Freescale Semiconductor
Application Note

Document Number: AN4349

Adding Device(s) to the CodeWarrior™ Flash Programmer for Power Architecture™ Processors

by: Grigoras Marius - Viorel

1. Introduction

This document explains how to add support for additional flash devices on the Flash Programmer for CodeWarrior™ Development Studio for Power Architecture™ Processors by using the Flash Tool Kit. To add support for a new flash device, you need to write a new flash programming algorithm and create some supporting files.

This document explains:

- Creation of a flash device XML configuration file
- Creation of a new target task
- Creation of an external flash algorithm
- Flash programmer examples
- Creation of a new flash utility
- Flash utility examples
- Troubleshooting flash programmer

Contents

2. Preliminary Background

Before you program or erase any flash device, you must ensure that the CPU can access it. For example, you might need a different debug setup that requires modifications to the debugger configuration file. Consider the following before you begin:

- Read the flash device ID to verify correct connection and programmability. See Troubleshooting the Flash Programmer for instructions.
- Many manufacturers use the same flash-device algorithms, so it is likely that flashes can be programmed using algorithms included with the CodeWarrior software. In addition, many manufacturers produce devices compatible with Intel or AMD.
- Check whether a new flash device can be programmed with an algorithm already included with the CodeWarrior software, as described in Select Flash Programming Algorithm.
- Follow the steps provided in Creating an External Flash Algorithm if the flash device cannot be programmed with an existing algorithm.

3. Flash Tool Kit (FTK) Overview

To add support for a new flash device, you may need to create some or all of the following:

- An XML configuration file for the new device that describes organization, which is combination of device size, bus width, and number of devices present on the board
- An XML configuration file for the board that specifies the flash device it must use, and tells where the RAM memory is located
- A flash device algorithm if none of the existing algorithms are compatible

4. Creating Flash Device XML Configuration File

In its default configuration, the CodeWarrior™ Flash Programmer for Power Architecture supports many flash devices. The configuration files are located at:
`{CodeWarrior}\PA\bin\plugins\support\Products\ProductData\FPDevices.mwpdb\FP`

To add a new device to the CodeWarrior Flash Programmer, you must add a new file that describes the device.

Listing 1 shows the file format.

Listing 1. Generic flash device file format

```
<device-file>
<device>
<content>
   <name>NameOfFlashDevice</name>
   <manufacturerid>MfgID</manufacturerid>
   <chiperase>TRUE or FALSE</chiperase>
<sectors>
   <sectorcount>NumberOfSectors</sectorcount>
   <sectorsize>SectorSize</sectorsize>
.
.
   <sectorcount>NumberOfSectors</sectorcount>
   <sectorsize>SectorSize</sectorsize>
</sectors>
<organization>
        <name>Capacity/BusWidth/NumberOfDevices</name>
        <id>DeviceID_ForBusWidth</id>
        <algorithm>FlashAlgorithmForVariant</algorithm>
        <utility>FlashUtilityForVariant</utility>
</organization>
.
.
<organization>
        <name>Capacity/BusWidth/NumberOfDevices</name>
        <id>DeviceID_ForBusWidth</id>
        <algorithm>FlashAlgorithmForVariant</algorithm>
        <utility>FlashUtilityForVariant</utility>
</organization>
</content>
</device>
</device-file>
```

To add flash programming support for a new flash device:

1. Locate the data sheet for the new device and note the following information about the flash device:
    a) Device name
    b) Manufacturer ID code
    c) Device ID codes (8-bit, 16-bit)
    d) Number of sectors
    e) Starting and ending address for each sector
    f) Whether the device can be chip erased
    g) Options for data bits per device (8-bits, 16-bits)
    h) Number of flash devices on target
    i) Which device is most similar in the device configurations
2. Examine the installed devices for the most similar definitions.
3. Copy/edit the definition to make the XML device files conform to the new device.

4.1. Device Name

This is a free-form text field that describes the flash device, taken directly from the data sheet. Use only displayable ASCII characters with no spaces. Some examples, found in the configurations folder are: AM29BDD160GT, AM29LV640M, and IN28F128J3.

The format is:

```
        <name>NameOfFlashDevice</name>
```

4.2. Manufacturer ID and Device ID Codes

The Manufacturer ID and Device ID codes are read from the flash device after a specific sequence of writes to the flash device. Although the data sheet lists both IDs, only the Device ID varies among the flash devices from a given vendor, the Manufacturer ID remains the same. If the flash device supports more than one bus width (8-bit, 16-bit), then it might have different Device ID for each mode. For example, AM29LV160BB.

The formats are:

```
        <manufacturerid>MfgID</manufacturerid>

        <id>DeviceID_ForBusWidth</id>
```

4.3. Chip Erasing

Some devices can be completely erased with one chip erase command and this is much faster than erasing the device, sector by sector. Set the chip erase value to TRUE if your flash device supports this feature.

The format is:

```
        <chiperase>TRUE or FALSE</chiperase>
```

4.4. Number of Sectors and Sector Size

The data sheet lists the information about number of sectors and sector size. If the data sheet lists sector maps and tables for both 8-bit and 16-bit data options, use the 8-bit data option. The CodeWarrior flash programming algorithms require byte-level addresses for each sector. This constraint simplifies the design of the CodeWarrior flash programming interface for several data-bus configurations and sizes. When the data sheet does not provide a byte-level address, the algorithm creates an 8-bit sector map for 16-, 32-, or 64-bit devices. Table 1 shows an example of converting a 16-bit sector map to an 8-bit map.

The formats are:

```
<sectorcount>NumberOfSectors</sectorcount>

<sectorsize>SectorSize</sectorsize>
```

The `sectorcount` value is decimal while the `sectorsize` is hexadecimal.

For example, consider AM29BDD160GT. The device has eight (8) sectors of 0x2000 bytes, each followed by 30 sectors of 0x10000 bytes and another eight (8) sectors of 0x2000.

The configuration file will contain:

```
<sectors>

   <sectorcount>8</sectorcount>

   <sectorsize>2000</sectorsize>

   <sectorcount>30</sectorcount>

   <sectorsize>10000</sectorsize>

   <sectorcount>8</sectorcount>

   <sectorsize>2000</sectorsize>

</sectors>
```

Table 1. Sector Map Conversion

16-bit Sector Map (64K word sectors)	8-bit Sector Map (128Kbyte sectors)
000000..00FFFF	00000..01FFFF
010000..01FFFF	20000..03FFFF
020000..02FFFF	40000..05FFFF
030000..03FFFF	60000..07FFFF

Older flash devices can have sectors of different sizes. If you use such an older device, ensure that each sector in the configuration file is of the correct size.

4.5. Organization Name Options

The information that must be specified here as an organization name includes: device size, bus width, and number of devices present on the board.

Device size is the size of the device. It can be expressed as KB or MB using K and M suffixes. Examples: 128K, 1M.

Many flash devices can be set to use either 8-data bits or 16-data bits depending on the status of a configuration pin (typically named BYTE#) on each device. The <*organization*> field uses this part of the flash definition, as described in the next paragraph. Your target uses only one configuration so you need to support only that configuration. Expanding your new definition to include the other configurations for this device, however, is a good design practice.
Your target may use one, two, or four devices at the same base address to support an 8-bit, 16-bit, 32-bit, or 64-bit data bus.

For example, two 8-bit flash devices side by side support a 16-bit data bus, and four 16-bit devices support a 64-bit data bus. The <*organization*> field summarizes each possible combination of device capacity, bus width, and number of devices used.

For example, *4Mx16x1* means *4MegaHalfwords by 16 data bits per device by 1 flash device*, resulting in a total of *4M 16-bit half words*. Similarly, *1Mx8x4* means *1MegaByte by 8 data bits per device by 4 flash devices*, resulting in *1M 32-bit words* and a 32-bit data bus presented to the processor.

The format is:

```
<organization>

        <name>Capacity*BusWidth*NumberOfDevices</name>

.

.

</organization>
```

4.6. Find Most Similar Device

To find a device most similar to the one for which support is introduced, perform these steps:

1. From the data sheet for target flash devices, determine whether the bus width is 8- or 16- data bits.
2. Read through the files in the configuration folder of the CodeWarrior™ Development Studio for Power Architecture installation and scan for devices from the same manufacturer with similar part names.
   For example, *AM29LV640D* is similar to *AM29LV641DU*, and *IN28F128J3* is similar to *IN28F640J3*.
3. Manufacturers often base new designs on the architecture of previous designs to ensure that new devices are virtually the same as the previous devices. However, the new devices may have

greater capacity or improved programming features, such as timing and operation. This pattern simplifies flash programming because the flash programming algorithms remain unchanged. Yet only the device names, sectors, and Device IDs change.

4. Open the *IN28F640J3.xml* file in a text editor and compare the entries with the ones in *IN28F128J3.xml*.

   For example, see how the latter was built as an extension of the former. Note also how the part number of your device may be only a revision letter different from a defined part.

   For example, the flash programmer considers the *AM29DL640B* to be the same as the *AM29DL640D* and the *AM29DL640G*. Thus, if you use a part number like this, program the flash programmer to use the defined part and you will not need to create a new file.

The format is:

```
<algorithm>FlashAlgorithmForVariant</algorithm>
```

FlashAlgorithmForVariant is the algorithm name without full path (just the .elf file name).

4.7. Select Flash Programming Algorithm

Flash programming algorithms differ depending on the flash manufacturer, bits per device organization, and the number of the flash devices used. The CodeWarrior Flash Programmer supports a number of algorithms that are already compiled *.elf executables. These files can be found at:

{CodeWarrior}\PA\bin\plugins\support\Flash_Programmer\EPPC

Create an algorithm file name by combining the fields: manufacturer, data bits per device, and number of flash devices. For example, the flash algorithm for two AMD29LV320MB devices, used in their 16-bit mode (BYTE# = 1), is amd16x2.elf.

The CodeWarrior™ Development Studio for Power Architecture has built-in flash programming algorithm support for AMD and Intel flash devices. If the device does not have built-in algorithm support, you can create your own algorithm and use it with the CodeWarrior Flash Programmer. For more information, see Creating an External Flash Algorithm.

4.7.1. AMD-Based or Spansion-Based Flash Programming Algorithms

AMD-based or Spansion-based devices use two types of flash programming algorithms: common and alternative.

If the flash memory device supports two types of connections, 8-bit or byte connection and 16-bit or word connection, then use an alternative algorithm.

In all other cases or for AMD flash devices that do not support two types of connections, use the common AMD algorithm (Table 2).

Flash command register addresses are the main difference between common and alternative algorithms. For example, command addresses for the common flash algorithm are: 0x555, 0x2aa, 0x555, whereas for alternative connection, these addresses are: 0xaaa, 0x555, 0xaaa.

Table 2. AMD Algorithms

Algorithm	Device(s)	Address Used	Algorithm File Name
AMD	One device that supports only 8-bit bus connection	0x555, 0x2aa, 0x555	amd8x1.elf
AMD	One device that supports both 8-bit and 16-bit bus connections in 8-bit mode	0x555, 0x2aa, 0x555	amd8x1alt.elf
AMD	Two devices that support only 8-bit bus connection	0x555, 0x2aa, 0x555	amd8x2.elf
AMD	Two devices that support both 8-bit and 16-bit bus connections in 8-bit mode	0xaaa, 0x555, 0xaaa	amd8x2alt.elf
AMD	Four devices that support only 8-bit bus connection	0x555, 0x2aa, 0x555	amd8x4.elf
AMD	Four devices that support both 8-bit and 16-bit bus connections in 8-bit mode	0xaaa, 0x555, 0xaaa	amd8x4alt.elf
AMD	One device that supports only 16-bit bus connection; one device that supports both 8-bit and 16-bit bus connections in 16-bit mode	0x555, 0x2aa, 0x555	amd16x1.elf
AMD	Two devices that support only 16-bit bus connection; two devices that support both 8-bit and 16-bit bus connections in 16-bit mode	0x555, 0x2aa, 0x555	amd16x2.elf

4.7.2. Intel-Based Flash Programming Algorithms

Support for Intel devices (Table 3) includes three types of flash programming algorithms:

- C3 – For Intel Advanced + Boot Block (C3)
- J3 – For Intel embedded flash memory (J3) and algorithms for Boot Block flash memory
- B3 – Advanced Boot Block (B3) flash memory families

Algorithm packages are written to comply with the J3, C3, or B3 data sheet documented functionality from Intel for each function: Read, Write, Erase, and ID checking.

Table 3. Intel Algorithms

Algorithm	Device(s)	Algorithm File Name
Intel	One C3 Intel flash device with 8-bit data connection	Intel8x1c3.elf
Intel	One J3 Intel flash device with 8-bit data	Intel8x1j3.elf

Algorithm	Device(s)	Algorithm File Name
	connection	
Intel	One boot flash memory device or advanced boot (B3) flash memory device with 8-bit data connection	Intel8x1.elf
Intel	Two C3 Intel flash devices with 8-bit data connection	Intel8x2c3.elf
Intel	Two J3 Intel flash devices with 8-bit data connection	Intel8x2j3.elf
Intel	Two boot flash memory devices or advanced boot (B3) flash memory devices with 8-bit data connection	Intel8x2.elf
Intel	Four C3 Intel flash devices with 8-bit data connection	Intel8x4c3.elf
Intel	Four J3 Intel flash devices with 8-bit data connection	Intel8x4j3.elf
Intel	Four boot flash memory devices or advanced boot (B3) flash memory devices with 8-bit data connection	Intel8x4.elf
Intel	One C3 Intel flash device with 16-bit data connection	Intel16x1c3.elf
Intel	One J3 Intel flash device with 16-bit data connection	Intel16x1j3.elf
Intel	One boot flash memory device or advanced boot (B3) flash memory device with 16-bit data connection	Intel16x1.elf
Intel	Two C3 Intel flash devices with 16-bit data connection	Intel16x2c3.elf
Intel	Two J3 Intel flash devices with 16-bit data connection	Intel16x2j3.elf
Intel	Two boot flash memory devices or advanced boot (B3) flash memory devices with 16-bit data connection	Intel16x2.elf

4.7.3. Flash Manufacturers Algorithms

Many manufacturers use flash device programming algorithms that are not bundled with their own devices. In many cases, these algorithms are same across multiple manufacturers. For example, AMIC 16x1 and AMD 16x1 flashes are programmed using the same algorithms.

