

GangPro430 Flash Programmer
for the TI's MSP430Fxx microcontrollers
Remote Control Programming User's Guide

Software version 4.5

PM015A02 Rev.15
April-05-2010

Elprotronic Inc.

Elprotronic Inc.

16 Crossroads Drive
Richmond Hill,
Ontario, L4E-5C9
CANADA

Web site: www.elprotronic.com
E-mail: info@elprotronic.com
Fax: 905-780-2414
Voice: 905-780-5789

Copyright © Elprotronic Inc. All rights reserved.

Disclaimer:

No part of this document may be reproduced without the prior written consent of Elprotronic Inc. The information in this document is subject to change without notice and does not represent a commitment on any part of Elprotronic Inc. While the information contained herein is assumed to be accurate, Elprotronic Inc. assumes no responsibility for any errors or omissions.

In no event shall Elprotronic Inc, its employees or authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claims for lost profits, fees, or expenses of any nature or kind.

The software described in this document is furnished under a licence and may only be used or copied in accordance with the terms of such a licence.

Disclaimer of warranties: You agree that Elprotronic Inc. has made no express warranties to You regarding the software, hardware, firmware and related documentation. The software, hardware, firmware and related documentation being provided to You “AS IS” without warranty or support of any kind. Elprotronic Inc. disclaims all warranties with regard to the software, express or implied, including, without limitation, any implied warranties of fitness for a particular purpose, merchantability, merchantable quality or noninfringement of third-party rights.

Limit of liability: In no event will Elprotronic Inc. be liable to you for any loss of use, interruption of business, or any direct, indirect, special incidental or consequential damages of any kind (including lost profits) regardless of the form of action whether in contract, tort (including negligence), strict product liability or otherwise, even if Elprotronic Inc. has been advised of the possibility of such damages.

END USER LICENSE AGREEMENT

PLEASE READ THIS DOCUMENT CAREFULLY BEFORE USING THE SOFTWARE AND THE ASSOCIATED HARDWARE. ELPROTRONIC INC. AND/OR ITS SUBSIDIARIES (“ELPROTRONIC”) IS WILLING TO LICENSE THE SOFTWARE TO YOU AS AN INDIVIDUAL, THE COMPANY, OR LEGAL ENTITY THAT WILL BE USING THE SOFTWARE (REFERENCED BELOW AS “YOU” OR “YOUR”) ONLY ON THE CONDITION THAT YOU AGREE TO ALL TERMS OF THIS LICENSE AGREEMENT. THIS IS A LEGAL AND ENFORCABLE CONTRACT BETWEEN YOU AND ELPROTRONIC. BY OPENING THIS PACKAGE, BREAKING THE SEAL, CLICKING “I AGREE” BUTTON OR OTHERWISE INDICATING ASSENT ELECTRONICALLY, OR LOADING THE SOFTWARE YOU AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THESE TERMS AND CONDITIONS, CLICK ON THE “I DO NOT AGREE” BUTTON OR OTHERWISE INDICATE REFUSAL, MAKE NO FURTHER USE OF THE FULL PRODUCT AND RETURN IT WITH THE PROOF OF PURCHASE TO THE DEALER FROM WHOM IT WAS ACQUIRED WITHIN THIRTY (30) DAYS OF PURCHASE AND YOUR MONEY WILL BE REFUNDED.

1. License.

The software, firmware and related documentation (collectively the “Product”) is the property of Elprotronic or its licensors and is protected by copyright law. While Elprotronic continues to own the Product, You will have certain rights to use the Product after Your acceptance of this license. This license governs any releases, revisions, or enhancements to the Product that Elprotronic may furnish to You. Your rights and obligations with respect to the use of this Product are as follows:

YOU MAY:

- A. use this Product on many computers;
- B. make one copy of the software for archival purposes, or copy the software onto the hard disk of Your computer and retain the original for archival purposes;
- C. use the software on a network

YOU MAY NOT:

- A. sublicense, reverse engineer, decompile, disassemble, modify, translate, make any attempt to discover the Source Code of the Product; or create derivative works from the Product;
- B. redistribute, in whole or in part, any part of the software component of this Product;

C. use this software with a programming adapter (hardware) that is not a product of Elprotronic Inc.

2. Copyright

All rights, title, and copyrights in and to the Product and any copies of the Product are owned by Elprotronic. The Product is protected by copyright laws and international treaty provisions. Therefore, you must treat the Product like any other copyrighted material.

3. Limitation of liability.

In no event shall Elprotronic be liable to you for any loss of use, interruption of business, or any direct, indirect, special, incidental or consequential damages of any kind (including lost profits) regardless of the form of action whether in contract, tort (including negligence), strict product liability or otherwise, even if Elprotronic has been advised of the possibility of such damages.

4. DISCLAIMER OF WARRANTIES.

You agree that Elprotronic has made no express warranties to You regarding the software, hardware, firmware and related documentation. The software, hardware, firmware and related documentation being provided to You “AS IS” without warranty or support of any kind. Elprotronic disclaims all warranties with regard to the software and hardware, express or implied, including, without limitation, any implied warranties of fitness for a particular purpose, merchantability, merchantable quality or noninfringement of third-party rights.



*This device complies with Part 15 of the FCC Rules. Operation is subject to the following two conditions:
(1) this device may not cause harmful interference and
(2) this device must accept any interference received, including interference that may cause undesired operation.*

NOTE: *This equipment has been tested and found to comply with the limits for a Class B digital devices, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one of more of the following measures:*

- * Reorient or relocate the receiving antenna*
- * Increase the separation between the equipment and receiver*
- * Connect the equipment into an outlet on a circuit different from that to which the receiver is connected*
- * Consult the dealer or an experienced radio/TV technician for help.*

Warning: *Changes or modifications not expressly approved by Elprotronic Inc. could void the user's authority to operate the equipment.*



This Class B digital apparatus meets all requirements of the Canadian Interference-Causing Equipment Regulations.

Cet appereil numerique de la classe B respecte toutes les exigences du Reglement sur le material brouilleur du Canada.

Table of Contents

1. Introduction	9
2. Getting Started	16
2.1 Self Test Program	16
2.2 MyGP430Prg Projects	18
2.3 API DLL Demo Programs	22
3. Example with FPA API DLL	30
3.1 Example with Single FPA API DLL	30
3.2 Example with Multi-FPA API DLL	31
4. List of the DLL instructions	35
4.1 Multi-FPA instructions	42
F_Trace_ON	42
F_Trace_OFF	42
F_OpenInstances	42
F_CloseInstances	43
F_OpenInstancesAndFPAs, F_OpenInstances_AndFPAs	43
F_Set_FPA_index	47
F_Get_FPA_index	48
F_Check_FPA_index	48
F_Disable_FPA_index	49
F_Enable_FPA_index	49
F_LastStatus	50
F_Multi_DLLTypeVer	50
F_Get_FPA_SN	51
4.2 Generic instructions	52
F_Check_FPA_access	52
F_Initialization	54

F_API_DLL_Directory	55
F_Close_All	55
F_GetSetup	56
F_ConfigSetup	56
F_SetConfig	69
F_GetConfig	69
F_Set_MCU_Name	70
F_Get_Device_Info	70
F_DispSetup	73
F_ReportMessage, F_Report_Message	74
F_GetReportMessageChar	75
F_DLLTypeVer	76
F_ConfigFileLoad, F_Config_FileLoad	77
F_Power_Target	79
F_Reset_Target	79
F_Get_Targets_Result	80
F_Get_Active_Targets_Mask	81
F_Get_Targets_Vcc	82
F_Set_fpa_io_state	82
4.3 Data Buffers access instructions	83
F_ReadCodeFile, F_Read_CodeFile	83
F_Get_CodeCS	85
F_ReadPasswFile, F_Read_PasswFile	85
F_Clr_Code_Buffer	87
F_Put_Byte_to_Code_Buffer	87
F_Get_Byte_from_Code_Buffer	88
F_Put_Byte_to_Password_Buffer	89
F_Get_Byte_from_Password_Buffer	89
F_Put_Byte_to_Gang_Buffer	90
F_Get_Byte_from_Gang_Buffer	91
4.4 Encapsulated instructions	92
F_AutoProgram	92
F_VerifyFuseOrPassword	94
F_Memory_Erase	94
F_Memory_Blank_Check	95
F_Memory_Write	95

F_Memory_Verify	95
F_Gang_Flash_Read	96
4.4 Sequential instructions	98
F_Open_Target_Device	99
F_Close_Target_Device	100
F_Segment_Erase	101
F_Sectors_Blank_Check	102
F_Write_Word	102
F_Write_Byte	103
F_Copy_Buffer_to_Flash	104
F_Copy_Gang_Buffer_to_Flash	104
F_Copy_Buffer_to_RAM	106
F_Copy_Gang_Buffer_to_RAM	107
F_Copy_RAM_to_Gang_Buffer	108
F_Copy_Flash_to_Gang_Buffer	109
F_Set_PC_and_RUN	110
F_Capture_PC_Addr	112
F_Synch_CPU_JTAG	112
F_Blow_Fuse	113
F_Adj_DCO_Frequency	115
F_Get_DCO_constant	115
F_Test_DCO_Frequency	116
F_Set_DCO_constant	116
F_Get_DCO_Freq_result	117
 <i>Appendix A</i>	 118
GangPro430 Command Line interpreter	118

1. Introduction

GangPro430 Flash Programmer (USB) can be remotely controlled from other software applications (Visual C++, Visual Basic etc.) via a DLL library. The Multi-FPA - allows to remotely control simultaneously up to sixteen Flash Programming Adapters (FPAs) significantly reducing programming speed in production.

Figure 1.1 shows the connections between PC and up to sixteen programming adapters. The FPAs can be connected to PC USB ports directly or via USB-HUB. Direct connection to the PC is faster but if the PC does not have required number of USB ports, then USB-HUB can be used. The USB-HUB should be fast, otherwise speed degradation can be noticed. When the USB hub is used, then the D-Link's Model No: **DUB-H7, P/N BDUBH7..A2** USB 2.0 HUB is recommended.

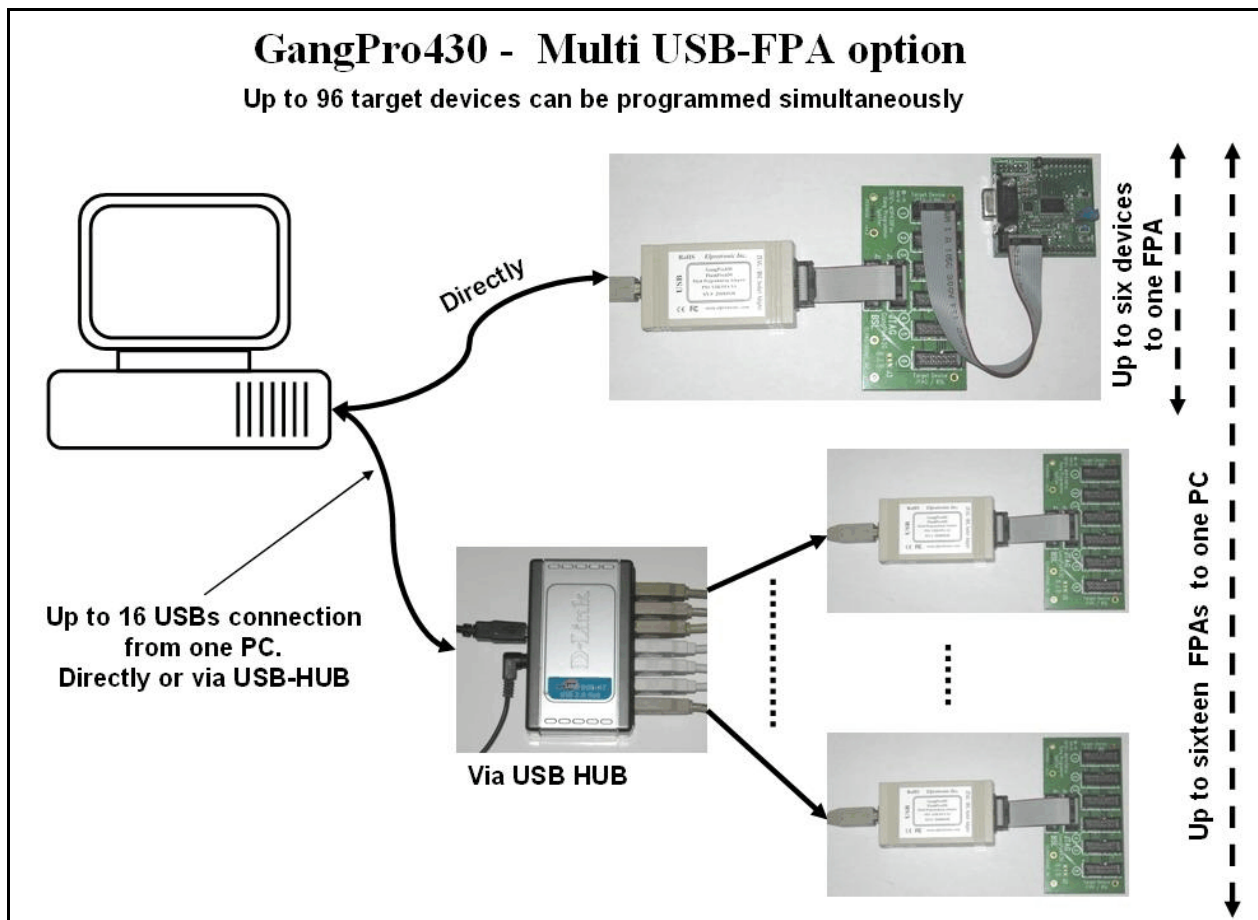


Figure 1.1

Target devices can be connected to Gang Programmer via Gang Splitter directly to JTAG connectors (Figure 1.1), or to the JTAG test points created on the PCB panel via nails (Figure 1.2). The second method allows to easy and fast program a whole panel with up to 96 target devices on the panel before depanelization.

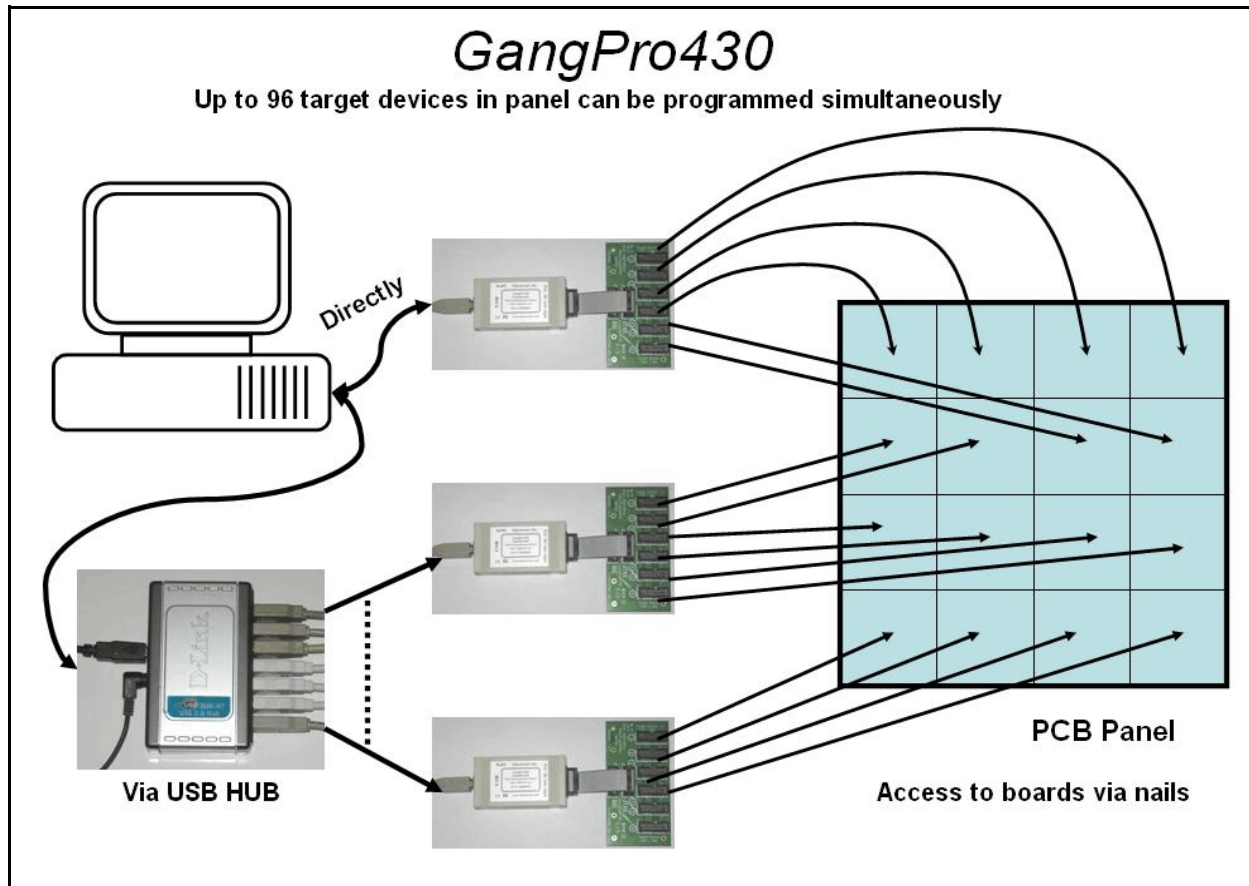


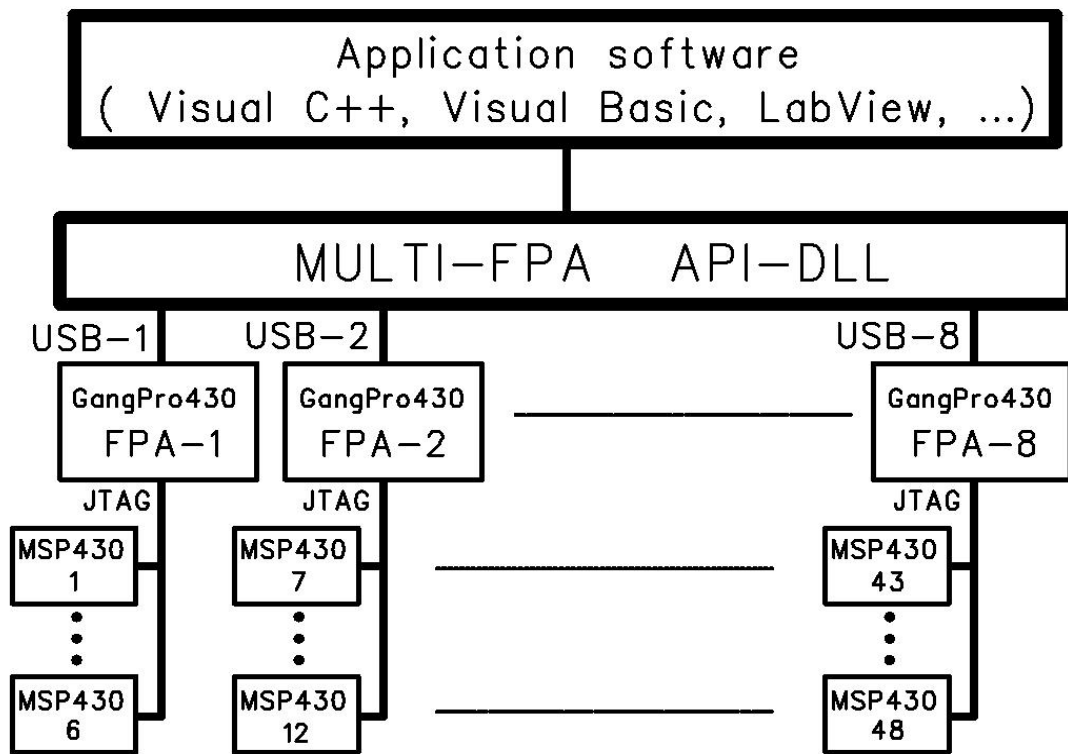
Figure 1.2

Block diagram of the Multi-FPA application DLL using eight FPAs is presented on the Figure 1.3.

To support this new Multi-FPA API-DLL feature, the software package contains seventeen dll files

- the Multi-FPA API-DLL selector
- sixteen standard single FPAs API-DLLs

Figure 1.4 shows the logical connections between these dll files.



Up to 48 Target Devices programmed simultaneously
with MULTI-FPA API-DLL

Figure 1.3

The main DLL file (GangPro430FPA.DLL) has the same name and structure like the old version of the single FPA DLL and is fully back compatible with the old API-DLL. This allows to use the new Multi-FPA API-DLL software package with the old application software written by customers - without software modification. In this case only one FPA will be used. All other FPAs becomes inactive. Only minor software modification is required to activate more than one FPA (up to sixteen) from application software.

The main Multi-FPA file (FPA-selector) allows to transfer API-DLL functions coming from application software to desired single application dll (GangPro430FPA1.DLL to GangPro430FPA16.DLL).

Note: Software package contains one USB-FPA DLL file - the GangPro430FPA1.DLL. Files GangPro430FPA2.DLL to GangPro430FPA16.DLL will be copied automatically if required.

Multi USB-MSP430-FPA

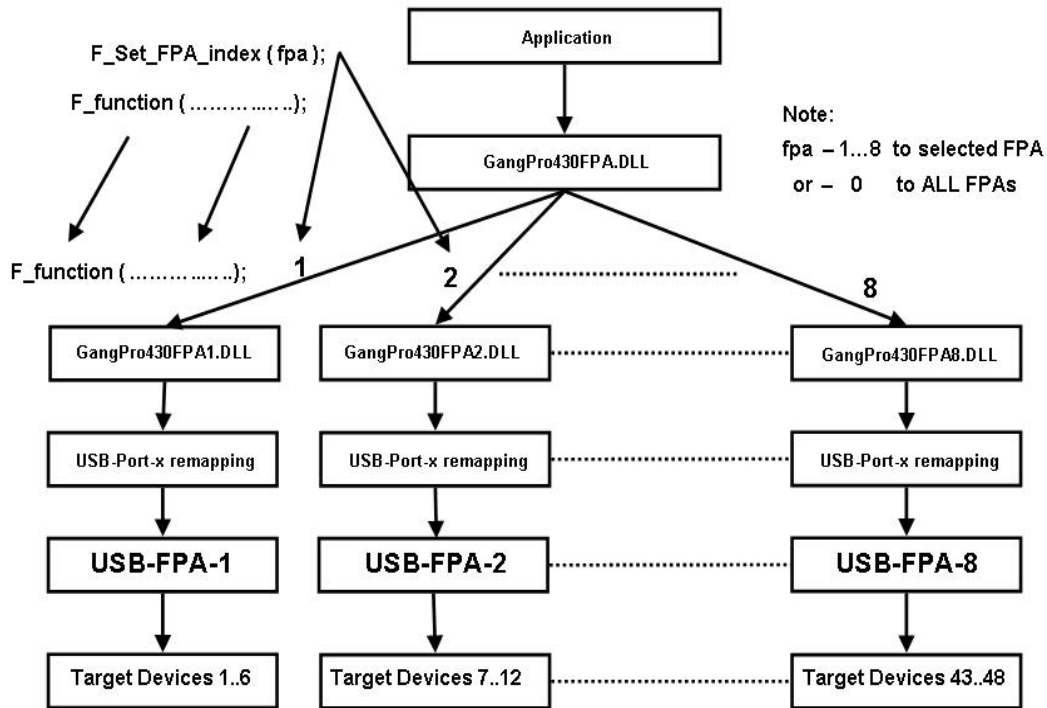


Figure 1.4

The GangPro430FPA.DLL file is transparent for all API-DLL functions implemented in the single FPA API-DLLs, or otherwise, to all old single APL DLLs functions. Desired destination FPA can be selected using the new function added to the Multi-FPA

```
F_Set_FPA_index( fpa );
```

where the

fpa = 1 to 64 when only one desired FPA required to be selected

or **fpa** = 0 when ALL active FPAs should be selected.

The selected FPA index modified by the `F_Set_FPA_index(fpa)` instruction can be modified at any time. By default, the FPA index is 1 and if only one FPA is used then **fpa** index does not need to be initialized or modified. When the **fpa** index 1 to 64 is used, then the result is coming back to application software from the single API-DLL via transparent Multi-FPA dll. When **fpa** index is 0 (ALL-FPAs) and results are the same from all FPAs, then the same result is passing back to

application software. If results are not the same, then the Multi-FPA dll is returning back value -1 (minus 1) and all recently received results can be read individually using function

F_LastStatus(fpa)

Most of the implemented functions allows to use the determined fpa index 1 to 64 or 0 (ALL-FPAs). When functions return specific value back, like read data etc, then only determined FPA index can be used (fpa index from 1 to 64). When the fpa index is 0 (ALL-FPAs) then almost all functions are executed simultaneously. Less critical functions are executed sequentially from FPA-1 up to FPA-64 but that process can not be seen from the application software.

When the inactive fpa index is selected, then return value from selected function is -2 (minus 2). When all fpa has been selected (fpa index = 0) then only active FPAs will be serviced. For example if only one FPA is active and fpa index=0, then only one FPA will be used. It is save to prepare the universal application software that allows to remote control up to sixteen FPAs and on the startup activate only desired number of FPAs.

It should be noticed, that all single API-DLLs used with the Multi-FPA DLL are fully independent to each other. From that point of view it is not required that transferred data to one FPA should be the same as the transferred data to the others FPAs. For example code data downloaded to FPA-1 can be different that the code data downloaded to the FPA-2, FPA-3 etc. But even in this case the programming process can be done simultaneously. In this case the desired code should be read from the code file and saved in the API-DLL-1, next code file data should be saved in the API-DLL-2 etc. When it is done, then the F_AutoProgram can be executed simultaneously with selected all active FPAs. All FPAs will be serviced by his own API-DLL and data packages saved in these dlls.

The following commands are supported in the DLL library:

- Initialization and termination communication with the programming adapter,
- Programmer configuration setup,
- Programming report message,
- Code data and password data read from the file,
- DC power target from the programming adapter,
- Reset target device,
- Auto program target device (erase, blank check, program and verify),
- Password or fuse verification,
- All or selected part of memory erase,

All or selected part of memory blank check,
 All or selected part of memory write,
 All or selected part of memory verify,
 All or selected part of memory read,
 Open or close communication with the target device,
 Selected memory segment erase,
 Selected part of memory blank check,
 Selected part of memory segment write,
 Selected part of memory segment read,
 Security fuse blow.

