IAR Embedded Workbench

Flash Loader Development Guide

UFLX-8

IAR SYSTEMS

EDITION NOTICE
Eighth edition: April 2021

Part number: UFLX-8

This guide applies to version 1.x of the IAR Flash Loader.

Internal reference: IJOA.

Contents

Introduction

- Introduction to flash loading
- The IAR Flash Loader

Introduction to flash loading

These topics are covered:

- Briefly about flash loading
- The Flash Loader
- Flash loading process overview
- Flash memory concepts

BRIEFLY ABOUT FLASH LOADING

Many development boards use flash memory as the primary code memory. When downloading and debugging a program, you cannot normally write directly from a debugger to flash memory, either for functional or performance reasons. Instead, you must use a dedicated program—a *flash loader*—that executes on the target system.

Note: Although a flash loader is mostly used for flash memory, it can also be used, for example, with external RAM or disk-like storage devices. In this guide, *flash memory* is the assumed type of memory.

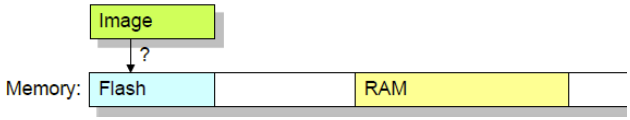
THE FLASH LOADER

A *flash loader* is usually a small dedicated program, which can program one or more flash memories. The flash loader is comprised of a set of functions, mainly for writing or erasing specified portions of flash memory. The debugger downloads this program into RAM, where it is linked to an address.

To run the program, the debugger sets the PC to one of the functions in the flash loader, writes data and directives for that function into a RAM buffer, and executes the function. When the function returns, the execution process will reach a breakpoint to indicate that the function has finished. The debugger can then proceed to make further calls in the flash loading process. In short, the debugger calls functions in the flash loader.

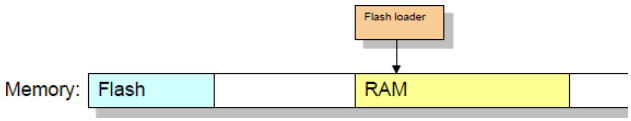
FLASH LOADING PROCESS OVERVIEW

This process overview illustrates how a flash loader downloads an application to flash memory. In the example, an application image needs to be downloaded, but C-SPY can only download data directly to RAM.

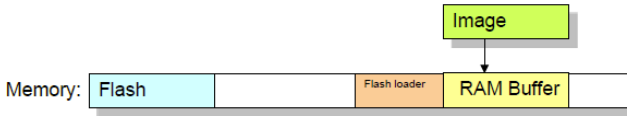


Downloading an application using a flash loader:

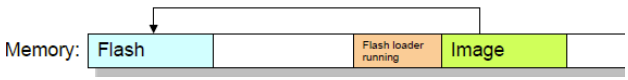
1. A dedicated flash loader is downloaded into RAM. Part of the RAM is reserved as a download buffer.



2. The image file to be downloaded is written to the RAM buffer.



3. The debugger starts the flash loader. The flash loader reads data from the RAM buffer and programs the flash memory.



4. The image file now resides in flash memory and can be started. The flash loader and the RAM buffer are no longer needed, and the full RAM is available to the application in the flash memory.

In practice, the process is more complicated—the RAM buffer is usually much smaller than the image file to be downloaded, and the flash programming is performed in several steps. See *The IAR Flash Loader process*, page 13.

FLASH MEMORY CONCEPTS

C-SPY uses these concepts to describe flash memory characteristics:

- **Page**—A page is the smallest writable unit of the flash memory. Many flash memories are restricted to the smallest writable unit, and cannot write less than, for example 128 or 256 bytes, in a single write operation. C-SPY will not request the flash loader write anything smaller than a page, and will use padding to fill out a page if necessary. Some flash memories have no such restrictions, and can specify a page size of 1 byte.
- **Block**—A block is the smallest erasable unit of the flash memory. For example, a flash memory with a 256-byte page size could still require that flash memory should be erased in 4-Kbyte chunks. The block size must always be a multiple of the page size. A flash memory can consist of several blocks of different sizes. It can also lack such restrictions, in which case the block size would be the same as the page size.
- **Base address**—The base address is the start address of the flash memory when it is written. Some flash memories are simply memory mapped to a fixed address range, and the base address is then the start of that range. Other flash memories are mapped to different addresses when being programmed, and when the application is executing later. The base address is then the address where these memories are mapped when being programmed, Yet other flash memories are not memory mapped at all, but work more like external disk-like devices. The base address is then simply the preferred address to be used for the start of the memory when it is being programmed.

From the C-SPY perspective, a flash memory starts at a given address and consists of a sequence of blocks—possibly of different sizes—each of which consists of a number of pages. The sequence can also contain gaps.

The IAR Flash Loader

The IAR Flash Loader is a dedicated flash loader program that you can use to perform flash loading for a specific board using IAR Embedded Workbench. The IAR Flash Loader has two main functions—`FlashWrite` and `FlashErase`—for writing and erasing specified portions of flash memory.

To perform flash loading using IAR Embedded Workbench, you must first implement the IAR Flash Loader, and then activate the program in the IDE:

- *Implementing the IAR Flash Loader*—you implement the IAR Flash Loader from framework source code, supplied with IAR Embedded Workbench, and device-specific source code that you provide. See *Implementing the IAR Flash Loader*, page 11.
- *Activating the IAR Flash Loader*—once you have built the IAR Flash Loader, you must activate the program in the IDE. See *Activating the IAR Flash Loader*, page 21.

To perform flash loading without RAM, see *Flash loading without RAM*, page 27.

IAR FLASH LOADER CONFIGURATION FILES

The IAR Flash Loader uses two configuration files:

- *Flash memory configuration file* (`.flash`) is an XML file that describes the flash memory elements to the debugger. These include, for example, the flash memory base address, the flash loader name, and flash memory characteristics, such as block and page size. The file also specifies which flash loader to use.

  For more information, see *The flash memory configuration file*, page 43.
- *Flash memory system configuration file* (`.board`) is an XML file that specifies the information needed to perform flash loading for a specific board. You can prepare this file in advance for various development boards, and create or modify the file in the IAR Embedded Workbench IDE.

  For more information, see *The flash memory system configuration file*, page 46.

Implementing the IAR Flash Loader

Implementing the IAR Flash Loader

The IAR Flash Loader is a dedicated flash loader program that you can use to perform flash loading for a specific board using IAR Embedded Workbench. You implement the IAR Flash Loader from this source code:

- **Framework source code**—supplied in the IAR Embedded Workbench installation directory under *target*`\src\flashloader`, or available for download as open source under an Apache license from `http://links.iar.com/cmsis/IAR_flashloader_framework_200.zip`. C-SPY uses labels and variables defined in the framework source code to interact with the flash loader. See *Framework source code*, page 12.
- **Device-specific source code**—a set of functions that you implement to build the IAR Flash Loader. Includes the `FlashWrite`, `FlashErase`, and `FlashInit` functions, and other optional functions. See *Device-specific source code functions*, page 12.

Note: When you implement the flash loader, consider these points:

- The flash loader must be linked to an address in the RAM.
- The flash loader cannot contain an entry point like a `main` function.

FRAMEWORK SOURCE CODE

The following files are part of the IAR Flash Loader framework source code:

Files	Description
`flash_config.h`	A template for your own configuration file. You should make a copy of the template and edit the copy according to the guidelines in the file.
`flash_loader_extra.h`	Additional framework declarations, rarely needed by your source code.
`flash_loader.h`	Framework declarations, for example, the C prototypes of your source code.
`flash_loader_asm.s`	Low-level, processor-specific framework source code. Note that the file might have a different extension depending on your microcontroller. This file should be part of your flash loader project. This file should be used as is and not modified.
`flash_loader.c`	Framework source code. This file should be part of your flash loader project. It should be used as is and not modified.

