Application Note 111

Flash Programming

Document number: ARM DAI 0111
Issued: May 2003

ARM

Application Note 111
Flash Programming

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
May 2003	A	First release

Proprietary notice

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction

This application note discusses various methods for programming flash memory on ARM based systems.

It contains the following sections:

1. This introduction.
2. An explanation of the AXD Flash Download Utility. This chapter describes how AXD can be used to write a binary file to the flash memory on an ARM Integrator/AP board.
3. An explanation of flash programming functionality provided by the ARM Firmware Suite. This includes the ARM Flash Utility, the Integrator/AP Boot monitor, and the ARM Flash Library.
4. A thorough explanation of a simple flash download utility written in C.
5. A detailed description of how to port the utility discussed in section 4 to program flash on other boards. As an example, this section shows how to port the utility to work on the ARM Evaluator-7T.
6. References for further information.
7. Appendix discussing the Common Flash Interface (CFI).

Note *If you are using the RealView Debugger (RVD), it is still possible to use the utility discussed in section 4 of this application note. In addition, RVD also contains an integrated flash programming mechanism as described in Application Note 110 "Flash Programming with RVD".*

2 AXD Flash Download Option

2.1 About the Flash Download Option

When you invoke the flash download option from within AXD, a simple flash programming utility is loaded into RAM on the target board. The ARM core then executes this routine, which uses semihosting to read a file on the host computer, and programs this into the flash memory. The downloaded file must be in plain binary format. Refer to the *ADS 1.2 Linker and Utilities Guide* for information on converting an ELF format file to plain binary using the *fromelf* utility.

2.1.1 Compatibility

The default Integrator/AP version of the flash download code is supplied as a binary in `install_directory\bin\flash.li`. This can be used to program the Intel DT28F320 devices fitted to the ARM Integrator/AP board.

Note *This version of the Flash downloader works only with the ARM Integrator/AP board. The ARM Integrator/AP cannot work in big-endian mode. A dummy flash.bi file is provided that issues a warning if you attempt to use it.*

Note *ADS 1.1 and earlier provided a flash downloader that targeted the ARM Development Board (PID) rather than Integrator. However the principles of how this works are the same.*

2.1.2 Integrator/AP Switch Settings

The switch settings on the Integrator/AP board (listed below) select which code is run following reset or power-on.

Config 1: S1-1 off (Down)

- causes the board to boot directly from Application Flash
- Flash at address `0x24000000` (the application Flash) is aliased at `0x0` and execution starts from here after reset. The application code will normally 'remap' the memory map (so that RAM appears at address `0x0`).

Config 2: S1-1 on and S1-4 on

- causes the boot monitor to run and give a command prompt
- 'Boot ROM' at address `0x20000000` is aliased at `0x0` and execution starts from here after reset. The boot monitor subsequently remaps so that RAM appears at address `0x0`.

Config 3: S1-1 on and S1-4 off

- causes boot monitor to run and then jump to Application flash
- the boot monitor performs remapping of ROM/RAM
- the boot monitor can be programmed to jump to any image in application flash.

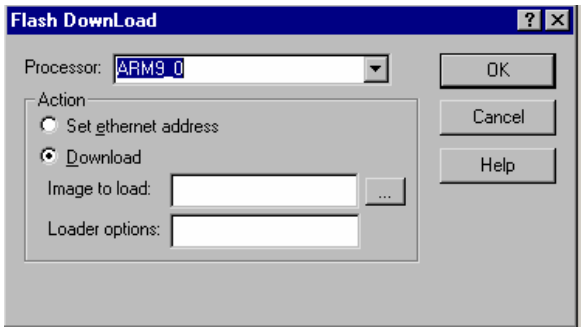
For flash programming, booting with switch 1 on is recommended to ensure that ROM/RAM remapping is performed before AXD attempts to download the flash programming code. This required as the flash programming code is downloaded to address 0x8000, so there must be RAM present at this location.

