Creating Flash Algorithms with Eclipse

Application Note 190

Released on: August, 2007

ARM®

DAI0190A

Creating Flash Algorithms with Eclipse
Application Note 190

Release Information

The following changes have been made to this application note.

Table 1 Change history

Date	Issue	Change
August 2007	A	First release

Proprietary Notice

Confidentiality Status

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

1 Introduction

The ARM *RealView Development Suite* (RVDS) 3.1 includes the Eclipse IDE and several plug-ins to integrate Eclipse with other supplied tools. An update to Eclipse is available from the ARM website to provide the capability to work with flash memory devices, including creating new flash algorithms and writing data to flash. To install these updates, configure your Eclipse to search for updates from `http://www.arm.com/eclipse/`.

Creating a new flash algorithm requires the *RealView Compilation Tools* (RVCT) to build your source code, as well as *RealView Debugger* (RVD) to debug and test the resulting executable. RVCT and RVD are both supplied with RVDS 3.1. The Eclipse Flash Programmer plug-in communicates with your target hardware using *RealView ICE* (RVI). RVI is available to purchase separately from ARM.

This application note describes in detail how to create, debug, test and use new flash algorithms for the Eclipse Flash Programmer plug-in. It is assumed you have both RVDS 3.1 and RVI 1.5 or later installed, are familiar with writing C code and have access to the datasheets for your flash device and your target hardware. If you are using a version of RVDS greater than 3.1, check on the ARM website for updates to this application note because there may have been changes that affect flash functionality.

Refer to the *RealView Development Suite Eclipse Plug-ins User Guide* for general information about using Eclipse and the Flash Programmer plug-in. This document is installed as part of RVDS or is available on the ARM web site from:
`http://www.arm.com/pdfs/DUI0330B_eclipse_user_guide.pdf`

You may also find the following related application notes on the ARM website useful:

- *Application Note 110: Flash Programming with RealView Debugger*
  `http://www.arm.com/pdfs/AN110.zip`
- *Application Note 111: Flash Programming*
  `http://www.arm.com/pdfs/an111.zip`

2 Basic Principles

This section describes the basic principles of flash devices and how these relate to the flash algorithms used by the Eclipse Flash Programmer plug-in.

2.1 Flash Device Width

Flash devices typically have a width of 8-, 16- or 32-bits. For 8- and 16-bit flash devices it is common to place multiple flash devices in parallel to occupy a 32-bit bus. An example is ARM's Versatile PB926EJ-S board, which contains two 16-bit flash devices in parallel.

In the case that flash devices are placed in parallel on the bus, you can choose to access the flash devices sequentially or in parallel.

If the flash devices are accessed sequentially, then you erase, read or write data on one flash device, and then erase, read or write data on the next flash device, and so on through all the flash devices on the bus.

In the parallel approach, you access all the flash devices at the same time using 32-bit memory accesses; this means duplicating commands four times for 8-bit flash devices and two times for 16-bit flash devices to fill a 32-bit word, for example. Accessing flash devices in parallel assumes that they all use the same command set and have the same block structure, which is normally the case. However, you must be careful when polling parallel flash devices for their status, because the flash devices may complete operations at slightly different speeds; this means you must wait for all the devices to signal completion before proceeding with the next operation.

The recommended approach is to access flash devices in parallel because this results in faster flash operations.

2.2 Flash Device Structure

Flash devices are manufactured in many different sizes and with many different block configurations. The flash programmer needs to know the structure of the flash device so that it can correctly partition your data into blocks. You must implement the `flashGetBlockStructure()` method, returning an array of `FlashBlockDesc` structures that describes the block structure of your flash device.

The easiest way to describe the block structure of your flash device is to return a pointer to a `FlashBlockDesc` array that is statically defined in your algorithm, and which is hardcoded with the correct values. This approach works well when you are always using a known flash device because you can look up the necessary values in the flash device datasheet.

An alternative is to query the block structure of your flash device at run-time, for example using the *Common Flash Interface* (CFI). This approach is more complex to implement, but has the advantage that your algorithm works across a wider range of flash devices without modification. In this case, you must still return a pointer to a statically defined array, because your algorithm cannot rely on having a heap set-up when it is run from the Eclipse Flash Programmer plug-in.

When returning the block structure it is important that you take into account whether your target is using multiple flash devices in parallel. In this case, the flash programmer treats the parallel devices as a single device, so you must calculate the effective block size as being the block size of a single device multiplied by the number of devices in parallel on the bus.

2.3 Flash Device Write-Protection

Flash devices typically provide a means of write-protecting the entire device in hardware, for example with a dedicated write-protect pin. If your target hardware uses this write-protection feature then you must disable write-protection during erase and write operations and should consider re-enabling write-protection afterwards.

The safest approach is to disable write-protection only during the erase and write methods in your algorithm. An alternative is to disable write-protection globally in the `flashSelect()` method and re-enable it in the `flashDeselect()` method.

2.4 Flash Device Block Locking

Many flash devices provide a means to lock individual blocks or groups of blocks so that they cannot be erased or written to, and in this case the default state of the flash device out of reset is usually to have all the blocks locked.

If your flash device supports block locking, you must unlock each block prior to erasing or writing to it, and should consider locking it again afterwards. The locking and unlocking must be performed in the erase and write methods in your algorithm as you process each block of data. The exact details of how to lock and unlock blocks are dependent on your flash device.

2.5 Flash Device Timeouts

Many operations on a flash device, particularly writing, take a long time to complete. It is useful to be able to specify a timeout on lengthy operations, so that the Eclipse Flash Programmer plug-in can abort an operation if it fails to complete. Specifying a timeout is especially useful when developing a new flash algorithm, because the algorithm may contain errors that cause it to hang.

You can specify a timeout for erase, read, write and verify operations by implementing the `flashGetEraseBlockTimeout()`, `flashGetReadBlockTimeout()`, `flashGetWriteBlockTimeout()` and `flashGetVerifyBlockTimeout()` methods respectively. The timeouts are per block and are specified in ms. There is also `flashGetEraseDeviceTimeout()` to specify the timeout for erasing the entire flash device. The easiest solution is to determine the timeouts from the datasheet of your flash device and to hard-code these values into your algorithm. Datasheets for flash devices often list typical and worst-case timeout values; you must use the worst-case timeout values to ensure that flash operations are not terminated prematurely.

2.6 Flash Erase Operations

Most flash devices need to be erased prior to having data written to them. Flash devices allow erasing individual blocks and sometimes erasing the entire device. The erase operation causes the flash contents to be set to a manufacturer specified erase value, which typically has all bits set.

If your flash device requires erasing, then you must implement one or both of the `flashEraseDevice()` or `flashEraseBlocks()` methods. Given a choice of erasing methods, it is recommended that you first implement the `flashEraseBlocks()` method, because this allows finer grained erasing by the flash programmer. Implementing the `flashEraseDevice()` method is useful where erasing the whole flash device is substantially faster than erasing it block by block, and where you expect users of your flash algorithm to frequently want to erase or write to the entire flash device.