Table 1: Device-specific source code functions

Copies of the files and flash loader examples are located in the IAR Embedded Workbench installation directory under *`target`*`\src\flashloader`.

DEVICE-SPECIFIC SOURCE CODE FUNCTIONS

The following functions are used to implement the IAR Flash Loader:

Macro	Description
`FlashChecksum`	Optional. Enables checksum verification.
`FlashErase`	Erases one flash memory block.
`FlashInit`	The first function called in the flash loader and which can initialize the flash memory. Can also provide extra information to C-SPY before flash programming starts, and can override the properties specified in the flash memory configuration file.
`FlashSignoff`	Optional. Cleans up after flash loading.
`FlashWrite`	Writes or copies a number of bytes of data from the RAM buffer to the flash memory.

Table 2: Device-specific source code functions

For information about these functions, see *IAR Flash Loader functions*, page 39.

The IAR Flash Loader process

The two most important functions used by the IAR Flash Loader are `FlashWrite` and `FlashErase`:

- `FlashWrite`—writes or copies a number of bytes (always a multiple of the page size) of data from the RAM buffer to the flash memory.
- `FlashErase`—erases one flash memory block.

Using data from the image file, C-SPY repeatedly writes data to the RAM buffer and invokes the `FlashWrite` function in the flash loader, with these constraints:

- Writing is sequential, starting at the lower address
- The buffer will always contain a multiple of the page size
- The buffer is padded whenever the data does not naturally fill a page

Before the first page of any given block is written, the `FlashErase` function is invoked to erase that whole block.

For reference information on these functions, see *IAR Flash Loader functions*, page 39.

THE IAR FLASH LOADER PROCESS

This describes the IAR Flash Loader process in detail:

1 C-SPY reads a flash memory system configuration file (`.board`) and specifies one or more flash loading passes, one for each flash memory device on the board.

2 For each pass, a specific address range (or subset) of the original image file (application to be downloaded) is specified. The image file (ELF) is split into a separate image file for each pass. If there is only one pass, the original image file is used as is.

3 Each pass specifies a flash memory configuration file (`.flash`) which, among other things, designates a specific flash loader.

4 C-SPY downloads the flash loader of the current pass to RAM:

- The macro file of the current pass in C-SPY, specified in the `.flash` file, is loaded.
- If this is the first pass:
  - The debugger connects to the device. If the C-SPY I-jet driver is used, the `execUserProbeConnect` and `execUserCoreConnect` macro functions are called.
  - The device is reset according to the **Project>Options** settings. If the I-jet debug probe is used, the `execUserProbeReset` and `execUserCoreConnect` macro functions are called.

- The `execUserFlashInit` macro function is called.
- The flash loader executable is downloaded in RAM.
- The `execUserFlashPreset` macro function is called.
- A software reset is done.
- The `execUserFlashReset` macro function is called.

5 If the pass specifies an offset, all records from the image file are relocated accordingly.

6 If the label `FlashPreInitEntry` exists, C-SPY sets `PC` to it.

The execution is started and C-SPY regains control when the execution reaches a special breakpoint.

If the flash loader is relocatable, the `FlashPreInitEntry` function sets up the system, so that the debugger can find all relocated symbols.

7 C-SPY sets `PC` to `FlashInit`—or technically, to a label that will subsequently call `FlashInit`.

8 Parameters are written to the RAM buffer.

9 Execution is started, `FlashInit` is executed, and C-SPY regains control when execution reaches a special breakpoint. `FlashInit` can override some information from the `.flash` file, such as the page size and block layout.

10 C-SPY partitions the data from the image file into suitable pieces with respect to the flash memory page and block layout, and to the size of the RAM buffer.

If the `<aggregate>` tag value is `1`, C-SPY will try to use the RAM download buffer more efficiently by combining write operations to more than one block. This is a useful performance optimization if, and only if, block sizes are significantly smaller than the RAM buffer, so that at least two (or preferably more) blocks will fit in the download buffer. This element requires that the flash loader can program more than one block in a single operation.

11 If the block is being written to for the first time, the block must first be erased, go to step 11. If this is not the first time the block has been written to, go to step 13.

12 The RAM parameters are assigned the size of the block and its address.

13 C-SPY sets `PC` to `FlashErase` and starts execution. When the function is done, the breakpoint is reached.

14 C-SPY writes a piece of the image file to the RAM buffer (the image size that is downloaded to the RAM buffer is a multiple of the page size).

15 C-SPY sets `PC` to `FlashWrite` and starts the execution. When the function is done, the breakpoint is reached.

16 If there is more data, the procedure returns to step 10. If not:

- C-SPY sets `PC` to `FlashChecksum` and starts execution. When the function is done, the breakpoint is reached. This step is optional.
- C-SPY sets `PC` to `FlashSignoff` and starts execution. When the function is done, the breakpoint is reached. This step is optional.
- The `execUserFlashExit` macro function is called.
- The C-SPY macro file of the current pass is unloaded, see the first point under step 4.

17 If there are more passes, the process returns to step 3.

18 The debug information that corresponds to the final application is read. First, these steps occur:

- If **Project>Download>Download active application** or **Project>Download>Download file** are selected, the device is disconnected and the remaining points in step 17, and step 18 are skipped.
- The macro file(s) specified according to **Project>Options>Debugger>Setup>Setup macros** settings are loaded. The macro file(s) specified by the `DeviceMacros` parameter in the `.i79` file are also loaded.
- The device is reset according to **Project>Options>Debugger** settings.
- The `_ExecDevicePreload` and `execUserPreload` macro functions are called.
- The debug information that corresponds to the final application is read.
- The download is verified (if set in **Project>Options>Debugger>Download**).
- The `_ExecDevicePreReset` and `execUserPreReset` macro functions are called.
- A software reset is done.
- The `_ExecDeviceReset` and `execUserReset` macro functions are called.
- The `_ExecDeviceSetup` and `execUserSetup` macro functions are called.

19 C-SPY sets `PC` to the start address of the final application.

Information about the macro functions

For reference information about the macro functions, see the *C-SPY® Debugging Guide for Arm*. The `_ExecDevice...` macro functions work like the corresponding `execUser...` macro functions.

The new namespace `_ExecDevice` should be used in the device support scope. The `execUser` namespace should be used in the application scope. For each defined setup

macro function, C-SPY first calls the `_ExecDevice...` macro function and then the `execUser...` macro function, so the application macro can override any action in the corresponding device support macro.

The IAR Flash Loader configuration files

The IAR Flash Loader uses two configuration files:

- The flash memory configuration file
- The flash memory system configuration file

THE FLASH MEMORY CONFIGURATION FILE

The flash memory configuration file is an XML file (with the filename extension `.flash`) that describes relevant flash memory properties to the debugger. The file contains both mandatory and optional elements.

