IAR Embedded Workbench flash loader

User Guide

The purpose of this guide is to give an understanding of the flash loader concept in IAR Embedded Workbench. You will get information about how to use the flash loader in the IAR Embedded Workbench IDE. The document also describes how to write and debug your own flash loader driver. Finally, the flash loader framework API functions are described in detail.

Note: In this document the notation *xx* represents the numeric part of the filename extension for the processor you are using.

Contents

CONTENTS 2

USING FLASH LOADERS 3

THE FLASH LOADER MECHANISM 3

BUILD CONSIDERATIONS 4

SETTING UP THE FLASH LOADER(S) 4

USING AN OPTIONAL FLASH LOADER C-SPY MACRO FILE 4

FLASH LOADER OVERVIEW DIALOG BOX 5

FLASH LOADER CONFIGURATION DIALOG BOX 5

THE FLASH LOADER IN DEPTH 7

INTERFACING WITH THE FLASH LOADER FRAMEWORK 8

FLASH LOADER DRIVER EXAMPLE 8

BUILDING YOUR OWN FLASH LOADER 9

DEBUGGING A FLASH LOADER 10

THE FLASH LOADER FRAMEWORK API 11

INITIALIZE FUNCTIONS 11

ARGUMENT PASSING FUNCTIONS 12

C-SPY USER INTERFACE FUNCTIONS 12

FILE FUNCTIONS 13

Using flash loaders

A flash loader is a program agent that is downloaded to the target. It fetches your application from the C-SPY debugger and programs it into flash memory. The flash loader uses the file I/O mechanism to read the application program from the host. You can select one or several flash loaders, where each flash loader loads a selected part of your application. This means that you can use different flash loaders for loading different parts of your application.

A set of flash loaders is provided with IAR Embedded Workbench. In addition to these, more flash loaders can be provided by chip manufacturers and third-party vendors. The flash loader API, documentation, and implementation examples are available to make it possible for you to implement your own flash loader.

THE FLASH LOADER MECHANISM

The following illustration shows the flash loader mechanism:

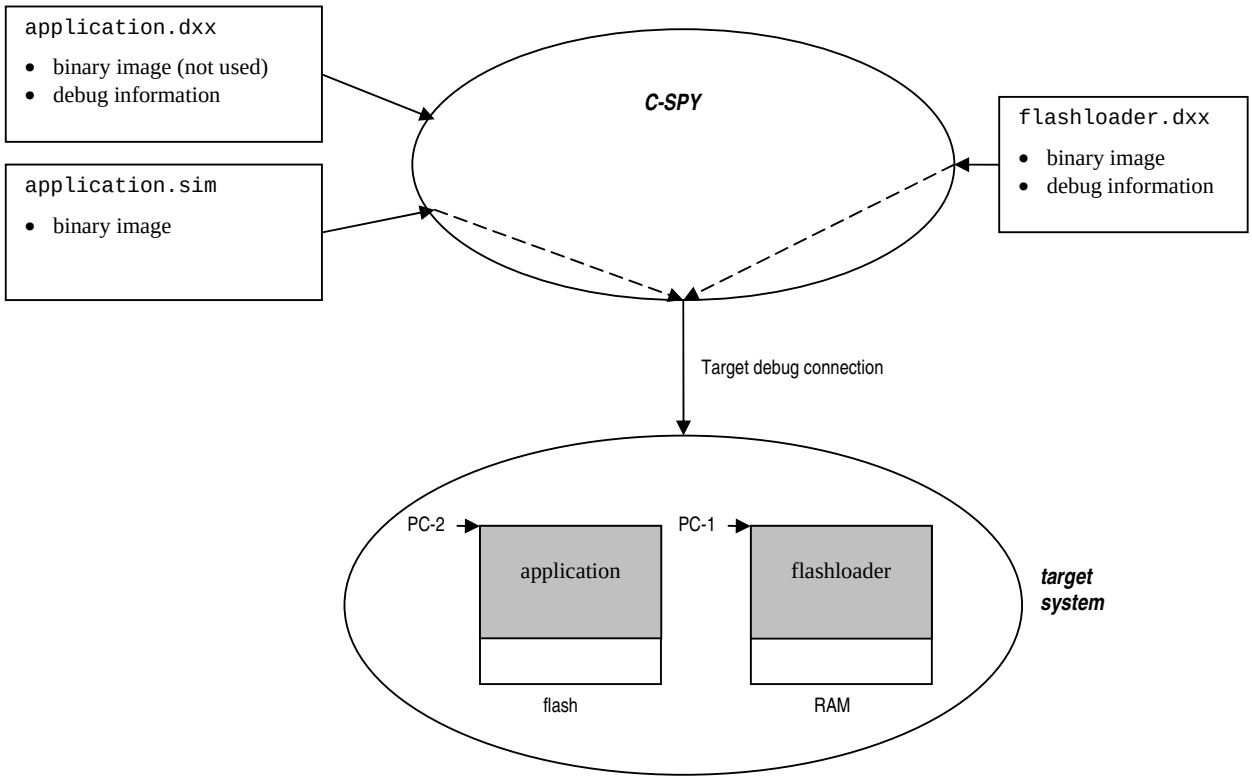

Figure 1: Flash Loader mechanism

The following steps will be performed when the debug session starts:

1. C-SPY downloads the flash loader into target RAM.
2. C-SPY starts execution of the flash loader (PC-1).
3. The flash loader opens the file holding the application code.
4. The flash loader reads the application code and programs it into flash memory.
5. The flash loader terminates.
6. C-SPY switches context to the user application (PC-2).

The steps 1 to 5 are performed for each selected flash loader.

BUILD CONSIDERATIONS

When you build an application that will be downloaded to flash, special consideration is needed. Two output files must be generated. The first is the usual UBROF file (`dxx`) that provides the debugger with debug and symbol information. The second file is a simple-code file (filename extension `sim`) that will be opened and read by the flash loader when it downloads the application to flash memory.

The simple-code file must have the same path and name as the UBROF file except for the filename extension.

To create the extra output file, choose **Project>Options** and select the **Linker** category. Select the **Allow C-SPY-specific extra output file** option. On the **Extra Output** page, select the **Generate extra output file** option. Choose the **simple-code** output format and the format variant **None**. Do not override the default output file.

SETTING UP THE FLASH LOADER(S)

To use a flash loader for downloading your application:

1. Choose **Project>Options**.
2. Choose the debugger you are using in the category list and click the **Download** or **Flash Loader** tab depending on your product installation.
3. Select the **Use Flash loader(s)** option, and click the **Edit** button.
4. The **Flash Loader Overview** dialog box is displayed and it lists all currently available flash loaders; see *Flash Loader Overview dialog box*, page 5. You can either select a flash loader or select **New** to open the **Flash Loader Configuration** dialog box.

In the **Flash Loader Configuration** dialog box, you can configure the download. For reference information about the different flash loader options, see *Flash Loader Configuration dialog box*, page 5.

USING AN OPTIONAL FLASH LOADER C-SPY MACRO FILE

You can use a C-SPY setup macro to setup the target system before loading the flash loader to RAM. One example of when this is needed is for targets where the RAM is not functional after a reset; the macro is used for setting up the necessary registers for proper RAM operation.

The following criteria must be met for a macro function to be executed before downloading the flash loader:

- The macro file must be located in the same directory as the flash loader.
- The macro file must have the filename extension `mac`.
- The name of the macro file must be the same as the flash loader.
- The setup macro function `execUserFlashInit()` must be defined in the macro file. This macro function will be called by the debugger before the flash loader is loaded into RAM. Note that during debugging of the flash loader, the flash loader must be running as an application and the setup macro `execUserPreload()` must be used instead of `execUserFlashInit()`.

The macro function `execUserFlashExit()` can be used for restoring device settings after the completion of the flash loader, if necessary.

