GangPro-CC Flash Programmer

for the CC series devices - Chipcon product from TI

Remote Control Programming User's Guide

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.

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.

FC

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.

C E

Table of Contents

1. Introduction

      GangPro-CC 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 eight Flash Programming Adapters (FPAs) significantly reducing programming time in production.

      Figure 1.1 shows the connections between PC and up to eight 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.
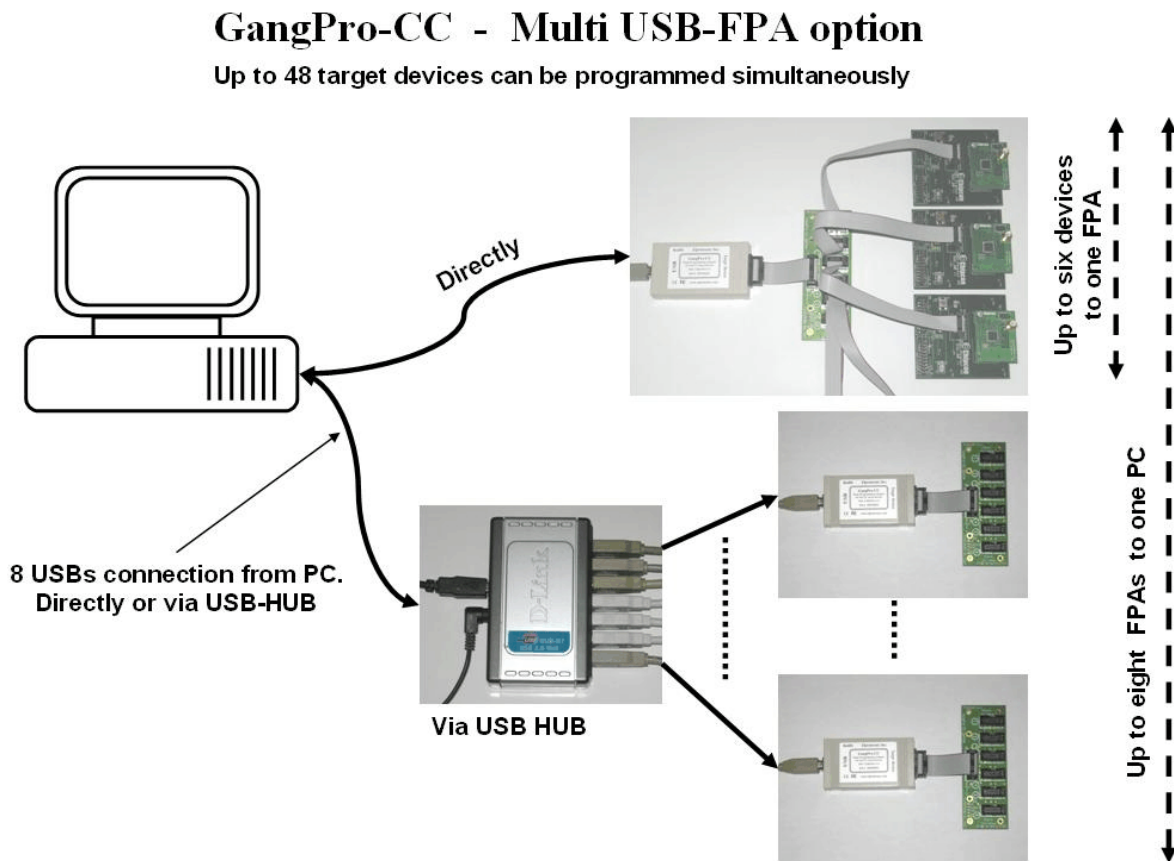
Figure 1.1

Block diagram of the Multi-FPA application DLL is presented on the Figure 1.2.

To support the Multi-FPA API-DLL feature, the software package contains nine dll files

GangPro-CC Multi-FPA API-DLL

Application Software (C++, LabVIEW etc.)

Multi-FPA API-DLL

Multi-FPA DLL Selector and Task Manager

API-DLL — API-DLL .... API-DLL

Simultaneous processes

USB-1 USB-2 USB-8

FPA-1 ...... FPA-2 .... FPA-8

Device-6 Device-12 .... Device-48
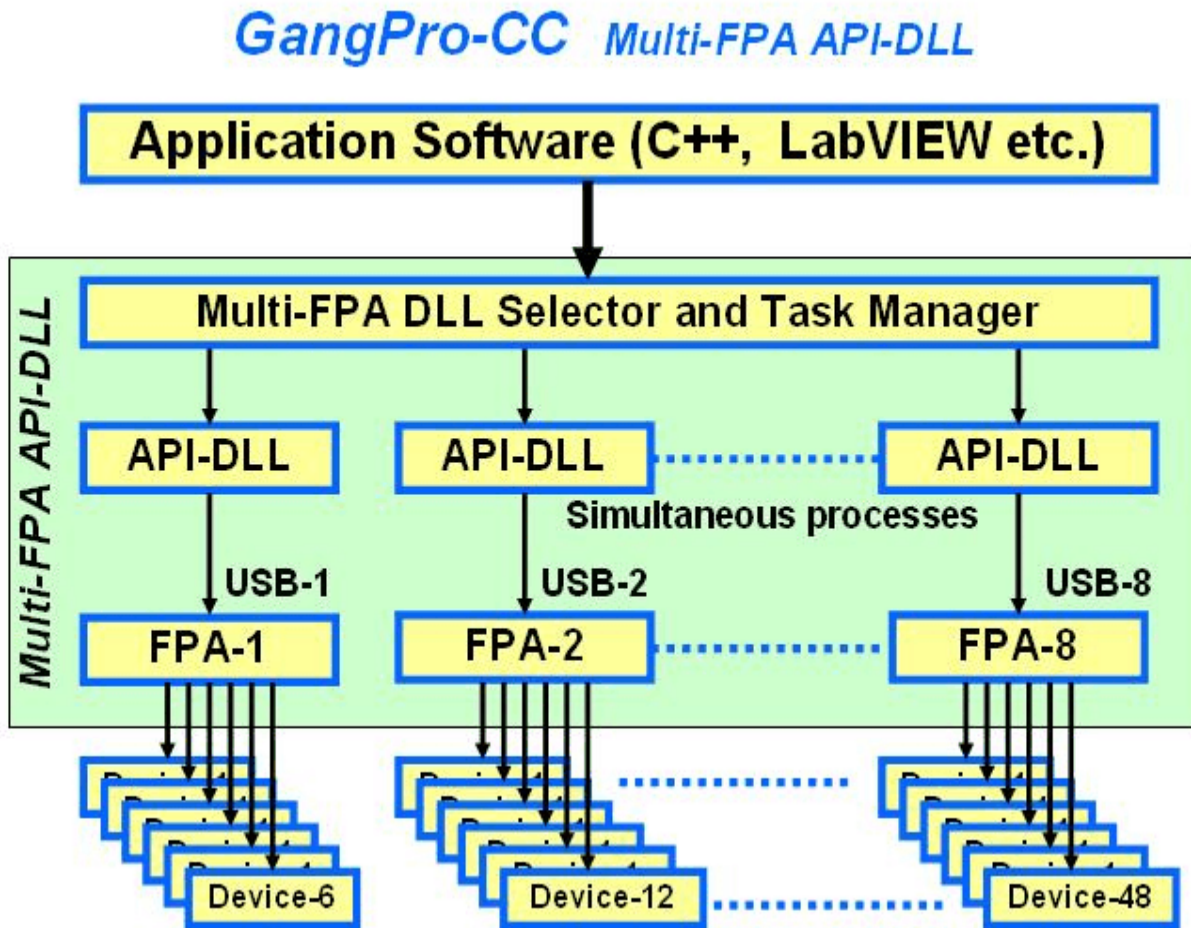
Figure 1.2

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

Figure 1.3 shows the logical connections between dll files.

GangPro-CC Multi FPA API-DLL
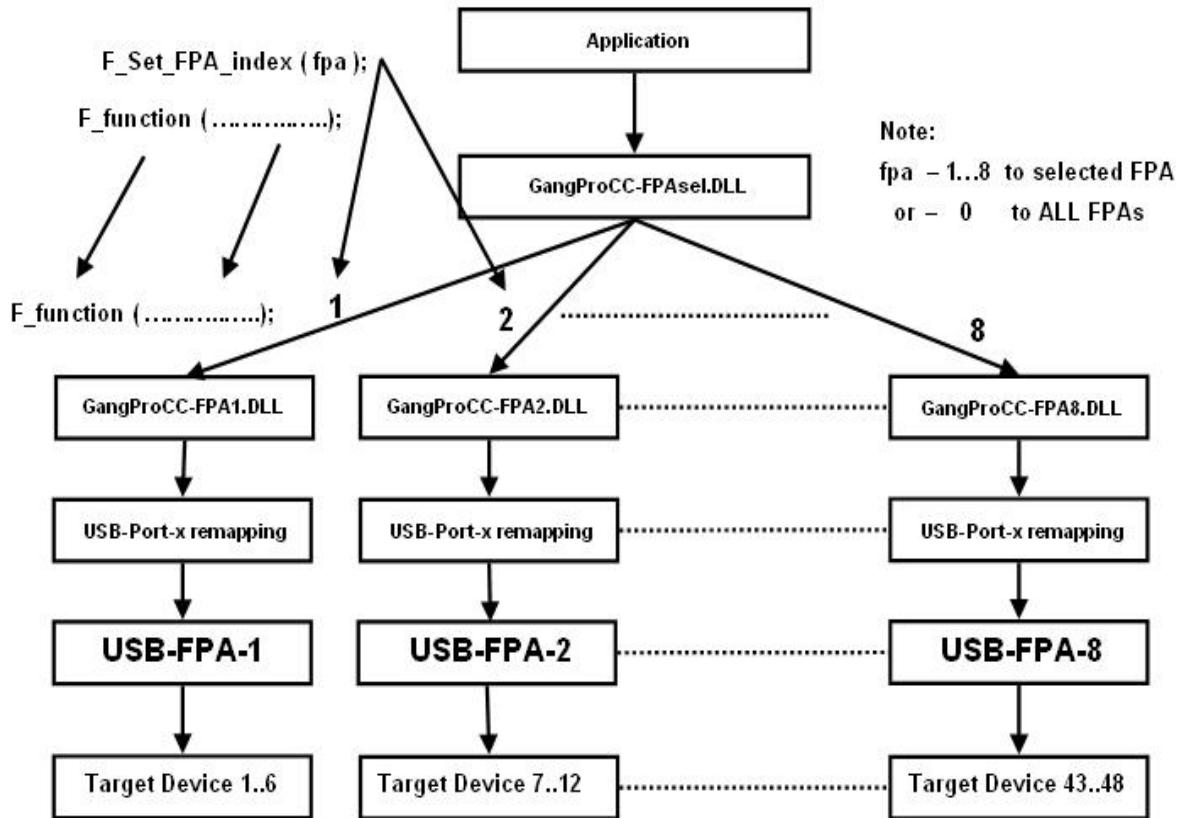
Figure 1.3

The main GangProCC-FPAsel.dll (Multi-FPA selector) allows to transfer API-DLL functions coming from an application software to desired single application dll (GangProCC-FPA1.dll to GangProCC-FPA8.dll).

Note: *Software package contains one GangProCC-FPA1.DLL. Files GangProCC-FPA2.DLL to GangProCC-FPA8.DLL will be copied automatically if required.*

The GangProCC-FPAsel.dll is transparent for all API-DLL functions implemented in the single API-DLLs functions. Desired destination FPA can be selected using the selector function added to the Multi-FPA selector (GangProCC-FPAsel.dll).

