|  |  |
| --- | --- |
| DENOMINAZIONE :  description | **DAQs4Comm SW application is useful to read / write (also automatically by reading a list of commands)** **and log the Digital Input / Output Data from all the ADVANTECH USB 5856 connected to PC** |
| RIFERIMENTI :  references | [www.advantech.com](http://www.advantech.com) (Advantech website) |
|  |  |
|  |  |

**LIST OF EFFECTIVE PAGES**

**DAQs4Comm – USER MANUAL**

**THE EFFECTIVE PAGES IN THIS PUBLICATION CONSIST OF:**

1 through 24

# REVISIONI

Modifications

|  |  |  |
| --- | --- | --- |
| **§** | **Descrizione Revisione** | **Ed.** |
|  | **Description of Modification** | **Issue** |
|  | **EMISSION** | **-** |

**INDEX**

REVISIONI 3

1. INSTALLATION 5

1.1 ADVANTECH Drivers Installation 5

1.2 ADVANTECH HW Setup 5

1.3 DAQs4Comm SW Installation 5

2. DAQs4COMM FUNCTIONS 8

2.1 Device Configuration 8

2.1.1 Example 1 (2 devices connected): 9

2.2 DQ Read & Write 10

2.2.1 Digital Input Values 11

2.2.2 Integer Values to Write 12

2.2.3 Save Log Data 12

2.2.4 Device Sketch 14

2.2.5 AUTO MODE 15

2.2.5.1 Syntax Rules To Be Followed For Creating a Valid Command .txt File to Load 16

2.2.5.2 BUTTONS 20

2.2.5.2.1 LOAD COMMANDS 20

2.2.5.2.2 EXECUTE 22

2.2.5.2.3 OPEN LOG FILE 22

2.2.5.2.4 STOP 23

# INSTALLATION

The full installation is made in 3 steps:

* ADVANTECH Drivers Installation
* ADVANTECH HW Setup
* DAQs4Comm SW Installation

## ADVANTECH Drivers Installation

First install the Advantech Drivers from the website; check only the Navigator Framework, USB-5800 Driver, USB-5801 Driver, USB-5817 Driver, USB-5820 Driver and all the SDK components.

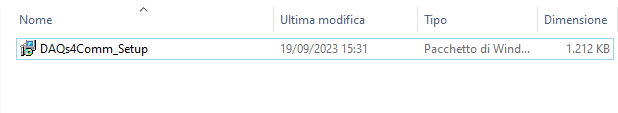
## ADVANTECH HW Setup

To avoid misunderstanding just pay attention on how the connection between all devices should be made.

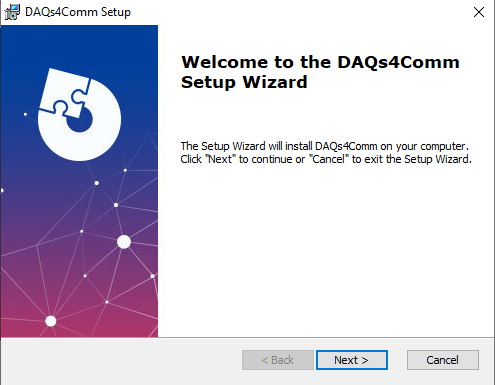
First decide how many devices ADVANTECH USB 5856 you need (no more than 5 devices in cascaded), then connect only the first one (**setted via HW with BID 0**) directly to PC. The second one (**setted via HW with BID 1**) must be connected to the first one via dedicated cable, the third one (**setted via HW with BID 2**) must be connected to the second one in the same way, and so on…

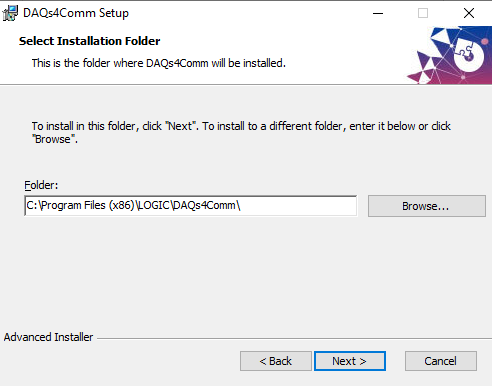
## DAQs4Comm SW Installation

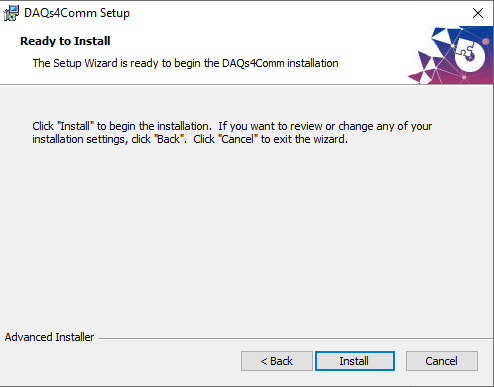
Run the **DAQs4Comm\_Setup** Windows package setup

