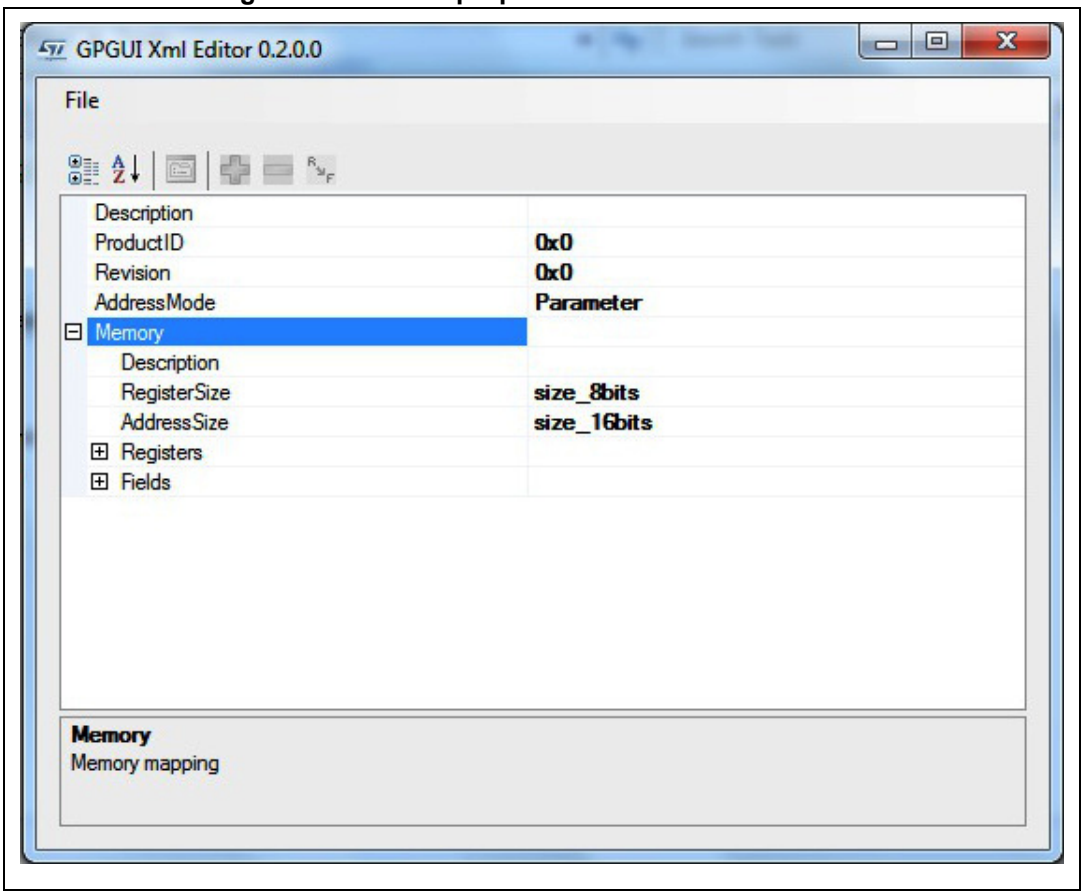
Figure 16. XML Parameters definition file Fields declaration

```
        <Field>
                <Description>Restore  After Reboot</Description>
                <FieldType>enumeration</FieldType>
                <Name>RESTORE</Name>
                <StartAddress>RESTORE</StartAddress>
                <BitOffset>0</BitOffset>
                <Width>1</Width>
                <Min>0</Min>
                <Max>1</Max>
                <Dictionary>
                 <value>0</value>
                <name>Disable</name>
                </Dictionary>
                <Dictionary>
                <value>1</value>
                <name>Enable</name>
                </Dictionary>
        </Field>
        <Field>
                ...
        </Field>
    </Memory>
</Component>
```