FLASH LOADER OVERVIEW DIALOG BOX

The **Flash Loader Overview** dialog box—available from the **Debugger>Download** page—lists all defined flash loaders. If you have selected a device on the **General Options>Target** page for which there is a flash loader, this flash loader is by default listed in the **Flash Loader Overview** dialog box.

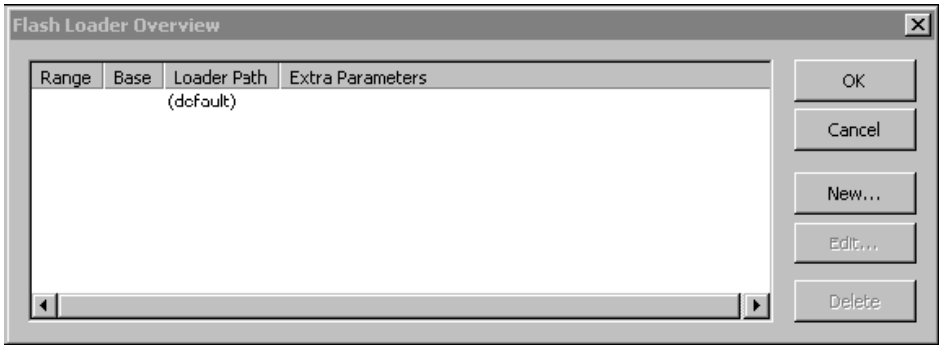

Figure 2: Flash Loader Overview dialog box

The following buttons are available:

Button	Description
OK	The selected flash loader(s) will be used for downloading your application to memory.
Cancel	Standard cancel.
New	Opens the **Flash Loader Configuration** dialog box where you can specify what flash loader to use; see *Flash Loader Configuration dialog box*, page 5.
Edit	Opens the **Flash Loader Configuration** dialog box where you can modify the settings for the selected flash loader; see *Flash Loader Configuration dialog box*, page 5.
Delete	Deletes the selected flash loader configuration.

Table 1: Function buttons in the Flash Loader Overview dialog box

FLASH LOADER CONFIGURATION DIALOG BOX

In the **Flash Loader Configuration** dialog box—available from the **Flash Loader Overview** dialog box—you can configure the download.

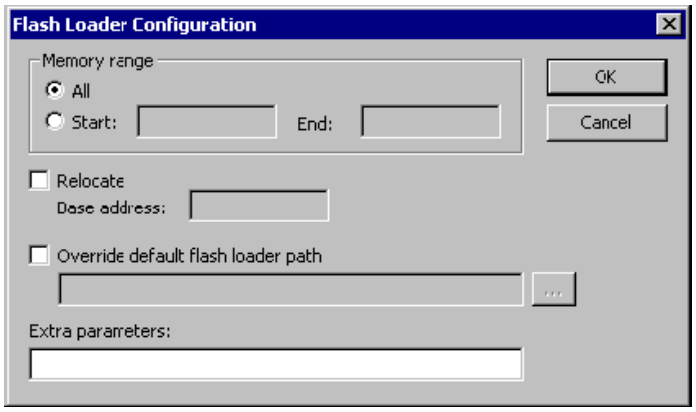

Figure 3: Flash Loader Configuration dialog box

Memory range

Use the **Memory range** options to specify the part of your application to be downloaded to flash memory. Choose between:

All	The whole application is downloaded using this flash loader.
Start/End	The part of the application available in the memory range will be downloaded. Use the **Start** and **End** text fields to specify the memory range.

Relocate

Use the **Relocate** option to override the default flash base address. The default base address used for writing the first byte—the lowest address—to flash is specified in the linker command file used for your application. However, it can sometimes be necessary to override the flash base address and start at a different location in the address space. This can, for example, be necessary for devices that remap the location of the flash memory.

Use the **Base address** text box to specify a numeric value for the new base address. You can use the following numeric formats:

123456	Decimal numbers
0x123456	Hexadecimal numbers
0123456	Octal numbers

Override default flash loader path

A default flash loader is selected based on your choice of device on the **General Options>Target** page. To override the default flash loader, select **Override default flash loader path** and specify the path to the flash loader you want to use. A browse button is available for your convenience.