F_Set_FPA_index(fpa);

where the

fpa = 1 to 8 when the 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 8 is used, then the result is coming back to application software from the single API-DLL via transparent Multi-FPA selector. When the 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 selector DLL is returning 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 8 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 8). 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-8 but that process can not be seen from the application software.

        When the inactive fpa index is selected, then return value from the 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 eight FPAs and on the startup activate only desired number of FPAs.

        It should be noticed, that all single API-DLLs (GangProCC-FPA1.dll to GangProCC-FPA8.dll) used with the Multi-FPA DLL (GangProCC-FPAsel.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 GangPro-CC 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:

C:\Program Files\Elprotronic\CCxx\USB GangPro-CC\API-DLL

GangProCC-FPAsel.dll	- Multi-FPA selection / task manager API- DLL
GangProCC-FPA1.dll	- Single api-DLL for the UAB-FPA
GangProCC-Dll.h	- header file for C++
GangProCC-Dos-Dll.h	- header file for C++ (Borland) or DOS

GangProCC-FPAsel.lib	- lib file for C++
GangProCC-FPAsel-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 GangProCC-FPAsel.dll contains two groups of the same functions used in C++ application and Visual Basic (or similar) applications All procedure names used in Visual Basic are starting from **VB_xxxx**, when the 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 the C++ application is created, then following files should be copied to the source application directory:

| GangProCC-Dll.h | - header file for C++ |
| GangProCC-FPAsel.lib | - lib file for C++ (Microscoft Visual C++) |

or

| GangProCC-Dos-Dll.h | - header file for C++ |
| GangProCC-FPAsel-BC.lib | - lib file for C++ (Borland C++) |

and to the release/debug application directory

GangProCC-FPAsel.dll	- Multi-FPA selection / task manager DLL
GangProCC-FPA1.dll	- API-DLL for the USB-FPA
config.ini	- default configuration file for the FPAs

FPAs-setup.ini   - FPAs- vs USB ports configuration file

Executable application software package in C++ or when application in Visual Basic is created, then following files should be copied to the source or executable application directory:

GangProCC-FPAsel.dll
GangProCC-FPA1.dll
config.ini
FPAs-setup.ini

All these files 'as is' should be copied to destination location, where an application software using the DLL library.
The config.ini file has default setup information. This file can be modified and taken directly form the GangPro-CC Flash Programmer application software. To create required config.ini file the standard GangPro-CC Flash programmer software should be open and required setup (memory option, 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. All files and source code are located in:

   C:\Program Files\Elprotronic\CCxx\USB GangPro-CC\API-DLL-Demo\Cpp\Demo-1
and
   C:\Program Files\Elprotronic\CCxx\USB GangPro-CC\API-DLL-Demo\Cpp\Demo-8x6

Figure 1.4 shows the logical connections between flash programming adapter (FPA) and six target devices. This connection can be done using the GangPro-CC splitter (see schematic presented on figure 1.5)
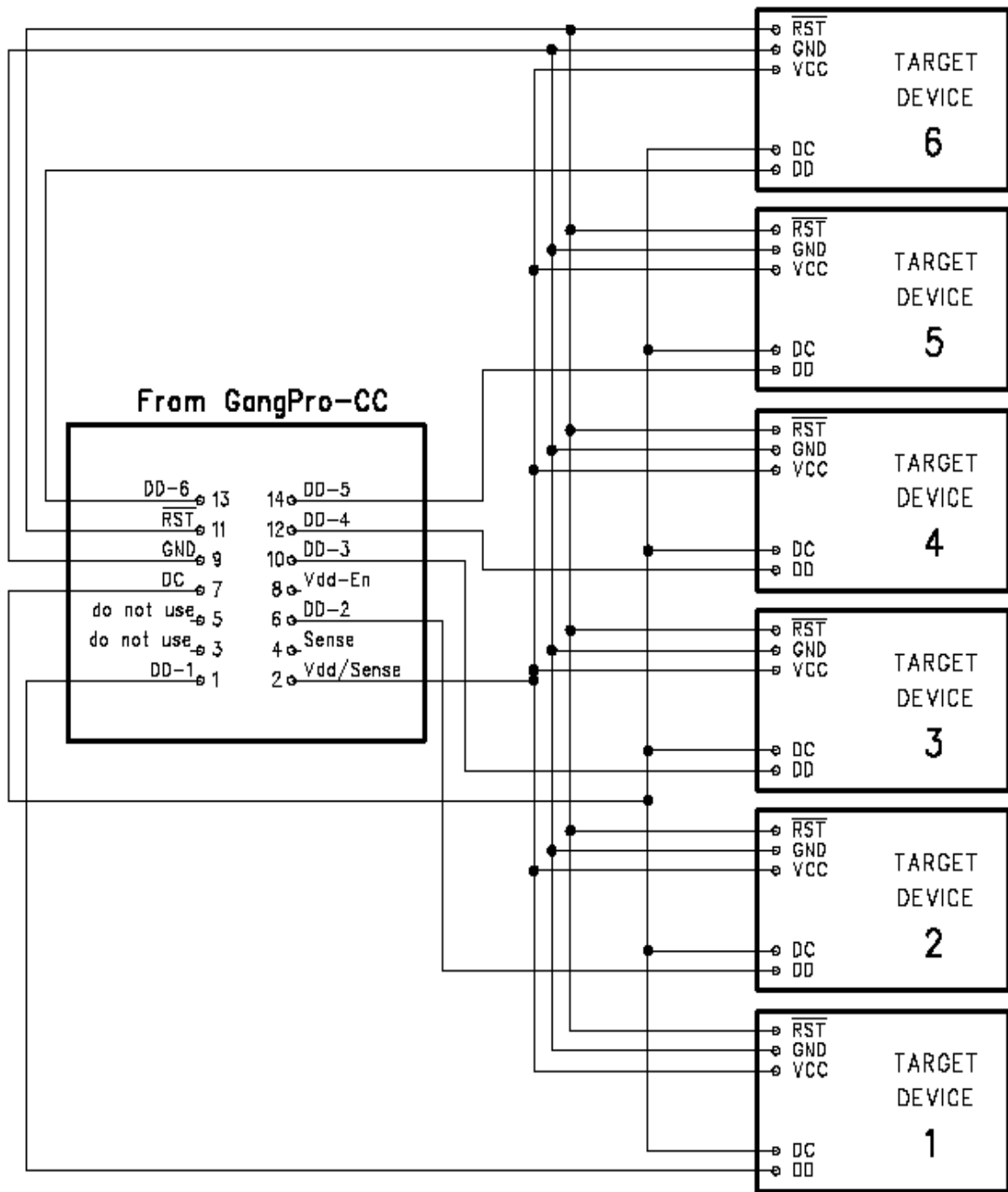
Figure 1.4
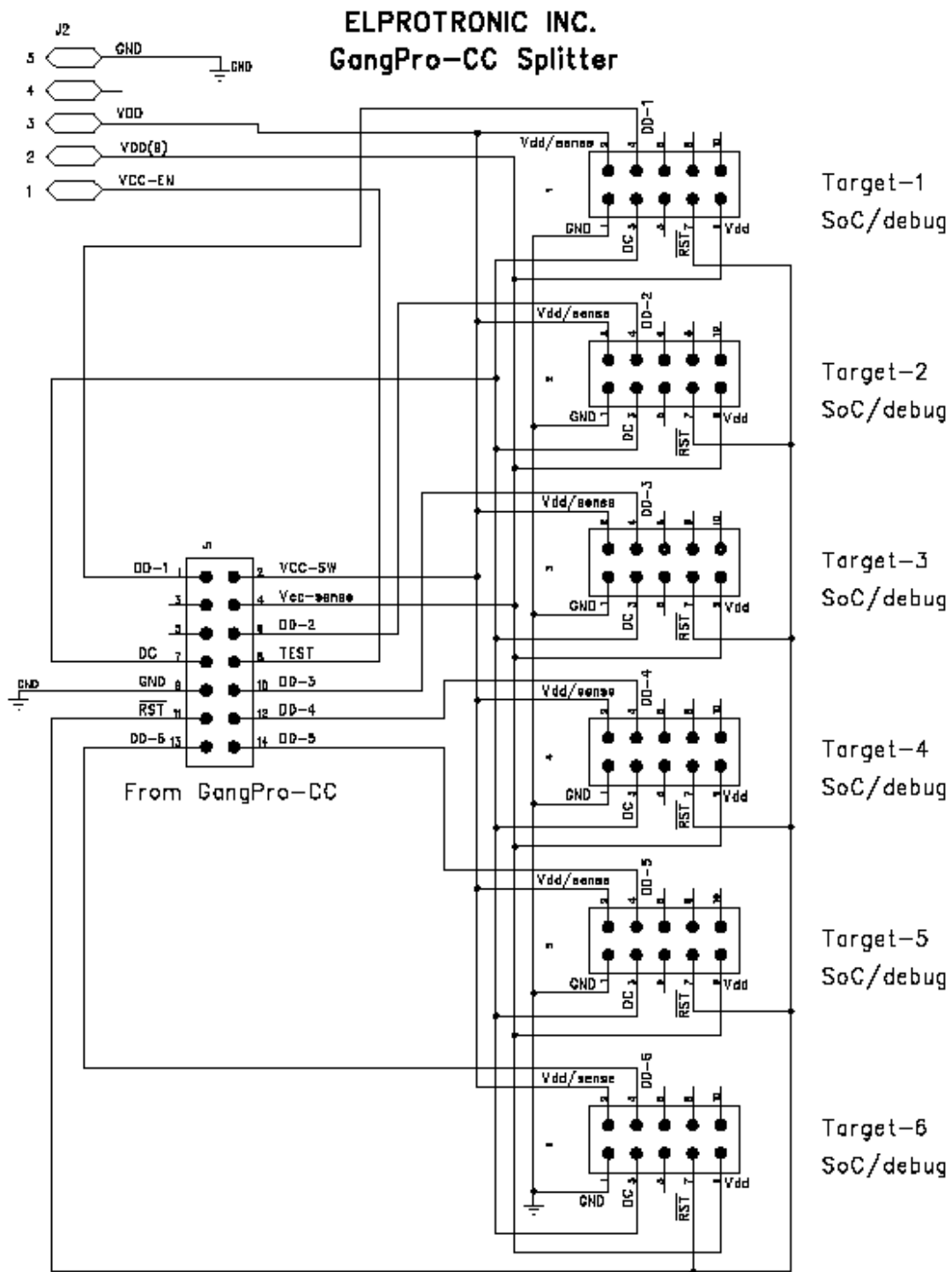
Figure 1.5

2. Demo program

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:\Program Files\Elprotronic\CCxx\USB GangPro-CC\API-DLL-Demo\Cpp\Demo-1

Program can be activated by selecting the GangProCC-DLL-DemoCpp.exe located in the \release subdirectory. This demo program can also be activated from the windows menu:

Start->Program->Elprotronic-Flash Programmer->(CCxx) USB GangPro-CC->API-DLL-Demo-Cpp
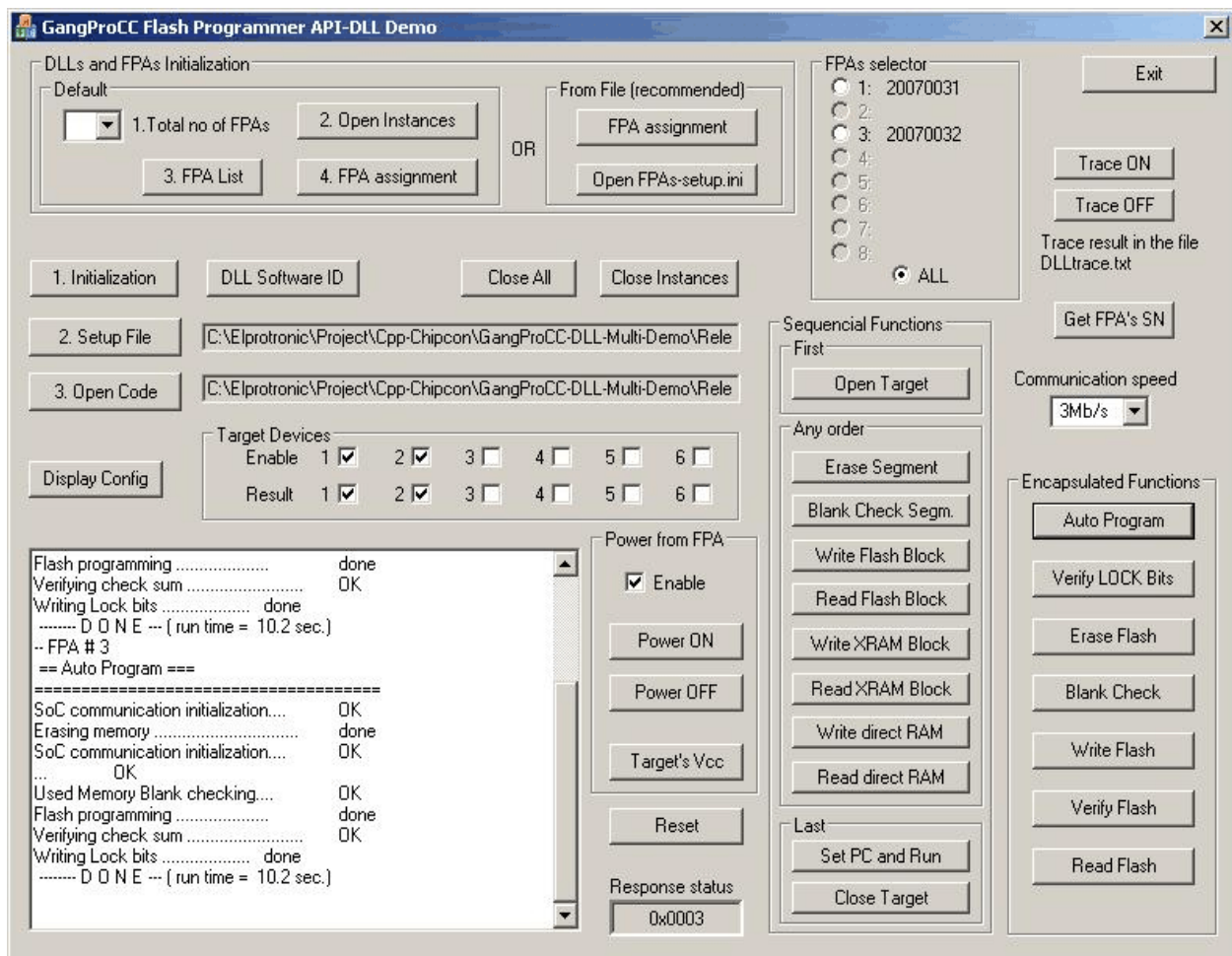
Figure 2.1   Demo program dialogue screen using DLLs.

When the demo program is activated, then the following dialogue screen is displayed (see figure 2.1). At the beginning the USB-FPA's configuration file should be created, that contains list off all FPAs used in the application. Using the *Notepad* editor the default FPA configuration file *'FPAs-setup.ini'* should be opened by pressing the *"Open FPAs-setup.ini"* button in the dialogue screen.

Take a serial numbers from the FPAs labels and write it on the desired FPAs locations FPA-1 up to FPA-8. If for example two FPAs will be used with SN 20070031 and 20070032 then the contents of the FPA's configuration file will be as follows:

```
;=======================================================================
; USB-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-8     *
;   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   20070031
FPA-2   20070032
```

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 **GangProCC-DLL-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 assigment'* located inside the frame named *'From File (recommended)'.* 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 8). Number of instances to be opened is calculated automatically, one per available and valid FPA. On described example with two FPAs in the *'FPAs selector'* will display two valid FPAs with list of used FPAs' serial numbers. All, others FPA-x fields will be disabled. In this example only two DLL instances becomes opened. Valid FPA indexes becomes 1,3 and ALL.

> *Note: When one or more FPA adapters are connected to PC and the "FPAs-setup.ini" does not contain valid FPA serial numbers, then the first detected FPA (and only one) will be activated.*

Other method (old method - not recommended) that allows to open required number of instances uses *'2. Open Instances'* button in *'Default'* 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 FPA 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 assigment'* located inside the frame named *'From File (recommended)*).

When the FPA(s)s has been assigned then all adapters should be activated by pressing the *'1. Initialization'* button. This initialization calls the DLL function **F_Initialization** and communication between programming adapter and PC is established. Report message is displayed in the report window (uses the **F_ReportMessage** function). By default the *config.ini* file is empty and to make a required programmer setup the setup file should be downloaded to programmer. It can be done by pressing the button *' 2.Setup File'*, (executing **F_ConfigFileLoad** DLL function). Setup file can be prepared first using standard GangPro-CC programming software with GUI. Also desired code file can be downloaded by pressing the 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_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 GangPro-CC Flash Programming software (GUI) is executed. Also buttons *Power ON/OFF, RESET* has the same action as related buttons in standard programmer. Refer to the *GangPro-CC Flash Gang Programmer for the CC series devices - User's Manual* for details of these functions.

In the central part on dialogue screen there are buttons that can call the sequential DLL functions.

Button Open Target	- **F_Open_Target_Device();**
Button Erase Segment	- **F_Segment_Erase(....);**
Button Blank Check Segm.	- **F_Sectors_Blank_Check(....);**
Button Write Flash Block	- **F_Copy_Buffer_to_Flash(....);**
and	- **F_Copy_Gang_Buffer_to_Flash(....);**
Button Read Flash Block	- **F_Copy_Flash_to_Gang_Buffer(....)**
Button Write XRAM Block	- **F_Copy_Buffer_to_XRAM(....);**
and	- **F_Copy_Gang_Buffer_to_XRAM(....);**
Button Read XRAM Block	- **F_Copy_XRAM_to_Gang_Buffer(....);**
Button Write direct RAM	- **F_Copy_Buffer_to_direct_RAM(....);**
and	- **F_Copy_Gang_Buffer_to_direct_RAM(....);**
Button Read direct RAM	- **F_Copy_direct_RAM_to_Gang_Buffer(....);**
Button Set PC and Run	- **F_Set_PC_and_RUN(....);**
Button Close Target	- **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. On the end of sequential communication the button *Close Traget* (calling **F_Close_Target_Device** DLL function) should be pressed.

In the 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 (segment size 1 or 2k)

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 - the same data to all target devices using function

        F_Copy_Buffer_to_Flash(0x1020, 8);

and write 8 bytes to the flash memory at location 0x1028 to 0x102F - unique data to each target devices using function

        F_Copy_Gang_Buffer_to_Flash(0x1028, 8);

Following data are written no targets:
Target no 1 -> 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
Target no 2 -> 10 11 12 13 14 15 16 17 28 29 2A 2B 2C 2D 2E 2F
Target no 3 -> 10 11 12 13 14 15 16 17 38 39 3A 3B 3C 3D 3E 3F
Target no 4 -> 10 11 12 13 14 15 16 17 48 49 4A 4B 4C 4D 4E 4F
Target no 5 -> 10 11 12 13 14 15 16 17 58 59 5A 5B 5C 5D 5E 5F
Target no 6 -> 10 11 12 13 14 15 16 17 68 69 6A 6B 6C 6D 6E 6F

Read Flash Block   Read 64 bytes from each target devices 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 XRAM Block   Write 8 bytes to XRAM at location 0xF000 to 0xF007 - the same data to all target devices using function

        F_Copy_Buffer_to_XRAM(0xF000, 8);

and write 8 bytes to XRAM at location 0xF008 to 0xF00F - unique data to each target devices using function

        F_Copy_Gang_Buffer_to_XRAM(0xF008, 8);

Following data are written no targets:
Target no 1 -> 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
Target no 2 -> 30 31 32 33 34 35 36 37 48 49 4A 4B 4C 4D 4E 4F
Target no 3 -> 30 31 32 33 34 35 36 37 58 59 5A 5B 5C 5D 5E 5F
Target no 4 -> 30 31 32 33 34 35 36 37 68 69 6A 6B 6C 6D 6E 6F
Target no 5 -> 30 31 32 33 34 35 36 37 78 79 7A 7B 7C 7D 7E 7F
Target no 6 -> 30 31 32 33 34 35 36 37 88 89 8A 8B 8C 8D 8E 8F

Read XRAM Block   Read 16 bytes XRAM starting from address at 0xF000. On the report screen 16 bytes from each target devices taken from addresses 0xF000 to 0xF00F are displayed

Write direct RAM   Write 8 bytes to direct RAM at location 0x60 to 0x67 - the same data to all target devices using function

           F_Copy_Buffer_to_direct_RAM(0x60, 8);

and write 8 bytes to direct RAM at location 0x68 to 0x6F - unique data to each target devices using function

           F_Copy_Gang_Buffer_to_direct_RAM(0x68, 8);

Following data are written no targets:
Target no 1 -> 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
Target no 2 -> 40 41 42 43 44 45 46 47 58 59 5A 5B 5C 5D 5E 5F
Target no 3 -> 40 41 42 43 44 45 46 47 68 69 6A 6B 6C 6D 6E 6F
Target no 4 -> 40 41 42 43 44 45 46 47 78 79 7A 7B 7C 7D 7E 7F
Target no 5 -> 40 41 42 43 44 45 46 47 88 89 8A 8B 8C 8D 8E 8F
Target no 6 -> 40 41 42 43 44 45 46 47 98 99 9A 9B 9C 9D 9E 9F

Read direct RAM   Read 16 bytes direct RAM starting from address at 0x60. On the report screen 16 bytes from each target devices taken from addresses 0x60 to 0x6F are displayed

Set PC and Run   Small program is written and downloaded to each target devices.
Following program is saved in the XRAM at location stated from 0xF100.

```
//Downloaded code to RAM - address 0xF100
  0xC2, 0xAF,         // clr    EA (Enable mask)
  0xE4,               // clr    A
  0x90, 0xF0, 0x00,   // mov  DPTR, #0xF000
  0x79, 0x10,         // mov  R1, #0x10
loop:
  0xF0,               // movx  @dptr,A
  0x04,               // inc    A
  0xA3,               // inc    dptr
  0xD9,0xFB,          // DJNZ        R1,loop
//done, fake a breakpoint
  0xA5,               // DB    0xA5
  0x00,               // NOP
```

Written program is modifying contents of the XRAM at locations 0xF000 to 0xF00F. The new XRAM vales should be 0x00, 0x01, 0x02,..... 0x0F. When the Set PC and Run program is pressed, then the contents described above is downloaded to XRAM and function

Set_PC_and_RUN(1, 0xF100);

is executed. When program is finished, then XRAM at location should be modified. XRAM contents can be read by pressing the ***Read XRAM Block*** button.

To make a test do following steps (from the beginning):

1. FPA List
2. Initialization
3. Setup File
4. Power from FPA - Enable
5. Open Target
6. Write XRAM Block
7. Read XRAM Block (remember XRAM contents)
8. Set PC and RUN
9. Read XRAM Block (compare with the contents from point 7).

On the right part of the dialogue screen are located buttons with encapsulated functions like Autoprogram, Erase Flash etc. Encapsulated functions are not requires to call the **F_Open_Target_Device**() function. All functions, including "Open..", "Close..." are build-in inside the encapsulated functions (see chapter 4 for details).