Table 4 lists algorithms, device compatibility, and other information for flash manufacturers.

Table 4. Flash Manufacturers

Manufacturer	Algorithm	Comments
Alliance	Flash programming algorithms used for AMD (Spansion) should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.alsc.com/
AMD	Algorithms are supported in the CodeWarrior Flash Programmer	AMD does not produce its own flash devices any more – founder of the Spansion Company. Manufacturer's site: http://www.spansion.com/
AMIC	Depending on the particular flash device for flash programming, the same flash programming algorithms used for AMD (Spansion) or Atmel should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.amictechnology.com/
Atmel	Algorithms are supported in the CodeWarrior Flash Programmer	Manufacturer's site: http://www.atmel.com/
Catalyst	Flash programming algorithms used for Intel should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.catsemi.com/index.html Most of the flash devices from Catalyst are identical to the flash devices from Intel. For example, the CAT28F001 from Catalyst is the same as Intel E28F001.
EON	Flash programming algorithms used for AMD (Spansion) should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.eonsdi.com/ Most of the flash devices from EON have direct references to the AMD flash devices.
Fujitsu	Flash programming algorithms used for AMD (Spansion) should be usable (check flash device specification from manufacturer).	Fujitsu no longer produces its own flash devices – founder of the Spansion Company. Manufacturer's site: http://www.spansion.com/
Hyundai	Flash programming algorithms used for AMD (Spansion) should be usable (check flash device specification from manufacturer).	Hyundai founded new company for semiconductors, named Hynix. Manufacturer's site: http://www.hynix.com Most of the flash devices from Hynix have direct references to the AMD flash devices.
Intel	Algorithms are supported in the CodeWarrior Flash Programmer	Manufacturer's site: http://www.intel.com/
Micron	Flash programming algorithms used for Intel should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.micron.com/ Most of the flash devices from Micron have direct references to the Intel flashes.
MXIC	Flash programming algorithms used for AMD (Spansion) should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.mxic.com.tw Most of the flash devices from MXIC have direct references to the AMD flash devices.
Samsung	Flash programming algorithms are not supported in the CodeWarrior Flash Programmer.	Manufacturer's site: www.samsung.com/products/semiconductor/OneNAND Samsung uses its own algorithm for flash programming, not compatible with other vendors.
Sharp	Flash programming algorithms used for Intel should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.sharpsma.com

Manufacturer	Algorithm	Comments
Spansion	Algorithms are already supported in the CodeWarrior Flash Programmer.	Manufacturer's site: http://www.spansion.com/
SST	Depending on the particular flash device used for flash programming, the same flash programming algorithms used for AMD (Spansion), AMD, or Intel should be usable (check flash device specification from manufacturer).	Produces flash devices compatible with Intel, AMD, and Atmel Manufacturer's site: http://www.sst.com/about/
ST	Flash programming algorithms used for AMD (Spansion) should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.st.com
Toshiba	Flash programming algorithms used for Intel should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.semicon.toshiba.co.jp/eng
White	Flash programming algorithms used for AMD (Spansion) should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.wedc.com/
Winbond	Flash programming algorithms used for AMD (Spansion) should be usable (check flash device specification from manufacturer).	Manufacturer's site: http://www.winbond-usa.com/mambo/content/ view/289/553/

4.8. Set Verify Type

The verify operation can be done in two ways: on target and on host. If the verify operation is done on host, the CodeWarrior software reads data from the target and compares it to the one that was recently programmed. When the operation is done on target, a flash utility and the data to be verified will be downloaded.
The format is:

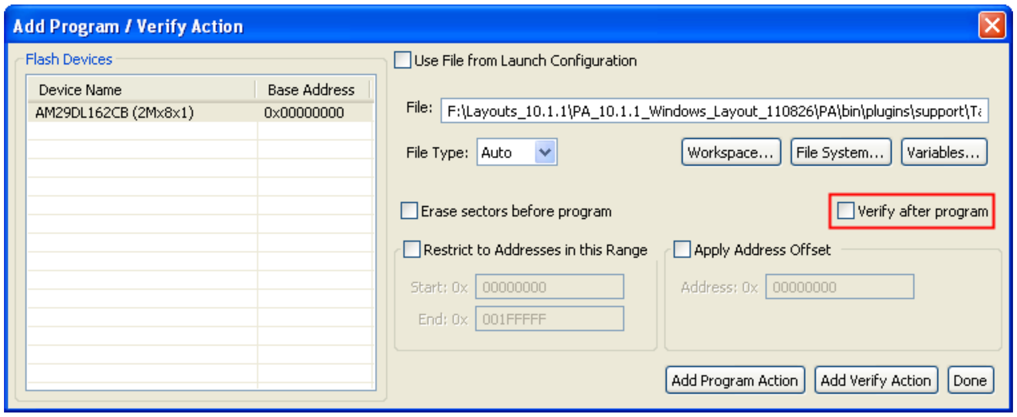
```
<ontargetverify>TRUE or FALSE</ontargetverify>
```

4.9. Set Verify after Program option

The Verify after Program option is intended to improve the user experience by reducing the time needed to program and verify a file on the target board. Currently, the program and verify operations are performed separately. The file is downloaded to the target for each operation (except NOR, where the verify operation is made on the host itself), which has a significant impact on the run time. To combine both the operations, perform the following steps:

1. Add `<verifyafterprogram>` tag in all the XML Flash Programmable devices.
2. The **Verify after program** check box appears in the **Add Program / Verify Action** dialog box. It indicates that a verify operation is combined with the program operation for the file. Ensure to select **Verify after program** check box.

Figure 1. Add Program / Verify Action dialog box



Add Program / Verify Action

Flash Devices

Device Name	Base Address
AM29DL162CB (2Mx8x1)	0x00000000

☐ Use File from Launch Configuration

File: F:\Layouts_10.1.1\PA_10.1.1_Windows_Layout_110826\PA\bin\plugins\support\Ta

File Type: Auto

☐ Erase sectors before program

☐ Verify after program

☐ Restrict to Addresses in this Range

Start: 0x 00000000

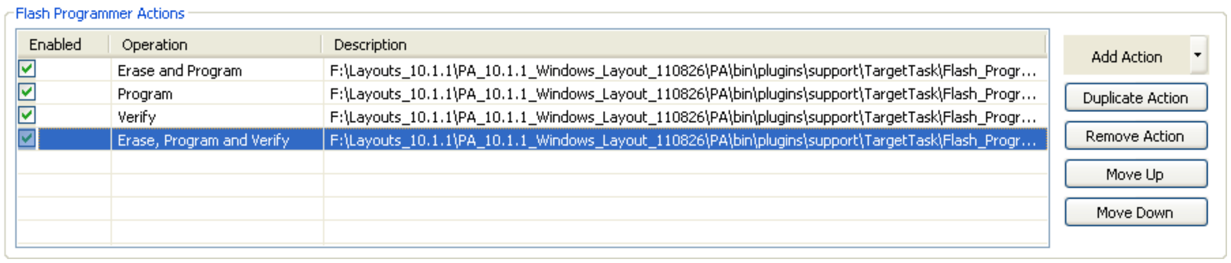
End: 0x 001FFFFF

☐ Apply Address Offset

Address: 0x 00000000

3. Click **Done** to close the **Add Program / Verify Action** dialog box.
The new operation gets listed under the **Operation** list of the target task window.

Figure 2. Target Task Window – Flash Programmer Actions Panel



Flash Programmer Actions

Enabled	Operation	Description
✓	Erase and Program	F:\Layouts_10.1.1\PA_10.1.1_Windows_Layout_110826\PA\bin\plugins\support\TargetTask\Flash_Progr...
✓	Program	F:\Layouts_10.1.1\PA_10.1.1_Windows_Layout_110826\PA\bin\plugins\support\TargetTask\Flash_Progr...
✓	Verify	F:\Layouts_10.1.1\PA_10.1.1_Windows_Layout_110826\PA\bin\plugins\support\TargetTask\Flash_Progr...
✓	Erase, Program and Verify	F:\Layouts_10.1.1\PA_10.1.1_Windows_Layout_110826\PA\bin\plugins\support\TargetTask\Flash_Progr...

4.10. Select Flash Utility

The flash algorithms are used for erase and program operations. Blank check, checksum, and sometimes verify operations (depending on the value of `<ontargetverify>`) are done with another program, called flash utility. If the flash device is memory mapped (NOR type), then we can use the default `FlashUtility.elf`. This provides support only for blank check and checksum. The verify operation must be done on host, so `<ontargetverify>` should be set to FALSE. If we have a NAND or SPI device, then a special utility must be written. The format is:

```
<utility>FlashUtilityForVariant</utility>
```

4.11. Add Flash Device in Database

All flash devices are kept in a common database. When a flash device is added from the flash programmer user interface, it reads the database and displays all devices found. To ensure that the devices appear correctly, perform these steps:

1. Add the file in database
2. Change the manifest that specifies which devices exist
3. The device configuration file must be copied in

{CodeWarrior}\PA\bin\plugins\support\Products\ProductData\FPDevices.mwpdb\FP. For this example, assume that the name is `NewFlashDevice.xml`.

4. Change the manifest file – {CodeWarrior}\PA\bin\plugins\support\Products\ProductData\FPDevices.mwpdb\product-manifest.xml.
5. Add a new section in the `<device>` tag that specifies a new file exists.

Listing 2 shows the beginning of the manifest file.

Listing 2. Beginning of product-manifest.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE product-manifest>
<product-manifest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.metrowerks.com/schemas/2003/IDE/ProductManifest.xsd
">
  <product-description>
    <name>FP</name>
    <common-product>FP_EPPC</common-product>
    <version>1.0</version>
  </product-description>

  <product-files>
    <product>
      <file>
        <name>FP_EPPC</name>
        <version>1.0</version>
        <path>FPDevProductData.xml</path>
      </file>
    </product>
    <device>
      <file>
        <name>AM29BDD160GT</name>
        <version>0</version>
        <path>FP/AM29BDD160GT.xml</path>
       </file>
```

6. Add the new file anywhere in `<device>` tag. For this example, it will be added at the beginning.

Listing 3 shows the new entry marked in bold.

Listing 3. New Entry Marked in Bold

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE product-manifest>
```

```
<product-manifest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.metrowerks.com/schemas/2003/IDE/ProductManifest.xsd
">
<product-description>
    <name>FP</name>
    <common-product>FP_EPPC</common-product>
    <version>1.0</version>
  </product-description>

  <product-files>
    <product>
      <file>
        <name>FP_EPPC</name>
        <version>1.0</version>
        <path>FPDevProductData.xml</path>
      </file>
    </product>
    <device>
      <file>
        <name>NewFlashDevice</name>
        <version>0</version>
        <path>FP/NewFlashDevice.xml</path>
      </file>
      <file>
        <name>AM29BDD160GT</name>
        <version>0</version>
        <path>FP/AM29BDD160GT.xml</path>
      </file>
```

NOTE It is highly recommended to back up the manifest file before updating it. If an error occurs, the CodeWarrior software may not be able to parse some devices.

5. Creating a New Target Task

All Flash operations run through the Target Task Framework. To open the **Target Tasks** view:

1. Select **Window > Show View > Other** from the IDE menu bar. The **Show View** dialog box appears.
2. Select **Debug > Target Tasks**. The **Target Tasks** view appears.

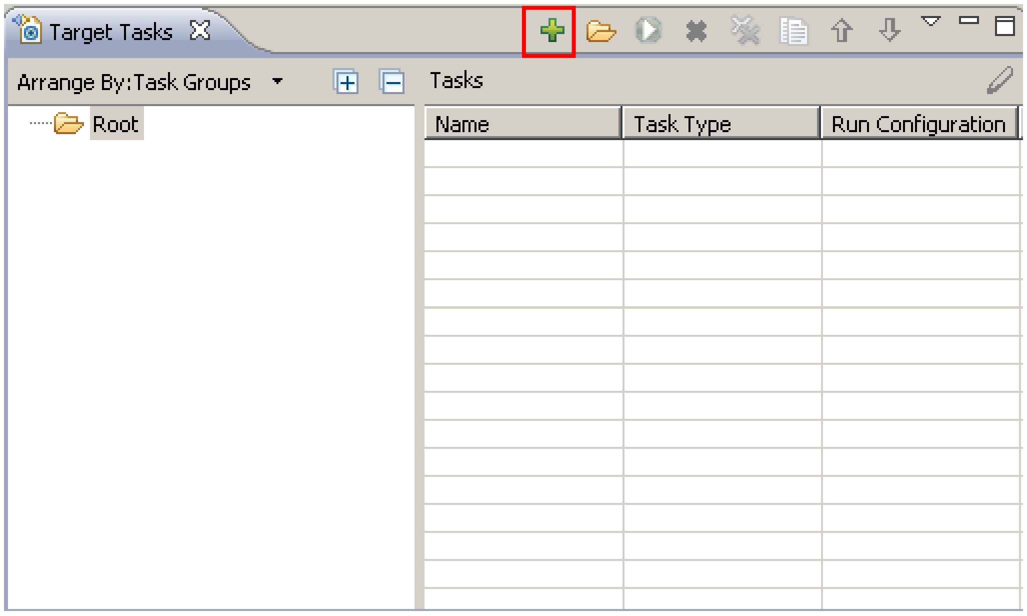
Next, create a Flash Programmer task representing the starting point for any flash operation. This defines the flash device to be used and the memory buffer, and creates some default actions.