Extra parameters

Some flash loaders define their own set of specific options. Use this text box to specify options to control the flash loader. For information about any additional flash loaders and flash loader options to those described here, see the release notes delivered with your IAR Embedded Workbench installation.

The flash loader in depth

The flash loader is a native application developed using IAR Embedded Workbench. The task of the flash loader is to read the application binary image from the host using file I/O, unpack the image, and write the image to flash memory. The flash loader is partitioned in two parts; the flash loader framework and the flash loader driver.

The flash loader framework implements functionality that is common to all flash loaders. This includes reading of the binary image from the debugger, a mechanism for passing user arguments (options) to the flash loader, and support for creating GUI elements for user interaction. The GUI elements available are a message box, message log, and a progress bar. The progress bar is by default handled by the framework. The source code for the framework is written by IAR Systems and included in the IAR Embedded Workbench product.

The flash loader driver is a small piece of code that handles the actual writing to flash memory. A set of flash loader drivers for various devices is included in the product. Due to the simplicity of the flash loader driver it is possible for you to write a custom driver for a device that is not supported by IAR Systems.

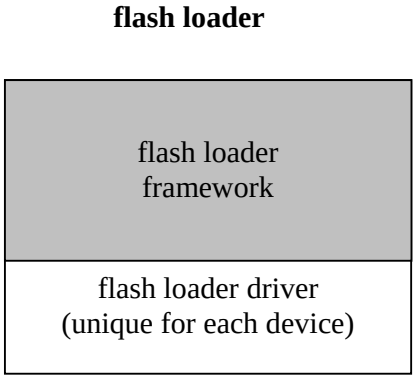

Figure 3: Flash loader parts

Flash loaders must follow the naming convention `Flash<device>.dxx`. For example a flash loader built for the hypothetical device IAR X99 should be named `FlashIarX99.dxx`.

The source code for flash loaders, provided by IAR Systems is located in:

`<target>\src\flashloader\framework`	The source code including API header file for the flash loader framework.
`<target>\src\flashloader\<vendor>\Flash<device>`	The source code for individual flash loader drivers including project files.

Flash loader executables provided by IAR Systems are located in:

`<target>\config\flashloader\<vendor>\Flash<device>.dxx`	The executables for individual flash loader drivers.
`<target>\config\flashloader\<vendor\Flash<device>.mac`	An optional C-SPY macro file. If a macro file with the same name as the flash loader executable is present, it will be loaded and executed before the flash loader is placed in RAM. This is useful for devices where certain I/O registers must be initialized for the RAM to work correctly.

Interfacing with the flash loader framework

The flash loader framework will first initialize the flash loader driver. At this point the driver can perform various initializations, but must at a minimum register its write function with the framework.

After initialization, the framework will use the write function of the driver to transfer one byte at a time to the flash loader. Depending on the flash algorithm used, it might be necessary to buffer bytes in the driver to fill a complete sector before writing the sector to the flash memory. The last write to the driver from the framework will be a flush request to allow for the driver to flush any remaining data in the sector buffer. If the flash loader driver does not buffer any data, the flush operation can simply be ignored.

The flash loader driver does not return any error status to the framework. Should an error occur in the driver, the driver is responsible for notifying the user by calling the `FlMessageBox()` API function and then terminating the flash loader using the `FlErrorExit()` function.