Note *Depending upon the core module used, it may be necessary to set* `$top_of_memory` *to* `0x40000`*, unless an SDRAM DIMM is fitted to the core module.*

2.2 Using the Flash Downloader from AXD

Follow these steps to use the Flash downloader from AXD:

1. Ensure that there is RAM at address `0x0` on the Integrator. This can be achieved by resetting with switch 1 'on'.
2. Start AXD, and connect to the target (either via Multi-ICE or Angel)
3. Select 'Flash Download...' from the 'File' menu. The Flash Download dialog is displayed:



4. Specify the input information (see below) or click Browse to select a binary file to download. You can either use the default block, image, and address values or enter new values:
   - If you do not enter any loader information, the downloader uses the default values:

     Image number 128

     Block number 0

     Image base 0x24000000
   - If you require different values than the defaults, enter these in the 'Loader options' field, in the format:

     ```
     [a<address> |or| b<block_no>] i<image_no> *name
     ```

     For example:

     ```
     b5 i5 *my_image
     ```
5. Click **OK**. The Flash downloader reads the binary file and displays the download settings in the Console processor view.
6. Edit the settings, if required, or press Enter. The Console view displays a message when the data has been written to the Flash.

3 ARM Firmware Suite Flash support

The ARM Firmware Suite is a collection of utilities and support code related to driving hardware. It consists of several components, supplied both as source code (to allow porting to new platforms) and as pre-built images for a range of development boards. A number of flash related elements are included, as described below.

3.1 ARM Flash Utility

ARM Flash Utility (AFU) can be used for managing and storing data in flash memory. Information about the downloaded code is also programmed into the Flash, in the format used by the Integrator's bootmonitor. This enables you to use ARM boot systems to run the code on the board.

Programming flash with AFU on supported targets can be achieved by loading the pre-built afu.axf image (provided with ARM Firmware Suite) into the debugger and running it.

This utility offers more functionality than the standard AXD Flash Download option. For example, with AFU you can delete an image, or a particular block of flash. These operations are performed by using a simple command line interface. The AFU also supports a wide range of input file formats, including ELF and Motorola S-record. Full details of this utility can be found in chapter 7 of the *AFS 1.4 Reference Guide*.

3.2 Boot Monitor

The Integrator/AP is provided with a boot monitor pre-loaded into flash, that is capable of simple flash management. Communication with boot monitor can be achieved via a connection to serial port A on the ARM Integrator/AP (e.g. via Windows HyperTerminal).

The boot monitor is completely independent of AXD. Programming flash this way is slow and not recommended, unless you do not have access to AXD. The boot monitor's flash commands are listed below:

'`BI`' sets the default flash image number to the image number specified

'`E`' for erasing all flash

'`V`' validates and displays the contents of the application flash and the SIBs

'`L`' loads Motorola S-records into flash

See the *AFS 1.4 Reference Guide,* chapter 3, for more information on boot monitor.

3.3 ARM Flash Library

The AXD Flash download facility, the AFU and the Bootmonitor all make use of the ARM flash library. This a library of flash management routines, provided with the ARM Firmware Suite (AFS), that is designed to be target independent. Where necessary, call back functions are used to implement any target specific functionality.

The library is designed to operate with boards (such as the Integrator) that contain large areas of flash memory. This space can be used to store many independent programs and associated data, and the library implements a filing system to organize these. The Flash library is not suitable for simply providing write access to flash memory.

3.3.1 Library structure and API

The Flash library defines an API which abstracts the physical flash devices used, and allows 'logical' devices to be defined which may use multiple physical devices (or parts thereof).

This abstraction is managed through the use of two linked lists of structures, defined in the Flash library. The physical devices are defined by structures of type `flashPhysicalType`, which contains information of the size, location and type of Flash. These structures also contain pointers to functions for performing basic flash operations (write, read, erase etc). The logical devices are defined in a list of `flashType` structures. These contain a pointer to the `flashPhysicalType` structure describing the first physical flash device that is to be used for this logical device, an offset into this device, and a size. The offset allows the first part of a Flash device to be used for a different logical device. If the size exceeds the remaining space in the Flash device, subsequent devices from the linked list of `flashPhysicalType` structures will be used.