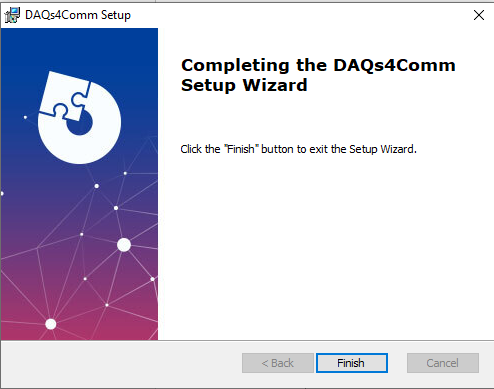


Then proceed to install as shown:









# DAQs4COMM FUNCTIONS

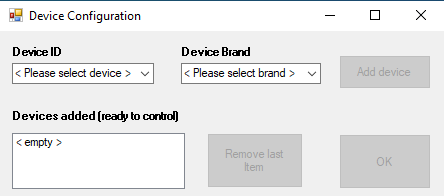
The main application is thought to be opened after the **Device Configuration** is completed.

It is made up by two different interactions with the User, all implemented in two separated windows, as follows:

* **Device Configuration**: allows the user to configure the in-build hardware wants to control
* **DQ Read & Write**: the main application to read and write all the signals from the USB ADVANTECH 5856 in both manual or automatic mode

## Device Configuration

Double click on the **DAQs4Comm.exe** application. The first window that appears is the following:



Select the **Device ID** and **Device Brand (“ADVANTECH”),** then is possible to push **Add device** as shown in the **Example 1**.

Note that there is an insertion control, so:

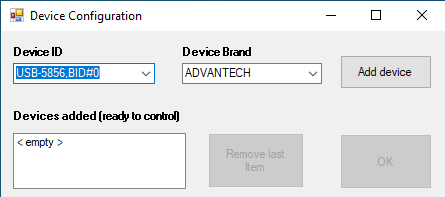
* it is not possible to add new device if both **Device ID** and **Device Brand** are not selected;
* it is not possible to add the same device more than 1 time;
* it is not possible to remove a device if the list of **Devices added** is empty.

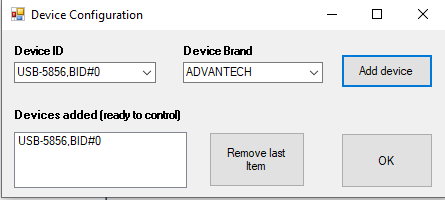
### Example 1 (2 devices connected):

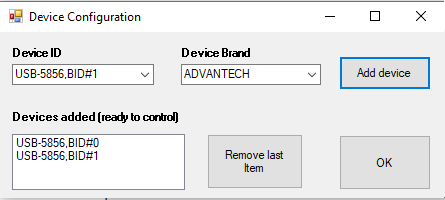
In this example there are **2 devices ADVANTECH USB 5856** connected.

**Remember** (as wrote before in **ADVANTECH HW Setup**) how must be the connections between the devices to avoid inconsistencies.

A suggestion to avoid SW inconsistencies is to add all the devices from **BID 0** (added on the **top – first row** in the list **Devices added**) to **BID N** (the latter inserted on the bottom) as shown below:



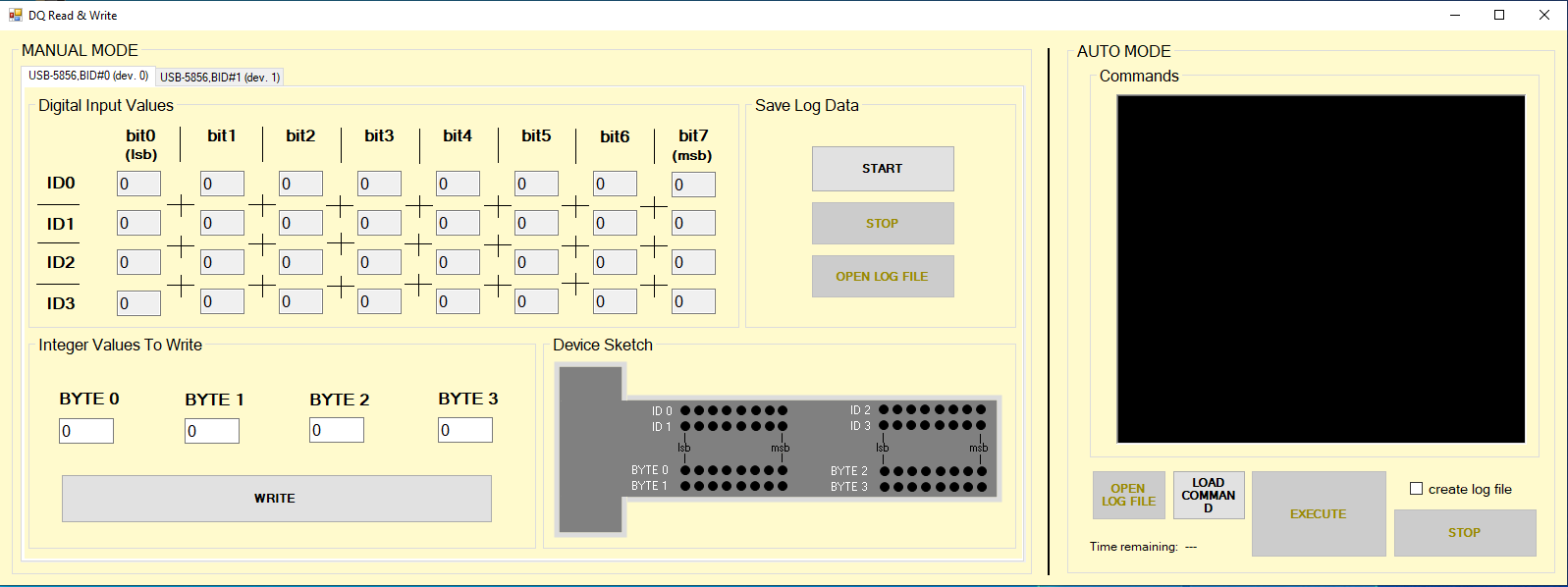




Then push **OK**.

## DQ Read & Write

If all the devices are correctly connected, the display of the Digital Inputs **from bit0 (lsb) to bit7** **(msb)** and **from ID0 to ID3** contains not empty values as shown below:



The core of the application in **MANUAL MODE** is made by four main groups (updated based on the signals received from each device) included different functionalities:

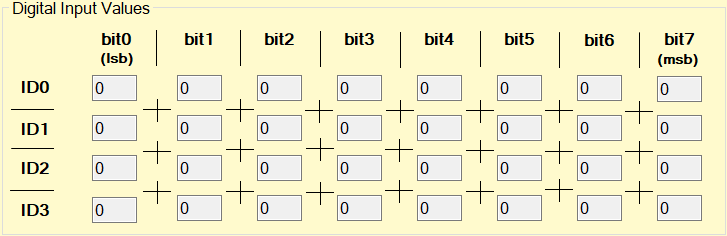
* **Digital Input Values:** display the values of all 32 Digital Input channels (from bit0 to bit7 and from ID0 to ID3);
* **Integer Values to Write:** contains the four int8 data to be written, each one for each byte. There are 4 bytes to be written, from BYTE 0 to BYTE 3;
* **Save Log Data:** includes 3 buttons (START, STOP and OPEN LOG FILE) for acquiring all Digital Input data to create the log file.
* **Device Sketch:** it is a sketch of the device representing what happens in terms of received / written signals. There is one for each tab so for each device connected.

There is also a global functionality, the **AUTO MODE**, from which is possible to set and read commands automatically from all the devices by a list of commands with specific syntax rules.

Note that every millisecond the logic runs only for the tab page (device) selected. So if we are on the tab page of device X, the consequence is that only device X is read until the tab page changes.

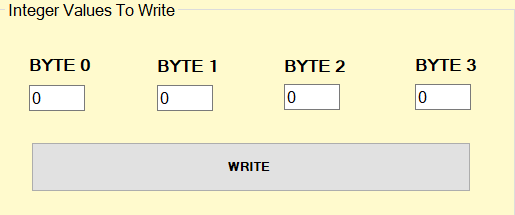
### Digital Input Values

It contains the values for the 32 Digital Input (from bit0 to bit7 and from ID0 to ID3) updated automatically every 1 millisecond since the application is open and only for the specific tab page (device) selected. This functionality is useful when the operator wants to see if the values make sense without necessarily creating a log file.



### Integer Values to Write

It contains the four **int8** values each one correspond to the four byte that can be written on the device of the tab page selected



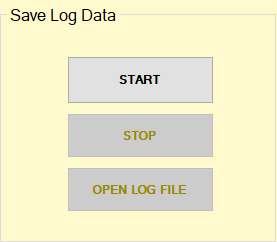
The value of each byte can be modified by changing the **int8** value of the corresponding byte.

Note that there is an insertion control, so it is possible to write only a value from 0 to 255, simply by push **WRITE**, otherwise an error message appears.

### Save Log Data

It contains all buttons needed to **START** and **STOP** log data received from the proper device **ADVANTECH USB 5856** corresponds to the tab page selected. It creates **.csv** file named **LogDI\_\_USB-5856,BID#D,** where “**D**” is the device **ID**.

