IAR Embedded Workbench flash loader developer guide

Overview

This document provides:

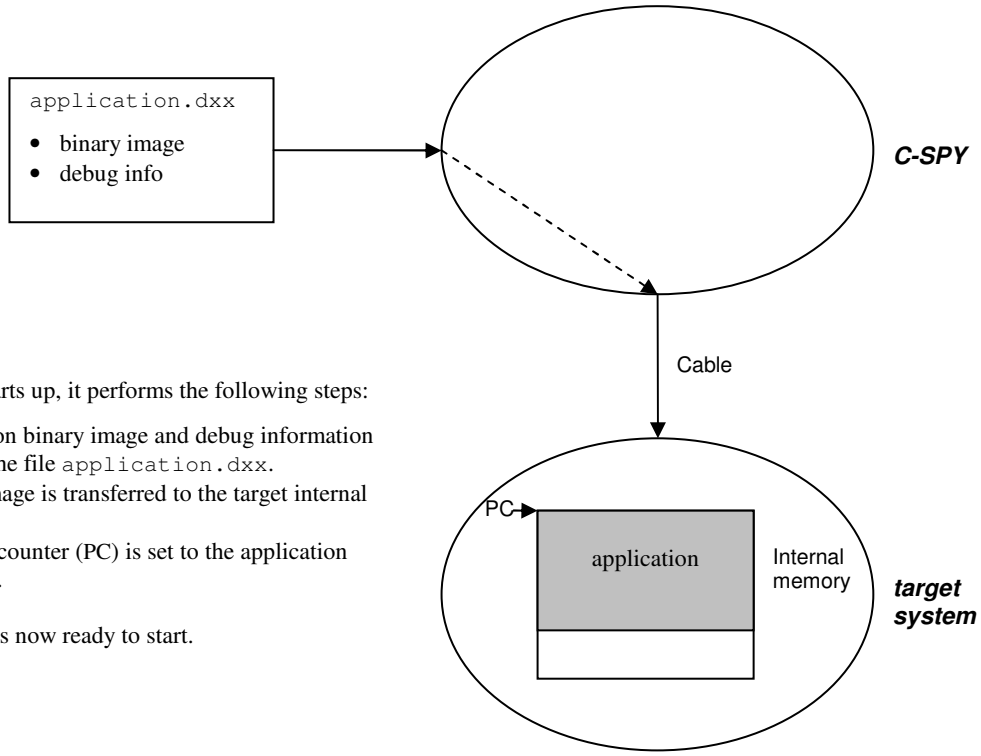
- A functional overview of application downloads into internal memory as well as application downloads into flash memory. The internal memory considered in this case can be either RAM or ROM depending on the programming capabilities of your system. Flash memory can be either internal or external.
- Information about the flash loader and its division into a framework and a driver component.

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 is used and stands for two digits, which are identifiers for the processor you are using.

Application download into internal memory

Application download into internal memory takes place during the C-SPY start-up sequence and is handled by C-SPY itself. The download is handled by writing data to the target using the connection between PC and target system.



When C-SPY starts up, it performs the following steps:

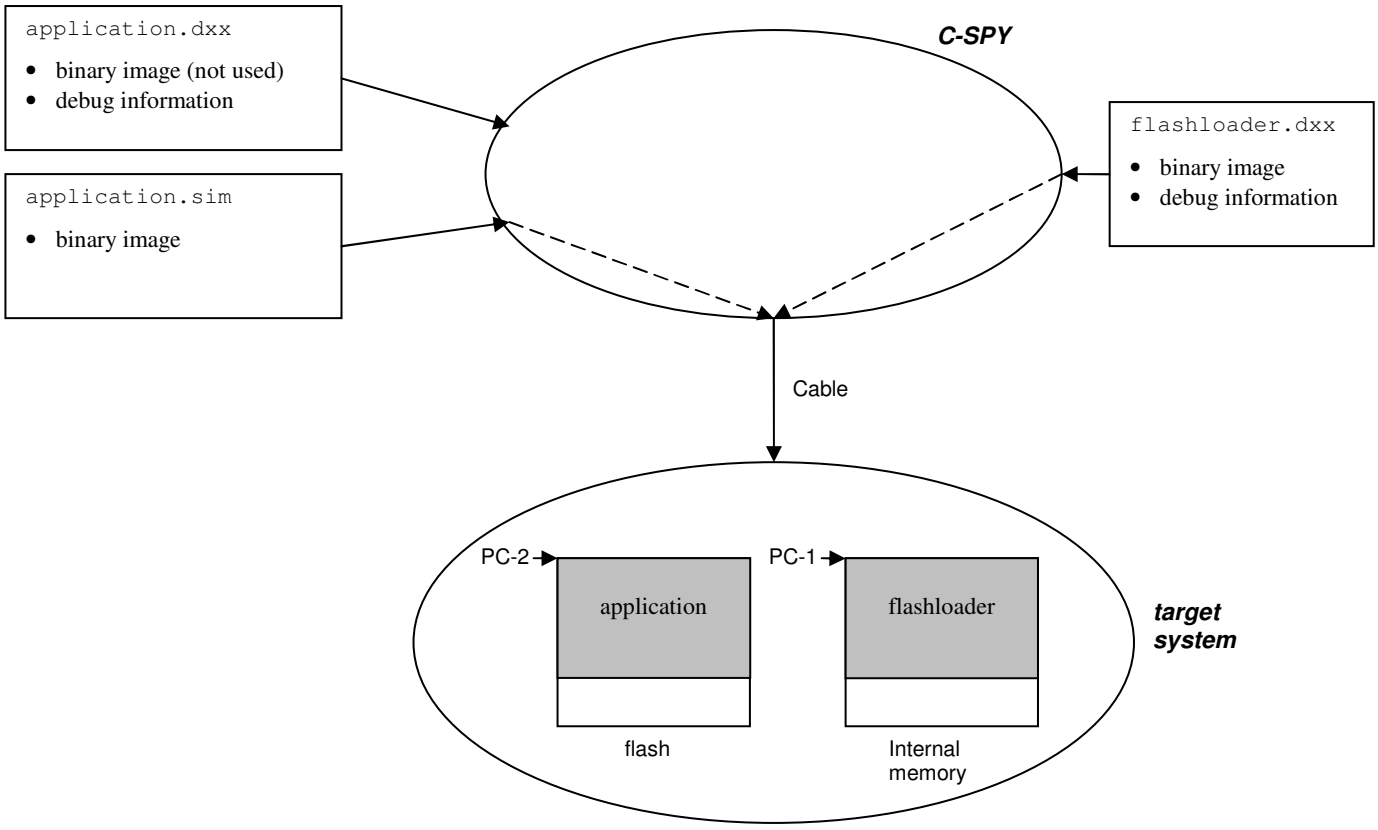
- The application binary image and debug information is read from the file `application.dxx`.
- The binary image is transferred to the target internal memory.
- The program counter (PC) is set to the application entry point in.

The application is now ready to start.

Application download into flash memory

Application download into flash takes place during the C-SPY start-up sequence and is handled by a special program agent called the flash loader. The flash loader is downloaded into target internal memory, and the flash loader will then write the application to the flash memory. The linker will create two output files, the first is the normal UBROF object format file (filename extension `dxx`), and the second is a simple binary format file (simple-code, extension `sim`). The simple-code format is easy to unpack and is compact, important factors for a flash loader executing on the target hardware.

The flash loader is a normal EW application which can be developed and debugged in the Embedded Workbench.



When C-SPY starts up, it performs the following steps:

- The flash loader binary image is read from the file `flashloader.dxx`.
- The binary image is transferred to the target internal memory.
- The program counter (PC-1) is set to the flash loader entry point and execution is started.
- The flash loader loads the application binary image from the file `application.sim` (using file I/O through the HW connection), and programs the application binary image into the flash memory.
- C-SPY reads the debug information from the file `application.dxx` and sets the program counter (PC-2) to the application entry point in flash.

The application is now ready to start.

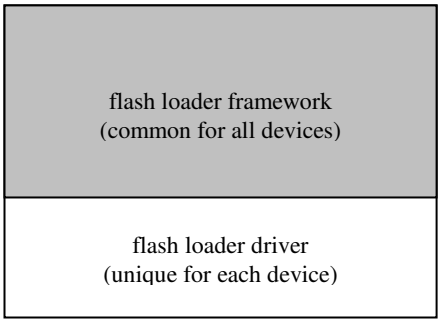
The flash loader

The flash loader is a native EW application developed using the 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 is common to all flash loaders; the source code for the framework is written by IAR Systems and included with the *IAR Embedded Workbench*. The flash loader driver is a small piece of code that handles the actual writing to flash memory. Example flash loader drivers are included in the product. Due to the simplicity of the flash loader driver it is easy for you to write a custom driver for a certain device.

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.

flash loader



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 are located in

`\src\flashloader\framework`	The source code including API header file for the flash loader framework.
`\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

`\config\flashloader\<vendor>\Flash<device>.dxx`	The executables for individual flash loader drivers.
`\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 internal memory. This is useful for devices where certain I/O registers must be initialized for the internal memory to work correctly.

The optional flash loader C-SPY macro file

A C-SPY macro may need to be executed to setup the target system before loading the flash loader to the internal memory. One example of when this is needed is for targets where the internal memory is not functional after a reset; the macro is used for setting up the necessary registers for proper internal memory 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.