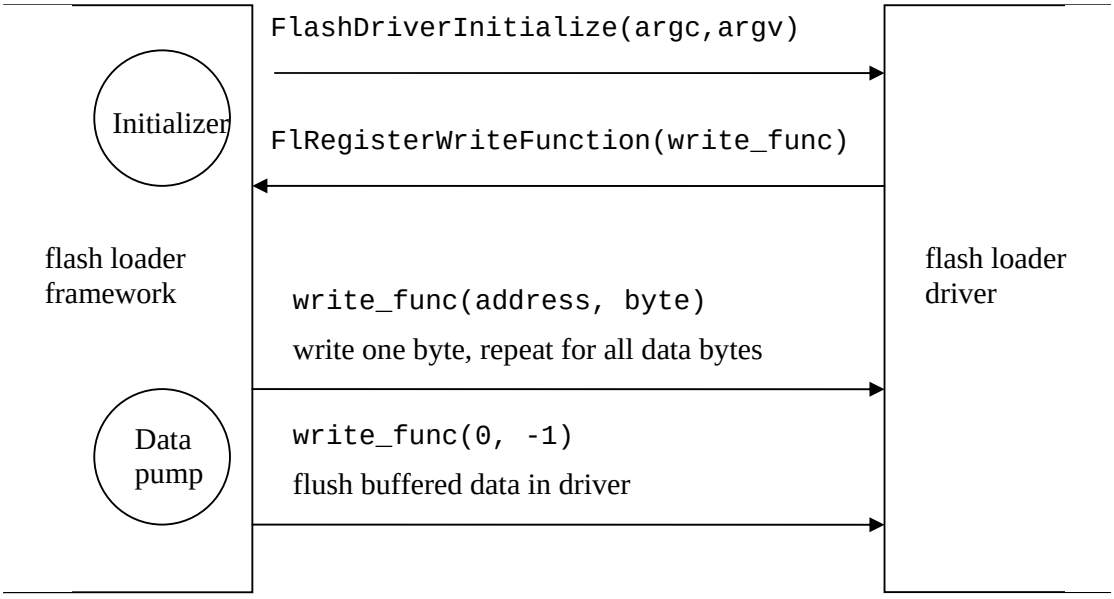

Figure 3: Flash Loader framework

FLASH LOADER DRIVER EXAMPLE

This example shows how to write a flash loader driver for a device. For simplicity the example device have a flash that is easy to program, a simple flash algorithm can be used that allows individual bytes to be written to flash memory in a single operation. The example also shows how to read a user specified option to inform which clock frequency the device is running at.

```
// Flash loader driver example.

#include <stdio.h>
#include <stdlib.h>
#include "Interface.h" // The flash loader framework API declarations.


// The CPU clock speed, the default value 4000 kHz is used if no clock
//option is found.
```

```
static int clock = 4000;


// Write one byte to flash at addr.
// If byte == -1 the flash loader framework signals a flush operation
// at the end of the input file.
static void FlashWriteByte(unsigned long addr, int byte)
{
  unsigned char* ptr = (unsigned char*)addr;

  if (byte == -1)
    return;  // Simply return when the flush operation is requested.

  // Insert device specific instructions here to enable write access
  // to the flash device.
  // ...

  *ptr = byte; // Write data byte to flash.

  // If an error occurs when writing to flash, this can be communicated
  // to the user by using code like
  //   if (ret != STATUS_CMD_SUCCESS)
  //   {
  //     FlMessageBox("CMD_ERASE_SECTORS failed.");
  //     FlErrorExit();
  //   }
  // A message box will be displayed by C-SPY and the downloading will
  // terminate after the user has clicked the OK button.
}


void FlashDriverInitialize(int argc, char const* argv[])
{
  const char* str;

  // Register the flash write function.
  FlRegisterWriteFunction(FlashWriteByte);

  // See if user has passed a clock speed option.
  // If not, the default CCLK value is used.
  str = FlFindOption("--clock", 1, argc, argv);
  if (str)
  {
    clock = strtoul(str, 0, 0);
  }
}
```

For an example on how to implement a flash loader with a sector buffer, refer to the source code for the flash loader drivers in the IAR Embedded Workbench installation.

Building your own flash loader

Perform the following steps to build your own flash loader:

1. Make a copy of one of the existing flash loader projects from `<target>\src\flashloader\`. Make sure that the include file path used by the compiler

includes both the flash loader framework directory `<target>\src\flashloader\framework` and the directory of the flash loader driver.