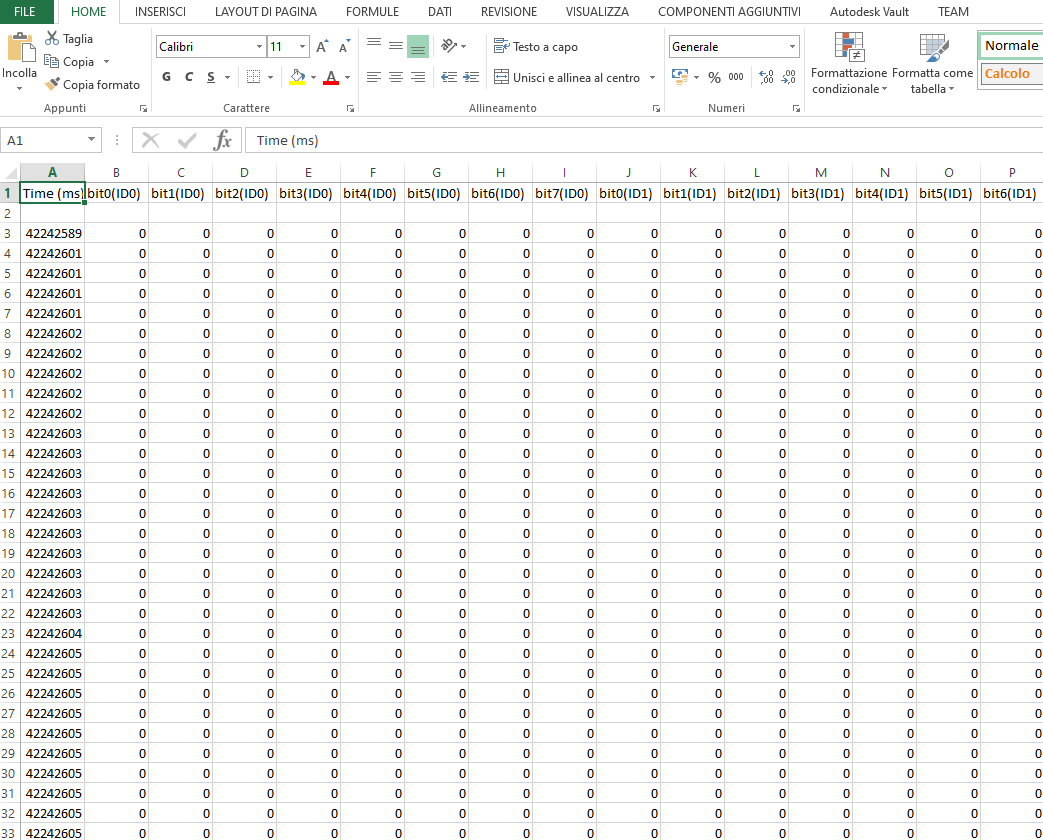
The group is as shown:



Initially, the buttons **STOP** and **OPEN** **Log** **File** are disabled simply because no registration has been made yet.

Press **START** to start logging data then press **STOP** to close the **.csv** log before open it by pressing the button **Open Log File.**

The **LogDI\_\_USB-5856,BID#D.csv** file is saved at the same path of **DAQs4Comm.exe** and should be like that

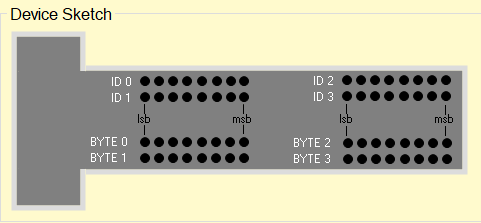


The first column indicates the time in milliseconds taken from the Windows time with System Library only just to have an idea about the recording duration with a tolerance of approximately 15 milliseconds.

The other columns indicates the value of the Digital Inputs from bit0 to bit7 and for each IDs (from ID0 to ID3).

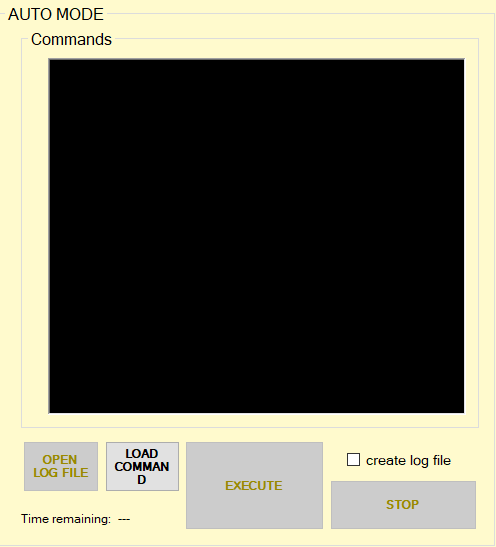
### Device Sketch

It represent the real device input and output configurations. It is the same configuration that should be shown physically in the device of the tab page selected.



