FlashPro2000 Flash Programmer

Multi-FPA API-DLL User's Guide

Software version 1-1

PM034A02 Rev.1
May-2009

Elprotronic Inc.

Elprotronic Inc.

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

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

Copyright © Elprotronic Inc. All rights reserved.

Disclaimer:

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

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

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

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

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

END USER LICENSE AGREEMENT

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

1. License.

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

YOU MAY:

A. use this Product on many computers;

B. make one copy of the software for archival purposes, or copy the software onto the hard disk of Your computer and retain the original for archival purposes;

C. use the software on a network

YOU MAY NOT:

A. sublicense, reverse engineer, decompile, disassemble, modify, translate, make any attempt to discover the Source Code of the Product; or create derivative works from the Product;

B. redistribute, in whole or in part, any part of the software component of this Product;

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

2. Copyright

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

3. Limitation of liability.

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

4. DISCLAIMER OF WARRANTIES.

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

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

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

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

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

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

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

Table of Contents

1. Introduction 8

2. Getting Started 13
- 2.1 MyFP2000Prg Projects 13

3. Example with API DLL 16
- 3.1 Example with single FPA 16
- 3.2 Example with Multi-FPA API DLL 17

4. List of the DLL instructions 20
- 4.1 Multi-FPA instructions 23
  - F_Trace_ON 23
  - F_Trace_OFF 23
  - F_OpenInstances 24
  - F_CloseInstances 24
  - F_OpenInstancesAndFPAs 25
  - F_Set_FPA_index 29
  - F_Get_FPA_index 30
  - F_Check_FPA_index 30
  - F_Disable_FPA_index 30
  - F_Enable_FPA_index 31
  - F_LastStatus 31
  - F_Multi_DLLTypeVer 32
  - F_Get_FPA_SN 32
- 4.2 Generic instructions 33
  - F_Check_FPA_access 33
  - F_Initialization 34
  - F_Close_All 35
  - F_SetConfig 36
  - F_GetConfig 41
  - F_Get_Device_Info 41
  - F_DispSetup 45
  - F_ReportMessage, F_Report_Message 45

F_GetReportMessageChar 46
F_DLLTypeVer 47
F_ConfigFileLoad 48
F_Reset_Target 49
F_Get_Targets_Vcc 50
4.3 Data Buffers access instructions 51
F_ReadCodeFile 51
F_Get_CodeCS 52
F_ReadPasswFile 53
F_Clr_Code_Buffer 54
F_Put_Word_to_Code_Buffer 54
F_Get_Word_from_Code_Buffer 55
F_Put_Word_to_CSM_Buffer 56
F_Get_Word_from_CSM_Buffer 56
F_Put_Word_to_Buffer 57
F_Get_Word_from_Buffer 57
4.4 Encapsulated instructions 59
F_AutoProgram 59
F_Verify_CSM_Password 60
F_Memory_Erase 61
F_Memory_Blank_Check 61
F_Memory_Write 62
F_Memory_Verify 62
F_Memory_Read 62
F_Write_CSM_Password 63
4.5 Sequential instructions 64
F_Open_Target_Device 64
F_Close_Target_Device 65
F_Segment_Erase 65
F_Sectors_Blank_Check 66
F_Write_Word_to_RAM 67
F_Read_Word 67
F_Copy_Buffer_to_Flash 68
F_Copy_Flash_to_Buffer 69

1. Introduction

The **FlashPro2000** adapter can be remotely controlled from other software applications (Visual C++, Visual Basic etc.) via a DLL library. The Multi-FPA - allows to remotely control simultaneously up to sixteen Flash Programming Adapters (USB-FPAs) significantly reducing programming speed in production.

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

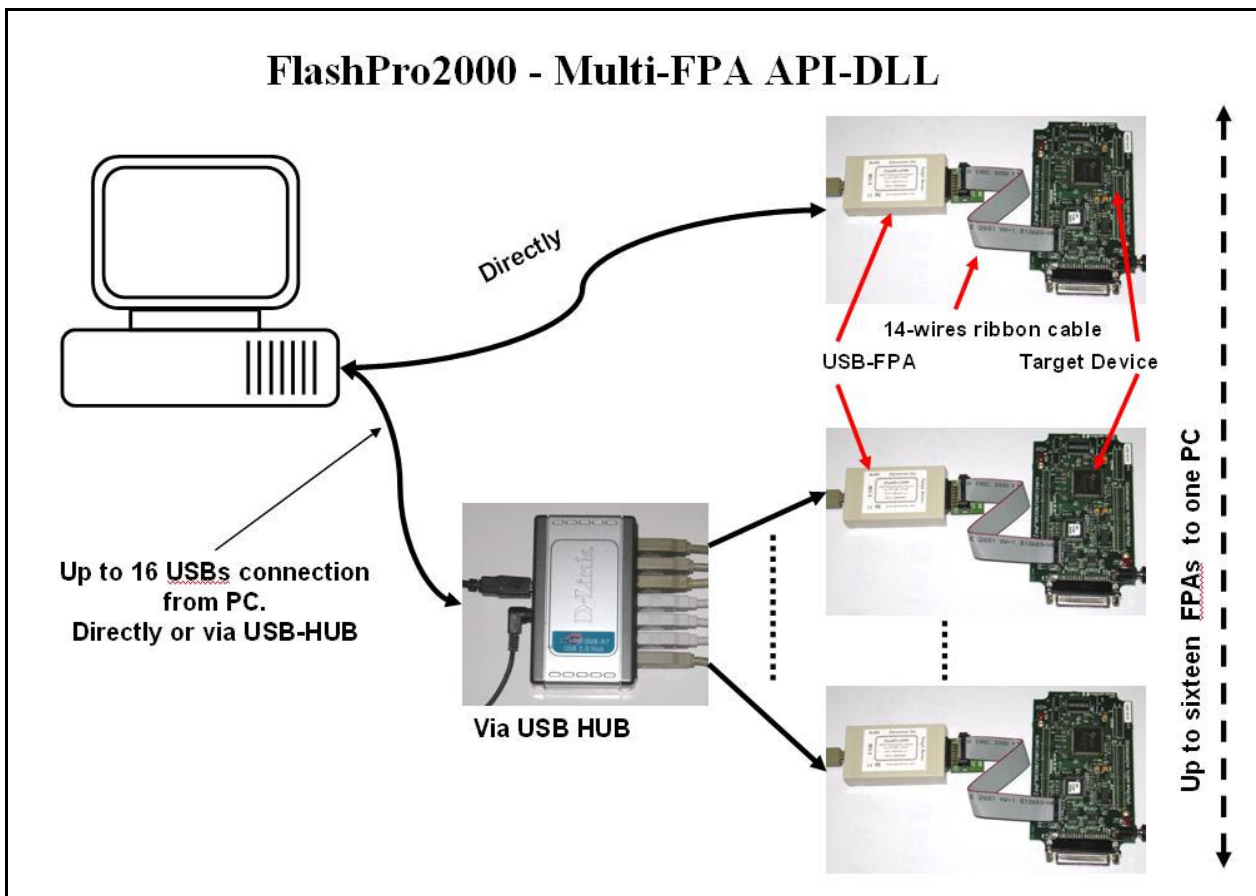

Figure 1.1

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

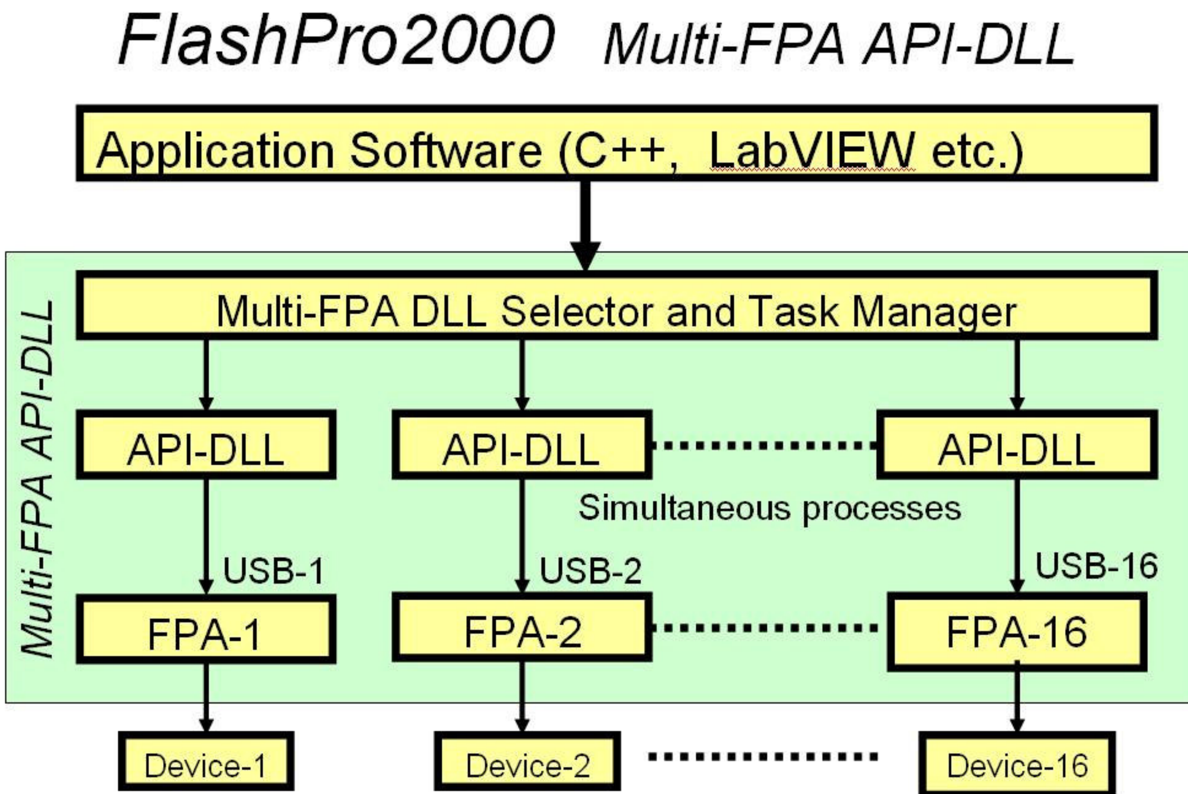

Figure 1.2

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

- the Multi-FPA API-DLL selector (FlashPro2000-FPAsel.dll)
- sixteen standard single FPAs API-DLLs (FlashPro2000-FPA1.dll, ...)