5.1. Create a New Target Task

To create a new task, perform these steps:

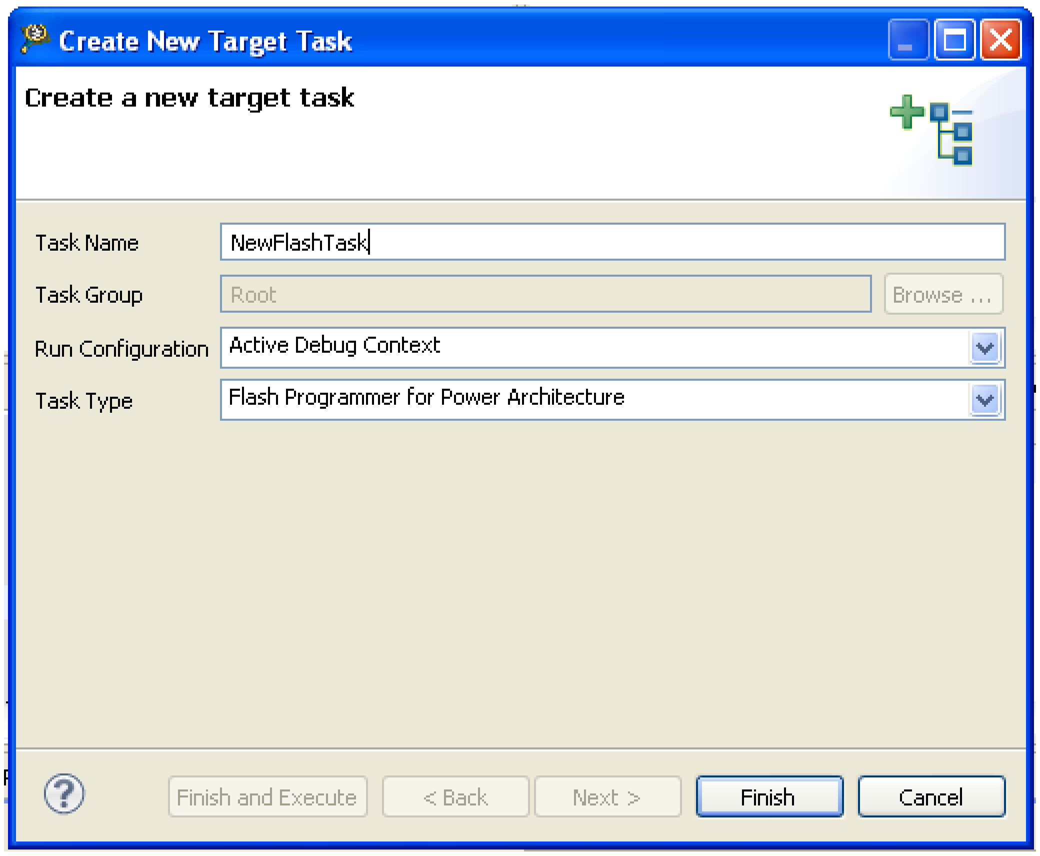
1. Click the **Create new target task** (“+”) icon on the **Target Tasks** view toolbar, as shown in Figure 3.

Figure 3. Create a New Target Task Icon



2. The **Create New Target Task** wizard appears (Figure 4). Specify information in the following fields:
   - **Task Name:** Name of the target task.
   - **Task Group:** Group where the task is to be created. If only Root exists this option is disabled.
   - **Run Configuration:** Each task must be associated with an existing Launch Configuration or Active Debug Context. This association is required to be able to make a connection to the target when doing operations over the flash. **Active Debug Context** means a connection is already established and only the task needs to be executed. Use Active Debug Context for generic tasks or when it is not known which Launch Configurations are available.
   - **Task Type:** Type of task created. For Power Architecture, select *Flash Programmer for Power Architecture*.

Figure 4. Create New Target Task Wizard



Task Name: NewFlashTask
Task Group: Root
Run Configuration: Active Debug Context
Task Type: Flash Programmer for Power Architecture

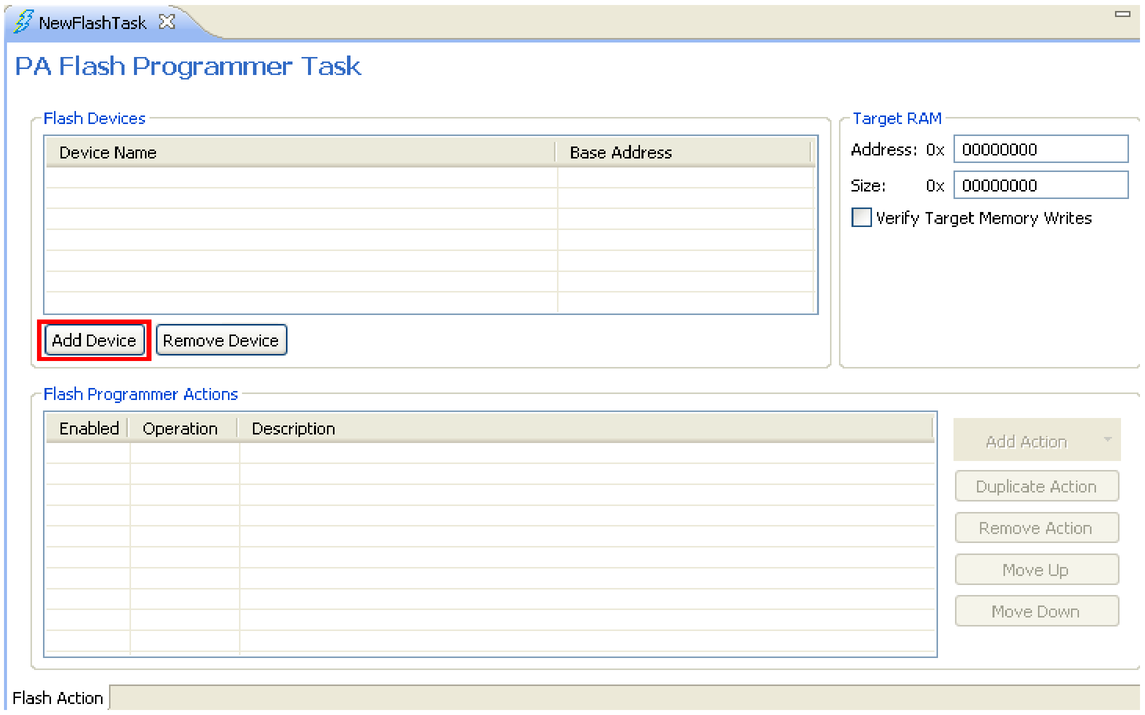
3. Click **Finish**. The editor for the new task appears.

5.2. Add a Device

To add a flash device to the Flash Devices table in the Flash Programmer Task editor, perform these steps:

1. Select the Flash Programmer task to which you want to add a Flash device.
2. Click **Add Device** in the Flash Programmer Task editor (Figure 5).

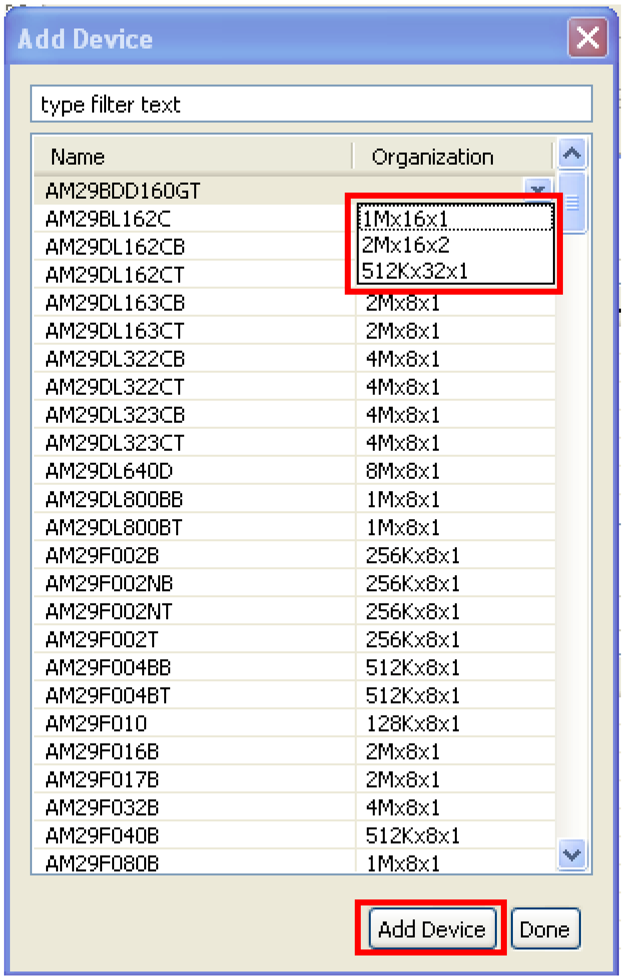
Figure 5. Add Device in Flash Programmer Task Editor



The **Add Device** dialog box appears (Figure 6).

3. Select the new device, named *NewFlashDevice* in this example, and check the available organizations. Here, AMD16x1 is defined. If more than one device is present, then select the devices that suit the board where the task will be executed.
4. Click **Add Device**. Both of them are shown in Figure 6.

Figure 6. Select Organization and Add Device Button



Name	Organization
AM29BDD160GT	
AM29BL162C	
AM29DL162CB	
AM29DL162CT	
AM29DL163CB	2Mx8x1
AM29DL163CT	2Mx8x1
AM29DL322CB	4Mx8x1
AM29DL322CT	4Mx8x1
AM29DL323CB	4Mx8x1
AM29DL323CT	4Mx8x1
AM29DL640D	8Mx8x1
AM29DL800BB	1Mx8x1
AM29DL800BT	1Mx8x1
AM29F002B	256Kx8x1
AM29F002NB	256Kx8x1
AM29F002NT	256Kx8x1
AM29F002T	256Kx8x1
AM29F004BB	512Kx8x1
AM29F004BT	512Kx8x1
AM29F010	128Kx8x1
AM29F016B	2Mx8x1
AM29F017B	2Mx8x1
AM29F032B	4Mx8x1
AM29F040B	512Kx8x1
AM29F080B	1Mx8x1

A pop-up window displays indicating that the device has been added.

5. Click **Done**. The selected devices are added in the **Flash Devices** table.

5.3. Populate Default Values

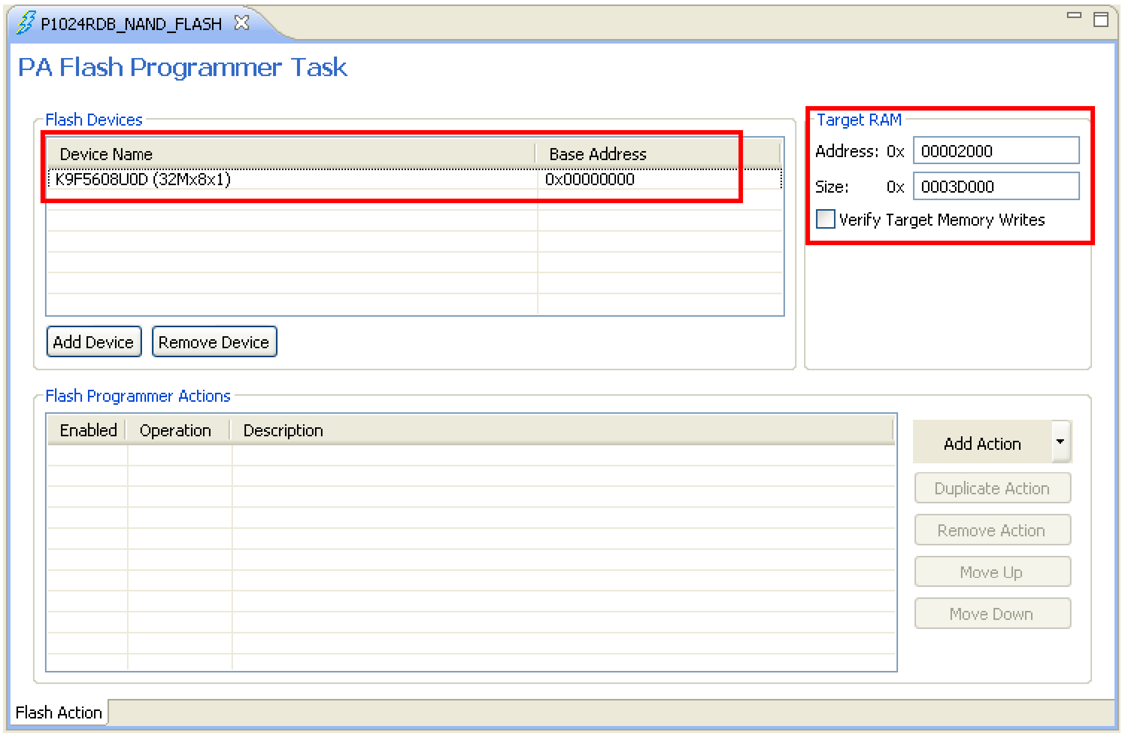
Ensure that the following default values are populated correctly for the flash devices specified in the target task:

- **Base Address:** Specifies that the flash device is memory mapped and this address must be defined in the “Flash Devices” table next to device name.
- **Address** in Target Ram panel**:** Specifies the start address of the memory where an algorithm is downloaded on the target for performing operations on the flash devices.
- **Size:** Specifies size of the memory buffer for algorithm. The size must be large enough to fit the algorithm and data that must be programmed. In case the buffer is not big enough, an error will be displayed when executing the task. The smallest size needed is specified in the Size field.