### AUTO MODE

The **AUTO MODE** is the group shown below



Initially the situation is as follows:

* the button **OPEN LOG FILE** is disabled simply because no first record (with **create log file** checked) has been made yet.
* The **LOAD COMMAND** button is enabled and, by pushing on it, is possible to load a **.txt** file containing all the commands needs to do automatic tests.
* the **EXECUTE** button is disabled simply because the list of commands is empty or if there is at least one syntax error in the last command file loaded.
* the **STOP** button is disabled because no one execution is running. It is possible to STOP the execution before its completion
* the flag **create log file** that recordsall the commands in AUTO MODEis initially unchecked by default. It is possible to check / uncheck it depending on if the need is to have or not the log file
* The **Time remaining** is “**---‘’** simply because no list of commands is loaded or when it is not possible to have a time estimation based on the code loaded, for example if there is at least one JMP (unconditioned jump) or JE, JNE (conditioned jump) command

#### Syntax Rules To Be Followed For Creating a Valid Command .txt File to Load

There are two main examples in folder “Source Code\command\_examples’’; one of these containing all the instruction’s type will be defined.

An first example of **command (.txt file)** with 2 controlled devices (dev.0 and dev. 1) and with at least one jump instruction is as follows:

0;1 0 0 0;4000

1;16 11 1 4;1000

1;255 255 255 255;3000 :: Accendi pompa

JMP 4

0;0 0 0 0;4000 :: Primo jump

1;255 255 255 255;4000

CMP 0;0 0 0 0 & 1;0 0 0 0

1;14 25 2 255;1000

0;16 11 1 4;4000

JE 15

0;255 255 255 255;3000 :: Accendi motore

0;0 0 0 0;2000 :: comment

1;0 0 0 0;2000

0;255 255 255 255;3000

0;14 25 2 255;1000

1;16 11 1 4;4000 :: Secondo jump

0;255 255 255 255;3000

JNE 2

0;0 0 0 0;1000

1;0 0 0 0;4000

In this case the time remaining does not update because it is at least one jump so the time estimation could not be done.

A second example is without any jump instruction:

0;1 0 0 0;4000

0;16 11 1 4;1000

1;255 255 255 255;3000 :: Accendi pompa

0;0 0 0 0;4000 :: Primo jump

1;255 255 255 255;4000

1;14 25 2 255;1000

0;16 11 1 4;4000

0;255 255 255 255;3000 :: Accendi motore

0;0 0 0 0;2000 :: ff

1;0 0 0 0;2000

0;255 255 255 255;3000

0;14 25 2 255;1000

1;16 11 1 4;4000 :: Secondo jump

0;255 255 255 255;3000

0;0 0 0 0;1000

1;0 0 0 0;4000

In this case the time remaining is updated correctly with 44000 ms.

As it is clear, there are many different type of instructions.

Here below are shown some example, one for each instruction’s type:

1. **0;1 0 0 0;4000 :: comment**

It means: write on dev. 0 the following values for the 4 bytes: BYTE 0=1 BYTE 1=0 BYTE 2=0 BYTE 3=0, then wait 4000 milliseconds. So in this case the syntactic format is:

**device num;BYTE0 BYTE1 BYTE2 BYTE3;wait time :: comment**

Each byte must be an integer number (**int8**) between 0 and 255, otherwise that line will appears with red words.

The wait time indicated in milliseconds must be a positive integer value, otherwise that line will appears with red words.

The device num must be an integer between 0 (dev 0 setted with BID 0 is the only device directly connected to PC via USB) and the number of devices connected to PC subtracted by 1, otherwise that line will appears with red words.

It is possible to add eiter or not the comment; if there is, the separator with the rest of the code row must be realized by “**::**’’

1. **JMP 4 :: comment**

It means: jump unconditionally to the row number 4. Pay attention on that because the enumeration of rows starting from 0!So in this case the syntactic format is:

**JMP row\_num :: comment**

The row number must be a positive integer between 0 and the row number of the last instruction, otherwise that line will appears with red words.

It is possible to add eiter or not the comment; if there is, the separator with the rest of the code row must be realized by “**::**’’

1. **CMP 0;120 15 14 1 & 1;0 6 0 3 ::** **comment**

It means: compares the values read by dev. 0 and dev. 1 with the corresponding 4 bytes values indicated on that row, ordered from ID0 to ID3 for each device considered. If each value read corresponds to the value indicated on that row, the result is **1** (true), otherwise is **0** (false).

In this example, if for dev. 0 the byte ID0 is equal to 120, ID1 is equal to 15, ID2 is 14 and ID3 is 1 **AND** for dev. 1 we have ID0=0, ID1=6, ID2=0 and ID3=3, then the result is **1** (true), otherwise is **0** (false).

Each byte must be an integer number (**int8**) between 0 and 255, otherwise that line will appears with red words.