Figure 1.3 shows the logical connections between these dll files.

The main Multi-FPA dll (FPA-selector - FlashPro2000-FPAsel.dll) allows to transfer API-DLL functions coming from application software to desired single application dll (FlashPro2000-FPA1.dll to FlashPro2000-FPA16.dll).

The FlashPro2000-FPAsel.dll file is transparent for all API-DLL functions implemented in the single FPA API-DLLs functions. Desired destination FPA can be selected using the function

F_Set_FPA_index(fpa);

where the

fpa = 1 to 16 when only one desired FPA required to be selected

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

The selected FPA index modified by the F_Set_FPA_index(fpa) instruction can be modified at any time. By default, the FPA index is 1 and if only one FPA is used then fpa index does not need to be initialized or modified. When the fpa index 1 to 16 is used, then the result is coming back to application software from the single API-DLL via transparent Multi-FPA dll. When fpa index is 0 (ALL-FPAs) and results are the same from all FPAs, then the same result is passing back to application software. If results are not the same, then the Multi-FPA dll is returning back value -1 (minus 1) and all recently received results can be read individually using function

F_LastStatus(fpa)

Most of the implemented functions allows to use the determined fpa index 1 to 16 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 16). 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-16 but that process can not be seen from the application software.

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

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

The following commands are supported in the DLL library:

Initialization and termination communication with the programming adapter,
Programmer configuration setup,
Programming report message,
Code data and password data read from the file,
Reset target device,
Auto program target device (erase, blank check, program and verify),
CSM Password verification,

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

The FlashPro2000 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\C2000\USB FlashPro2000\API-DLL

FlashPro2000-FPAsel.dll	- Multi-FPA selection/distribution DLL
FlashPro2000-FPA1.dll	- API-DLL for FPA adapter
FlashPro2000-Dll.h	- generic header file for dll
FlashPro2000-FPAsel-BC.lib	- lib file for Borland VC++
FlashPro2000-FPAsel.lib	- lib file for MS VC++
C2000-Errors-list.cpp	- Errors list description
C2000-Errors-list.h	- Errors list definitions
config.ini	- default configuration file for the FPAs (optional)

The entry dll (FlashPro2000-FPAsel.dll) contains two groups of the same functions used in C++ application and Visual Basic applications All procedure names used in the Visual Basic are starting from **VB_xxxx**, (and have the **_stdcall** calling declaration) when procedure names used in the C++ are starting from **F_xxxx (**and have the **_Cdecl** calling declaration).

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

FlashPro2000-Dll.h	- header file for C++
FlashPro2000-FPAsel.lib	- lib file for C++
C2000-Errors-list.cpp	- (Optional) Errors list description
C2000-Errors-list.h	- Errors list definitions

and to the release/debug application directory

FlashPro2000-FPAsel.dll	- Multi-FPA selection/distribution DLL

FlashPro2000-FPA1.dll	- API-DLL for FPA adapter
config.ini	- (optional) default configuration file for the FPAs

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

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

FlashPro2000-FPAsel.dll	- Multi-FPA selection/distribution DLL
FlashPro2000-FPA1.dll	- API-DLL for FPA adapter
config.ini	- (optional) default configuration file for the FPAs

All these files 'as is' should be copied to destination location, where application software using DLL library of the FlashPro2000 Flash programmer is installed. The config.ini file has default setup information. The config.ini file can be modified and taken directly form the FlashPro2000 Flash Programmer (GUI) application software. To create required config.ini file the GUI FlashPro2000 Flash programmer software should be open and required setup (memory option, interface select 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.

2. Getting Started

2.1 MyFP2000Prg Projects

The ***MyFP2000Prg*** projects are examples of using the Multi-FPA API-DLL with Microsoft Visual Studio 7.0 (2002). They are intended to help users create their own application that uses the API-DLL by providing a simple starting point. When using Visual Studio C++ include the following files should be included to your program:

FlashPro2000-Dll.h
FlashPro2000-FPAsel.lib
FP2000FPA0Lib.cpp
FP2000FPA0Lib.h
C2000-Error-list.cpp
C2000-Error-list.h

The above files are located in the following directory:
...\Elprotronic\C2000\USB FlashPro2000\API-DLL-MyPrg\Cpp\scr

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

FlashPro2000-FPAsel.dll
FlashPro2000-FPA1.dll
config.ini (optional)

The easy demo project MyFP2000Prg uses API-DLLs and files listed above is located in directory

...\Elprotronic\C2000\USB FlashPro2000\API-DLL-MyPrg\Cpp\MyFP2000Prg

and are included for demonstration purposes only. The sample project can be opened by selecting the project file **MyFP2000Prg.vcproj** located in directory

...\Elprotronic\C2000\USB FlashPro2000\API-DLL-MyPrg\Cpp\MyFP2000Prg

The following dialog box will be displayed when project executed (see figure 2.1).

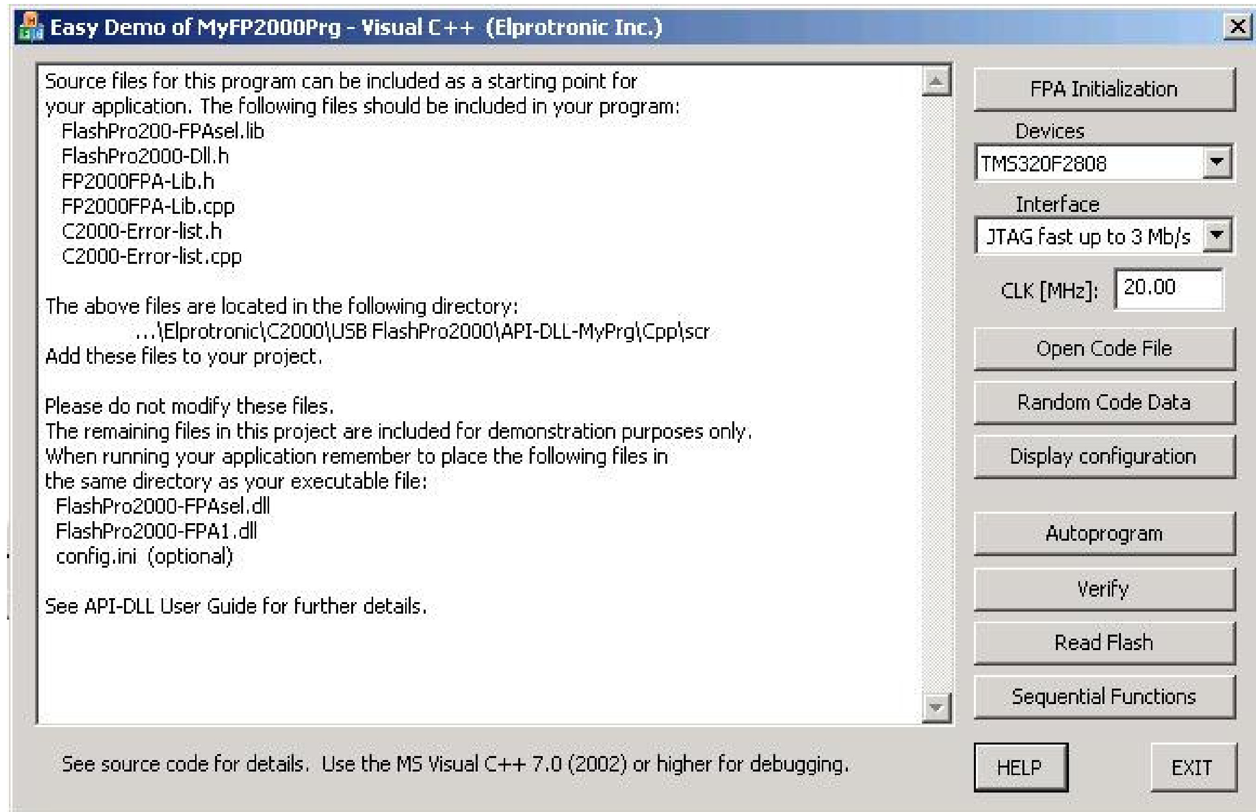

Figure 2.1

Dialog box contains few buttons, that call procedures listed in the mentioned above files. See contents in the M**yFP2000PrgDlg.cpp** file located in the project directory, how these procedures are called from application software. There are several useful procedures located in the **FP2000FPA-Lib.cpp** file that significantly simplify the FPA initialization process. See comments for each procedures located in this file.

