IAR Embedded Workbench flash loader

User Guide

The purpose of this guide is to give an understanding of the flash loader mechanism 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. 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

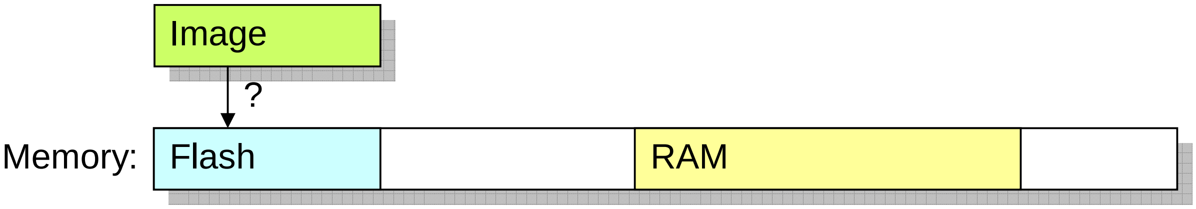
Background

Many development boards use flash memory as the primary code memory. Flash memory can normally not be written directly from C-SPY when a program is to be downloaded and debugged, but must instead be "burned or programmed" by a program executing on the target system. This document describes the mechanisms employed by the C-SPY debugger.

Note: This mechanism is useful for any sort of memory that cannot be written directly by the debugger, but instead has to be written by a program executing on the target system. Although it is mostly used for flash memory, it can for example also be used for various forms of external RAM or even disk-like storage devices. However, for the rest of this document, we will simply assume that we are dealing with flash memory.

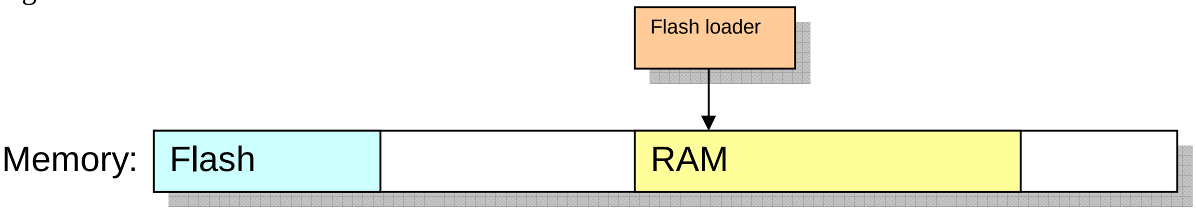
The process in brief

The problem is to download a program image to flash memory, when C-SPY can only directly download data to RAM.



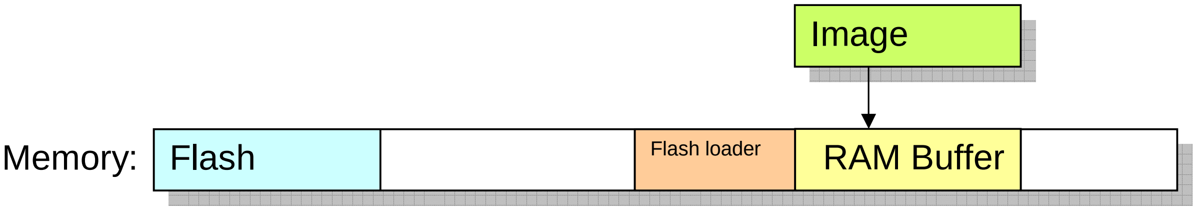
How can we download an image into flash memory?

The solution consists of a few steps, the first of which is to download a dedicated flash loader program into RAM.



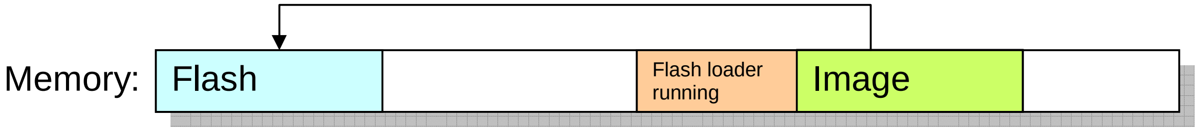
The flash loader program is downloaded into RAM

Part of RAM is also reserved for a download buffer. Next, the image is written to the RAM buffer.