- **Verify Target Writes:** Checks if the memory is correctly written. This is done by reading the memory written after each write command. This allows you to check if the RAM memory is correctly initialized. By default, it comes unchecked due to the loss of speed that comes with the overhead of reading memory each time.

All these values must be correct for the board where the flash device is located. Figure 7 shows the default values for board P1024RDB with K9F5608U0D NAND flash device. All fields that must be filled are highlighted with red.

Figure 7. Populate Default Values



Device Name	Base Address
K9F5608U0D (32Mx8x1)	0x00000000

Address: 0x 00002000

Size: 0x 0003D000

☐ Verify Target Memory Writes

5.4. Create Default Actions

The various flash programmer actions that can be added to a target task are:

- Erasing the whole flash device using Chip Erase
- Blank checking the whole device
- Programming the file from Launch Configuration used to connect to the target
- Verifying the file from Launch Configuration used to connect to the target

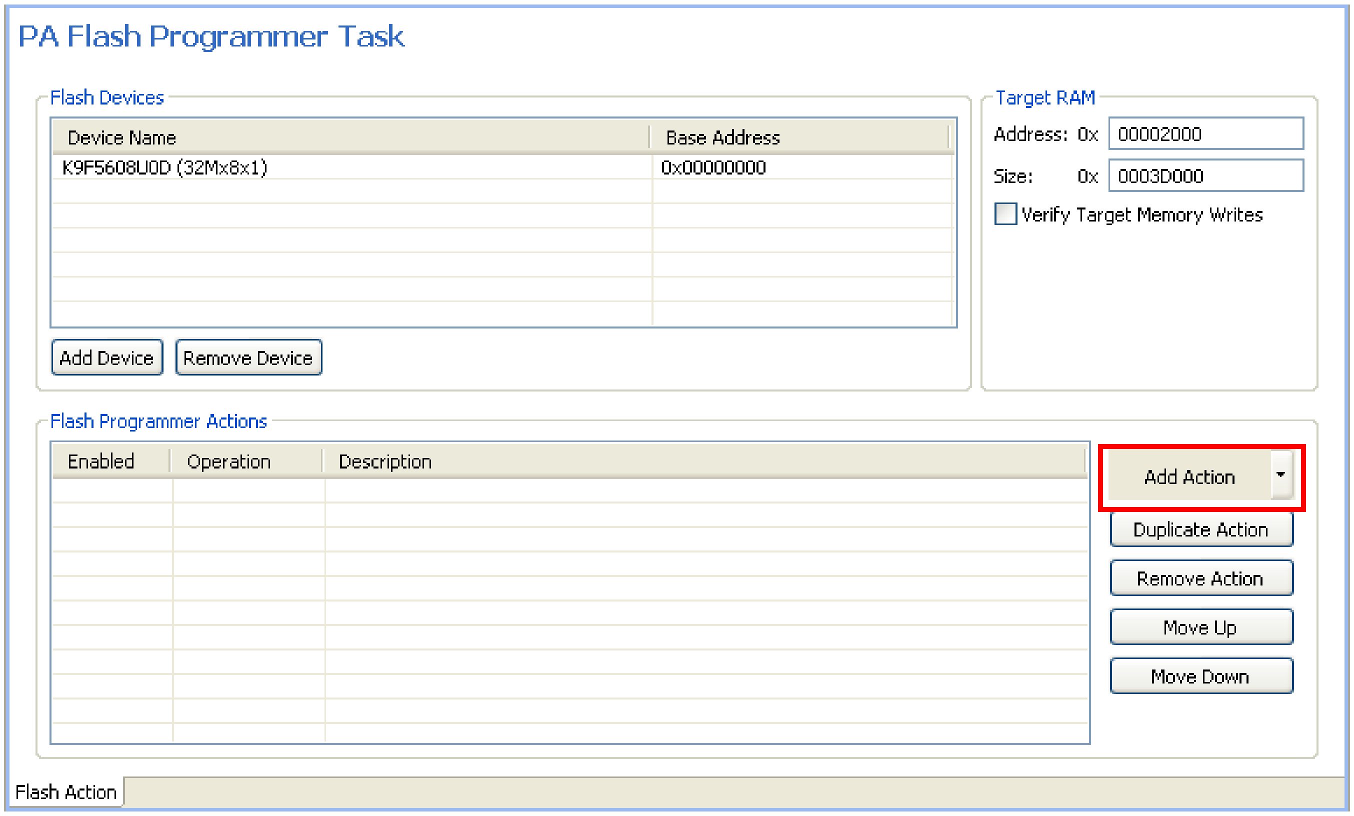
Each target task supplied with CodeWarrior has the following target tasks:

- Erase
- Blank check

- Program
- Verify
- Checksum
- Diagnostics
- Dump flash
- Protect
- Unprotect
- Secure
- Unsecure

All these operations can be selected using **Add Action** button. In the below steps, this step is not mentioned to avoid the repetition.

You can associate the above actions with the target task using the buttons in the **Flash Programmer Actions** section in the **Flash Programmer Task** editor. You can arrange the order of the actions using the **Move Up/Move Down** buttons.


PA Flash Programmer Task

Flash Devices

Device Name	Base Address
K9F5608U0D (32Mx8x1)	0x00000000

Target RAM

Address: 0x 00002000

Size: 0x 0003D000

Verify Target Memory Writes

Flash Programmer Actions

Enabled	Operation	Description

5.4.1. Erase / Blank Check Action

The erase action is added with **Add Erase / Blank Check Action** button. The dialog box shown in Figure 8 will be displayed.

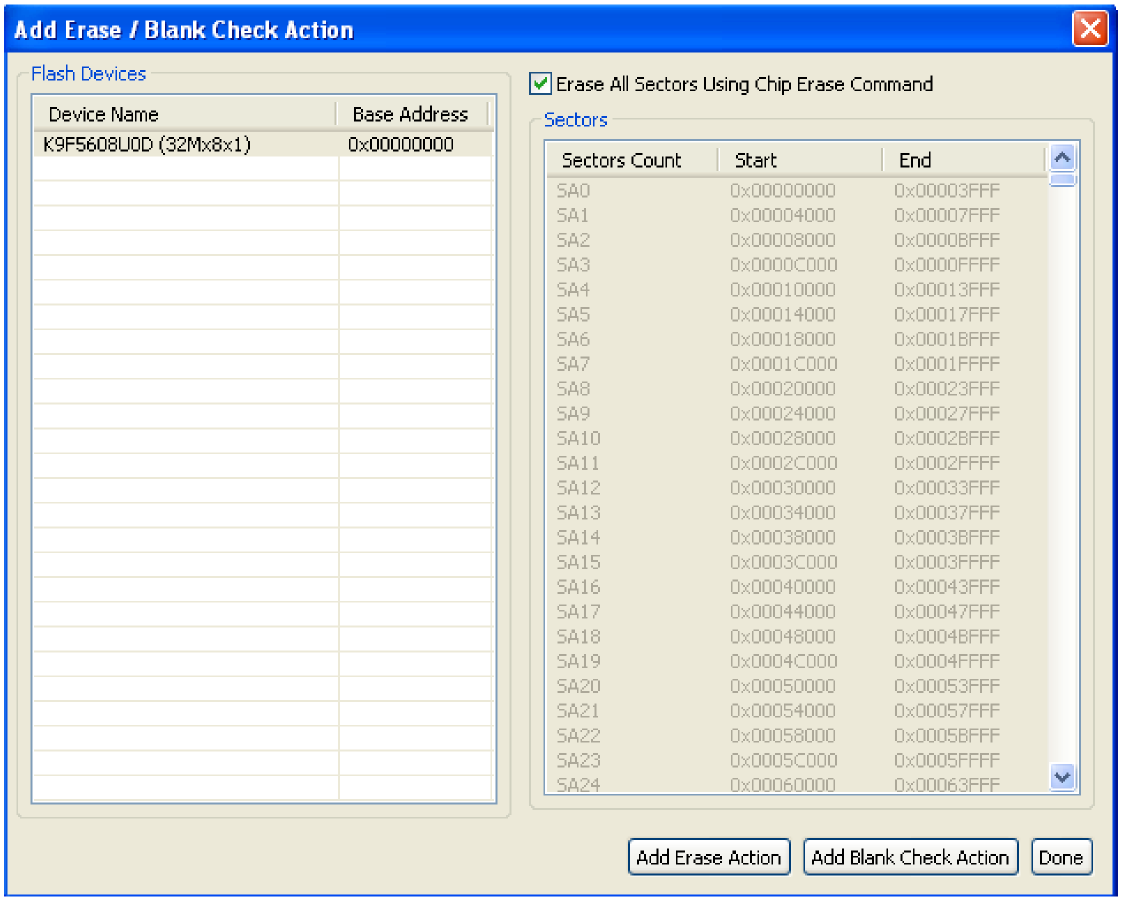
The **Add Erase Action** button enables you to erase a selected sector from the flash device and the **Add Blank Check Action** button checks the erased sectors to verify if they have been fully erased.

To add an erase / blank check action, follow these steps:

1. Select the Flash Programmer task to which you want to add erase/blank check actions.
2. Click the **Add Erase / Blank Check Action** button in the **Flash Programmer Task** editor. The **Add Erase / Blank Check Action** dialog box appears.
3. Select the flash device to which you want to add the erase/blank check action.
4. Select a sector from the **Sectors** table and click **Add Erase Action** to add an erase operation on the selected sector. You can select multiple sectors by holding Ctrl key while selecting the sectors.
5. Select a sector from the **Sectors** table and click **Add Blank Check Action** to add a blank check operation on the selected sector.
6. Check the **Erase All Sectors Using Chip Erase Command** check box to erase entire flash. To erase only a part of the device, leave it unselected and select only some of the sectors. If the size of the file to be programmed is unknown (for example, for an `elf` file, where only the necessary sections are downloaded on the flash), then you can use "**Erase sectors before program**" option. Using this option is better than erasing the entire flash, because for NAND flashes, the erase operation will fail if bad blocks are found. Also, to write something in flash, you need to erase it first or you will not be able to write the new information.

NOTE For more details on these operations, see Flash Programming Algorithm for AMD 16x1 Flash Devices.

Figure 8. Add Erase / Blank Check Action Dialog



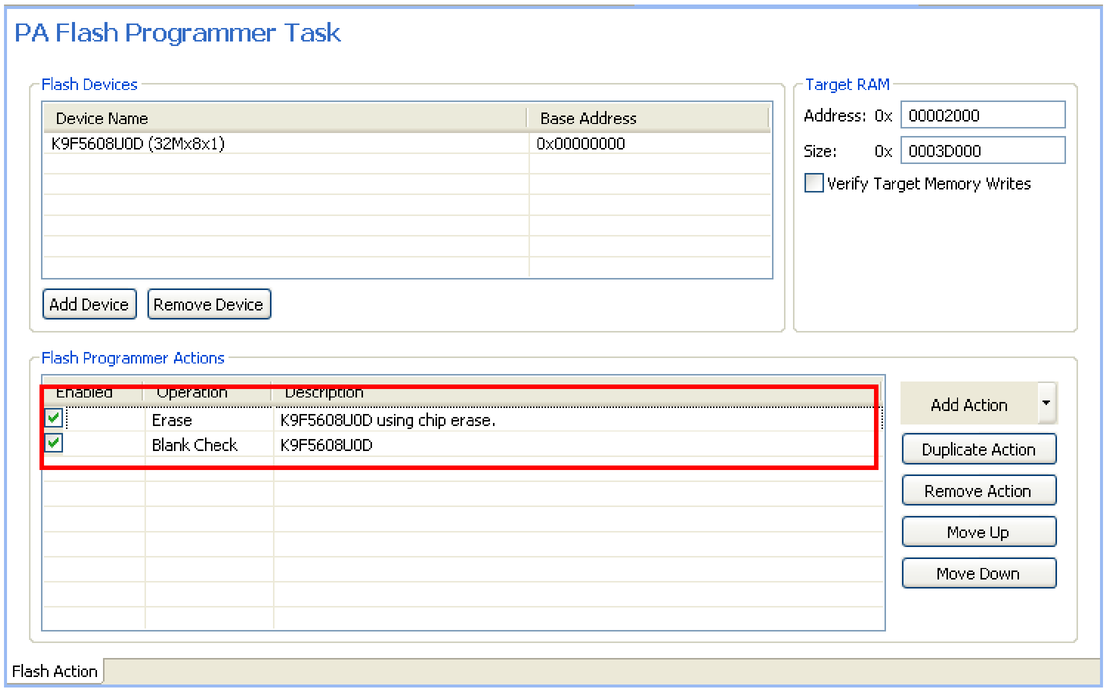
✓ Erase All Sectors Using Chip Erase Command

Device Name	Base Address
K9F5608U0D (32Mx8x1)	0x00000000

Sectors Count	Start	End
SA0	0x00000000	0x00003FFF
SA1	0x00004000	0x00007FFF
SA2	0x00008000	0x0000BFFF
SA3	0x0000C000	0x0000FFFF
SA4	0x00010000	0x00013FFF
SA5	0x00014000	0x00017FFF
SA6	0x00018000	0x0001BFFF
SA7	0x0001C000	0x0001FFFF
SA8	0x00020000	0x00023FFF
SA9	0x00024000	0x00027FFF
SA10	0x00028000	0x0002BFFF
SA11	0x0002C000	0x0002FFFF
SA12	0x00030000	0x00033FFF
SA13	0x00034000	0x00037FFF
SA14	0x00038000	0x0003BFFF
SA15	0x0003C000	0x0003FFFF
SA16	0x00040000	0x00043FFF
SA17	0x00044000	0x00047FFF
SA18	0x00048000	0x0004BFFF
SA19	0x0004C000	0x0004FFFF
SA20	0x00050000	0x00053FFF
SA21	0x00054000	0x00057FFF
SA22	0x00058000	0x0005BFFF
SA23	0x0005C000	0x0005FFFF
SA24	0x00060000	0x00063FFF

7. Click **Done** to close the **Add Erase / Blank Check Action** dialog box. The added erase / blank check actions appear in the **Flash Programmer Actions** table (Figure 9).