The first procedure named

Scan_all_FPAs()

searches all FPAs connected to your PC via USB ports. As the results, adapter serial numbers of the detected FPAs are located in the FPA_SN_list[k] where k = 0 up to 15. Up to sixteen FPA SN can be located in this data block. SN list are located starting from FPA_SN_list[0].

get_devices_list()

The ***get_devices_list()*** procedure takes a list of supported target devices containing MCU name, flash start and end addresses etc. from API-DLL .The MCU list is saved in the following structure

```
typedef struct
{
      char name[DEVICE_NAME_SIZE];
      int  index;
      long flash_start_addr;
      long flash_end_addr;
      long OTP_start_addr;
      long OTP_end_addr;
      long RAM_size;
      int  group;
      int  double_ID;

} DEVICELIST;
DEVICELIST DeviceList[300];
```

Up to 100 devices can be saved in DeviceList. When required, the size of this data block can be increased in the future. Currently, device list contains about 30 devices. The device names in the ***DeviceList*** are sorted in alphabetic order. Alphabetical order is convenient for users, however the API-DLL requires fixed MCU index when selecting the particular MCU. In the structure above the MCU index required by API-DLL is located in

DeviceList[k].index

and procedure setting the required MCU becomes as follows

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

All other useful procedures that can be useful are listed below

int set_default_config(void);
int write_data_to_buffer(int dest, long addr, long size, UINT16 *data);
int read_data_from_buffer(int source, long addr, long size, UINT16 *data);

See the ***MyFP2000PrgDlg.cpp*** file how to strat-up communication with FPA and how to use the API-DLLs instructions.

3. Example with API DLL

3.1 Example with single FPA

The code example described below uses one programming adapter. The Multi-FPA API-DLL selector should be select for FPA-1 only. The ***fpa_index*** should be set to 1 or should be unmodified. The default value of the fpa_index when one adapter is detected only is 1.

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

```
response = F_OpenInstancesAndFPAs( “*# *” );
           // DLL and FPA (one only) initialization
if( response == 0 )
    {
      //The FPA has not been found. Exit from the program.
    }
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 one FPA, and run an autoprogram with the same features like an autoprogram in the standard FlashPro2000 (GUI) software.

1. Download data to target device

```
F_OpenInstancesAndFPAs( “*# *” ); // DLL and FPA (one only) initialization
if( response == 0 )
    {
      //The FPA has not been found. Exit from the program.
    }
F_Set_FPA_index( 1 );            // select FPA 1 only
F_Initialization( );             // init FPA
  //- functions above initialized at the startup only ----
F_ReadConfigFile(  filename );          // read configuration data and save
                                        // to API-DLLs
F_ReadCodeFile(  format, filename );    // read code data and save to DLL

  do
   {
    status = F_AutoProgram( 1 );        //start autoprogram
    if ( status != TRUE )
    {
     //’status’ contains status error number
     // see the C2000-Error-list.h and C2000-Error-list.cpp list
     // and error description
```

```
        ..............................
      }
      else
      {
          ..............................................
      }
    } while(1); //make an infinite loop until last target device programmed
.................................................................
   //- functions below called at the end of session
F_CloseInstances();
```

Note: The ***F_OpenInstancesAndFPAs(..)*** and ***F_Initialization()*** functions should be called once and the startup and the ***F_CloseInstances()*** function should be called as the last one after all functions are finished in similar way like the ***FlashPro2000*** GUI software is opening once and closed at the end when job is finished. The startup initialization take few seconds (when the ***F_OpenInstancesAndFPAs(..)*** and ***F_Initialization()*** are executed) until dll installation is established and desired firmware downloaded to FPA adapter(s). Application software should call the initialization procedures at the startup only, and close access to API-DLL at the end, when all tests of a lot of units are finished. Closing instances and opening it again is a waist a time.

3.2 Example with Multi-FPA API DLL

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

Application DLL should be initialized first, before other DLLs instruction can be used.

```
response = F_OpenInstancesAndFPAs( FPAs-setup.ini );
    // DLL and FPA initialization
if( response == 0 )
    {
      //The FPA has not been found. Exit from the program.
    }
F_Set_FPA_index( ALL_ACTIVE_FPA );        // select all FPA's
F_Initialization( );                      // init all FPA's
```

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

Below is an example of the simplified (without error handling procedures) application program written in C++ that allows to initialize all dlls and FPA, and run an autoprogram with the same features like autoprogram in the standard FlashPro2000 application software.

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

```
  response = F_OpenInstancesAndFPAs( FPAs-setup.ini);
             // DLL and FPA initialization
  if( response == 0 )
      {
        //The FPA has not been found. Exit from the program.
      }
  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(  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 first target device can be the same (but it is not required) as code data downloaded to second target device 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(  filename1 );            // read code data and save to
                                           // API-DLL-1
  F_Set_FPA_index( 2 );                    // select  FPA 2
  F_ReadCodeFile(  filename2 );            // read code data and save to
                                           // API-DLL-2
.................................................................
  F_Set_FPA_index(7 );                     // select  FPA 7
  F_ReadCodeFile(  filename7 );            // read code data and save to
                                           // API-DLL-7
  F_Set_FPA_index( 8 );                    // select  FPA 8
  F_ReadCodeFile( 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();
```

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

4. List of the DLL instructions

Application DLLs files are the same for the application software written under Visual C++, Visual Basic, LabView etc. From that reason the API-DLL not transfers the pointers from the API-DLL to application, because Visual Basic (or other software) will not be able to use these functions. When a lot of data are transferred from API-DLL to application, then these data should be read item by item.

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

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

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

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

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

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

Note: Inputs / outputs has been defined as INP_X, and LONG_X. Both of them are defined as 4 bytes long (see FlashPro2000-Dll.h header file)
#define INP_X _int32
#define LONG_X _int32
Make sure that an application using the DLL file has the same length of desired data.

Figure 4.1 shows the structure of the Multi-FPA API-DLL. It shows that the Multi-FPA DLL is used to communicate with the user application as well as the target devices. Each of the target devices is accessed by a single DLL associated with it. When more then one FPA is needed, up to 16 DLLs can be created to communicate with up to 16 devices at a time. Each instance of an FPA-DLL contains its own copy of buffers, as shown in Figure 4.2

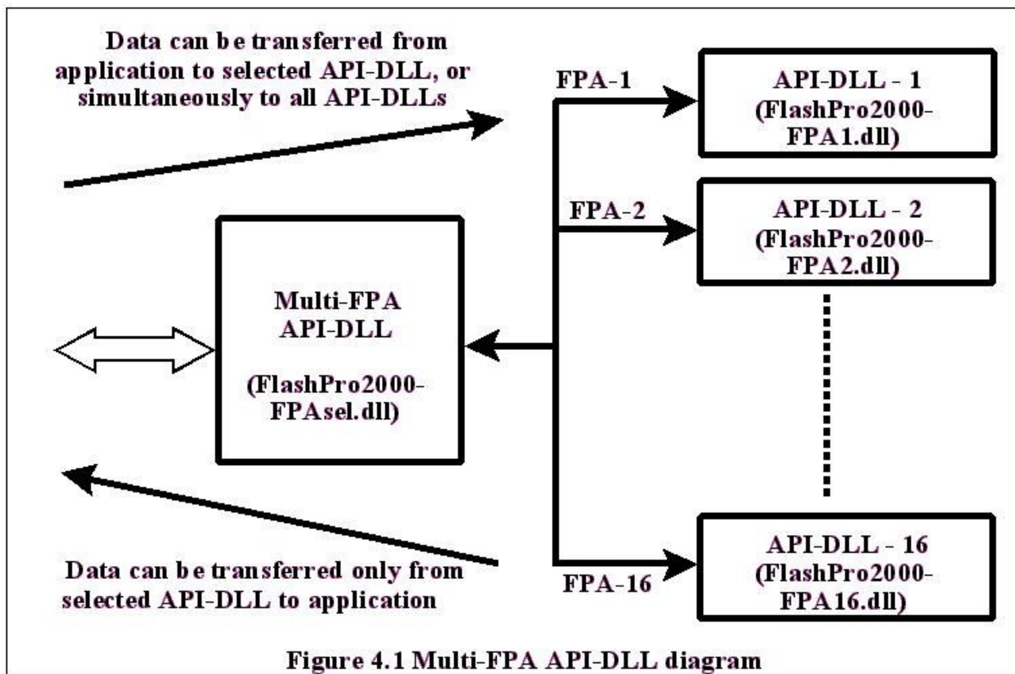

Figure 4.1 Multi-FPA API-DLL diagram

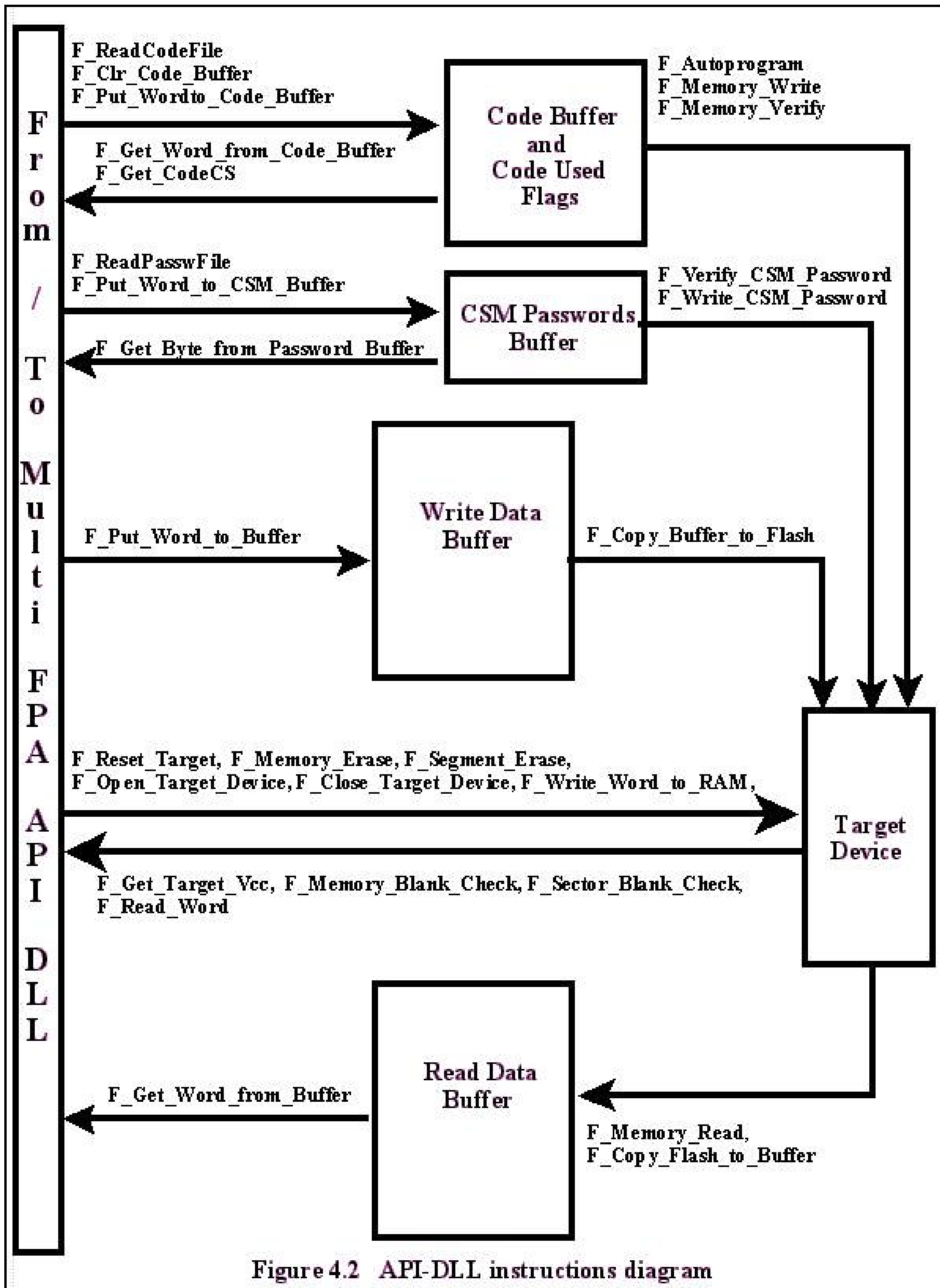

Figure 4.2 API-DLL instructions diagram

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 sixteen 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 and also allows to have individual access to any FPA is required.

F_Trace_ON

F_Trace_ON - This function activate the tracing.

The F_Trace_ON() opens the DLLtrace.txt file located in the current directory and records all API-DLL instructions called from the application software. This feature is useful for debugging. When debugging is not required then tracing should be disabled. Communication history recorded in the 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.*

Syntax:

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

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 is recommended.

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
```