GangPro430 Flash Programmer software package contains all required files to remotely control programmer from a software application. When software package is installed then by default the DLL file, library file and header file are located in:

for the USB version programmer:

C:\Program Files\Elprotronic\MSP430\USB GangPro430\API-DLL

GangPro430FPA.dll	- Multi-FPA selection/distribution DLL
GangPro430FPA1.dll	- USB-FPA DLL
GangPro430-Dll.h	- header file for C++
GangPro430-Dos-Dll.h	- header file for C++ (Borland) or DOS
GangPro430FPA.lib	- lib file for C++
GangPro430FPA-BC.lib	- lib file for C++ (Borland)
config.ini	- default configuration file for the FPAs
FPAs-setup.ini	- FPAs- vs USB ports configuration file

The GangPro430FPA.dll contains two groups of the same functions used in C++ application and Visual Basic applications. All procedure names used in Visual Basic are starting from **VB_XXXX**, when procedure names used in C++ are starting from **F_XXXX**. All functions starting from **F_XXXX** using the **_Cdecl** declarations used in C++ . Function names starting from **VB_XXXX** has the **_stdcall** calling declaration required in Visual Basic.

Reminding files listed above are required in run time - to initialize the flash programming adapter (config.ini) and USB setup (FPAs-setup.ini).

When C++ application is created, then following files should be copied to the source application directory:

GangPro430-Dll.h - header file for C++
GangPro430FPA.lib - lib file for C++

and to the release/debug application directory

GangPro430FPA.dll - Multi-FPA selection/distribution DLL
GangPro430FPA1.dll - USB-FPA DLL
config.ini - default configuration file for the FPAs
FPAs-setup.ini - FPAs- vs USB ports configuration file

Executable application software package in C++ the requires following files

GangPro430FPA.dll - Multi-FPA selection/distribution DLL
GangPro430FPA1.dll - USB-FPA DLL
config.ini - default configuration file for the FPAs
FPAs-setup.ini - FPAs- vs USB ports configuration file

When application in Visual Basic is created, then following files should be copied to the source or executable application directory:

GangPro430FPA.dll - Multi-FPA selection/distribution DLL
GangPro430FPA1.dll - USB-FPA-1 DLL
config.ini - default configuration file for the FPAs
FPAs-setup.ini - FPAs- vs USB ports configuration file

All these files 'as is' should be copied to destination location, where application software using DLL library of the MSP430 Flash programmer is installed.

Config.ini file has default setup information. Config.ini file can be modified and taken directly from the GangPro430 Flash Programmer (GUI) application software. To create required config.ini file the standard GangPro430 Flash programmer software should be open and required setup (memory option, JTAG/SBW communication speed etc) should be created. When this is done, programming software should be closed and the config.ini file with the latest saved configuration copied to destination location. Note, that the configuration setup can be modified using DLL library function.

Software package has a demo software written under Visual C++.net (7.0 - 2002), Visual Basic.net (7.1 - 2003) and LabVIEW (7.1). All files and source code are located in:

C:\Program Files\Elprotronic\MSP430\USB GangPro430\API-DLL-Demo\Cpp
C:\Program Files\Elprotronic\MSP430\USB GangPro430\API-DLL-Demo\VBnet
C:\Program Files\Elprotronic\MSP430\USB GangPro430\LabVIEW

2. Getting Started

2.1 Self Test Program

The software package contains the GangPro430 Self Test program, that allows to test functionality of the flash programming adapter, users target device and connections between these units. Software package use the Multi-FPA API-DLLs. In the test results printout are listed the DLL functions with syntax, that has been used. This printout is useful to find-out source of the problems, as well as can be used at the startup when your application software uses one programming adapter only. Software can be activated from the Start menu

Start -> Programs -> Elprotronic-Flash Programmers -> (MSP430) GangPro430 -> GangPro430 Self Test
or by running the program **GangPro430SelfTest.exe** from the location

C:\Program Files\Elprotronic\MSP430\USB GangPro430\SelfTest

The Figure 2.1 presents the GUI of the gangPro430 Self Test.

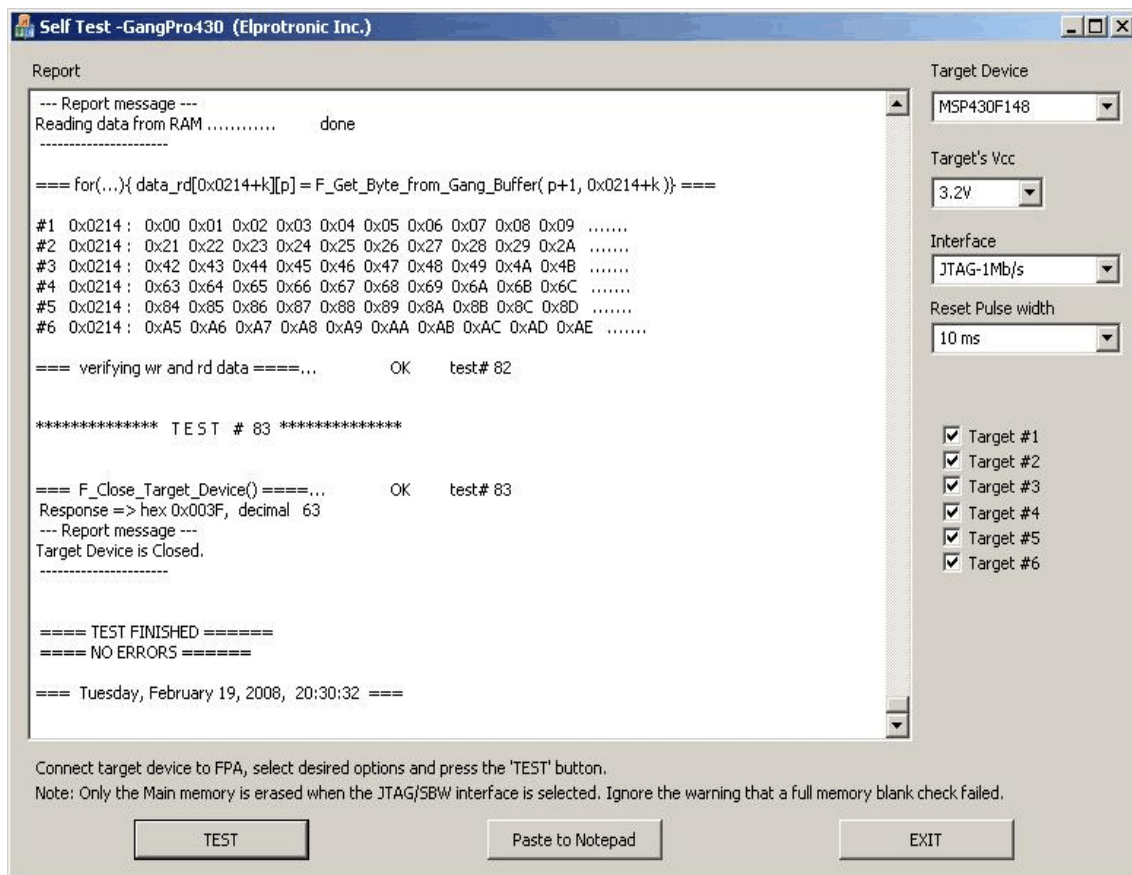


Figure 2.1

Connect the Flash Programming Adapter (FPA) to PC (USB port), connect your target devices to FPA, select desired options in following selectors (see Figure 2.1) - “**Target Device**”, “**Target’s Vcc**” “**Interface**” and “**Reset Pulse width**” and active **Target #** numbers. When it is done then press the button “**TEST**”. When test is finished, then check if there is no any errors. Detailed test report is displayed. The test report can be paste to Notepad and saved if required.

Following conditions are used during the test:

1. **JTAG** and **Spy-Bi-Wire** interface is used:

- * Erased and programmed MAIN memory only. The info memory (0x1000 to 0x10FF) is not erased and not modified. The DCO calibration data in the F2xx are not modified. During the test it can be displayed warning that All memory blank check failed, that of course is normal. But selected memory blank check must be OK (the full MAIN memory in this case).
- * All bytes of the main memory are erased, blank checked and programmed with the randomly generated data used as a code data. Whole MAIN memory content is verified (check sum) and also read whole data and verified byte by byte.
- * One sector (location 0xFC00 to 0xFDFF) is erased and blank checked. Also contents of the two closer sectors are verified if there are not erased. Small block of data are saved and verified in the mentioned sector.
- * Word write/read to TACCR0 (0x172) register.
- * Byte/Word manipulation are used in the part of the RAM.

2. **BSL** interface is used:

- * Due to unknown access password, the whole Flash memory - MAIN and INFO are erased. In the F2xx microcontrollers the DCO calibration data will be erased. There is no way to save the DCO data if the BSL password is unknown. The DCO data can be calibrated using the GangPro430 GUI package software when the JTAG or Spy-Bi-Wire access is available (when the **JTAG** fuse is not blown). See the GangPro430 manual for details.
- * All MAIN memory is tested in the same way as it is used with the **JTAG/Spy-Bi-Wire** interface
- * Word write/read to TACCR0 (0x172) register.
- * Access to RAM if size of the RAM if higher then 256 bytes. Access to RAM space 0x200 to 0x2FF is blocked due to stack and firmware located in this RAM location.

Note: The first test (Vcc value when the power is OFF) can be failed, if the external power is connected or if the blocking capacitor on your target device connected to the Vcc line if high.

The Vcc should be below 0.4V when the power is OFF, tested 2 seconds after switching-off the power from FPA, otherwise test failed.

The Self Test programming software package is located in directory

C:\Program Files\Elprotronic\MSP430\USB GangPro430\SelfTest

and contains following files

GangPro430FPA.dll	- Multi-FPA selection/distribution DLL
GangPro430FPA1.dll	- USB-FPA DLL
config.ini	- default configuration file for the FPAs
GangPro430SelfTest.exe	- executable file

To run the executable file ***GangPro430SelfTest.exe*** in the other location the files listed above should be copied “as is” to destination directory.

2.2 MyGP430Prg Projects

The ***MyGP430Prg*** (*My Gang Pro 430 Programmer*) projects are examples of using the Multi-FPA API-DLL with Microsoft Visual Studio 7.0 (2002) and for Microsoft Visual Basic 6.0. They are intended to help users create their own application that uses the API-DLL by providing a simple starting point. When using Visual Studio C++ include the following files should be included to your program:

GangPro430FPA.lib
GangPro430-Dll.h
GP430FPA-Lib.h
GP430FPA-Lib.cpp
GP430SamplePrg.h
GP430SamplePrg.cpp

The above files are located in the following directory:

...\Elprotronic\MSP430\USB GangPro430\API-DLL-MyPrg\Cpp\scr

To run your application you will need to allow your application access to the Multi-FPA dynamically linked library. A simple way to do this is to copy the following files into your directory where executable file is located:

GangPro430FPA.dll
GangPro430FPA 1.dll
Config.ini (optional)

You can modify files GP430SamplePrg.cpp and GP430SamplePrg.h in a way that suits your application. However, the remaining files should not be modified. These files can be found in directory

...\\Elprotronic\MSP430\USB GangPro430\API-DLL-MyPrg\Cpp\MyGP430Prg

and are included for demonstration purposes only. The sample project can be opened by selecting the project file **MyGP430Prg.vcproj** located in directory

...\\Elprotronic\MSP430\USB GangPro430\API-DLL-MyPrg\Cpp\MyGP430Prg

The following dialogue box will be displayed when project executed (see figure 2.2).

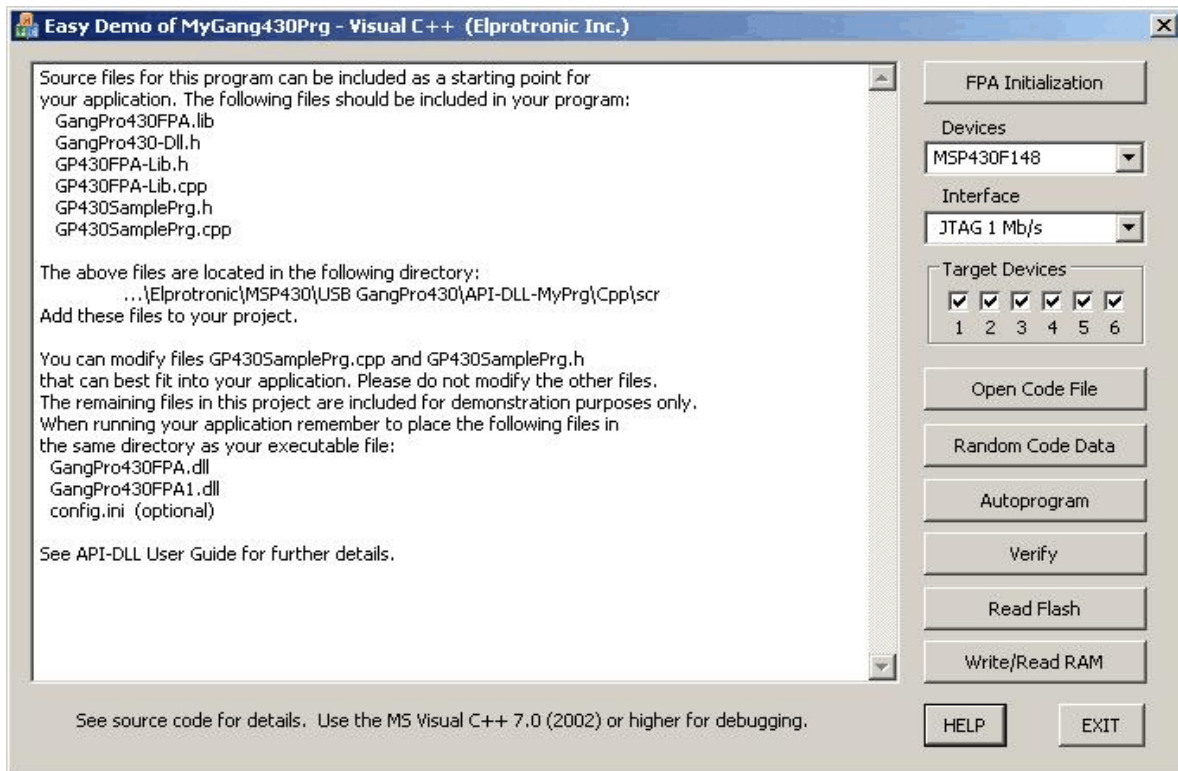


Figure 2.2

Dialogue box contains few buttons, that call procedures listed in the mentioned above files. See contents in the **MyGP430Prg.cpp** file located in the project directory, how these procedures are called from application software. There are several useful procedures located in the **GP430FPA-Lib.cpp** file that significantly simplify the FPA initialization process. See comments for each procedures located in this file.

The first procedure named

get_FPA_and_MSP_list(.....)

searches all FPAs connected to your PC via USB ports. As the results, adapter serial numbers of the detected FPAs are located in the FPA_SN_list[k] where k = 0 up to 15. Up to sixteen FPA SN can be located in this data block. SN list are located starting from FPA_SN_list[0]. The same procedure also takes a list of supported MSP microcontrollers containing MCU name, flash start and end addresses etc. from API-DLL. The MCU list is saved in the following structure

```
typedef struct
{
    char name[DEVICE_NAME_SIZE];
    int index;
    long flash_start_addr;
    long flash_end_addr;
    long info_flash_start_addr;
    long info_segm_size;
    long no_of_info_segm;
    long RAM_size;
} DEVICELIST;
DEVICELIST DeviceList[300];
```

Up to 300 MCUs can be saved in DeviceList. When required, the size of this data block can be increased in the future. Currently, device list contains about 120 MCUs. The MCU names in the ***DeviceList*** are sorted in alphabetic order. Alphabetical order is convenient for users, however the API-DLL requires fixed MCU index when selecting the particular MCU. In the structure above the MCU index required by API-DLL is located in

DeviceList[k].index

and procedure setting the required MCU becomes as follows

```
F_SetConfig( CFG_MICROCONTROLLER, DeviceList[k].index );
```

The second procedure that can be called after the *get_FPA_and_MSP_list(.....)* procedure has finished successfully, is the *AssignFPAs(.....)* procedure that activates the DLLs and assign desired FPAs. When these two procedures are finished successfully, the programmer is ready to work. See procedure

GP430_FPA_initialization()

located in *MyGP430SamplePrg.cpp* file how to call procedures above and what the next step should be.

The same procedures as described above have been implemented in the software package using Visual Basic 6.0. When the Visual Basic 6.0 is used, then the following files should be included to your program

```
GangPro430Def.bas  
GP430FPA-Lib.bas  
GP430SamplePrg.bas
```

The above files are located in the following directory

...\\Elprotronic\\MSP430\\USB GangPro430\\API-DLL-MyPrg\\VB6

When running your application, remember to place the following files in the same directory as your executable file:

```
GangPro430FPA.dll  
GangPro430FPA 1.dll  
Config.ini (optional)
```

You can modify file *GP430SamplePrg.bas* to best fit into your application needs. Other files should not be modified. The remaining files in this project are located in directory

...\\Elprotronic\\MSP430\\USB GangPro430\\API-DLL-MyPrg\\VB6

and are included for demonstration purposes only. Project can be activated by selecting the project file MyGP430Prg.vbp located in directory

...\\Elprotronic\MSP430\USB GangPro430\API-DLL-MyPrg\VB6

The following dialogue box will be displayed when the project will be executed (see figure 2.3).

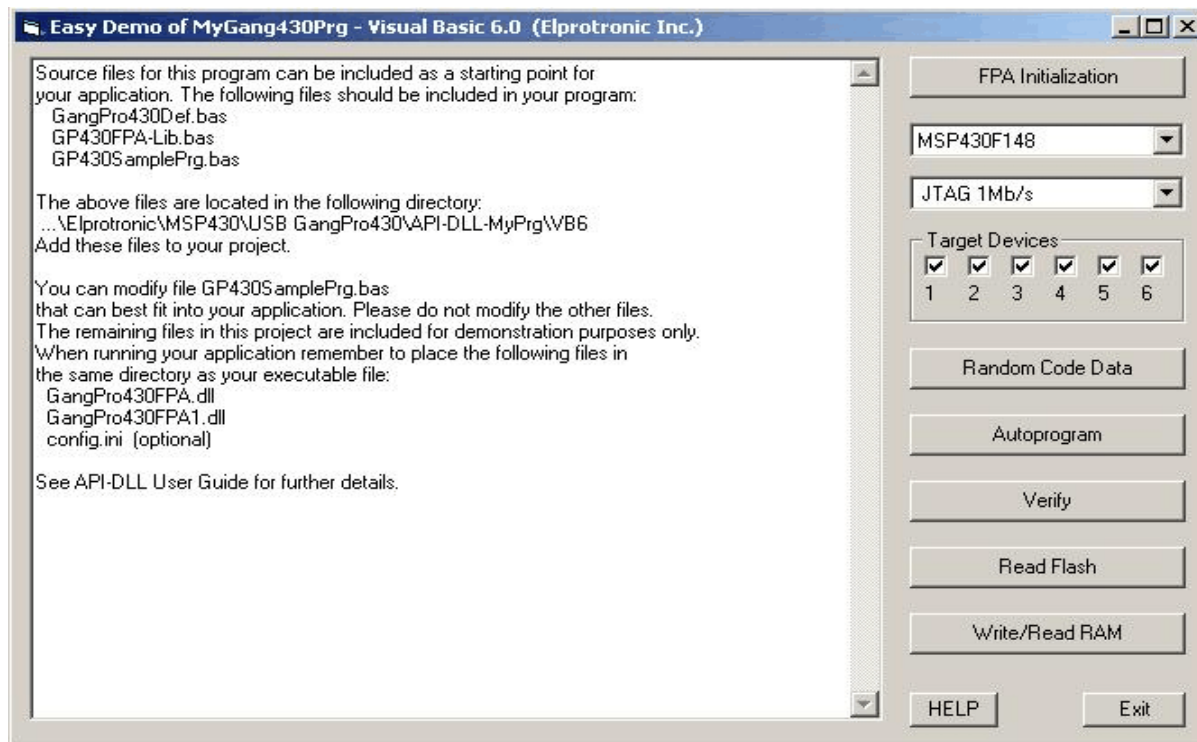


Figure 2.3

All procedures implemented in Visual Basic 6.0 are the same as those implemented in Visual C++. See description above. Procedures written in VB6 are located in the GP430FPA-Lib.bas file. Example how to use these procedures are located in the GP430SamplePrg.bas file. API-DLL function declaration and constant definition are located in the GangPro430Def.bas file.

2.3 API DLL Demo Programs

The first demo program is small GUI program with a lot of buttons allowing to separately call functions using DLL library package software. Source code and all related project files are located in the following directory:

C++ version

C:\Program Files\Elprotronic\USB GangPro430\API-DLL-Demo\Cpp\Demo-1\Exe

Program can be activated by selecting the GangPro430DLL-Demo-Cpp.exe. Demo program can also be activated from the windows menu:

Start->Program->Elprotronic MSP430 Flash Programmer->USB GangPro430->API-DLL-Demo-Cpp
When demo program is activated, then dialogue screen appears (see figure 2.1). When the one FPA is used then the button OpenInstancesAndFPAs(“*# *”) should be pressed. When this button is pressed, then the DLL function named

```
F_OpenInstancesAndFPAs ( "*# *" );
```

is executed. The first adapter connected to PC will be assigned to programming software.

When the particular serial number should be detected, if more then one adapters are connected to PC, then the FPA-Setup.ini file should be created with the listed desired FPA SN. Using the *Notepad* editor open the default FPA’s configuration file ‘*FPAs-setup.ini*’ taken from the Elprotronic’s

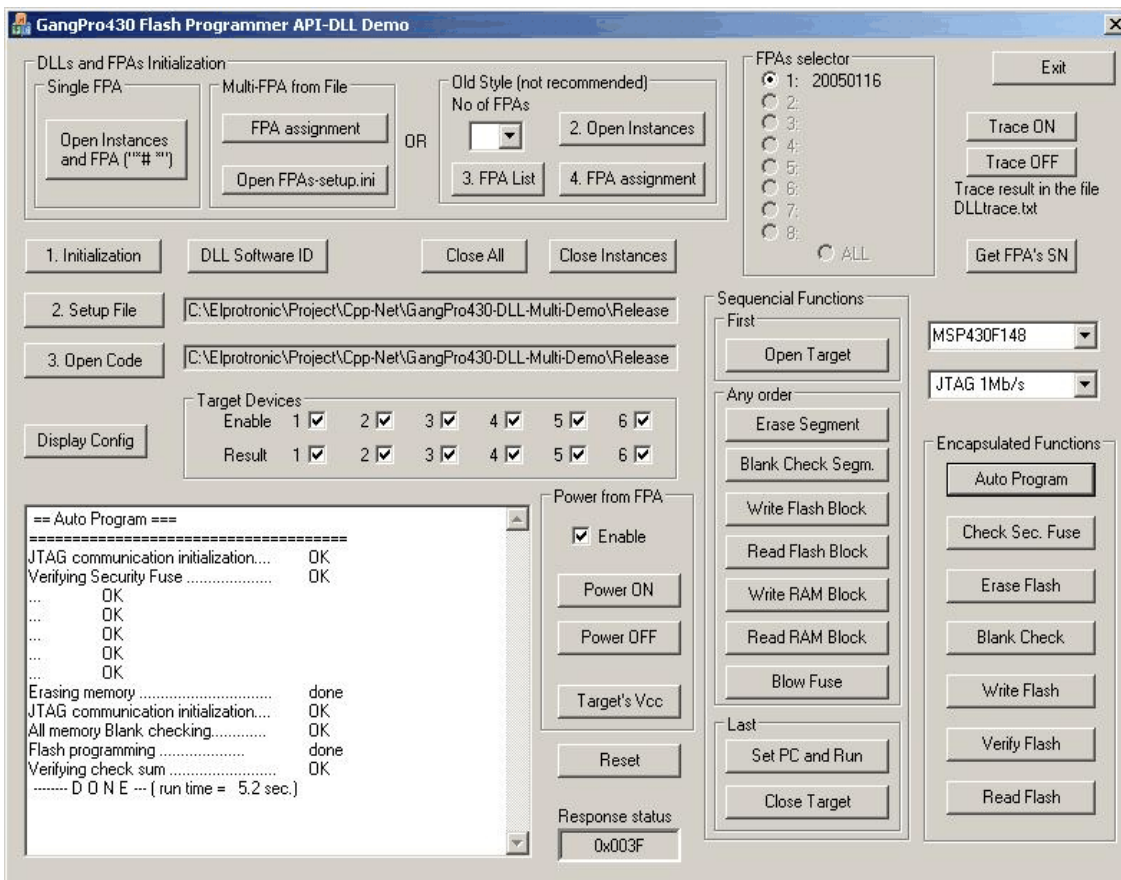


Figure 2.1 Demo program dialogue screen using DLLs.

directory

C:\Program Files\Elprotronic\USB GangPro430\API-DLL-Demo\Cpp\Exe

and write the serial numbers of the FPA's to be used. Take a serial numbers from the FPAs labels and write it on the desired FPAs locations FPA-1 up to FPA-64. If three FPAs will be used with SN 20050481, 20050482 and 20050483 then contents of the FPA's configuration file will be as follows:

```
=====
; USB-MSP430-FPA configuration setup *
; Elprotronic Inc. *
;-----
; up to eight FPA can be specified and connected via USB to PC *
; syntax: *
;   FPA-x   Serial Number *
;   where FPA-x can be FPA-1, FPA-2, FPA-3 .... up to FPA-64 *
;   Serial number - get serial number from the desires FPA's label *
;   Minimum one FPA's must be specified *
;   FPA-x order - any *
; *
; e.g (without semicolon - comment) *
; *
;FPA-1   20050116 *
;FPA-3   20050199 *
;FPA-5   20050198 *
=====
FPA-1   20050482
FPA-2   20050483
FPA-3   20050481
```

Note, that only lines without comments (without semicolon on the front) are used by software. All lines with comment are ignored. The FPA's serial numbers and FPA's indexes can be listed in any order and with gap like FPA-1, FPA-5 etc. without FPA-2, 3 etc. Minimum one valid FPA with correct SN must be specified. Up to eight FPAs can be declared. When the FPA's configuration file is created then file should be saved using name starting from **FPA** and with extention **ini** e.g **FPAs-setup.ini**.