Most of the Flash library functions take a `flashType` structure as a parameter. They use this parameter to locate the physical device to use. The structure describing this device is then used to locate the functions to program that device.

The Flash library maintains a structure in the Flash which allows storage of multiple independent images. An image is stored in as many blocks as necessary, and information about the image is stored in a number of structures in the final block used. If the final block does have enough space for these structures, an extra block will be used. No Flash library structures are stored before the image, so in a system when a Flash device starts at address 0x0, the first instruction of the image will be at the reset vector.

4 Example Flash Programming Utility

4.1 Introduction

This section presents a simple C utility, written to demonstrate flash programming on an ARM based system. This utility does not use the ARM Flash Library, as it aims to demonstrate simple functions for writing to flash memory. No filing system or image management is implemented.

The utility is written for the ARM Integrator/AP or /CP development boards, which use Intel Flash. In section 5, an example is shown explaining how to port the utility to the ARM Evaluator-7T board, which uses SST flash devices.

4.2 Example Utility files

The example utility consists of five files:

- `Flashdemo.c`, discussed in section 4.2.1
- `Flash.h`, discussed in section 4.2.2
- `Intel_Flash_Routines.c`, discussed in section 4.2.3
- `Integrator_target.c`, discussed in section 4.2.4
- `Integrator.h`, also discussed in section 4.2.4. This requires slight modification for the Integrator/CP, as described in section 4.6.

4.2.1 Flashdemo.c

This is the main example code. It is intended to demonstrate the use of the flash routines, rather than be a fully functional flash programming utility.

Apart from standard C library headers, this source file only relies upon `Flash.h`, which specifies the interface to the flash functions.

The utility programs a file from the host computer into flash, using semihosting operations. It starts by checking if any parameters were passed to the utility (as `argv` and `argc`). If so, it assumes that the first parameter is a filename, and attempts to open this file. If this fails, the user is prompted for a filename. If a further argument was supplied, this is assumed to be the base address to program in Flash, otherwise the user is also prompted for this.

Functions from the API in `flash.h` are then called to perform the flash programming, as described in section 4.3.

4.2.2 Flash.h

This file describes a minimal interface to the flash programming functions. It is intended that these definitions should remain unchanged when functions for new flash devices are added. This interface is divided into 2 sections:

- Flash specific functions, implemented in `Intel_Flash_Routines.c`.
- Target specific functions, implemented in `<target>_target.c`.

Section 4.3 describes each of the functions in this interface.

4.2.3 Intel_Flash_Routines.c

This file contains all of the code for accessing the Intel flash memory. These functions have been written and tested using the Integrator/AP development platform, but is intended to be generic for all Flash devices using the Intel Basic Command Set. It does not make use of the Common flash interface (CFI) implemented on the Integrator's flash devices (see section 7 for a brief explanation of CFI).

This file relies upon `integrator.h` to define the location, size and geometry of the Flash. In order to support other targets containing Intel Flash, only the included file should need to be changed.

Note *This code has only been tested on the Integrator development platforms. In some places, the code is conditionally compiled depending on symbols such as fc32x1. These symbols represent the access width of the Flash, and only fc16x2 (two, 16 bit wide flash devices used in parallel) has been tested.*

4.2.4 Integrator_target.c & Integrator.h

These files define platform (rather than flash) specific functions and parameters. `Flash_Write_Enable` and `Flash_Write_Disable` are defined in `flash.h`, hence need to be implemented even if they do not need to do anything. For the Integrator/AP platform, they write to the EBI (External Bus Interface) registers to enable/disable write access.

4.3 Flash API

Flash memory consists of a number of blocks. These are often, but not always, of uniform size. Flash devices only allow an entire block to be erased at once – which sets all storage bits within the block to a logical 1 (hence erased flash memory reads as 0xFFFFFFFF). Individual words of memory can be written, but the location must be in the erased state first. Programming and erasing flash memory usually requires writing a sequence of data values (usually termed commands) to specific locations with the flash device memory space.