A few flash devices combine the erase and write operations together. In this case you can omit the erase methods from your flash algorithm.

2.7 Flash Write Operations

The write operation causes data to be written to the flash device. The simplest writing scheme allows writing a word of data to any address in a block. Many flash devices provide more advanced writing schemes, such as using a small buffer, and use of these can significantly speed up write operations. When developing a new flash algorithm it is recommended that you first implement the simplest writing scheme, and when that is working correctly, then consider whether attaining a faster writing speed is important enough to require a more complex implementation.

2.8 Flash Read Operations

From the end-user perspective, most operations on the flash device involve writing data. However, there are some circumstances where the flash programmer internally needs to read the contents of flash. When the user requests a write and the data does not entirely fill a block, then optionally, the flash programmer fills the remainder of the block with the existing contents of flash, and this requires reading the block contents prior to erasing and writing it. Reading the contents of flash is also required after writing data to flash, when the user has requested a verify operation and where the flash algorithm does not implement a verify method.

NOR flash devices are normally mapped into target memory, so the Eclipse Flash Programmer plug-in can read the contents of flash by directly reading from the corresponding memory addresses. If for some reason it is not possible to read the contents of flash directly from memory, then you must implement the `flashReadBlocks()` method in your flash algorithm.

For NAND devices you must always implement the `flashReadBlocks()` method in your flash algorithm. This is because although NAND devices have their control interface mapped into memory, the data itself is not directly accessible without sending commands to the flash state machine.

2.9 Flash Verify Operations

After writing some data to a flash device, a verify operation is optionally performed by the Eclipse Flash Programmer plug-in. The purpose of the verify operation is to check that the data has been written correctly. Performing a verify helps to detect faulty flash devices as well as errors in the flash algorithms.

The Eclipse Flash Programmer plug-in internally implements the verify operation by reading the contents of the flash device and comparing it against the data that was written. The read uses the `flashReadBlocks()` method in the flash algorithm, if implemented, and otherwise attempts to read directly from the base address of the flash device in memory.

You can optionally implement the `flashVerifyBlocks()` method in your flash algorithm if you need control over how the verify operation is performed.

2.10 Buffering

The Eclipse Flash Programmer plug-in and the test harness transfer data to and from the flash algorithm using a buffer. The address of the buffer is passed as a parameter to the `flashWriteBlocks()`, `flashReadBlocks()` and `flashVerifyBlocks()` methods.

The Eclipse Flash Programmer allocates as much RAM as possible for the buffer, to try to access as many blocks as possible in a single operation. In this case the buffer is always a whole multiple of the block size.

If there is insufficient RAM available to buffer even a single block, then the Eclipse Flash Programmer plug-in and the test harness switch to a sub-block access scheme. In this case, the buffer size starts off at the block size, and is repeatedly halved until it fits into RAM. When the

flashWriteBlocks(), flashReadBlocks() and flashVerifyBlocks() methods are called, the blocks->count parameter is set to 1 and the blocks->size parameter is set to the buffer size, which in this case is less than the size of the whole block.

Some flash devices place restrictions of the alignment of the buffer or the minimum or maximum buffer size. There is no way for flash algorithms to express these restrictions to the Eclipse Flash Programmer plug-in. Instead your flash algorithm must allocate a buffer internally that does meet the requirements and copy data between it and the buffer that is supplied to the method.

2.11 Error Handling

All flash algorithm methods return values from the FlashError enumeration to indicate whether an error occurred or not. There are a number of pre-defined error codes to handle common error situations. You can define your own error codes by adding values greater or equal to FlashError_CustomErrors, and can provide user-visible error strings for these by implementing the flashGetErrorMessage() method in your algorithm.

2.12 Accessing Hardware

When accessing your flash device and any other hardware peripherals from C you must use volatile memory accesses so that the compiler does not optimize them away. To assist with this, macros M8, M16 and M32 are provided in the auto-generated C file when you create a new flash project to allow you to perform 8-, 16- and 32-bit volatile memory accesses respectively.

2.13 Stack and Heap and C Libraries

When the Eclipse Flash Programmer plug-in calls a method in your flash algorithm it creates a stack of at least 256 bytes. This means you can perform a limited number of method calls internally within your flash algorithm and store local variables on the stack.

Flash algorithms are linked against either the microlib or standard C libraries. This means you can use the standard C library functions in your flash algorithm. The C libraries also supply helper code, such as integer division, for processors that do not provide this functionality natively.

When running the flash test harness in a debugger such as RVD, the debugger initializes the C library heap. This means you can use C library functions such as malloc() during testing. However, no heap is initialized when flash algorithms are run from the Eclipse Flash Programmer plug-in, so you must not use heap-based C library functions in your final flash algorithm.

Similarly, debuggers such as RVD provide semihosting facilities that allow you to use printf() in your code and have the output appear in the debugger. Using printf() and other semihosted calls in your flash algorithm is useful for debugging and testing. However, you must not use semihosted calls in the algorithm you import into the flash programmer because the flash programmer does not support semihosting.

2.14 Coping with Low Amounts of RAM

If your target only has a small amount of RAM then you may find it difficult to fit the flash algorithm within it and for the Eclipse Flash Programmer plug-in or the test harness to allocate enough RAM for a buffer.

To cope with low amounts of RAM, you should first attempt to reduce the size of your flash algorithm. Switching to the microlib C library can save a lot of memory relative to the standard C library (see *Stack and Heap and C Libraries*). On processors that support both the ARM and

Thumb instruction sets, you may see a reduction in code size by changing the compiler settings to use the Thumb instruction set (ARM is the default). You can also increase the compiler optimization level for the `Debug` configuration (defaults to -O0), at the expense of debug visibility; the optimization level for the `Release` configuration is already -O2.

Many of the methods in the flash algorithm are optional, for example, you do not typically need to implement the `flashReadBlocks()` method for NOR flash devices because the Eclipse Flash Programmer plug-in can read the contents of flash directly. You should review the flash methods you have implemented in your flash algorithm and remove any that are not absolutely necessary. All the flash methods are defined as weak, so that you can remove them completely without generating linker errors.

Assuming the flash algorithm fits into RAM, there still may not be enough RAM available to allocate a buffer big enough to hold an entire flash block. In this case, the Eclipse Flash Programmer plug-in and the test harness attempt to allocate a smaller buffer and use sub-block accesses (see *Buffering* on page 6).

2.15 Interrupts

The Eclipse Flash Programmer plug-in disables interrupts prior to calling any methods in a flash algorithm. This is because if interrupt handlers are located in flash memory, then erasing or writing to the flash device can cause problems to occur if the interrupt handlers are called.

If your flash algorithm requires interrupts to be enabled and you are sure that your interrupt handlers are not located in flash then you must re-enable interrupts in each of your flash methods that require it.