4.3 General purpose XML File Editor

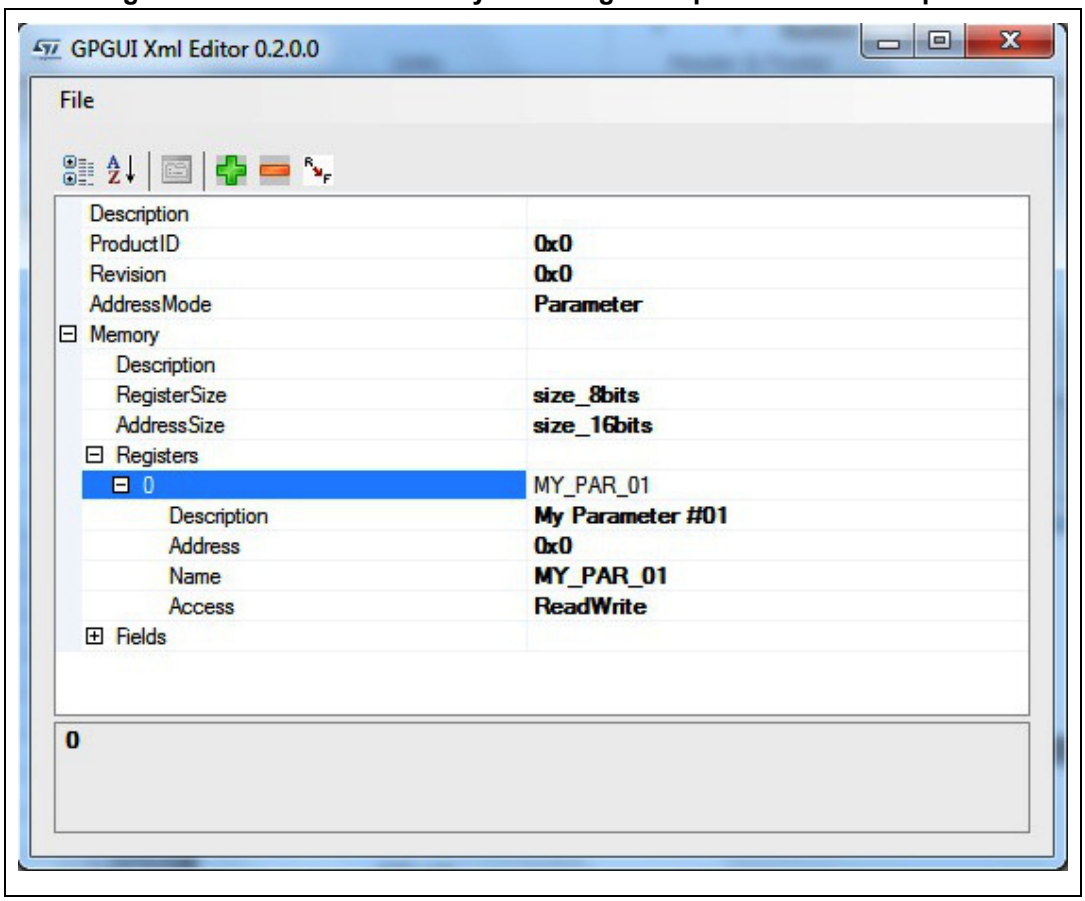
The general purpose XML File Editor is aimed for computer aided specification of the XML files that can be integrated with the general purpose GUI for both the parameters configuration and debug mode. Launching the GPFileEditor.exe will open the XML File Editor appearing as shown in *Figure 17*.