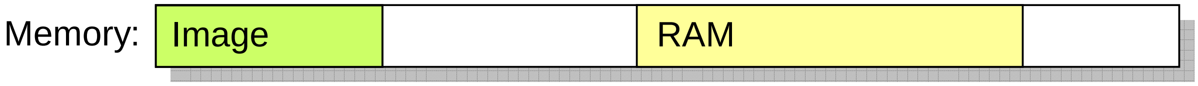
The program image is written to the RAM buffer

Then the flash loader is started by C-SPY. The flash loader reads data from the RAM buffer and programs flash memory.



The flash loader program writes data to flash memory

Now the image resides in flash memory and can be started. The flash loader program and the RAM buffer are no longer needed, so RAM is fully available to the program in the flash memory.



Flash loading is complete

In practice, the process is a bit more complicated. For example, the RAM buffer is usually much smaller than the image to be downloaded, so the flash programming must be performed in several steps.

Technical overview

To implement flash loading, there are two major components involved: The flash loader program and the flash memory configuration files.

The flash loader program

A flash loader program is a usually rather small program which knows how to program a certain flash memory device, or family of devices. It consists of a small set of functions, mainly for erasing or writing designated portions of the flash memory. C-SPY downloads this program into RAM (it must be linked to an address in RAM). To run the program, C-SPY positions the PC at one of the functions in the flash loader, writes data and directives for that function into a RAM buffer, and starts execution. When the function returns, execution will hit a breakpoint. C-SPY will then know that the function has finished and can proceed to make further calls to continue the flash loading process. In essence, C-SPY “calls” functions in the flash loader.

The flash memory configuration files

The flash memory device configuration file is an XML file (file extension `.flash`) which describes for C-SPY all relevant properties of a certain flash device, such as the base address of the flash memory, and details such as block and page sizes (see below), and also specifies which flash loader program to use.

The flash memory system specification file is an XML file (file extension `.board`) which describes for C-SPY the flash loading properties of the complete development board. This can sometimes contain references to more than one flash device (through the appropriate `.flash` files), if the board has more than one type of flash memory that needs to be programmed separately, in several *passes*. In this case, the file also specifies specific address ranges of the image which belongs to different flash devices.

One `.board` file fully specifies the information needed to perform flash loading for a specific board. Such a file can be prepared in advance for various development boards, and can also be created or modified in the Project Options dialog of the IAR Embedded Workbench IDE.

Note that a `.board` file doesn't necessarily describe *everything* about the flash memory of a certain board, but rather everything that is needed for using the board with a given IAR Embedded Workbench project. For example, if a board contains two flash devices, where one is used only for a boot loader and the other for the application, only one of them would be relevant for any given project. There would then be two different `.board` files, one for each kind of project.

Flash memory concepts

To accommodate a large range of different flash devices, C-SPY uses a few concepts which detail the characteristics of flash devices.

Page A page is the smallest writable unit of the flash memory. Many flash devices cannot write less than e.g. 128 or 256 bytes in a single write operation. C-SPY will never request the flash loader to write anything smaller than a page, and uses padding if necessary to fill out a page. Of course, some flash devices have no such restrictions, and can specify a page size of 1.

Block A block is the smallest erasable unit of the flash memory. For example, a flash device with a 256-byte page size could still require that flash memory be erased in 4kbyte chunks. Block size must always be a multiple of the page size. A flash device can consist of several blocks of different sizes. It can also have no such restrictions at all, in which case the block size would be the same as the page size.

Base Address This is the address of the start of the flash memory device, *when it is written*. Some flash memories are simply memory mapped into a fixed address range, and the Base Address is then the start of that range. Other flash memories are mapped into different addresses when being programmed and when the application is later executing. The Base Address is then the address where they 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 perspective of C-SPY then, a flash memory device 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 flash programming process in somewhat greater detail

The two most important functions in the flash loader are called `FlashWrite` and `FlashErase`. The former writes, or copies, a number of bytes (always a whole number of **pages**) of data from the RAM buffer to flash memory. The latter erases one flash memory **block**. Using data from the image file, C-SPY repeatedly writes data to the RAM buffer and invokes the `FlashWrite` operation in the flash loader, with the following "constraints":

1. Writing is sequential, starting at the lowest address.
2. The buffer will always contain an integral number of pages.
3. The buffer is padded whenever the data does not naturally fill a page.