Some flash devices also have additional features, such as allowing the entire chip to be erased in one operation, individual blocks to be 'locked', and quicker mechanisms for writing entire blocks. The API defined in `flash.h` is designed to expose the common flash features (such as block erase but single word write), to allow efficient handling of Flash devices. However, it hides features of specific devices, whilst providing a flexible enough interface for the flash functions to make use of these features if appropriate.

For example, a `Flash_Write_Area` function is defined to write to an area of memory. In some devices, this will simply call `Flash_Write_Word` repeatedly. However, the Intel Flash Functions use this function to utilize the block write capabilities of the Intel device.

A "non-destructive write" function has not been included (where an area within a block can be written without erasing the rest of the block). With most flash devices, this would involve reading (and storing) the area of the block that will not be written to, then erasing the whole block and writing this data back. This would increase the memory requirements of the flash functions, and probably make them reliant on dynamic memory allocation. However, the interface makes implementation of this in the application program easy, where memory management can be tailored more easily to the specific application.

4.3.1 Flash_Write_Word

This function writes a single word (32 bits) to Flash memory. It is passed the address to write to, and the data value to write there. It does not erase the location first, so should only be called with an address that has already been erased. The physical access size to flash should be transparent to callers of this function, which may need to perform, for example, two individual half-word writes.

4.3.2 Flash_Write_Area

This function writes a block of data to Flash memory. This area does not have to be contained within a single block, but the calling application does have to ensure that the entire area is erased before calling this function. It is passed the base address of the area to be programmed, a pointer to the source data, and the amount of data to write in bytes.

This function returns PASS (0) or FAIL (1).

4.3.3 Flash_Calc_Blocks

This function returns the area of flash that will need to be erased before programming a given area. In other words, it returns the start and total size of all blocks which overlap with the memory area passed in. This allows the application to check whether any valid data it has stored previously will be erased (and/or store this data to write back later), without the application needing to know anything about the particular Flash devices block structure.

This function is passed two variables by reference, which hold base address and size (in bytes) of the area specified. These variables are updated with the base address and size of the area that will need to be erased.

This function returns PASS (0) or FAIL (1). FAIL could be used to indicate that the area lay outside the Flash memory, though the example code always return PASS.

4.3.4 Flash_Erase_Blocks

Given a base address and size, this function erases all blocks which overlap this area. As the area does not have to coincide with block start addresses and size, the application can either call `Flash_Calc_Blocks` first or not, depending upon whether it needs to check the region that will be erased.

4.3.5 Flash_Init

Any flash specific code that must be run before the flash can be written to.

4.3.6 Flash_Close

Any flash specific code that must be run after the flash writing is no longer required.

4.3.7 Flash_Write_Enable

Target (as opposed to flash) specific initialization code.

4.3.8 Flash_Write_Disable

Target specific finalization code (such as disabling write access to the flash device).

4.4 Intel Flash Functions

The Flash used on the Integrator/AP development board is Intel DT28F320. These are 16 bit wide devices, with two used in parallel to provide 32 bit wide access. 8 devices are used in total (arranged 4x2) to provide 32 Mbytes. These parameters are specified in `integrator.h`.

Although the access width to memory is normally hidden from the application by the memory controller, it is important to know when programming flash, as commands must be written to particular physical devices. For example, if only one, 16 bit flash device was used, writing commands to erase a block at the start address of the flash device would erase a continuous block. As 2 parallel devices are used, doing this would erase alternate halfwords over twice the memory range. It is therefore necessary to issue block erase

commands on both halves of the data bus, though this can be done in a single, 32 bit write.

4.4.1 Static functions

Five static functions are implemented in `Intel_Flash_Routines.c`. These are marked static as they do not form part of the API defined in `flash.h`, and are not intended to be called from the main application.