Figure 9. Erase and Blank Check Actions Added



Device Name	Base Address
K9F5608U0D (32Mx8x1)	0x00000000

Address: 0x 00002000

Size: 0x 0003D000

☐ Verify Target Memory Writes

Enabled	Operation	Description
✓	Erase	K9F5608U0D using chip erase.
✓	Blank Check	K9F5608U0D

5.4.2. Add Program / Verify Action

The Add Program / Verify Action button enables you to add program or verify flash actions for a flash device. To add a program or verify action, perform these steps:

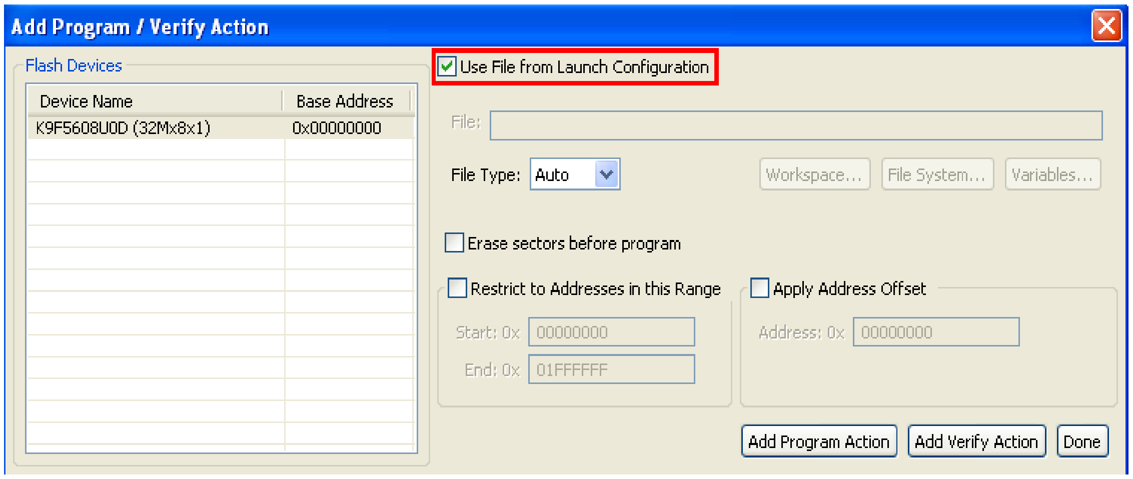
1. Select the Flash Programmer Task to which you want to add program or verify actions.
2. Click **Add Program / Verify Action** in the **Flash Programmer Task** editor. The **Add Program / Verify Action** dialog box appears.
3. Select the flash device to which you want to add a program or verify an action. In case the board has multiple flash devices, then the destination must be selected.
4. Check the **Use File from Launch Configuration** check box if you want to program/verify the launch configuration file. Alternatively, specify the file name and file path in the **File** text box or click the **Workspace**, **File System**, or **Variables** buttons to select the desired file.
   - Click the **Workspace** button to select a file from the current Eclipse workspace
   - Click the **File System** button to browse through the Windows file system and select the file
   - Click the **Variables** button to insert variables in the path
5. Define how the file should be read by selecting appropriate option from the File Type drop-down list. The following options are available:

- **Auto:** The Flash Programmer detects files of type Elf and Srec
- **Elf:** Elf executable file
- **Srec:** Motorola .s19 file format
- **Binary:** The file is read in binary format, no content interpretation is done

6. Check the **Restrict to Addresses in this Range** check box to define a range for flash accesses. Any program/verify action performed outside this range is ignored. You can specify the range in the **Start** and **End** text boxes, respectively.
7. Check the **Apply Address Offset** check box to apply an offset to the image to be written to the flash device. You can specify the offset in the **Address** text box. This value is added to the (computed) start address of the file.

   The start address is zero for binary files or read from the file header. In case you want to use a binary file and the flash is not mapped to zero, then enable the offset and set the value to the base address of the flash.
   The settings are displayed in Figure 10.
8. Click **Add Program Action** to add a program action for the flash device.
9. Click **Add Verify Action** to verify an action for the flash device.
10. Click **Done** to close the **Add Program / Verify Action** dialog box.

Figure 10. Add Program / Verify Action Dialog Box


Add Program / Verify Action

Flash Devices

Device Name	Base Address
K9F5608U0D (32Mx8x1)	0x00000000

[✓] Use File from Launch Configuration

File:

File Type: Auto

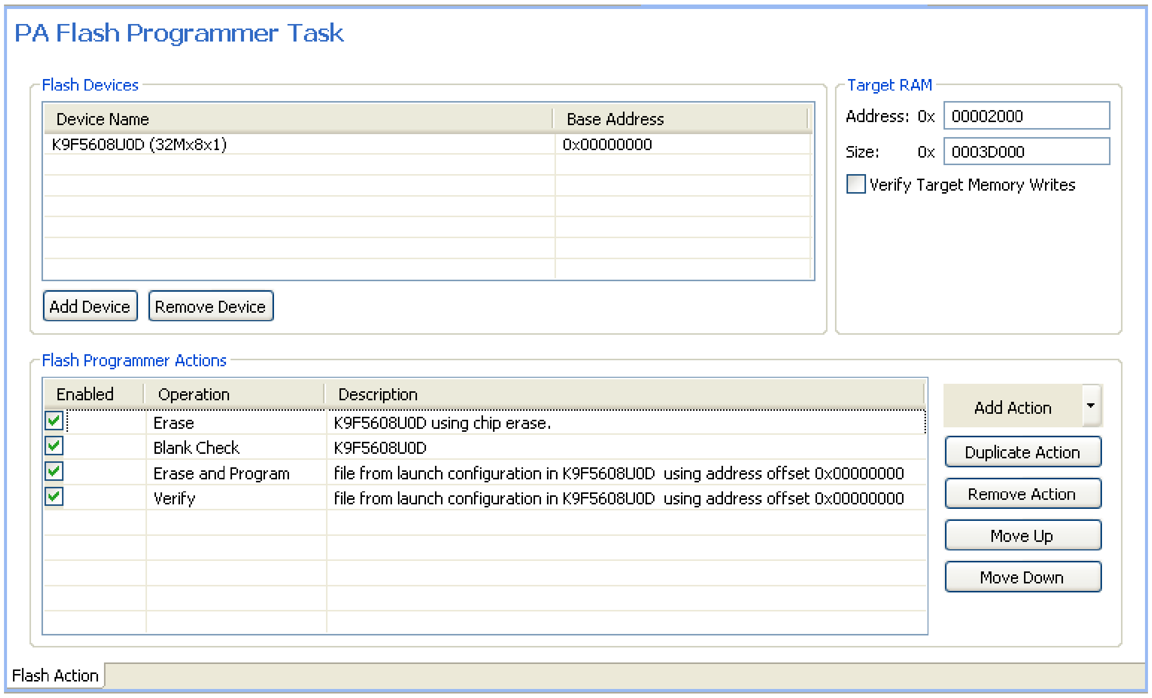
[ ] Erase sectors before program

[ ] Restrict to Addresses in this Range
Start: 0x 00000000
End: 0x 01FFFFFF

[ ] Apply Address Offset
Address: 0x 00000000

The added program / verify actions appear in the **Flash Programmer Actions** table as in the Figure 11.

Figure 11. Flash Programmer Actions Table



Device Name	Base Address
K9F5608U0D (32Mx8x1)	0x00000000

Address: 0x 00002000

Size: 0x 0003D000

☐ Verify Target Memory Writes

Enabled	Operation	Description
✓	Erase	K9F5608U0D using chip erase.
✓	Blank Check	K9F5608U0D
✓	Erase and Program	file from launch configuration in K9F5608U0D using address offset 0x00000000
✓	Verify	file from launch configuration in K9F5608U0D using address offset 0x00000000

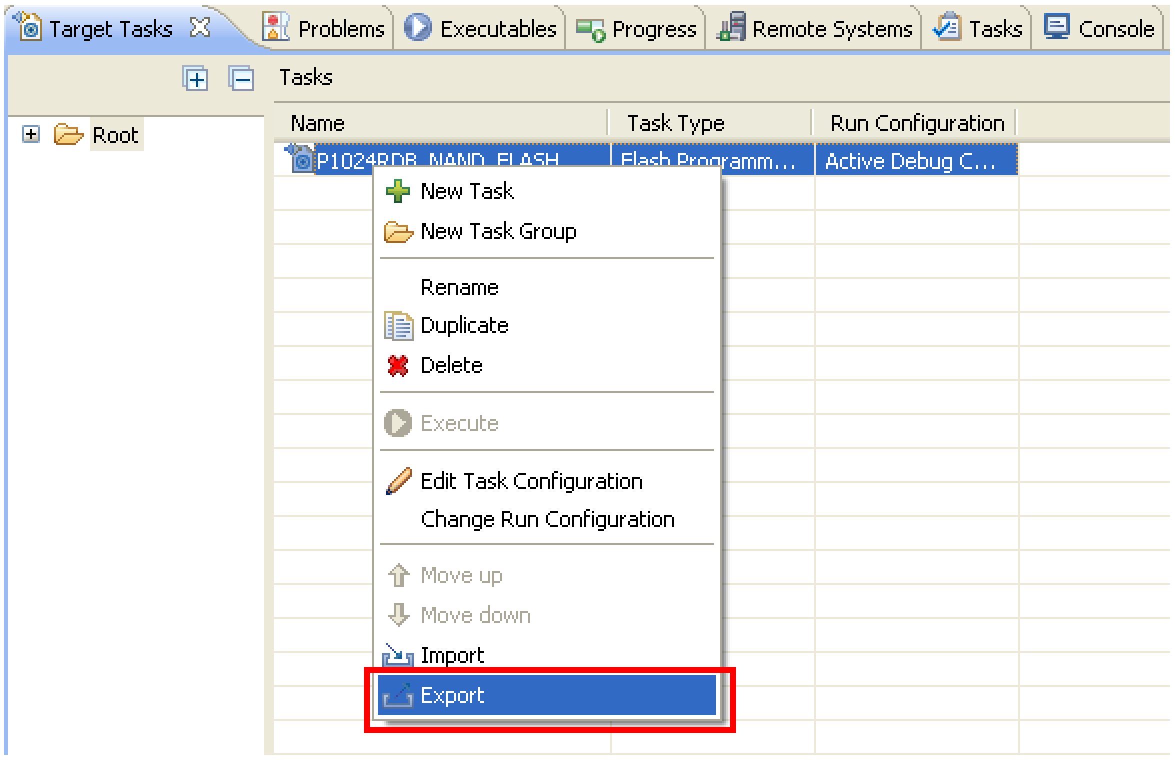
The task is now complete and should be saved using Ctrl + S or **Save** icon from toolbar.

5.5. Export the Target Task

The final step is to export the task to an .xml file. This allows us to import the task later or from another machine. To export a task, perform these steps:

1. In the **Target Tasks** view, select the task you want to export. Right-click the task and select **Export** as shown in Figure 12. Alternatively, click the **Export** icon available on the **Target Tasks** view toolbar.

Figure 12. Export Target Task



The **Save As** dialog box appears.