Figure 17. General purpose XML File Editor window



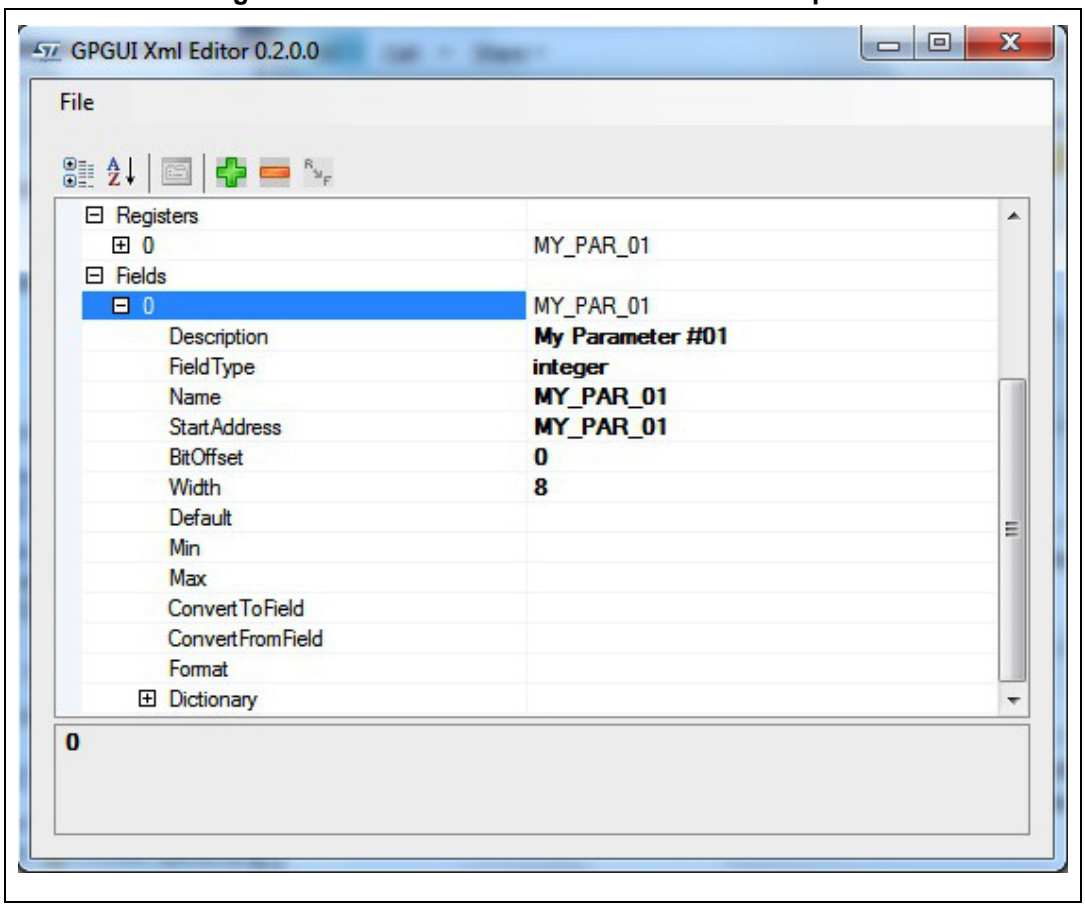
As you can see, the editor makes easier to specify all the settings required in a general purpose GUI XML file like Product ID, Revision number, whether you want to access registers in the debug mode or Parameters in the parameter configuration mode, the Memory structure and more. Then you can declare registers by simply adding registers using the [+] button which adds an “empty” register to be filled with the desired properties.