Return value:

```
TRUE
```

F_OpenInstancesAndFPAs

F_OpenInstancesAndFPAs - API-DLL initialization in the PC and programming adapters scan and 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. Function is opening the number of the desired API-DLL and assigning the desired adapters to available USB ports. Regardless of the USB port open sequence and connection of the USB-FPA, the F_OpenInstancesAndFPAs instruction is reading the FPA's list, scanning all available adapters connected to any USB ports and assigning the indexes to all adapters according to contents of the FPA list (from string or configuartion file). All adapters 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 );
```

Parameters:

1. When the first two characters in the List string are *#, then reminding characters of the string contain a list of desired FPAs serial numbers or IDs assigned to FPA-1, -2, ...-n indexes, eg.

```
"*# 20060123, 20060234, 20060287"
```

2. When the first two characters in the List string are not *#, then the string contain file name or full path of the file with a 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:

"*# 20090123, 20090346, 20090222, 20090245"

or

"*# 20090123 20090346 20090222 20090245"

List of the acceptable numbers or IDs for USB-FPA adapters:

1. FPAs **serial number** - 8 digits eg. 20090222
   eg, "*# 20090123 20090346 20090222 20090245"
   Four USB-FPA will be used with SN as listed above

FPA-1	20090123
FPA-2	20090346
FPA-3	20090222
FPA-4	20090245

   If from any reason the listed adapter is not found, then the FPA-x becomes empty. All other adapters will have the same FPA-x indexes as specified in the list eg if the FPA SN is missing, then only the FPA-3 will be empty. The FPA-4 will have the same position as before.

FPA-1	20090123
FPA-2	20090346
FPA-3	Empty
FPA-4	20090245

2. ID **'*'** - to select any adapter - USB-FPA. No other adapters can be specified after this definition.
   eg, "*# 20090123 20090346 20090222 *"
   Last one will be any adapter USB-FPA not listed before.

Initialization examples:

```
1.    F_OpenInstancesAndFPAs( "*# *" ); // only one any adapter
```

or

```
2.    F_OpenInstancesAndFPAs( snlist ); // hardcoded SN list
```

2.The FPA list in the configuration file:

String -> `"C:\Program Files\Elprotronic\FPAs-setup.ini"`

The FPA list can be specified in the file using the same rules as the definitions described above. Each defined adapter is listed after FPA-index s below eg:

```
;===================================================================
; USB-FPA configuration setup                                      *
; Elprotronic Inc.                                                 *
;-------------------------------------------------------------------
; up to sixteen 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-16     *
;   Serial number - get serial number ir ID from the desires       *
;   adapter's label                                                *
;  Minimum one FPA's must be specified                             *
;  FPA-x order - any                                               *
;                                                                  *
;FPA-1   20090116                                                  *
;FPA-3   20090199                                                  *
;FPA-5   20090198                                                  *
;===================================================================

FPA-1   20090123
FPA-2   20090234

; 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   20090116
FPA-2   20090199
FPA-3   *
```

or

```
FPA-1   *
```

when only one adapter (any adapter) is used.

Example:

1. Only one FPA is used:

```
F_OpenInstancesAndFPAs( “*# *” ); //DLL startup and FPA assignment
            //by default - FPA-1 is selected.
            //The F_Set_FPA_index(1) is not required.
F_Initialization();               //FPA 1 initialization
F_ReadConfigFile( filename );     //download configuration to DLLs.
F_ReadCodeFile( filename );       //download code file to DLLs.
```

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

2. More then one FPA is used.

```
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( filename );
            //download the same code file to all DLLs.
do
{
  status = AutoProgram(1);
            //start autoprogram to all FPAs simultaneously.
  if( status != TRUE )
      {
        if( status == FPA_UNMATCHED_RESULTS )
         {
           // service software when results from FPAs are not the same
         }
        else
         {

         }
        .......................
      {
  .......................
} while(1);
F_CloseInstances();
      // release DLLs from memory
```

F_Set_FPA_index

F_Set_FPA_index - Select desired FPA index (desired DLL instance)

VALID FPA index - (1 to 16) 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  = 16
             or 0 -> ALL_ACTIVE_FPA

note: instead of '0' value it can be used global defined
      ALL_ACTIVE_FPA that is defined as
#define    ALL_ACTIVE_FPA    0
in the header file
```

Return value:

```
      TRUE - if used fpa index is valid
      FPA_INVALID_NO - if used fpa index is not activated or out of range
note:  FPA_INVALID_NO  ->  -2 (minus 2)
```

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

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

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

F_Get_FPA_index

F_Get_FPA_index - Get current FPA index

Syntax:

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

Return value:

```
current FPA index
```

F_Check_FPA_index

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

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

Syntax:

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

Return value:

```
current FPA index ( 0, 1 to 16)
or -2 (minus two) FPA_INVALID_NO
```

F_Disable_FPA_index

F_Disable_FPA_index - Disable desired FPA index (desired DLL instance)
VALID FPA index - (1 to 16)

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  = 16
```

F_Enable_FPA_index

F_Enable_FPA_index - Enable desired FPA index (desired DLL instance)

VALID FPA index - (1 to 16)

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  = 16
```

F_LastStatus

F_LastStatus - Get current FPA index

VALID FPA index - (1 to 16)

Syntax:

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

Parameters:

```
fpa - FPA index of the desired status
      fpa index -> 1..16
```

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<=16; n++)   status[n] = F_LastStatus(n);

where n -> desired FPA index
and get the last status data from FPA-1, 2, .. up to .16

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 = 0x6000   - Multi-FPA API-DLL for FlashPro430
DLL ID = 0x7000   - Multi-FPA API-DLL for GangPro430
DLL ID = 0x8000   - Multi-FPA API-DLL for FlashPro-CC
DLL ID = 0x9000   - Multi-FPA API-DLL for GangPro-CC
DLL ID = 0xC000   - Multi-FPA API-DLL for FlashPro2000
DLL ID = 0xD000   - Multi-FPA API-DLL for GangPro2000
Version = (0x0FFF & VALUE)
```

F_Get_FPA_SN

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

Syntax:

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

Parameters:

```
fpa - FPA index of the desired status
      fpa index -> 1..16
```

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)(0xFFFFFFFE)
```

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 (USB-FPA) connected to specified USB drivers (USB driver index from 1 to 16)

VALID FPA index (DLL instance number) - (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.