2. Browse to the desired location, specify the file name, and click the **Save** button.

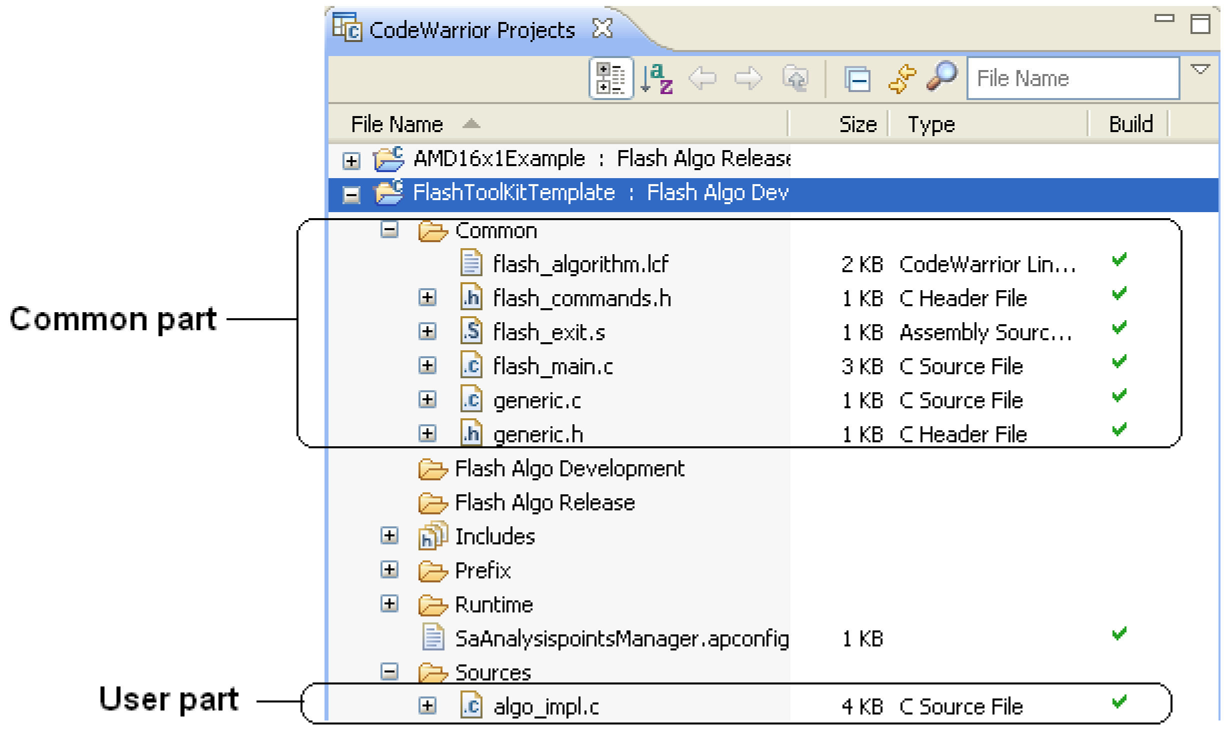
The saved task can be imported later using the **Import** button on the **Target Tasks** view toolbar.

6. Creating an External Flash Algorithm

6.1. Flash Tool Kit (FTK) Overview

The Flash Tool Kit helps you develop flash programming algorithms for the CodeWarrior Flash Programmer, as shown in Figure 13. This section provides important information needed before you begin creating a flash programming algorithm.

Figure 13. Flash Tool Kit



File Name	Size	Type	Build
AMD16x1Example : Flash Algo Release			
FlashToolKitTemplate : Flash Algo Dev			
Common			
flash_algorithm.lcf	2 KB	CodeWarrior Lin...	✓
flash_commands.h	1 KB	C Header File	✓
flash_exit.s	1 KB	Assembly Sourc...	✓
flash_main.c	3 KB	C Source File	✓
generic.c	1 KB	C Source File	✓
generic.h	1 KB	C Header File	✓
Flash Algo Development			
Flash Algo Release			
Includes			
Prefix			
Runtime			
SaAnalysispointsManager.apconfig	1 KB		✓
Sources			
algo_impl.c	4 KB	C Source File	✓

6.2. Flash Tool Kit General Structure

The flash programmer Flash Tool Kit (FTK) application is divided into three different sets of files:

- FTK Common Files (No Modification Needed): Includes initialization and other files. This component is common for all flash devices and should not be changed while developing the new flash programming algorithm. It consists of the following files:
  - `flash_algorithm.lcf`: The linker command file. This file is set up according to the rules for the flash programming applet allocation in physical memory.
  - `flash_commands.h`: The header file with API to the CodeWarrior Flash Programmer commands definition.
  - `generic.h`: The header file with the generic data structures and definitions used by the flash programming algorithms.
  - `flash_main.c`: The main function and API to the CodeWarrior Flash Programmer.
  - `flash_exis.s`: The exit point for the flash programming applet.
- User Files (Implement Algo): Includes the flash device specific files. This component is modified for any flash devices depending on the flash programming algorithm to be used. It consists of the following files:
  - `algo_impl.c`: It functions to implement for the flash device flash algorithm, such as **ID,**

erase_sector, erase_chip, write.

- Run-time files:
  - `ppc_eabi_init.c`: The flash programmer start-up initialization file.

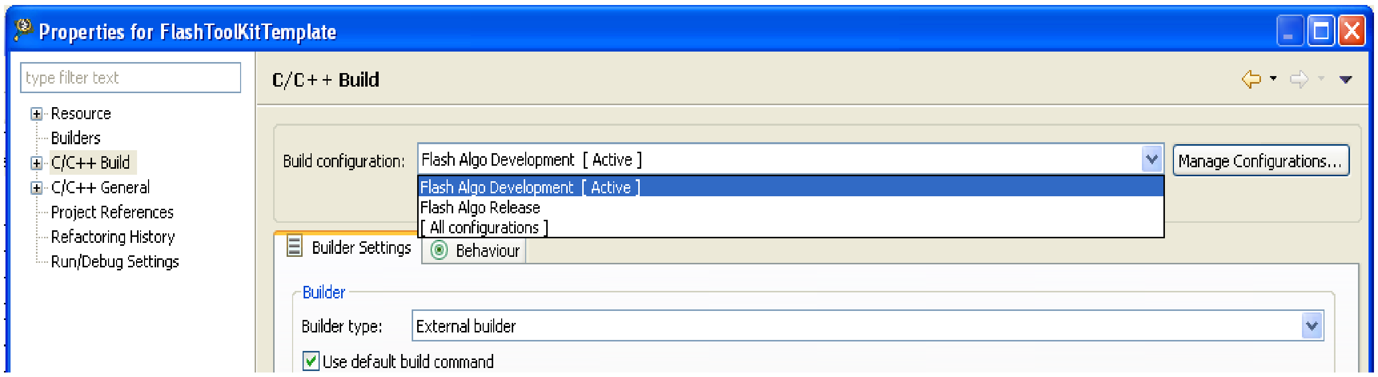
To create the new algorithm for flash programming, make all changes to the `algo_impl.c` (flash device algorithm implementation).

6.3. Flash Tool Kit Build Targets

Several build targets are predefined in the Flash Tool Kit (FTK):

- **Flash Algo Development:** The flash algorithm development and test application. The **ELF** executable file, created in Flash Algo Development, should be used to develop, debug, and test the new CodeWarrior Flash Programmer algorithm.
- **Flash Algo Release:** To create flash algorithm applet, the CodeWarrior Flash Programmer uses the **ELF** executable file, created in Flash Algo Release. This build target shares the flash device algorithm with the Flash Algo Development build target; it differs, however, because it cannot be debugged or tested (Figure 14).

Figure 14. Flash Tool Kit Targets



Build configuration: Flash Algo Development [Active]

Flash Algo Development [Active]
Flash Algo Release
[All configurations]

Builder type: External builder

☑ Use default build command

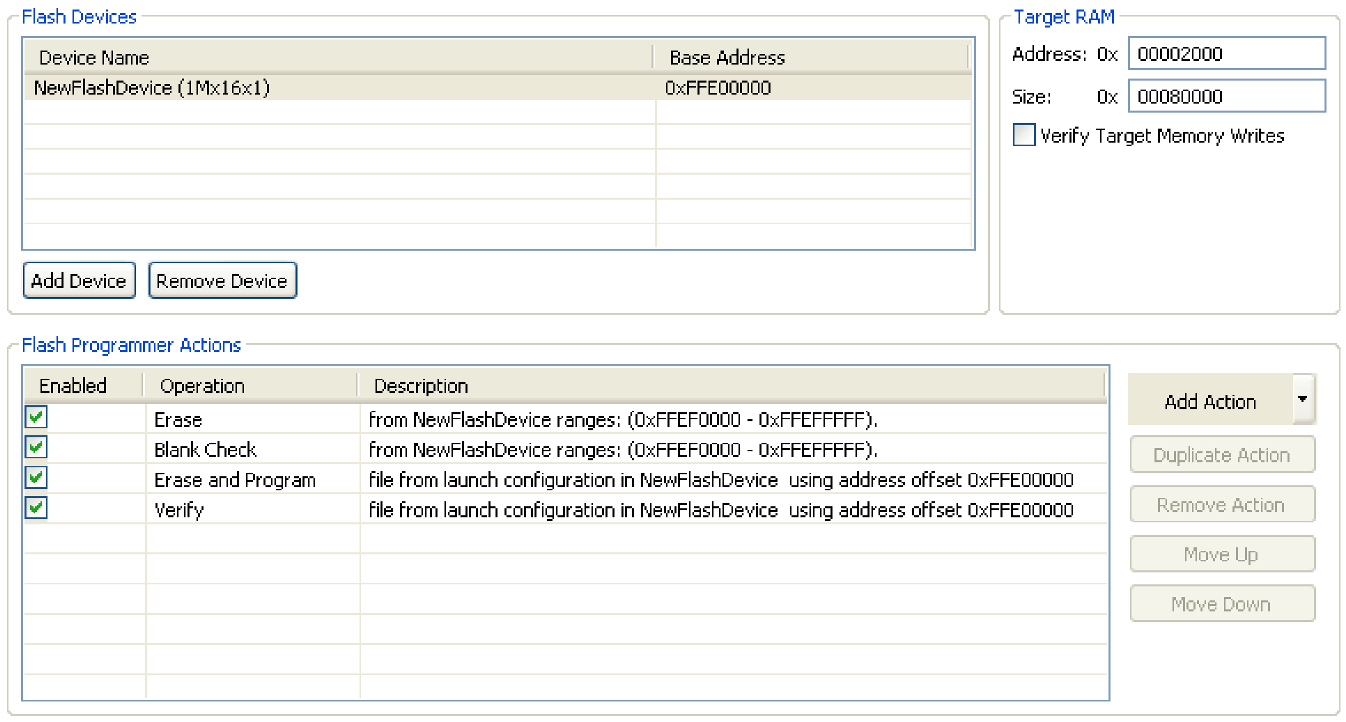
6.4. Flash Programmer API

The CodeWarrior Flash Programmer communicates with the flash programming algorithm applet through five different commands:

- get ID
- erase sector
- erase chip
- program
- verify

The CodeWarrior Flash Programmer uses an exchange zone in target memory to communicate with the flash applet. The Flash Programmer Target Configuration specifies the Target Memory Buffer; the exchange zone is at the start of this buffer, as shown in Figure 15.

Figure 15. Target Configuration Buffer Memory Area Start Address



Device Name	Base Address
NewFlashDevice (1Mx16x1)	0xFFE00000

Address: 0x 00002000

Size: 0x 00080000

☐ Verify Target Memory Writes

Enabled	Operation	Description
✓	Erase	from NewFlashDevice ranges: (0xFFEF0000 - 0xFFEFFFFF).
✓	Blank Check	from NewFlashDevice ranges: (0xFFEF0000 - 0xFFEFFFFF).
✓	Erase and Program	file from launch configuration in NewFlashDevice using address offset 0xFFE00000
✓	Verify	file from launch configuration in NewFlashDevice using address offset 0xFFE00000

Parameter_block_t Structure

For the detailed description of the `Parameter_block_t` structure, see Listing 4.

Listing 4. Parameter_block_t structure details

```
typedef struct pb {
unsigned long function; /* What function to perform ? */
pointer_t base_addr; /* where are we going to operate */
unsigned long num_items; /* number of items */
retval_t result_status;
pointer_t items;
unsigned long is_timebase_enabled;
  unsigned long  padding[3];     /* Padding for CheckSpeed Parameters - these are not used
for PA & SC - only for MCU (added already in every generic.h MCU algorithm)*/
  unsigned long      ccsrbar;           /* Base address for memory mapped registers */
  unsigned long      core_type;         /* Core Type - used for handle between different
processor FP controller structs */
  unsigned long      controller_offset;        /* Controller Offset */
  unsigned long      controller_type;          /* Controller Type */
  unsigned long  padding1[37];   /* Padding for the gap space between params proc and
Additional params*/
```

```
  unsigned long  additionalParameterNum;      /* number of additional parameters sent by FP
Plugin */
  unsigned long  additionalParameterStart;    /* start address for Additional parameters
*/unsigned long  padding2[7];    /* Padding for other additional parameters*/
} parameter_block_t;
```

Listing 4 definitions:

- `Function`: Command to be executed from the CodeWarrior Flash Programmer.
- `base_addr`: Start address of the flash memory.
- `num_items`: Number of items to be transferred from the CodeWarrior Flash Programmer to the flash programming applet.
- `result_status`: Status of the command; through this field, the flash programming applet notifies the CodeWarrior Flash Programmer about the status of the command being executed. Some values for `result_status` are defined in `generic.h`; however, the user can define more, if needed.
- `items`: Start address of the data to be transferred from the CodeWarrior Flash Programmer to the flash programming applet.
- `Is_timebase_enabled`: A time-out mechanism can be implemented in the algorithm if timebase is enabled.
- Padding[3]: Used only for internal purposes.
- Ccsrbar: Base address for memory mapped registers.
- Core_type: Used for handling different processor FP controller structures (can be e500v1, e500v2, and so on – see generic.h for all macro values).
- Controller_offset: controller offset address
- Controller type: Controller type (can be eLBC, IFC, eSPI, and so on – see generic.h for all macro values).
- Padding1: Used only for internal purposes.
- additionalParamenterNum: Number of additional parameters sent by FP plugin.
- additionalParamterStart: Start address for additional parameters.
- Padding2: Padding for additional parameters.