2.16 Watchdog Timers

Some target hardware contains a watchdog timer that causes the system to be reset if the watchdog timer is not poked periodically. If your target hardware contains a watchdog timer then you must either disable it during flash operations or modify your flash algorithm to poke it periodically.

If your watchdog timer can be disabled then add the necessary code to the `flashSelect()` method. You can optionally add code to the `flashDeselect()` method to re-enable your watchdog timer after flash operations are complete.

Alternatively, insert code in your flash algorithm to poke the watchdog timer on each loop iteration. The main loops in your flash algorithm are likely to be in the methods that implement erase, read, write and verify operations.

3 Developing a New Flash Algorithm

This section describes how to create a new flash algorithm project in the Eclipse IDE, test and debug the algorithm in RVD and then import the finished algorithm into the Eclipse Flash Programmer plug-in.

3.1 Creating a new Flash Algorithm Project

To create a new flash algorithm project in the Eclipse IDE:

1. Select **File → New → Project...** from the Eclipse main menu. The **New Project** dialog opens.
2. Select **New Flash Device Project for ARM RVDS** in the **Flash Programmer** category in the list of project types (see Figure 1), and click **Next**.

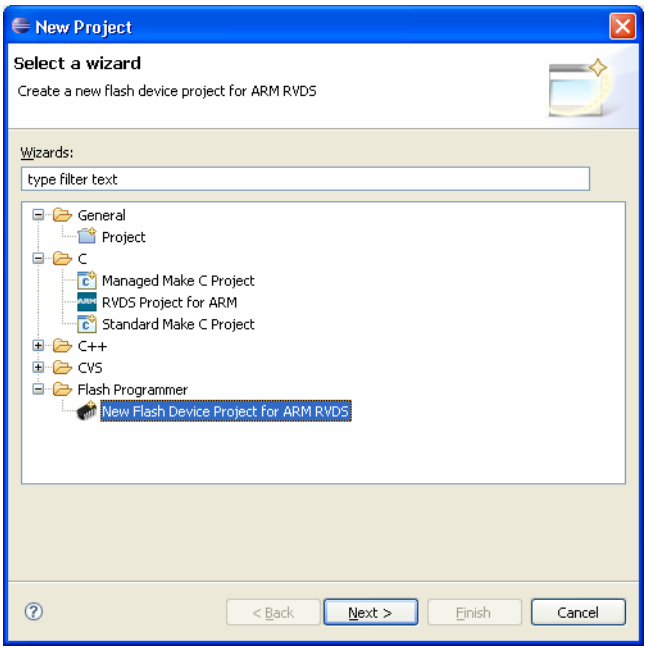

Figure 1 The New Project Wizard

3. Enter a name for your new project and click **Next**.
4. Enter a flash device name for your flash algorithm. The name is displayed by the Eclipse Flash Programmer plug-in when referring to your algorithm, so should choose a meaningful name.
5. Select the width of your flash device from the drop-down menu, little or big endian byte order, and the CPU that the algorithm is compiled for.
6. Enter the base address of your flash device and the start address of RAM on your target hardware (see Figure 2 on page 10). These values are used during testing and debugging of your flash algorithm. Values may be entered in decimal or hexadecimal (prefixed with '0x'). Click **Next**.

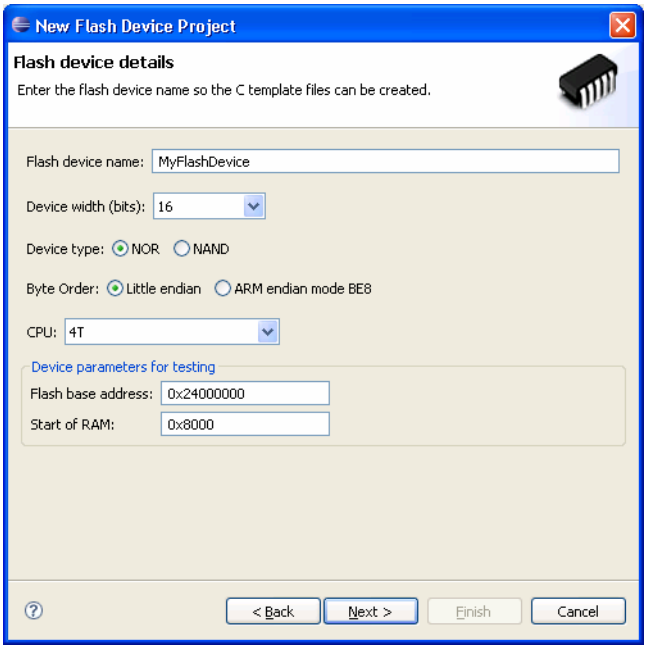

Figure 2 Configuring a new flash project

7. You can select which build configurations your flash project contains. The Debug configuration is set-up for debugging your flash algorithm. The Release configuration has compiler optimization enabled and is suitable for deploying your finished flash algorithm. It is recommended that you do not change these settings. Click **Finish**.

8. Your new flash algorithm project is inserted into your Eclipse workspace. The project contains a `.c` file with the flash device name that you entered into the wizard, which you must edit to implement your flash algorithm. The interface that your algorithm must implement is described in the `flashDevice.h` header file. The project also contains `flashDeviceTest.c`, which is a test harness that you can use to test and debug your flash algorithm.

3.2 Building a Flash Algorithm Project

Your new flash algorithm project builds automatically each time you save a source file if the **Build Automatically** option is checked in the **Project** menu. If this option is not checked, you can start a build by selecting **Project → Build Project** from the Eclipse main menu.

When you first create a new flash algorithm project the Debug configuration is selected by default. You should use the Debug configuration to test and debug your flash algorithm. When your algorithm is working correctly, switch to the Release configuration to build an optimized version of your algorithm, and use this build when importing the algorithm into the Eclipse Flash Programmer plug-in.

To change the build configuration:

1. Select your flash algorithm project in the **C/C++ Projects** pane.

2. Choose your desired build configuration from the **Project → Active Build Configuration** menu.

3.3 Editing the Flash Algorithm

Inside your flash algorithm project is a .c file with the flash device name that you gave when creating the project. You must edit this file to implement your flash algorithm, following the comments in this file and in the `flashDevice.h` header file and using the guidance elsewhere in this application note.

At a minimum you must implement the `flashGetBlockStructure()` and `flashWriteBlocks()` methods. Depending on your target hardware you may also need to implement the `flashSelect()` method, for example to configure access to your flash device. It is recommended that you implement and test these three methods first, before adding further functionality.

Other files in the project include:

- `flashDeviceTest.h`: Header file declaring the test harness. You do not need to modify this file.
- `flashDeviceTest.c`: Implementation of the test harness. You can step through this code in a debugger to debug your flash algorithm. There are some options near the top of this file that you can edit to change the behavior of the tests.
- `testMain.c`: Defines a `main()` function used when linking the `Debug` configuration of your flash algorithm. The `main()` function calls the test harness, passing in the address of your flash device. You do not need to modify this file.
- `dummyMain.c`: Defines a `main()` function used when linking the `Release` configuration of your flash algorithm. You do not need to modify this file.