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      LONG_X      F_Check_FPA_access ( INT_X USB_index );
```

Parameters:

```
Index: USB driver index from 1 to MAX_USB_DEV_NUMBER
         where MAX_USB_DEV_NUMBER = 16
```

Return value:

```
0 - FALSE
>0 - Detected USB-FPA 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] > 20000000 ) 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]= 20090123
//    SN[2]= 20090147
//    SN[3]= 0                - adapter not present
//    SN[4]= 20090135
// and desired assignment
//    FPA-1   20090123
//    FPA-2   20090135
//    FPA-3   20090147
// 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] = 20090123 to FPA-1
      F_Set_FPA_index( 2 );   // select access to the second instance
      F_Check_FPA_access( 4 ); //assign FPA SN[4] = 20090135 to FPA-2
      F_Set_FPA_index( 3 );   // select access to the third instance
      F_Check_FPA_access( 2 ); //assign FPA SN[2] = 20090147 to FPA-3

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

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

F_Initialization

F_Initialization - Programmer initialization.

VALID FPA index - (1 to 16) 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_OpenInstancesAndFPAs("*# *")** if none of the instances has not been activated before. Also the FPA index is selected to 1 by default. That allows to use old application software without calling the new type of Multi-FPA functions.

When the **F_Initialization** is called then:

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

Programming adapter must be connected to the USB 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.
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
```

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 16) 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
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
```

Example:

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

F_SetConfig

F_SetConfig - Setup one item of the programmer's configuration.

VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

The F_SetSetup can modify configuration of the programmer. Current programmer setup can be read using function setup F_GetSetup. When data is taken from the programmer, then part or all of the configuration data can be modified and returned back to programmer using F_SetConfig 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 index list in the F_SetConfig for details below.

Note: See theFlashPro2000-Dll.h header file for the list of the latest indexes, definitions etc.

Syntax:

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

Return value:

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

Example:

```
.......................
.......................
   F_SetConfig( CFG_MICROCONTROLLER, MCU_Index );
.......................
```

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

CFG_MICROCONTROLLER	1
CFG_INTERFACE	2
CFG_RESET_TIME_INDEX	3
CFG_RESET_PULSE_TIME	4
CFG_RESET_IDLE_TIME	5
CFG_APPL_START_EN	6
CFG_APPL_RUN_TIME	7
CFG_RELEASE_JTAG	8
CFG_BEEP_EN	9
CFG_VERIFYMODE	10
CFG_FLASH_ERASE_MODE	11
CFG_FLASH_NOT_ERASE_IFBLANK	12
CFG_DONOT_OVERWRITE_OTP	13
CFG_DEF_OTP_WRITE_EN	14
CFG_DEF_OTP_START_ADDR	15
CFG_DEF_OTP_STOP_ADDR	16
CFG_DEF_FLASH_ERASE_EN	17
CFG_DEF_ERASE_START_ADDR	18
CFG_DEF_ERASE_STOP_ADDR	19
CFG_RETAIN_DEF_DATA_EN	20
CFG_RETAIN_START_ADDR	21
CFG_RETAIN_STOP_ADDR	22
CFG_FLASH_READ_MODE	23
CFG_DEF_FLASH_READ_EN	24
CFG_DEF_READ_FLASH_START_ADDR	25
CFG_DEF_READ_FLASH_STOP_ADDR	26
CFG_DEF_OTP_READ_EN	27
CFG_DEF_READ_OTP_START_ADDR	28
CFG_DEF_READ_OTP_STOP_ADDR	29
CFG_CSMLOCK_PASSWORD_INDEX	30
CFG_CSMLOCK_PASSWORD_ENABLE	31
CFG_CLK_FERQ_IN_KHZ	32
CFG_JTAG_CHAIN_POS	33
CFG_JTAG_CHAIN_SIZE	34
CFG_JTAG_IRSIZE_DEVICE_1	35
CFG_JTAG_IRSIZE_DEVICE_2	36
CFG_JTAG_IRSIZE_DEVICE_3	37
CFG_JTAG_IRSIZE_DEVICE_4	38
CFG_JTAG_IRSIZE_DEVICE_5	39
CFG_JTAG_IRSIZE_DEVICE_6	40

```
[ CFG_MICROCONTROLLER  1 ]
```

```
uProcIndex                    - Microcontroller type selection
      TMS320Fxx         0 - TMS320F240
                        1 - TMS320F242
                        2 - ..........
      See the latest MCU list and indexes in the FlashPro2000 (GUI) software.
      Run software -> list available under pull down menu
          Setup-> MSP list
      Also the MCU index and MCU names can be taken from the instruction
      F_Get_Device_Info(). See description of this instruction in this manual
      for details.
```

[CFG_INTERFACE 2]

```
Interface                     - JTAG/SBW/BSL interface selection
      COMM_SCI_BOOT                 0     SCI-BOOT interface
      COMM_JTAG_FAST                1     JTAG fast
      COMM_JTAG_SLOW                2     JTAG slow
```

[CFG_RESET_TIME_INDEX 3]

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

[CFG_RESET_PULSE_TIME 4]

```
      CustomResetPulseTime    value 1 to 2000 step 1 in miliseconds
        valid only when the ResetTimeIndex = RESET_CUSTOM_INDEX
```

[CFG_RESET_IDLE_TIME 5]

```
      CustomResetIdleTime     value 1 to 2000 step 1 in miliseconds
        valid only when the ResetTimeIndex = RESET_CUSTOM_INDEX
```

[CFG_APPL_START_EN 6]

```
  ApplicationStartEn          -   reset   and   start the microcontroller's
                              application software when flash is successfully
                              programmed
  APPLICATION_KEEP_RESET      0     - Hardware Reset Line permanent LOW
  APPLICATION_TOGGLE_RESET    1     - Generate RESET Pulse (Pulse Low)
  APPLICATION_NOT_RESET       2     - Do not modify Reset Line state
  APPLICATION_JTAG_RESET      3     - JTAG software Reset
```

[CFG_APPL_RUN_TIME 7]

```
      Value in seconds - 0 to 120 - when 0 - infinite time
```

[CFG_RELEASE_JTAG 8]

DEFAULT_JTAG_3ST	0
DEFAULT_JTAG_HI	1
DEFAULT_JTAG_LO	2

[CFG_BEEP_EN 9]
```
Enable (1) / disable (0)
```

[CFG_VERIFYMODE 10]
```
VERIFY_NONE_INDEX 0     - no verification
VERIFY_STD_INDEX  1     - standard verification (read and verify)
VERIFY_FAST_INDEX 2     - fast verification (calculate CS and verify)
```

[CFG_FLASH_ERASE_MODE 11]

ERASE_NONE_MEM_INDEX	0	
ERASE_ALL_MEM_INDEX	1	(OTP and Flash)
ERASE_PRG_ONLY_MEM_INDEX	2	(Flash only)
ERASE_INFILE_MEM_INDEX	3	(taken from File)
ERASE_DEF_CM_INDEX	4	(defined OTP and Flash option)
WRITE_OTP_MEM_ONLY_INDEX	5	(OTP only)

[CFG_FLASH_NOT_ERASE_IFBLANK 12]
```
Enable (1) / disable (0)
```

[CFG_DONOT_OVERWRITE_OTP 13]
```
Enable (1) / disable (0)
```

[CFG_DEF_OTP_WRITE_EN 14]
```
Enable defined OTP erase option
```

[CFG_DEF_OTP_START_ADDR 15]
```
OTP Start Address when the defined flash erase option is selected
```

[CFG_DEF_OTP_STOP_ADDR 16]
```
OTP End Address when the defined flash erase option is selected
```

[CFG_DEF_FLASH_ERASE_EN 17]
```
Enable defined flash erase option
```

[CFG_DEF_ERASE_START_ADDR 18]
```
Flash Start Address when the defined flash erase option is selected
```

[CFG_DEF_ERASE_STOP_ADDR 19]
```
Flash End Address when the defined flash erase option is selected
```

[CFG_RETAIN_DEF_DATA_EN 20]
```
Enable (1) / disable (0)
```

[CFG_RETAIN_START_ADDR 21]

Retain Data Start Address

[CFG_RETAIN_STOP_ADDR 22]

Retain Data End Address

[CFG_FLASH_READ_MODE 23]

```
READ_ALL_MEM_INDEX      0     - Read all OTP and Flash memory
READ_PRGMEM_ONLY_INDEX  1     - Read Flash memory only
READ_INFOMEM_ONLY_INDEX 2     - Read OTP memory only
READ_DEF_MEM_INDEX      3     - Read OTP and Flash memory defined by
                                ReadStartAddr and ReadStopAddr
```

[CFG_DEF_FLASH_READ_EN 24]

```
Enable (1) / disable (0)
Defined Read Flash memory option
```

[CFG_DEF_READ_FLASH_START_ADDR 25]

```
Flash Start Address when the defined Read option is selected
```

[CFG_DEF_READ_FLASH_STOP_ADDR 26]

```
Flash End Address when the defined Read option is selected
```

[CFG_DEF_OTP_READ_EN 27]

```
Enable (1) / disable (0)
Defined Read OTP memory option
```

[CFG_DEF_READ_OTP_START_ADDR 28]

```
OTP Start Address when the defined Read option is selected
```

[CFG_DEF_READ_OTP_STOP_ADDR 29]

```
OTP End Address when the defined Read option is selected
```

[CFG_CSMLOCK_PASSWORD_INDEX 30]

CSM_DEFAULT_INDEX 0
CSM_CODE_FILE_INDEX 1
CSM_PASSWORD_FILE_INDEX 2
CSM_DEFINED_INDEX 3

[CFG_CSMLOCK_PASSWORD_ENABLE 31]

```
Enable (1) / disable (0)
```

[CFG_CLK_FERQ_IN_KHZ 32]

```
External CLK frequency in kHz
```

[CFG_JTAG_CHAIN_POS 33]

```
Devices position in the JTAG chain
  - from 1 to number of devices in the JTAG chain
```

[CFG_JTAG_CHAIN_SIZE 34]
```
      Number of devices in the JTAG chain- from 1 to  6
```

[CFG_JTAG_IRSIZE_DEVICE_1 35]
```
      IR register size of the first device in the JTAG chain
```

[CFG_JTAG_IRSIZE_DEVICE_2 36]
[CFG_JTAG_IRSIZE_DEVICE_3 37]
[CFG_JTAG_IRSIZE_DEVICE_4 38]
[CFG_JTAG_IRSIZE_DEVICE_5 39]
[CFG_JTAG_IRSIZE_DEVICE_6 40]
```
      IR register size of the 2-nd, 3-th .... device in the JTAG chain
```

Note: See theFlashPro2000-Dll.h header file for the list of the latest indexes, definitions etc.