The mandatory elements include:

- `exe`—the path to the flash loader
- `flash_base`—the flash memory base address
- `page`—the flash memory page size
- `block`—the block layout of the flash memory

For example:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<flash_device>
  <exe>$TOOLKIT_DIR$\config\flash\P8_family\flash_p8.out</exe>
  <flash_base>0x20000</flash_base>
  <page>256</page>
  <block>2 0x100</block>
  <block>3 0x100</block>
</flash_device>
```

For reference information on the flash memory configuration file, see *The flash memory configuration file*, page 43.

THE FLASH MEMORY SYSTEM CONFIGURATION FILE

The *flash memory system configuration file* is an XML file (with the filename extension `.board`) that specifies the information needed to perform flash loading for a specific board using IAR Embedded Workbench.

A board can contain one or more flash memories, each with its own flash memory configuration (`.flash`) file, and during flash programming, each flash memory is programmed in a separate pass.

If the board has more than one type of flash memory that must be programmed separately in several passes, this file can contain references to more than one flash memory through flash memory configuration (`.flash`) files. In this case, the flash memory system configuration (`.board`) file also specifies image address ranges that belong to different flash memories.

If a board contains two flash memories, where one is used for a boot loader and the other for the application, only one of them will be relevant for any given project. In this case you need to have two different flash memory system configuration (`.board`) files—one for each kind of project.

You can prepare this file in advance for various development boards, and create or modify the file using the **Project>Options>Debugger>Download** dialog box in the IAR Embedded Workbench IDE, see *Activating the IAR Flash Loader*, page 21.

For example:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<flash_board>
  <pass>
    <loader>$TOOLKIT_DIR$\config\flash\flash_p8_2a.flash</loader>
    <range>CODE 0x20000 0x207ff</range>
    <abs_offset>0x10000</abs_offset>
  </pass>
  <ignore>CODE 0x22000 0x220ff</ignore>
  <pass>
    <loader>$TOOLKIT_DIR$\config\flash\flash_p8_2b.flash</loader>
    <range>CODE 0x20800 0x21000</range>
    <abs_offset>0x10000</abs_offset>
  </pass>
</flash_board>
```

For reference information on the flash memory system configuration file, see *The flash memory system configuration file*, page 46.

Overriding the flash memory configuration file

The `FlashInit` function is the first function called in the flash loader, and as such, can be used to provide extra information to C-SPY before flash programming starts. To provide extra information, you override the properties specified in the flash memory

configuration (.flash) file using a set of constants defined in the flash_loader_extra.h header file.

Internally, this functionality requires access to a structure variable, theFlashParams, defined in the framework, which is used for passing information between C-SPY and the flash loader.

The variable theFlashParams is declared in the header file, like this:

```
typedef struct {
  uint32_t base_ptr;
  uint32_t count;
  uint32_t offset_into_block;
  void *buffer;
  uint32_t block_size;
} FlashParamsHolder;

extern FlashParamsHolder theFlashParams;
```

The constants include:

Constant	Description
LAYOUT_OVERRIDE_BUFFER	Overrides the flash loader that will be executed
LAYOUT_OVERRIDE_BUFSIZE	Overrides the block layout
SET_BUFSIZE_OVERRIDE	Overrides the download buffer size
SET_PAGESIZE_OVERRIDE	Overrides the page size

Table 3: IAR Flash Loader constants

For reference information on these constants, see *Constants to override the flash memory configuration file*, page 48.

COMBINING OVERRIDING CONSTANTS

If required, you can combine these constants to override the properties specified in the flash memory configuration (.flash) file:

- LAYOUT_OVERRIDE_BUFFER
- SET_BUFSIZE_OVERRIDE
- SET_PAGESIZE_OVERRIDE

For example:

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size
{
  strcpy(LAYOUT_OVERRIDE_BUFFER, "2 0x100,7 0x200,7 0x1000");
  SET_PAGESIZE_OVERRIDE(128);    // New page size
  SET_BUFSIZE_OVERRIDE(0x1000);  // New buffer size
  return RESULT_OK | OVERRIDE_LAYOUT
        | OVERRIDE_PAGESIZE | OVERRIDE_BUFSIZE;
}
```

IAR Flash Loader source code example

The following example shows the source code for a complete flash loader—except the source code for the framework—but with a flash programming algorithm which simply copies bytes from the RAM buffer to the destination address:

```
#include "flash_loader.h"

uint32_t FlashInit(void *base_of_flash, uint32_t image_size,
                         uint32_t link_address, uint32_t flags)
{
  return RESULT_OK;
}