Connect all required FPA's to USB connectors and run the **GangPro430DLL-Demo-Cpp.exe** demo software. First the DLL instances should be opened and all connected FPA's should be assigned to desired FPA's indexes. It is recommended to press the button '**FPA assignent**' located inside the frame named '**Multi-FPA from File**'. When this button is pressed, then the DLL function named

```
F_OpenInstancesAndFPAs( FileName );
```

is called. The list of defined FPA's serial numbers are taken from the FPAs configuration file and assigned all FPAs to desired FPA indexes (1 to 64). Number of instances to be opened is calculated

automatically, one per available and valid FPA. On described example with three FPAs in the '*FPAs selector*' will display three valid FPAs with list of used FPAs' serial numbers. All, others FPA-x fields will be disabled. In this example only three DLL instances becomes opened. Valid FPA indexes becomes 1,2,3 and ALL.

Other method that allows to open required number of instances uses '*2. Open Instances*' button in '*Old Style (not recommended)*' frame. First the number of the instances should be defined in the '*Total no of FPAs*' location. When the '*2. Open Instances*' button is pressed, then the DLL function named

```
F_OpenInstances( no );
```

called where 'no' - number of instances to be opened.. When the dll instances are opened, then it is possible to check the access to the FPAs connected to PC via USB ports. Pressing button '*3. FPA list*' (function **F_Check_FPA_access(index)**; called in loop for index = 1,2,3..8) allows to check the access to these adapters. On the end the button '*4. FPA assignment*' (function **F_Check_FPA_access(index)**; with desired '*fpa*' and USB indexes) allows to assign desired FPA adapter to '*fpa*' index. All these steps can be done automatically when the function **F_OpenInstancesAndFPAs(FileName)** described above is used (used button '*FPA assignment*' located inside the frame named '*Multi-FPA from File*').

When FPAs has been assigned then the '*1. Initialization*' button should be pressed. This initialization button calls the DLL function

```
F_Initialization();
```

and communication between programming adapter and PC is established. Report message is displayed on the edit window using **F_ReportMessage** function. By default config.ini file is empty and to make a required programmer setup the setup file can be downloaded to the programmer. It can be done by pressing button '*2.Setup File*', (executing **F_ConfigFileLoad** DLL function). Setup file can be prepared first using standard programmer software. Also desired code file can be downloaded by pressing button '*3. Open Code*' (executing **F_ReadCodeFile** DLL function).

There are seven buttons located on the right side of demo dialogue screen. Each of them calls one encapsulated function from the following list - **F_AutoProgram**, **F_VerifyFuseOrPassword**, **F_Memory_Erase**, **F_Memory_Blank_Check**, **F_Memory_Write**, **F_Memory_Verify** and **F_Memory_Read**

When any of these button is pressed, then a function, exactly in the same way how it is done in the standard MSP430 GangPro430 Flash Programmer software, is executed. Also buttons **Power ON/OFF**, **RESET** has the same action as related buttons in standard programmer. Refer to the *GangPro430 USB-MSP430 Gang Flash Programmer User's Manual* for details of these functions.

In the central part on dialogue screen there are buttons to simulate sequential DLL functions. Each of them calls one function from the sequential function list - **F_Open_Target_Device**, **F_Segment_Erase**, **F_Sectors_Blank_Check**, **F_Memory_Write_Data**, **F_Memory_Read_Data** and **F_Close_Target_Device**.

When a sequential function is called then *Open Target* (calling **F_Open_Target_Device** DLL function) must be pressed first. After that any button calling a sequential function can be pressed - in any order and as many times as required. To end sequential communication the button *Close Target* (calling **F_Close_Target_Device** DLL function) should be pressed.

In presented demo software all sequential functions have very small task to perform to demonstrate how to use the DLL functions. See source code of the DLL-Demo program in the software package in the ..\Demo-DLL subdirectory for details.

- Erase Segment:** Erase segment at location 0x1000 to 0x107F
- Blank Check Segm.** Segment blank check Erase at location 0x1000 to 0x107F
- Write Flash Block** Write 8 bytes to the flash memory at location 0x1020 to 0x1027.
Written data are different for all target devices
Target no 1: 0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18
Target no 2: 0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28
Target no 3: 0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38
Target no 4: 0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48
Target no 5: 0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58
Target no 6: 0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68
- Read Flash Block** Read 64 bytes starting from the flash memory address at 0x1000. On the report screen only 16 bytes from each target devices taken from addresses 0x1020 to 0x102F are displayed
- Write RAM Block** Write 16 bytes to RAM at location 0x0440 to 0x044F.
Written data are different for all target devices
Target no 1: 0x10,0x11,0x12,0x13,0x14,0x15,0x16,.....,0x1F
Target no 2: 0x20,0x21,0x22,0x23,0x24,0x25,0x26,.....,0x2F
Target no 3: 0x30,0x31,0x32,0x33,0x34,0x35,0x36,.....,0x3F
Target no 4: 0x40,0x41,0x42,0x43,0x44,0x45,0x46,.....,0x4F
Target no 5: 0x50,0x51,0x52,0x53,0x54,0x55,0x56,.....,0x5F
Target no 6: 0x60,0x61,0x62,0x63,0x64,0x65,0x66,.....,0x6F
Make sure that your target device has RAM at location 0x0400 and up.

- Read RAM Block*** Read 16 bytes RAM starting from address at 0x0440. On the report screen 16 bytes from each target devices taken from addresses 0x0440 to 0x044F are displayed
- Make sure that your target device has RAM at location 0x0400 and up.
- Blow Fuse*** Blow the security fuse. Function is protected and to be able to blow the security fuse the setup file should be modified with blow security fuse enabled option. Also warning message is displayed before blow security function becomes executed.
- Set PC and Run*** Small program is written and downloaded to each target devices. Program is saved in RAM at location stated from 0x0400. Written program is modifying contents of the RAM at locations 0x0440 to 0x0447. New RAM vales should be 0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0 When microcontrollers PC is modified from current location to 0x0400 and RUN, then contents of the RAM at location should be modified. To make a test do following test:
1. FPA List
 2. Initialization
 3. Setup File
 4. Power from FPA - Enable
 5. Open Target
 6. Write RAM Block
 7. Read RAM Block (remember RAM contents)
 8. Set PC and RUN (on the END target can be closed)
 9. Open Target
 10. Read RAM Block (compare with the contents from point 7).
- Make sure that your target device has RAM at location 0x0400 and up.

Also make a test using encapsulated functions - Autoprogram, Erase Flash etc. Encapsulated functions do not requires to open the target devices. All functions, including “Open..”, “Close...” are build-in in the encapsulated functions (see chapter 4 for details).

The second demo program is small GUI program with limited numbers of functions that allows to program up to 48 target devices. The programming status report of all units are displayed. Source code and all related project files are located in the following directory:

C:\Program Files\Elprotronic\MSP430\USB GangPro430\API-DLL-Demo\Cpp\Demo-8x6\release

Program can be activated by selecting the GangPro430DLL-8x6-Demo.exe. Demo program can also be activated from the windows menu:

Start->Program->Elprotronic MSP430 Flash Programmer->USB GangPro430->API-DLL-8x6-Demo

The “*FPA-setup.ini*” file must be created before. To do that open the “*FPA-setup.ini*” file and write the serial numbers of the Gang Programming adapters connected to PC. Only FPA listed in the “*FPA-setup.ini*” file will be activated. In the next step the standard GangPro430 software should be used that allows to access up to six target devices. Using the GangPro430 software the required configuration setup should be created (select microcontroller type, memory options etc) and saved using option “*Save Setup us..*”. When the “*FPA-setup.ini*” and configuration file is prepared, then the demo software can be used.

As the first the “*1. FPA assignment*” button should be pressed and the file “*FPA-setup.ini*” should be selected. Software will assign all specified FPAs to USB ports. If process is finished, then

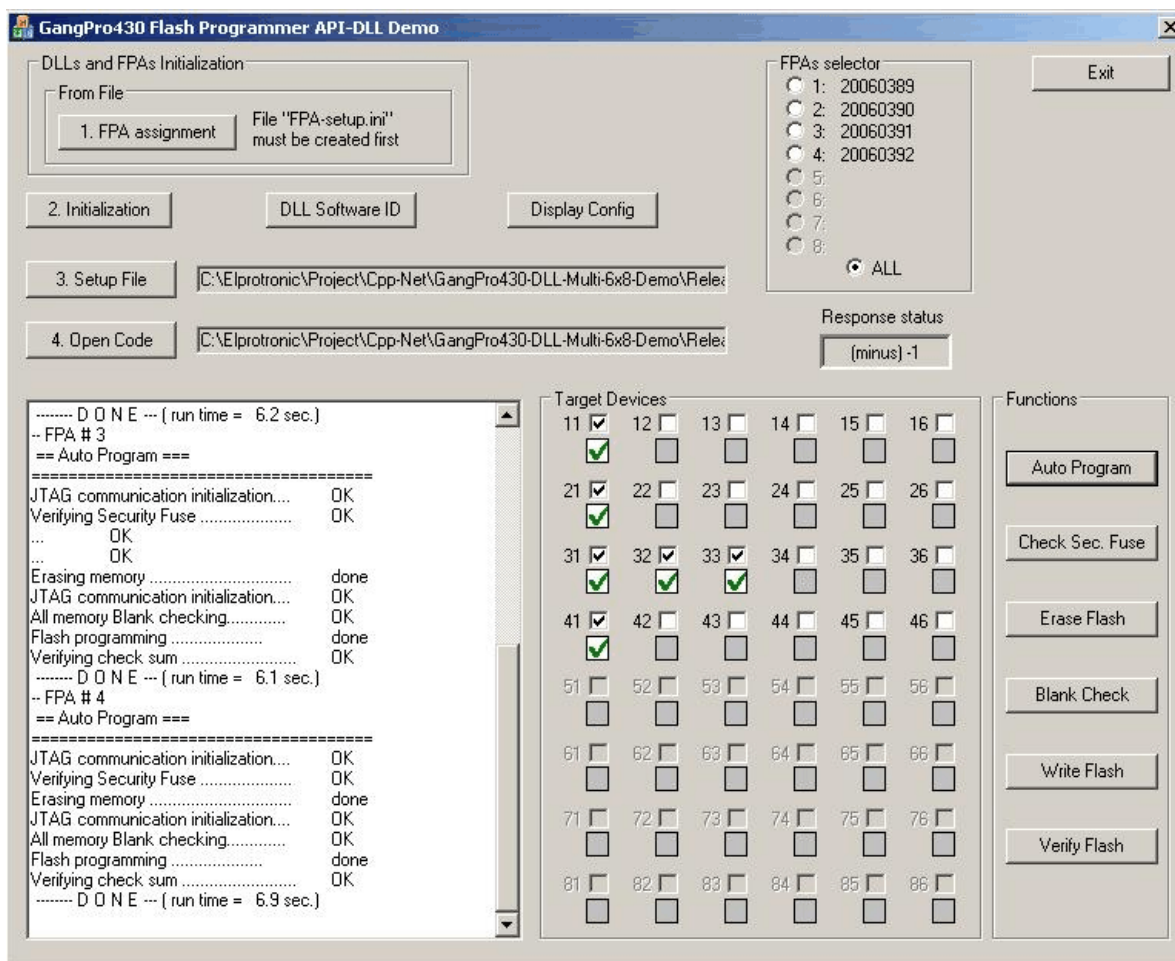


Figure 2.2 - Multi-FPA API-DLL demo dialogue screen.

the “**2. Initialization**” button should be pressed and as the next one the “**3 Setup file**” button should be used. Select the configuration file prepared by the GangPro430 software.. The same configuration setup will be used in all FPAs. On the end the “**4.Open Code**” should be used and required code file should be selected. Now the programmers are ready to download code file to up to 48 target devices. In the “**Target Devices**” group box is possible to select up to 48 desired target devices.

3. Example with FPA API DLL

Application DLLs files are the same for the application software written under Visual C++, Visual Basic, LabView etc. First should be created destination directory, where the executable files and DLLs will be located. Make a copy off all required files from the Elprotronic's directory to your destination directory. The files described in the chapter 1 should be copied to the executable destination directory. It is recommended to start the standard GangPro430 programming software to verify if the hardware and the drivers setup are correct. Using the GangPro430 programming software make a fully functional setup that satisfy your requirements. When it is done, save the existing programmer configuration setup using "*File->Save Setup as..*". Copy and paste the required files to your destination directory, where your application software is installed. It is recommended to use the demo program to verify if the setup in your PC and destination directory is done correctly. To do that copy the executable file

GangPro430DLL-Demo-Cpp.exe

from location directory

C:\Program Files\Elprotronic\USB GangPro430\API-DLL-Demo\Cpp\Exe

to your destination where your application software is installed and run the demo program. Follow instruction described in chapter 2 how to use the demo program.

3.1 Example with Single FPA API DLL

The code example described below uses single FPA API-DLL. The multi-FPA API-DLL selector should be select for FPA-1 only. All instructions related to single FPA are detailed described in the chapters 4.2, 4.3, 4.4 and 4.5. Instructions specific to multi-FPA features described in the chapter 4.1. When the single FPA functions are used, then `fpa_index = 1` should be set.

Initialization opening procedure for the USB-FPA can be as follows:

```
F_OpenInstancesAndFPAs( "*"#" ); // DLL and FPA (one only) initialization
F_Set_FPA_index( 1 );           // select FPA 1 for
F_Initialization( );           // init FPA
```

Below is an example of the simplified (without error handling procedures) application program written in C++ that allows to initialize all dlls and FPA, and run an autoprogram with the same features like autoprogram in the standard GangPro430 application software.

1. Download data to target device

```
F_OpenInstancesAndFPAs( "*" # "*" ); // DLL and FPA (one only) initialization
F_Set_FPA_index( 1 ); // select FPA 1
F_Initialization( ); // init FPA
//-^^^ instructions above should be called only once at the startup ^^-----
F_ReadConfigFile( filename ); // read configuration data and save
// to API-DLLs
F_ReadCodeFile( format, filename ); //read code data and save to DLL

do
{
    status = F_AutoProgram( 1 ); //start autoprogram
    if ( status != TRUE )
    {
        status = F_LastStatus(1);
        .....
    }
    else
    {
        .....
    }
} while(1); //make an infinite loop until last target device programmed
.....
F_CloseInstances();
```

3.2 Example with Multi-FPA API DLL

The code example described below uses Multi-FPA API-DLL. The multi-FPA API-DLL is a shell that allows to transfer incoming instructions from application software to desired FPA's. All instructions related to single FPA are detailed described in the chapters 4.2, 4.3 and 4.4. Instructions specific to multi-FPA features described in the chapter 4.1.

Application DLL should be initialized first, before other DLLs instruction can be used. Initialization opening procedure for the USB-FPA can be as follows:

```
F_OpenInstancesAndFPAs( FPAs-setup.ini); // DLL and FPA initialization
F_Set_FPA_index( ALL_ACTIVE_FPA ); // select all FPA's
F_Initialization( ); // init all FPA's
```

In example above number of the opened USB-FPAs are specified in the *'FPAs-setup.ini'*

Below is an example of the simplified (without error handling procedures) application program written in C++ that allows to initialize all dlls and FPA, and run an autoprogram with the same features like autoprogram in the standard FlashPro430 application software.

1. Download data to all target devices (uses USB-FPAs)

```
F_OpenInstancesAndFPAs( FPAs-setup.ini); // DLL and FPA initialization
F_Set_FPA_index( ALL_ACTIVE_FPA );      // select all FPA's
F_Initialization( );                    // init all FPA's
//-^^^ instructions above should be called only once at the startup ^^-----

F_ReadConfigFile( filename );           // read configuration data and save
                                         // to all API-DLLs
F_ReadCodeFile( format, filename );     // read code data and save to all
                                         // API-DLLs

do
{
    status = F_AutoProgram( 1 );
        //start autoprogram-to program all targets simultaneously with
        //the same downloaded data to all target devices.
    if ( status != TRUE )
    {
        if ( status == FPA_UNMACHED_RESULTS )
        {
            for (n=1; n<=MAX_FPA_INDEX; n++ ) status[n] = = F_LastStatus( n);
            .....
        }
        else
        {
            .....
        }
    }
    } while(1); //make an infinite loop until last target device programmed
    .....
//-vvv instruction below should be called only once at the end vvv-----

F_CloseInstances();
```


Note, that all single API-DLL are independent from each others and it is not required that all data and configuration should be the same for each API-DLLs (each FPAs, or target devices) . For example - code data downloaded to the target devices connected to first FPA can be the same (but it is not required) as code data downloaded to the target devices connected to second FPA etc. In the example below the downloaded code to target devices are not the same .

2. Download independent data to target devices (uses USB-FPAs)

```

F_OpenInstancesAndFPAs( FPAs-setup.ini); // DLL and FPA initialization
F_Set_FPA_index( ALL_ACTIVE_FPA );      // select all FPA's
F_Initialization( );                    // init all FPA's
//-^^^ instructions above should be called only once at the startup ^^-----
.....
F_Set_FPA_index( ALL_ACTIVE_FPA );      // select all FPA's
F_ReadConfigFile( filename );           // read configuration data and save
                                          // to all API-DLLs
F_Set_FPA_index( 1 );                   // select FPA 1
F_ReadCodeFile( format, filename1 );    // read code data and save to
                                          // API-DLL-1
F_Set_FPA_index( 2 );                   // select FPA 2
F_ReadCodeFile( format, filename2 );    // read code data and save to
                                          // API-DLL-2
.....
F_Set_FPA_index(7 );                    // select FPA 7
F_ReadCodeFile( format, filename7 );    // read code data and save to
                                          // API-DLL-7
F_Set_FPA_index( 8 );                   // select FPA 8
F_ReadCodeFile( 8, format, filename8 ); // read code data and save to
                                          // API-DLL-8
F_Set_FPA_index( ALL_ACTIVE_FPA );      // select all FPA's
do
{
    status = F_AutoProgram( 1 );
    //start autoprogram - to program all targets simultaneously
    //with the independent downloaded data to all target devices.

    if ( status != TRUE )
    {
        if ( status == FPA_UNMACHED_RESULTS )
        {
            for (n=1; n<=MAX_FPA_INDEX; n++ ) status[n] = = F_LastStatus( n);
            .....
        }
        else

```

```
    {  
        .....  
    }  
}  
} while(1); //make an infinite loop until last target device programmed  
.....  
//--vvv instruction below should be called only once at the end vvv----  
F_CloseInstances();
```

See source code in the DEMO program written in Visual C++, Visual Basic or LabView for more detail.

4. List of the DLL instructions

All DLL instructions are divided to four groups - related to Multi-FPA selector, single FPA generic, single FPA encapsulated and single FPA sequential instructions. Multi-FPA specific instructions are related to the Multi-FPA DLL only. Generic instructions are related to initialization programmer process, while encapsulated and sequential instructions are related to target device's function. Encapsulated and sequential instructions can write, read, and erase contents of the target device's flash memory.

Multi-FPA specific instructions are related to load and release the single-FPA dlls, selection of the transparent path and sequential/simultaneous instructions transfer management. All other instructions are related to single FPAs.

Generic instructions are related to initialization programmer process, configuration setup and data preparation, Vcc and Reset to the target device. Generic instructions should be called first, before encapsulated and sequential instruction.

Encapsulated instructions are fully independent executable instructions providing access to the target device. Encapsulated instructions can be called at any time and in any order. When called then all initialization communication with the target device is starting first, after that requested function is executed and at the end communication with the target device is terminated and target device is released from the programming adapter.

The encapsulated functions should be mainly used for programming target devices. These functions perform most tasks required during programming in an easy to use format. These functions use data provided in Code Files, which should be loaded before the encapsulated functions are used. To augment the functionality of the encapsulated functions, sequential functions can be executed immediately after to complete the programming process.

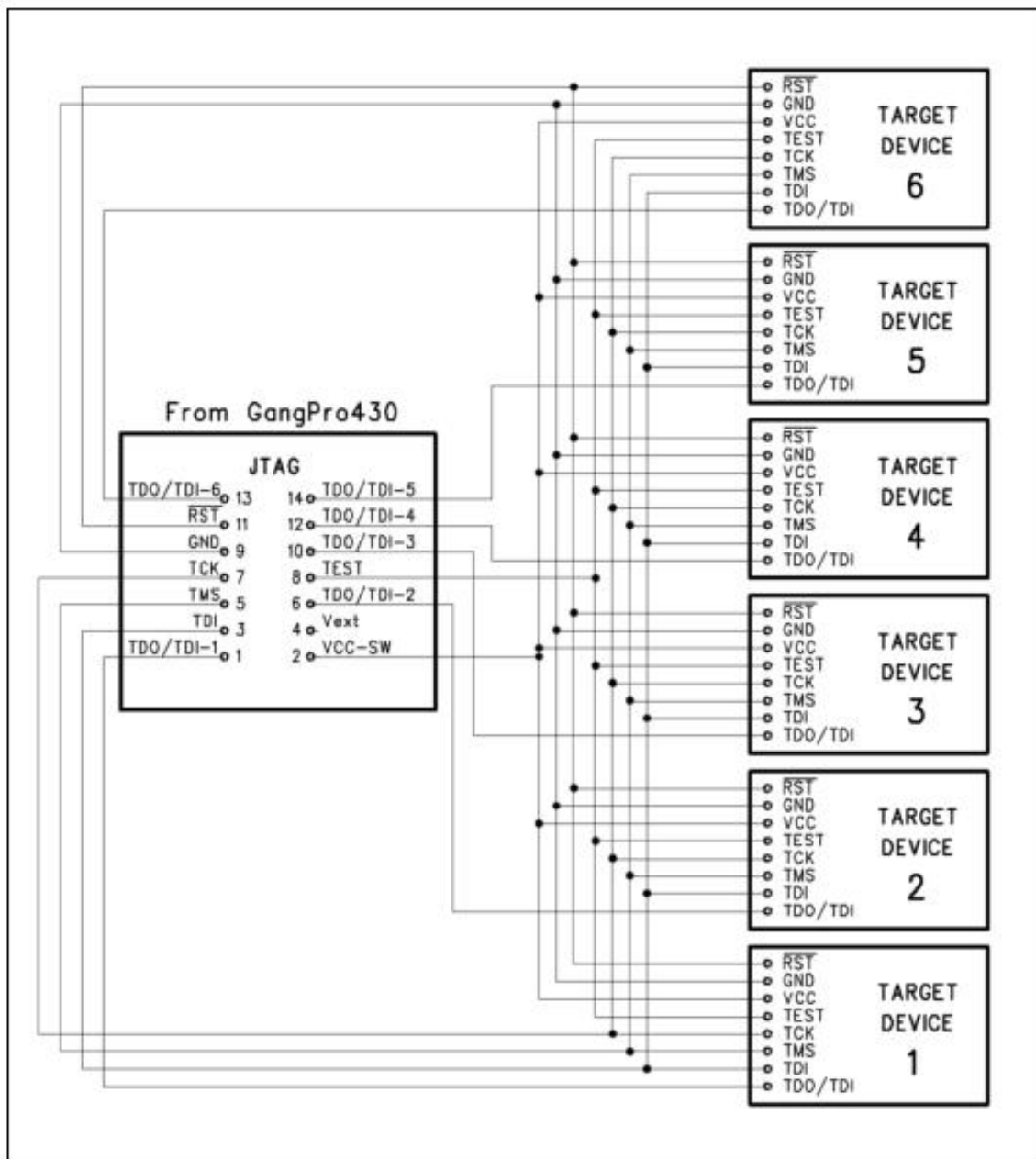
Sequential instructions allow access to the target device in a step-by-step fashion. For example, a typical sequence of instructions used to read data from the target device would be to open the target device, then read data and then close the target device. Sequential instruction have access to the target device only when communication between target device and programming adapter is initialized. This can be done when *Open Target Device* instruction is called. When communication is established, then any number of sequential instruction can be called. When the process is finished, then at the end *Close Target Device* instruction should be called. When communication is terminated, then sequential instructions can not be executed.