Example:
```
      F_SetConfig( CFG_INTERFACE, COMM_JTAG_FAST );
      F_SetConfig( CFG_CSMLOCK_PASSWORD_ENABLE , 0 );
      F_SetConfig(  CFG_FLASH_ERASE_MODE , ERASE_ALL_MEM_INDEX );
```

F_GetConfig

F_GetConfig - Get one item of the programmer's configuration.

VALID FPA index - (1 to 16)

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

Index's list - see F_SetConfig

Return value:
```
      Requested setup parameter;
      1 - TRUE
      -2 (0xFFFFFFFE)   - FPA_INVALID_NO
```

Example:
```
   .......................
      Interface = F_GetConfig( CFG_INTERFACE );
  .......................
```

F_Get_Device_Info

F_Get_Device_Info - Get information related to selected microcontroller.

VALID FPA index - (1 to 16)

Syntax:

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

where index:

```
DEVICE_NAME                     0
DEVICE_NAME_SIZE                20
DEVICE_FLASH_START_ADDR         20
DEVICE_FLASH_END_ADDR           21
DEVICE_OTP_START_ADDR           22
DEVICE_OTP_END_ADDR             23
DEVICE_RAM_SIZE                 24
DEVICE_GROUP                    25
DEVICE_ALREADY_DEFINED          26
```

Return value:

```
-1 (0xFFFFFFF)    -  invalid data
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or
index - 0 to 19  ->      device name - char by char starting from index->0
                         => T eg.  TMS320F2808
index 0 -> 'T'
index 1 -> 'M'
index 2 -> 'S'
index 3 -> '3'
index 4 -> '2'
index 5 -> '0'
index 6 -> 'F'
index 7 -> '2'
index 8 -> '8'
index 9 -> '0'
index 10 -> '8'
index 11 -> 0x0000  -> end of string
index 11 to 19 -> after end of string - irrelevant data.
index 20 -> Flash Start Address     eg 0x3E8000  (for F2808)
index 21 -> Flash End  Address      eg 0x3F7FFF  (for F2808)
index 22 -> OTP Start Address       eg 0x3D7800  (for F2808)
index 23 -> OTP End  Address        eg 0x3D7FFF  (for F2808)
index 24 -> RAM size                eg 0x4800    (for F2808)
```

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

Below is an example of the procedure that can take names of all supported devices by the API-DLL. The max size can be tested from the API-DLL, until device name is empty when the microprocessor index is incremented from the zero up to max value. In the example below is assumed that te max

number of supported devices is 100, however this value can be dynamically modified if required. In the procedure below the names and uP index are saved in the `DEVICELIST` structure, where the name and index pair are kept in the same `DEVICELIST DeviceList[]` element. When the `DeviceList[]` is created, then all names are kept in the alphabetic order. Below is example how to read devices information. These procedure can be found in the FP2000FPA-lib.cpp - see the MyFP2000Prg.

```
#include "FlashPro2000-Dll.h"
#define   MAX_NO_OF_DEVICES   100

typedef struct
{
      char name[DEVICE_NAME_SIZE];
      int  index;
      long flash_start_addr;
      long flash_end_addr;
      long OTP_start_addr;
      long OTP_end_addr;
      long RAM_size;
      int  group;
      int    double_ID;
}DEVICELIST;

DEVICELIST DeviceList[MAX_NO_OF_DEVICES];

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

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

      *tmp.name = '\0';
      tmp.index = 0;
      tmp.flash_start_addr = 0;
      tmp.flash_end_addr = 0;
      tmp.OTP_start_addr = 0;
      tmp.OTP_end_addr = 0;
      tmp.RAM_size = 0;
```

```
    for(k=0; k<MAX_NO_OF_DEVICES; k++)
          DeviceList[k] = tmp;

    if( F_Check_FPA_index() == FPA_INVALID_NO )
          return( FPA_INVALID_NO );

    index_bak = F_GetConfig( CFG_MICROCONTROLLER );
    max_up_index = 0; p=0;
    for(k=0; k<MAX_NO_OF_DEVICES; k++)
    {
     F_SetConfig( CFG_MICROCONTROLLER, k );
     for( n = 0; n<DEVICE_NAME_SIZE; n++ )
      DeviceList[p].name[n]= char(0xFF& F_Get_Device_Info( DEVICE_NAME+n));
      if( DeviceList[p].name[0] == 0 ) break;
      if( strlen( DeviceList[p].name ) < 5 )  continue;
                  //processor not supported
      DeviceList[p].index = k;
      DeviceList[p].flash_start_addr =
F_Get_Device_Info(DEVICE_FLASH_START_ADDR);
      DeviceList[p].flash_end_addr =
          F_Get_Device_Info( DEVICE_FLASH_END_ADDR );
      DeviceList[p].OTP_start_addr =
          F_Get_Device_Info( DEVICE_OTP_START_ADDR );
      DeviceList[p].OTP_end_addr = F_Get_Device_Info( DEVICE_OTP_END_ADDR );
      DeviceList[p].RAM_size = F_Get_Device_Info( DEVICE_RAM_SIZE );
      DeviceList[p].group = F_Get_Device_Info( DEVICE_GROUP );
      DeviceList[p].double_ID = F_Get_Device_Info( DEVICE_ALREADY_DEFINED );
      max_up_index = p;
      p++;
     }
    F_SetConfig( CFG_MICROCONTROLLER, index_bak );  //restore uP index
          //sort names in the table from min to max.
    if( max_up_index > 0 )
     {
      for( k=0; k<max_up_index; k++ )
       {
        st = FALSE;
        for( n=1; n<=max_up_index; n++ )
         {
          if( strcmp( DeviceList[n-1].name, DeviceList[n].name ) < 0 )
                 continue;
          st = TRUE;
          tmp = DeviceList[n-1];
          DeviceList[n-1] = DeviceList[n];
          DeviceList[n] = tmp;
         }
        if( st == FALSE) break;
       }
      }
    Max_MCU_index = max_up_index;
```

```
 return(max_up_index);
}
```

F_DispSetup

F_DispSetup - Copy programmer's configuration to report message buffer in text form.
VALID FPA index - (1 to 16)

Syntax:

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

Return value:

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

Example:

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

F_ReportMessage, F_Report_Message

F_ReportMessage - Get the last report message from the programmer.
or F_Report_Message
VALID FPA index - (1 to 16)

When any of the DLL functions is activated, a message is created and displayed on the dynamically created programmer's dialog box. At the end of execution the dialog box is closed and function returns back to the application program. Reported message is closed as well. The last report message can be read by application program using F_ReportMessage function. When F_ReportMessage is called, then report message up to REPORT_MESSAGE_MAX_SIZE characters.
The REPORT_MESSAGE_MAX_SIZE is de fined in the FlashPro2000-dll.h and the value is 2000. Make sure to declare characters string length no less then 2000 characters.
When F_ReportMessage is called then at the end the internal report message buffer in the programmer software is cleared. When F_ReportMessage is not called after every communication

with the target device, then the report message will collect all reported information up to 2000 last characters.

Syntax:

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

Return value:

none

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

Example:

```
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.

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

void CMspPrgDemoDlg::Disp_report_message()
{
    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 16)

See comment for the **F_ReportMessage** function.

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

Syntax:

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

Return value:

```
      Requested character from the Report Message buffer.   1 - TRUE
```

Example:

```
char text[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.

```
............................................
Cstring Message = "";
............................................
void CMspPrgDemoDlg::Disp_report_message()
{
  char text[REPORT_MESSAGE_MAX_SIZE];     //must be min. size - 2000
  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_DLLTypeVer

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

VALID FPA index - (1 to 16)

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 FlashPro430 - Parallel Port
DLL ID = 0x2000   - Single-DLL for FlashPro430 - USB
DLL_ID = 0x3000   - Single-DLL for GangPro430  - USB
DLL_ID = 0x4000   - Single-DLL for FlashPro-CC - USB
DLL_ID = 0x5000   - Single-DLL for GangPro-CC  - USB
DLL_ID = 0xA000   - Single-DLL for FlashPro2000- USB
DLL_ID = 0xB000   - Single-DLL for GangPro2000 - USB
Version = (0x0FFF & VALUE)
```

Example:

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

F_ConfigFileLoad

F_ConfigFileLoad - Modify programmer's configuration setup according to data taken from the specified configuration file.