uint32_t FlashWrite(void *block_start,
                    uint32_t offset_into_block,
                    uint32_t count,
                    char const *buffer)
{
  char *to = (char*)block_start + offset_into_block;
  while (count--)
  {
     *to++ = *buffer++;
  }
  return RESULT_OK;
}
uint32_t FlashErase(void *block_start, uint32_t block_size)
{
  char *p = (char*)block_start;
  while (block_size--)
  {
     *p++ = 0;
  }
  return RESULT_OK;
}
```

The parameters to `FlashWrite` and `FlashErase`, in combination with the flash memory base address given in `FlashInit`, fully specify the addresses of the portions of the flash memory to be programmed. Thus, a given flash loader can be used for any number of different flash devices, with different total size, page size, or block layout, provided that they all use the same flash programming algorithm. The flash memory configuration file (`.flash`) is used for specifying such variations between flash memories.

For reference information on these functions, see *IAR Flash Loader functions*, page 39.

Using the IAR Flash Loader

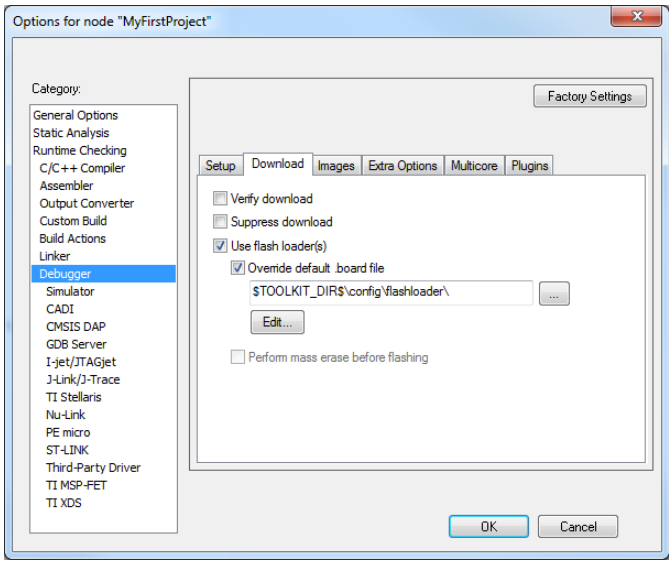
- Activating the IAR Flash Loader
- Generating debug information for use with the IAR Flash Loader
- Debugging the flash loader

Activating the IAR Flash Loader

When you have built the IAR Flash Loader, you must activate the use of the flash loader program in the IDE.

To activate the IAR Flash Loader:

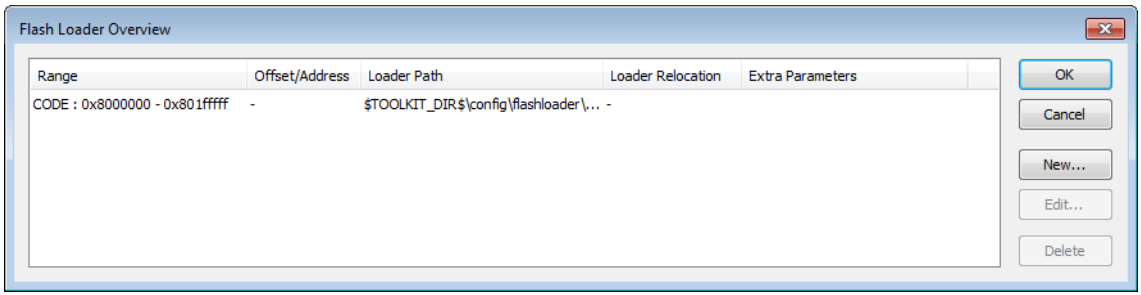
1 In the IAR Embedded Workbench IDE, choose **Project>Options>Debugger >Download**.



2 Select **Use flash loader(s)** and **Override default board file**, and specify a flash memory system configuration file (`.board`). Use the browse button to select a predefined file suitable for your system.

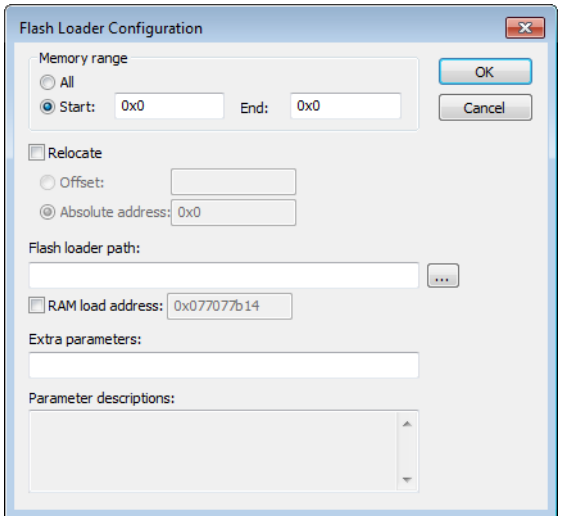
3 If no such file is available, you can create a new file or click **Edit** to modify an existing file. The **Flash Loader Overview** dialog box is displayed showing one row of information for each separate flash memory on the board, or for each flash loader pass.

Note: If you edit one of the predefined files located in the IAR Embedded Workbench installation directory, you will be prompted to save the modified file to a different directory.



Click **New** to define a new pass, **Edit** to modify an existing pass, or **Delete** to remove a pass from list.

4 If you click **New** or **Edit**, the **Flash Loader Configuration** dialog box is displayed. You can use this dialog box to configure the IAR Flash Loader.



Configure the flash memory system configuration file as required, and click **OK**.

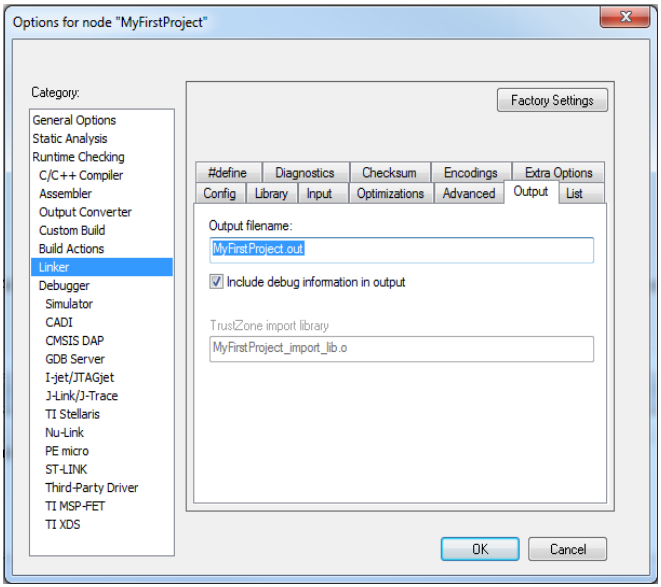
5 A flash memory system configuration file (`.board`) is created. See *The flash memory system configuration file*, page 46.

Generating debug information for use with the IAR Flash Loader

For the IAR Flash Loader to interact with the IAR C-SPY Debugger, the flash loader must be built with debug information generated from the ILINK linker. You specify the options to generate this debug information when you build the IAR Embedded Workbench project.

To generate debug information for use with the IAR Flash Loader:

1 In the IAR Embedded Workbench IDE, choose **Project>Options>Linker>Output**.



2 Select **Include debug information in output**.

Debugging the flash loader

The flash loader is not a standalone program with a `main` function, and is therefore not easy to debug. To enable debugging the flash loader, you should make a simple test harness when you develop it.

The test harness for debugging the flash loader should contain a `main` function which calls `FlashInit`, `FlashWrite`, and `FlashErase` with suitably prepared, or generated, data and parameters. The program should be linked to a RAM address, from which it can be debugged as a normal application until the basic flash programming code is correct.

When you use the flash loader in the flash loading process, you can view the process details in the generated log file.

GENERATED DEBUGGING LOG FILE

When you start a debug session which uses flash loaders, the debugger generates a log file named `flash0.trace` in the project directory (`$PROJ_DIR$`) where the active project file (`.ewp`) resides. This file is only generated if a file with that name already exists in that directory.

To enable trace output, create an empty file named `flash0.trace`, and trace output will be produced every time a debug session with flash loading is started, until the generated log file is removed.

If there are multiple flash loading passes, multiple trace files will be generated (`flash0.trace`, `flash1.trace`, etc.). However, you only need to create `flash0.trace` to enable tracing.

This is an example log file:

```
File generated Thu Nov 14 16:48:57 2019

Pass 1 of 1
Starting fragment-style flashloader pass.
FlashInitEntry is at 0x20000348
FlashWriteEntry is at 0x20000350
FlashEraseWriteEntry is at 0x20000358
FlashBreak is at 0x200000C0
FlashBufferStart is at 0x20000400
FlashBufferEnd is at 0x2001EF84
theFlashParams is at 0x2001EF84
FlashChecksumEntry not found
FlashSignoffEntry is at 0x20000360
page size is 8 (0x8)
filler is 0xff
buffer size is 125824 (0x1eb80) (0x1eb84 before rounding)
SimpleCode records (after offset):
  Record 0: @ 0x8000000[66592 (0x10420) bytes] 0x8000000 -
0x801041f [8 20 0]
```

```
Base of flash at 0x8000000
->init       : base @ 0x8000000, image size 0x10420
    Args: (argc = 1)
        --x32
  timing(init): 0.0000 (CPU) 0.0100 (elapsed)
Transaction list:
  Transaction @ 0x8000000 + 0x0 (0x10420 bytes) 5 packet(s).
    Will erase 5 block(s):
      0: 0x8000000 (0x4000 bytes)
      1: 0x8004000 (0x4000 bytes)
      2: 0x8008000 (0x4000 bytes)
      3: 0x800c000 (0x4000 bytes)
      4: 0x8010000 (0x10000 bytes)
->multi_erase: 5 blocks (0x28 bytes in buffer) [0 0 0]
  timing(erase): 0.0468 (CPU) 1.5400 (elapsed)
->write: @ 0x8000000 (0x10420 bytes, offset 0x0 into block @
0x8000000) [8 20 0]
  timing(write): 0.0312 (CPU) 0.2700 (elapsed)
->signoff
  timing(signoff): 0.0000 (CPU) 0.0100 (elapsed)
Duration: 0.23 (CPU)  3.53 (elapsed)
  of which on target: 0.0780 (CPU) 1.8300 (elapsed)
Flash loading pass finished
```

The log file lists these details:

- addresses of key functions in the flash loader, and basic properties of the flash memory and flash loader
- data records from the image to be downloaded
- sequence of write and erase operations, which list the start address, size, and at the end of the line, the three first bytes of the data for that operation
- optionally, at the end of the list file, checksum operations.

Flash loading without RAM

Flash loading without RAM

Some devices have extremely little RAM or do not allow access to RAM, so flash loading cannot be performed the usual way using downloadable flash loader programs. Instead, you must use C-SPY macros to perform the flash loading. You can leverage the existing flash loader framework and essentially implement the C functions of the downloadable flash loaders as equivalent C-SPY macro functions.

These topics are covered:

- Macro-based flash loading files
- Macro-based flash loading functions

MACRO-BASED FLASH LOADING FILES

To support macro-based flash loading, you need these components:

- A separate flash memory configuration file (`.flash`) that specifies macro-based flash loading
- A corresponding flash memory system configuration file (`.board`) file that references the appropriate `.flash` file
- An associated macro file (`.mac`), which normally only contains helper macros like `execUserFlashReset`, that must implement a small set of flash loading functions, see *Macro-based flash loading functions*, page 28.

Flash memory configuration file (.flash) for macro-based flash loading

The flash memory configuration file (`.flash`) that specifies macro-based flash loading is only slightly different than the usual file to specify flash loading.

The `exe` tag should contain the single word `MACRO` instead of a path to a flash loader executable. You can tentatively use `<online>1</online>` to support flash breakpoints.

Flash memory system configuration file (.board) for macro-based flash loading

This flash memory system configuration file (`.board`) must reference the corresponding flash memory configuration file (`.flash`) for macro-based flash loading.

You need separate `.board` files for standard flash loading and macro-based flash loading respectively.

Macro file (.mac) for macro-based flash loading

This `.mac` file must contain the flash loading functions, which are C-SPY macro language implementations of the corresponding functions in the on-target flash loader.

Note: When flash loading without RAM, you must specify the macro (`.mac`) file in the flash memory configuration file (`.flash`) using the element `macro`.

MACRO-BASED FLASH LOADING FUNCTIONS

When performing macro-based flash loading, the functions are used in a different manner than the standard functions.

FlashInit function for macro-based flash loading

The `FlashInit` function used for macro-based flash loading is a simplified version of the standard `FlashInit` function.