NOTE `items` and `base_addr` are of type `pointer_t`, which is a union that can accommodate different types (such as `char *, short *`). This arrangement makes the algorithm scalable, so it can be used for 8-bit, 16-bit, or 32-bit flash.

ID

The CodeWarrior Flash Programmer uses the getting chip ID command right after the flash algorithm is loaded to the memory buffer to check if the applet runs. For the `ID` command, the CodeWarrior Flash Programmer:

- Loads the flash programming applet to the target board
- Sets the command ID, as shown in the function field of Listing 4
- Runs flash programming applet
- Waits while flash applet stops execution
- Checks the status of the command being executed, as shown in the `result_status` field of Listing 4

fEraseChip

The full chip erase command is called by the CodeWarrior Flash Programmer when a full chip erase is performed. For the `fEraseChip` command, the CodeWarrior Flash Programmer:

- Loads the flash programming applet to the target board
- Sets the command `fEraseChip`, as shown in the `function field` of Listing 4
- Runs the flash programming applet
- Waits while the flash applet stops execution
- Checks the status of the command being executed, as shown in the `result_status` field of Listing 4

NOTE Some flash devices do not support the full chip erase command. Check the flash device's specifications, available from the manufacturer.

fEraseSector

The CodeWarrior Flash Programmer calls the sector erase command to erase a set of flash memory sectors. For the `fEraseSector` command, the Flash Programmer:

- Loads the flash programming applet to the target board
- Sets the command fEraseSector, as shown in the function field of Listing 4
- Specifies number of blocks to be erased, as shown in the num_items field of Listing 4
- Specifies start-up address of each block to be erased, as shown in the items field of Listing 4
- Runs flash programming applet

- Waits while flash applet stops execution
- Checks the status of the command being executed, as shown in the result_status field of Listing 4

fWrite

The fWrite program buffer command is called by the flash programmer to program a set of values at a specific address. For the `fWrite` command, the CodeWarrior Flash Programmer:

- Loads the flash programming applet to the target board
- Sets the command fWrite, as shown in the function field of Listing 4
- Specifies number of bytes to be programmed, as shown in the num_items field of Listing 4
- Specifies start-up address of data to be programmed, as shown in the items field of Listing 4
- Runs flash programming applet
- Waits while flash applet stops execution
- Checks the status of the command being executed, as shown in the result_status field of Listing 4

fVerify

The fVerify command is usually not implemented in flash algorithms. For all NOR flash devices, a separate application (*`flash_utility.elf`*) performs verification. Your CodeWarrior product includes this program. This application reads memory from the target and compares each returned value with the desired value.

However, you can enable the fVerify command by modifying the flash device's XML description file. If you make this change, the verify operation is done by the flash algorithm. For NAND and SPI flash devices, the verify operation must be done by the flash algorithm because for these flash devices, verification cannot be achieved just by reading memory.

6.5. Create a New Flash Programming Algorithm

This section provides you step-by-step instructions on how to use the Flash Tool Kit to create a new CodeWarrior Flash Programmer flash programming algorithm for a flash device, which is not integrally supported by the CodeWarrior software. The steps are given below:

1. Store original versions of the Flash Tool Kit files from the CodeWarrior delivery to some directory on your machine.
2. Copy `FlashToolKitTemplate` folder from *`{CodeWarrior}\PA\PA_Tools\FlashToolKit\FlashToolKitTemplate`* to a different directory, where CodeWarriorInstallDir specifies the location where the CodeWarrior software is installed.

3. Import Flash Tool Kit Template project:
   a) Select **File → Import**. The Import dialog box appears.
   b) Select **General → Existing Projects** into Workspace and click **Next**. The **Import Projects** page appears.
   c) Click **Browse** to select the parent folder where you have copied the `FlashToolKitTemplate` folder.
   d) The **Projects** list will show all available projects in the `FlashToolKitTemplate` folder. If you see projects other than `FlashToolKitTemplate`, then deselect all other projects and click **Finish**.
   e) Ensure the `Flash Algo Development` build target is selected.
4. Select **Run → Debug Configuration**. The **Debug Configuration** dialog box appears.
5. Expand the **CodeWarrior Download** tree node and select the desired launch configuration.
6. Click the **Debugger** tab in the right panel.
7. Select the required target processor from the **Target Processor** drop-down list.
8. Specify required target initialization and memory configuration files for the connected hardware in the Target initialization file and Memory Configuration File text boxes.
   a) For supported Freescale Evaluation Boards, you can use the debugger configuration files (*.tcl), and the debugger memory files (*.mem) available with the CodeWarrior Development Studio:
   `{CodeWarrior}\PA\PA_Support\Initialization_Files`

NOTE In case of custom hardware design, the debugger configuration file and the memory mapping file must be written. Also, the memory initialization file for the flash device should be checked before trying to create the new flash programming algorithm.

9. Modify the `algo_impl.c` file:
   a) The flash algorithm functionality file, `algo_impl.c`, should be modified and filled with the correct programming commands, as recommended by the flash device manufacturer.
10. Modify ID function in the algo_impl.c file:
    a) By default, the ID function in the `algo_impl.c` file looks as shown in Listing 5.
    b) The following definitions pertain to Listing 5:
       - `parameter_block_t *p_pb` – Pointer to the parameter_block_t structure to be passed to the ID function
       - `retval_t` – Result of the function execution
    c) The correct command sequence should be created for the ID function based on the

recommendations of the flash device manufacturer, as described in Implementing ID Function for AMD 16x1 Flash Devices.

Listing 5. ID function template in algo_impl.c file

```
retval_t ID(parameter_block_t *p_pb)
{
    retval_t result = 0;
    volatile unsigned long* item_addr = (p_pb->items).l;
    /* Add code: the correct access size depending on the bus must be used for the base_addr
*/
    volatile unsigned short *base_addr = (p_pb->base_addr).w;

    /* Add code: first of all reset the device.
     The fID is not called in the new flash programmer plugin therefore
     the flash chip must always be brought into the read state.
    */
    /* Add code: read the device ID */

    /* we currently assume that we have the right value */
    /* anyway, the IDE have to care about the flash ID and compare with the xml file */
    return result;
}
```

11. Modify the `erase_sector` function:

    a) By default, the `erase_sector` function in the `algo_impl.c` file appears as shown in Listing 6.

    b) Listing 6 definitions:

    - `parameter_block_t *p_pb` – Pointer to the parameter_block_t structure to be passed to the `erase_sector` function
    - unsigned long `sect_index` – Index of the sector to be erased
    - `retval_t` – Result of the function execution

    c) Based on recommendations from the flash device manufacturer, the correct command sequence must be created for flash-sector erasing, as described in Implementing erase_sector Function for AMD 16x1 Flash Devices.

    d) For some NOR flash devices, you can check if the sector you want to delete is protected; this verification is described in Implementing erase_sector Function for AMD 16x1 Flash Devices.

Listing 6. Function template erase_sector in algo_impl.c

```
retval_t erase_sector(parameter_block_t *p_pb, unsigned long sect_index)
{
    int timed_out, got_it;
    retval_t result = 0;
    /* Add code: the correct access size depending on the bus must be used for the base_addr
*/

    volatile unsigned short *base_addr = ((unsigned short **)(p_pb->items).w)[sect_index];

    /* Add code: verify if the sector is protected */

    /* Add code: first of all reset the device.
     The fID is not called in the new flash programmer plugin therefore
     the flash chip must always be brought into the read state.
    */

    /* Add code: erase one sector */

    /* Add code: wait for status */

    /* Add code: handle error (and timeout if needed) */

    /* Add code: put back the flash in read state */

    return result;
}
```

12. Modify the `erase_chip` function:

    a) By default, the erase_chip function in the `algo_impl.c` file looks as shown in Listing 7.

    b) Listing 7 definitions:

       - `parameter_block_t *p_pb`: Pointer to the parameter_block_t structure to be passed to the `erase_chip` function
       - `retval_t`: Result of the function execution

    c) Create the correct command sequence for full flash chip erasing based upon recommendations from the flash device manufacturer, as shown in Implementing erase_chip Function for AMD 16x1 Flash Devices.

    d) For some NOR flash devices, you can check if any of the sectors are protected; this verification is described in Implementing erase_sector Function for AMD 16x1 Flash Devices.

Listing 7. Function template erase_chip in algo_impl.c

```
retval_t erase_chip(parameter_block_t *p_pb)
{
    int errors = 0;
    retval_t result = 0;
    unsigned short stat;
    int got_it;
    /* Add code: the correct access size depending on the bus must be used for the base_addr
*/
    volatile unsigned short *base_addr = (p_pb->base_addr).w;

    /* Add code: verify if the sectors are protected */

    /* Add code: first of all reset the device.
     The fID is not called in the new flash programmer plugin therefore
     the flash chip must always be brought into the read state.
    */

    /* Add code: erase one sector */

    /* Add code: wait for status */

    /* Add code: handle error (and timeout if needed) */

    /* Add code: put back the flash in read state */

    return result;
}
```

13. Modify the `write` function:
    a) By default, the write function in the `algo_impl.c` file looks as shown in Listing 8.
    b) Listing 8 definitions:
        - `parameter_block_t *p_pb`: Pointer to the parameter_block_t structure to be passed to the write function
        - `retval_t`: Result of the function execution
    c) Create the correct command sequence for flash device programming according to the recommendations of the flash device manufacturer, as described in Implementing write Function for AMD 16x1 Flash Devices.
    d) For some NOR flash devices, you can check if any of the sectors that are involved in the writing process are protected; this verification is described in Implementing erase_sector Function for AMD 16x1 Flash Devices.

Listing 8. Function template write in algo_impl.c

```
retval_t write(parameter_block *p_pb)
{
    int timed_out, got_it;
    unsigned long i;
    unsigned short stat;
    retval_t errors = 0;
    /* Add code: the correct access size depending on the bus must be used for the base_addr
*/
```

```
    volatile unsigned short *base_addr = (p_pb->base_addr).w;

    /* Add code: verify if the sectors involved in the writing process are protected */

    /* Add code: first of all reset the device.
     The fID is not called in the new flash programmer plugin therefore
     the flash chip must always be brought into the read state.
    */


    /* Add code: program the bytes pointed in the buffer : p_pb->items,
        they are  p_pb->num_items bytes
        handle error (and timeout if needed) for each of the program sequence
    */


    /* Add code: put back the flash in read state */

  return errors;
}
```

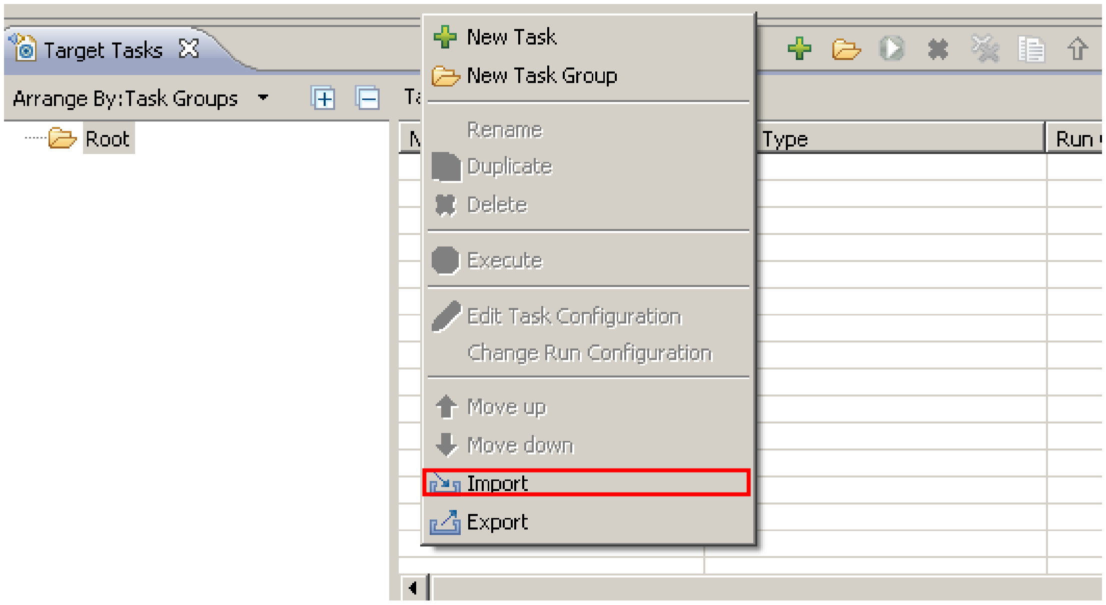
NOTE See the flash device manufacturer for the flash device memory organization. See hardware description for the flash device addressing.

14. Compile Flash Algo Development target:
    a) During new algorithm creation and testing, use the Flash Algo Development build target of the Flash Development Kit. Compile the Flash Algo Development target with the `flash_algo.c` file, which is modified for the flash programming procedures. Compilation will result in creation of a new `flash_alg_debug.elf` file.
15. Flash algorithm unit test:
    a) To test a flash programming algorithm, define custom flash device parameters in *flash_main.c*. You can select the operation you want to perform and the address in flash to which the operation will be applied.
16. Compile Flash Algo Release target:
    a) When the flash programming algorithm for the new flash device works correctly (as confirmed in unit testing), compile the Flash Algo Release target. The output of the Flash Algo Release target — `flash_alg_release.elf` — must be copied to the following folder:
    `{CodeWarior}\PA\bin\plugins\support\Flash_Programmer\EPPC`
17. Add a new flash device to the flash programmer:
    a) See Creating Flash Device XML Configuration File for information on adding a new flash device.