The device num must be an integer between 0 (dev 0 set with BID 0 is the only device directly connected to PC via USB) and the number of devices connected to PC subtracted by 1, otherwise that line will appears with red words.

It is possible to add eiter or not the comment; if there is, the separator with the rest of the code row must be realized by “**::**’’

It is possible also to add condition for only one device as follows

**CMP device num;ID0 ID1 ID2 ID3 ::** **comment**

In this last case if all the 4 bytes read by dev. ‘’**device num”** are equal to ID0 ID1 ID2 ID3 indicated by the row, then the result is **1** (true), otherwise is **0** (false). Also in this case it is possible to add eiter or not the comment; if there is, the separator with the rest of the code row must be realized by “**::**’’

Pay attention that if there is more than 1 condition, each one of these must be separated by “**&**’’

1. **JE 6 :: comment**

It means: jump to row number 6 only if the last CMP executed gaves result **1** (true). Pay attention on the fact that the enumeration of rows starting from 0!

In this case the syntactic format is:

**JE row\_num :: comment**

The row number must be a positive integer between 0 and the row number of the last instruction, otherwise that line will appears with red words.

It is possible to add eiter or not the comment; if there is, the separator with the rest of the code row must be realized by “**::**’’

1. **JNE 6 :: comment**

It means: jump to row number 6 only if the last CMP executed gaves result **0** (false). Pay attention on the fact that the enumeration of rows starting from 0!

In this case the syntactic format is:

**JNE row\_num :: comment**

The row number must be a positive integer between 0 and the row number of the last instruction, otherwise that line will appears with red words.

It is possible to add eiter or not the comment; if there is, the separator with the rest of the code row must be realized by “**::**’’