The second demo program is a 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\CCxx\USB GangPro-CC\API-DLL-Demo\Cpp\Demo-8x6

Program can be activated by selecting the GangProCC-8x6-Demo.exe in the \release subdirectory, or can be activated from the windows menu:

Start->Program->Elprotronic-Flash Programmer->(CCxx) USB GangPro-CC->API-DLL-8x6-Demo-Cpp

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. At the beginning the *"1. FPA assignment"* button should be pressed and *"FPA-setup.ini"* file selected. Software will assign all specified FPAs to USB ports. If process is finished, then the *"2. Initialization", "3 Setup file"* and *"4.Open*
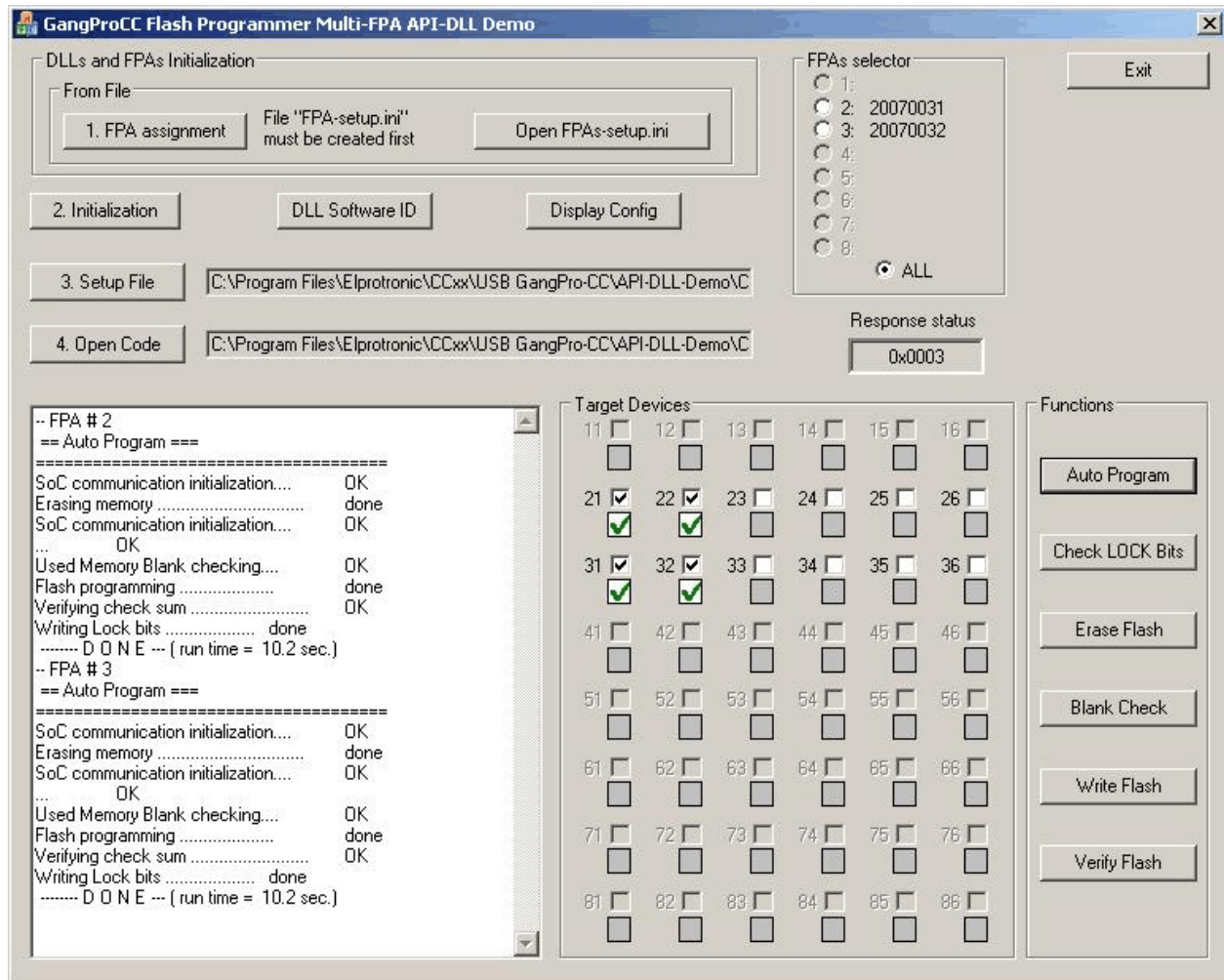
Figure 2.2 - Multi-FPA API-DLL demo dialogue screen.

Code" buttons should be used.

> **Note:** *Desired Gang Programmers configuration can be created using standard GangPro-CC (GUI) software. The required configuration setup should be created (select device type, memory options etc) and saved using option* **"Save Setup us.."**. *Saved configuration file can be used "as is" by the Multi-FPA API-DLL (used in the described demo software above).*

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. Programming results will be displayed in the report screen and in the status icons - each per one target device.

3. Getting Started

The Multi-FPA API-DLL software package is the same for the application written under Visual C++, Visual Basic, LabView etc. When the desired application is created then all files from the Elprotronic directory

C:\Program Files\Elprotronic\CCxx\USB GangPro-CC\API-DLL

should be copied to the executable destination directory.

It is recommended to start the standard GangPro-CC (GUI) programming software to verify if the hardware and the drivers setup are correct. Using the GangPro-CC programming software the fully functional setup that satisfy desired requirements should be created. When it is done then using the *"File->Save Setup as.."* option the configuration file should be saved. This file can be used "as is" in the application uses Multi-FPA API-DLL. Copy and paste the required configuration to your destination directory, where your application software is installed. It is recommended to use the demo program and verify if the setup in your PC and destination directory are done correctly. To do that copy the executable file

       GangProCC-DLL-DemoCpp.exe

 from location

C:\Program Files\Elprotronic\CCxx\USB GangPro-CC\API-DLL-Demo\Cpp\Demo-1\release

to your destination location where your application software is installed. 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 API-DLL always uses two API-DLL - selector DLL and AOI-DLL. When one FLA is used only, then selector DLL always sould select the first FPA. Also it is recommended to use the first detected FPA, since only one FPA is typically connected to PC. The application software can be simplified in this case. 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.

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 the FPA, and run an autoprogram with the same features like an autoprogram in the standard GangPro-CC (GUI) software.

1. Download data to target device

```
F_OpenInstancesAndFPAs( "*# *" );        // DLL and FPA initialization
F_Set_FPA_index( 1 );                    // select FPA-1
F_Initialization( );                     // init FPA
F_ReadConfigFile(  filename );           // read configuration data and save
                                         // to API-DLL
F_ReadCodeFile(  format, filename );     // read code data and save to DLL
  do
   {
    status = F_AutoProgram( 1 );
         //start autoprogra
    if ( status != TRUE )
     {
      status = F_LastStatus( 1 );
      .........................................
     }
   } 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 the application software to desired FPA. 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.

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 the 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 an autoprogram in the standard GangPro-CC (GUI) 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
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
..................................................................
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
```

```
.............................................................
 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
 .............................................................
 F_CloseInstances();
```

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 defined as 4 bytes long (see header file)

#define   INP_X   _int32
Make sure that an application using the DLL file has the same length of desired data.

The DLL instructions have the following convention related to the target device number and target device mask set/result (see also figure 1.4 and 1.5).

Table 4.1

DD connector pin #	Debug Data #	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:

All target devices connected to the Gang Programmer should be the same type. Do not mixed the different type of the target devices.

4.1 Multi-FPA instructions

The Multi-FPA API-DLL instructions are related to Multi-FPA selector only. These instructions allows to initialize all single applications 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 activate the tracing.

Syntax:

```
void    MSPPRG_API      F_Trace_ON( void );
```

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 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.

Syntax:

```
void    MSPPRG_API      F_Trace_OFF( void );
```

F_OpenInstances

F_OpenInstances - API-DLL initialization in the PC.

Instruction must be called first - before all other instruction. Instead this function the F_OpenInstancesAndFPAs can be used.

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 te 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 = 16
```
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 = 16
```
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 configuartion 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 te 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
 or
3.    // scanned available FPA's SN list
```

```
long SN[MAX_USB_DEV_NUMBER+1], Snr[MAX_USB_DEV_NUMBER+1];
CString snlist;
char * buf[20];

      F_OpenInstances( 1 );   // DLL initialization – one instance
      F_Set_FPA_index( 1 );   // select access to the first instance
      n = 0;        //number of detected FPAs
      for( k=1; k<=MAX_USB_DEV_NUMBER ; k++ )
       {
        SN[k] = F_Check_FPA_access(k);
               if ( SN[k] > 20000000 ) n++;
       }
      F_CloseInstances();

      // write your own procedure .......
      // remap available FPAs SN to desired FPAs order from SN[k] to Snr[p]

      snlist = "*#";
      for( k=1; k<=n ; k++ )
      {
        sprintf( buf, " %8.8li", Snr[k] );
        snlist += buf;
      }
      F_OpenInstances_AndFPAs( snlist );
```

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-8
;   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_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 like MSPlist.ini 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_APIDLL_Directory( char*  APIDLLpath );
```

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

F_Set_FPA_index

F_Set_FPA_index - Select desired FPA index (desired DLL instance)
VALID FPA index - *(1 to 8) or 0 (ALL FPAs).*
Syntax:
```
     INT_X      MSPPRG_API  F_Set_FPA_index ( BYTE fpa );
```

Parameters:
```
     fpa -> 1 to MAX_FPA_INDEX  where MAX_FPA_INDEX  = 8
           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)
```

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_Disable_FPA_index

F_Disable_FPA_index   - Disable desired FPA index (desired DLL instance)
VALID FPA index   - (1 to 8)

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  = 8
```

F_Enable_FPA_index

F_Enable_FPA_index   - Enable desired FPA index (desired DLL instance)
VALID FPA index   - (1 to 8)

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  = 8
```

F_LastStatus

F_LastStatus   - Get current FPA index

VALID FPA index   - (1 to 8)

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

Parameters:
```
fpa – FPA index of the desired status
      fpa index -> 1..8
```
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<=8; n++)   status[n] = F_LastStatus(n);
where n -> desired FPA index
and get the last status data from FPA-1, 2, .. up to .8

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 = 0x1000   - Single DLL for the Parallel Port MSP430-FPA
DLL ID = 0x2000   - Single DLL for the USB MSP430-FPA (FlashPro430)
DLL ID = 0x3000   - Single API-DLL for the GangPro430
DLL ID = 0x4000   - Single API-DLL for the FlashPro-CC
DLL ID = 0x5000   - Single API-DLL for the GangPro-CC
DLL ID = 0x6000   - Multi-FPA API-DLL for the FlashPro430
DLL ID = 0x7000   - Multi-FPA API-DLL for the GangPro430
DLL ID = 0x8000   - Multi-FPA API-DLL for the FlashPro-CC
DLL ID = 0x9000   - Multi-FPA API-DLL for the GangPro-CC
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:
```
INT_X       MSPPRG_API  F_Get_FPA_SN ( BYTE fpa );
```

Parameters:
```
fpa - FPA index of the desired status
     fpa index -> 1..8
```
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 16)

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 te communication with the FPA adapter without modifying the USB ports assignment.

VALID FPA index (DLL instance number) - (1 to 8)

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 16 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    INT_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 = 16
```

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_DLLTypeVer

F_DLLTypeVer   - Get information about DLL software type and software revision.

VALID FPA index   - (1 to 8)

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 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 Parallel Port MSP430-FPA
     DLL ID = 0x2000   - Single DLL for the USB MSP430-FPA (FlashPro430)
     DLL ID = 0x3000   - Single API-DLL for the GangPro430
     DLL ID = 0x4000   - Single API-DLL for the FlashPro-CC
     DLL ID = 0x5000   - Single API-DLL for the GangPro-CC
     DLL ID = 0x6000   - Multi-FPA API-DLL for the FlashPro430
     DLL ID = 0x7000   - Multi-FPA API-DLL for the GangPro430
     DLL ID = 0x8000   - Multi-FPA API-DLL for the FlashPro-CC
     DLL ID = 0x9000   - Multi-FPA API-DLL for the GangPro-CC
     Version = (0x0FFF & VALUE)
```

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

F_Initialization

F_Initialization   - Programmer initialization.