3.4 Changing C Library

Flash algorithms can be built against either the standard C library or the microlib C library. The standard C library is selected by default, but you might want to use the microlib C library instead because it is smaller.

You can select a specific C library by adding one of `--library_type=standardlib` or `--library_type=microlib` to the **Extra options** box in the **ARM RealView Linker** section in the properties for your project. Although you can also specify the library type in the **ARM RealView Compiler** or **ARM RealView Assembler** sections of the project properties, the Eclipse Flash Programmer plug-in only uses the settings from the linker when importing flash algorithms.

If you choose to use the microlib C library, in addition to changing the library type used by the linker, you must also add code to your test project to initialize a stack and heap. You should add stack and heap initialization code to the `testMain.c` file so that it is only used when running the test harness. When your finished flash algorithm is imported and used from within Eclipse, a source file containing stack and heap initialization code is automatically created and linked into your algorithm by the Eclipse Flash Programmer plug-in if your algorithm uses the microlib C library.

For further information on both the standard and microlib C libraries and how to initialize the microlib C library, see the *RealView Compilation Tools Libraries and Floating Point Support Guide*. For further information on changing project properties, see the *RealView Development Suite Eclipse Plug-ins User Guide*.

3.5 Debugging and Testing your Flash Algorithm

The flash algorithm is a normal C program so it can be loaded and run in a debugger such as RVD. The Debug build configuration includes the `flashDeviceTest.c` test harness that you can use to debug and test your flash algorithm.

To load your flash algorithm into RVD:

1. In the **C/C++ Projects** pane, navigate to the build directory for your current build configuration, for example Debug, in your project, and select the `.axf` file that contains your flash algorithm.
2. Select **Run → Debug As → Load into RealView Debugger** from the Eclipse main menu. RVD starts.
3. Configure and connect to your target hardware in RVD. After connecting the image should load automatically. If the image does not load, repeat steps 1 and 2 whilst remaining connected to your target in RVD.

You can now run the executable in the RVD to test your flash algorithm. The test harness exercises all aspects of your flash algorithm and uses semihosting to output the results to the RVD console (see Figure 3).

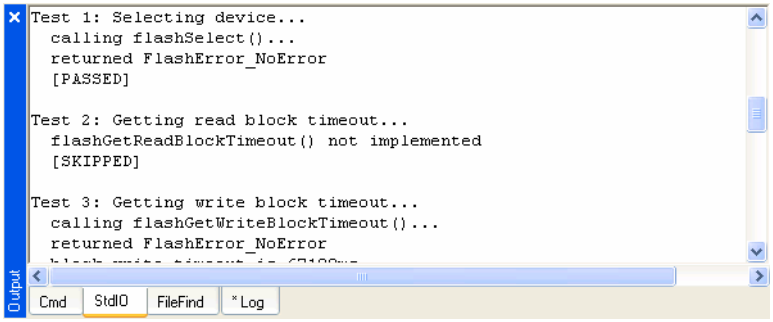

Figure 3 Test results in the RVD console

For each test the test harness outputs a short description of the purpose of the test, followed by some logging during the test, finally followed by the overall result. Results from the test harness are one of `PASSED`, `FAILED` or `SKIPPED`.

Tests are skipped if the test harness detects that the necessary functionality is not available in your flash algorithm. You should examine skipped tests to ensure that this outcome is consistent with what you have implemented.

Failed tests indicate a problem in your flash algorithm. You can use the log generated during the test to help diagnose the problem. If the built-in logging is insufficient, you can modify the test harness or your algorithm to include additional `printf()` statements. However, if you add `printf()` statements to your flash algorithm you must be careful to remove them before importing the algorithm into the Eclipse Flash Programmer plug-in, because the Eclipse Flash Programmer plug-in does not support semihosting. Alternatively you can rerun the tests and use the standard debugger facilities to step through the tests and examine variables and memory.

If you are unsure about some aspect of your flash device's operation, you can use the memory window in RVD to manually poke commands and data at the flash device and observe the results. If you do this you may find it necessary to edit the memory map in RVD so that correct sized memory accesses are used for your flash device. You can edit the memory map in RVD either using the Memory Map tab in the Process Control pane or by creating a *Board/Chip Definition* (BCD) file. See the RVD documentation for further details.

The test harness contains some extended tests that are disabled by default. These extended tests check that each bit in the flash device can be written as zero and written as one. You can enable these extended tests by uncommenting the `#define EXTENDED_TESTS` line near the top of the `flashDeviceTest.c` file. Enabling the extended tests increases the time taken to run the tests.

3.6 Importing Flash Algorithms into the Flash Programmer

After you have created your flash algorithm and are satisfied that it works correctly you can import the algorithm into the Eclipse Flash Programmer plug-in.

To import your algorithm:

1. Select **Target → Flash Device Manager** from the Eclipse main menu. The **Flash Device Manager** dialog opens (see Figure 4).

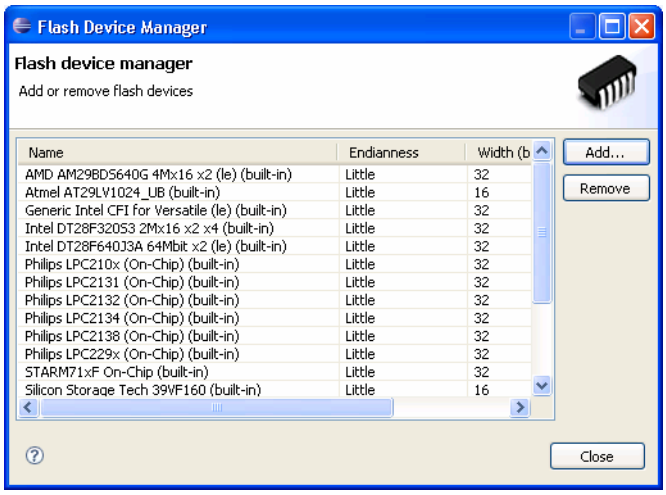

Figure 4 The Flash Device Manager dialog

2. Click **Add…** to open the **Import Flash Device** wizard.

3. Select the project containing your flash algorithm from the drop-down list (see Figure 5). Click **Next**.

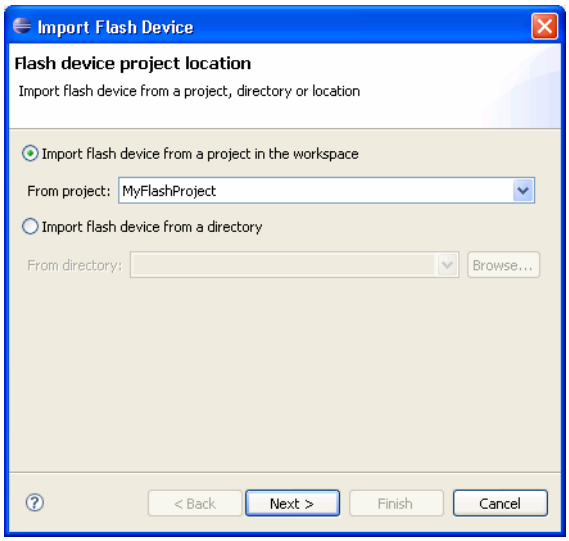

Figure 5 Importing a flash algorithm from a project

4. Select a build configuration from the drop-down list. It is recommended that you select the Release configuration, because this build has optimization enabled and results in faster flash operations. Whichever build configuration you select, make sure that you have built it recently so that it matches the current state of your source code.