Figure 18. How to add an entry in the registers/parameters description



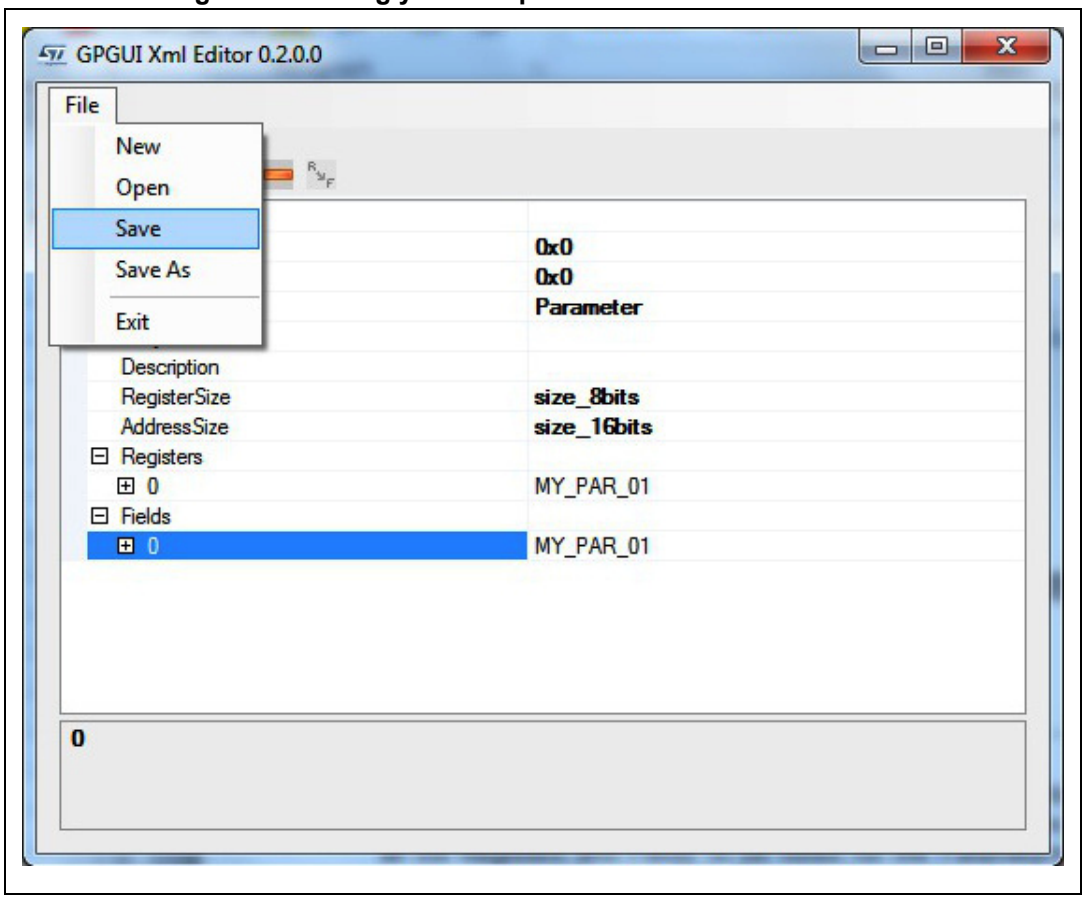
As can be seen here in *Figure 18*, we filled the new empty register creating MY_PAR_01, a 8-bit parameter with read/write access which is labeled with the index 0x0. Now selecting with the cursor the register name, by simply clicking on the [R->F] button, the tool automatically generates a field definition basing on the register. Please note that you can specify the StartAddress by using the name of the respective register in place of the correspondent index.

Figure 19. How to add a field in the fields description



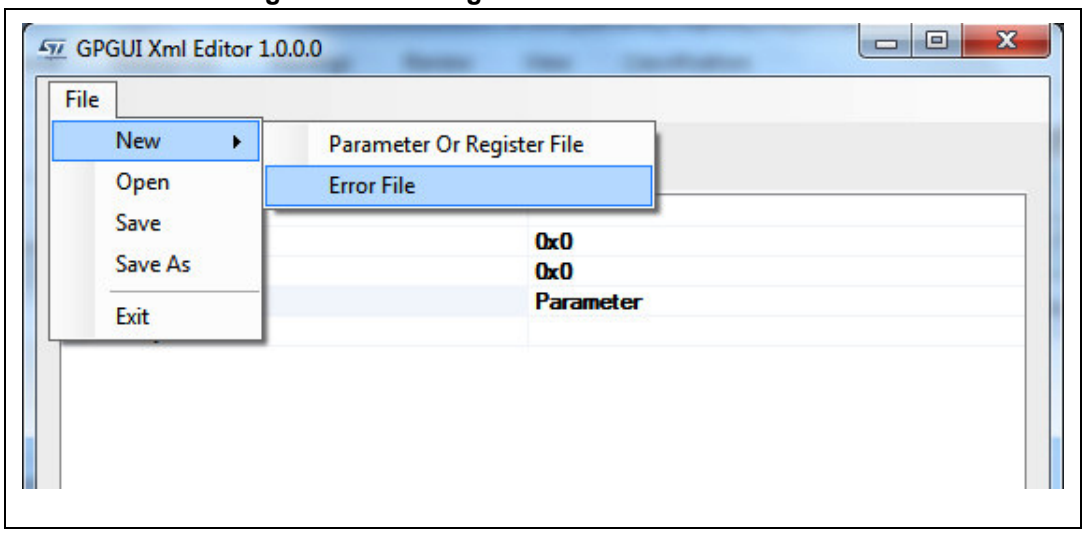
Another possible way to add a field is to go into the fields section and push the [+] button. This will add an empty field description to be associated with one or more registers definition. Once you specified all the Registers and Fields to be listed for the parameters configuration mode or for the debug mode, you can save it in the XML format to be used by the general purpose GUI or to be retrieved for further modifications by the general purpose GUI XML File Editor.