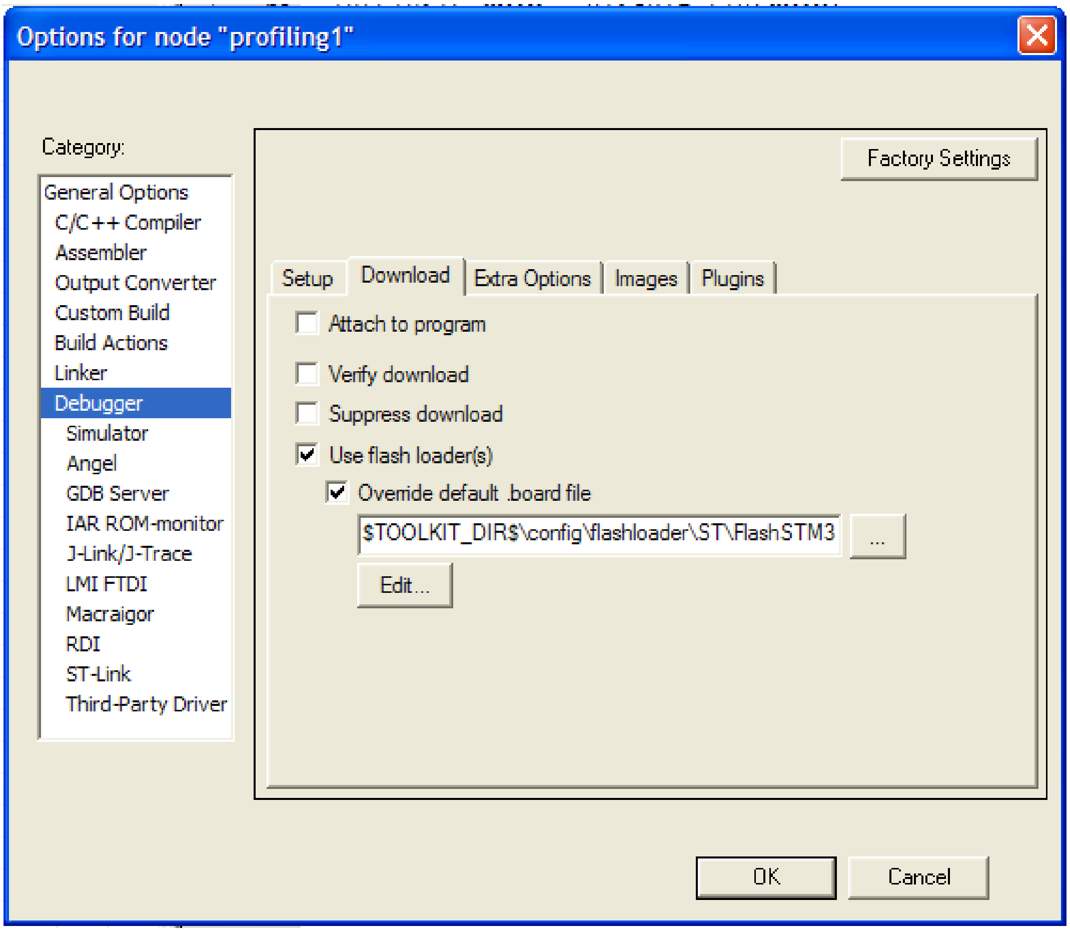
4. Before the first page of any given block is written, the `FlashErase` function is invoked to erase that whole block.

In more detail, this is what happens:

1. The program to be downloaded into flash memory exists as an image file. A flash memory system specification file (`.board`) is read by C-SPY and specifies one or more flash loading *passes*, one for each flash memory device on the board.
2. Each pass specifies a specific address range, or subset, of the original image file. The image file is split accordingly into a separate image file for each pass. If there is only one pass, the original image file is used in its entirety.
3. Each pass specifies a flash device (a `.flash` file) which, among other things, designates a specific flash loader program.
4. C-SPY downloads the flash loader program of the current pass into RAM
5. If the pass specifies an offset, all records from the image file are relocated accordingly.
6. C-SPY sets PC to `FlashInit` (or technically to a label that will subsequently call `FlashInit`)
7. Parameters and data are written to the RAM buffer.
8. Execution is started, `FlashInit` is run, and C-SPY regains control when execution then hits a special breakpoint. `FlashInit` has the opportunity to override some information from the `.flash` file, such as the page size and block layout.
9. C-SPY partitions the data from the image file into suitable pieces with respect to the page and block layout of the flash memory, and to the size of the RAM buffer.
10. If we are about to make the first write operation to a certain block, it must first be erased. Otherwise proceed to step 13.
11. Write block address and size into the RAM parameters.
12. Set PC to `FlashErase` and start execution. Wait until the breakpoint is hit.
13. Write some of the data to the RAM buffer.
14. Set PC to `FlashWrite` and start execution. Wait until the breakpoint is hit.
15. If there is more data, go back to step 10.
16. If there are more passes, go back to step 3.
17. Read the debug information corresponding to the final program.
18. Set PC to the start address of the final program.
19. Optionally run to main, etc

Using flash loaders

On the Download page in the Debugger section of the “Project Options…” dialog, flash loading is activated by specifying a flash memory system specification file (`.board`).