Flash_Unlock_Block

This function relates to a feature specific to the Intel Flash device, that allows individual blocks to be locked. Once a block is locked, it can not be written to again until a specific command sequence is written that unlocks all the blocks in a given device. This function will be called by the erase functions and write functions if they detect a block they are attempting to write to is locked. It will ensure both parallel flash devices are unlocked.

ReadyWait

After issuing a command to a flash device (eg erase block), the flash device may take some time to complete this action. No further commands should be written to the device until the action is complete. This function reads a status bit in the flash device which reflects whether the action is complete, and loops until it is.

Command

Issuing a command to the Intel Flash involves writing a particular data value to a particular location. This function performs that write, ensuring that both parallel devices are written to simultaneously.

Flash_Write_Block

This writes an area of memory, but this area must be entirely within one block. This allows a 16 word buffer within the flash device to be used, improving the write speed. Calling `Flash_Write_Area` with an address range which happens to fall within one block will simply result in a call to this function. Calling `Flash_Write_Area` with an address range that spans multiple blocks will result in a call to this function for each block overlapped.

This function first calls `Flash_Unlock_Block` to ensure that the block is not locked. The address passed simply needs to be within the correct block. After this, it issues the `BLOCK_WRITE_MODE` command to put the flash into the correct mode, and checks the status bit. `ReadyWait()` can not be used for this, as the `BLOCK_WRITE_MODE` command must be re-issued if it was not accepted.

The number of words to write into the buffer is then calculated, and written to an address in the correct block. The maximum number of words that can be written to the buffer is 16, but there will not always be this many words left to write. The buffer is actually 16 half-words in size, but as we are programming 2 devices simultaneously we can write 16 words.

The `PROGRAM_VERIFY` command is then issued, which allows `ReadyWait()` to check the status bit. If there is more data left to program, the whole process is repeated.

Flash_Erase_Block

This erases a single block. The block is first unlocked, then an erase command followed by a confirm command is written. The device is then put into `READ_STATUS` mode, and `ReadyWait()` called to wait for completion before the status is check to see whether the erase was successful. Before returning, the device is put back into `READ_ARRAY` mode, which allows the flash contents to be read as normal.

4.4.2 API functions

Flash_Write_Word

Six different versions of the main part of this function are provided, conditionally compiled depending upon the device and bus width. For the Integrator, the fc16x2 section is compiled. First, the devices are put into write mode by issuing the `PROGRAM_COMMAND` command, then the data is written to the correct address. The devices are then put into `READ_STATUS`, so `ReadyWait()` can be called.

After `ReadyWait()`, the status is read. If this shows that the write failed because the block was locked, `Flash_Unlock_Block` is called, and the process repeated.

Before exiting, the devices are put back into `READ_ARRAY` mode, and the written word read back to verify a successful write. The function returns `PASS` or `FAIL` based on this verification.

Flash_Write_Area

The address range passed into this function is broken down into potentially multiple regions, where each region is contained in a single block. The static function `Flash_write_Block` is called for each of these regions.

Before doing this, the address range is checked to ensure that it fits inside the flash memory. If it does not, nothing is written, and `FAIL` is returned.

Flash_Calc_Blocks

The address range passed is combined with the `FLASH_BLOCK` size defined in `Integrator.h` to calculate the address range covered by all blocks overlapping with the address range passed in. `PASS` is always returned.

Flash_Erase_Blocks

This function calls `Flash_Erase_Block` for each block overlapping the address ranged passed in.

Flash_Init & Flash_Close

Both of these functions simply return, as no initialization or finalization code is required for these flash devices.

4.5 Using the Utility

4.5.1 As a standard application

Building the example utility will create an ARM eXecutable File (AXF) that can be loaded into RAM on a target board using a debugger such as AXD or RealView Debugger. The connection method to the target board may be Multi-ICE, RealView ICE or a third party JTAG tool. A connection via the Angel debug monitor may also be used, however this must not be running from the flash memory that is going to be programmed.

The target address and the path / name of the file to be programmed can be supplied as command line arguments, using the debugger's facilities for specifying these. However, if they are not supplied, the utility will prompt for these to be entered. In either case, the target address may be specified in decimal, or in hexadecimal by preceding the number with '0x'. The filename must be in the format used by the host system, on which the debugger is being run.