Figure 20. Saving your component definition to an XML file



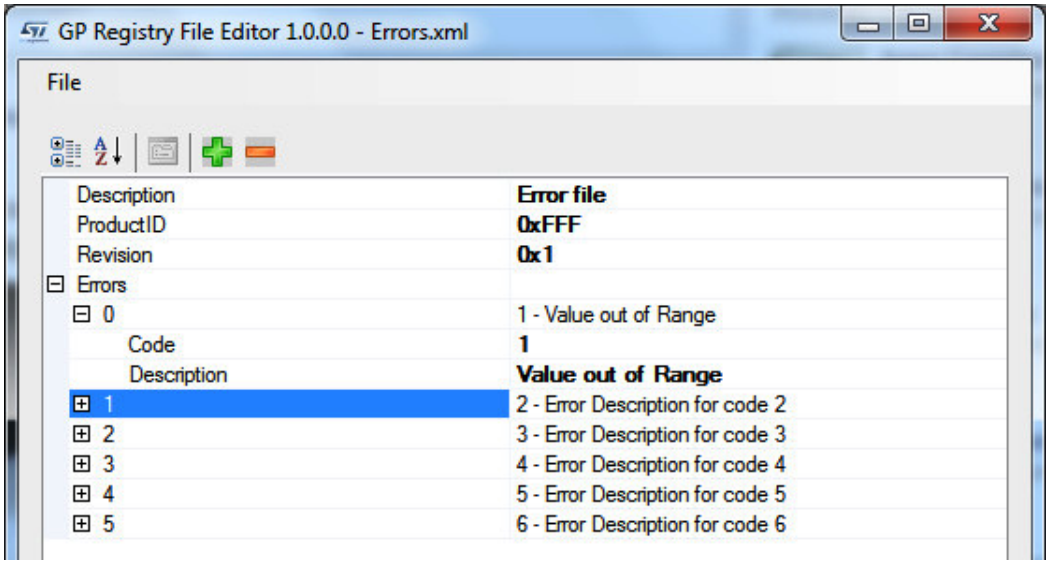
The general purpose XML File Editor also allows creating Errors definition files by simply choosing the proper file format while creating a new file.

Figure 21. Creating an XML Errors definition file



An XML Error definition file basically contains a dictionary of error codes characterized by a progressive positive index. Each error code gets associated to a proper error message defined for the specific application. This way, each time the firmware of the application returns as a response to the host a positive integer error code, a pop-up window will appear showing the error message defined to the respective error code. This is useful not only to handle the communication error but to make the general purpose GUI more flexible also handling logic error conditions for the given application.