□ Attach to program

□ Verify download

□ Suppress download

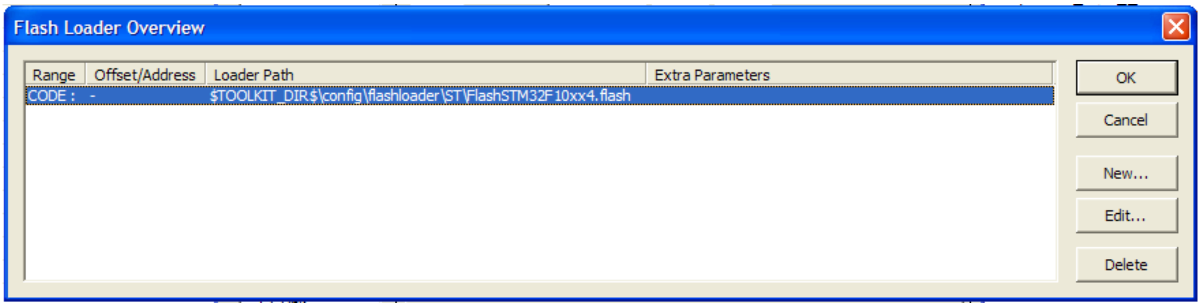
☑ Use flash loader(s)

☑ Override default .board file

$TOOLKIT_DIR$\config\flashloader\ST\FlashSTM3

Enabling flash loading on the Download pane

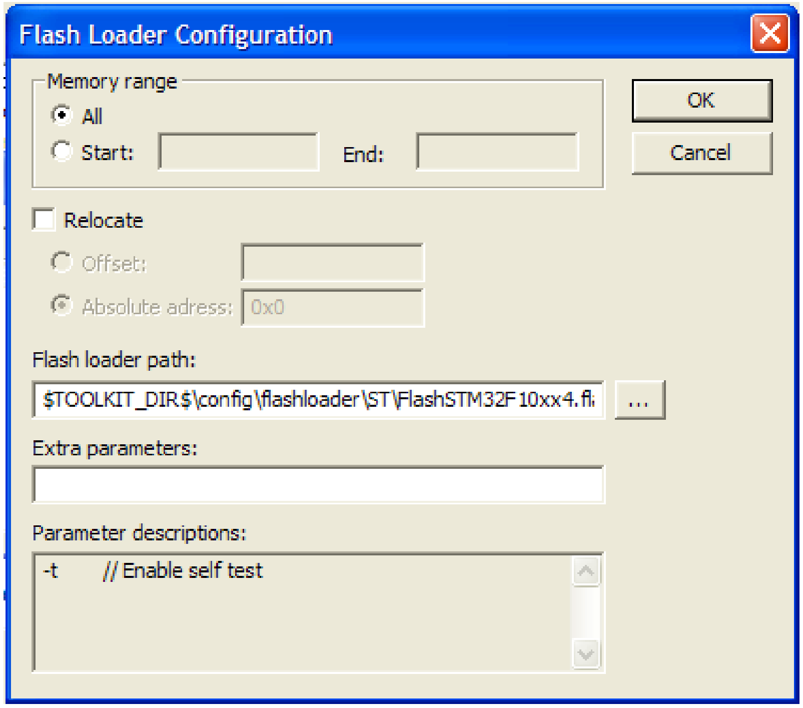
Use the button labeled "…" to select a suitable predefined file for your system. If no such file is available, you can create a new file, or modify an existing file, by clicking on the Edit… button. The following dialog will appear. (If you edit one of the predefined files in the IAR Embedded Workbench installation directory, you will be prompted to save the modified file to a different directory.)