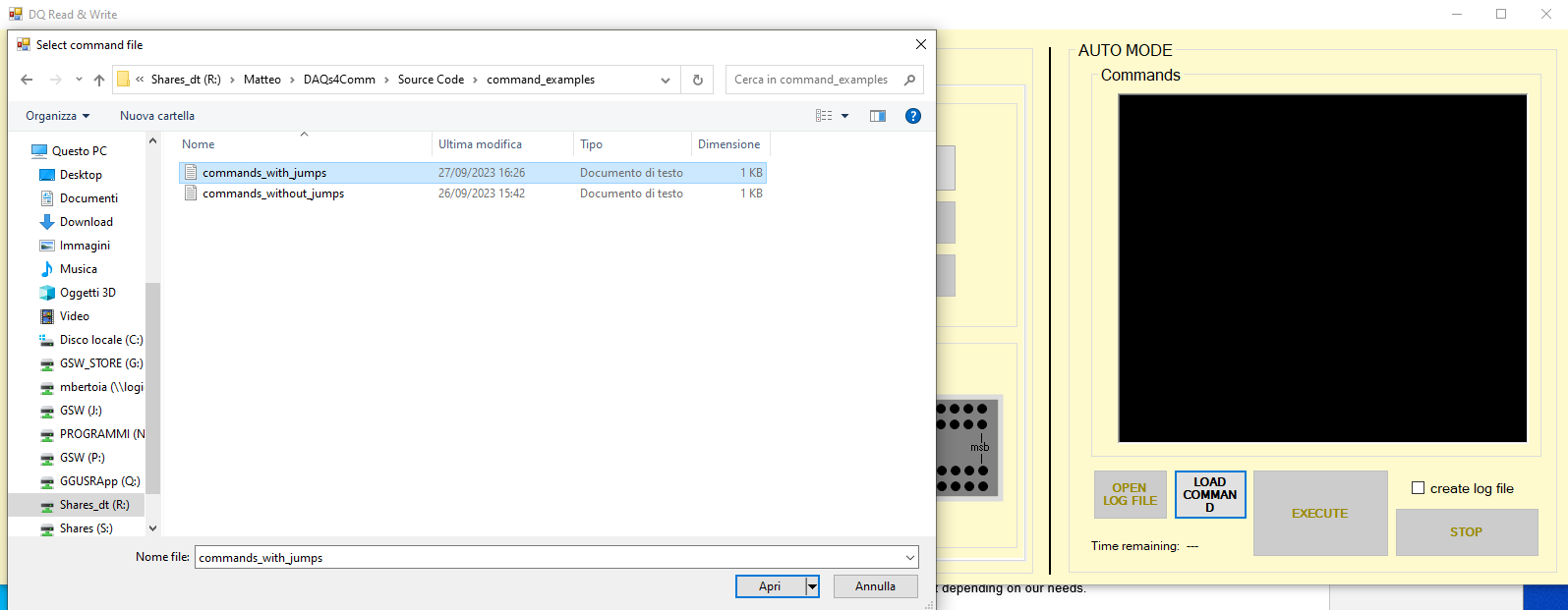
#### BUTTONS

##### LOAD COMMANDS

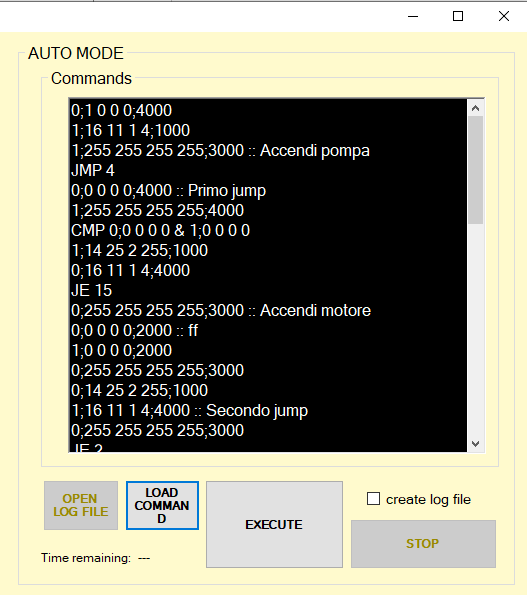
The button **LOAD COMMANDS** allows us to load a **.txt** file containing all the instructions (one instruction per line). Each line **must be** in agreement with the syntax rules indicated in the previous subsection, otherwise the lines appears with red words.

Remember that there are 2 main examples of good code in project folder “Source Code\command\_examples”, both of these respecting all the syntax rules defined previously. In particular one of these, named **commands\_with\_jumps**, contains all the instruction’s type defined in the previous section. A good way to proceed could be to take one of these example commands, then modifying it depending on our needs.

Here below an example **without errors** by load **command\_with\_jump.txt**:

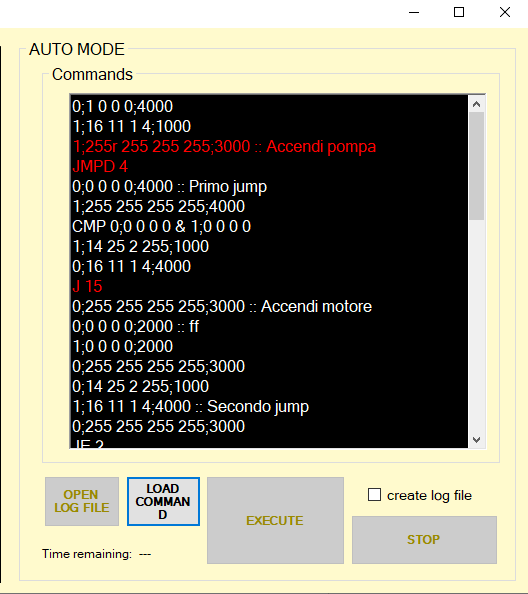


**AFTER PUSHING ON LOAD COMMANDS**

****

**AFTER SELECTED THE COMMAND .txt FILE**

If I try to load the same command file but with JMPD 4 instead of JMP 4 on row number 3, 255r instead of 255 on row number 2, J 15 instead of JE 15 on row number 9, the row should appears with red words as told before. This case is shown in figure below

****

**THE ERRORS DETECTED AT ROW NUMBER 2, 3 AND 9**

##### EXECUTE

The button **EXECUTE** allows us to execute the last code loaded.