```
FlashInit(base_of_flash,
          image_size,
          link_address,
          flags,
          args)
{
   return 0;
}
```

All arguments are integers (addresses or flags), except the last one. The only flags used for the flags argument are:

- `kERASE_ONLY_FLAG (0x1)`—A special return value (except `kRESULT_OK(0)` or `kRESULT_ERROR(1)`), namely `kRESULT_ERASE_DONE(3)`, should be returned if a full erase was performed by the `FlashInit` function.
- `kMASS_ONLY (0x2)`—If possible, perform a mass erase before flash loading proceeds. The subsequent erase/write sequence will then proceed normally (unless an error is returned), and it is up to the flash loader to ignore subsequent erase requests if a mass erase has already been done.

The `args` argument is a single string containing all arguments (separated by tab characters, but this should not be relied upon). Use the built-in macros `__argCount` and `__getArg` to access the arguments, see *Built-in macros for macro-based flash loading*, page 30.

If the return value also has the bit `0x10000` set (`kRESULT_OVERRIDE_LAYOUT`), C-SPY expects an alternative layout in the macro variable `flashLayoutOverride`, as follows:

```
__var flashLayoutOverride;

FlashInit(base_of_flash,
          image_size,
          link_address,
          flags,
          args)
{
  ...
  flashLayoutOverride = "4 0x4000,1 0x10000,7 0x20000,4 0x4000,
     1 0x10000,7 0x20000";
  return result | 0x010000; // kRESULT_OVERRIDE_LAYOUT
}
```

FlashWrite function for macro-based flash loading

The `FlashWrite` function used for macro-based flash loading is similar to the standard `FlashWrite` function.

The first three arguments are integer addresses or counts. The fourth argument is a byte buffer (that is, a macro language native string) of size count. It can be indexed as a byte array, for example, `buffer[7]`.

```
FlashWrite(block_start,
           offset_into_block,
           count,
           buffer)
{
   return 0;
}
```

FlashErase function for macro-based flash loading

The `FlashErase` function used for macro-based flash loading is similar to the standard `FlashErase` function. Both arguments are integers, an address and a size respectively.

```
FlashErase(block_start,
           block_size)
{
   return 0;
}
```

FlashSignoff function for macro-based flash loading

The `FlashSignoff` function for macro-based flash loading is optional. The `FlashSignoff` function is called if the function is defined.

```
FlashSignoff()
{
   return 0;
}
```

Built-in macros for macro-based flash loading

This section describes built-in macros that are available for all C-SPY macro programming, including macro-based flash loading.

__argCount

Syntax

```
__argCount(string)
```

Description

Returns the number of arguments embedded in the given string (this is primarily for the last parameter of the `FlashInit` function.) This corresponds to the value of `argc` in standard flash loaders.

__bytes2Word16, __bytes2Word32

Syntax

```
__bytes2Word16(buffer, offset)
__bytes2Word32(buffer, offset)
```

Description

Extracts a 16-bit or 32-bit word from the given *`buffer`* (string) starting at the given byte offset into the buffer, using the proper byte order, and returns it as an integer. The offset does not need to be aligned to a multiple of the word size.

Example

```
__bytes2Word16("ABCDEF", 2)
```

__getArg

Syntax `__getArg(n, string)`

Description Returns the argument specified with the *n* value from the given string. This corresponds to `argv[n]` in a standard flash loader.

__makeString

Syntax `__makeString(count, char)`

Description Creates a new *buffer* (string) with *count* bytes.

Example `__makeString(7, 'b')`

__readMemoryBuffer

Syntax `__readMemoryBuffer(count, addr, zone)`

Description Reads *count* bytes from the specified address and returns them as a macro string.

Example `__readMemoryBuffer(4, 0X1000, "Memory")`

__writeMemoryBuffer

Syntax `__writeMemoryBuffer(buffer, count, addr, zone)`

Description Writes *count* bytes from the specified *buffer* (string) to the specified address.

Example `__writeMemoryBugger("ABCDEF", 4, 0X1000, "Memory")`

Flash loading with a relocatable flash loader

Relocatable flash loader flash loading files

There are devices with so many different memory configurations that having one flash loader per configuration is impractical. One way to solve this is to use a macro-based flash loader, but that can lead to bad flashing performance. A better solution can be to build a relocatable flash loader that can be loaded to any memory location, instead of having a dedicated flash loader for every memory configuration.

Note: Building a relocatable flash loader is only possible if the IAR Embedded Workbench product you are using has full support for position-independent code and data (ROPI and RWPI).

To use a relocatable flash loader, you need:

- A separate flash memory configuration file (`.flash`) that specifies a relocatable flash loader
- A corresponding flash memory system configuration file (`.board`) file that references the appropriate `.flash` file
- A relocatable flash loader file (`.out`)

RELOCATABLE FLASH MEMORY CONFIGURATION FILE

The difference between an ordinary flash memory configuration file (`.flash`) and one that specifies a relocatable flash loader, is that the file that specifies a relocatable loader should contain the tag `<relocatable_exe>1</relocatable_exe>`.

Note: Flash breakpoints are not supported. For example, `<online>1</online>` is not allowed.

RELOCATABLE FLASH MEMORY SYSTEM CONFIGURATION FILE

The flash memory system configuration file (`.board`) must reference the corresponding flash memory configuration file (`.flash`) for the relocatable flash loader.

Building a relocatable flash loader

Building a relocatable flash loader requires very small code changes, but some specific compiler and linker settings are needed.

COMPILER SETTINGS

To make the flash loader relocatable, you must enable ROPI and RWPI for the entire project, either in the **Project>Options** dialog box, or from the command line. This includes disabling runtime initialization of static C variables (also called *dynamic read/write initialization*).

LINKER SETTINGS

To be easily relocatable, the flash loader requires an `.icf` file that places all code and data together, starting at address `0x0`, with the label `FlashPreInitEntry` first.

This linker configuration stub can be used as a starting point:

```
build for ram;
define region RAM_region = mem:[from 0x0 to 0xFF00];
define block CSTACK     with alignment = 8, size = 0x400 { };
define block .noinit {};
place at address 0x0000'0000 { block SB {
                               first symbol FlashPreInitEntry,
                               ro,
                               rw,
                               block .noinit,
                               block CSTACK
                             } };
```

Reference information

- Reference information on IAR Flash Loader
- IAR Flash Loader functions
- IAR Flash Loader variables
- The flash memory configuration file
- The flash memory system configuration file
- Constants to override the flash memory configuration file

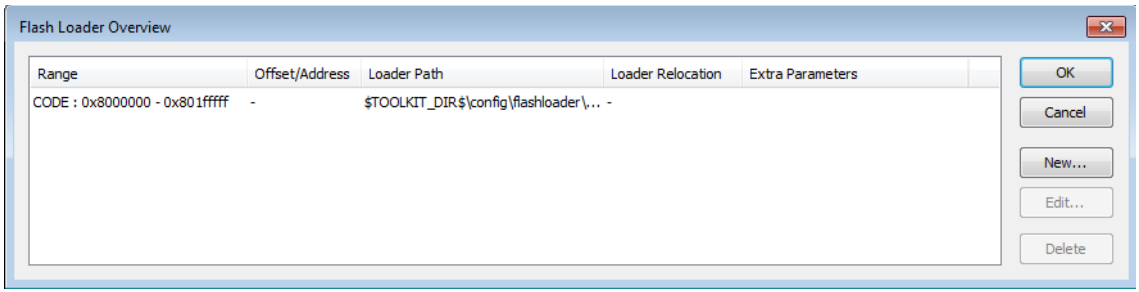
Reference information on IAR Flash Loader

Reference information about:

- *Flash Loader Overview dialog box*, page 35
- *Flash Loader Configuration dialog box*, page 37

Flash Loader Overview dialog box

The **Flash Loader Overview** dialog box is available from the **Project>Options>Debugger>Download** page.



This dialog box lists all defined flash loaders. If you have selected a device on the **Project>Options>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.

Requirements

Available for supported hardware debugger systems.

Display area

Each row in the display area shows how you have set up one flash loader for flashing a specific part of memory:

Range

The part of your application to be programmed by the selected flash loader.

Offset/Address

The start of the memory where your application will be flashed. If the address is preceded by an `A`, the address is absolute. Otherwise, it is a relative offset to the start of the memory.

Loader Path

The path to the flash loader `*.flash` file to be used (`*.out` for old-style flash loaders).

Loader Relocation

For relocatable flash loaders, this is the start of the target RAM memory where the flash loader will be downloaded.

Extra Parameters

List of extra parameters that will be passed to the flash loader.

Click on the column headers to sort the list by range, offset/address, etc.

Function buttons

These function buttons are available:

OK

The selected flash loader(s) will be used for downloading your application to memory.

Cancel

Standard cancel.

New

Displays a dialog box where you can specify what flash loader to use, see *Flash Loader Configuration dialog box*, page 37.

Edit

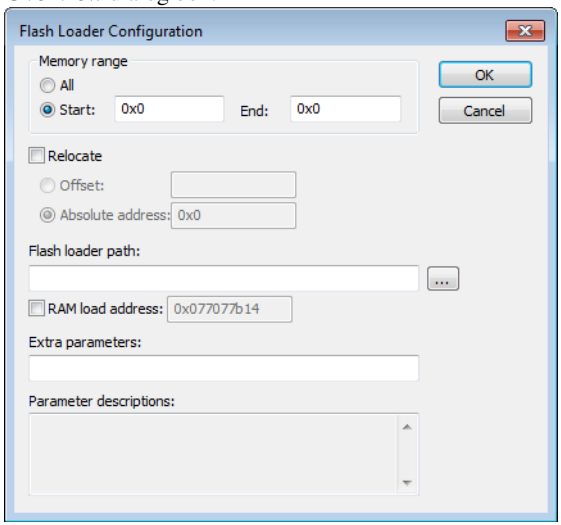
Displays a dialog box where you can modify the settings for the selected flash loader, see *Flash Loader Configuration dialog box*, page 37.

Delete

Deletes the selected flash loader configuration.

Flash Loader Configuration dialog box

The **Flash Loader Configuration** dialog box is available from the **Flash Loader Overview** dialog box.



Use the **Flash Loader Configuration** dialog box to configure the download to suit your board. A copy of the default `board` file will be created in your project directory.

Requirements

Available for supported hardware debugger systems.

Memory range

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

Specify the start and the end of the memory area for which part of the application will be downloaded.

Relocate

Overrides the default flash base address, in other words, relocates the location of the application in memory. This means that you can flash your application to a different location from where it was linked. Choose between:

Offset

A numeric value for a relative offset. This offset will be added to the addresses in the application file.

Absolute address

A numeric value for an absolute base address where the application will be flashed. The lowest address in the application will be placed on this address. Note that you can only use one flash loader for your application when you specify an absolute address.

You can use these numeric formats:

- `123456`, decimal numbers
- `0x123456`, hexadecimal numbers
- `0123456`, octal numbers

The default base address used for writing the first byte—the lowest address—to flash is specified in the linker configuration 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.

Flash loader path

Use the text box to specify the path to the flash loader file (`*.flash`) to be used by your board configuration.

RAM load address

If the flash loader is relocatable, this option overrides the default address in the target RAM memory that flash loader is downloaded to, in other words, relocates the flash loader. Use the text box to specify the address.

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 available flash loader options, see the **Parameter descriptions** field.

Parameter descriptions

Displays a description of the extra parameters specified in the **Extra parameters** text box.

IAR Flash Loader functions

These functions are used with IAR Flash Loader:

Macro	Description
`FlashChecksum`	Optional. Enables checksum verification.
`FlashErase`	Erases one flash memory block.
`FlashInit`	The first function called in the flash loader and which can initialize the flash memory. Can also provide extra information to C-SPY before flash programming starts, and can override the properties specified in the flash memory configuration file.
`FlashSignoff`	Optional. Cleans up after flash loading.
`FlashWrite`	Writes or copies a number of bytes of data from the RAM buffer to the flash memory.

Table 4: Summary of IAR Flash Loader functions

FlashChecksum

Syntax

```
OPTIONAL_CHECKSUM
uint32_t FlashChecksum(void const *block_start, uint32_t
block_size)
```

Parameters

`block_start`	Points to the first byte of the block to erase.
`block_size`	Specifies the size, in bytes, of the block to erase.

Description

This is an optional function. You implement this function to enable checksum verification of the downloaded flash memory content. You implement it with a helper function from the framework, for example, `Crc16`.

Note: The `OPTIONAL_CHECKSUM` macro is needed to make sure that this optional function and its framework wrapper are both included in the linked flash loader.

Return value

Either `RESULT_OK` or `RESULT_ERROR`.

Example

```
OPTIONAL_CHECKSUM
uint32_t FlashChecksum(void const *begin, uint32_t count
{
  return Crc16((uint8_t const *)begin, count);
}
```

FlashErase

Syntax

```
uint32_t FlashErase(void *block_start, uint32_t block_size)
```

Parameters

block_start	Points to the first byte of the block to erase.
block_size	Specifies the size, in bytes, of the block to erase.

Description

Erases one flash memory block.

Return value

Either `RESULT_OK` or `RESULT_ERROR`.

FlashInit

Syntax