VALID FPA index   - (1 to 8) 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 (for the USB version programmer) or Parallel Port becomes open (for the parallel port version programmer).

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_Close_All

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

VALID FPA index   - (1 to 8) 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 8)
See F_ConfigSetup description for more details.

Syntax:
```
            MSPPRG_API  INT_X  F_GetSetup( CFG_BLOCK *config );
```
Return value:
```
        0 - FALSE
        1 - TRUE
```

F_ConfigSetup

F_ConfigSetup   - Setup programmer's configuration.
VALID FPA index   - (1 to 8) 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 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

See **F_Set_Config(....)** for detailed description of the all configuration data contents.

```
typedef struct
{
        INT_X  DeviceIndex;
        INT_X  PowerTargetEn;
        INT_X  CommSpeedIndex;
        INT_X  ResetTimeIndex;
        INT_X  CustomResetPulseTime;
        INT_X  CustomResetIdleTime;
        INT_X  RstVccOffTime;
        INT_X  ApplicationStartIndex;
        INT_X  ApplicationRunTime;
        INT_X  FlashEraseModeIndex;
        INT_X  FlashReadModeIndex;
        INT_X  FlashLockBits;
        INT_X  UnlockDebugBit;
        INT_X  LockBitsEn;
        INT_X  VerifyModeIndex;
        INT_X  IEEEAddeModeIndex;
        INT_X  ManIEEEAddeModeIndex;
        INT_X  BeepOKEn;
        INT_X  VccIndex;
        INT_X  TargetEnMask;
        INT_X  RetainDataEn;
        INT_X RetainDataStartAddr;
        INT_X  RetainDataStopAddr;
        INT_X  EraseDefBlock1En;
        INT_X  EraseDefBlock1StartAddr;
        INT_X  EraseDefBlock1StopAddr;
        INT_X  EraseDefBlock2En;
        INT_X  EraseDefBlock2StartAddr;
        INT_X  EraseDefBlock2StopAddr;
        INT_X  EraseDefBlock3En;
        INT_X  EraseDefBlock3StartAddr;
        INT_X  EraseDefBlock3StopAddr;
        INT_X  EraseDefBlock4En;
        INT_X  EraseDefBlock4StartAddr;
```

```
        INT_X  EraseDefBlock4StopAddr;
        INT_X  ReadDefBlock1En;
        INT_X  ReadDefBlock1StartAddr;
        INT_X  ReadDefBlock1StopAddr;
        INT_X  ReadDefBlock2En;
        INT_X  ReadDefBlock2StartAddr;
        INT_X  ReadDefBlock2StopAddr;
        INT_X  ReadDefBlock3En;
        INT_X  ReadDefBlock3StartAddr;
        INT_X  ReadDefBlock3StopAddr;
        INT_X  ReadDefBlock4En;
        INT_X  ReadDefBlock4StartAddr;
        INT_X  ReadDefBlock4StopAddr;
        INT_X  Spare1;
        INT_X  Spare2;
        INT_X  Spare3;
        INT_X  Spare4;
        INT_X  Spare5;
        INT_X  Spare6;
        INT_X  Spare7;
        INT_X  Spare8;
        INT_X  Spare9;
        INT_X  Spare10;
        INT_X  Spare11;
        INT_X  Spare12;
        INT_X  Spare13;
        INT_X  Spare14;
        INT_X  Spare15;
        INT_X  Spare16;
} CFG_BLOCK;
```

Syntax:
```
 MSPPRG_API      INT_X  F_ConfigSetup( CFG_BLOCK config );
```

Return value:
```
        0 – FALSE
        1 – TRUE
```

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.

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

```
        F_GetSetup( &config );
                        //API-DLL  - get configuration from the programmer
        config.CommSpeedIndex = SPEED_3MB_INDEX;
                        //select JTAG interface
        config.FlashEraseModeIndex = ERASE_ALL_MEM_INDEX;
                        //select all memory erase option
        F_ConfigSetup( config );
                        //API-DLL  - setup configuration in the programmer
```

F_SetConfig

F_SetConfig   - Setup one item of the programmer's configuration.
VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Similar to the **F_ConfigSetup**, but only one selected item from the `CFG_BLOCK` structure is modified.

Syntax:
  `MSPPRG_API        INT_X  F_SetConfig( INT_X index, INT_X data );`

Return value:
```
        0 – FALSE
        1 – TRUE
```
Example:
```
  ......................
  ......................
      F_SetConfig( CFG_MICROCONTROLLER, config.uProcIndex );
  or
      F_SetConfig( CFG_MICROCONTROLLER, 7 );
```


`Index's list`

```
CFG_MICROCONTROLLER                 1
CFG_POWERTARGETEN                   2
CFG_COMM_SPEED_INDEX                3
CFG_RESET_TIME_INDEX                4
CFG_RESET_PULSE_TIME                5
CFG_RESET_IDLE_TIME                 6
CFG_RSTVCC_OFF_TIME                 7
CFG_APPLSTARTEN                     8
CFG_APPL_RUN_TIME                   9
CFG_FLASHERASEMODE                  10
```

```
CFG_FLASHREADMODE                11
CFG_FLASH_LOCKBIT                12
CFG_UNLOCK_DEBUG_BIT             13
CFG_LOCK_BITS_EN                 14
CFG_VERIFYMODE                   15
CFG_IEEE_ADDR_MODE               16
CFG_MAN_IEEE_ADDR_MODE           17
CFG_BEEP_OK_EN                   18
CFG_VCCINDEX                     19
CFG_TARGET_EN_MASK               20
CFG_RETAIN_DEF_DATA_EN           21
CFG_RETAIN_START_ADDR            22
CFG_RETAIN_STOP_ADDR             23
CFG_ERASE_DEFBLOCK1_EN           24
CFG_ERASE1_START_ADDR            25
CFG_ERASE1_STOP_ADDR             26
CFG_ERASE_DEFBLOCK2_EN           27
CFG_ERASE2_START_ADDR            28
CFG_ERASE2_STOP_ADDR             29
CFG_ERASE_DEFBLOCK3_EN           30
CFG_ERASE3_START_ADDR            31
CFG_ERASE3_STOP_ADDR             32
CFG_ERASE_DEFBLOCK4_EN           33
CFG_ERASE4_START_ADDR            34
CFG_ERASE4_STOP_ADDR             35
CFG_READ_DEFBLOCK1_EN            36
CFG_READ1_START_ADDR             37
CFG_READ1_STOP_ADDR              38
CFG_READ_DEFBLOCK2_EN            39
CFG_READ2_START_ADDR             40
CFG_READ2_STOP_ADDR              41
CFG_READ_DEFBLOCK3_EN            42
CFG_READ3_START_ADDR             43
CFG_READ3_STOP_ADDR              44
CFG_READ_DEFBLOCK4_EN            45
CFG_READ4_START_ADDR             46
CFG_READ4_STOP_ADDR              47
CFG_IEEE_ADDR_LOCATION           48
CFG_IEEE_ADDR_LOC_MODE           49
CFG_IEEE_ADDR_LSB_FIRST          50



// -------------- CONFIG_BLOCK - definitions ----

//  CFG_MICROCONTROLLER        1

CC_ANY                     0
```

```
// 1  - CC1110F8
// 2  - CC1110F16
// 3  - CC1110F32
// 4  - CC2430F32
// 5  - CC2430F64
// 6  - CC2430F128
// 7  - CC2431F32
// 8  - CC2431F64
// 9  - CC2431F128
// 10 - CC2510F8
// 11 - CC2510F16
// 12 - CC2510F32
// 13 - CC2511F8
// 14 - CC2511F16
// 15 - CC2511F32


// CFG_POWERTARGETEN          2
//      0 -> PowerTarget Disable
//      1 -> PowerTarget Enable


//    CFG_COMM_SPEED_INDEX    3
SPEED_3MB_INDEX          1
SPEED_1MB_INDEX          2


//    CFG_RESET_TIME_INDEX    4
RESET_10MS_INDEX              0
RESET_100MS_INDEX            1
RESET_200MS_INDEX            2
RESET_500MS_INDEX            3
RESET_CUSTOM_INDEX          4
RESET_TOGGLE_VCC_INDEX      5


// CFG_RESET_PULSE_TIME       5
      time in ms


// CFG_RESET_IDLE_TIME        6
      time in ms


// CFG_RSTVCC_OFF_TIME        7
      time in ms


// CFG_APPLSTARTEN                 8

APPLICATION_KEEP_RESET        0
APPLICATION_TOGGLE_RESET      1
APPLICATION_TOGGLE_VCC        2
APPLICATION_SOFT_RESET        3


// CFG_APPL_RUN_TIME          9
```

```
        time in ms

// CFG_FLASHERASEMODE          10
ERASE_NONE_MEM_INDEX           0
ERASE_ALL_MEM_INDEX            1
ERASE_INFILE_MEM_INDEX         2
ERASE_DEF_CM_INDEX             3


// CFG_FLASHREADMODE           11
READ_ALL_MEM_INDEX             0
READ_PRGMEM_ONLY_INDEX         1
READ_INFOMEM_ONLY_INDEX        2
READ_DEF_MEM_INDEX             3


// CFG_FLASH_LOCKBIT           12
// CFG_UNLOCK_DEBUG_BIT        13
// CFG_LOCK_BITS_EN                    14


// CFG_VERIFYMODE              15
VERIFY_NONE_INDEX              0
VERIFY_STD_INDEX               1
VERIFY_FAST_INDEX             2


// CFG_IEEE_ADDR_MODE          16
AP_IEEE_ADDR_DISABLE           0
AP_WR_NEW_IEEE_ADDR            1
AP_RETAIN_CODE_WR_IEEE         2
AP_IEEE_ADDR_BLANK             3
AP_RETAIN_IEEE_ADDR            4
AP_ASSIGN_WR_IEEE_ADDR         5
AP_WR_IEEE_ADDR_FROM_FILE      6


// CFG_MAN_IEEE_ADDR_MODE      17
AP_IEEE_ADDR_DISABLE           0
AP_WR_NEW_IEEE_ADDR            1
AP_RETAIN_CODE_WR_IEEE         2


// CFG_BEEP_OK_EN              18


// CFG_VCCINDEX                19
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_TARGET_EN_MASK          20
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_RETAIN_DEF_DATA_EN        21
   0-disable  1-enable


//     CFG_RETAIN_START_ADDR         22
   0x00000 to 0x1FFFF


//     CFG_RETAIN_STOP_ADDR          23
   0x00000 to 0x1FFFF


//     CFG_ERASE_DEFBLOCK1_EN        24
   0-disable  1-enable


//     CFG_ERASE1_START_ADDR         25
   0x00000 to 0x1FFFF


//     CFG_ERASE1_STOP_ADDR          26
   0x00000 to 0x1FFFF


//     CFG_ERASE_DEFBLOCK2_EN        27
   0-disable  1-enable


//     CFG_ERASE2_START_ADDR         28
   0x00000 to 0x1FFFF


//     CFG_ERASE2_STOP_ADDR          29
   0x00000 to 0x1FFFF


//     CFG_ERASE_DEFBLOCK3_EN        30
   0-disable  1-enable


//     CFG_ERASE3_START_ADDR         31
   0x00000 to 0x1FFFF


//     CFG_ERASE3_STOP_ADDR          32
   0x00000 to 0x1FFFF


//     CFG_ERASE_DEFBLOCK4_EN        33
   0-disable  1-enable
```

```
//      CFG_ERASE4_START_ADDR        34
   0x00000 to 0x1FFFF


//      CFG_ERASE4_STOP_ADDR         35
   0x00000 to 0x1FFFF


//      CFG_READ_DEFBLOCK1_EN        36
   0-disable  1-enable


//      CFG_READ1_START_ADDR         37
   0x00000 to 0x1FFFF


//      CFG_READ1_STOP_ADDR          38
   0x00000 to 0x1FFFF


//      CFG_READ_DEFBLOCK2_EN        39
   0-disable  1-enable


//      CFG_READ2_START_ADDR         40
   0x00000 to 0x1FFFF


//      CFG_READ2_STOP_ADDR          41
   0x00000 to 0x1FFFF


//      CFG_READ_DEFBLOCK3_EN        42
   0-disable  1-enable


//      CFG_READ3_START_ADDR         43
   0x00000 to 0x1FFFF


//      CFG_READ3_STOP_ADDR          44
   0x00000 to 0x1FFFF


//      CFG_READ_DEFBLOCK4_EN        45
   0-disable  1-enable


//      CFG_READ4_START_ADDR         46
   0x00000 to 0x1FFFF


//      CFG_READ4_STOP_ADDR          47
   0x00000 to 0x1FFFF


//      CFG_IEEE_ADDR_LOCATION       48
   0x00000 to 0x1FFF8



//      CFG_IEEE_ADDR_LOC_MODE       49
   0-default  1-defined
```

```
//   CFG_IEEE_ADDR_LSB_FIRST        50
   0-disable  1-enable
```