4.5.2 Using the Example code from the AXD Flash download option

The example code is written to work with the “Flash download” option in AXD. In order to use this, the .axf file must be renamed `flash.li` (or `flash.bi` if a big-endian version is built), and placed in the <ads install>\bin directory.

When using the Flash download option, the file to be downloaded can be selected via the browse button. The base address to program this file to should be entered in the ‘options’ field (a decimal address can be used, or a hex address with a ‘0x’ prefix). If either of these are left blank (or are invalid) the code will prompt for new values.

4.6 Modifying the code for the Integrator/CP

The Integrator/CP platform uses Intel 28F640J3A120. This is very similar to the Flash parts used on the Integrator/AP, but has 128K blocks instead of 64K blocks. The memory map is also very similar to that of the AP. The application base address is still 0x24000000, though there is now 16MB of flash, with the last 256KB reserved for the bootloader.

In order to modify the utility for the Integrator/CP, it is only necessary to change 2 defines with integrator.h:

```
// Total Flash area in bytes
unsigned int FLASH_SIZE = 16*1024*1024 - 256*1024;
// Block Size
unsigned int FLASH_BLOCK = 128*1024;            // Block Size
```

The example code contains these changes made in a separate file (`IntegratorCP.h`), and a separate flash functions source file `Intel_flash_routines_cp.c` is provided with the `#include` changed to import this version of the header file.

5 Porting the example utility

5.1 Introduction

In this section, we take the example utility discussed in the previous section, and port it to the Evaluator-7T board.

This is intended to show the flash functions can be ported to different flash devices and boards, based on example code or algorithms supplied by the flash manufacturer.

5.2 Evaluator-7T Flash Memory Overview

The Evaluator-7T uses SST flash (S3C4510x01-QERO), which has different programming algorithms to the Intel Flash used on the Integrator. Only one, 16-bit device is used, which is 512Kbytes in size.

The flash memory starts at address 0x01800000. However, the bottom 128Kbytes is used by the boot strap loader and Angel. It is not advisable to reprogram this area, as without this boot code the board's memory map will not be set up, and it can be difficult to recover (impossible without a JTAG debugger connection). Therefore, this port of the code defines the flash as starting at address 0x01820000, so the area below this will not be erased.

5.3 SST Flash Overview

The SST flash is arranged in 64K blocks. However, it is also divided into 2K sectors, where an individual sector can be erased. Entire chip erasing is also supported.

Sector, block and chip erase operations require a specified sequence of 6 half words to be written to specified offsets with the relevant sector, block or chip (the first 5 steps are the same for all the erase operations, whilst the final data differs).

In order to program a halfword, the same first 2 cycles as the erase are performed, a different 3rd cycle (though the address is the same), followed by writing the required data to the required address.

Completion of the erase or write operation can be detected by reading a bit that will return the compliment of the correct data before completion, and correct data after completion. Alternatively a toggle-bit can be used, which reads 1 and 0 alternately before completion, and a constant value afterwards.

CFI identification is supported by the flash device, though not used in this example utility. Section 7 gives a brief explanation of CFI.

5.4 Code Changes

5.4.1 General

The main code in `Flashdemo.c` remains unchanged, as it only relies on the interface definition in `flash.h`, which also remains unchanged. No code is needed to enable access to the Flash on the Evaluator, so `Evaluator_target.c` simply contains dummy functions that return immediately.

`Evaluator.h` is based upon `Integrator.h`, but the parameters are altered to reflect the Flash geometry on the Evaluator. In this example, sector-erase operations are going to be used rather than block operations, so a block size of 2K is specified.

All other changes are in the flash routine source file, now called `SST_Flash_routines.c`

5.4.2 SST_Flash_routines

Definitions

The first change to make, is to #include the Evaluator header file, rather than the integrator header file. This defines all the parameters describing the Flash location, size and geometry.