```
#if USE_ARGC_ARGV
  uint32_t FlashInit(void *base_of_flash, uint32_t image_size,
                     uint32_t link_address, uint32_t flags,
                     int argc, char const *argv[]);
#else
  uint32_t FlashInit(void *base_of_flash, uint32_t image_size,
                     uint32_t link_address, uint32_t flags;
#endif;
```

Parameters

base_of_flash	Points to the first byte of the flash memory.
image_size	Specifies the size of the image, in bytes, to be written to flash memory.
link_address	Specifies the original link address of the first byte of the image, before any offsets, or if there are multiple passes, the first byte of the subset of the image used for this pass. Not all flash loaders need this parameter.

flags

Specifies optional flags. These flag values are defined:

FLAG_ERASE_ONLY—When set (when the expression, flags & FLAG_ERASE_ONLY, is non-zero), the flash loader has been invoked with the sole purpose of erasing the whole flash memory. If the flash memory supports a one-step global erase function, it can be invoked directly from FlashInit, which should return the constant RESULT_ERASE_DONE. Otherwise, C-SPY will continue and invoke the FlashErase function for each block.

FLAG_MASS_ERASE—When set, the FlashInit function is expected to perform a mass erase, if possible. Unless the function returns an error, the flash loader process will continue with the normal sequence of write/erase invocations.

Note that the sequence will be the same regardless of whether mass erase has been requested or not, or performed or not. If FlashInit has done a mass erase, subsequent erase operations can be skipped by the flash loader. If FlashInit has not done a mass erase, subsequent erase operations should be performed normally by the flash loader.

Description

The FlashInit function is the first function called in the flash loader, and can initialize the flash memory. This function can provide extra information to C-SPY before flash programming starts, such as unlocking the flash memory, or can override the properties specified in the flash memory configuration (.flash) file using a set of macros defined in the flash_loader_extra.h header file. See *Overriding the flash memory configuration file*, page 17.

There are two different prototypes for FlashInit, determined by the value of the pre-processor macro USE_ARGC_ARGV, which you specify in flash_config.h. If you need the argv/argc flexibility, you can specify arguments in the flash memory configuration (.flash) file, or in the **Project>Options>Debugger>Download** dialog box in the IAR Embedded Workbench IDE.

Return value

Either RESULT_OK or RESULT_ERROR, but also other return values can be used, see *Constants to override the flash memory configuration file*, page 48.

FlashSignoff

Syntax

```
OPTIONAL_SIGNOFF
uint32_t FlashSignoff()
```

Description

This is an optional function. You implement the function to perform a clean-up after flash loading has finished. The function is called after the last call to `FlashWrite` (or after `FlashChecksum`, if it exists).

Note: The `OPTIONAL_SIGNOFF` macro is needed to make sure that this optional function and its framework wrapper are both included in the linked flash loader.

Return value

Either `RESULT_OK` or `RESULT_ERROR`.

Example

```
OPTIONAL_SIGNOFF
uint32_t FlashSignoff()
{
  return RESULT_OK;
}
```

FlashWrite

Syntax

```
uint32_t FlashWrite(void *block_start,
                    uint32_t offset_into_block,
                    uint32_t count,
                    char const *buffer)
```

Parameters

block_start	Points to the first byte of the block into which this operation writes.
offset_into_block	Specifies how far into the current block that this write operation shall start. The absolute address of the first byte to write is *block_start* + *offset_into_block*.
count	Specifies the number of bytes to write.
buffer	A pointer to the buffer that contains the bytes to write.

Description

Writes or copies a number of bytes (always a multiple of the page size) of data from the RAM buffer to the flash memory.

Return value

Either `RESULT_OK` or `RESULT_ERROR`.

IAR Flash Loader variables

This variable is used by the IAR Flash Loader.

frameworkVersion

Syntax `frameworkVersion`

Description A variable in `flash_loader.c` used by the debugger to adapt to the framework version used by the flash loader.

Note: The framework version should never be changed by an individual flash loader. It is controlled by the framework code.

The flash memory configuration file

The flash memory configuration file is an XML file (with the filename extension `.flash`) which describes relevant flash memory properties to the debugger.

You can use constants to override the properties specified in the flash memory configuration file, see *Constants to override the flash memory configuration file*, page 48.

FILE CONTENTS

The flash memory configuration file contains the following mandatory and optional elements:

Mandatory elements

`block` Specifies the block layout of the flash memory using one or more of this element, in order. Each element contains a decimal count value followed by a block size in hexadecimal form. The element sequence should fully specify the sequence of blocks for the flash memory. In the file example, the flash device contains two blocks of size 0x100, followed by three blocks of size 0x200, for a total size of 0x800.

`exe` Specifies the path to the flash loader. The path can contain an argument variable, such as `$TOOLKIT_DIR$`.

`flash_base` Specifies the base address of the flash memory.

`page`	Specifies the flash memory page size.

Optional elements

`aggregate`	If this element contains `1`, C-SPY will try to use the RAM download buffer more efficiently by combining write operations to more than one block. This is a useful performance optimization if, and only if, block sizes are significantly smaller than the RAM buffer, so that at least two (or preferably more) blocks will fit in the download buffer. This element requires that the flash loader can program more than one block in a single operation.
`args`	Passes arguments to the flash loader, in the form `argc/argv` to the `FlashInit` function. The parameters should be separated by the *newline* character.
`args_doc`	Specifies descriptions of the parameters accepted by the flash loader `FlashInit` function, that is arguments listed as `args`. The descriptions are displayed in the **Flash Loader Configuration** dialog box in IAR Embedded Workbench. This element can contain multiple lines of text, separated by the *newline* character.
`checks`	If this element contains `0`, the error return values from the flash loader functions, such as `FlashWrite`, are not checked. Disabling these checks will slightly improve performance. Note that this element should only be used when these checks are not needed.
`filler`	Specifies a decimal number which is the byte value to use when padding write operations to page boundaries. The default value is `255`(`0xFF`).
`gap`	The specified block sequence can also contain one or more `gap` elements intermixed with `block` elements. Each gap element contains a gap size in hexadecimal form. The gap size specifies an area within the extent of the flash memory that does *not* contain flash memory. C-SPY will issue an error for any data in the image file that would be placed in a gap.

`macro`	Specifies the path to a C-SPY macro file which will be loaded when the flash loader is downloaded. These C-SPY macro functions will be called automatically if they are defined in this macro file: ● `execUserFlashInit` — called immediately before loading the flash loader ● `execUserFlashReset` — called immediately after the reset that follows the loading of the flash loader. ● `execUserFlashExit` — called immediately after flash loading has finished, but before the flash loader is unloaded.
`relocatable_exe`	Specifies that the flash loader is relocatable. This element is mutually exclusive with the `online` element.
`online`	Specifies that the flash loader supports flash breakpoints. This element is mutually exclusive with the `relocatable_exe` element.

FLASH LOADING MICROCONTROLLER VARIANTS

A microcontroller can have many variants, often with the same type of flash memory, but with different sizes and addresses, and possibly block layouts. For such a scenario, you might need several flash memory configuration (`.flash`) files, but only one flash loader.

The following table describes variants of the hypothetical *P8* processor family:

Variant	Flash size (Kbytes)	Flash base address	Flash block layout	Configuration file
P8_1	1	0x10000	4 * 0x100	flash_p8_1.flash
P8_2a	2	0x10000	8 * 0x100	flash_p8_2a.flash
P8_2b	2	0x10000	4 * 0x200	flash_p8_2b.flash
P8_4a	4	0x10000	8 * 0x100 4 * 0x200	flash_p8_4a.flash
P8_4b	4	0x20000	16 * 0x100	flash_p8_bb.flash

Table 5: Variants of the hypothetical P8 processor family

Because each of the processor variants has a flash memory of the same type, there are five different flash memory configuration (`.flash`) files, with each file specifying the same flash loader, requiring the same flash programming algorithm.

The flash memory system configuration file

The flash memory system configuration file is an XML file (with the filename extension `.board`) that specifies the properties of a development board with respect to flash memory.

FILE CONTENTS

At the highest level, the flash memory system configuration file contains one or more of the following elements:.

`ignore` — Specifies a subset of the original image file which should not be subject to flash loading. This element consists of a segment type (usually `CODE`), followed by the address of the first and last byte of the range in hexadecimal format. The element can be used, for example, when parts of the original image are to be downloaded to RAM, or when parts are already present in ROM. The element can be repeated to specify several ranges. This is useful to prevent warning messages about parts of the original image falling outside any of the flash memories.

`pass` — Specifies a single flash programming pass, for programming a single flash memory. Many flash memory system configuration files contain only one `pass` element.

Each `pass` element in the flash memory system configuration file consists of additional elements—mandatory and optional, see below.

Mandatory elements

Each `pass` element in the flash memory system configuration file contains the following element:

`loader` — Specifies the path to the flash memory configuration (`.flash`) file. The path can contain an argument variable, such as `$TOOLKIT_DIR$`.

Optional elements

Each `pass` element in the flash memory system configuration file can contain the following optional elements:

`abs_offset`	This element is used for writing the image to flash memory at an address different from the address when the image was placed by the linker. For example, if the flash memory is mapped to memory at an address when it is programmed, and then later remapped to another address when executed, you must use an appropriate offset to compensate when programming the flash memory. This element specifies an absolute address where the first byte of the image file should be placed.
`args`	Passes arguments to the flash loader, in the form `argc/argv` to the `FlashInit` function. The parameters should be separated by the *newline* character. The parameters are appended to any parameters specified in the flash memory configuration (`.flash`) file. If the flash loader handles parameters correctly, these parameters can override the ones specified in the flash memory configuration file.
`flash_base`	Specifies the base address of the flash memory when it is written to. If this element is included, it overrides the corresponding element in the flash memory configuration (`.flash`) file.
`range`	Specifies the subset of the original image file, which should be programmed in the flash memory. The contents of this element is a segment type (usually `CODE`), followed by the address of the first and last byte of the range in hexadecimal format. If there is only one flash memory, the default range is the range of the whole image file. Note that if there is more than one pass element, the range element is mandatory for defining how C-SPY is to partition the image.

`rel_offset`

This element is similar to `abs_offset`, but specifies a relative offset with which each record in the image file should be displaced before writing to flash. The offset can be either a positive or a negative number.

Note that the `abs_offset` and `rel_offset` elements are mutually exclusive, and cannot both be used in the same `pass` element.

Constants to override the flash memory configuration file

You can override the properties specified in the flash memory configuration (`.flash`) file using a set of constants in the `FlashInit` function. These constants are defined in the `flash_loader_extra.h` header file.

The constants are:

Constant	Description
`LAYOUT_OVERRIDE_BUFFER`	Overrides the flash loader that will be executed
`LAYOUT_OVERRIDE_BUFSIZE`	Overrides the block layout
`SET_BUFSIZE_OVERRIDE`	Overrides the download buffer size
`SET_PAGESIZE_OVERRIDE`	Overrides the page size

Table 6: Summary of IAR Flash Loader constants

For more information about using these constants, see *Overriding the flash memory configuration file*, page 17.

LAYOUT_OVERRIDE_BUFFER

Syntax