5. Check the `.o` object file that contains your flash algorithm and uncheck any other `.o` files (see Figure 6). The `.o` file for your flash algorithm normally has the same name as you gave the flash device when creating the flash algorithm project. Click **Finish**.

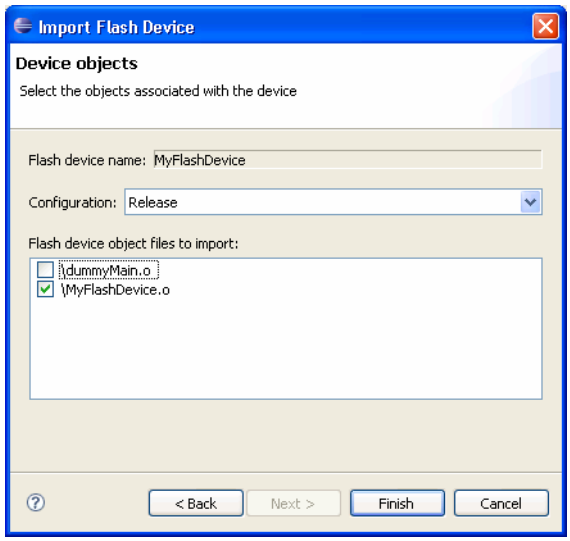

Figure 6 Selecting configuration and object files to import

6. The flash algorithm is imported and appears in the **Flash Device Manager** dialog. Click **Close**.

3.7 Creating a Flash Configuration

To use a flash algorithm in Eclipse you also need a flash configuration. Flash configurations supply additional parameters, such as the location and size of RAM on your target hardware, that are not algorithm specific. A flash configuration can contain multiple flash algorithms to cope with target hardware that contains multiple flash devices. There are built-in configurations for ARM's own boards. You must create a new flash configuration to use any imported flash algorithms.

To create a new flash configuration:

1. Select **Target → Manage Targets…** from the Eclipse main menu. The **Manage Targets** dialog opens.
2. Click the  button to create a new configuration.
3. Enter a name for your configuration, as well as the start of RAM and size of RAM on your target hardware. The flash algorithm is downloaded to RAM during flash operations, and the remainder of RAM is used for a stack and as a temporary buffer.
4. Click the **Add…** button to add your flash algorithm to the configuration.
5. Select your flash algorithm in the **Target details** table. The panel beneath the table changes to show any parameters needed by your flash algorithm.
6. Enter any parameters needed by your flash algorithm. All flash algorithms have a **Base address** parameter, which is the base address of your flash device in memory (see Figure 7 on page 15).

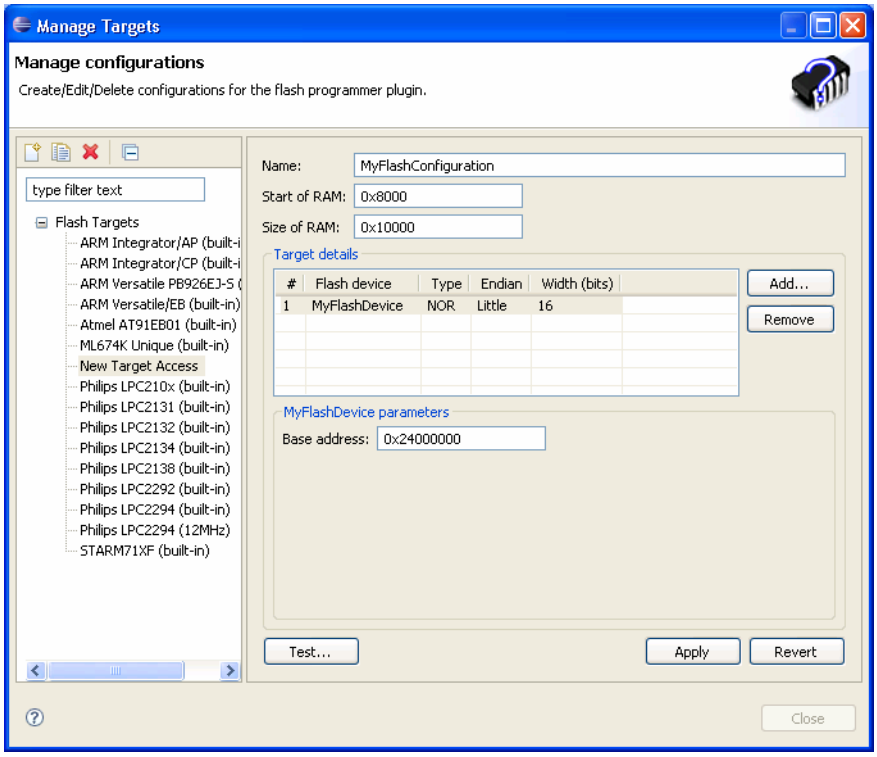

Figure 7 Creating a flash configuration

7. If your target hardware contains multiple flash devices you can add these to your configuration by repeating steps 4 to 6.

8. Click **Apply** to save your flash configuration.

3.8 Testing Imported Flash Algorithms

You can test imported flash algorithms from within Eclipse. The tests performed by Eclipse are similar to those that are present in the test harness in your flash algorithm project. The benefit of also running the Eclipse tests is that you can gain additional confidence that Eclipse has imported your flash algorithm correctly and is able to use it as expected.

To test an imported flash algorithm:

1. Select **Target → Manage Targets…** from the Eclipse main menu. The **Manage Targets** dialog opens.

2. Either select an existing flash configuration or create a new flash configuration (see *Creating a Flash Configuration* on page 14).

3. Click the **Test…** button. The **Test Target** dialog opens.

4. Select a connection method, and click **Configure…**. A configuration dialog opens that you must use to configure the connection to your target. After configuring your connection save and close this configuration dialog.

5. After closing the configuration dialog the **CPU** drop-down list in the **Test Target** dialog is populated with the available cores. Select the core that you want to connect to (see Figure 8 on page 16).