VALID FPA index - (1 to 16) 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 GUI FlashPro2000 Flash Programmer software. When setup from the file is downloaded, then old configuration setup is overwritten. New setup can be modified using **F_GetConfig** and **F_SetConfig** functions.

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

Syntax:

```
MSPPRG_API      INT_X  F_ConfigFileLoad( char * 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
            -2 (0xFFFFFFFE)    - FPA_INVALID_NO
      info is defined as follows:
        error ->  OPEN_FILE_OR_READ_ERR
```

Configuration file can be created using the FlashPro2000 GUI software. Run FlashPro2000 software, select desired configuration and save the file using option ***Save Setup as..*** `and_file_name`

Specified parameters in the configuration file can be listed in any order. Configuration file can specified few or all parameters. Parameter name and value must be separated by minimum one white character like space or tabulation. See the configuration file created by the FlashPro2000 software for details. Use the ***Notepad*** to open the configuration file..

Example:

```
      st = F_ConfigFileLoad( "c:\test\configfile.cfg" );
      if(( st & 1 ) == TRUE )
      {
           ....................
      }
      else
      {
           Info = st & 0xFFFE;
          ....................
          ....................
      }
```

F_Reset_Target

F_Reset_Target - Generate short RESET pulse on the target's device RESET line.
VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

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

Syntax:

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

Return value:

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

Example:

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

F_Get_Targets_Vcc

F_Get_Targets_Vcc - Get Vcc in [mV] supplied target device.
VALID FPA index - (1 to 16)

Syntax:

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

Return value:

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

4.3 Data Buffers access instructions

All data coming to of from target device can be saved in the temporary buffers (see Figure 4.2) located inside the API-DLL. The data saved in these buffers can be copied to target devices using an encapsulated or sequential functions. When the full block of data is ready to be saved (eg. code data), then the part of the data buffers can be modified by adding some unique data like serial numbers, calibration data etc. to each target before executing the flash programming process. Data buffers can be modified at any time, as long as the ***F_OpenInstancesAndFPAs(..)*** and ***F_Initialization()*** have been executed successfully. When more then one FPA are used then it is recommended to use only an executable instructions uses the data buffers for read and write. For example instruction ***F_Memory_Read()*** allows to make this process simultaneously. Results from each targets are saved in a Read Data buffers - one Read Buffer per one API-DLL. When the simultaneous process is done, then the content from each buffers can be individually read. The API-DLL contains four buffers (see Figure 4.2) - **Code**, **Password**, **Write Data** and **Read Data** buffers. Contents for the **Code** and **Password** buffers can be taken from the files, or data can be written directly to the specified buffer location. Data to the **Write Data** buffer can be written directly only, while data from the **Read Data** buffer can be read directly only. The FLASH memory can be programmed using contents taken from the **Code** buffer or from the **Write Data** buffer. Data to RAM, registers, I/O (seen as RAM) can be taken from **Write Data** buffer only. Contents from RAM, registers, I/O and flash are saved in **Read Data** buffer.

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

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

F_ReadCodeFile

F_ReadCodeFile - Read code data from the file and download it to internal buffer.
VALID FPA index - (1 to 16) 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 - all 16 bits width - 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 selected MCU microcontroller family will be downloaded to the internal Code buffer. Any code data located outside memory space of the selected TMS320Fxx microcontroller will be ignored and warning message will be created.

When the ***F_ReadCodeFile*** function is used then the full **Code** buffer is filled with data 0xFFFF and all flags are cleared (***empty*** flag) first. When the valid data are taken from the code buffer, the data is saved in buffer and flag modified from ***empty*** to ***used***.

Syntax:

```
MSPPRG_API      INT_X  F_ReadCodeFile( char * FileName );

  FileName:  file name including path, file name and extention
```

Return value:

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

Example:

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

F_Get_CodeCS

F_Get_CodeCS - Read code from internal code buffer and calculate the check sum.
VALID FPA index ***- (1 to 16).***

Syntax:

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

```
index - index of the desired code
```

```
Index = 1  - Calculate check sum of the code from internal code buffer.
       2  -  Return Code Cs used in the last Autprogram session.
       3  -  Return Memory Cs used in the last Autprogram session.
            Other Index values - reserved for the future option.
```

Return value:

```
      Calculated check sum  or
      -2 (0xFFFFFFFE)   - FPA_INVALID_NO
```

F_ReadPasswFile

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

VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

Function F_ReadPasswFile downloads part of the code from the file to internal memory buffer. From the code file only data related to the CSM password data are stored in the password memory buffer. All other data is ignored. Code file format and file name and location path of the desired file must be specified. Three file formats are supported - Texas Instruments text format, Motorola *.s19 format and Intel *.hex format.

Syntax:

```
 MSPPRG_API    INT_X  F_ReadPasswFile( char * FileName );

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

Return value:

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

Example:

```
      st = F_ReadPasswFile( "c:\test\demofile.txt" );
      if(( st & 1 ) == TRUE )
      {
          ....................
```

```
}
```

F_Clr_Code_Buffer

F_Clr_Code_Buffer - Clear content of the ***Code*** buffer.
VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

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

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

Return value:
```
0 - FALSE
1 - TRUE
-2 - FPA_INVALID_NO
```
Example:
```
.....................
F_Clr_Code_Buffer();
.....................
```

F_Put_Word_to_Code_Buffer

F_Put_Word_to_Code_Buffer - Write code data to Code buffer.
VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

Instruction allows to write contents of the code to code buffer instead using the ***F_ReadCodeFile*** instruction. Contents of the downloaded code data can be modified or filled with the new data, if code buffer has been cleared first (using ***F_Clr_Code_Buffer*** function).
Instruction write the data to ***Code*** buffer in specified address location and set the ***used*** flag in that location.

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

Syntax:
```
MSPPRG_API INT_X  F_Put_Word_to_Code_Buffer( LONG_X address,
```

```
INT_X data );
```

```
Parameters  value:
      code address - valid Flash or OTP address
      data         - 0x00 to 0xFFFF
```

Return value:

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

Example:

```
UINT16  code[0x20000];
      .....................
      F_Clr_Code_Buffer();
      for( address = addr_min;  address < Addr_max; address ++ )
      {
        F_Put_Word_to_Code_Buffer( address, code[address]);
      }
      .....................
```

F_Get_Word_from_Code_Buffer

F_Get_Byte_from_Code_Buffer - Read code data from code buffer.

VALID FPA index - (1 to 16)

Instruction allows to read or verify contents of the code from code buffer. Data returns value 0x0000 to 0xFFFF if in the particular **Code** buffer location the flag is set to ***used***, otherwise negative value of the error status is if returned.

Syntax:

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

```
Parameters  value:
      code address - valid Flash or OTP address
```

Return value:

```
      0x00 to 0xFFFF  - valid code data
      -1 (0xFFFFFFFF)   - code data not initialized on particular address
      -2 (0xFFFFFFFE)   - FPA_INVALID_NO
```

F_Put_Word_to_CSM_Buffer

F_Put_Word_to_CSM_Buffer - Write word to CSM password buffer.
VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

Instruction allows to write the CSM password to the CSM buffer.

Syntax:

```
MSPPRG_API      INT_X  F_Put_Word_to_CSM_Buffer( INT_X dest, INT_X address,
                                   INT_X data );
```

```
Parameters  value:
      destination  -  CSM_PASSWORD_FILE_INDEX          2
                      CSM_DEFINED_INDEX                3
      code address - 0 to 7
      data         - 0x0000 to 0xFFFF
```

Return value:

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

Example:

```
      .....................
      for( addr = 0;  addr < 8; addr ++ )
      {
        F_Put_Word_to_CSM_Buffer( CSM_DEFINED_INDEX, addr, password[addr]);
      }
      .....................
```

F_Get_Word_from_CSM_Buffer

F_Get_Word_from_CSM_Buffer - Read CSM password from code, password or defined password buffer.

VALID FPA index - (1 to 16)

Instruction allows to read or verify contents of the CSM Password buffer. Data returns value 0x0000 to 0xFFFF if in the particular **Password** buffer location the flag is set to ***used***, otherwise return value **-1** (minus one) if data is empty.

Syntax:

```
MSPPRG_API  INT_X  F_Get_Word_from_CSM_Buffer( INT_X dest INT_X address );
```

```
Parameters  value:
      destination   -   CSM_CODE_FILE_INDEX              1
                        CSM_PASSWORD_FILE_INDEX          2
                        CSM_DEFINED_INDEX                3
      address       -   0 to 7
```

Return value:

```
      0x0000 to 0xFFFF   - valid password data
      -1 (0xFFFFFFFF)    - password not initialized on particular address
      -2 (0xFFFFFFFE)    - FPA_INVALID_NO
```

F_Put_Word_to_Buffer

F_Put_Word_to_Buffer - Write word (UINT16) to temporary Write Data Buffer (See Figure 4.2)

VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

Syntax:

```
MSPPRG_API INT_X  F_Put_Word_to_Buffer( LONG_X address, INT_X data );
```

```
address:    temporary buffer address equal the OTP or Flash destination address
data:       UINT16 word to be written.
```

Return value:

```
      1 - TRUE if specified address is legal
      0-  FALSE - if address is not valid
      -2 - FPA_INVALID_NO.
```

Example:

```
  ....................
   ....................
      for( addr = addr_min; addr<=addr_max addr++ )
         st = F_Put_Word_to_Buffer( addr, data[addr] );
      st = F_Copy_Buffer_to_Flash( addr_min, size );
   ....................
```

F_Get_Word_from_Buffer

F_Get_Word_from_Buffer - Read one word from the temporary Read Data Buffer (see Figure 4.2)

VALID FPA index - (1 to 16)

Syntax:

```
MSPPRG_API      BYTE  F_Get_Word_from_Buffer( LONG_X address );
```

Return value:

```
Requested word from the specified address of the Read Data Buffer,
or negative value if address is invalid
```

4.4 Encapsulated instructions

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

- The Vcc is verified to be higher then 3.0V.
- communication interface between programming adapter and target device is initialized.
- Selected encapsulated instruction is executed (Autoprogram, Verify Fuse or Password, Memory Erase etc.).
- Communication between target device and programming adapter is terminated.
- 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 16) or 0 (ALL FPAs) executed simultaneously.

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

- initialization
- erase flash memory - restore retain data if enabled,
- confirm if memory has been erase,
- flash programming and verification,
- flash memory check sum verification,
- writing the CSM security password (if enabled).

Syntax:

```
MSPPRG_API      INT_X  F_AutoProgram( INT_X mode );
    mode = 0;
```

```
mode = 1 and up - reserved
```

Return value:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

Example:

```
...........................
if( F_Initialization() != TRUE )    //required API-Dll  - initialization
{
   // Initialization error
}
int st = F_ConfigFileLoad( "c:\test\configfile.cfg" );
if(( st & 1 ) != TRUE )
{
   ....................
}

F_SetConfig( ....., ..... ) // modify configuration if required

do{

   ....................   // prepare next microcontroller

      F_AutoProgram(0);
   ....................    //exit if the last microcontroller
                           // has been programmed

  } while(1);
 ....................
```

F_Verify_CSM_Password

F_Verify_CSM_Password -Verify the CSM Security Password.

VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed simultaneously.

Syntax:

```
MSPPRG_API     INT_X  F_Verify_CSM_Password( void );
```

Return value:

```
0 - FALSE    (JTAG fuse blown or BSL password wrong)
1 - TRUE     (valid access to target device)
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

F_Memory_Erase

F_Memory_Erase **-** Erase Target's Flash Memory
VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed simultaneously.

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

Syntax:

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

Return value:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

F_Memory_Blank_Check

F_Memory_Blank_Check **-** Check if the Target's Flash Memory is blank.
VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed simultaneously.

Syntax:

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

Return value:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

F_Memory_Write

F_Memory_Write **-** Write content taken from the Code file to the Target's Flash Memory.
VALID FPA index - (1 to 16) 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:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

F_Memory_Verify

F_Memory_Verify **-** Verify contents of the Target's Flash Memory and Code Buffer**.**
VALID FPA index - (1 to 16) 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 valid data taken from the Code Buffer 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:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

F_Memory_Read

F_Memory_Read - Read contents of the Target's Flash Memory and save it in the temporary Read Data buffer (see Figure 4.2)**.**

VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed simultaneously.

Syntax:

```
MSPPRG_API      INT_X  F_Memory_Read( void );
```

Return value:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

Example:

```
LONG_X addr;
   ....................
      st = F_Memory_Read();
      if ( st == TRUE )
       {
         for( addr = addr_min; addr<=addr_max; addr++)
           data[ addr ] = F_Get_Word_from_Buffer( addr );
       }
   ....................
```

F_Write_CSM_Password

F_Write_CSM_Password - Write CSM password to target device.

VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

To write the CSM security password, the flag CFG_CSMLOCK_PASSWORD_ENABLE in the configuration setup must be enable

Syntax:

```
MSPPRG_API      INT_X  F_Write_CSM_Password( void );
```

Return value:

```
0 - FALSE
1 - TRUE
-2 - FPA_INVALID_NO.
or Status - see error list
```

4.5 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.*

F_Open_Target_Device

F_Open_Target_Device - Initialization communication with the target device.
VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed simultaneously.

When **F_Open_Target_Device** is executed, then

- Vcc is verified to be higher then 3.0V.
- communication between programming adapter and target device is initialized.

Note: The correct CSM password should be downloaded to password or data buffer to be able to activate target devices if the CSM password is used. If password is unknown then access to the target device cannot be established

Target device is ready to get other sequential instructions.

Syntax:

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

Return value:

```
0 - FALSE   (communication failed)
1 - TRUE    (communication is OK)
2 - JTAG security blown - communication failed
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

Example:

```
int st;
....................
   F_Open_Target_Device();
....................
   F_Segment_Erase(0x3F0000);
   st = F_Sectors_Blank_Check( 0x3F0000, 0x0800 );
   if ( st != TRUE )
    { ............... }
....................
   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 16) 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.
- Target device is released from the programming adapter.

Syntax:

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

Return value:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

Example:

See example above (**F_Open_Target_Device**).

F_Segment_Erase

F_Segment_Erase - Erase any segment of the TMS320Fxx Flash memory.

VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

Parameters:

segment address - valid address of the flash segment - any address located in the space of the selected flash,

To erase a memory segment specify an address within that memory segment. For example to erase segment 0x3F0000-0x3F3FFF any address from the range 0x3F0000-0x3F3FFF can be specified.

Syntax:

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

Return value:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

Example:

```
....................
   F_Segment_Erase(0x3F0000);    // erase segment 0x3F0000 to 0x3F3FFF
   F_Segment_Erase(0x3F2345);    // erase the same segment
   F_Segment_Erase(0x3F4000);    // erase segment 0x3F4000 to 0x3F7FFF
....................
```

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 16) or 0 (ALL FPAs) executed sequentially.

Parameters:

start address

stop address

Syntax:

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

Return value:

```
0 - FALSE
1 - TRUE
```

```
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

F_Write_Word_to_RAM

F_Write_Word - Write one word to RAM, registers, IO etc. without FLASH or OTP.
VALID FPA index ***- (1 to 16) or 0 (ALL FPAs) executed sequentially.***

Note: Do not write any data to address below 0x8F00 where the Flash-API is located. Also do not modify the CLK registers

Write one word to any location of the target device. Write to Flash or OTP has no effect.
Parameters:
address
data - one word to be written to target device

Syntax:

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

Return value:

```
0 - FALSE
1 - TRUE
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
or Status - see error list
```

Example:

```
F_Write_Word( 0x8F00, 0x2143 );
```

F_Read_Word

F_Write_Word - Read one word from RAM, registers, IO, OTP, Flash etc.
VALID FPA index ***- (1 to 16) or 0 (ALL FPAs) executed sequentially.***

Read one word from any location of the target device.
Parameters:
address

Syntax:

```
MSPPRG_API  INT_X  F_Read_Word( LONG_X addr );
```

Return value:

```
data - if result is zero or positive
-2 (0xFFFFFFFE)   - FPA_INVALID_NO
minus status (see error list) if rewult is negative
```

Example:

```
int st;
      st = F_Read_Word( 0x8F00 );
      if( st >= 0 )
         data = st;
      else
         st = -st;   // see error list
```

F_Copy_Buffer_to_Flash

F_Copy_Buffer_to_Flash - Write "size" number of words from the Write Data Buffer (see Figure 4.2) to Flash or OTP. Starting address is specified in the "start address"**.**

VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

Syntax:

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

Parameters:

```
start address      - valid OTP or Flash Address
size               - No of word to be written to OTP or Flash
```

Return value:

```
1 - TRUE if data has been saved successfully
0 - FALSE.
-2 - FPA_INVALID_NO.
```

NOTE: ***Specified address in the Write Data Buffer is the same as a physical FLASH address.***

Note: Function is useful for writing small data block, usually shorter then 200 bytes, like calibration data, serial numbers etc. Function can also be used to writing longer data block, however for this purpose it is recommended to use an encapsulated function ***F_Memory_Write()*** described in this manual. The ***F_Copy_Buffer_to_Flash()*** (the same as the ***F_Memory_Write_Data()***

function) uses byte by byte flash write procedure. When the ***JTAG*** or ***Spy-Bi-Wire*** interface is used, then the ***F_Copy_Buffer_to_Flash()*** use the ***JTAG/SBW*** protocol to directly program the Flash memory. The ***F_Memory_Write()*** function first download the Flash Loader to RAM memory, and use the block write flash procedures, speeding up programming process.

Example:

```
  ....................
   ....................
      for( k = 0; k<0x100; k++ )
       {
         addr = 0x3F0000 + k;
         st = F_Put_Word_To_Buffer( addr, data[k] );
       }
      st = F_Copy_Buffer_to_Flash( 0x3F0000, 0x100 );
   ....................
```

F_Copy_Flash_to_Buffer

F_Copy_Flash_to_Buffer - Read specified in "size" number of bytes from the Flash and save it in the Read Data Buffer (see Figure 4.2). Starting address is specified in the "start address"**.**

VALID FPA index - (1 to 16) or 0 (ALL FPAs) executed sequentially.

Syntax:

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

Parameters:

```
start address      - valid OTP or Flash address,
size               - number of words to be read
```

Return value:

```
1 - TRUE if data has been read successfully
0 - FALSE.
-2 - FPA_INVALID_NO.
or Status - see error list
```

NOTE:
Specified address in the temporary flash buffer is the same as a physical FLASH address.

Example:

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

```
    st = F_Copy_Flash_to_Buffer( 0x3F0000, 0x100 );
    if( st == TRUE )
     {

       for( k = 0; k<100; k++ )
          {
            addr = k + 0x3F0000;
            data[k] = F_Get_Word_from_Buffer( addr );
          }
     }
    else
     {
        ...................
     }
 ....................
```