Figure 22. Editing an XML Error definition file

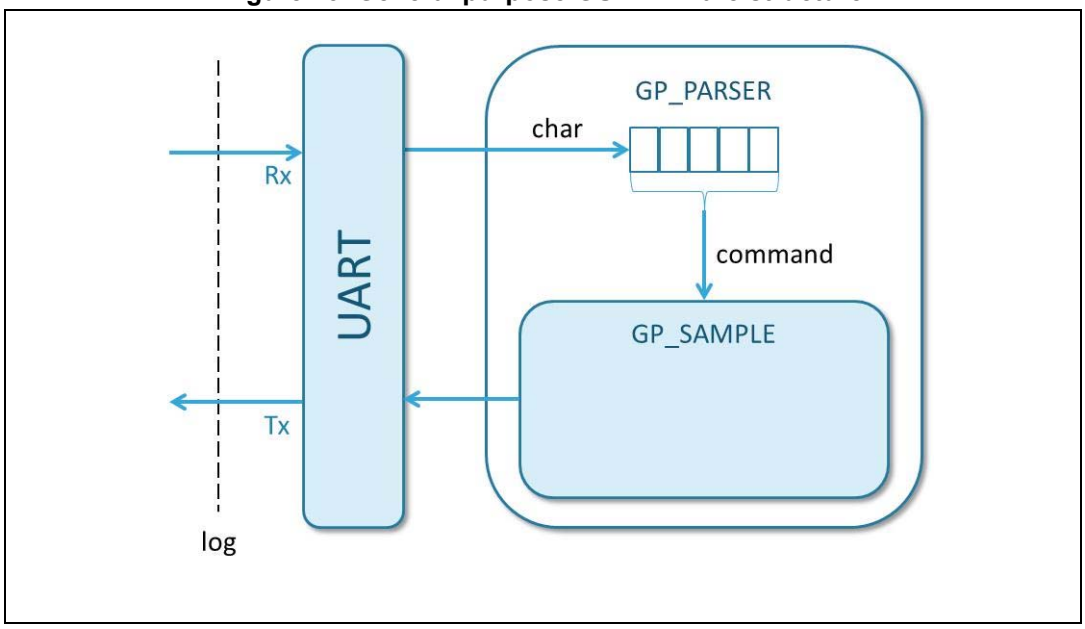


5 General purpose graphic user interface firmware

The general purpose GUI firmware is composed of a software IP that can be integrated with applications to communicate with the general purpose GUI using a protocol that will be specified in *Section 6 on page 28*. This protocol allows communication via the UART serial interface set of general purpose commands that are specified from *Table 4 on page 28* to *Table 7 on page 29*.

The UART communication is based on the transmitter and receiver interrupt handling transmit and receive buffers where the sent/received characters are stored to be processed. Therefore the firmware first of all properly initializes the UART peripheral and interrupts and gets the UART peripheral ready to handle the serial link. The UART link must be set with the 8-bit data, 1 stop bit and no parity bit, the LIN disabled and baud rate of 115200 bps.

Figure 23. General purpose GUI firmware structure



The core of the GPGUI firmware is the parser decoding each incoming bite. This is structured in two logical software layers: the outer is represented by the GP_PARSER (gp_parser.h, gp_parser.c) that basically implements the communication basic function, recognizing line breaks, translating bites in characters, etc. The inner layer is represented by the GP_SAMPLE (gp_sample.h, gp_sample.c) which effectively implements the GUI core functionalities.

5.1 Using the general purpose GUI firmware

The general purpose GUI firmware is available to the user both as a binary file and as an open source code for application developers to be integrated in STLUX/STNRG based applications. Starting from the open source GPGUI firmware, the developer will be able to easily build his own application from the “empty” main routine which simply initializes the serial port communication protocol environment.

5.2 General purpose GUI firmware integration

Integrating the general purpose GUI firmware into an existing application can be easily done by including the GP_PARSER and GP_SAMPLE files (both *.h and *.c) and properly configuring the UART communication protocol and interrupt routines as specified in the general purpose GUI project available for Cosmic, IAR and Raisonance environments.

5.2.1 General purpose GUI parametric firmware configuration

In particular to make the firmware suitable for all the various STLUX devices, the code has been made automatically configurable at compile time through definition of compilation flags defining the specific device we want to address. So it is advisable before compiling, to state what kind of the STLUX device we are using declaring the flags:

- _STLUX385A_
- _STLUX383A_
- _STLUX325A_
- _STLUX285A_
- _STNRG388A_
- _STNRG328A_
- _STNRG288A_

General purpose GUI parametric firmware auto configuration

This will optimize the code for the specific device at compile time, resulting in a shorter and more efficient code running on the board. In case none of the above declared compilation flags is set to identify a specific platform, the _AUTO_ID_ flag is set by default and the auto identification mode is enabled in order to make the firmware code less efficient but able to properly identify the device in the STLUX family and configure itself to properly operate with it.

In particular, the code configuration or auto configuration relates to different parts.

5.2.2 Option bytes

The general purpose GUI firmware has been specifically conceived not only as a tool to allow developers to easily integrate the GUI into their existing applications. It has been conceived also to start developing applications from scratch starting from the “empty application” general purpose GUI firmware.

Last but not least it has been thought to be downloaded to the STEVAL-ILL075V1 or STEVAL-ISA164V1 evaluation board as a fast prototyping platform allowing to pilot SMEDs and peripherals. To make this platform even more easily accessible, option bytes have been specifically set to easily allow the bootloading procedure at every power-up.

In particular the MSC_OPT0 and nMSC_OPT0 option bytes must be properly set according to the chosen platform configuration to enable the proper UART channel as the bootloading source.