F_GetConfig

F_GetConfig   - Get one item of the programmer's configuration.
VALID FPA index   - (1 to 8)

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

Syntax:
```
  MSPPRG_API        INT_X  F_GetConfig( INT_X index );
```

```
Index's list  – see F_SetConfig
```


Return value:
```
     Requested setup parameter;
```

Example:
```
   .....................
   .....................
     F_GetSetup( config );
     DeviceIndex = config.DeviceIndex;
or directly
     DeviceIndex = F_GetConfig( CFG_MICROCONTROLLER );
  .....................
```


F_DispSetup

F_DispSetup   - Copy programmer's configuration to report message buffer in text form.
VALID FPA index   - (1 to 8)

Syntax:
```
  MSPPRG_API        INT_X  F_DispSetup( void );
```


Return value:
```
     1 – TRUE;
```
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 8)

When any of the DLL functions is activated, a message is created and displayed on the dynamically created programmer's dialogue box. At the end of execution the dialogue 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

        REPORT_MESSAGE_MAX_SIZE   2000

characters is imported from the programmer software to the application software. Make sure to declare characters string length no less then REPORT_MESSAGE_MAX_SIZE characters. When the 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 REPORT_MESSAGE_MAX_SIZE 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:

```
#include "GangProCC-Dll.h";
char text[REPORT_MESSAGE_MAX_SIZE];
   .......................
   .......................
       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.

```
...........................................
#include "GangProCC-Dll.h";
Cstring Message = "";
...........................................

void CMspPrgDemoDlg::Disp_report_message()
{
  char text[REPORT_MESSAGE_MAX_SIZE];
    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 8)

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:
```
#include "GangProCC-Dll.h";
char text[REPORT_MESSAGE_MAX_SIZE];
INT_X k;
   .......................
   .......................
      for( k = 0; k< REPORT_MESSAGE_MAX_SIZE; 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.

```
.............................................
#include "GangProCC-Dll.h";
Cstring Message = "";
.............................................

void CMspPrgDemoDlg::Disp_report_message()
{
  char text[REPORT_MESSAGE_MAX_SIZE];
  INT_X k;
    for( k = 0; k< REPORT_MESSAGE_MAX_SIZE; 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_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 8) 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 Ccxx device family will be downloaded to the internal memory buffer. Any code data located outside memory space of the Ccxx device will be ignored and warning message will be created.

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)
```

  FileName: file name including path, file name and extention

Return value:
```
      (0xFFFe & info) | state
        where state is defined as follows:
            0 – FALSE
            1 – TRUE
         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 buffer and calculate the check sum.
VALID FPA index - (1 to 8).

Syntax:
```
 MSPPRG_API     INT_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.

Check Sum is calculated as an arithmetic sum of the 16-bits unsigned words form the valid code bytes. If the only one byte is present in the calculated word, then other byte is taken as a 0xFF. Check Sum result is 32 bits.

For example from the following code

      address   data

      0x0300   0xF2 0x12 0x23 0x34 0x78

Check Sum calculation

      Word 1   0x12F2

      Word 2   0x3423

      Word 2   0xFF78

      CS =   0x0001468D

Return value:

   Calculated check sum.

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 8) 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 GangPro-CC (GUI) Flash Programmer software. When the setup from the file is downloaded, then old configuration setup is overwritten. The 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 extention

Return value:
```
     0 – FALSE
     1 – TRUE
     (0xFFFe & info) | state
     where state is defined as follows:
          0 – FALSE
          1 – TRUE
   info is defined as follows:
      error ->  OPEN_FILE_OR_READ_ERR
```

Configuration file is a standard text file with the parameters name and value.

Example:
```
     st = F_ConfigFileLoad( "c:\test\configfile.cfg" );
     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 8) or 0 (ALL FPAs) executed sequentially.

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


Return value:
```
      0 – FALSE
      1 – TRUE
```


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 8) 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).

Syntax:
```
  MSPPRG_API  INT_X  F_Put_Byte_to_Code_Buffer( INT_X address, BYTE data );
```


Parameters value:
```
      code address – 0x0 to 0x1FFFF
      data         – 0x00 to 0xFF
```


Return value:
```
      0 – FALSE
      1 – TRUE
```


Example:
```
BYTE  code[MAX_FLASH_SIZE];
      .....................
      F_Clr_Code_Buffer();
      for( address = 0x0;  address < MAX_FLASH_SIZE; 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 8)

Instruction allows to read or verify contents of the code from code buffer

Syntax:
```
  MSPPRG_API      INT_X  F_Get_Byte_from_Code_Buffer( INT_X address );
```

Parameters value:
```
      code address – 0x0 to MAX_FLASH_SIZE-1 (0x1FFFF)
```

Return value:
```
      0x00 to 0xFF  – valid code data
      -1 (0xFFFF)   – code data not initialized on particular address
```

F_Put_IEEEAddr64_to_Gang_Buffer

F_Put_IEEEAddr64_to_Gang_Buffer   - Write IEEE address to Gang buffer.
VALID FPA index   - (1 to 8).

Instruction allows to write one unique IEEE address to Gang buffer. Contents of the IEEE address from the Gang buffers will be saved to target device when the F_Autoprogram(0) is executed and when in the configuration setup this option is enabled.

Syntax:
```
MSPPRG_API void F_Put_IEEEAddr64_to_Gang_Buffer(BYTE target_no, ULONG64 data );
```

Parameters value:
```
   target_no – 1 to 6
```

```
       data – 64 bits unsigned long integer number 0 to 0xFFFFFFFFFFFFFFFF
```

F_Put_IEEEAddr_Byte_to_Gang_Buffer

F_Put_IEEEAddr_Byte_to_Gang_Buffer - Write IEEE address to Gang buffer.
VALID FPA index   - (1 to 8).

Instruction allows to write one byte of the IEEE address to Gang buffer. Contents of the IEEE address from the Gang buffers will be saved to target device when the F_Autoprogram(0) is executed and when in the configuration setup this option is enabled.

Syntax:
MSPPRG_API INT_X F_Put_IEEEAddr64_to_Gang_Buffer(BYTE target_no, Byte no, BYTE data);

Instruction is functionally the same as the **F_Put_IEEEAddr64_to_Gang_Buffer**, but allows to transfer byte by byte of the 64 IEEE address to buffer. Function is used when the Visual Basic 6 is used, that not support the Int 64 bits data.
        no -> 0 to 7.
Index **no** MUST started from 0 and finished on 7 to transfer whole IEEEAddr.   When no = 0, the lowest byte of the IEEEAddr must be transferred. When no = 7, the highest byte of the IEEEAddr is transferred.

Parameters value:
```
   target_no – 1 to 6
   no – 0 to 7
   data – one byte from the 64 bits IEEE address
```

F_Get_IEEEAddr64_from_Gang_Buffer

F_Get_IEEEAddr64_from_Gang_Buffer - Read IEEE address contents from Gang buffer
VALID FPA index   - (1 to 8)

Syntax:
** MSPPRG_API ULONG64 F_Get_IEEEAddr64_from_Gang_Buffer(BYTE target_no);**

Parameters value:
```
     target_no – 1 to 6
```

Return value:

```
64 bits unsigned long integer number 0 to 0xFFFFFFFFFFFFFFFF
```

F_Get_IEEEAddr_Byte_from_Gang_Buffer

F_Get_IEEEAddr_Byte_from_Gang_Buffer - Read IEEE address contents from Gang buffer
VALID FPA index   - (1 to 8)

Syntax:
```
MSPPRG_API BYTE F_Get_IEEEAddr_Byte_from_Gang_Buffer( BYTE target_no, BYTE
no );
```

Instruction is functionally the same as the **F_Get_IEEEAddr64_from_Gang_Buffer**, but allows to transfer byte by byte of the 64 IEEE address from buffer. Function is used when the Visual Basic 6 is used, that not support the Int 64 bits data.

     no -> 0 to 7.

When no = 0, the lowest byte of the IEEEAddr is transferred. When no = 7, the highest byte of the IEEEAddr is transferred.

Parameters value:

```
target_no – 1 to 6
no – 0 to 7
```

Return value:

```
one byte of the IEEEAddr data
```

F_Get_Lock_Bits

F_Get_Lock_Bits   - Read Lock Bits taken from Gang buffer. Function
                                F_Verify_Lock_Bits() should be used first to read the clock bits
                                from target devices.

VALID FPA index   - (1 to 8)

Lock Protection bits

bit-4	0x10 -	Boot Block Lock		
		0	-	Page 0 is write protected
		1	-	Page 0 is writable, unless LSIZE is 000

bits3:1		LSIZE - Sets the size of the upper Flash area which is write-protected
bit 0	0x01	Debug lock bit

0	-	Disable debug command
1	-	Enable debug command

Syntax:

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

Parameters value:

```
target_no – 1 to 6
```

Return value:

```
0x100 | 8 bits Lock Bits value (see above)
       if debug bit is enabled or
0x100
       if debug bit is disabled (access to other bits is locked)
```

F_Power_Target

F_Power_Target   - Turn ON or OFF power from programming adapter to target device.

VALID FPA index   - (1 to 8)   or 0 (ALL FPAs) executed sequentially.

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

Note: `PowerTargetEn` *flag 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
```

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 8) 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
```
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 8)

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

Return value:
```
      INT_X – execution result for up to six target devices.
            TARGET_1_MASK      0x01
            TARGET_2_MASK      0x02
            TARGET_3_MASK      0x04
            TARGET_4_MASK      0x08
```

```
                    TARGET_5_MASK        0x10
                    TARGET_6_MASK        0x20

        Bit – 0 – false
        bit – set – true
```

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 8)

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

Return value:
```
      INT_X – execution result for up to six target devices – 0x00 to 0x3F.
            TARGET_1_MASK        0x01
            TARGET_2_MASK        0x02
            TARGET_3_MASK        0x04
            TARGET_4_MASK        0x08
            TARGET_5_MASK        0x10
            TARGET_6_MASK        0x20
        Bit – 0 – false
```

```
      bit - set - true
```

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 8)

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.
```

4.3 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. 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 2.0V.
- Communication between programming adapter and target device is initialized.
- Selected encapsulated instruction is executed (Autoprogram, 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 8) 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
- read retained data (if required)
- read IEEE address contents(if required)
- erase flash memory,
- memory blank check,
- flash programming,
- restoring or writing the new IEEE address (if required)
- restoring retained data (if required)
- flash memory verification (check sum verification of whole verification byte by byte),
- write lock protection bits (if required).
- switch-off Vcc from target device.