```
LAYOUT_OVERRIDE_BUFFER
```

Description

This constant can be used to override the flash loader specified in the flash memory configuration file (`.flash`) and specify another flash loader to execute.

You can use this constant, for example, when the flash loader detects that the flash memory does not match the capabilities of the flash loader, which typically occurs when the wrong flash loader has started. This is normally the result of misconfiguration, which many flash loaders cannot check.

If the flash loader can detect the flash memory type at runtime, the flash loader can report the flash memory name to C-SPY and prompt C-SPY to use another flash loader. You do this by putting a device identifier in the buffer and returning the special return value `RESULT_OVERRIDE_DEVICE`.

Note: The replacement flash loader is specified indirectly, as a flash memory identifier. C-SPY reads this identifier and uses the identifier as the key in a table lookup to locate another flash loader. The table is constructed like this:

- C-SPY finds all files with the filename extension `flashdict` in the `$TOOLKIT_DIR$\config\flashloader` directory (and all subdirectories).
- Each such file can contribute a portion of the table.

The file should look like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<loaders>
  <loader>
    <key>P8_16c</key>
    <path>$TOOLKIT_DIR$\config\flashloader\P8\f_p8_16c.flash
      </path>
  </loader>
  <loader>
    <key>P8_16d</key>
    <path>$TOOLKIT_DIR$\config\flashloader\P8\f_p8_16d.flash
      </path>
  </loader>
</loaders>
```

If the key is found anywhere in the table, the newly specified flash memory configuration file is used instead.

Return value

Either `RESULT_OK` or `RESULT_ERROR`.

Example

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size
{
  if (‘unexpected flash device was found’)
  {
    strcpy(LAYOUT_OVERRIDE_BUFFER, ”P8_16c”);
    return RESULT_OVERRIDE_DEVICE;
  }
}
```

LAYOUT_OVERRIDE_BUFSIZE

Syntax

```
LAYOUT_OVERRIDE_BUFSIZE
```

Description

This constant can be used to override the block layout of the flash memory specified in the flash memory configuration file (`.flash`). The block layout is normally specified

using the `block` and `gap` tags in the flash memory configuration file, this constant lets the flash loader determine the block layout by querying the flash memory itself.

If the flash loader wants to specify a layout, the flash loader should put the layout description in the flash download buffer and add the constant `OVERRIDE_LAYOUT` to the return value of `FlashInit`. You use this constant to add a pointer to the download buffer.

The syntax is the same as in the flash memory configuration (`.flash`) file (a decimal block count followed by a hexadecimal block size), except that blocks are separated by comma.

To specify a gap, use a block count of 0. For example, `"0 0x1000"` specifies a gap of 0x1000 bytes.

Return value

Either `RESULT_OK` or `RESULT_ERROR`.

Example

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size
{
  strcpy(LAYOUT_OVERRIDE_BUFSIZE, "2 0x100,7 0x200,7 0x1000");
  return RESULT_OK | OVERRIDE_LAYOUT;
}
```

SET_BUFSIZE_OVERRIDE

Syntax

```
SET_BUFSIZE_OVERRIDE
```

Description

This constant can be used to override the buffer size specified in the flash memory configuration file (`.flash`) and to set the `OVERRIDE_BUFSIZE` bit in the return value from the `FlashInit` function.

The download buffer size is normally determined by the addresses of two labels, `FlashBufferStart` and `FlashBufferEnd`, which get their addresses at link time. To use the same flash loader for multiple devices which only differ in RAM size, the flash loader can override the buffer size (if the flash loader can determine the actual amount of RAM available).

Note: Do not decrease the buffer size.

Return value

Either `RESULT_OK` or `RESULT_ERROR`.

Example

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size
{
  SET_BUFSIZE_OVERRIDE(0x1000);// New buffer size
  return RESULT_OK | OVERRIDE_BUFSIZE;
}
```

SET_PAGESIZE_OVERRIDE

Syntax

`SET_PAGESIZE_OVERRIDE`

Description

This constant can be used to override the page size specified in the flash memory configuration file (`.flash`) and to set the bit `OVERRIDE_PAGESIZE` in the return value from the `FlashInit` function.

Return value

Either `RESULT_OK` or `RESULT_ERROR`.

Example

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size
{
  SET_PAGESIZE_OVERRIDE(128);// New page size
  return RESULT_OK | OVERRIDE_PAGESIZE;
}
```