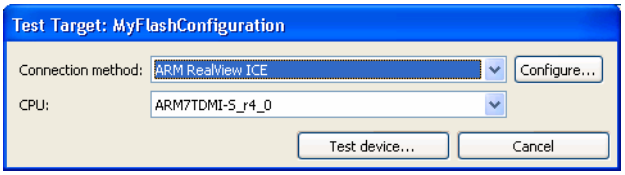

Figure 8 Configuring a test target

6. Click **Test device…** to begin testing your flash algorithm. During testing a progress dialog is shown and logging appears on the Eclipse console. Testing causes the current contents of the flash device to be lost.

7. The **Flash Programmer Results** dialog opens when testing is complete (see Figure 9). The results indicate whether the tests passed or failed and the time taken to perform various operations.

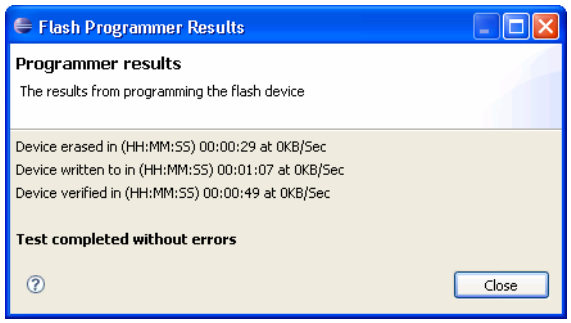

Figure 9 The tests results dialog

If a failure is indicated you must examine the log in the Eclipse console to determine the cause. Check that you have entered the correct parameters in your flash configuration. Check that your flash algorithm adheres to the instructions given in *Basic Principles* on page 4, such as not requiring a heap and coping with watchdog timers.

3.9 Using the Flash Programmer to Write an Image to Flash

After importing your flash algorithm into the Eclipse Flash Programmer plug-in you can use it to write images into flash.

To write an image contained within an open C/C++ project to flash:

1. In the **C/C++ Projects** pane, navigate to the image that you want to write and right-click on it to open the context menu.

2. Select **Send To → Send To…** from the menu. The **Send To** dialog opens (see Figure 10 on page 17).

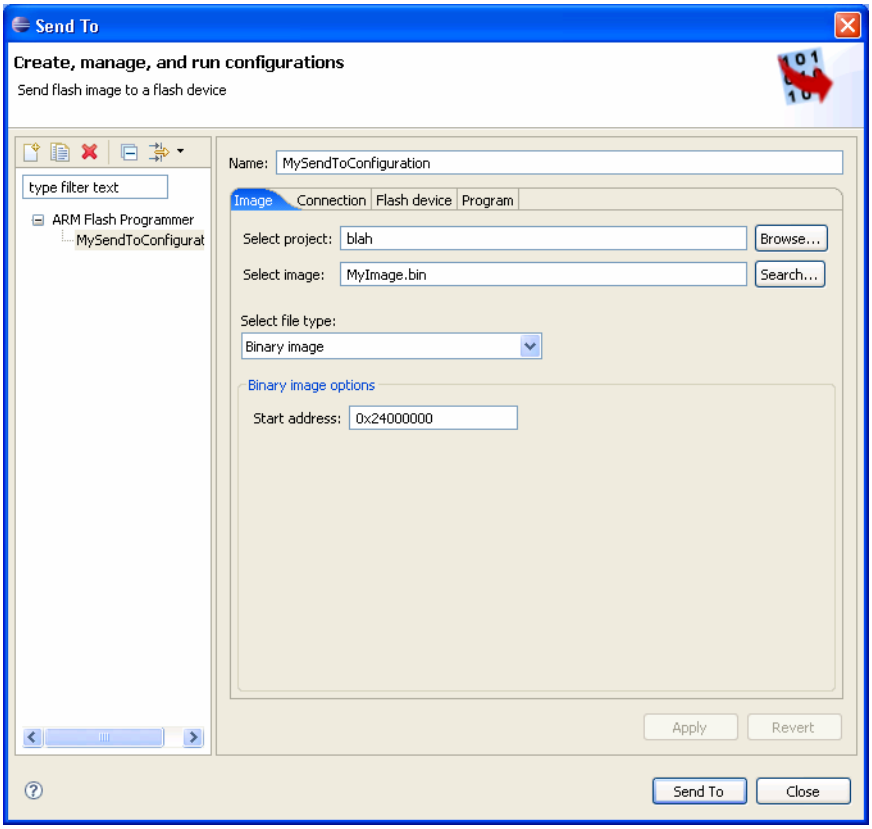

Figure 10 Editing a send to configuration

3. Either select an existing send to configuration or create a new one. To create a new send to configuration click the  button and then edit the **Name** field to give your configuration a name.
4. In the **Image** tab, configure the settings for your image.
5. In the **Connection** tab, configure a connection to your target hardware.
6. In the **Flash device** tab, select the flash configuration that contains your flash algorithm. To create a new flash configuration see *Creating a Flash Configuration* on page 14.
7. In the **Program** tab, select the erase method and whether a verify operation is performed.
8. Click **Apply** to save your send to configuration.
9. Click **Send To** to begin writing your image to flash.
10. The **Flash Programmer Results** dialog opens after programming is complete (see Figure 11 on page 18), indicating success or failure and the time taken for various operations.

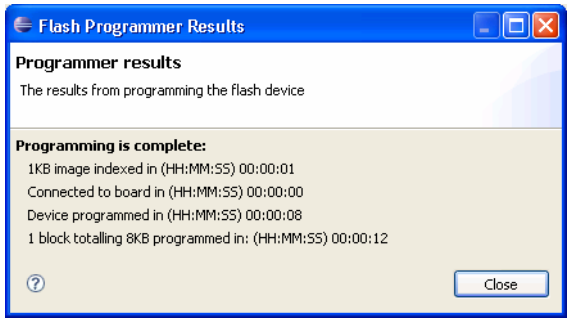

Figure 11 The Flash Programmer Results dialog

3.10 Exporting your Flash Algorithm to RVD

Flash algorithms created in Eclipse can be exported as a *Flash MEthod* (FME) file for use with RVD. You should ensure that your flash algorithm is fully working and tested in Eclipse before exporting it because it is difficult to diagnose FME problems in RVD.

There are some differences between the flash algorithms used by the Eclipse Flash Programmer plug-in that those used by RVD that you must be aware of:

- The `flashGetBlockStructure()` method in your Eclipse flash algorithm is not used by RVD. Instead RVD determines the block structure using data encoded in the FME file. When you export an Eclipse flash algorithm as an FME file you are prompted to enter the block structure of your flash device.
- Flash algorithms for Eclipse do not specify a buffer size for temporary data and instead leave it up to the Eclipse Flash Programmer plug-in to allocate a buffer using whatever RAM is available. In contrast, FME files encode the address and size of the buffer within themselves, and this makes it harder for a single FME file to work effectively across a range of different hardware targets.
- RVD does not support the `flashReadBlocks()` method, and instead reads the contents of flash directly from memory.
- RVD does not support timeouts on flash operations.
- To use an FME file with RVD you must write a *Board/Chip Definition* (BCD) file that references it, and configure RVD to load your BCD file.