Note: Inputs / outputs has been defined as INP_X, and LONG_X. Both of them are defined as 4 bytes long (see MSPPrG-DLL.h header file)

```
#define INP_X    _int32
```

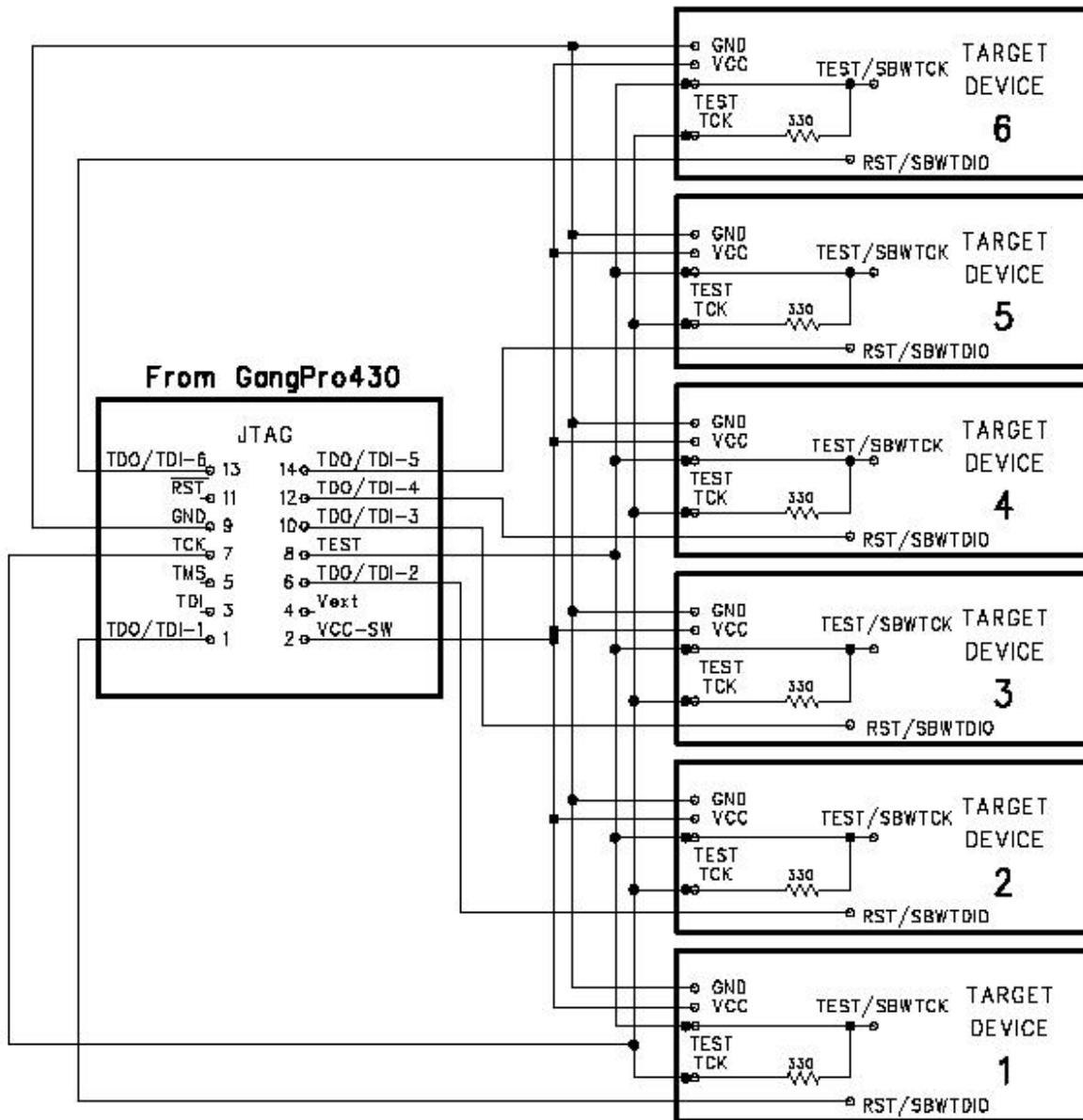
```
#define LONG_X  _int32
```

Make sure that an application using the DLL file has the same length of desired data.



GangPro430 with the target devices connection diagram

Figure 4.1



Note: Connection between TEST (pin-8) and target's devices can be removed if blow the JTAG security fuse is not required.

Connection of the GangPro430 with target's Spy-By-Wire interface.

Figure 4.2

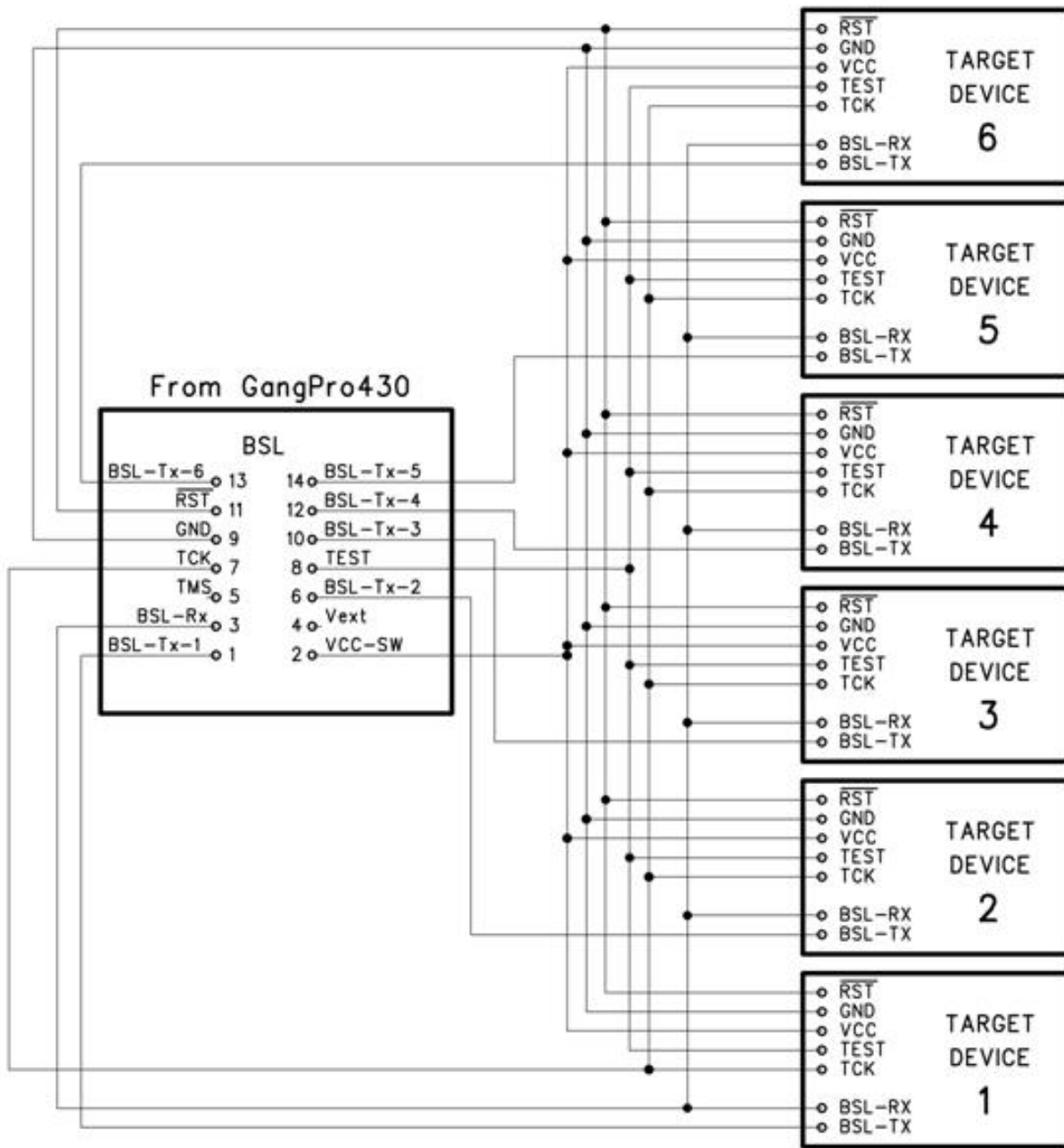


Figure 4.3 Gang BSL connection

The DLL instructions have the following convention related to the target device number and target device mask set/result.

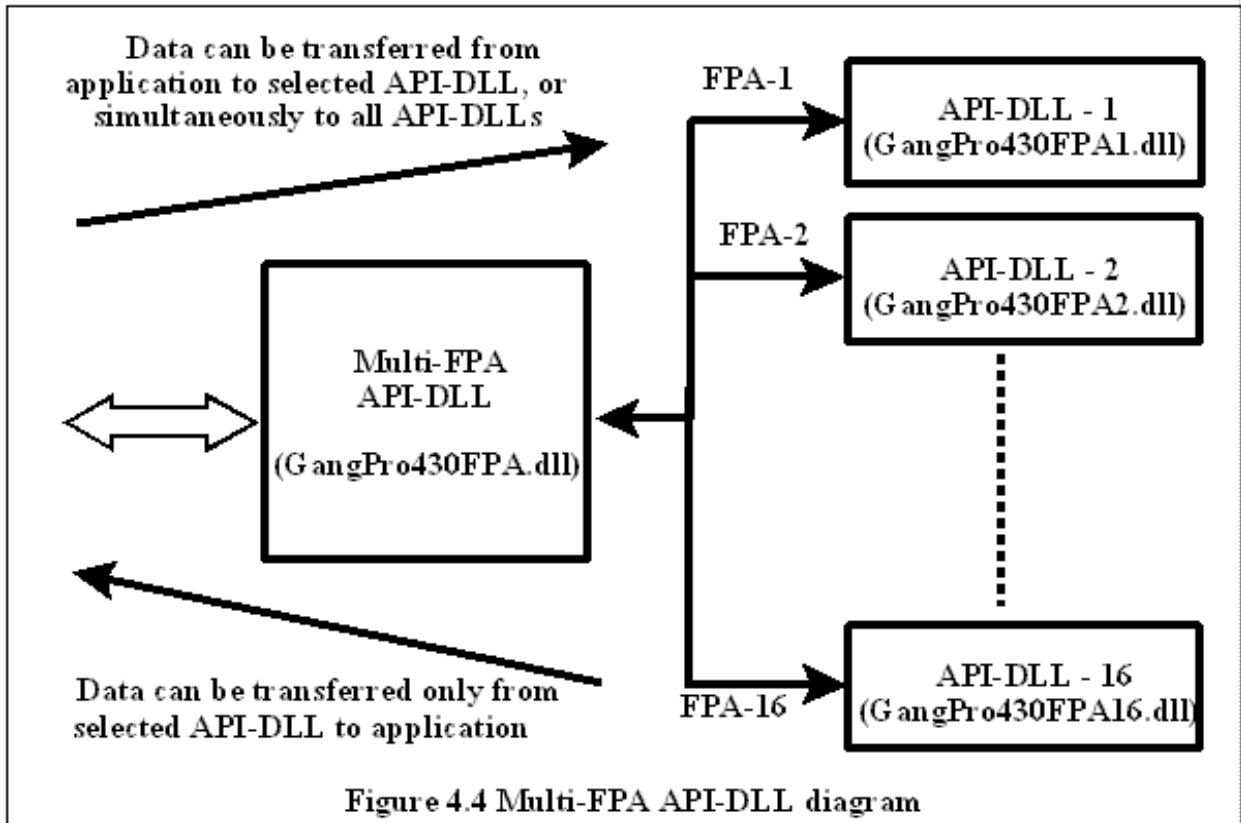
Table 4.1

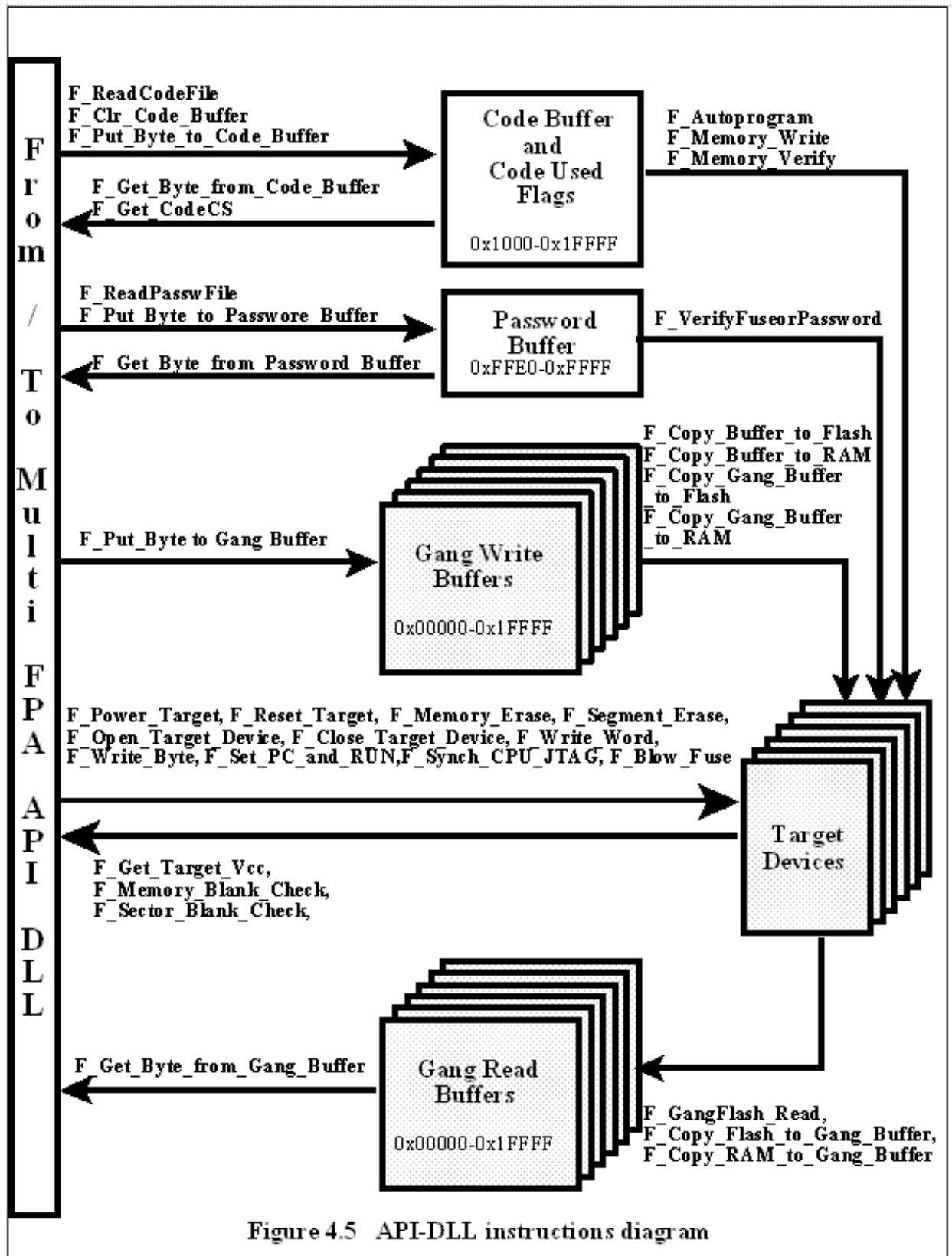
JTAG/SBW connector pin #	TDO/TDI #	Target device #	Target Device MASK
1	1	1	0x01 (01h)
6	2	2	0x02 (02h)
10	3	3	0x04 (04h)
12	4	4	0x08 (08h)
14	5	5	0x10 (10h)
13	6	6	0x20 (20h)

Important information:

When the target device is connected to the Gang Programmer and it is disabled in the Gang Programmer software, then due to parallel connection of the JTAG/SBW/BSL interface with the others target devices, contents of the disabled target devices can be erased, modified etc. Contents of the disabled target devices is not verified. Make a sure that all connected target devices are enabled in the programming software. Disable the target device in the programming software only if you would like to ignore particular target device. Also make sure, that all target devices are the same. Do not mixed the different type of the target devices.

Figure 4.4 shows the structure of the Multi-FPA API-DLL. It shows that the Multi-FPA DLL is used to communicate with the user application as well as the target devices. Each of the target devices is accessed by a single DLL associated with it. When more then one FPA is needed, up to 64 DLLs can be created to communicate with up to $6*64 = 384$ target devices at a time. Each instance of an FPA-DLL contains its own copy of buffers, as shown in Figure 4.5





4.1 Multi-FPA instructions

The Multi-FPA API-DLL instructions are related to Multi-FPA selector only. These instructions allow to initialize all single application DLLs and select the instruction patch between application software and desired FPA and sequential/simultaneous instructions transfer management. Up to eight independent FPAs can be remotely controlled from the application software. All instructions from application software can be transferred to one selected FPA or to all FPAs at once. That feature allows to increase programming speed up to eight times and also allows to have individual access to any FPA is required.

F_Trace_ON

F_Trace_ON - This function activates the tracing.

The `F_Trace_ON()` opens the `DLLtrace.txt` file located in the current directory and records all API-DLL instructions called from the application software. This feature is useful for debugging. When debugging is not required, then tracing should be disabled. Communication history recorded in the last session can be viewed in the `DLLtrace.txt` located in the directory where the API-DLL file is located. When the new session is established, then the file `DLLtrace.txt` is erased and new trace history is recorded.

***Note:** Tracing is slowing the time execution, because all information passed from application software to API-DLL are recorded in the `dlltrace.txt` file.*

F_Trace_OFF

F_Trace_OFF - Disable tracing, See **F_Trace_ON** for details.

F_OpenInstances

F_OpenInstances - API-DLL initialization in the PC.

Instruction must be called first - before all other instructions. Instead of this function, the `F_OpenInstancesAndFPAs` is recommended.

Important: It is **not recommended** to use this function. Function used only for compatibility with the old software. Use the `F_OpenInstancesAndFPAs` instead.

Do not use the **F_OpenInstances** or **F_Check_FPA_access** after using the **F_OpenInstancesAndFPAs**. The **F_OpenInstancesAndFPAs** is assigning the FPAs to USB ports and it is not recommended to reassign once again the USB port using the **F_Check_FPA_access** function. To check the communication activity with FPA use the **F_Get_FPA_SN** function that allows to check the communication with the FPA adapter without modifying the USB ports assignment.

Syntax:

```
INT_X      MSPPRG_API  F_OpenInstances ( BYTE no );
```

Parameters:

no -> number of the single API-DLL to be open
no -> 1 to MAX_USB_DEV_NUMBER
where MAX_USB_DEV_NUMBER = 64

Return value:

number of opened instances

F_CloseInstances

F_CloseInstances - Close all active API-DLLs and free system memory.

Syntax:

```
INT_X      MSPPRG_API  F_CloseInstances ( void );
```

Parameters:

void no -> 1 to MAX_USB_DEV_NUMBER
where MAX_USB_DEV_NUMBER = 64

Return value:

TRUE

F_OpenInstancesAndFPAs, F_OpenInstances_AndFPAs

F_OpenInstancesAndFPAs - API-DLL initialization in the PC and FPA scan and
or **F_OpenInstances_AndFPAs** assignment to desired USB port according to contents of the
FPA's list specified in the string or FPA's configuration file.

Instruction must be called first - before all other instruction. Instead this function the **F_OpenInstances** can be used. Function can be used only when the USB FPA are used. When the USB-FPA is used, then the function **F_OpenInstancesAndFPAs** is recommended in the initialization

process. Function is very convenient - automatically is opening the number of the desired API-DLL and assigning the desired FPA to available USB ports. Regardless of the USB port open sequence and connection of the FPA to USB ports, the `F_OpenInstancesAndFPAs` instruction is reading the FPA's list, scanning all available FPAs connected to any USB ports and assigning the indexes to all FPAs according to contents of the FPA list (from string or configuration file). All FPAs not listed in the FPA configuration file and connected to USB ports are ignored.

Important: Do not use the `F_Check_FPA_access` after using the `F_OpenInstancesAndFPAs`. The `F_OpenInstancesAndFPAs` is assigning the FPAs to USB ports and it is not recommended to reassign once again the USB port using the `F_Check_FPA_access` function. To check the communication activity with FPA use the `F_Get_FPA_SN` function that allows to check the communication with the FPA adapter without modifying the USB ports assignment.

Syntax:

```
INT_X      MSPPRG_API  F_OpenInstancesAndFPAs( char * List );  
INT_X      MSPPRG_API  F_OpenInstances_AndFPAs( CString List );
```

Parameters:

1. When the first two characters in the List string are `*#`, then string reminding characters contains list of desired FPAs serial numbers assigned to FPA-1, -2, ...-n indexes, eg.
"`*# 20060123, 20060234, 20060287`"
2. When the first two characters in the List string are not `*#`, then string contains file name or full path of the file with list of the FPA's serial numbers, eg.
"`C:\Program Files\Elprotronic\FPAs-setup.ini`"

Return value:

number of opened instances

1. The FPA list in the string:

String -> "`*# SN1, SN2, SN3, SN4, SN5...`"

Where the

- SN1- FPA's serial number that should be assigned to FPA-1 index
- SN2- FPA's serial number that should be assigned to FPA-2 index
- etc.

As a delimiter the comma `,` or white space can be used.

Example:

```
“*# 20060123, 0, 20060346, 20060222, 20060245”
```

or

```
“*# 20060123 0 20060346 20060222 20060245”
```

In example above the FPAs will be assigned as follows:

```
FPA-1 20060123
FPA-2 0 //empty - FPA is not assigned
FPA-3 20060346
FPA-4 20060222
FPA-5 20060245
```

In the FPA list can be specified ONE adapter with any serial number when the character ‘*’ is used instead the FPA’s serial number. Only one ‘*’ character can be specified in the FPA list and must be located on the end of valid SN list. All other serial numbers specified after ‘*’ will be ignored. This option allows to specify any FPA when the only one adapter is used eg.

```
“*# *”
```

FPA-1 -> Any FPA is one adapter is connected, or the first detected adapter, if more then one adapters are connected.

or if more then one adapter is used

```
“*# 20060123 *”
```

```
FPA-1 20060123
FPA-2 - first detected adapter (excluding already assigned adapters), if
more adapters are connected.
```

When the ‘*’ is inside the FPA list, eg.

```
“*# 20060123 * 20060137 20060166”
```

then the last two FPA’s SN will be ignored

```
FPA-1 20060123
FPA-2 - first detected adapter (excluding already assigned adapters), if
more adapters are connected.
FPA-3 - not assigned
```

Initialization example:

1. F_OpenInstances_AndFPAs(“*# *”); // only one FPA - any SN
or
2. F_OpenInstances_AndFPAs(snlist); // hardcoded SN list

2.The FPA list in the configuration file:

String -> “C:\Program Files\Elprotronic\FPAs-setup.ini”

Example of the FPA configuration file:

```
;    -> semicolon - comment
; Syntax of the FPAs configuration specified
; FPA-x   Serial Number
; where  FPA-x can be  FPA-1, FPA-2, FPA-3 .... up to FPA-64
;   e.g
FPA-1   20050116
FPA-3   20050199
FPA-5   20050198
FPA-6   20050205
; FPA-x can be listed in any order and can contain gaps,
; like above without FPA-2, FPA-4
; When list like above is used, then following fpa can be valid
; fpa -> 1,3,5,6
; NotePad editor can be used to create the FPA configuration file.
```

When the '*' is used instead FPA's SN, then any FPA will be accepted. The '*' can be used only once and on the end of the FPA's list eg.

```
FPA-1   20050116
FPA-3   20050199
FPA-5   *
```

or

```
FPA-1   *
```

when only one adapter (any adapter) is used.

Example:

```
F_OpenInstancesAndFPAs( FPAs-setup.ini );
                        //DLL startup and FPA assignment
F_Set_FPA_index (ALL_ACTIVE_FPA);
                        //select all available FPAs
F_Initialization();
                        //init all FPAs
F_ReadConfigFile( filename );
                        //download the same configuration to all DLLs.
F_ReadCodeFile( format, filename );
                        //download the same code file to all DLLs.

do
{
    status = AutoProgram(1);
```

```

        //start autoprogram to all FPAs simultaneously.
    if( status != TRUE )
    {
        if( status == FPA_UNMATCHED_RESULTS )
        {
            // service software when results from FPAs are not the same
        }
        else
        {
            .....
        }
        .....
    }
} while(1);
F_CloseInstances();
// release DLLs from memory

```

F_Set_FPA_index

F_Set_FPA_index - Select desired FPA index (desired DLL instance)

VALID FPA index - (1 to 64) or 0 (ALL FPAs).

Syntax:

```
INT_X      MSPPRG_API  F_CSet_FPA_index ( BYTE fpa );
```

Parameters:

fpa -> 1 to MAX_FPA_INDEX where MAX_FPA_INDEX = 64
or 0 -> ALL_ACTIVE_FPA

note: instead of '0' value it can be used global defined

ALL_ACTIVE_FPA that is defined as

```
#define ALL_ACTIVE_FPA 0
```

in the header file

Return value:

TRUE - if used fpa index is valid

FPA_INVALID_NO - if used fpa index is not activated or out of range

note: FPA_INVALID_NO -> -2 (minus 2)

IMPORTANT: When any function is trying to access the invalid FPA, then return value from this function is -2 (FPA_INVALID_NO)

Note: When index *ALL_ACTIVE_FPA (0)* is used, then all data can be transferred from application to all active FPA's (API-DLLs). However, when the data is transferred from FPA (or API-DLLs) to the application, then the FPA index CANNOT be *ALL_ACTIVE_FPA (0)*. Index must select desired FPA. When the simultaneous process is required eg. reading flash contents from all target devices, then the *F_Copy_All_Flash_to_Buffer()* should be called after the *F_Set_FPA_index(ALL_ACTIVE_FPA)*. When finished, the contents of each buffer (inside each API-DLLx) can be read using the *F_Set_FPA_index(1), F_Set_FPA_index(2)*, and *F_Get_Byte_from_Buffer(..)*. See below

```

F_Set_FPA_index (ALL_ACTIVE_FPA); //select all available FPAs
F_Copy_All_Flash_to_Buffer();      //simultaneous process
for( fpa=1; fpa=fpa_max; fpa++ )
{
    if( F_Set_FPA_index(fpa) == FPA_INVALID_NO ) continue;
    for(addr = addr_min; addr <= addr_max; addr++)
    {
        data[addr][fpa-1] = F_Get_Byte_from_Buffer(addr);
    }
}

```

F_Get_FPA_index

F_Get_FPA_index - Get current FPA index

Syntax:

```
BYTE MSPPRG_API F_Get_FPA_index ( void );
```

Return value:

```
current FPA index
```

F_Check_FPA_index

F_Check_FPA_index - Get current FPA index and check if index is valid.

Similar function to the *F_Get_FPA_index*, however, while the *F_Get_FPA_index* is returning current FPA index regardless if the index is valid or not, simply returning the value set by the function *F_Set_FPA_index(..)*. The *Check_FPA_index* will return -2 (minus two) *FPA_INVALID_NO* if FPA is pointing not initialized FPA (dll instance).

Syntax:

```
INT_X MSPPRG_API F_Check_FPA_index ( void );
```


Return value:

current FPA index (0, 1 to 64)
or -2 (minus two) FPA_INVALID_NO

F_Disable_FPA_index

F_Disable_FPA_index - Disable desired FPA index (desired DLL instance)
VALID FPA index - (1 to 64)

Function allows to disable communication with selected FPA adapter. From application point of view, all responses will be the same as from the not active FPA. Communication with target devices connected to selected FPA will be stopped. When the F_Set_FPA_index(0) will be used, then selected FPA will be ignored. Result will not be presented in the Status results (Status and F_LastStatus(..)).

Syntax:

```
void MSPPRG_API F_Disable_FPA_index ( BYTE fpa );
```

Parameters:

fpa -> 1 to MAX_FPA_INDEX where MAX_FPA_INDEX = 64

F_Enable_FPA_index

F_Enable_FPA_index - Enable desired FPA index (desired DLL instance)
VALID FPA index - (1 to 64)

Function allows to enable communication with selected FPA adapter if the mentioned FPA has been disabled using the function F_Disable_FPA_index(...). By default, all FPAs are enabled.