2 Replace and rename the two files `Flash<device>.c` and `Flash<device>.h` with appropriate names and an implementation that matches your device.

3 Set up the linker command file to match your device. Make a copy of an existing linker command file that places the code in RAM and modify the address definition lines:

```
-DMEMSTART=40000000
-DMEMEND=40003FDF
```

The actual addresses used must map to your target hardware. Note that both code and data for the flash loader is downloaded to RAM, which is why the ROM and RAM sections map to the same memory area. Stack and heap sizes should be kept at a minimum; the framework requires around 300 bytes of stack size. Note that the numbers are specified as hexadecimal values.

```
-D_CSTACK_SIZE=180
-D_IRQ_STACK_SIZE=40
-D_HEAP_SIZE=0
```

The read buffer must be as large as possible to increase download performance, each debug connection transaction is costly and maximizing the number of bytes per transfer will increase performance. The flash loader framework uses the memory between the heap segment and `RAMEND` (as specified in the linker command file) for the read buffer. This guarantees that the read buffer gets all of the remaining memory. The framework will give an error if the resulting read buffer becomes smaller than 256 bytes as lower numbers will severely hurt performance.

4 To set the required build options, choose **Project>Options** and select **Linker** from the category list. On the **Output** page, select **With I/O emulation modules.** Click **OK.** The resulting output file will have the `dxx` filename extension.

5 To build the project; choose **Project>Make**.

The flash loader can now be used for downloading an application into flash. Perform the following steps to use your own flash loader:

1 Open your application project, and open the debugger download options dialog box. Enable the **Flash download** option and select the **Override default flash loader** option, browse to the flash loader output file you created. Any options that must be passed to your flash loader can be written in the **Flash loader arguments** text field.

2 Starting the debugger will now use your flash loader to download the application program to flash.

Debugging a flash loader

Debugging a flash loader can be done in the same way as an ordinary application. Note that it cannot be debugged when installed in the debugger as a flash loader; it can only be debugged when the flash loader is the current project in IAR Embedded Workbench.

The flash loader framework has a debug environment that is controlled by C preprocessor macro variables defined in the header file `DriverConfig.h`, which is included by the header file `Config.h` in the framework. In the `Config.h` file you can see what variables that can be overridden in `DriverConfig.h`.

There are a few things that are different when running the flash loader as a standalone application in the debugger. To enable the framework debug environment, the debug macro variable `DEBUG` must be set. The file to be flashed must be explicitly setup using the macro variable `DEBUG_FILE`.

The `argc/argv` argument passing mechanism does not work in standalone debugging, arguments have to be hard coded using the C preprocessor macro variable `DEBUG_ARGS`.

The flash loader framework API

This section describes the API functions implemented in the flash loader framework. Many of these functions are of little interest for most flash loader drivers but are included for completeness.

The functions are marked to indicate the intended usage:

- *Mandatory* means that a flash loader driver must implement this function.
- *Optional* means that a flash loader driver may use this function depending on the needed functionality.
- *Framework only* means that the function is used by the flash loader framework and should normally not be used by the flash loader driver.

Function prototypes for all API functions are defined in the header file `<target>\src\flashloader\framework\Interface.h`.

INITIALIZE FUNCTIONS

```
void FlashDriverInitialize(int argc, char const* argv[]);
```

Usage: mandatory

This function must be defined by the flash loader driver and is used by the flash loader framework to initialize the flash loader driver.

The number of flash arguments is passed in `argc`.

The flash arguments are passed in the `argv` array.