Table 2. UART line selection option bytes

Device	MSC_OPT0	nMSC_OPT0	UART pinout
STLUX385A	0x11	0xEE	GPIO0[0] GPIO0[1]
STLUX383A	0x11	0xEE	GPIO0[0] GPIO0[1]
STLUX325A	0x31	0xCE	GPIO0[4] GPIO0[5]
STLUX285A	0x31	0xCE	GPIO0[4] GPIO0[5]
STNRG388A	0x11	0xEE	GPIO0[0] GPIO0[1]
STNRG328A	0x31	0xCE	GPIO0[4] GPIO0[5]
STNRG288A	0x31	0xCE	GPIO0[4] GPIO0[5]

Moreover, to allow the bootloader to wait for a code source for a second right after each reset, the OPTBL and nOPTBL option bytes must be set to 0x55AA.

In the main general purpose GUI firmware, a specific function takes care of properly setting these option bytes to grant anyway bootloading compliance.

5.2.3 UART channel configuration

As for the bootloading option bytes, defining the kind of the device for which the code is being compiled also sets the proper UART ports configuration to ensure the serial interface connection for the GUI.

As previously stated, defining no specific device will result in auto identification mode running firmware which will automatically determine the device and set the UART ports according to *Table 2*.

5.3 General purpose parser API

Table 3. General purpose parser functions

Header	Input parameters	Output parameters	Functionality
ParseByte	toProcess: it is an input character	-	Parses an incoming character to identify the start of an incoming command and interprets it.
LineBreakDetection		-	Must be called in case LineBreak interrupt is received to restart parsing new commands.
SetError	isError: it can be TRUE or FALSE	-	It tells the parser whether an error has occurred or not. In case isError is FALSE, the error is not echoed.
Restore_Regs		-	It checks whether the RESTORE flag is enabled. Then if the CRC check is OK, it restores all the registers configuration previously dumped to the EEPROM.
InitCallback		-	It initializes all the parser commands callbacks and gets the parser ready to receive input characters.

In order to customize the general purpose GUI for a specific application, few simple steps are needed. The first step is to include the GPGUI firmware files and to properly initialize the UART peripheral. To ease this step a basic empty application just running the GPGUI is available. This empty application can also be used as a starting point to develop a new application from scratch.

The second step is to choose a set of parameters for the specific application that will be included in the XML Parameters definition file and also appended to the list of Parameters in the GetRegVal /SetRegVal firmware functions. Also the set of registers in the debug mode can be modified according to the specific application by simply modifying the XML Registers definition file. This customization will make all the application relevant registers and variables available to be modified and monitored during the execution of the application. If needed, further checks and special functions (i.e.: regulation loops) can be added in the Push() function that's performed every time a push event occurs.

The last customization that can be done is relative to the Store()/Restore() functionalities, where each software developer will be able to declare his own list of parameters and registers to be stored in the EEPROM and eventually restored at each power-up.

5.4 General purpose GUI FW upload

Uploading the general purpose GUI to your device is an easy task and can be accomplished basically in two ways. The first way is via the SWIM interface using the supported tools, Cosmic, IAR Systems® and Raisonance. For more information about how to upload a bit file, please refer to the specific tool user manual.

The second way to upload your bit file is via bootloading procedure. This can be done using the STLUX Flash loader demonstrator tool provided by ST. Bootloading is a feature enabled by default on brand new devices which have never been programmed. For more details on how to bootload your code on the STLUX / STNRG, please refer to the application note AN4656.

6 General purpose graphic user interface serial protocol

The communication over the serial port is exchanged using a protocol based on short messages. This is aimed for having a tool to easily handle parameters and monitor platform registers for the application configuration and debug purpose with reduced data exchange over the serial interface and optimized memory usage. The protocol uses different messages of fixed length according to the purpose of it.