Syntax:

```
void MSPPRG_API F_Enable_FPA_index ( BYTE fpa );
```

Parameters:

fpa -> 1 to MAX_FPA_INDEX where MAX_FPA_INDEX = 64

F_LastStatus

F_LastStatus - Get current FPA index

VALID FPA index - (1 to 64)

Syntax:

```
INT_X MSPPRG_API F_LastStatus ( BYTE fpa );
```

Parameters:

fpa - FPA index of the desired status
fpa index -> 1..64

Return value:

Last status from the desired FPAs

All F_XXX functions returns the same parameters (status) as the original API_DLL is returning. When function is transferred to all API-DLLs (when the fpa=0) then returned parameter (status) is the same as the returned value from the API-DLLs when the ALL returned values ARE THE SAME. If not, then returned value is

FPA_UNMATCHED_RESULTS

(value of the FPA_UNMATCHED_RESULTS is minus 1).

To get the returned values from each FPAs, use the

```
For( n=1; n<=64; n++) status[n] = F_LastStatus( n);
```

where n -> desired FPA index

and get the last status data from FPA-1, 2, .. up to 64

F_Multi_DLLTypeVer

F_Multi_DLLTypeVer function returns integer number with DLL ID and software revision version.

Syntax:

```
MSPPRG_API INT_X F_Multi_DLLTypeVer( void );
```

Return value:

```
VALUE = (DLL ID) | ( 0x0FFF & Version)
DLL ID = 0x4000   - Multi-DLL for the FlashPro
DLL ID = 0x5000   - Multi-DLL for the GangPro
Version = (0x0FFF & VALUE)
```

F_Get_FPA_SN

F_Get_FPA_SN - Get FPAs Serial number assigned to selected FPA-index (selected DLL instance number).

Syntax:

```
LONG_X            MSPPRG_API   F_Get_FPA_SN ( BYTE fpa );
```

Parameters:

```
fpa - FPA index of the desired status
fpa index -> 1..64
```

Return value:

```
Serial number of the selected FPA
or FPA_INVALID_NO - if used fpa index is not activated or out of range.
note: FPA_INVALID_NO -> -2 (minus 2)
```

4.2 Generic instructions

Generic instructions are related to initialization programmer process, configuration setup and preparation data, turning ON and OFF target's DC and RESET target device. Any communication with the target device is provided when any of the generic instruction is executed. Generic instructions should be called before encapsulated and sequential instruction.

F_Check_FPA_access

F_Check_FPA_access - Check available Flash Programming Adapter connected to specified USB drivers (USB driver index from 1 to 64)

VALID FPA index (DLL instance number) - (1 to 64)

Important: It is **not recommended** to use this function. Function used only for compatible with the old software. Use the **F_OpenInstancesAndFPAs** instead.

Do not use the **F_OpenInstances** or **F_Check_FPA_access** after using the **F_OpenInstancesAndFPAs**. The **F_OpenInstancesAndFPAs** is assigning the FPAs to USB ports and it is not recommended to reassign once again the USB port using the **F_Check_FPA_access** function. To check the communication activity with FPA use the **F_Get_FPA_SN** function that allows to check the communication with the FPA adapter without modifying the USB ports assignment.

F_Check_FPA_access should be called as a first function when the *.dll is activated. Function returns serial number of the detected flash programming adapter, or zero, if programming adapter has not been detected with selected USB driver. Up to 64 USB drivers can be scanned.

To make a Multi-FPA software back compatible, the **F_Check_FPA_access** procedure is calling the function **F_OpenInstances** if none of the instances has not been activated before. That allows to use old application software without calling the new type of Multi-FPA functions.

Syntax:

```
MSPPRG_API            LONG_X            F_Check_FPA_access ( INT_X USB_index );
```

Parameters:

Index: USB driver index from 1 to MAX_USB_DEV_NUMBER
where MAX_USB_DEV_NUMBER = 64

Return value:

0 - FALSE
>0 - Detected FPA's Serial Number

Example:

```
long SN[MAX_USB_DEV_NUMBER+1];
F_OpenInstances( 1 ); // DLL initialization - one instance
F_Set_FPA_index( 1 ); // select access to the first instance
n = 0; //no of detected FPAs
for( k=1; k<MAX_USB_DEV_NUMBER ; k++ )
{
    SN[k] = F_Check_FPA_access(k);
    if ( SN[k] > 0 ) n++;
}
F_CloseInstances(); // DLL initialization - one instance
F_OpenInstances( n ); // Open 'n' instances - one per FPA

// Find desired FPAs SN and assign the FPAs serial number every time to the same
// FPA-index.
// For example if the
// SN[1]= 20060123
// SN[2]= 20060147
// SN[3]= 0 - adapter not present
// SN[4]= 20060135
// and desired assignment
// FPA-1 20060123
// FPA-2 20060135
// FPA-3 20060147
// then following sequence instructions can be used

F_Set_FPA_index( 1 ); // select access to the first instance
F_Check_FPA_access( 1 ); //assign FPA SN[1] = 20060123 to FPA-1
F_Set_FPA_index( 2 ); // select access to the second instance
F_Check_FPA_access( 4 ); //assign FPA SN[4] = 20060135 to FPA-2
F_Set_FPA_index( 3 ); // select access to the third instance
F_Check_FPA_access( 2 ); //assign FPA SN[2] = 20060147 to FPA-3

F_Set_FPA_index( ALL_ACTIVE_FPA ); // select all active instances
F_Initialization() // All FPAs initialization

.....
```

F_Initialization

F_Initialization - Programmer initialization.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) *executed sequentially.*

F_Initialization function should be called after the communication with the FPA adapter is established. To make a Multi-FPA software back compatible, the F_Initialization procedure is calling the function **F_OpenInstances** if none of the instances has not been activated before. That allows to use old application software without calling the new type of Multi-FPA functions. In this case the **F_Check_FPA_access** function can be used to activate communication between PC and Programming Adapter. When the **F_Check_FPA_access** is not called then by default the USB driver number "1" is selected.

When the **F_Initialization** is called then:

- all internal data is cleared or set to the default value,
- initial configuration is downloaded from the config.ini file,
- USB driver is initialized if has not been initialized before.

Programming adapter must be connected to the USB or Parallel Port to establish communication between PC and programming adapter. Otherwise the F_Initialization will return FALSE result.

Syntax:

```
MSPPRG_API            INT_X F_Initialization( void );
```

Return value:

- 0 - FALSE
- 1 - TRUE
- 4 - Programming adapter not detected.

Example:

```
.....  
F_API_DLL_Directory( "....." ) // optional - see F_API_DLL_Directory()  
If( F_Initialization() != TRUE ) //required API-Dll - initialization  
{  
    // Initialization error  
}  
.....
```

F_API_DLL_Directory

F_API_DLL_Directory - The DLL directory location.

VALID FPA index - *irrelevant - the same directory location for all DLLs.*

The **F_API_DLL_Directory** command can specify the directory path where the DLLs are located. This command is not mandatory and usually is not required. But in some application software (like in the LabVIEW) the default location of the DLLs is not transferred to the DLL. In this case the related files with DLLs located in the same directory where the DLLs are located can not be find. To avoid this problem the full path of the directory where the DLLs are located can be specified. The **F_API_DLL_Directory** must be used before **F_Initialization()** function.

Syntax:

```
MSPPRG_API void F_API_DLL_Directory( Cstring APIDLLpath );  
or MSPPRG_API void F_API_DLL_Directory( char* APIDLLpath );
```

Example:

```
.....  
F_API_DLL_Directory( "C:\\Program Files\\Test\\LabVIEW" );  
    // directory where the API-DLLs are located  
    //      C:\\Program Files\\Test\\LabVIEW  
If( F_Initialization() != TRUE ) //required API-Dll - initialization  
{  
    // Initialization error  
}  
.....
```

F_Close_All

F_Close_All - Close communication with the programming adapter and release PC memory.

VALID FPA index - *(1 to 64) or 0 (ALL FPAs) executed sequentially.*

F_Close_All function should be called as the last one before *.dll is closed. When the F_Close_All is called then communication port becomes closed and all internal dynamic data will be released

from the memory. To activate communication with the programmer when the function F_Close_All has been used the F_Initialization function must be called first.

Syntax:

```
MSPPRG_API      INT_X  F_Close_All( void );
```

Return value:

- 0 - FALSE
- 1 - TRUE

Example:

```
F_Initialization();           //required API-Dll - initialization
.....
.....
F_Close_All;
.....
```

F_GetSetup

F_GetSetup - Get configuration setup from the programmer.

VALID FPA index - (1 to 64)

See F_ConfigSetup description for more details.

Syntax:

```
MSPPRG_API      INT_X  F_GetSetup( CONFIG_BLOCK *config );
```

Return value:

- 0 - FALSE
- 1 - TRUE
- 2 (0xFFFFFFFFE) - FPA_INVALID_NO

F_ConfigSetup

F_ConfigSetup - Setup programmer's configuration.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

The F_ConfigSetup can modify configuration of the programmer. When the F_ConfigSetup is called, then the structure data block is transferred from the software application to the programmer software. Current programmer setup can be read using function setup F_GetSetup. When data block is taken from the programmer, then part or all of the configuration data can be modified and returned back to programmer using F_ConfigSetup function. Configuration data structure and available data for all listed items in this structure are defined below. Listed name and indexes in the [] brackets are related to the F_SetConfig and F_GetConfig instructions.

Note: See the GangPro430-Dll.h header file for the list of the latest indexes, definitions etc.

Note Currently number of parameters used in configuration exceed the structure created for this goal. The configuration structure is not modified that allows to use the new API-DLL with customer's old application software without modifying it. The new API-DLL is back-compatible with the old ones. The new configuration data are accessible via F_SetConfig and F_GetConfig instructions.

Syntax:

```
MSPPRG_API      INT_X  F_ConfigSetup( CONFIG_BLOCK config );
```

Return value:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFF) - FPA_INVALID_NO
```

```
typedef struct
{
    INT_X  Interface;
    INT_X  uProcIndex;
    INT_X  PowerTargetEn;
    INT_X  spare_0;
    INT_X  ResetTimeIndex;
    INT_X  FlashEraseModeIndex;
    INT_X  EraseInfoSegmA;
    INT_X  EraseInfoSegmB;
    LONG_X EraseFlashStartAddr;
    LONG_X EraseFlashStopAddr;
    INT_X  FlashReadModeIndex;
    INT_X  ReadInfoSegmA;
    INT_X  ReadInfoSegmB;
    LONG_X ReadStartAddr;
```

```

LONG_X ReadStopAddr;
INT_X VerifyModeIndex;
INT_X BlowFuseEn;
INT_X ApplicationStartEn;
INT_X BeepEnable;
INT_X EraseInfoSegmC;
INT_X EraseInfoSegmD;
INT_X DefEraseMainMemEn;
INT_X ReadInfoSegmC;
INT_X ReadInfoSegmD;
INT_X JtagSpeedIndex;
INT_X VccIndex;
LONG_X CustomResetPulseTime;
LONG_X CustomResetIdleTime;
INT_X RstVccToggleTime;
INT_X TargetEnMask;
} CONFIG_BLOCK;

```

Indexes used by the **F_SetConfig** and **F_GetConfig** functions

CFG_INTERFACE	0	
CFG_MICROCONTROLLER	1	(See new function F_Set_MCU_Name(..))
CFG_POWERTARGETEN	2	
CFG_COMMSPEED	3	
CFG_RESEETIME	4	
CFG_FLASHERASEMODE	5	
CFG_ERASEINFOA	6	
CFG_ERASEINFOB	7	
CFG_ERASESTARTADDR	8	
CFG_ERASESTOPADDR	9	
CFG_FLASHREADMODE	10	
CFG_READINFOA	11	
CFG_READINFOB	12	
CFG_READSTARTADDR	13	
CFG_READSTOPADDR	14	
CFG_VERIFYMODE	15	
CFG_BLOWFUSE	16	
CFG_APPLSTARTEN	17	
CFG_BEEPEN	18	
CFG_ERASEINFOC	19	
CFG_ERASEINFOD	20	
CFG_DEFERASEMAINEN	21	
CFG_READINFOC	22	
CFG_READINFOD	23	

CFG_JTAGSPEEDINDEX	24
CFG_VCCINDEX	25
CFG_CUSTOMRESETPULSETIME	26
CFG_CUSTOMRESETIDLETIME	27
CFG_RSTVCCTOGGLETIME	28
CFG_TARGET_EN_INDEX	29
CFG_POP_UP_EN	30
CFG_JTAG_SPEED	31
CFG_BSL_ENH_ENABLE	32
CFG_BSL_ENH_INDEX	33
CFG_RETAIN_CAL_DATA_INDEX	34
CFG_RETAIN_DEF_DATA_INDEX	35
CFG_RETAIN_START_ADDR_INDEX	36
CFG_RETAIN_STOP_ADDR_INDEX	37
CFG_APPLPRGRUNTIME	38
CFG_RELEASEJTAGSTATE	39
CFG_RUNTIMETDIGENEN	40
CFG_RUNTIMETDIGENDIV	41
CFG_RUNTIMETDIGENPRESCALER	42
CFG_DCO1_CAL_ENABLE	43
CFG_DCO2_CAL_ENABLE	44
CFG_DCO3_CAL_ENABLE	45
CFG_DCO4_CAL_ENABLE	46
CFG_DCO5_CAL_ENABLE	47
CFG_DCO6_CAL_ENABLE	48
CFG_DCO7_CAL_ENABLE	49
CFG_DCO8_CAL_ENABLE	50
CFG_DCO1_CAL_FREQ_INDEX	51
CFG_DCO2_CAL_FREQ_INDEX	52
CFG_DCO3_CAL_FREQ_INDEX	53
CFG_DCO4_CAL_FREQ_INDEX	54
CFG_DCO5_CAL_FREQ_INDEX	55
CFG_DCO6_CAL_FREQ_INDEX	56
CFG_DCO7_CAL_FREQ_INDEX	57
CFG_DCO8_CAL_FREQ_INDEX	58
CFG_DCO_DEFINED_ADDR_EN	59
CFG_DCO_DEFINED_ADDRESS	60
CFG_RESERVED_1	61 //empty - for compatibility with FlashPro430
CFG_APPL_RESETVCCTIME	62
CFG_MASSERASE_AND_INFOA_EN	63
CFG_DCO_CONST_2XX_VERIFY_EN	64

CFG_DCOCAL_2XX_EN	65
CFG_FIRST_BSL_PASSW_INDEX	66
CFG_CS_TYPE_INDEX	69
CFG_CS_INIT_INDEX	70
CFG_CS_RESULT_INDEX	71
CFG_CS_CODE_OVERWIRIE_EN	72
CFG_CS_POLYNOMINAL	73
CFG_CS1_CALC_EN	74
CFG_CS1_START_ADDR	75
CFG_CS1_END_ADDR	76
CFG_CS1_RESULT_ADDR	77
CFG_CS2_CALC_EN	78
CFG_CS2_START_ADDR	79
CFG_CS2_END_ADDR	80
CFG_CS2_RESULT_ADDR	81
CFG_CS3_CALC_EN	82
CFG_CS3_START_ADDR	83
CFG_CS3_END_ADDR	84
CFG_CS3_RESULT_ADDR	85
CFG_CS4_CALC_EN	86
CFG_CS4_START_ADDR	87
CFG_CS4_END_ADDR	88
CFG_CS4_RESULT_ADDR	89
CFG_BSL_FLASH_WR_EN	90
CFG_BSL_FLASH_RD_EN	91
CFG_DCO_EXT_RESISTOR_EN	92

[CFG_INTERFACE 0]

Interface - JTAG/BSL interface selection

INTERFACE_JTAG	0	- JTAG Interface
INTERFACE_BSL	1	- BSL Interface
INTERFACE_SBW	3	- Spy-Bi-Wire - 2 wires

[CFG_MICROCONTROLLER 1] - supported, but not recommended -

(See new function **F_Set_MCU_Name(..)**)

- Microcontroller type selection

MSP430_ANY	0	- Any microcontroller - from F1xx or F4xxst
	1	- MSP430F110
	2	- MSP430F1101
	3	- MSP430F1101A
	4	- MSP430F1111A

etc.

See the latest MSP430 list and indexes in the FlashPro430 (GUI) software.

Run software -> list available under pull down menu

Setup-> MSP list

Also the MCU index and MCU names can be taken from the instruction F_Get_Device_Info(). See description of this instruction in this manual for details.

[CFG_POWERTARGETEN 2]

PowerTargetEn - Power Target from the Programming Adapter
0 - disable
1 - enable

[CFG_RESETTIME 4]

ResetTimeIndex - Reset Pulse time setup
RESET_50MS_INDEX 0 - USB->10ms, PP->50ms Reset Pulse time
RESET_100MS_INDEX 1 - 100 ms Reset Pulse time
RESET_200MS_INDEX 2 - 200 ms Reset Pulse time
RESET_500MS_INDEX 3 - 500 ms Reset Pulse time
RESET_CUSTOM_INDEX 4
RESET_TOGGLE_VCC_INDEX 5
RESET_SOFT_JTAG_INDEX 6

[CFG_FLASHERASEMODE 5]

FlashEraseModeIndex - Flash Write/Erase/Verify mode index
ERASE_NONE_MEM_INDEX 0 - Write and verify only. No erase flash
ERASE_ALL_MEM_INDEX 1 - Erase/Write/Verify all memory
ERASE_PRG_ONLY_MEM_INDEX 2 - Erase/Write/Verify program memory only
(without info segments A abd B)
ERASE_INFILE_MEM_INDEX 3 - Erase only segments used by the code taken
from the file.
Leave other segment unmodified.
ERASE_DEF_CM_INDEX 4 - Erase/Write/Verify only specified by data
EraseSegmA, B, C, D
EraseFlashStartAddr, EraseFlashStopAddr

[CFG_ERASEINFOA 6]

EraseSegmA - Info Segment A (0x1080-0x10fF) Erase/Write/Verify,
- in the MSP430F2xx - (0x10C0-0x10fF)
if FlashEraseModeIndex = ERASE_DEF_CM_INDEX

[CFG_ERASEINFOB 7]

EraseSegmB - Info Segment B (0x1000-0x107F) Erase/Write/Verify,
- in the MSP430F2xx - (0x1080-0x10BF)
if FlashEraseModeIndex = ERASE_DEF_CM_INDEX

[CFG_ERASEINFOC 19]

```

EraseSegmC          - MSP430F2xx only - Info Segment C (0x1040-0x107F)
                    Erase/Write/Verify,
                      if FlashEraseModeIndex = ERASE_DEF_CM_INDEX

[ CFG_ERASEINFOD 20 ]
EraseSegmD          - MSP430F2xx only - Info Segment B (0x1000-0x107F)
                    Erase/Write/Verify,
                      if FlashEraseModeIndex = ERASE_DEF_CM_INDEX
0      - Do not Write/Erase/Verify in segment A,B,C or D
1      - Write/Erase/Verify in segment A,B,C or D

[ CFG_ERASESTARTADDR 8 ]
EraseFlashStartAddr - Program Memory Start Address (0x1100-0x1FFFE)
                    Erase/Write/Verify,
                      if FlashEraseModeIndex = ERASE_DEF_CM_INDEX 0x1100 to 0x1FFFE

[ CFG_ERASESTOPADDR 9 ]
EraseFlashStopAddr  - Program Memory Stop Address (0x1101-0x1FFFF)
                    Erase/Write/Verify,
                      if FlashEraseModeIndex = ERASE_DEF_CM_INDEX 0x1101 to 0x1FFFF

[ CFG_FLASHREADMODE 10 ]
FlashReadModeIndex - Read Flash memory mode
  READ_ALL_MEM_INDEX 0 - Read all Flash memory
  READ_PRGMEM_ONLY_INDEX 1 - Read only program memory (0x1100-0x1FFFF)
  READ_INFOMEM_ONLY_INDEX 2 - Read only Info Flash memory (0x1000-0x10FF)
  READ_DEF_MEM_INDEX 3 - Read Flash memory defined by
                        ReadSegmA, B, C, D
                        ReadStartAddr and ReadStopAddr

[ CFG_READINFOA 11 ]
ReadSegmA          - Read from the Info Segment A

[ CFG_READINFOB 12 ]
ReadSegmB          - Read from the Info Segment B

[ CFG_READINFOC 22 ]
ReadSegmC          - Read from the Info Segment B

[ CFG_READINFOC 23 ]
ReadSegmD          - Read from the Info Segment B
                    if FlashReadModeIndex = READ_DEF_MEM_INDEX

[ CFG_READSTARTADDR 13 ]
ReadStartAddr      0x1100 to 0x1FFFE ;see above

[ CFG_READSTOPADDR 14 ]

```

```

ReadStopAddr          0x1101 to 0x1FFFF ;see above

[ CFG_VERIFYMODE 15 ]
VerifyModeIndex      - Program Verification method
    VERIFY_NONE_INDEX 0    - no verification
    VERIFY_STD_INDEX  1    - standard verification (read and verify)
    VERIFY_FAST_INDEX 2    - fast verification (calculate CS and verify)

[ CFG_BLOWFUSE 16 ]
BlowFuseEn          - Blow the security fuse - only using JTAG/SBW interface
    0                - disable
    1                - enable when called from F_Blow_Fuse()
    3                - enable when called from F_Blow_Fuse() or
                    F_Autoprogram(..)

[ CFG_APPLSTARTEN 17 ]
ApplicationStartEn  - reset and start the microcontroller's application
                    software when flash is successfully programmed
    APPLICATION_KEEP_RESET  0    - Hardware Reset Line permanent LOW
    APPLICATION_TOGGLE_RESET 1    - Hardware Reset (Pulse Low)
    APPLICATION_TOGGLE_VCC  2    - Toggle VCC Reset
    APPLICATION_JTAG_RESET  3    - JTAG software reset

[ CFG_BEEPEN 18 ]
BeepEnable          - beep on the end of flash programming
    0                - disable
    1                - enable

[ CFG_JTAGSPEEDINDEX 24 ]
JtagSpeedIndex      - valid for the USB model only
    JTAG_SPEED_1MB_INDEX  0 // JTAG 1 Mb/s, Spy-Bi-Wire->FAST
    JTAG_SPEED_400K_INDEX 2 // JTAG 400 kb/s, Spy-Bi-Wire->SLOW

[ CFG_VCCINDEX 25 ]
VccIndex            - valid for the USB-MSP430-FPA version 2.x and higher
    VCC_2V2_INDEX        0
    VCC_2V4_INDEX        1
    VCC_2V6_INDEX        2
    VCC_2V8_INDEX        3
    VCC_3V0_INDEX        4
    VCC_3V2_INDEX        5
    VCC_3V4_INDEX        6
    VCC_3V6_INDEX        7

[ CFG_CUSTOMRESETPULSETIME 26 ]
CustomResetPulseTime  value 1 to 1000 step 1 in milliseconds

```

valid only when the ResetTimeIndex = RESET_CUSTOM_INDEX

[CFG_CUSTOMRESETIDLETIME 27]
 CustomResetIdleTime value 1 to 2000 step 1 in milliseconds
 valid only when the ResetTimeIndex = RESET_CUSTOM_INDEX

[CFG_RSTVCCTOGGLETIME 28]
 RstVccToggleTime value 50 to 5000 step 1 in milliseconds
 valid only when the ResetTimeIndex = RESET_TOGGLE_VCC_INDEX

[CFG_TARGET_EN_INDEX 29]
 TARGET_1_MASK 0x01
 TARGET_2_MASK 0x02
 TARGET_3_MASK 0x04
 TARGET_4_MASK 0x08
 TARGET_5_MASK 0x10
 TARGET_6_MASK 0x20
 mask can be defined as a logical sum of
 TARGET_1_MASK | TARGET_2_MASK TARGET_6_MASK

[CFG_POP_UP_EN 30]
 PopUpEnable - enable or disable to display pop-up message in run time
 0 - warning message popup disable
 1 - enable all
 2 - disable all

[CFG_BSL_ENH_ENABLE 32] - BSL mode (valid for BSL version 2.0 and higher)
 (See FlashPro430 Manual - chapter 9)
 0 - disable
 1 - enable

[CFG_BSL_ENH_INDEX 33]
 (See FlashPro430 Manual - chapter 9)
 BSL_ENH_DISABLE 0 Disable access via BSL
 BSL_ENH_NONE 1 Do not erase flash if the BSL password is invalid
 (allows to make a next try)
 BSL_ENH_ERASE 2 Erase whole flash if the BSL password is invalid
 (one try only)

[CFG_RETAIN_CAL_DATA_INDEX 34]
 (See FlashPro430 Manual - chapter 6)
 0 - Disable retain the DCO Calibration Data for the F2xx family.
 1 - Enable retain the DCO Calibration Data for the F2xx family. Data are
 saved in the INFO flash memory at location 0x10F8 to 0x10FF

[CFG_RETAIN_DEF_DATA_INDEX 35]
 (See FlashPro430 Manual - chapter 6)
 0 - Disable retain the user defined data in flash.

1 - Enable retain the user defined data in flash. Data specified in the user defined location (see indexes below) will be restored after erase and program.

[CFG_RETAIN_START_ADDR_INDEX 36]
start address - start address for the user defined retain data (protected data) in flash.
Size of protected data can not exceed 256 bytes.

[CFG_RETAIN_STOP_ADDR_INDEX 37]
stop address - stop address for the user defined retain data (protected data) in flash.
Size of protected data can not exceed 256 bytes.

[CFG_APPLPRGRUNTIME 38]
data 0 - unlimited time
1 to 120 seconds - (limited time)

[CFG_RELEASEJTAGSTATE 39]
DEFAULT_JTAG_3ST 0
DEFAULT_JTAG_HI 1
DEFAULT_JTAG_LO 2

[CFG_RUNTIMETDIGENEN 40]
0-disable, 1-enable

[CFG_RUNTIMETDIGENDIV 41]
data 1 to 255

[CFG_RUNTIMETDIGENPRESCALER 42]
RUNTIMETDIPRESCALER_6MHZ 0
RUNTIMETDIPRESCALER_2MHZ 1

[CFG_DCO1_CAL_ENABLE 43]
[CFG_DCO2_CAL_ENABLE 44]
[CFG_DCO3_CAL_ENABLE 45]
[CFG_DCO4_CAL_ENABLE 46]
[CFG_DCO5_CAL_ENABLE 47]
[CFG_DCO6_CAL_ENABLE 48]
[CFG_DCO7_CAL_ENABLE 49]
[CFG_DCO8_CAL_ENABLE 50]
0-disable, 1-enable

[CFG_DCO1_CAL_FREQ_INDEX 51]
[CFG_DCO2_CAL_FREQ_INDEX 52]
[CFG_DCO3_CAL_FREQ_INDEX 53]

[CFG_DCO4_CAL_FREQ_INDEX 54]
 [CFG_DCO5_CAL_FREQ_INDEX 55]
 [CFG_DCO6_CAL_FREQ_INDEX 56]
 [CFG_DCO7_CAL_FREQ_INDEX 57]
 [CFG_DCO8_CAL_FREQ_INDEX 58]
 range from 100 to 16000 (100kHz to 16 MHz)
 see the MSP430 data sheet for acceptable freq range.

[CFG_DCO_DEFINED_ADDR_EN 59]
 0-disable, 1-enable

[CFG_DCO_DEFINED_ADDRESS 60]
 DCO constants address in flash
 Default -> 0x10F0

[CFG_APPL_RESETVCCTIME 62]
 ApplResetVccTime value 50 to 5000 step 1 in miliseconds
 valid only when the ApplicationStartEn = APPLICATION_TOGGLE_VCC

[CFG_MASSERASE_AND_INFOA_EN 63]
 Valid only for MSP430F2xx
 0-disable - INFO-A erase disabled when the mass memory erase is enabled.
 1-enable - INFO-A erase enabled when the mass memory erase is enabled.

[CFG_DCO_CONST_2XX_VERIFY_EN 64]
 Valid only for MSP430F2xx

 0-disable, 1-enable

[CFG_DCOCAL_2XX_EN 65]
 Valid only for MSP430F2xx
 0-disable, 1-enable -> make a DCO calibration if DCO constants are
 invalid.

[CFG_FIRST_BSL_PASSW_INDEX 66]
 0 - FIRST_BSL_PASSW_DEFAULT
 1 - FIRST_BSL_PASSW_FROM_PASSW_FILE
 2 - FIRST_BSL_PASSW_FROM_CODE_FILE
 3 - FIRST_BSL_PASSW_EMPTY

[CFG_CS_TYPE_INDEX 69]
 0 - " n o n e "
 1 - "Arithmetic sum (8b / 16b)"
 2 - "Arithmetic sum (8b / 32b)"
 3 - "Arithmetic sum (16b / 16b)"