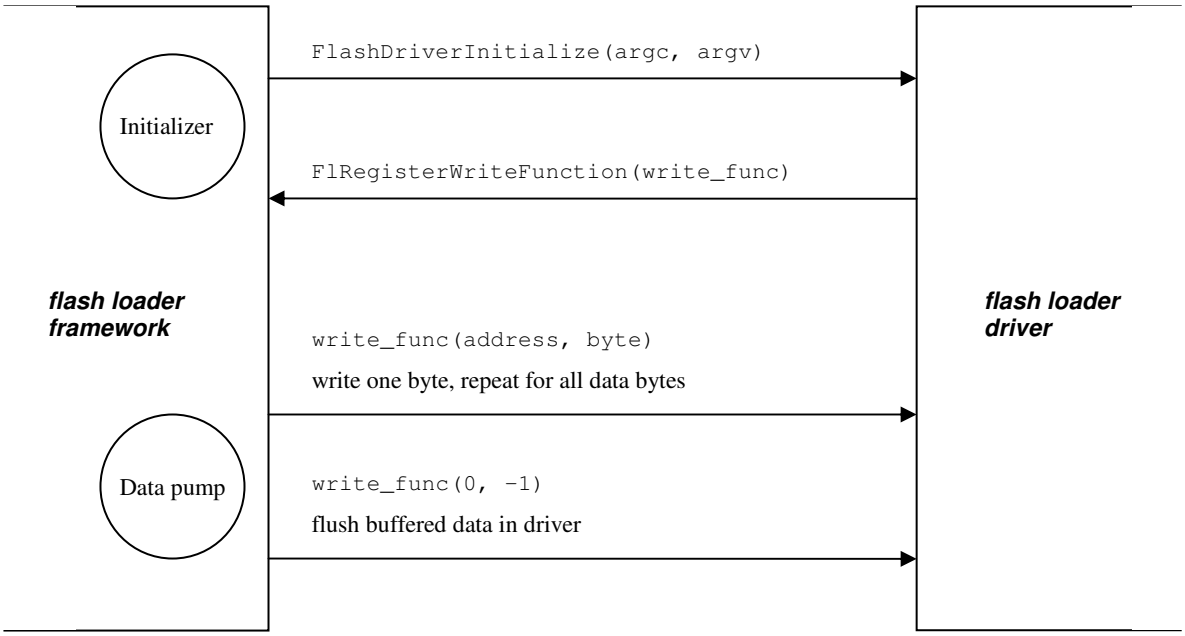
- The macro function `execUserFlashInit()` must be defined in the macro file, this macro function will be called by the debugger before the flash loader is loaded into internal memory.

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 a 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 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.



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.

For an example on how to implement a flash loader with a sector buffer, refer to the Philips flash loader driver in the Embedded Workbench installation.

```
// 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 some 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);
  }
}
```

Building a flash loder

Make a copy of an existing flash loader, for example `\src\flashloader\Example`.

Make sure that the include file path used by the compiler includes both the flash loader framework directory `\src\flashloader\framework` and the directory of the flash loader driver.

Replace and rename the two files `FlashLoaderExample.c` and `FlashLoaderExample.h` with appropriate names and an implementation that matches your device.

The linker control file must be setup to match your device. Use an existing file for your derivative `(\config\lnk<derivative>.xcl)` or create a new one by copying an existing file and modify the address ranges. The actual addresses used must map to your target hardware.

Note that if your system does not allow programming of ROM memory, both code and data for the flash loader must be downloaded into RAM. This is achived by adjusting the linker file so that 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. Heap size shall be set to 0. The flash loader framework will use the memory between the segment BUF_START and BUF_END (as specified in the linker control file) for the read buffer. This guarantees that the read buffer gets all remaining memory available. The read buffer must be as large as possible to increase download performance, each transaction is costly and maximizing the number of bytes per transfer will increase performance. The framework will give an error if the resulting read buffer becomes smaller than 256 bytes as lower numbers will severely hurt performance.

The flash loader application must be built with the linker option **With I/O emulation modules** set on the linker options output dialog box. The resulting output file will have a `dxx` filename extension.

The flash loader can now be used for downloading an application into flash. In the Embedded Workbench, 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.

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

Debugging a flash loder

Debugging a flash loader can be done in the same way as an ordinary application. It should be noted 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 the 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 by the flash loader framework. Many of these functions may be of little interest for most flash loader drivers but are included for completeness.

The functions have been marked to indicate the intended usage. *Mandatory* means that a flash loader driver must implement this function. A function marked *optional* indicates that a flash loader driver may use this function depending on the needed functionality. *Framework only* is functions used by the flash loader framework and should not normally be used by the flash loader driver.

Function prototypes for all API functions are defined in the header file `\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 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);`

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

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

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 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 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.

`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 FlErrorExit();`

Usage: optional

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

FILE FUNCTIONS

`int FlFileOpen(char* name);`

Usage: framework

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.

`void FlFileClose(int fd);`

Usage: framework

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