Figure 24. GUI serial protocol messages length

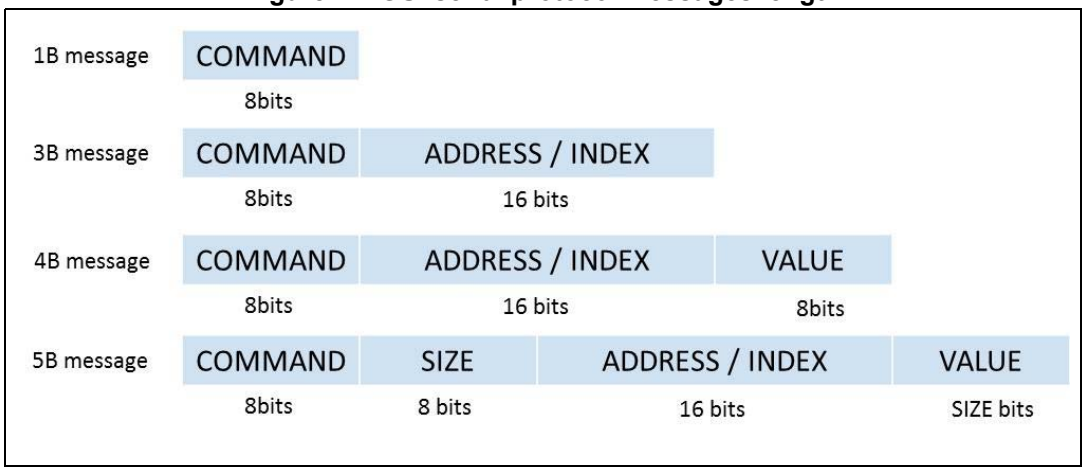


Table 4. 1 byte length messages

Command	Size	Code	Input parameters	Response	Description
SetASCII	1 B	0x61 ('a')	-	-	Requests the target to run in the ASCII mode.
SetBinary	1 B	0x62 ('b')	-	-	Requests target to run in the binary mode.
GetID	1 B	0x69 ('i')	-	Product ID + Revision number	Requests the target to return the Product ID and Revision number.
GetProductName	1 B	0x74 ('t')	-	Product name	Requests the target to return the product name.
GetRegCount	1 B	0x63 ('c')	-	Number of parameters	Requests the target to return the total amount of parameters.

After getting a GetProductName, the target will return a string identifying the exact product featured on-board. The possible product names will be divided into two families.

The STLUX family:

- STLUX385A
- STLUX383A
- STLUX325A
- STLUX285A

The STNRG family:

- STNRG388A
- STNRG328A
- STNRG288A

Table 5. 3 bytes length messages

Command	Size	Code	Input parameters	Response	Description
getRegName	3 B	0x6E ('n')	Parameter index (2 B)	Current parameter name	Requests the target to return the name of a parameter.
GetRegVal	3 B	0x72 ('r')	Parameter index (2 B)	Current parameter value + OK / error	Requests the target to return the current value of a parameter.
Store	3 B	0x73 ('s')	Product ID and Revision number	OK / error	Requests the target to perform a store procedure. Stores register values into the Flash.
Push	3 B	0x70 ('p')	Product ID and Revision number	OK / error	In case it sets the modified registers values, then it requests the target to and performs a push.

Table 6. 4 bytes length messages

Command	Size	Code	Param.	Response	Description
SetRegVal	4 B	0x77 ('w')	Parameter index (2 B) + value (1 B)	OK / error	Requests the interface to set the value of a parameter.
getRegValByAddr	4 B	0x3C ('<')	Size (1 B) + register physical address (2 B)	Current register value + OK / error	Requests the interface to return the current value of a register at a specified physical address.

Table 7. 5 bytes length messages

Command	Size	Code	Param.	Response	Description
setRegValByAddr	5 B	0x3E ('>')	Size (1 B) + register physical address (2 B) and value (1 B)	OK / error	Requests the interface to set the value of a register at a specified physical address.

Each message is sent byte after byte from the PC host. The target replies at every byte replies with the echo of the received byte. After receiving the whole command line, it appends to the last byte echo an error code (1 byte) and the reply to the command.

The error code will be zero in case the command execution succeeded or will be a positive integer in case of error. Every specific number will identify a precise type of error which will be associated to an error message defined in the file Errors.xml as described in *Section 4.2.2 on page 16*. Here below in you can find an example of the log of the typical serial communication between the host PC and target application.

Figure 25. Typical serial communication log among GUI and application

7 Revision history

Table 8. Document revision history

Date	Revision	Changes
05-Apr-2016	1	Initial release.