Range	Offset/Address	Loader Path	Extra Parameters
CODE :	-	$TOOLKIT_DIR$\config\flashloader\ST\FlashSTM32F10xx4.flash	

Managing the flash loader passes

It displays one row of information for each separate flash memory device on the board, or for each flash loader pass. Click New… to define a new pass, Edit… to modify an existing pass, or Delete to remove a pass from the list. New… or Edit... both lead to the next dialog.



Flash Loader Configuration

Memory range

● All

○ Start: End:

□ Relocate

○ Offset:

● Absolute adress: 0x0

Flash loader path:

$TOOLKIT_DIR$\config\flashloader\ST\FlashSTM32F10xx4.fl

Extra parameters:

Parameter descriptions:

-t // Enable self test

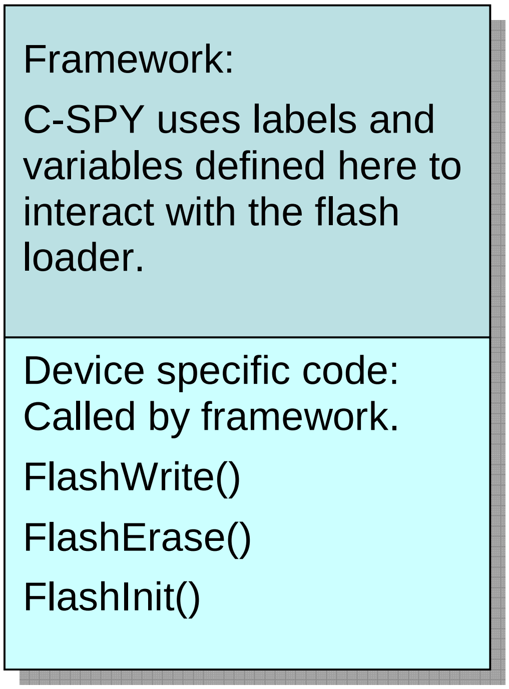
Specifying the details of a flash loader pass

The Memory range section specifies which subset of the full debug file to use for this pass. The Relocate section specifies an optional relocation of the debug file data before programming flash. The flash loader path field specifies a `.flash` file for this particular flash memory device. The Extra parameters field contains space-separated command line arguments to pass to the flash loader program (in the form of `argc/argv` parameters to the `FlashInit` function.) The Parameter descriptions field displays information about the allowed extra parameters, if available.

The end result of using these dialogs is one `.board` file, which specifies the full flash loading sequence.

Creating a flash loader program

The flash loader is a native application that you can develop using the IAR Embedded Workbench. It consists of two parts: the flash loader framework code, supplied by IAR, and the device specific code. You will write the device specific code, normally as a small set of C functions.



Framework:

C-SPY uses labels and variables defined here to interact with the flash loader.

Device specific code: Called by framework.

FlashWrite()

FlashErase()

FlashInit()

A simple example

The following example shows the source code for a complete flash loader (except the framework code), 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 flash memory which shall be programmed. A given flash loader can thus be used for any number of different flash devices, with different total size, page size or block layout, provided that they all employ the same flash programming algorithm. The flash memory device configuration file (`.flash`) is used to specify such variations between flash memory devices.

The reference section at the end of this document lists all framework functions in detail.

Building the flash loader

Create a new EWARM project including copies of these files, found under arm\src\flashloader\framework2 in a EWARM installation directory:

`flash_loader.c` Framework code. This file should be a member of your flash loader project, but should not be modified.

`flash_loader.h` Framework declarations, for example the C prototypes of your C functions.

`flash_loader_extra.h` Additional framework declarations, rarely needed by your code.

`flash_loader_asm.s` Framework code, of the low level, processor specific, kind. This file may have a different name, or at least a different extension, for different processor architectures. This file should be a member of your flash loader project, but should not be modified.

`template\flash_config.h` Template for your own configuration file. You should copy this to a file called `flash_config.h` and edit it appropriately.

Set up your EWARM project according to the specific characteristics of your device. This includes settings for the compiler, the linker with the linker command file, etc. Read more in the *IAR Embedded Workbench® IDE User Guide for ARM®* and in the *IAR C/C++ Development Guide for ARM®.*

Flash memory device configuration files

A flash memory device configuration file (`.flash`) is an XML file which specifies properties of a certain flash memory device, including which flash loader to use for programming it. Below is an example of such a file.

```
<?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 0x200</block>
</flash_device>
```

Mandatory tags

`<exe>`
The path to the flash loader program. The path can contain a variable such as `$TOOLKIT_DIR$`.

`<flash_base>`
This is the Base Address of the flash memory when it is written to, as described previously.