To export a flash algorithm as an FME file:

1. Select **File → Export…** from the Eclipse main menu. The **Export** wizard opens (see Figure 12 on page 19).

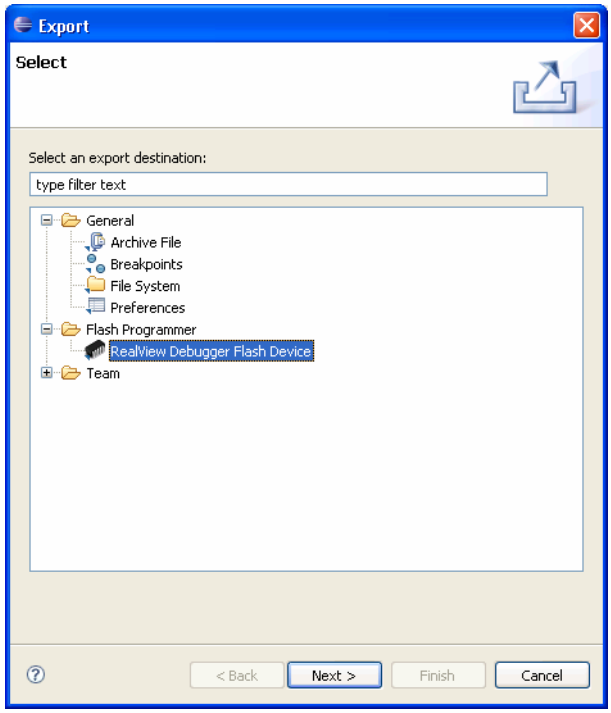

Figure 12 The Export Wizard

2. Select **RealView Debugger Flash Device** and click **Next**.

3. Select the flash device to export, either from an already installed device or from an Eclipse flash project (see Figure 13). Enter the name and location for the exported FME file. Click **Next**.

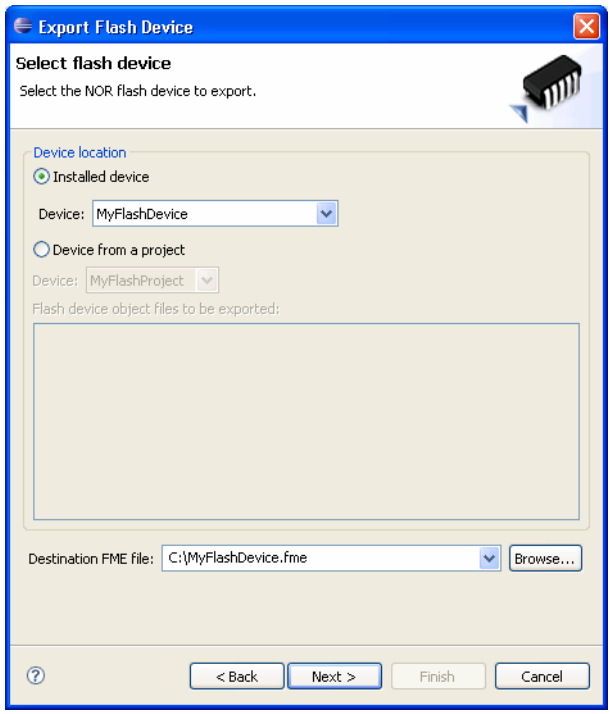

Figure 13 Selecting a flash device to export

4. Describe the block structure of the device. Click the **Add…** button to manually enter the block structure, or click the **Query device for structure…** button to attempt to automatically determine the block structure by connecting to your target hardware and running your Eclipse flash algorithm. Click **Next**.

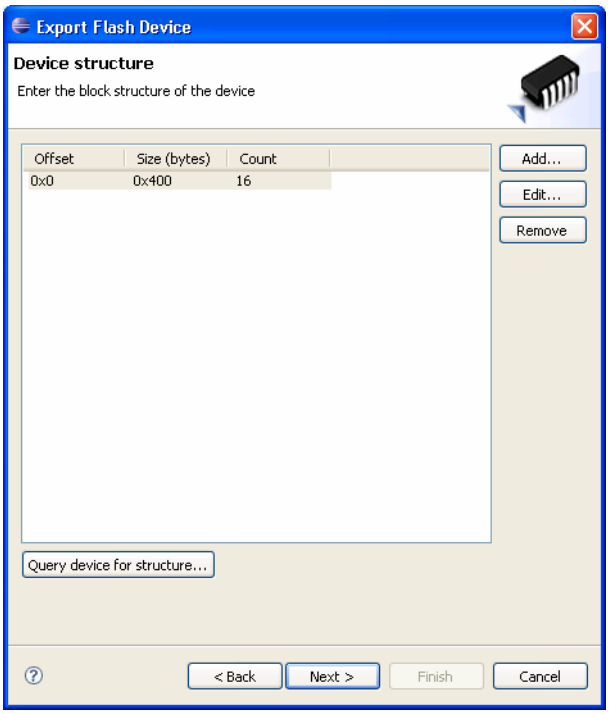

Figure 14 Describing the block structure of the device

5. Enter other parameters needed by the FME file, including the start address of RAM on your target hardware and the size of the buffer to be used during programming. For maximum programming performance, the buffer size should be as large as possible, within the limits of the RAM available on your target hardware and taking into account that the amount of available RAM is reduced by the code and any other data required by your flash algorithm. RVD places an upper limit of 65535 bytes on the buffer size.

   The **Needs clock** check box specifies whether the user is required to enter a clock speed parameter when using the FME file in RVD. Most flash algorithms do not need to know the clock speed of the target hardware, so **Needs clock** should be left unchecked. Flash algorithms for the Philips LPC2xxx microcontrollers are an example of where **Needs clock** must be checked.

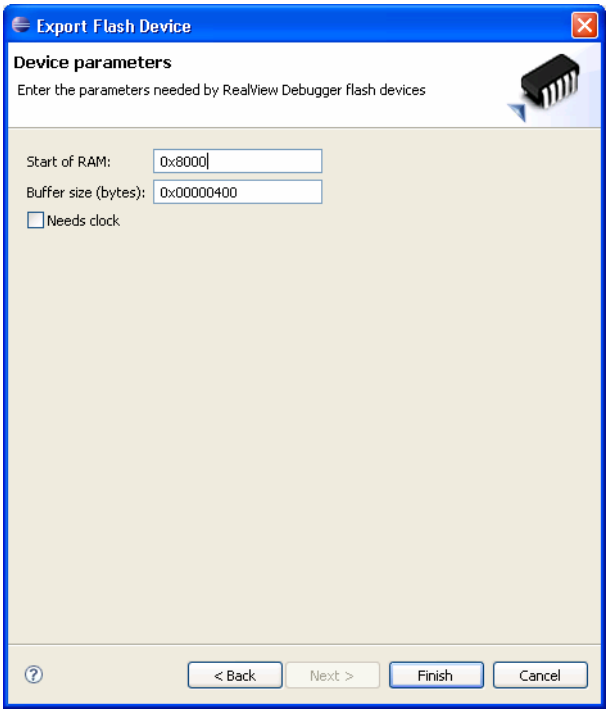

Figure 15 Entering other device parameters

6. Click **Finish**. The FME is written to the location you specified in step 3.

To use an exported FME file with RVD you must additionally write a BCD file describing the location of the FME file and where in memory the flash device exists.