```

4 - "Arithmetic sum          (16b / 32b )"
5 - "CRC16 (Poly = 0x11021)  ( 8b / 16b )"
6 - "CRC16 defined polynomial ( 8b / 16b )"
7 - "CRC32 (Poly = 0x04C11DB7) ( 8b / 32b )"
8 - "CRC32 defined polynomial ( 8b / 32b )"

[ CFG_CS_INIT_INDEX          70 ]
  0 - CS_INIT_VALUE_0_INDEX
  1 - CS_INIT_VALUE_1_INDEX
  2 - CS_INIT_VALUE_ADDR_INDEX

[ CFG_CS_RESULT_INDEX        71 ]
  0 - "As Is"
  1 - Inverted

[ CFG_CS_CODE_OVERWIRIE_EN   72 ]
  0 - disable
  1 - enable

[ CFG_CS_POLYNOMINAL         73 ]
  - polynomial value

[ CFG_CS1_CALC_EN            74 ]
[ CFG_CS2_CALC_EN            78 ]
[ CFG_CS3_CALC_EN            82 ]
[ CFG_CS4_CALC_EN            86 ]
  0 - disable
  1 - enable

[ CFG_CS1_START_ADDR         73 ]
[ CFG_CS2_START_ADDR         79 ]
[ CFG_CS3_START_ADDR         83 ]
[ CFG_CS4_START_ADDR         87 ]
  - Start address value

[ CFG_CS1_END_ADDR           76 ]
[ CFG_CS2_END_ADDR           80 ]
[ CFG_CS3_END_ADDR           84 ]
[ CFG_CS4_END_ADDR           88 ]
  - End address value

[ CFG_CS1_RESULT_ADDR        77 ]
[ CFG_CS2_RESULT_ADDR        81 ]
[ CFG_CS3_RESULT_ADDR        85 ]
[ CFG_CS4_RESULT_ADDR        89 ]
  - Result address value

```

```
[ CFG_BSL_FLASH_WR_EN          90 ]
  Sum of enabled BSL sectors
  - 0x01 - BSL Segment 0 (0x1000-0x11FF)
  - 0x02 - BSL Segment 1 (0x1200-0x13FF)
  - 0x04 - BSL Segment 2 (0x1400-0x15FF)
  - 0x08 - BSL Segment 3 (0x1600-0x17FF)
```

```
[ CFG_BSL_FLASH_RD_EN          91 ]
  Sum of enabled BSL sectors
  - 0x01 - BSL Segment 0 (0x1000-0x11FF)
  - 0x02 - BSL Segment 1 (0x1200-0x13FF)
  - 0x04 - BSL Segment 2 (0x1400-0x15FF)
  - 0x08 - BSL Segment 3 (0x1600-0x17FF)
```

```
[ CFG_DCO_EXT_RESISTOR_EN      92 ]
  0 - disable
  1 - enable
```

Note: See the `GangPro430-Dll.h` header file for the list of the latest indexes, definitions etc.

Example:

Example below shows the method of modification of the programmers configuration setup. First the current setup from the programmer is uploaded to the application, after that some of the parameters have been modified and at the end the modified setup is returned back to the programmer.

```
CONFIG_BLOCK      config;          //programmer's configuration data
.....

F_GetSetup( &config );
           //API-DLL - get configuration from the programmer
config.Interface = INTERFACE_JTAG;
           //select JTAG interface
config.BlowFuseEn = 0;
           //disable fuse blow option
config.FlashEraseModeIndex = ERASE_ALL_MEM_INDEX;
           //select all memory erase option
F_ConfigSetup( config );
           //API-DLL - setup configuration in the programmer
```

The same configuration can be read/set using the by the **F_SetConfig** function as follows

```
F_SetConfig( CFG_INTERFACE, INTERFACE_JTAG );
F_SetConfig( CFG_BLOWFUSE, 0 );
```

```
F_SetConfig( CFG_FLASHERASEMODE, ERASE_ALL_MEM_INDEX );
```

F_SetConfig

F_SetConfig - Setup one item of the programmer's configuration.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Similar to the **F_ConfigSetup**, but only one item from the **CONFIG_BLOCK** structure is modified.

Syntax:

```
MSPPRG_API INT_X F_SetConfig( INT_X index, LONG_X data );
```

See index list in the **F_ConfigSetup** for details.

Return value:

```
0 - FALSE  
1 - TRUE  
-2 - FPA_INVALID_NO
```

Example:

```
.....  
.....  
    F_SetConfig( CFG_MICROCONTROLLER, config.uProcIndex );  
.....
```

F_GetConfig

F_GetConfig - Get one item of the programmer's configuration.
VALID FPA index - (1 to 64)

Similar to the **F_GetSetup**, but only one item from the **CONFIG_BLOCK** structure is read.

Syntax:

```
MSPPRG_API LONG_X F_GetConfig( INT_X index );
```

Index's list - see **F_SetConfig**

Return value:

Requested setup parameter;
-2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
.....  
.....  
    config.Interface = F_GetConfig( CFG_INTERFACE );  
.....
```

F_Set_MCU_Name

F_Set_MCU_Name - Set microcontroller type.

VALID FPA index - (1 to 64)

The F_Set_MCU_Name(..) replaced the old function

```
F_SetConfig( CFG_MICROCONTROLLER, MCU_index );
```

The MCU name must be entered exactly with the same name as it is in the GUI software - MCU type pull down menu.

Syntax:

```
MSPPRG_API      INT_X  F_Set_MCU_Name( char * MCU_name );
```

Return value:

>= 0 - Index of the selected MCU
-1 - error - invalid MCU name.

Example:

```
.....  
    F_Set_MCU_Name( "MSP430F5438" );  
.....
```

F_Get_Device_Info

F_Get_Device_Info - Get information related to selected microcontroller.

VALID FPA index - (1 to 64)

Syntax:

```
MSPPRG_API      INT_X  F_Get_Device_Info( INT_X index );
```

where index:

DEVICE_NAME	0
DEVICE_NAME_SIZE	20
DEVICE_MAIN_FLASH_START_ADDR	20
DEVICE_MAIN_FLASH_END_ADDR	21
DEVICE_INFO_START_ADDR	22
DEVICE_INFO_SEGM_SIZE	23
DEVICE_NO_INFO_SEGMENTS	24
DEVICE_RAM_SIZE	25

Return value:

```
-1 (0xFFFFFFFF)   - invalid data
-2 (0xFFFFFFFFE) - FPA_INVALID_NO
or
index - 0 to 19  ->   device name - char by char starting from index->0
                    => M eg.  MSP430F149

index 0 -> 'M'
index 1 -> 'S'
index 2 -> 'P'
index 3 -> '4'
index 4 -> '3'
index 5 -> '0'
index 6 -> 'F'
index 7 -> '1'
index 8 -> '4'
index 9 -> '9'
index 10 -> 0x0000  -> end of string
index 11 to 19 -> after end of string - irrelevant data.
index 20 -> MAIN Flash Start Address  eg 0x1100  (for F149)
index 21 -> MAIN Flash End  Address  eg 0xFFFF  (for F149)
index 22 -> INFO start address        eg 0x1000
index 23 -> INFO segment size         eg 0x0080  (for F149)
index 24 -> No of INFO segments       eg 0x0002  (for F149)
index 25 -> RAM size                  eg 0x0800  (for F149)
```

Note: The device info is related to selected microprocessor. Desired index processor should be first set in the configuration using *F_SetConfig(CFG_MICROCONTROLLER, uP_index)*;

Below is an example of the procedure that can take names of all supported devices by the API-DLL. The max size can be tested from the API-DLL, until device name is empty when the microprocessor

index is incremented from the zero up to max value. In the example below is assumed that the max number of supported devices is 300, however this value can be dynamically modified if required. In the procedure below the names and uP index are saved in the `DEVICELIST` structure, where the name and index pair are kept in the same `DEVICELIST DeviceList[]` element. When the `DeviceList[]` is created, then all names are kept in the alphabetic order. Microprocessor name and corresponded microprocessor index used by the API-DLL can be taken from following elements:

```

        Microprocessor name (string)      <- DeviceList[k].name;
        Microprocessor index (int)       <- DeviceList[k].index;

#include "MSPPrg-Dll.h"
#define   MAX_NO_OF_DEVICES   300

typedef struct
{
    char name[DEVICE_NAME_SIZE];
    int  index;
}DEVICELIST;

DEVICELIST DeviceList[MAX_NO_OF_DEVICES];

.....
    response = F_OpenInstancesAndFPAs( "*# *" ); //get first FPA
    if( response > 0 )
    {
        response = F_Set_FPA_index( 1 );
        response = F_Initialization();
        get_device_names();           //now you can read data from API-DLL
    }
.....

int  get_device_names( void )
{
    int n,k, st, index_bak, max_up_index;
    DEVICELIST tmp;

    tmp.index = 0;  *tmp.name = '\0';

    index_bak = F_GetConfig( CFG_MICROCONTROLLER ); //get current uP index
    for(k=0; k<MAX_NO_OF_DEVICES; k++)
        DeviceList[k] = tmp;           //clr device list

    max_up_index = 0;
    for(k=0; k<MAX_NO_OF_DEVICES; k++)

```



```

{
  F_SetConfig( CFG_MICROCONTROLLER, k );          //set new uP index
  for( n = 0; n<DEVICE_NAME_SIZE; n++ )
  {
    DeviceList[k].name[n] = char(0xFF & F_Get_Device_Info( DEVICE_NAME+n ));
  }
  if( DeviceList[k].name[0] == 0 ) break;          //break if name is empty
  DeviceList[k].index = k;
  max_up_index = k;
}
F_SetConfig( CFG_MICROCONTROLLER, index_bak ); //restore uP index

//sort names in the table from min to max.
if( max_up_index > 0 )
for( k=0; k<max_up_index; k++ )
{
  st = FALSE;
  for( n=1; n <= max_up_index; n++ )
  {
    if( strcmp( DeviceList[n-1].name, DeviceList[n].name ) >=0 )
    {
      st = TRUE;
      tmp = DeviceList[n-1];
      DeviceList[n-1] = DeviceList[n];
      DeviceList[n] = tmp;
    }
  }
  if( st == FALSE) break;
}
return( max_up_index );
}

```

F_DisSetup

F_DisSetup - Copy programmer's configuration to report message buffer in text form.
VALID FPA index - (1 to 64)

Syntax:

```
MSPPRG_API      INT_X   F_DisSetup( void );
```

Return value:

```
1 - TRUE;
-2 (0xFFFFFFFF) - FPA_INVALID_NO
```

Example:

```
.....
.....
F_DispSetup();
Disp_report_message();
//see F_ReportMessage or F_GetReportMessage for details
.....
```

F_ReportMessage, F_Report_Message

F_ReportMessage - Get the last report message from the programmer.

or **F_Report_Message**

VALID FPA index - (1 to 64)

When any of the DLL functions is activated, a message is created and displayed on the dynamically created programmer's dialog box. At the end of execution the dialog box is closed and function returns back to the application program. Reported message is closed as well. The last report message can be read by application program using F_ReportMessage function. When F_ReportMessage is called, then report message up to 1000 characters is imported from the programmer software to the application software. Make sure to declare characters string length no less than 1000 characters. When F_ReportMessage is called then at the end the internal report message buffer in the programmer software is cleared. When F_ReportMessage is not called after every communication with the target device, then the report message will collect all reported information up to 1000 last characters.

Syntax:

```
MSPPRG_API void F_ReportMessage( char * text );
MSPPRG_API char* F_Report_Message( void );
```

note: **F_Report_Message** is available only with the Multi-FPA API-DLL.

Return value:

none

Example:

```
char text[1002];
.....
.....
    F_ReportMessage( text );
.....
```

Example below shows how to take a message and display it in the scrolling box. The Edit box with the ID e.g. IDC_REPORT must be created first.

```
.....
Cstring Message = "";
.....

void CMspPrgDemoDlg::Disp_report_message()
{
    char text[1002];                //must be min. size - 1000
    F_ReportMessage( text );        //API-Dll - get last report message
    Message = text;
    SetDlgItemText(IDC_REPORT, Message.GetBuffer(Message.GetLength()));
    CEdit* pEdit = (CEdit*) GetDlgItem(IDC_REPORT);
    pEdit->LineScroll(pEdit->GetLineCount(), 0);
    UpdateWindow();
}
```

F_GetReportMessageChar

F_GetReportMessageChar - Get one character of the the last report message from the programmer.

VALID FPA index - (1 to 64)

See comment for the **F_ReportMessage** function.

F_GetReportMessageChar allows to get character by character from the report message buffer. This function is useful in the Visual Basic application, where all message can not be transfered via pointer like it is possible in the C++ application.

Syntax:

```
MSPPRG_API      char F_GetReportMessageChar( INT_X index );
```

Return value:

Requested character from the Report Message buffer. 1 - TRUE

Example:

```
char text[1002];
INT_X k;
.....
.....
for( k = 0; k< 1000; k++ )
    text[k] = F_GetReportMessageChar( k );
.....
```

Example below shows how to take a message and display it in the scrolling box. The Edit box with the ID e.g. IDC_REPORT must be created first.

```
.....
Cstring Message = "";
.....

void CMspPrgDemoDlg::Disp_report_message()
{
    char text[1002];           //must be min. size - 1000
    INT_X k;
    for( k = 0; k< 1000; k++ )
        text[k] = F_GetReportMessageChar( k );
    Message = text;
    SetDlgItemText(IDC_REPORT, Message.GetBuffer(Message.GetLength()));
    CEdit* pEdit = (CEdit*) GetDlgItem(IDC_REPORT);
    pEdit->LineScroll(pEdit->GetLineCount(), 0);
    UpdateWindow();
}
```

F_DLLTypeVer

F_DLLTypeVer - Get information about DLL software type and software revision.
VALID FPA index - (1 to 64)

F_DLLTypeVer function returns integer number with DLL ID and software revision version and copying text message to report message buffer about DLL ID and software revision. Text content can be downloaded using one of the following functions

`F_GetReportMessageChar(index)`
 or `F_ReportMessage(text)`

Syntax:

```
MSPPRG_API      INT_X F_DLLTypeVer( void );
```

Return value:

```
VALUE = (DLL ID) | ( 0x0FFF & Version)
DLL ID = 0x1000   - Single-DLL for the FlashPro430 - Parallel Port
DLL ID = 0x2000   - Single-DLL for the FlashPro430 - USB
DLL ID = 0x3000   - Single-DLL for the GangPro430 - USB
DLL ID = 0x6000   - Multi-DLL for the FlashPro430
DLL ID = 0x7000   - Multi-DLL for the GangPro 430
Version = (0x0FFF & VALUE)
-2 (0xFFFFFFFFE) - FPA_INVALID_NO
```

Example:

```
INT_X id;
.....
.....
    id = F_DLLTypeVer();
    Disp_report_message();
        //see F_ReportMessage or F_GetReportMessage for details
.....
```

F_ConfigFileLoad, F_Config_FileLoad

F_ConfigFileLoad - Modify programmer’s configuration setup according to data taken
 or **F_Config_FileLoad** from the specified configuration file.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) *executed sequentially.*

The **F_ConfigFileLoad** function can download the programmer setup from the external setup file. Setup file can be created using standard MSP430 Flash Programmer software. When setup from the file is downloaded, then old configuration setup is overwritten. New setup can be modified using **F_GetSetup** and **F_ConfigSetup** functions.

Location path and file name of the config file must be specified.

Syntax:

```
MSPPRG_API      INT_X  F_ConfigFileLoad( char * filename );  
MSPPRG_API      INT_X  F_ConfigFileLoad( Cstring filename );
```

filename - configuration file name including path, file name and extension

Return value:

```
0 - FALSE  
1 - TRUE  
(0xFFFe & info) | state  
where state is defined as follows:  
    0 - FALSE  
    1 - TRUE  
   -2 (0xFFFFFFFF) - FPA_INVALID_NO
```

info is defined as follows:
error -> OPEN_FILE_OR_READ_ERR

note: **F_Config_FileLoad** is available only when the multi-FPA dll is used.

Configuration file is a standard text file with the parameters name and value. Below is a list of used parameters. See *F_ConfigSetup* for contents of the listed parameters.

Specified parameters in the configuration file can be listed in any order. Configuration file can specified few or all parameters. Specified parameter not listed above will be ignored.

Parameter name and value must be separated by minimum one white character like space or tabulation. File name and full path cannot have a white space. All characters after semicolon up to the end of line are treated as a comments.

Below is an example of configuration file. Note, that not all parameters are listed in the configuration example.

Example:

```
st = F_ConfigFileLoad( "c:\test\configfile.cfg" );  
if(( st & 1 ) == TRUE )  
{  
    .....
```

```

}
else
{
    Info = st & 0xFFFFE;
    .....
    .....
}

```

F_Power_Target

F_Power_Target - Turn ON or OFF power from programming adapter to target device.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Function F_Power_Target switches ON or OFF power from the programming adapter to the target device.

Note: PowerTargetEnflag must be set to TRUE (1) in the configuration setup to switch the power from the programming adapter ON.

Syntax:

```
MSPPRG_API      INT_X  F_Power_Target( INT_X OnOff );
```

Return value:

```

0 - FALSE
1 - TRUE
-2 (0xFFFFFFFF) - FPA_INVALID_NO

```

Example:

```

.....
F_Power_Target( 1 );           // Turn Power ON
.....
F_Power_Target( 0 );          // Turn Power OFF
.....

```

F_Reset_Target

F_Reset_Target - Generate short RESET pulse on the target's device RESET line.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Function F_Reset_Target resets target device and target device's application program can start. Length of the RESET pulse time is specified by ResetTimeIndex in configuration setup. See F_ConfigSetup description for details.

Syntax:

```
MSPPRG_API          INT_X F_Reset_Target( void );
```

Return value:

- 0 - FALSE
- 1 - TRUE
- 2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
.....  
F_Reset_Target( void );  
.....
```

F_Get_Targets_Result

F_Get_Targets_Result - Get target's devices result mask. When particular bit is set then the selected target device result is positive, otherwise failed.

VALID FPA index - (1 to 64)

Syntax:

```
MSPPRG_API          INT_X F_Get_Targets_Result( void );
```

Return value:

- INT_X - execution result for up to sic target devices.
- Bit - 0 - false
- bit - set - true
- 2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
INT_X st;  
.....  
.....
```



```

F_SetConfig( CFG_TARGET_EN_INDEX, 0x0F );
//enable to program four target
//devices - no 1,2,3 and 4
F_Autoprogram(0);
st = F_Get_Targets_Result(); //if st == 0x0F then all four
//selected target devices have been
//programmed and verified
//if st == 0x0B then programming of the target device no 3 failed.
.....

```

F_Get_Active_Targets_Mask

F_Get_Active_Targets_Mask - Get mask of the active targets. When particular bit is set then the selected target is active, otherwise target is not present, disabled by user, not active or disabled by software during access test.

VALID FPA index - (1 to 64)

Syntax:

```
MSPPRG_API      INT_X F_Get_Active_Targets_Mask( void );
```

Return value:

```

INT_X - execution result for up to sic target devices - 0x00 to 0x3F.
Bit - 0 - false
bit - set - true
-2 (0xFFFFFFFF) - FPA_INVALID_NO

```

Example:

```

INT_X st;
.....
.....
F_SetConfig( CFG_TARGET_EN_INDEX, 0x0F );
//enable to program four target
//devices - no 1,2,3 and 4
F_Open_Target_Device();

st = F_Get_Active_Targets_Mask(); //if st == 0x0F then communication
//with all four selected target
//devices have been activated.
//if st == 0x0B then access to the target device no '3' failed.

```

```

.....
.....

F_Close_Target_Device();
st = F_Get_Active_Targets_Mask();
    // st will be 0. Access to all devices have been closed.
.....

```

F_Get_Targets_Vcc

F_Get_Targets_Vcc - Get Vcc in [mV] supplied target device.

VALID FPA index - (1 to 64)

Syntax:

```
MSPPRG_API INT_X F_Get_Targets_Vcc( void );
```

Return value:

```

INT_X - Vcc in milivolts e.g 3000 -> 3.0 V
        or (-1) if USB-FPA is not active.
        -2 (0xFFFFFFFF) - FPA_INVALID_NO

```

F_Set_fpa_io_state

F_Set_fpa_io_state - Set state of the Reset, Vcc and Jtagl lines

VALID FPA index - (1 to 64) or 0 (ALL FPAs) *executed sequentially.*

Syntax:

```
MSPPRG_API INT_X F_Set_fpa_io_state( BYTE jtag, BYTE reset, BYTE Vccon void );
```

```

jtag ->  TMS, TCK, TDI (output from FPA)
        0 -> DEFAULT_JTAG_3ST
        1 -> DEFAULT_JTAG_HI
        2 -> DEFAULT_JTAG_LO
reset -> 0 -> output from FPA RESET - LO
        1 -> output from FPA RESET - HI
VccOn -> 0 -> output Vcc from FPA - OFF
        1 -> output Vcc from FPA - ON
        (level 2.2V to 3.6V set in CFG_VCCINDEX )

```

4.3 Data Buffers access instructions

All data coming to of from target device can be saved in the temporary buffers (see Figure 4.5) located inside the API-DLL. The data saved in these buffers can be copied to target devices using an encapsulated or sequential functions. When the full block of data is ready to be saved (eg. code data), then the part of the data buffers can be modified by adding some unique data like serial numbers, calibration data etc. to each target before executing the flash programming process. Data buffers can be modified at any time, as long as the *F_OpenInstancesAndFPAs(..)* and *F_Initialization()* have been executed successfully. When more then one FPA are used then it is recommended to use only an executable instructions uses the data buffers for read and write. Results from each targets are saved in a Gang Read Data buffers - one Gang Read Buffer per one API-DLL. When the simultaneous process is done, then the content from each buffers can be individually read. The API-DLL contains four buffers (see Figure 4.5) - **Code**, **Password**, **Gang Write Data** and **Gang Read Data** buffers. Contents for the **Code** and **Password** buffers can be taken from the files, or data can be written directly to the specified buffer location. Data to the **Gang Write Data** and **Gang Read Data** buffers can be written/read directly. Gang Buffers contains six data buffers - dedicated one data buffer per one target device. The FLASH memory can be programmed using contents taken from the **Code** buffer or from the **Gang Write Data** buffers. Data to RAM, registers, I/O (seen as RAM) can be taken from **Gang Write Data** buffers only. Contents from RAM, registers, I/O and flash are saved in **Gang Read Data** buffers.

Note: The **Code** buffer contains two items inside - data and flag in each address location. Data is related to the written value 0 to 0xFF, while flag - *used* or *empty* informs is the particular byte is used and should be programmed, verified etc, or if it is empty and should be ignored even if data is 0xFF. All flags are cleared when the new code from the file is downloaded, or if the *F_Clr_Code_Buffer()* instruction is used.

Below are listed the data buffers access between an application and API-DLL buffers instruction.

F_ReadCodeFile, F_Read_CodeFile

F_ReadCodeFile - Read code data from the file and download it to internal buffer.

or **F_Read_CodeFile**

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Function *F_ReadCodeFile* downloads code from the file to internal memory buffer. Code file format and file name and location path of the desired file must be specified. Three file formats are supported

- Texas Instruments text format, Motorola *.s19 format and Intel *.hex format. When file is downloaded then contents of this file is analysed. Only code memory location valid for the MSP430 microcontroller family will be downloaded to the internal Code buffer. Any code data located outside memory space of the MSP430 microcontroller will be ignored and warning message will be created. When the ***F_ReadCodeFile*** function is used then the full **Code** buffer is filled with data 0xFF and all flags are cleared (***empty*** flag) first. When the valid data are taken from the code buffer, the data is saved in buffer and flag modified from ***empty*** to ***used***.