`<page>`
This is the flash memory page size.

`<block>`
The block layout of the flash memory is specified by one or more of these tags, *in order*. Each tag contains a (decimal) count followed by a (hexadecimal) block size. This tag sequence should fully specify the sequence of blocks making up the flash memory. In the example above, the flash device contains two blocks of size 0x100, followed by three blocks of size 0x200, for a total size of 0x800.

Optional tags

`<gap>`
The ordered block sequence can also contain one or more `gap` tags intermixed with `block` tags. Each `gap` tag contains a hexadecimal gap size. It specifies an area within the extent of the flash memory device which doesn’t contain flash memory. C-SPY will signal an error if the image file contains data that would be placed into a gap.

`<macro>`
The path to a C-SPY macro file, which will be loaded in conjunction with downloading the flash loader program. There are three C-SPY macros that will be called automatically, if they are defined in this file: `execUserFlashInit` will be called immediately before loading the flash loader, `execUserFlashReset` will be called directly after the reset that follows the loading, and `execUserFlashExit` will be called after flash loading has finished, before the flash loader program is unloaded.

`<filler>`
A decimal number which specifies the byte value to use when padding write operations to page boundaries. The default value is 255 (0xff).

`<checks>`
If this is 0, it disables checks for the error code return value from the flash loader functions (such as `FlashWrite`), for a slight performance boost. Use only when experience tells you that this error checking *really* is superfluous.

`<aggregate>`

If this 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, preferably more, blocks will fit in the download buffer. It requires that the flash loader program can program more than one block in a single operation.

`<args>`

This can contain arguments to the flash loader, in the form of `argc/argv` parameters to the `FlashInit` function. The parameters should be separated by *newlines* in this field.

`<args_doc>`

This is a text field which should contain descriptions of the parameters accepted by the flash loader `FlashInit` function. It is shown in the Flash Loader Configuration dialog. It can contain multiple lines of text, separated by newlines.

Often, a certain processor is available in many variants, each with the same type of flash memory but with different sizes and addresses, and possibly block layouts. For such a scenario, only one flash loader program would be needed, but several flash memory specification files. Consider the following table describing some variants of a hypothetical “P8” processor family.

Variant	Flash size (kbyte)	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_4b.flash

There would be five different flash memory configuration files, but each of the configuration files would specify the same flash loader program, because each of the processor variants has a flash memory device of the same type, requiring the same flash programming algorithm.

Flash memory system configuration files

A flash memory system configuration file (`.board`) is an XML file which specifies properties of a certain development board with respect to flash memory devices. Below is an example of such a file, which specifies two flash programming passes for two different flash memory devices.

```
<?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>
  <pass>
    <loader>$TOOLKIT_DIR$\config\flash\flash_p8_2b.flash</loader>
    <range>CODE 0x20800 0x21000</range>
    <abs_offset>0x10000</abs_offset>
  </pass>
</flash_board>
```

Mandatory tags

`<loader>` The path to the flash memory device file (`.flash`). The path can contain a variable such as `$TOOLKIT_DIR$`.

Optional tags

`<range>` This specifies the subset of the original image file which should be programmed in this particular flash memory device. It consists of a segment name (usually `CODE`), followed by the address of the first and last byte of the range, in hex. If there is only one flash memory device, the range defaults to the whole image file. If there is more than one pass, this field is mandatory, because C-SPY needs to know how to partition the image.

`<abs_offset>` This is used to write the image to flash memory at an address different from the address where it was placed by the linker. For example, if the flash memory device is mapped into the memory at a certain address when it is programmed and then later remapped to another address when executing, we would need to use an appropriate offset to compensate when programming the flash memory. The `abs_offset` field specifies an absolute address where the first byte of the image file should be placed.

`<rel_offset>` This is like the `abs_offset` field, but specifies a relative offset with which each record in the image file should be displaced before writing to flash. This can be either a positive or a negative number. Only one of `abs_offset` and `rel_offset` can be used for the same pass.

`<flash_base>` This is the Base Address of the flash memory when it is written to, as described previously. If this tag is present, it overrides the corresponding tag in the `.flash` file.

`<args>` This can contain arguments to the flash loader, in the form of `argc/argv` parameters to the `FlashInit` function. The parameters should be separated by *newlines* in this field. These parameters are appended to any parameters given in the `.flash` file. If the flash loader program handles parameters appropriately, these parameters can thus override the ones in the device file.