Flash arguments allow parameters to be passed from the C-SPY flash options dialog to the flash loader. A typical example of when this can be used is for passing CPU clock speed to the flash loader driver.

```
void FlRegisterWriteFunction(WriteFunctionType write_func);
```

Usage: mandatory

Used by the flash loader driver to register the write function with the flash loader framework. The argument `write_func` is the function pointer to the write function. The framework will call this function for every byte to be flashed. When the framework calls the write function, it will pass the byte and the address of where to write the byte as parameters. The sequence of addresses is guaranteed to be increasing but not contiguous (gaps may be present). The flash loader driver must call this function from `FlashDriverInitialize()`.

```
typedef void (*WriteFunctionType)(unsigned long address, int byte);
```

Usage: mandatory

This type declaration defines the function pointer type of the write function that must be defined in the flash loader driver.

```
unsigned long FlGetBaseAddress();
```

Usage: optional

Optional function to get the flash base address set by the user in the IDE. This function returns 0xFFFFFFFF if no flash base address is set by the user. For example, AMD compatible flash devices do not rely on any control registers for program and erase procedures. Instead, bus write sequences using various address offsets from the flash base address are used. In this case it will be necessary to know the flash base address to adapt to possible flash remapping.

ARGUMENT PASSING FUNCTIONS

```
const char* FlFindOption(char* option, int with_value, int argc, char
const* argv[]);
```

Usage: optional

The function looks for the specified option in the argument array `argv`.

The `with_value` parameter specifies if the function is used to see if an option exists in `argv` or if the value of the option should be returned. The value of an option is the next argument after the matching argument in `argv`. Set `with_value` to 0 when checking for a flag option like `--small_ram`. Set `with_value` to 1 when checking for an option with value like `--speed 14600`.

The `argc` parameter is the number of arguments in the `argv` array. The `argv` parameter is an array of string pointers. The `argc/argv` parameters in `FlashDriverInitialize()` can be used directly when calling this function.

The function returns a null pointer if the option is not found in `argv`.

The function returns a pointer to the argument after the matching option if `with_value` is set to 1.

The function returns a pointer to the entry in `argv` that matches the option if `with_value` is set to 0.

```
int FlMakeArgs(char* args, char const* argv[]);
```

Usage: framework only

Takes a string with space/tab separated options and makes an `argv` string array with one array element per option. The `argv` character pointer array must be large enough to accommodate all options in the `args` string.

The function returns the resulting number of strings of the `argv` array (number of arguments).

C-SPY USER INTERFACE FUNCTIONS

```
void FlErrorExit();
```

Usage: optional

Terminates the flash loader and signals C-SPY that the flash download failed.

```
void FlMessageBox(char* msg);
```

Usage: optional

C-SPY will display a message box window with the text given by the `msg` parameter. Text may be split on multiple lines by embedding newlines (\n) in the message string. The flash loader execution will halt until the message box OK button is pressed.

```
void FlMessageLog(char* msg);
```

Usage: optional

C-SPY will display a log message given by `msg` in the debug log window. Text may be split on multiple lines by embedding newlines (\n) in the message string.

```
void FlOverrideProgressBar();
```

Usage: optional

Overrides the default progress bar implemented by the flash loader framework. In most cases the flash loader driver will not need to handle the progress bar as this is handled by the input file read routines in the framework. If the flash loader driver implements its own progress bar, it must disable the default implementation by calling this function.

```
void FlProgressBarCreate(char* title);
```

Usage: optional

C-SPY will create a progress bar window. The title parameter string will be displayed above the progress bar.

```
void FlProgressBarDestroy();
```

Usage: optional

C-SPY will close the progress bar window.

```
void FlProgressBarUpdate(int progress);
```

Usage: optional

C-SPY will update the progress bar to reflect the value of the `progress` parameter. The valid range of `progress` is 0..100. The progress bar must be created for this function to succeed. The number of calls to `FlProgressBarValue()` must be kept as low as possible (<= 10), this is to reduce the number of (slow) transactions on the JTAG bus.

FILE FUNCTIONS

```
void FlFileClose(int fd);
```

Usage: framework only

Closes the open file associated with file handle `fd`.

```
int FlFileOpen(char* name);
```

Usage: framework only

The function opens a file for binary mode reading.

Returns a file handle if the open succeeded.

Returns -1 if the file could not be opened.

```
int FlFileReadByte(int fd);
```

Usage: framework

Reads one byte from the open file associated with file handle `fd`.

The function returns the byte read, or -1 if end of file is reached.