Here is a minimal BCD file that you can copy:

```
[BOARD=MYBOARD]
Advanced_Information.ARM={\
    Memory_Block.MY_FLASH_DEVICE={\
        flash_type="C:/MyFlashDevice.fme"\
        :start=0x0\
        :length=0x4000\
        :access=Flash\
        :description="my flash device"\
    }\
}
```

You must edit the `start` and `length` to be the start address and length in bytes of your flash device and edit `flash_type` to be the full path to your exported FME file. You can optionally change the name of the BCD file by editing BOARD and give a description of your flash device using `description`.

To use the new BCD file copy it into the `etc` directory within your RVD installation, for example: `C:\Program Files\ARM\RVD\Core\3.1\881\win_32-pentium\etc`. You can then select the BCD file from within the RVD **Connection Properties** window using the name given on for `BOARD`. After connecting to your target hardware, you should see your flash device listed in the RVD **Memory Map** pane and should be able to write images to flash.

Further information on writing BCD files is available in the RVD documentation and on the ARM web site:

- *Application Note 142: Generating and Using BCD files with RVD*
  `http://www.arm.com/pdfs/AN142_BCD_files_RVD_revB.zip`

4 An Example Flash Algorithm

This application note is accompanied by an example flash algorithm for the built-in NOR flash on ARM's Versatile PB926EJ-S board. You can use this example as a guide when creating your own flash algorithm. The following sections explain the key points of the example algorithm.

4.1 Command Set

The flash devices used on the Versatile PB926EJ-S board use the Intel command set. The commands are defined using macros for ease of use throughout the source code.

4.2 Device Structure

The Versatile PB926EJ-S board contains two 16-bit flash devices in parallel. To achieve maximal performance the flash algorithm accesses both flash devices in parallel using 32-bit accesses. A macro, EXPAND, is used to convert 16-bit commands and data from the flash device data sheet into 32-bit values to be written to the bus:

```
#define EXPAND(value)
    (((uint32_t)(value)) | (((uint32_t)(value)) << 16))
```

When reading the flash device status register, the `EXPAND` macro is also used to convert the 16-bit expected values to 32-bits. This allows a direct comparison against the 32-bit values read from the bus.

When polling the flash device to wait for an operation to complete, care is taken to ensure that both devices are complete before proceeding. This is necessary because both flash devices may operate at slightly different speeds due to process variations and local temperature and voltage fluctuations.

The block structure of the flash device is determined at run-time by querying the CFI information block. The querying of the block structure is performed in the `flashSelect()` method, and the results are stored in a static array for later use in the `flashGetBlockStructure()` method. The array is static rather than dynamic because flash algorithms running in Eclipse cannot rely on the C library being initialized or on having a heap available. The array size is fixed at four; it is extremely unlikely that a flash block contains more than three block groups and one extra entry in the array is needed to terminate it.

The code that queries the block structure only uses the values from one flash device, even though the bus contains two devices in parallel. This is because the flash devices are assumed to use identical block structures, so it is easiest to query one device and multiply the block sizes by two. The flash programmer does not support parallel flash devices using different block structures.

4.3 Write-Protection

The Versatile PB926EJ-S board provides a configuration register, `SYS_FLASH`, which controls the write-protect pin on the flash devices, and the default state is with write-protection enabled. To allow erasing and writing to the flash device the `flashSelect()` method disables write-protection:

```
M32(SYS_FLASH) = M32(SYS_FLASH) | 1;
```

To prevent accidental modification of the flash device contents after flash operations are complete, the `flashDeselect()` method re-enables write-protection:

```
M32(SYS_FLASH) = M32(SYS_FLASH) & 0xFFFFFFFE;
```


This code in `flashSelect()` and `flashDeselect()` for writing to `SYS_FLASH` is the only part of the flash algorithm that is specific to the ARM Versatile PB926EJ-S board.

4.4 Timeouts

The timeouts for erasing a block and writing a block are determined by querying the CFI information block in the flash device. The CFI information block provides values for typical and worst-case operation. The timeouts returned to the flash programmer are those for worst-case operation.

The timeouts are read within the `flashSelect()` method and stored in global variables for later use by the `flashGetEraseBlockTimeout()` and `flashGetWriteBlockTimeout()` methods. The initial reading of the timeouts is located within the `flashSelect()` method because that method also contains other code for querying the CFI information block.

4.5 Erasing Blocks

The `flashEraseBlocks()` method is implemented to erase blocks. It iterates over the blocks, first unlocking them, then performing an erase, and then locking them. The unlocking is necessary because the default state of many flash devices is to have the blocks locked. The locking after the erase is optional, but is included to prevent accidental modification of the flash contents.

After erasing each block the flash device status register is checked for any errors. If an error is detected then the method returns immediately without attempting further erase operations.

The flash blocks are always left in read mode after erasing is complete. Care is taken to ensure that read mode is selected even in the case of an error being detected.

4.6 Writing Blocks

The `flashWriteBlocks()` method is implemented to write blocks. It iterates over the blocks, first unlocking them, then writing the buffer contents to the flash device, and then locking them. The unlocking is necessary because the default state of many flash devices is to have the blocks locked. The locking after the erase is optional, but is included to prevent accidental modification of the flash contents.

The writing code supports both word programming and buffer programming modes. Word programming mode is the simplest to implement and is supported by most CFI compatible flash devices. Buffer programming mode writes to a group of flash locations in parallel and results in faster programmer; however, it is more complex to implement and is not supported by all flash devices. The choice of programming method is determined in the `flashSelect()` method by querying the size of the write buffer in the CFI information block.

After writing each block the flash device status register is checked for any errors. If an error is detected then the method returns immediately without attempting further write operations.

The flash blocks are always left in read mode after writing is complete. Care is taken to ensure that read mode is selected even in the case of an error being detected.

4.7 Unimplemented Methods

The `flashReadBlocks()` and `flashVerifyBlocks()` methods are not implemented because the contents of the flash device can be read directly from memory by the Eclipse Flash Programmer plug-in. The `flashEraseDevice()` method is not implemented because full functionality is achieved with just the `flashEraseBlocks()` method and there is little performance advantage in this case in separately being able to erase the entire device.

These unimplemented methods have been completely removed from the algorithm to reduce the code size and to prevent the flash programmer from calling them unnecessarily.