Framework reference

The following functions *must* be implemented by a flash loader:

FlashWrite

```
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 write operation writes.

`offset_into_block` This is 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` The number of bytes to write.

`buffer` This points to the buffer containing the bytes to write.

The return value should be one of `RESULT_OK` or `RESULT_ERROR`.

FlashErase

```
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` The size of the block, in bytes.

The return value should be one of `RESULT_OK` or `RESULT_ERROR`.

FlashInit

```
#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;
```

As shown above, there are two different prototypes for `FlashInit`, determined by the value of the preprocessor macro `USE_ARGC_ARGV` which you must specify in `flash_config.h`. Use the flavor with arguments if you need the extra flexibility. The actual arguments can be specified in the flash memory configuration file, or in the Project Options dialog in the IAR Embedded Workbench IDE.

Parameters:

`base_of_flash` Points to the first byte of the whole flash memory range.

`image_size` The size of the whole image that is to be written to flash memory, in bytes.

`link_address` 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 will need this parameter.

`flags` Contains optional flags. See below for details.

Advanced FlashInit functionality

The `FlashInit` function is the first function called in the flash loader. As such, it affords an opportunity to provide extra information to C-SPY before the actual flash programming starts. This takes the form of a number of optional overrides of the properties specified in the flash memory configuration file.

The overrides are specified using a set of macros defined in the optional header file `flash_loader_extra.h`. Internally, this functionality requires access to a structure variable defined in the framework, which is used to pass information back and forth between C-SPY and the flash loader. The variable is called `theFlashParams` and is also declared in the header file, as follows:

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

extern FlashParamsHolder theFlashParams;
```

Overriding page size

To override the page size, use the `SET_PAGESIZE_OVERRIDE` macro and set the bit `OVERRIDE_PAGESIZE` in the return value, as exemplified below:

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

Overriding buffer size

The download buffer size is normally determined by the position of two labels, `FlashBufferStart` and `FlashBufferEnd`, which get their positions at link time. In order to be able to use the same flash loader for multiple derivatives which only differ in RAM size, the flash loader can override the buffer size (if it can determine the actual amount of RAM available, of course). Use the `SET_BUFSIZE_OVERRIDE` macro and set the `OVERRIDE_BUFSIZE` bit in the return value from `FlashInit` function. Don't try to decrease the buffer size.

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

Overriding block layout

Normally, the flash memory configuration file specifies the block layout of the flash memory, using the `<block>` tags (and `<gap>` tags). Sometimes it is more practical to let the flash loader program determine the block layout by "querying" the flash memory device itself in some fashion. If the flash loader wants to specify a layout, it should put the layout description in the flash download buffer and add the constant `OVERRIDE_LAYOUT` to the result code of `FlashInit`. A pointer to the download buffer is available using the `OVERRIDE_BUFFER_PTR` macro. The syntax is the same as in the file (a decimal block count followed by a hexadecimal block size), except that blocks are separated by comma, as in the example below:

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

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

Combining overrides

All of the above overrides can be combined. The example below uses all overrides.

```
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;
};
```

Overriding the flash loader itself

The most drastic override is if the flash loader detects that the flash memory device doesn’t match the capabilities of the flash loader, in essence that the wrong flash loader has been started. This would normally be the consequence of some misconfiguration, and many flash loaders would, or could, not even check this. But in the event that a flash loader can detect the flash memory device at runtime, it has an opportunity to report the device to C-SPY and let C-SPY try again with another flash loader. This is done by putting a device identifier in the buffer and returning the special result code `RESULT_OVERRIDE_DEVICE`, for example as follows:

```
uint32_t FlashInit(void *base_of_flash, uint32_t image_size)
{
  if ('unexpected flash device was found')
  {
    strcpy(OVERRIDE_BUFFER_PTR, "P8_16c");
    return RESULT_OVERRIDE_DEVICE;
  }
}
```

Note that the replacement flash loader is specified indirectly, as a flash memory device identifier. This identifier is read by C-SPY and is then used as the key in a table lookup to locate another flash loader program. The table is constructed as follows:

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

Flags

The flags parameter to FlashInit specifies optional flag bits. Only one flag value is currently defined.

FLAG_ERASE_ONLY

If this bit is set (if 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 device supports a one-stop global erase function, it can be invoked directly from FlashInit, and FlashInit should then return the constant RESULT_ERASE_DONE. Otherwise, C-SPY will continue and invoke the FlashErase function for each block.

FlashChecksum

This is an optional function. You need to implement it if you want to enable checksum verification of the downloaded flash memory contents, but you can implement it with a helper function from the framework (Crc16), as follows:

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

Note the OPTIONAL_CHECKSUM macro. This is needed to make sure that this optional function and its framework wrapper are both included in the linking of the flash loader.

FlashSignoff

This is an optional function. You can implement it if you need to perform some cleanup after flash loading has finished. It is called after the last call to FlashWrite (or after FlashChecksum if it exists).

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

Note the OPTIONAL_SIGNOFF macro. This is needed to make sure that this optional function and its framework wrapper are both included in the linking of the flash loader.

Debugging

Since the flash loader program is not a standalone program, with a main function, it cannot very easily be debugged.

While developing the flash loader, it is probably best to first make a very simple test harness containing a main function which simply calls FlashInit, FlashWrite and FlashErase with suitably prepared, or generated, data and parameters. This program should of course be linked to a RAM address and can then be debugged as a normal application until the basic flash programming code seems correct.

Then, when the flash loader is used in the actual flash loading process, you will get some help from a log file which describes the flash loading process in some detail. When you start a debug session which uses flash loaders, a log file named `flash0.trace` is generated in the project directory (`$PROJ_DIR$`), the directory 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, simply create an empty file called `flash0.trace` in that directory, and trace output will be produced every time a debug session with flash loading is started, until the file is removed again.

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

This is a sample log file:

```
File generated Tue Apr 28 11:10:59 2009

Starting fragment-style flashloader pass.
FlashInitEntry is at 0x001001F0
FlashWriteEntry is at 0x00100200
FlashEraseWriteEntry is at 0x00100208
FlashBreak is at 0x001001EC
FlashBufferStart is at 0x00100300
FlashBufferEnd is at 0x001FFF00
FlashChecksumEntry is at 0x00100210
FlashSignoffEntry is at 0x00100218
page size is 16 (0x10)
filler is 0xff
buffer size is 1047552 (0xffc00)
SimpleCode records (after offset):
  Record 0: @ 0x0 [60 (0x3c) bytes] 0x0 - 0x3c
  Record 1: @ 0x80 [9034 (0x234a) bytes] 0x80 - 0x23ca
Base of flash at 0x0
->init       : base @ 0x0, image size 23ca
      Args: (argc = 3)
          D:\dev\marran\test\printf\Debug\Exe\printf.out
          -b
          -a
  timing(init): 0.0000 (CPU) 0.0000 (elapsed)
Transaction list:
  Transaction @ 0x0 (0x23d0 bytes) 11 packet(s).
    Will erase 11 block(s):
      0: 0x0 (0x100 bytes)
      1: 0x100 (0x100 bytes)
      2: 0x200 (0x200 bytes)
      3: 0x400 (0x200 bytes)
      4: 0x600 (0x200 bytes)
      5: 0x800 (0x200 bytes)
      6: 0xa00 (0x200 bytes)
      7: 0xc00 (0x200 bytes)
      8: 0xe00 (0x200 bytes)
      9: 0x1000 (0x1000 bytes)
      10: 0x2000 (0x1000 bytes)
->multi_erase: 11 blocks (0x58 bytes in buffer) [0 0 0]
  timing(erase): 0.2813 (CPU) 0.2810 (elapsed)
->write      : @ 0x0 (0x23d0 bytes, offset 0x0 into block @ 0x0) [18 f0 9f]
  timing(write): 0.3125 (CPU) 0.3120 (elapsed)
->checksum   : @ 0x0 (0x3c bytes)
  timing(checksum): 0.0313 (CPU) 0.0320 (elapsed)
->checksum   : @ 0x80 (0x234a bytes)
  timing(checksum): 6.0625 (CPU) 6.0780 (elapsed)
->signoff
  timing(signoff): 0.0000 (CPU) 0.0000 (elapsed)
Duration:   7.88 (CPU)   7.91 (elapsed)
  of which on target: 6.6875 (CPU) 6.7030 (elapsed)
Flash loading pass finished
```

It starts by reporting the addresses of some key functions in the flash loader, and some basic properties of the flash memory and flash loader. It then lists the data records from the image to be downloaded. The main part is the sequence of write and erase operations, each containing the start address, the size, and, at the end of the line, the three first bytes of the data for that operation. Optionally, at the end, follows one or more checksum operations.