Syntax:

```
MSPPRG_API      INT_X  F_ReadCodeFile( int file_format, char * FileName );
MSPPRG_API      INT_X  F_Read_CodeFile( int file_format, CString FileName );
```

file_format:

```
FILE_TI_FORMAT      (1) for TI (*.txt) format
FILE_MOTOROLA_FORMAT (2) for Motorola (*.s19, *.s28 or *.s37)
FILE_INTEL_FORMAT   (3) for Intel (*.hex)
FILE_IAR_D43_FORMAT (4) for IAR (UBROF9) *.d43 format
or FILE_DEBUG_A43_FORMAT (5) for IAR HEX or Motorola debug format
```

FileName: file name including path, file name and extension

note: **F_Read_CodeFile** is available only when the Multi-FPA dll is used.

Return value:

```
(0xFFFFe & info) | state
where state is defined as follows:
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFF) - FPA_INVALID_NO
info is defined as follows:
warning -> CODE_IN_ROM
          CODE_IN_RAM
          CODE_OUT_OF_FLASH
          CODE_OVERWRITTEN
error ->  INVALID_CODE_FILE
          OPEN_FILE_OR_READ_ERR
```

Example:

```
int st;
.....
st = F_ReadCodeFile( FILE_TI_FORMAT, "c:\test\demofile.txt" );
if(( st & 1 ) == TRUE )
```

```

{
    .....
}
else
{
    if ( st & CODE_IN_ROM ) {.....}
    if ( st & CODE_OUT_OF_FLASH ) {.....}
    if ( st & INVALID_CODE_FILE ) {.....}
    if ( st & OPEN_FILE_OR_READ_ERR ) {.....}
    .....
    .....
}

```

F_Get_CodeCS

F_Get_CodeCS - Read code from internal code buffer and calculate the check sum.
VALID FPA index - (1 to 64).

Syntax:

```
MSPPRG_API LONG_X F_Get_CodeCS( int index );
```

index - index of the desired code

Index = 1 - Calculate check sum of the code from internal code buffer.

Other Index values - reserved for the future option.

Return value:

Calculated check sum or

-2 (0xFFFFFFFF) - FPA_INVALID_NO

F_ReadPasswFile, F_Read_PasswFile

F_ReadPasswFile - Read code password data from the file and download it to internal buffer.

or F_Read_PasswFile

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Function F_ReadPasswFile downloads part of the code from the file to internal memory buffer. From the code file only data related to the password data (location 0xFFE0 to 0xFFFF) are stored in the password memory buffer. All other data is ignored. Code file format and file name and location path of the desired file must be specified. Three file formats are supported - Texas Instruments text

format, Motorola *.s19 format and Intel *.hex format.

Syntax:

```
MSPPRG_API INT_X F_ReadPasswFile( INT_X file_format, char * FileName );  
MSPPRG_API INT_X F_Read_PasswFile( INT_X file_format, CString FileName );
```

file_format -> specify code file format - TI (*.txt), Motorola (*.s19, *.s28, *.s37), Intel (*.hex), IAR UBROF9 (*.d43) or IAR debug (*.a43) format

```
FILE_TI_FORMAT          (1) for TI (*.txt) format  
FILE_MOTOROLA_FORMAT    (2) for Motorola (*.s19, *.s28 or *.s37)  
FILE_INTEL_FORMAT       (3) for Intel (*.hex)  
FILE_IAR_D43_FORMAT     (4) for IAR (UBROF9) *.d43 format  
or FILE_DEBUG_A43_FORMAT (5) for IAR HEX or Motorola debug format
```

FileName -> full file name including path, file name and extention

note: F_Read_PasswFile is available only when the Multi-FPA dll is used.

Return value:

```
(0xFFFe & info) | state  
where state is defined as follows:  
0 - FALSE  
1 - TRUE  
-2 (0xFFFFFFFF) - FPA_INVALID_NO  
info is defined as follows:  
error -> INVALID_CODE_FILE  
OPEN_FILE_OR_READ_ERR  
PASSWORD_NOT_FOUND
```

Example:

```
st = F_ReadPasswFile( FILE_TI_FORMAT, "c:\test\demofile.txt" );  
if(( st & 1 ) == TRUE )  
{  
    .....  
}  
else  
{  
    Info = st & 0xFFFE;  
    .....  
    .....  
}
```

F_Clr_Code_Buffer

F_Clr_Code_Buffer - Clear content of the *Code* buffer.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Function fill the full Code buffer with data *0xFF* and clear all flags to *empty* value.

Syntax:

```
MSPPRG_API INT_X F_Clr_Code_Buffer( void );
```

Return value:

```
0 - FALSE  
1 - TRUE  
-2 - FPA_INVALID_NO
```

Example:

```
.....  
F_Clr_Code_Buffer();  
.....
```

F_Put_Byte_to_Code_Buffer

F_Put_Byte_to_Code_Buffer - Write code data to Code buffer.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Instruction allows to write contents of the code to code buffer instead using the *F_ReadCodeFile* instruction. Contents of the downloaded code data can be modified or filled with the new data, if code buffer has been cleared first (using *F_Clr_Code_Buffer* function).

Instruction write the data to *Code* buffer in specified address location and set the *used* flag in that location.

Note: Writing the 0xFF to the specified location where the other then the 0xFF data was located do not remove the contents from the buffer in fully. The new data (0xFF) will be written to **Code** buffer location, but flag still will be set to *used*. Use the *F_Clr_Code_Buffer()* instruction to fully clear the **Code** buffer before writing the new data block.

Syntax:

```
MSPPRG_API INT_X F_Put_Byte_to_Code_Buffer( LONG_X address,
```

BYTE data);

Parameters value:

code address - 0x1000 to 0x1FFFF
data - 0x00 to 0xFF

Return value:

0 - FALSE
1 - TRUE
-2 - FPA_INVALID_NO

Example:

```
BYTE code[0x20000];  
.....  
F_Clr_Code_Buffer();  
for( address = 0x1000; address < 0x20000; address ++ )  
{  
    F_Put_Byte_to_Code_Buffer( address, code[address]);  
}  
.....
```

F_Get_Byte_from_Code_Buffer

F_Get_Byte_from_Code_Buffer - Read code data from code buffer.
VALID FPA index - (1 to 64)

Instruction allows to read or verify contents of the code from code buffer. Data returns value 0x00 to 0xFF if in the particular **Code** buffer location the flag is set to *used*, otherwise return value **-1** (minus one) if data is empty.

Syntax:

```
MSPPRG_API INT_X F_Get_Byte_from_Code_Buffer( LONG_X address );
```

Parameters value:

code address - 0x1000 to 0x1FFFF

Return value:

0x00 to 0xFF - valid code data
-1 (0xFFFFFFFF) - code data not initialized on particular address
-2 (0xFFFFFFFFE) - FPA_INVALID_NO

F_Put_Byte_to_Password_Buffer

F_Put_Byte_to_Password_Buffer - Write code data to password buffer.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Instruction allows to write contents of the code to code buffer instead using the F_ReadPasswordFile instruction.

Note: All 32 bytes of the password data must be written to the **Password** buffer to make a valid password for the BSL access.

Syntax:

```
MSPPRG_API      INT_X  F_Put_Byte_to_Password_Buffer( LONG_X address,  
                                                       BYTE data );
```

Parameters value:

code address - 0xFFE0 to 0xFFFF

data - 0x00 to 0xFF

Return value:

0 - FALSE

1 - TRUE

-2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
BYTE code[0x20000];  
.....  
for( address = 0xFFE0; address <= 0xFFFF; address ++ )  
{  
    F_Put_Byte_to_Password_Buffer( address, code[address]);  
}  
.....
```

F_Get_Byte_from_Password_Buffer

F_Get_Byte_from_Password_Buffer - Read code data from password buffer.
VALID FPA index - (1 to 64)

Instruction allows to read or verify contents of the code from **Password** buffer. Data returns value 0x00 to 0xFF if in the particular **Password** buffer location the flag is set to *used*, otherwise return value **-1** (minus one) if data is empty.

Syntax:

```
MSPPRG_API INT_X F_Get_Byte_from_Password_Buffer( LONG_X address );
```

Parameters value:

code address - 0xFFE0 to 0xFFFF

Return value:

- 0x00 to 0xFF - valid code data
- 1 (0xFFFFFFFF) - code data not initialized on particular address
- 2 (0xFFFFFFFFE) - FPA_INVALID_NO

F_Put_Byte_to_Gang_Buffer

F_Put_Byte_to_Gang_Buffer - Write byte to Gang Write Data buffer.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Syntax:

```
MSPPRG_API INT_X F_Put_Byte_to_Gang_Buffer( BYTE target no,  
LONG_X address, BYTE data );
```

- target no: destination target's number from 1 to 6
- address: temporary buffer address equal the RAM or Flash destination address (0x0000 to 0x1FFFF)
- data: Byte to be written.

Return value:

- 1 - TRUE if specified address is legal (0x0000 to 0x1FFFF)
- 0- FALSE - if address is not valid
- 2 - FPA_INVALID_NO.

NOTE: Specified address in the temporary RAM or Flash buffer is the same as a physical RAM/FLASH address.

Example:

.....

```

.....
for( n=0; n<MAX_TARGET_DEVICE_NUMBER; n++)
  for( addr = 0x1000; addr<0x1020; addr++ )
    st = F_Put_Byte_to_Gang_Buffer( n, addr, data[addr][n] );
st = F_Copy_Gang_Buffer_to_Flash( 0x1000, 0x20 );
.....

```

F_Get_Byte_from_Gang_Buffer

F_Get_Byte_from_Gang_Buffer - Read one byte from the Gang Read Data buffer.
VALID FPA index - (1 to 64)

Syntax:

```

MSPPRG_API      BYTE  F_Get_Byte_from_Gang_Buffer( BYTE target_no,
                                                    LONG_X address );

```

Return value:

Requested byte from the specified address of the RAM/Flash temporary buffer.

Example:

see **F_Copy_All_Flash_To_Buffer**.

NOTE: Specified address in the temporary RAM or Flash buffer is the same as a physical RAM/FLASH address.

4.4 Encapsulated instructions

Encapsulated functions are powerful and easy to use. When called then all device actions from the beginning to the end are done automatically and final result is reported as TRUE or FALSE. Required configuration should be set first using **F_GetSetup** and **F_ConfigSetup** functions. Also Code file and Password File (if required) should be opened first. Encapsulated function has following sequence:

- Power from the programming adapter becomes ON if `PowerTargetEn` in configuration setup is enabled.
- Vcc is verified to be higher then 2.7V.
- JTAG/SBW or BSL communication between programming adapter and target device is initialized. JTAG/SBW or BSL interface is selected in configuration setup in `Interface`.
- Selected encapsulated instruction is executed (Autoprogram, Verify Fuse or Password, Memory Erase etc.).
- Communication between target device and programming adapter is terminated.
- Power from the programming adapter becomes OFF (if selected).
- Target device is released from the programming adapter.

F_AutoProgram

F_AutoProgram - Target device program with full sequence - erase, blank check, program, verify and blow security fuse (if enabled).

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed simultaneously.

Auto Program button is the most frequently function when programming microcontrollers in the production process. Auto Program function activates all required procedures to fully program and verify the flash memory contents. Typically, when flash memory needs to be erased, *Auto Program* executes the following procedures:

- initialization
- erase flash memory - restore retain data (including DCO constants) if enabled,
- confirm if memory has been erase,
- flash programming and verification,
- flash memory check sum verification,
- blowing the security fuse (if flag `BlowFuseEn = 3`).

Syntax:

```

MSPPRG_API      INT_X  F_AutoProgram( INT_X mode );
mode = 0;
mode = 1 and up - reserved

```

Return value:

```

Target devices result mask - 0x00 to 0x3F
When bit is '1' - TRUE
          '0' - FALSE
          -2 (0xFFFFFFFF) - FPA_INVALID_NO

```

Example:

```

.....
if( F_Initialization() != TRUE )    //required API-Dll - initialization
{
    // Initialization error
}

F_GetSetup( &config ); //API-DLL - get configuration from the programmer
..... // modify configuration if required
F_ConfigSetup( config ); // download setup to programmer

int st = F_ConfigFileLoad( "c:\test\configfile.cfg" );
if(( st & 1 ) != TRUE )
{
    Info = st & 0xFFFFE;
    .....
}

do{

    ..... // prepare next microcontrollers
    targets_mask = 0x3F //active all six target devices
    F_SetConfig( CFG_TARGET_EN_INDEX, (INT_X)targets_mask );
    if( F_AutoProgram(0) == targets_mask )
    {
        //all target devices programmed
    }
    else
    {
        //some targets has nod been programmed
    }

    ..... //exit if the last microcontrollers
           // has been programmed

} while(1);
.....

```

F_VerifyFuseOrPassword

F_VerifyFuseOrPassword -Verify the Security fuse if JTAG/SBW interface is active, or verify the password access if BSL interface is active.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed simultaneously.

Syntax:

```
MSPPRG_API      INT_X  F_VerifyFuseOrPassword( void );
```

Return value:

Target devices result mask - 0x00 to 0x3F
When bit is '1' - TRUE
 '0' - FALSE
 -2 (0xFFFFFFFF) - FPA_INVALID_NO

F_Memory_Erase

F_Memory_Erase - Erase Target's Flash Memory

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed simultaneously.

Erase flash size, or sector to be erased, should be specified in the configuration setup. When mode erase flag is set to one, then all memory will be erased, regardless erase memory configuration setup value. When the *Retain Data* are specified (including DCO constants in the F2xx), then retain data are read before erase process, and restored after the erase process.

Syntax:

```
MSPPRG_API      INT_X  F_Memory_Erase( INT_X mode );  
mode = 0 -> erase space specify by the FlashEraseModeIndex;  
mode = 1 -> erase all Flash memory, regardless FlashEraseModeIndex;
```

Return value:

Target devices result mask - 0x00 to 0x3F
When bit is '1' - TRUE
 '0' - FALSE
 -2 (0xFFFFFFFF) - FPA_INVALID_NO

F_Memory_Blank_Check

F_Memory_Blank_Check - Check if the Target's Flash Memory is blank.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed simultaneously.

Syntax:

```
MSPPRG_API          INT_X F_Memory_Blank_Check( void );
```

Return value:

```
Target devices result mask - 0x00 to 0x3F  
When bit is '1' - TRUE  
                '0' - FALSE  
-2 (0xFFFFFFFF) - FPA_INVALID_NO
```

F_Memory_Write

F_Memory_Write - Write content taken from the Code file to the selected Target Devices Flash Memory.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed simultaneously.

Syntax:

```
MSPPRG_API          INT_X F_Memory_Write( INT_X mode );  
mode = 0;  
mode = 1 and up - reserved
```

Return value:

```
Target devices result mask - 0x00 to 0x3F  
When bit is '1' - TRUE  
                '0' - FALSE  
-2 (0xFFFFFFFF) - FPA_INVALID_NO
```

F_Memory_Verify

F_Memory_Verify - Verify contents of the selected Target Devices Flash Memory and Code file.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed simultaneously.

Note: During the verification process either all memory or just the selected part of the memory is verified, depending on settings specified in the configuration setup `FlashEraseModeIndex`. Only data taken from the Code file are compared with the target's flash memory. If size of the flash memory is bigger than code size then all reminding data in flash memory is ignored.

Syntax:

```
MSPPRG_API      INT_X  F_Memory_Verify( INT_X mode );  
mode = 0;  
mode = 1 and up - reserved
```

Return value:

```
Target devices result mask - 0x00 to 0x3F  
When bit is '1' - TRUE  
          '0' - FALSE  
          -2 (0xFFFFFFFF) - FPA_INVALID_NO
```

F_Gang_Flash_Read

F_Gang_Flash_Read - Read contents of the selected Target Devices Flash Memory and save it in the Gang Read Buffers.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed simultaneously.

Size of the read memory is defined in the configuration setup

`FlashReadModeIndex`, `ReadSegmA`, `ReadSegmB`, `ReadStartAddr`, `ReadStopAddr`.

All data will be saved in the internal Read Gang Buffer. Contents from the Read Gang Buffer can be taken using function

```
BYTE F_Get_Byte_from_Gang_Buffer( BYTE target_no, LONG_X addr );
```

Syntax:

```
MSPPRG_API      INT_X  F_Gang_Flash_Read( void );
```

Return value:

```
Target devices result mask - 0x00 to 0x3F  
When bit is '1' - TRUE  
          '0' - FALSE  
          -2 (0xFFFFFFFF) - FPA_INVALID_NO
```


NOTE: *Specified address in the Gang Read Buffers is the same as a physical Flash address in target device.*

Example:

```
BYTE data[0x20000][MAX_TARGET_DEVICE_NUMBER];
unsigned int addr,n;
.....
    st = F_Gang_Flash_Read();
    if ( st != 0 )
    {
        for( n=0; n<MAX_TARGET_DEVICE_NUMBER; n++)
            for( addr=0x1000; addr<=0x1FFFF; addr++)
                data[addr][n] = F_Get_Byte_from_Gang_Buffer( n, addr );
    }
.....
```

4.4 Sequential instructions

Sequential instructions allow access to the target device in any combination of the small instructions like erase, read, write sector, modify part of memory etc. Sequential instruction have an access only when communication between target device and programming adapter is initialized. This can be done when *F_Open_Target_Device* instruction is called. When communication is established, then any of the sequential instruction can be called. When the process is finished, then at the end *F_Close_Target_Device* instruction should be called. When communication is terminated, then sequential instructions can not be executed.

Note: Erase/Write/Verify/Read configuration setup is not required when sequential instructions are called. Also code file is not required to be downloaded. All data to be written, erased, and read is specified as a parameter to the sequential functions. Data downloaded from the code file is ignored in this case.

Very important:

The sequential functions allows to program words in the FLASH memory on any flash space location. Also the same bytes / words can be programmed few times. Software is not be able to control how many times the same location of the flash has been programmed between erasures. User should take a full responsibility for programming the flash memory according to the MSP430 specifications. See TI's data sheets and manuals for details.

The following flash programming limitation should be taken to consideration:

1. The same word or byte can not be programmed more then twice between erasures. Otherwise, damage can occur.
2. In byte/word mode, the internally-generated programming voltage is applied to the complete 64-byte block, each time a byte or word is written, for 32 of the 35 f_{FTG} cycles. With each byte or word write, the amount of time the block is subjected to the programming voltage accumulates. The cumulative programming time, t_{CPT} , must not exceeded for any block. If the cumulative programming time is met, the block must be erased before performing any further writes to any address within the block. The cumulative time for the older MSP430 microcontrollers (F1xx, F4xx) is typically 4 ms. For the newer ones - 10 ms. See the device-specific datasheet for specifications. .

The FTG frequency used in the USB-MSP430-FPA with the single word (two bytes) programming mode is 428 kHz. This means that programming time of the single word is approx 75 us. Programming time of the one byte would be the same.

Cumulative time for the 64 bytes uses byte write mode would be approx.

$$t_{CPT} = 64 * 75 \text{ us} = 4.8 \text{ ms.}$$

This time can exceed the cumulative time for the older MSP430 microcontrollers. From that reason the USB-MSP430-FPA uses the word write mode to flash that allows to decrease 2 times the cumulative time.

F_Open_Target_Device

F_Open_Target_Device - Initialization communication with the target device.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed simultaneously.

When **F_Open_Target_Device** is executed, then

- Power from the programming adapter becomes ON if PowerTargetEn in configuration setup is enabled.
- Vcc is verified to be higher than 2.7V.
- JTAG/SBW or BSL communication between programming adapter and target device is initialized.

Note: The correct BSL password should be downloaded to password or data buffer to be able to activate target devices if BSL or Fast BSL interface is used. If password is unknown the use encapsulated **F_Memory_Erase()** function first.

Target device is ready to get other sequential instructions.

Syntax:

```
MSPPRG_API          INT_X F_Open_Target_Device( void );
```

Return value:

Target devices result mask - 0x00 to 0x3F

When bit is '1' - TRUE

'0' - FALSE

-2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
int st, mask;
long addr;
.....
mask = 0x3F;                               //enable all six target devices
F_SetConfig( CFG_TARGET_EN_INDEX, (INT_X)mask );
F_Open_Target_Device();
.....
F_Segment_Erase(0x1000);
st = F_Sectors_Blank_Check( 0x1000, 0x107f );
if( st != mask )
{ ..... }
for( addr = 0x1000; addr<0x1020; addr++ )
    F_Put_Byte_to_Gang_Buffer( 1, addr, data(addr) )
F_Copy_Buffer_to_Flash( 0x1000, 0x20 );
    //copy content from Gang Buffer no '1' to all target
F_Segment_Erase(0x4200);
F_Segment_Erase(0x4400);
F_Segment_Erase(0x4600);
.....
F_Close_Target_Device();
.....
```

F_Close_Target_Device

F_Close_Target_Device - Termination communication between target device and programming adapter.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Instruction should be called on the end of the sequential instructions. When **F_Close_Target_Device** instruction is executed then:

- Communication between target device and programming adapter is terminated.
- Power from the programming adapter becomes OFF (if selected).
- Target device is released from the programming adapter.

Syntax:

```
MSPPRG_API      INT_X  F_Close_Target_Device( void );
```

Return value:

```
Target devices result mask - 0x00 to 0x3F
    When bit is '1' - TRUE
                '0' - FALSE
    -2 (0xFFFFFFFF) - FPA_INVALID_NO
```

Example:

See example above (**F_Open_Target_Device**).

F_Segment_Erase

F_Segment_Erase - Erase any segment of the MSP430 Flash memory.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) *executed sequentially.*

Parameters:

segment address - Even number from 0x1000 to 0x1FFFE,

To erase a memory segment specify an address within that memory segment. For example to erase segment 0x2000-0x21FF any address from the range 0x2000 to 0x21FF can be specified. To erase all memory segments, erase the memory segment by segment, or used the encapsulated instruction

```
F_Memory_Erase(1);
```

*Note: When encapsulated instruction is executed, then next access to the sequential instruction can be accessed only when **F_Open_Target_Device** instruction is called again.*

Syntax:

```
MSPPRG_API INT_X F_Segment_Erase( LONG_X address );
```

Return value:

```
Target devices result mask - 0x00 to 0x3F
When bit is '1' - TRUE
          '0' - FALSE
          -2 (0xFFFFFFFF) - FPA_INVALID_NO
```

Example:

```
.....
F_Segment_Erase(0x4000); // erase segment 0x4000 to 0x41FF
F_Segment_Erase(0x4100); // erase the same segment
F_Segment_Erase(0x1010); // erase INFO segment 0x1000 to 0x107F
.....
```

F_Sectors_Blank_Check

F_Sectors_Blank_Check - Blank check part or all Flash Memory. Start and stop address of the tested memory should be specified.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Parameters:

start address - Even number from 0x1000 to 0x1FFFE,

stop address - Odd number from 0x1001 to 0x1FFFF,

Syntax:

```
MSPPRG_API INT_X F_Sectors_Blank_Check( LONG_X start_addr,  
                                         LONG_X stop_addr );
```

Return value:

Target devices result mask - 0x00 to 0x3F

When bit is '1' - TRUE

'0' - FALSE

-2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
.....  
F_Sectors_Blank_Check (0x1000, 0x107F) ; //INFO secto blank check  
F_Sectors_Blank_Check (0x8000, 0xFFFF) ; //32 kB memory size blank check  
F_Sectors_Blank_Check (0x1220, 0x123f) ; //part of sector blank check  
.....
```

F_Write_Word

F_Write_Word - Write one word (two bytes) to RAM, registers, IO etc. without FLASH.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Note: When the **BSL** or **Fast BSL** is used then an access to RAM location 0x200 to 0x2FF is blocked. This RAM area is used by stack and firmware for the BSL or Fast BSL.

Write one word to any location of the target devices. Write to Flash has no effect.

Parameters:

address - **Even** address of RAM, register, I/O etc.
data - one word to be written to target device

Syntax:

```
MSPPRG_API INT_X F_Write_Word( LONG_X addr, INT_X data );
```

Return value:

Target devices result mask - 0x00 to 0x3F
When bit is '1' - TRUE
'0' - FALSE
-2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
F_Write_Word( 0x0124, 0x2143 );
```

F_Write_Byte

F_Write_Word - Write one byte to RAM, registers, IO etc. without FLASH.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

NOTE: Instruction not supported in **BSL** and **Fast BSL**.

Write one byte to any location of the target devices. Write to Flash has no effect.

Parameters:

address - **Any** address from 0x0000 to 0x1FFFF,
data - one byte to be written to target device

Syntax:

```
MSPPRG_API INT_X F_Write_Byte( LONG_X addr, BYTE data );
```

Return value:

Target devices result mask - 0x00 to 0x3F
When bit is '1' - TRUE
'0' - FALSE
-2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
F_Write_Byte( 0x33, 0x20 );
```

F_Copy_Buffer_to_Flash

F_Copy_Buffer_to_Flash - Write data from the Gang Buffer no '1' to all selected target devices.
VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Parameters:

start address - Even number from 0x1000 to 0x1FFFE,
size - Even number 2 and up
block of data in bytes to be written.

Syntax:

```
MSPPRG_API INT_X F_Copy_Buffer_to_Flash( LONG_X start_addr, INT_X size );
```

Return value:

Target devices result mask - 0x00 to 0x3F
When bit is '1' - TRUE
'0' - FALSE
-2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
long addr;  
int mask;
```

```
.....  
for( addr = 0x1000; addr<0x2100; addr++ )  
    F_Put_Byte_to_Gang_Buffer( 1, addr, (BYTE)(0xFF & addr));  
.....  
mask = 0x3F; //enable all six target devices  
F_SetConfig( CFG_TARGET_EN_INDEX, (INT_X)mask );  
F_Open_Target_Device();  
F_Copy_Buffer_to_Flash( 0x1000, 0x20 );  
F_Copy_Buffer_to_Flash( 0x1050, 0x20 );  
F_Copy_Buffer_to_Flash( 0x2000, 0x100 );  
.....
```

F_Copy_Gang_Buffer_to_Flash

F_Copy_Gang_Buffer_to_Flash - Write data block from Write Gang Buffers to Target Devices.
- Data from Gang Buffer no'1' to Target Device no '1'
- Data from Gang Buffer no'2' to Target Device no '2'

.....
- Data from Gang Buffer no '6' to Target Device no '6'

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

NOTE: Instruction not supported in **BSL** and **Fast BSL**.