A block of defines then set up the values used for the various flash commands. These are completely different from the command set used for the Intel Flash. The `INVERT_BIT` and `TOGGLE_BIT` are masks to obtain the relevant bit to check for completion, as described in section 5.3.

ReadyWait

This function performs exactly the same task as for the Intel flash (polling a status bit until the Flash operation is complete). However on the SST device we wait until this bit stops toggling, rather than looking for a specific value.

Flash_Erase_Blocks

This function is completely unchanged, as it only relies on the `FLASH_BLOCK` to divide an arbitrary area erase into individual block erase operations.

Flash_Erase_Block

This function changes completely from the Intel Flash version. The lack of a block-lock mechanism make this routine simpler, as this does not have to be checked for. The function simply write the 5 specified commands to set up an erase operation, then writes the command to specify this as a sector erase to the base address of the sector to erase. `ReadyWait` is then called before returning pass.

Flash_Write_area

This flash device does not have any buffered write, or block write functionality. There is therefore no point writing a `Flash_Write_Block` function for writing an area contained in a single block, as there was for the Intel flash routines. This function therefore just becomes a loop to call `Flash_Write_Word` for each word in the address range.

Flash_Calc_Blocks

This function is completely unchanged, as it only relies on the `FLASH_BLOCK` to perform some calculations.

Flash_Write_Word

As before, different versions of the main part of this code are provided for different device and bus widths. Only the 16x1 option is compiled for the Evaluator-7T.

As this function writes a 32 bit word, but the flash must be programmed with halfword commands and data, the write is performed as 2 separate operations controlled by a for loop. The 3 cycle command sequence for a write is issued, followed by the appropriate halfword of data to the desired address. `ReadyWait` is called before the function reads back the data (to check for success) and returns with the appropriate value.

Command

The function to write a command to the flash is very similar to the Intel Flash version, but takes a halfword command rather than a byte command

Unimplemented functions

`Flash_Write_Block` and `Flash_Unlock_Block` are simply removed, as they relate to features not present on the SST Flash (and are not part of the API).

Possible improvements

The sector, block and chip erase functions of the SST flash could be utilized by the `Flash_erase_area` function to speed up large erase operations, whilst still allowing the fine erase granularity.

This would involve implementing routines to erase blocks and the entire chip, and extending the `Flash_erase_area` function to make use of these. Additional parameters (to represent the sector and chip size) would need to be added to `Evaluator.h`, but this would not effect `flash.h`, and therefore not effect `Flashdemo.c`.

6 References

For details of the ARM Firmware Suite, refer to:

- *AFS 1.4 Reference Guide*
  ARM DUI 0102F

For details of AXD and RealView Debugger, refer to:

- *ADS 1.2 AXD and armsd Debuggers Guide*
  ARM DUI 0066D
- RealView Debugger 1.6 User guide
  ARM DUI 0153B

For details of flash devices discussed in this Application note, refer to:

- Intel Flash datasheets
  Available from http://www.intel.com/design/flash/datashts/index.htm
- SST Flash datasheet
  Available from http://www.sst.com/products/part_finder.xhtml

For details of Flash programming features specific to RealView Debugger, refer to:

- ARM Application Note 110 “Flash Programming with RVD”
  ARM DAI 0110A, http://www.arm.com/arm/Application_Notes

7 Appendix - CFI flash

Many flash devices conform to the CFI (Common Flash Interface). However, this does not mean these devices have common command sets and programming algorithms.

The CFI standard provides a common way of obtaining a parameter block from a flash device. This parameter block contains details such as the flash size, block structure etc, as well as numbers denoting a manufacture and command set used. Software utilizing the CFI must interpret these numbers, and select programming functions appropriately.

This mechanism allows software to support many different flash devices, and automatically detect which type of device is being used. This is very useful for applications where the flash device may be changed (eg if removable flash cards are used), but is not as useful for applications running in an environment where the flash type is known and fixed.

CFI commands are supported in addition to the specific command set on any particular device, and do not have to be used to program the device.