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
```

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 & 0xFFFE;
    ...................
}

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_Verify_Lock_Bits

F_Verify_Lock_Bits   -Verify the Lock debug Bit. If debug access is disabled, then only debug bit is verified. Other bits are not accessible when the debug bit enable is clear. Result is saved in the Gang Buffer. Results from the Gang Buffer can be taken using function **F_Get_Lock_Bits.**

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed simultaneously.

Syntax:
```
            MSPPRG_API        INT_X  F_Verify_Lock_Bits( void );
```
Return value:
```
    Target devices result mask – 0x00 to 0x3F
      When bit is '1' – TRUE
                 '0' – FALSE
```

F_Memory_Erase

F_Memory_Erase   **-** Erase Target's Flash Memory
VALID FPA index   - (1 to 8) 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.

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
```

F_Memory_Blank_Check

F_Memory_Blank_Check   **-** Check if the Target's Flash Memory is blank.

VALID FPA index   - (1 to 8) 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
```

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 8) 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
```

F_Memory_Verify

F_Memory_Verify   **-** Verify contents of the selected Target Devices Flash Memory and
                                    Code file**.**
VALID FPA index   - (1 to 8) 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 then 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
```

F_Gang_Flash_Read

F_Gang_Flash_Read   - Read Flash memory from selected or all Target Devices**.**
VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed simultaneously.

Size of the read memory size is defined in the configuration setup
All data will be saved in the internal Read Gang Buffer. Contents from the Raed Gang Buffer can be
taken using function
```
     BYTE  F_Get_Byte_from_Gang_Buffer( BYTE target_no, INT_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
```

Example:
```
BYTE data[MAX_FLASH_SIZE][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=0x0; addr<=0x1FFFF; addr++)
          data[addr][n] = F_Get_Byte_from_Gang_Buffer( n, addr );
     }
  ....................
```

F_Write_IEEE_Address

F_Write_IEEE_Address   **-** Write content taken from the IEEE Address Gang buffer to target devices. Write IEEE address option should be enabled in the configuration setup.

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed simultaneously.

Syntax:
```
INT_X  MSPPRG_API F_Write_IEEE_Address( void );
```

Return value:
```
Target devices result mask – 0x00 to 0x3F
 When bit is '1' – TRUE
                '0' – FALSE
```

Example:

```
F_Put_IEEEAddr64_to_Gang_Buffer( 1, 0x0123456789ABCDE0 );
F_Put_IEEEAddr64_to_Gang_Buffer( 2, 0x0123456789ABCDE1 );
.............................................................
F_Put_IEEEAddr64_to_Gang_Buffer( 6, 0x0123456789ABCDE5 );
F_Write_IEEE_Address();
```

F_Read_IEEE_Address

F_Read_IEEE_Address   **-** Read IEEE Addresses from target devices and save it in the IEEE gang buffer

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed simultaneously.

Syntax:
```
INT_X  MSPPRG_API F_Read_IEEE_Address( void );
```

Return value:
```
Target devices result mask – 0x00 to 0x3F
```

```
      When bit is '1' – TRUE
                   '0' – FALSE
```

Example:
```
#define "GangProCC-Dll.h";
ULONG64 IEEE[6];

    F_Read_IEEE_Address();
    IEEE[0] = F_Get_IEEEAddr64_from_Gang_Buffer( 1 );
    IEEE[1] = F_Get_IEEEAddr64_from_Gang_Buffer( 2 );
  ...........................................................
    IEEE[5] = F_Get_IEEEAddr64_from_Gang_Buffer( 6 );
```

F_Write_Lock_Bits

F_Write_Lock_Bits   - Write lock bits to target devices. Contents of the lock bits should be set first using configuration setup instructions.

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Syntax:
```
          MSPPRG_API  INT_X  F_Write_Lock_Bits( void );
```

Return value:
```
     Target devices result mask – 0x00 to 0x3F
       When bit is '1' – TRUE
                    '0' – FALSE
```

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 CCxx 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 the CCxx flash device two or four bytes are programmed simultaneously. This means - programmed bytes should be prepared first and flashed as a block with two or four bytes length. Otherwise four independent bytes programmed separately will be programmed four times - one time with required data and 3 times with 0xFF data.

   Note: CCxx devices with max flash size up to 32 kB have two bytes size programming word in the flash, while the CCxx with bigger flash memory size (up to 128 kB), have four bytes size writing word.

F_Open_Target_Device

F_Open_Target_Device   - Initialization communication with the target device.
VALID FPA index   - (1 to 8) 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 then 2.0V.
  - communication between programming adapter and target device is initialized.

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
```

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(0x4000);
  ...................
    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 8) 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
```

Example:
See example above (**F_Open_Target_Device**).

F_Segment_Erase

F_Segment_Erase - Erase any segment of the CCxx Flash memory.
VALID FPA index - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Parameters:

segment address - Any address form the desired segment space addresses

To erase a memory segment specify an address within that memory segment. For example to erase segment 0x2000-0x27FF any address from the range 0x2000 to 0x27FF 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( INT_X address );
```

Return value:
```
     Target devices result mask – 0x00 to 0x3F
       When bit is '1' – TRUE
                   '0' – FALSE
```

Example:
```
  ...................
    F_Segment_Erase(0x4000);      // erase segment 0x4000 to 0x47FF
    F_Segment_Erase(0x4100);      // erase the same segment
  ....................
```

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 8) or 0 (ALL FPAs) executed sequentially.

Parameters:
      start address   -   Even number from 0x0 to 0x1FFFE,
      stop address   -   Odd number from 0x1 to 0x1FFFF,,

Syntax:
```
MSPPRG_API  INT_X  F_Sectors_Blank_Check( INT_X start_addr,
                                          INT_X stop_addr );
```

Return value:
```
     Target devices result mask – 0x00 to 0x3F
       When bit is '1' – TRUE
                   '0' – FALSE
```

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_Byte_to_XRAM

F_Write_Byte_to_XRAM   - Write one byte to XRAM.
VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Write one byte to any XRAM location of the target devices.
Parameters:

        address   - address where XRAM is located 0xDF00 to 0xFFFF,

        data   - one byte to be written to target device

Syntax:
```
  MSPPRG_API  INT_X  F_Write_Byte_to_XRAM( INT_X addr, BYTE data );
```

Return value:
```
      Target devices result mask – 0x00 to 0x3F
        When bit is '1' – TRUE
                    '0' – FALSE
```

Example:
```
      F_Write_Byte_to_XRAM( 0xF010, 0x21 );
```

F_Write_Byte_to_direct_RAM

F_Write_Byte_to_direct_RAM   - Write one byte to direct RAM.
VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Write one byte to any direct RAM location of the target devices.
Parameters:

        address   - address where RAM is located 0x00 to 0xFF,

        data   - one byte to be written to target device

Syntax:

```
    MSPPRG_API  INT_X  F_Write_Byte_to_direct_RAM( INT_X addr, BYTE data );
```

Return value:
```
        Target devices result mask – 0x00 to 0x3F
          When bit is '1' – TRUE
                       '0' – FALSE
```

Example:
```
        F_Write_Byte_to_direct_RAM( 0x60, 0x33 );
```

F_Copy_Buffer_to_Flash

F_Copy_Buffer_to_Flash   - Write data from the Gang Buffer no '1' ta all selected target devices.
VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Parameters:

        start address   - Flash address from 0x0 to 0x1FFFF,

        size   - Size from 1 to MAX_FLASH_SIZE (0x20000) - block of data in bytes
                                   to be written.

Syntax:
```
  MSPPRG_API  INT_X  F_Copy_Buffer_to_Flash( INT_X start_addr, INT_X size );
```

Return value:
```
        Target devices result mask – 0x00 to 0x3F
          When bit is '1' – TRUE
                       '0' – FALSE
```

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, 0x1100 );
   ....................
```

F_Copy_Gang_Buffer_to_Flash

F_Copy_Gang_Buffer_to_Flash   - Write data block from 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 8) or 0 (ALL FPAs) executed sequentially.

Parameters:
>   start address   - Flash address from 0x0 to 0x1FFFF,
>   size   - Size from 1 to `MAX_FLASH_SIZE` (0x20000) - block of data in bytes
>   to be written.

Syntax:
```
MSPPRG_API INT_X  F_Copy_Gang_Buffer_to_Flash( INT_X start_addr, INT_X size);
```


Return value:
```
     Target devices result mask – 0x00 to 0x3F
       When bit is '1' – TRUE
                    '0' – FALSE
```


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_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 8) or 0 (ALL FPAs) executed sequentially.

Syntax:
```
MSPPRG_API      INT_X  F_Copy_Flash_to_Gang_Buffer( INT_X start_address,
                                    INT_X  size );
```

Parameters:
```
    start address
    size
```

Return value:
```
      Target devices result mask – 0x00 to 0x3F
      When bit is '1' – TRUE
                  '0' – FALSE
```

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_Copy_Buffer_to_XRAM

F_Copy_Buffer_to_XRAM - Write "size" number of bytes from the temporary XRAM/Flash gang buffer no '1' to XRAM. Starting address is specified in the "start address".

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Syntax:
```
 MSPPRG_API   INT_X  F_Copy_Buffer_to_XRAM( INT_X start_address, INT_X  size );
```

Parameters:
```
     start address    – physical XRAM address 0xDF00 to 0xFFFF
     size             – size in bytes
```

Return value:
```
     Target devices result mask – 0x00 to 0x3F
       When bit is '1' – TRUE
                  '0' – FALSE
```

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

Example:
```
  ...................
   ...................
     for( addr = 0xF220; addr<0xF300; addr++ )
        st = F_Put_Byte_To_Gang_Buffer( 1, addr, data[addr] );
     st = F_Copy_Buffer_to_XRAM( 0xF220, 0xE0 );
   ...................
```

F_Copy_Gang_Buffer_to_XRAM

F_Copy_Gang_Buffer_to_XRAM - Write "size" number of bytes from the temporary XRAM/Flash Gang Buffers to XRAM.

Data from Gang Buffer no '1' to XRAM of the Target's no '1'

Data from Gang Buffer no '2' to XRAM of the Target's no '2'

....

Data from Gang Buffer no '6' to XRAM of the Target's no '6'

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Syntax:

```
MSPPRG_API    INT_X   F_Copy_Gang_Buffer_to_XRAM( INT_X start_address,
                                    INT_X  size );
```

Parameters:

```
    start address     – physical XRAM address 0xDF00 to 0xFFFF
    size              – size in bytes
```

Return value:

```
    Target devices result mask – 0x00 to 0x3F
    When bit is '1' – TRUE
                '0' – FALSE
```

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

Example:

```
....................
 ....................

    for( n=0; n<MAX_TARGET_DEVICE_NUMBER; n++)
      for( addr = 0xF220; addr<0xF240; addr++ )
        st = F_Put_Byte_To_Gang_Buffer( n, addr, data[addr][n] );
    st = F_Copy_Gang_Buffer_to_XRAM( 0xF220, 0x20 );
....................
```

F_Copy_XRAM_to_Gang_Buffer

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

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Syntax:

```
MSPPRG_API    INT_X  F_Copy_XRAM_to_Gang_Buffer( INT_X start_address,
                                INT_X  size );
```

Parameters:

```
    start address     – physical XRAM address 0xDF00 to 0xFFFF
    size              – size in bytes
```

Return value:

```
    Target devices result mask – 0x00 to 0x3F
```

```
        When bit is '1' – TRUE
                    '0' – FALSE
```

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

Example:
```
...................
 ...................
    st = F_Copy_XRAM_to_Gang_Buffer( 0xF220, 0xE0 );
    if( st == TRUE )
     {
       for( n=0; n<MAX_TARGET_DEVICE_NUMBER; n++)
         for( addr = 0xF220; addr<0xF300; addr++ )
           data[addr][n] = F_Get_Byte_from_Gang_Buffer( n, addr );
     }
    else
     {
        ...................
     }
 ...................
```

F_Copy_Buffer_to_direct_RAM

F_Copy_Buffer_to_direct_RAM   - Write "size" number of bytes from the temporary XRAM/Flash gang buffer no '1' to direct RAM. Starting address is specified in the "start address"**.**

VALID FPA index   *- (1 to 8) or 0 (ALL FPAs) executed sequentially.*

Syntax:
  MSPPRG_API   **INT_X**   **F_Copy_Buffer_to_direct_RAM(INT_X start_address, INT_X size);**

Parameters:
```
     start address    – direct RAM address 0x00 to 0xFF
     size             – size in bytes
```

Return value:
```
     Target devices result mask – 0x00 to 0x3F
       When bit is '1' – TRUE
                   '0' – FALSE
```

Example:

```
...................
...................
    for( addr = 0x60; addr<0x7f; addr++ )
        st = F_Put_Byte_To_Gang_Buffer( 1, addr, data[addr] );
    st = F_Copy_Buffer_to_direct_RAM( 0x60, 0x20 );
...................
```

F_Copy_Gang_Buffer_to_direct_RAM

F_Copy_Gang_Buffer_to_direct_RAM - Write "size" number of bytes from the temporary Gang Buffers to direct_RAM.

Data from Gang Buffer no '1' to direct RAM of the Target's no '1'

Data from Gang Buffer no '2' to direct RAM of the Target's no '2'

....

Data from Gang Buffer no '6' to direct RAM of the Target's no '6'

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Syntax:
```
MSPPRG_API   INT_X   F_Copy_Gang_Buffer_to_direct_RAM( INT_X start_address,
                                    INT_X  size );
```
Parameters:
```
    start address    – direct RAM address 0x00 to 0xFF
    size             – size in bytes
```
Return value:
```
    Target devices result mask – 0x00 to 0x3F
    When bit is '1' – TRUE
                '0' – FALSE
```

F_Copy_direct_RAM_to_Gang_Buffer

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

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

Syntax:
```
MSPPRG_API      INT_X  F_Copy_direct_RAM_to_Gang_Buffer( INT_X start_address,
                                    INT_X  size );
```
Parameters:

```
        start address      - direct RAM address 0x00 to 0xFF
        size               - size in bytes
```
Return value:
```
        Target devices result mask – 0x00 to 0x3F
        When bit is '1' – TRUE
                    '0' – FALSE
```

F_Put_Byte_to_Gang_Buffer

F_Put_Byte_to_Gang_Buffer - Write byte to temporary Gang buffer.

VALID FPA index   - (1 to 8) or 0 (ALL FPAs) executed sequentially.

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


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

Return value:
```
        TRUE if specified address is legal (0x0000 to 0x1FFFF) otherwise FALSE.
```

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

NOTE: DLL contains two Gang Buffers - one is dedicated to the data READ from target devices, second one is dedicated to the data to be WRITE to the target devices. Contents of the Gang Buffers can not be verified by writing and read the same data to the Gang Buffers e.g.

```
            F_Put_Byte_to_Gang_Buffer (1, 0x1000, 5);
            data = F_Get_Byte_from_Gang_Buffer(1, 0x1000 );
```

Read data can be other that '5'.

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 temporary RAM/Flash Gang buffer**.**

VALID FPA index   - (1 to 8)

Syntax:
```
MSPPRG_API      BYTE  F_Get_Byte_from_Gang_Buffer( BYTE target_no,
                                INT_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: *DLL contains two Gang Buffers - one is dedicated to the data READ from target devices, second one is dedicated to the data to be WRITE to the target devices. Contents of the Gang Buffers can not be verified by writing and read the same data to the Gang Buffers e.g.*

F_Set_PC_and_RUN

F_Set_PC_and_RUN - Instructions allows to run program in microcontroller from specified PC in the XRAM or Flash location. Program should be downloaded first using the Write to Flash or XRAM procedures.

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 8) or 0 (ALL FPAs) executed sequentially.

Syntax:
```
MSPPRG_API INT_X  F_Set_PC_and_RUN( INT_X xram_en, INT_X PC_address );
```
Parameters:
```
xram_en     – 0 -> run program located in Flash
            – 1 -> run program located in XRAM
address     – set Program Counter to address and run
```

Return value:
```
Target devices result mask – 0x00 to 0x3F
When bit is '1' – TRUE
              '0' – FALSE
```

F_Copy_MCU_Data_to_Buffer

F_Copy_MCU_Data_to_Buffer - Get the MCU status or device ID/silicon version and save result in the Gang Buffer. Use the function **F_Get_MCU_Data_from_Buffer**(..) To get result from desired target.

VALID FPA index - (1 to 8) or 0 (ALL FPAs) executed sequentially.

```
INT_X  MSPPRG_API     F_Copy_MCU_Data_to_Buffer( INT_X Type );
```

Parameters:
```
Type: GET_MCU_ID        ( 1 )
        Result
            Higher byte (bits 15:8) MCU ID
```

```
                        0x01  – CC1110F8
                        0x85  – CC2430F64
                        0x89  – CC2431F32
                        0x81  – CC2510F8
                        0x91  – CC2511F8
                Lower byte  (bits 15:8) Silicon version ID


        GET_MCU_STATUS     ( 2 )
        Result     one byte
                0x80  – Chip Erase Done
                0x40  – PCON Idle
                0x20  – CPU halted
                0x10  – Power Mode 0
                0x08  – Halt Status
                0x04  – Debug Locked
                0x02  – Oscillator stable
                0x01  – Stack overflow
```

Return value:

```
        Target devices result mask – 0x00 to 0x3F
        When bit is '1' – TRUE
                     '0' – FALSE
```

F_Get_MCU_Data_from_Buffer

F_Get_MCU_Data_from_Buffer   - Get the latest result created by the function F_Copy_MCU_Data_to_Buffer();

VALID FPA index   - (1 to 8)

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

Parameters value:

```
        target_no – 1 to 6
```

Return value:

```
        data from the MCU buffer.
```

Appendix A

GangPro-CC 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 or script file to execute the API-DLL functions. All required files are located in the directory

   C:\Program Files\Elprotronic\CCxx\USB GangPro-CC\CMD-line

and contains

   GP-CC-commandline.exe   -> command line shell interpreter

   GangProCC-FPAsel.dll   -> standard API-DLL files

   GangProCC-FPA1.dll   -> ----,,,,,--------

All API-DLL files should be located in the same directory where the **GP-CC-commandline.exe** is located. To start the command line interpreter, the **GP-CC-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 **GP-CC-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 GangPro-CC - with defined CCxx type, code file etc.

Press ENTER - result ->1   (OK)

Type:

    F_AutoProgram(0)

Press ENTER - result ->3   (3-> 0000 0011 -> programmed two targets -> OK)
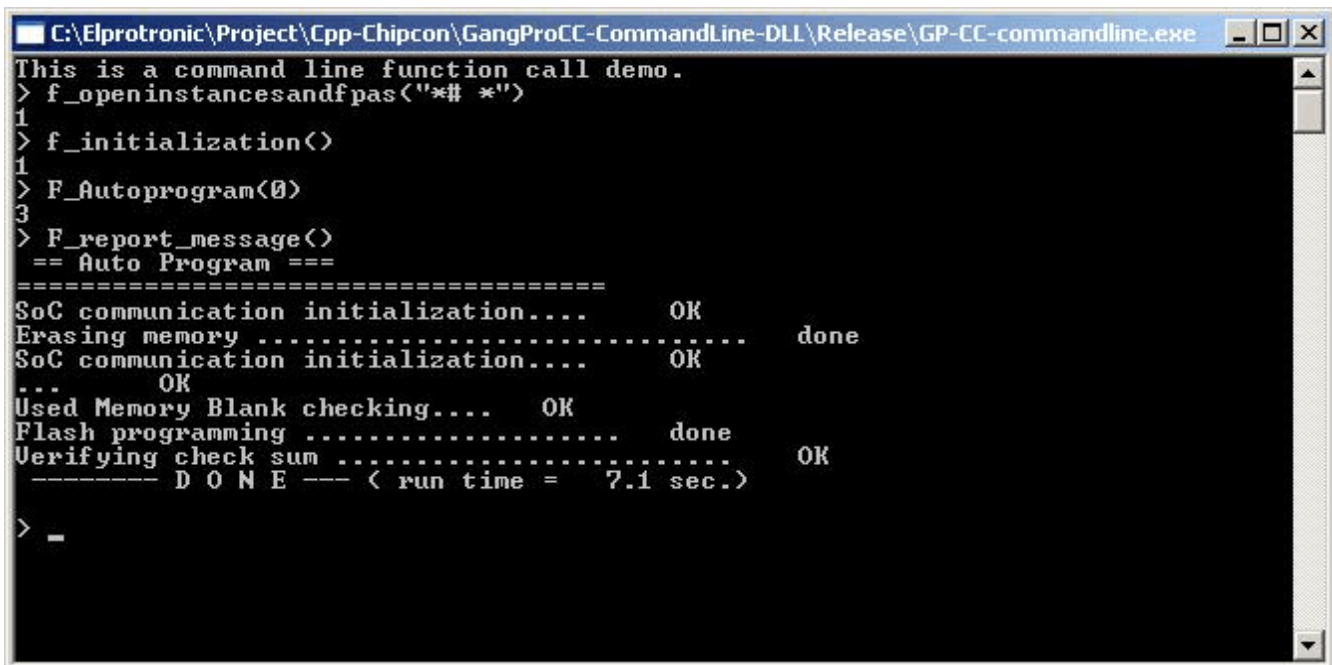
Type:

    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 **GP-CC-commandline.exe** program.

Figure A-1

Example-2:

Run the **GP-CC-commandline.exe** and type the following commands:

```
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 )
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_IEEEAddr64_to_Gang_Buffer( target_no, "Hex data string")
F_Get_IEEEAddr64_from_Gang_Buffer( target_no )
F_Get_Lock_Bits( target_no )
```

F_AutoProgram(0)

F_Verify_Lock_Bits()

F_Memory_Erase(mode)

F_Memory_Blank_Check()

F_Memory_Write(mode)

F_Memory_Verify(mode)

F_Gang_Flash_Read()

F_Write_IEEE_Address()

F_Read_IEEE_Address()

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_Write_Lock_Bits()

F_Write_Byte_to_XRAM(addr, data)

F_Write_Byte_to_direct_RAM(addr, data)

F_Copy_Buffer_to_XRAM(start_addr, size)

F_Copy_Gang_Buffer_to_XRAM(start_addr, size)

F_Copy_XRAM_to_Gang_Buffer(start_addr, size)

F_Copy_Buffer_to_direct_RAM(start_addr, size)

F_Copy_Gang_Buffer_to_direct_RAM(start_addr, size)

F_Copy_direct_RAM_to_Gang_Buffer(start_addr, size)

F_Set_PC_and_RUN(xram_en, PC_addr)

F_Copy_MCU_Data_to_Buffer(type)

F_Get_MCU_Data_from_Buffer(target_no)

F_Get_Targets_Vcc()

F_Get_Targets_Result()

F_Get_Active_Targets_Mask()

F_Disable_FPA_index(fpa)

F_Enable_FPA_index(fpa)

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.*