Procedure allows to save unique data to each target devices. All unique data are saved in target devices simultaneously. The unique data should be prepared first and saved in Gang Buffer - dedicated to target 1,2...6.

Note: The unique data writing process is much slower (around 10 times) that writing the same data all targets. From that reason this procedure should be used only for saving unique data like unique serial numbers, calibration data etc.

Syntax:

```
MSPPRG_API INT_X F_Copy_Gang_Buffer_to_Flash( LONG_X start_addr,  
                                             INT_X size);
```

Parameters:

- start address - Even number from 0x1000 to 0x1FFFE,
- size - Even number 2 and up,
- block of data in bytes to be written.

Return value:

- Target devices result mask - 0x00 to 0x3F
- When bit is '1' - TRUE
- '0' - FALSE
- 2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
long addr;  
int n, mask;  
.....  
for( n=0; n<MAX_TARGET_DEVICE_NUMBER; n++)  
    for( addr=0x1000; addr<=0x1020; addr++)  
        F_Put_Byte_to_Gang_Buffer( n, addr, data[addr][n] );  
.....  
mask = 0x3F; //enable all six target devices  
F_SetConfig( CFG_TARGET_EN_INDEX, (INT_X)mask );
```

```

F_Open_Target_Device();
F_Copy_Gang_Buffer_to_Flash( 0x1000, 0x20 );
.....

```

F_Copy_Buffer_to_RAM

F_Copy_Buffer_to_RAM - Write “size” number of bytes from Gang Write Buffer no ‘1’ to RAM. Starting address is specified in the “start address”.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Note: When the **BSL** or **Fast BSL** is used then an access to RAM location 0x200 to 0x2FF is blocked. This RAM area is used by stack and firmware for the BSL or Fast BSL.

Syntax:

```

MSPPRG_API      INT_X  F_Copy_Buffer_to_RAM( LONG_X start_address,
                                           LONG_X  size );

```

Parameters:

start address - Even address
size - Even number

Return value:

Target devices result mask - 0x00 to 0x3F
When bit is ‘1’ - TRUE
 ‘0’ - FALSE
 -2 (0xFFFFFFFF) - FPA_INVALID_NO

NOTE: *Specified address in the temporary RAM/Flash buffer is the same as a physical RAM address.*

Example:

```

.....
for( addr = 0x220; addr<0x300; addr++ )
    st = F_Put_Byte_To_Gang_Buffer( 1, addr, data[addr] );
st = F_Copy_Buffer_to_RAM( 0x220, 0xE0 );
.....

```

F_Copy_Gang_Buffer_to_RAM

F_Copy_Gang_Buffer_to_RAM - Write “size” number of bytes from the temporary RAM/Flash Gang Buffers to RAM.

Data from Gang Buffer no ‘1’ to RAM of the Target’s no ‘1’

Data from Gang Buffer no ‘2’ to RAM of the Target’s no ‘2’

.....
Data from Gang Buffer no ‘6’ to RAM of the Target’s no ‘6’

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

NOTE: Instruction not supported in **BSL** and **Fast BSL**.

Procedure allows to save unique data to each target devices. All unique data are saved in target devices simultaneously. The unique data should be prepared first and saved in Gang Buffer - dedicated to target 1,2...6.

Note: The unique data writing process is much slower (around 10 times) that writing the same data all targets. From that reason this procedure should be used only for saving unique data like unique register contents, calibration data etc.

Syntax:

```
MSPPRG_API      INT_X  F_Copy_Gang_Buffer_to_RAM( LONG_X start_address,  
                                     LONG_X size );
```

Parameters:

start address - Even address

size - Even number

Return value:

Target devices result mask - 0x00 to 0x3F

When bit is ‘1’ - TRUE

‘0’ - FALSE

-2 (0xFFFFFFFF) - FPA_INVALID_NO

NOTE: *Specified address in the temporary RAM/Flash buffer is the same as a physical RAM address.*

Example:

```

.....
.....

for( n=0; n<MAX_TARGET_DEVICE_NUMBER; n++)
  for( addr = 0x220; addr<0x240; addr++ )
    st = F_Put_Byte_To_Gang_Buffer( n, addr, data[addr][n] );
st = F_Copy_Gang_Buffer_to_RAM( 0x220, 0x20 );
.....

```

F_Copy_RAM_to_Gang_Buffer

F_Copy_RAM_to_Gang_Buffer - Read specified in "size" number of bytes from the RAM and save it in the temporary gang buffer. Starting address is specified in the "start address".

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Syntax:

```

MSPPRG_API      INT_X  F_Copy_RAM_to_Gang_Buffer( LONG_X start_address,
                                                    LONG_X size );

```

Parameters:

- start address - Even address
- size - Even number

Return value:

- Target devices result mask - 0x00 to 0x3F
- When bit is '1' - TRUE
- '0' - FALSE
- 2 (0xFFFFFFFF) - FPA_INVALID_NO

NOTE: Specified address in the temporary RAM/Flash buffer is the same as a physical RAM address.

Example:

```

.....
.....

st = F_Copy_RAM_to_Gang_Buffer( 0x220, 0xE0 );
if( st == TRUE )
{
  for( n=0; n<MAX_TARGET_DEVICE_NUMBER; n++)
    for( addr = 0x0220; addr<0x0300; addr++ )

```

```

        data[addr][n] = F_Get_Byte_from_Gang_Buffer( n, addr );
    }
else
    {
        .....
    }
.....

```

F_Copy_Flash_to_Gang_Buffer

F_Copy_Flash_to_Gang_Buffer - Read specified in "size" number of bytes from Flash and save it in the temporary gang buffer. Starting address is specified in the "start address".

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Syntax:

```

MSPPRG_API      INT_X  F_Copy_Flash_to_Gang_Buffer( LONG_X start_address,
                                                    LONG_X  size );

```

Parameters:

start address - Even address
size - Even number

Return value:

Target devices result mask - 0x00 to 0x3F
When bit is '1' - TRUE
'0' - FALSE
-2 (0xFFFFFFFF) - FPA_INVALID_NO

NOTE: *Specified address in the temporary RAM/Flash buffer is the same as a physical RAM address.*

Example:

```

.....
.....
st = F_Copy_Flash_to_Gang_Buffer( 0x2220, 0xE0 );
if( st == TRUE )
{
    for( n=0; n<MAX_TARGET_DEVICE_NUMBER; n++)
        for( addr = 0x2220; addr<0x2300; addr++ )
            data[addr][n] = F_Get_Byte_from_Gang_Buffer( n, addr );
}
else

```

```

    {
        .....
    }
    .....

```

F_Set_PC_and_RUN

F_Set_PC_and_RUN - Instructions allows to run program in microcontroller from specified PC in the RAM or Flash location. Program should be downloaded first using the Write to Flash or Ram procedures. When the processor is running then the JTAG is disconnected from the CPU. The CPU can be controlled from JTAG again when the instruction **F_Open_Target_Device** or **F_Synch_CPU_JTAG** is used.

Note: *The **F_Open_Target_Device** instruction is resetting the CPU. All internal registers states are set to default value. The **F_Synch_CPU_JTAG** is synchronizing the CPU and JTAG on fly. The CPU is stopped, but all registers have not been modified.*

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

NOTE: Instruction not supported in BSL and Fast BSL.

Syntax:

```
MSPPRG_API      INT_X F_Set_PC_and_RUN( LONG_X PC_address );
```

Return value:

```

    When bit is '1' - TRUE
                '0' - FALSE
                -2 (0xFFFFFFFF) - FPA_INVALID_NO

```

Example:

```

unsigned char rd_data[0x100];

.....
F_Autoprogram(0); // download the test code
F_Open_Target_Device();
.....
F_Set_PC_and_RUN( PC1 ) // run the test program from location PC1
for( k = 0; k<1000; k++)
{
    F_Capture_PC_Addr( 0 );
    st = 1;
    for( target = 1; target<6; target++)

```

```

        {
            if( F_Capture_PC_Addr( target ) < PC_addr_min_1 ) st = 0;
            if( F_Capture_PC_Addr( target ) > PC_addr_max_1 ) st = 0;
        }
        if( st == 1 ) break;
        delay (...);
    }
    F_Synch_CPU_JTAG();
    F_Memory_Read_Data( address, size, byte_buffer );
                        // read the test-1 result from the RAM
.....
    F_Set_PC_and_RUN( PC2 ) // run the test program from location PC2
    for( k = 0; k<1000; k++)
    {
        F_Capture_PC_Addr( 0 );
        st = 1;
        for( target = 1; target<6; target++)
        {
            if( F_Capture_PC_Addr( target ) < PC_addr_min_2 ) st = 0;
            if( F_Capture_PC_Addr( target ) > PC_addr_max_2 ) st = 0;
        }
        if( st == 1 ) break;
        delay (...);
    }
    F_Synch_CPU_JTAG();
    F_Memory_Read_Data( address, size, byte_buffer );
                        // read the test-2 result from the RAM
.....
.....
.....
    F_Set_PC_and_RUN( PCn ) // run the test program from location PCn
    for( k = 0; k<1000; k++)
    {
        F_Capture_PC_Addr( 0 );
        st = 1;
        for( target = 1; target<6; target++)
        {
            if( F_Capture_PC_Addr( target ) < PC_addr_min_3 ) st = 0;
            if( F_Capture_PC_Addr( target ) > PC_addr_max_3 ) st = 0;
        }
        if( st == 1 ) break;
        delay (...);
    }
    F_Synch_CPU_JTAG();
    F_Memory_Read_Data( address, size, byte_buffer );
                        // read the test-n result from the RAM

    F_Close_Target_Device();

```

```
.....  
F_Autoprogram(0); // download the final code
```

F_Capture_PC_Addr

F_Capture_PC_Addr - Instructions monitoring the PC address on fly without stopping the MCU

VALID FPA index - (1 to 64)

NOTE: Instruction not supported in **BSL** and **Fast BSL**.

Syntax:

```
MSPPRG_API INT_X F_Capture_PC_Addr( BYTE target_no );
```

```
target_no  
0 - Capture PC address in all targets  
1-6 - get result of the PC-address from the selected target.  
F_Capture_PC_Addr( 0 ) must be executed first.
```

Return value:

```
0 - FALSE;  
>0 - Capture PC address
```

Example:

See example in the **F_Set_PC_and_RUN** instruction.

F_Synch_CPU_JTAG

F_Synch_CPU_JTAG - Instructions allows to synchronize CPU with JTAG and stop the CPU when the **F_Set_PC_and_RUN** has been executed.

*Note: When the CPU is executing wrong code with critical error, or hardware **RESET** has been used, then only the **F_Open_Target_Device** can recover the JTAG communication with CPU.*

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

NOTE: Instruction not supported in **BSL** and **Fast BSL**.

Syntax:

```
MSPPRG_API      INT_X  F_Synch_CPU_JTAG( void );
```

Return value:

```
When bit is '1' - TRUE
              '0' - FALSE
              -2 (0xFFFFFFFF) - FPA_INVALID_NO
```

Example:

See example in the **F_Set_PC_and_RUN** instruction.

F_Blow_Fuse

F_Blow_Fuse - Blow the security fuse instruction.

VALID index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

To access the security fuse and blow it, the flag **BlowFuseEn** in the configuration setup must be enable (BlowFuseEn = 1 or 3) and JTAG/SBW communication with the target device must be selected. Otherwise function returns - FALSE.

The fuse can also be blown when the **F_AutoProgram** instruction is executed with enabled blow security fuse option.

When the BSL communication is active then instruction **F_Blow_Fuse** cannot be successfully executed and returns - FALSE.

***Note:** When **BlowFuseEn** in configuration setup is enabled and **F_AutoProgram** function is executed, then at the end of autoprogram process security fuse will be blown. No other access to the target device via JTAG/SBW interface will be possible. If some communication with the target device is required after autoprogram, like to modify some calibration data etc., then **BlowFuseEn** flag should be disabled before **F_AutoProgram** is called. Flag **BlowFuseEn** should be enabled at the end of communication with the target device just before execution of the **F_Blow_Fuse** instruction and disabled after.*

NOTE: Instruction not supported in **BSL** and **Fast BSL**.

Syntax:

```
MSPPRG_API      INT_X  F_Blow_Fuse( void );
```

Return value:

Target devices result mask - 0x00 to 0x3F
When bit is '1' - TRUE
 '0' - FALSE
 -2 (0xFFFFFFFF) - FPA_INVALID_NO

Example:

```
int st;
.....
do{
..... // prepare next microcontroller
F_SetConfig( CFG_INTERFACE, INTERFACE_JTAG); //select JTAG interface
F_SetConfig( CFG_BLOWFUSE, 0 ) //disable fuse blow option
..... // modify configuration if required
st = F_AutoProgram(0);
..... // do extra communication with
..... // the target device

F_SetConfig( CFG_BLOWFUSE, 1 ) //enable fuse blow option
st = F_Blow_Fuse( void ); // Blow the security Fuse
F_SetConfig( CFG_BLOWFUSE, 0 ) //disable fuse blow option
.....
..... // break; if the last microcontroller
..... // has been programmed
} while(1);
.....

or

do{
..... // prepare next microcontroller
F_SetConfig( CFG_INTERFACE, INTERFACE_JTAG); //select JTAG interface
F_SetConfig( CFG_BLOWFUSE, 1 )
//enable blow security fuse using F_Blow_Fuse() function only.
//The F_AutoProgram will not blow the fuse.
..... // modify configuration if required
st = F_AutoProgram(0); // Fuse blow is disabled
..... // do extra communication with
..... // the target device
st = F_Blow_Fuse( void ); // Blow the security Fuse
..... // break; if the last microcontroller
..... // has been programmed
} while(1);

or

do{
..... // prepare next microcontroller
F_SetConfig( CFG_INTERFACE, INTERFACE_JTAG); //select JTAG interface
```

```

F_SetConfig( CFG_BLOWFUSE, 3 )
    //enable blow security fuse when the F_AutoProgram is executed
    //(if all passed)
                                // modify configuration if required
st = F_AutoProgram(0);          // Fuse blow is disabled
.....                          // break; if the last microcontroller
                                // has been programmed
} while(1);

```

F_Adj_DCO_Frequency

F_Adj_DCO_Frequency - Adjust DCO to desired frequency and return register value for that frequency.

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Syntax:

```
MSPPRG_API      INT_X F_Adj_DCO_Frequency( INT_X freq_Hz );
```

freq_Hz - 100000 to 16000000 (100kHz to 16 MHz)

Return value:

FALSE - 0;

TRUE MASK - (0x01 to 0x3F)

When function is finished then the DCO constants can be read using F_Get_DCO_constants

Example:

```

F_Adj_DCO_Frequency( 2000000 );
for( target=1; target<=6; target++)
    DCO_data[target] = F_Get_DCO_constant( target);

```

F_Get_DCO_constant

F_Get_DCO_constant - Get the DCO constant after adjusting the DCO with F_Adj_DCO_Frequency(...) Function

VALID FPA index - (1 to 64).

Syntax:

```
MSPPRG_API      INT_X F_Get_DCO_constant( BYTE target_no );
```

Return value:

DCO constant value (16-bits word)

Example:

See F_Adj_DCO_Frequency(..) function

F_Test_DCO_Frequency

F_Test_DCO_Frequency - Measure DCO frequency for desired DCO register value

VALID FPA index - (1 to 64) or 0 (ALL FPAs) executed sequentially.

Syntax:

```
MSPPRG_API      INT_X F_Test_DCO_Frequency( void );
```

Return value:

FALSE - 0;

TRUE MASK - (0x01 to 0x3F)

Example:

```
for( target=1; target<=6; target++)
    F_Set_DCO_constant( target, 0x1234 );
F_Test_DCO_Frequency();
for( target=1; target<=6; target++)
    DCO_freq[target] = F_Get_DCO_Freq_result( target );
```

F_Set_DCO_constant

F_Set_DCO_constant - Set the DCO constant for DCO test frequency

VALID FPA index - (1 to 64).

Syntax:

```
MSPPRG_API INT_X F_Set_DCO_constant( BYTE target_no, INT_X DCO_constant );
```

Return value:

0 - invalid target number

1 - ok

Example:

See F_Test_DCO_Frequency(..) function

F_Get_DCO_Freq_result

F_Get_DCO_Freq_result - Get the DCO frequency test result from the
F_Test_DCO_Frequency();
VALID FPA index - (1 to 64).

Syntax:

```
MSPPRG_API INT_X F_Get_DCO_Freq_result( BYTE target_no );
```

Return value:

DCO frequency in Hz

Example:

See F_Test_DCO_Frequency(..) function

Appendix A

GangPro430 Command Line interpreter

The **Multi-FPA API-DLL** can be used with the command line interpreter shell. This shell allows to use the standard Command Prompt windows to execute the API-DLL functions. All required files are located in the directory

C:\Program Files\Elprotronic\MSP430\USB GangPro430\CMD-line

and contains

GP430-commandline.exe	-> command line shell interpreter
GangPro430FPA.dll	-> standard API-DLL files
GangPro430FPA1.dll	-> ----,-----

All API-DLL files should be located in the same directory where the **GP430-commandline.exe** is located. To start the command line interpreter, the **GP430-commandline.exe** should be executed.

Command Syntax:

instruction_name (parameter1, parameter2,)

parameter:

1. string (file name etc.) - "filename"
2. numbers

	integer decimal	eg. 24
or	integer hex	eg. 0x18

Note: Spaces are ignored

Instructions are not case sensitive

F_OpenInstancesAndFPAs("*"# "*")

and **f_openinstancesandfpas("*"# "*")**

are the same.

Example-1:

Run the **GP430-commandline.exe**

Type:

```
F_OpenInstancesAndFPAs( "*"# *" ) // open instances and find the first adapter (any SN)
```

Press ENTER - result ->1 (OK)

Type:

```
F_Initialization() //initialization with config taken from the config.ini  
//setup taken from the GangPro430 - with defined MSP430 type, code file etc.
```

Press ENTER - result ->1 (OK)

Type:

```
F_AutoProgram( 0 )
```

Press ENTER - result ->63 (63-> 0011 1111 -> programmed six targets -> OK)

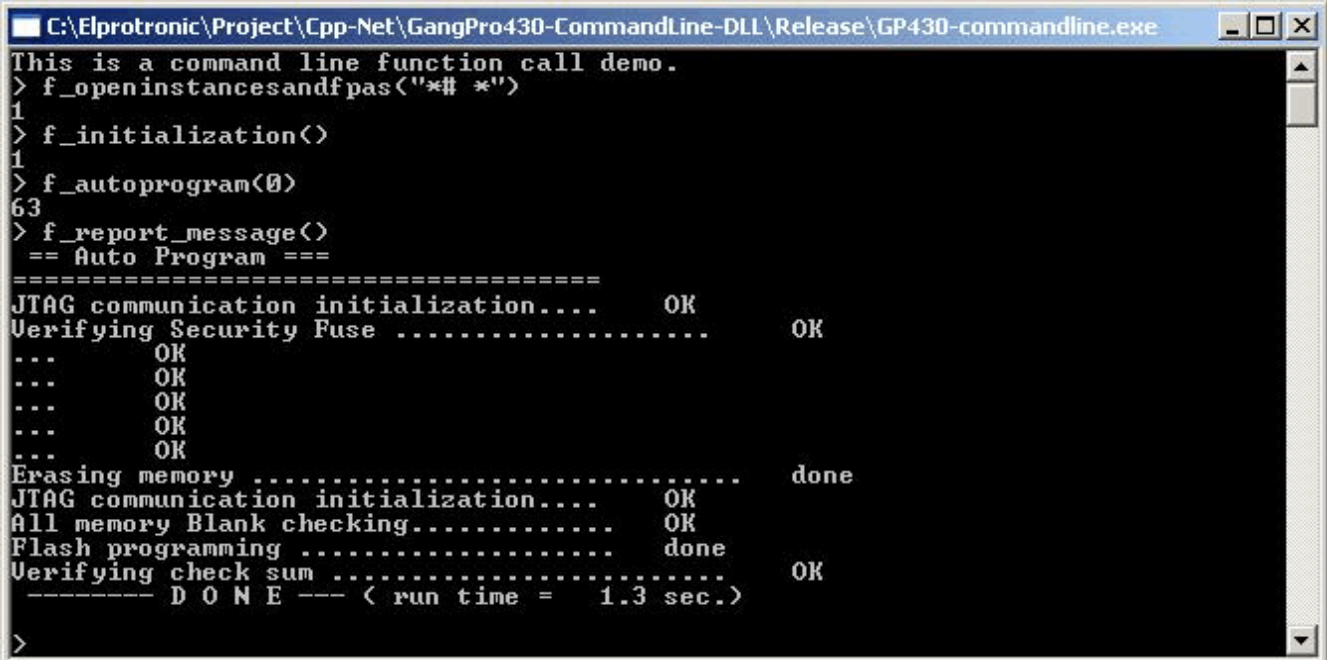
Type:

```
F_Report_Message()
```

Press ENTER - result -> displayed the last report message (from the F_AutoProgram(0))

See figure A-1 for result:

Type **quit()** and press ENTER to close the **GP430-commandline.exe** program.



```
C:\Elprotronic\Project\Cpp-Net\GangPro430-CommandLine-DLL\Release\GP430-commandline.exe
This is a command line function call demo.
> f_openinstancesandfpas( "*"# *" )
1
> f_initialization()
1
> f_autoprogram(0)
63
> f_report_message()
== Auto Program ==
=====
JTAG communication initialization.... OK
Verifying Security Fuse ..... OK
... OK
... OK
... OK
... OK
... OK
Erasing memory ..... done
JTAG communication initialization.... OK
All memory Blank checking..... OK
Flash programming ..... done
Verifying check sum ..... OK
----- D O N E --- ( run time = 1.3 sec.)
>
```

Figure A-1

Example-2:

```
F_OpenInstancesAndFPAs( "*"# *" )      // open instances and find the first adapter (any SN)
F_Initialization()
F_Report_Message()
F_ConfigFileLoad( "filename" )         //put vaild path and config file name
F_ReadCodeFile( 1, "FileName" )       //put vaild path and code file name (TI.txt format)
F_AutoProgram( 0 )
F_Report_Message()
.....
.....
F_Put_Byte_to_Gang_Buffer( 1, 0x8000, 0x11 )
F_Put_Byte_to_Gang_Buffer( 1, 0x8001, 0x21 )
.....
F_Put_Byte_to_Gang_Buffer( 1, 0x801F, 0xA6 )
F_Open_Target_Device()
F_Segment_Erase( 0x8000 )
F_Copy_Buffer_to_Flash( 0x8000, 0x20 )
F_Copy_Flash_to_Gang_Buffer( 0x8000, 0x20 )
F_Get_Byte_from_Gang_Buffer( 1, 0x8000 )
F_Get_Byte_from_Gang_Buffer( 2, 0x8000 )
F_Get_Byte_from_Gang_Buffer( 1, 0x8001 )
F_Get_Byte_from_Gang_Buffer( 2, 0x8001 )
.....
F_Get_Byte_from_Gang_Buffer( 1, 0x801F )
F_Get_Byte_from_Gang_Buffer( 2, 0x801F )
F_Close_Target_Device()
quit()
```

List of command line instructions

```
quit()           ;close the command interpreter program
help()           ;display list below
F_Trace_ON()
```


F_Trace_OFF()
F_OpenInstances(no)
F_CloseInstances()
F_OpenInstancesAndFPAs("FileName")
F_Set_FPA_index(fpa)
F_Get_FPA_index()
F_LastStatus(fpa)
F_DLLTypeVer()
F_Multi_DLLTypeVer()
F_Check_FPA_access(index)
F_Get_FPA_SN(fpa
F_APIDLL_Directory("APIDLLpath")
F_Initialization()
F_DispSetup()
F_Close_All()
F_Power_Target(OnOff)
F_Reset_Target()
F_Report_Message()
F_ReadCodeFile(file_format, "FileName")
F_Get_CodeCS(dest)\n");
F_ReadPasswFile(file_format, "FileName")
F_ConfigFileLoad("filename")
F_SetConfig(index, data)
F_GetConfig(index)
F_Put_Byte_to_Gang_Buffer(target_no, addr, data)
F_Get_Byte_from_Gang_Buffer(target_no, addr)
F_Clr_Code_Buffer()
F_Put_Byte_to_Code_Buffer(addr, data)
F_Put_Byte_to_Password_Buffer(addr, data)
F_Get_Byte_from_Code_Buffer(addr)
F_Get_Byte_from_Password_Buffer(addr)
F_AutoProgram(0)
F_VerifyFuseOrPassword()
F_Memory_Erase(mode)
F_Memory_Blank_Check()
F_Memory_Write(mode)

F_Memory_Verify(mode)
F_Gang_Flash_Read()
F_Open_Target_Device()
F_Close_Target_Device()
F_Segment_Erase(address)
F_Sectors_Blank_Check(start_addr, stop_addr)
F_Copy_Buffer_to_Flash(start_addr, size)
F_Copy_Gang_Buffer_to_Flash(start_addr, size)
F_Flash_to_Gang_Buffer(start_addr, size)
F_Blow_Fuse()
F_Write_Word(addr, data)
F_Write_Byte(addr, data)
F_Copy_Buffer_to_RAM(start_addr, size)
F_Copy_Gang_Buffer_to_RAM(start_addr, size)
F_Copy_RAM_to_Gang_Buffer(start_addr, size)
F_Set_PC_and_RUN(PC_addr)
F_Synch_CPU_JTAG()
F_Get_Targets_Vcc()
F_Get_Targets_Result()
F_Get_Active_Targets_Mask()
F_Disable_FPA_index(fpa)
F_Enable_FPA_index(fpa)
F_Customize(dest, data)

See chapter 4 for detailed description of the instructions listed above.

Note: *Not all instructions listed in the chapter 4 are implemented in the command line interpreter. For example - all instructions uses pointers are not implemented, however this is not limiting the access to all features of the API-DLLs, because all instructions uses pointers are implemented also in the simpler way without pointers.*