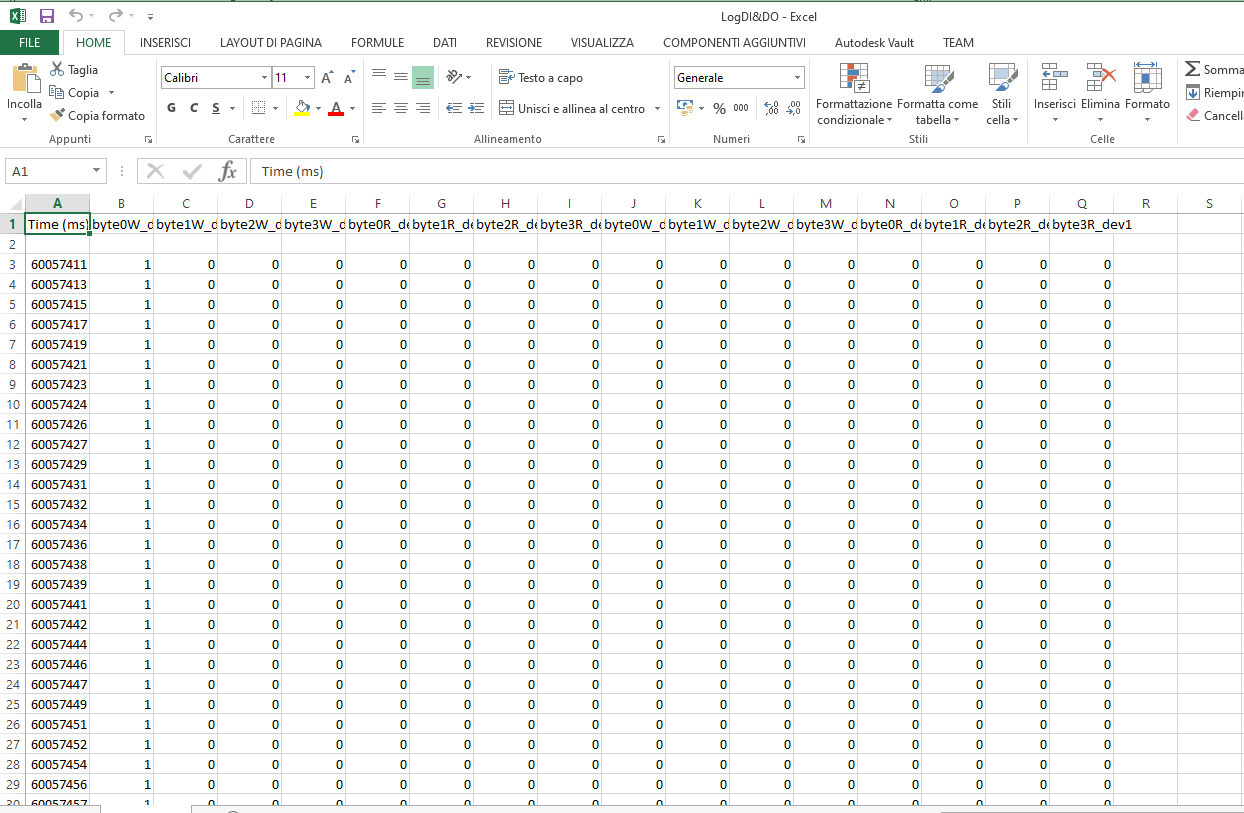
Each row corresponds to one instruction. This button is enabled only if the last code loaded has not red word lines (errors), or if it is not empty, otherwise is disabled.

Remember to check or uncheck **create log file** depending on if we want to record or not all the commands.

##### OPEN LOG FILE

The button **OPEN LOG FILE** is enabled only if at least one record is made with **create log file** checked. Push on it for open automatically the last command’s record named **LogDI&DO.csv**.

An example of the log file record with 2 devices connected is as follows



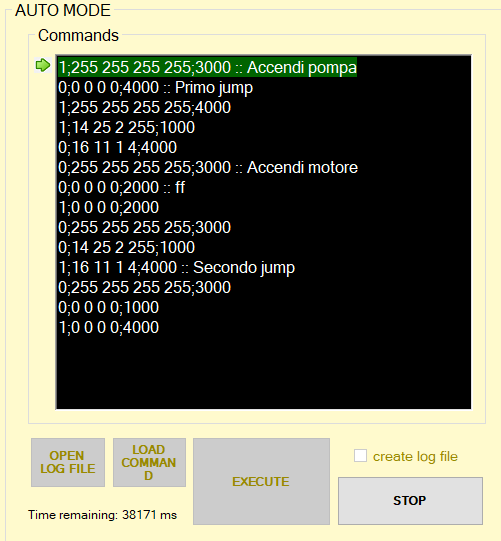
The first column is the computer time captured in milliseconds, while the next columns contains the values of the 4 bytes written followed by the 4 bytes read for each device connected.

##### STOP

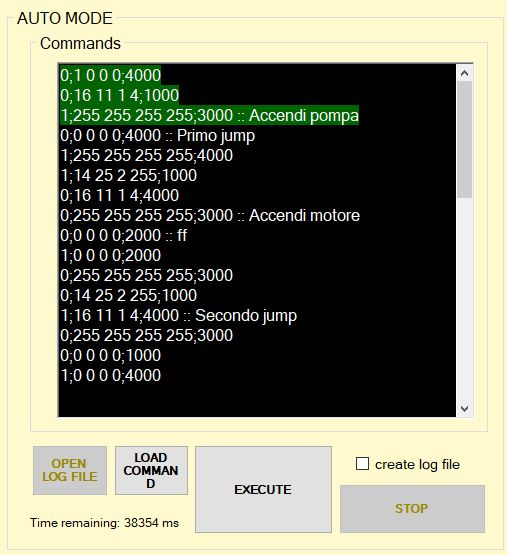
The **STOP** button allows us to finish the execution of the code before its end.

When the user push STOP button, it is immediately known which are the executed code lines until that moment.

In the following figures it is shown the situation a moment before and a moment after pushing the button. Note that when the button STOP is pushed or when the code execution finishes naturally, the last input & output configuration is reported to all the device correctly in MANUAL MODE

****

**Before PUSHING STOP BUTTON**

****

**AFTER PUSHING STOP BUTTON**