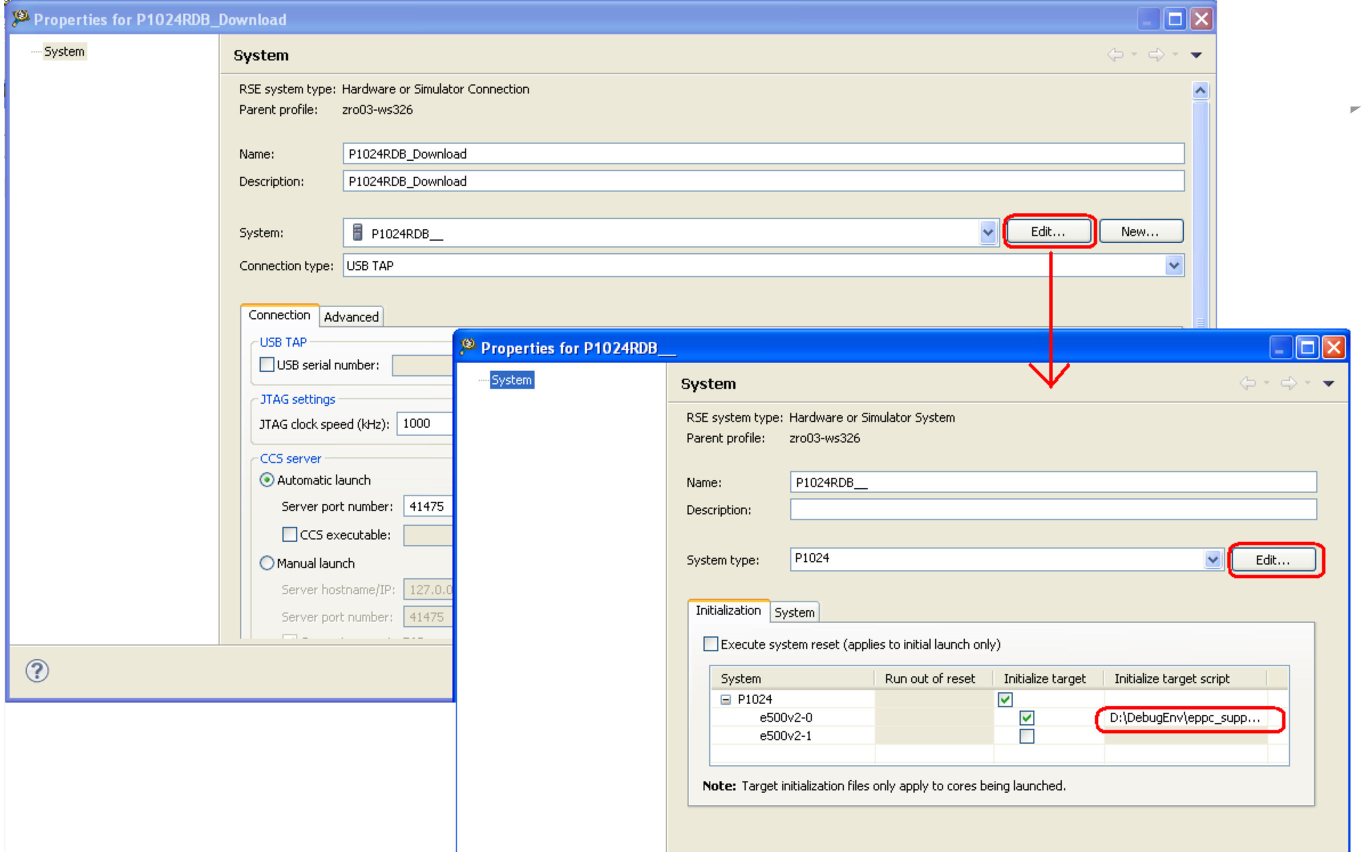
18. Create a new default target task:
    a) See Creating a New Target Task for information on creating a new default target task.
19. Set flash device configuration in flash programmer:
    a) Restart the Eclipse IDE so that the Eclipse IDE can use the new data from the updated manifest file.
    b) Select **Window > Show View > Other… > Debug > Target Tasks** from the menu bar to open the **Target Tasks** view.
    c) Right-click the **Tasks** table and select **Import** from the context menu (Figure 16).

       Alternatively, click the icon on the **Target Tasks** view toolbar to import a task. The **Open** dialog box appears.

Figure 16. Import Task



    d) Browse to the location of the task that you want to import, select the required task, and click **Open**. The imported task gets added to the **Tasks** table.
    e) Make a new Download Run Configuration (Right-click the project, select **Debug As -> Debug Configurations**) specific for your board (below is an example for P1024RDB)
       - **Select the project** – Flash Tool Kit Template and the debug application `flash_alg_debug.elf`



RSE system type: Hardware or Simulator Connection
Parent profile: zro03-ws326
Name: P1024RDB_Download
Description: P1024RDB_Download
System: P1024RDB__ Edit... New...
Connection type: USB TAP

USB TAP
☐ USB serial number:
JTAG settings
JTAG clock speed (kHz): 1000
CCS server
● Automatic launch
Server port number: 41475
☐ CCS executable:
○ Manual launch
Server hostname/IP: 127.0.0
Server port number: 41475

RSE system type: Hardware or Simulator System
Parent profile: zro03-ws326
Name: P1024RDB__
Description:
System type: P1024 Edit...

☐ Execute system reset (applies to initial launch only)

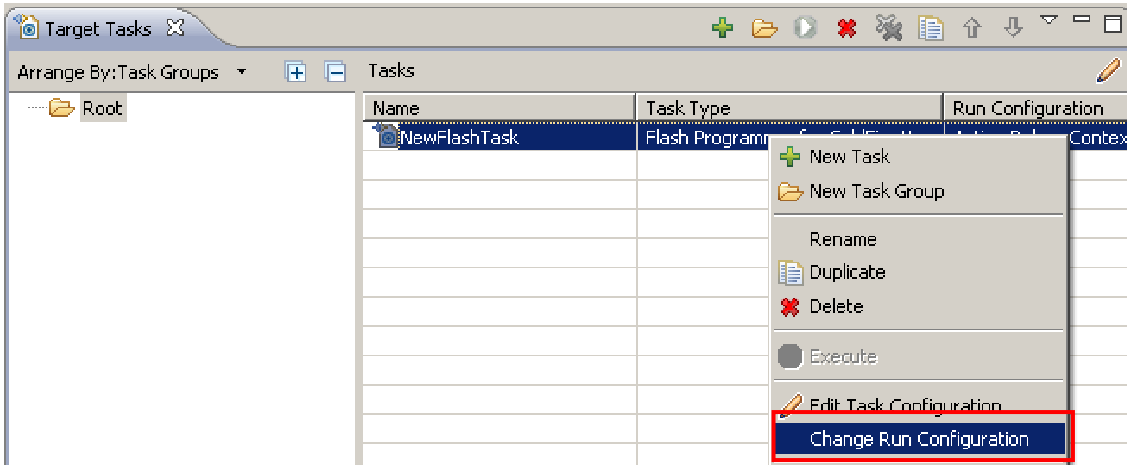
System	Run out of reset	Initialize target	Initialize target script
⊟ P1024		✓	
e500v2-0		✓	D:\DebugEnv\eppc_supp...
e500v2-1		☐	

Note: Target initialization files only apply to cores being launched.

- **Make a new connection** – select the Initialization file (refer step 5), the System and Connection type (usb, etap, gtap).

f) Check the **Initialize target** check box to select the **Initialize target script** for core0 from the system list. Right-click the imported task and select **Change Run Configuration** (Figure 17).

Figure 17. Change Run Configuration



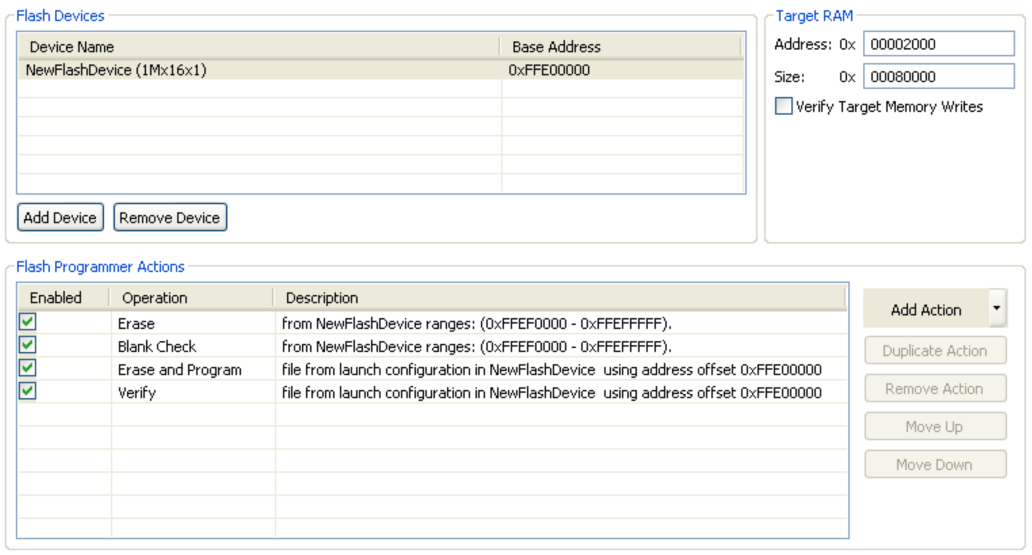
Name	Task Type	Run Configuration
NewFlashTask	Flash Programm	Contex

g) The **Run Configuration** dialog box appears. Select the Run Configuration from the drop-down list and click **OK**.

h) Double click the task to open the task in the editor area.

i) Check if the flash device used is the newly introduced. As an example, the device has been named NewFlashDevice (Figure 18).

Figure 18. Flash Task Editor



Device Name	Base Address
NewFlashDevice (1Mx16x1)	0xFFE00000

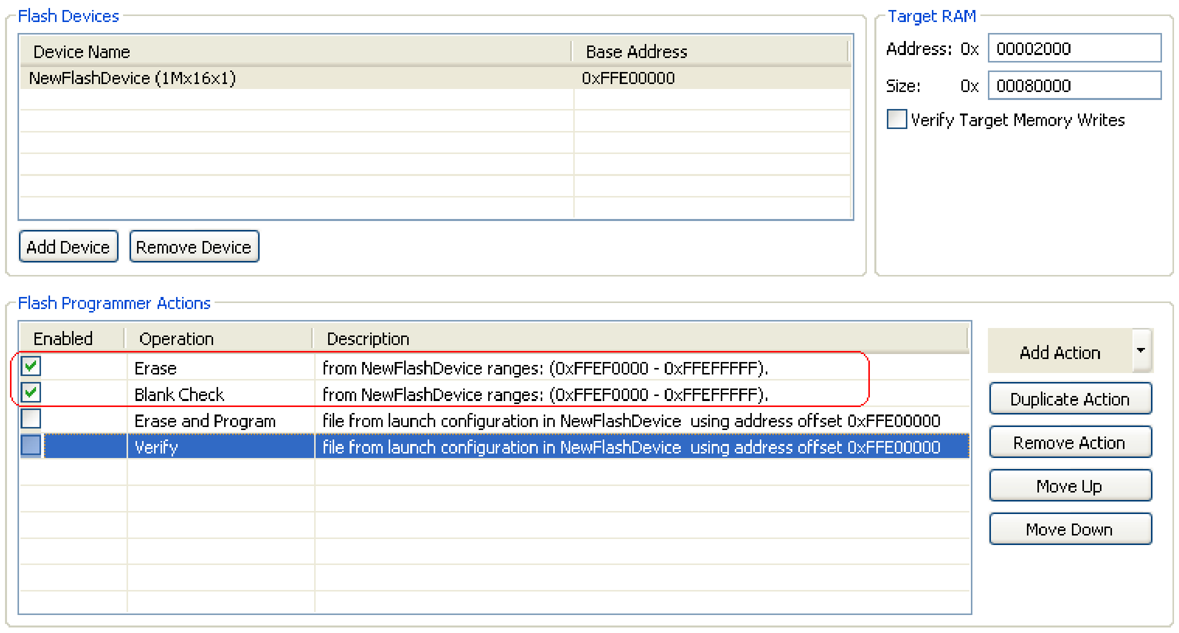
Address: 0x 00002000
Size: 0x 00080000
☐ Verify Target Memory Writes

Enabled	Operation	Description
✓	Erase	from NewFlashDevice ranges: (0xFFEF0000 - 0xFFEFFFFF).
✓	Blank Check	from NewFlashDevice ranges: (0xFFEF0000 - 0xFFEFFFFF).
✓	Erase and Program	file from launch configuration in NewFlashDevice using address offset 0xFFE00000
✓	Verify	file from launch configuration in NewFlashDevice using address offset 0xFFE00000

20. Erase and blank check the device:

a) Select only Erase and Blank Check actions (Figure 19).

Figure 19. Erase and Blank Check actions selected



Device Name	Base Address
NewFlashDevice (1Mx16x1)	0xFFE00000

Address: 0x 00002000
Size: 0x 00080000
☐ Verify Target Memory Writes

Enabled	Operation	Description
✓	Erase	from NewFlashDevice ranges: (0xFFEF0000 - 0xFFEFFFFF).
✓	Blank Check	from NewFlashDevice ranges: (0xFFEF0000 - 0xFFEFFFFF).
☐	Erase and Program	file from launch configuration in NewFlashDevice using address offset 0xFFE00000
☐	Verify	file from launch configuration in NewFlashDevice using address offset 0xFFE00000

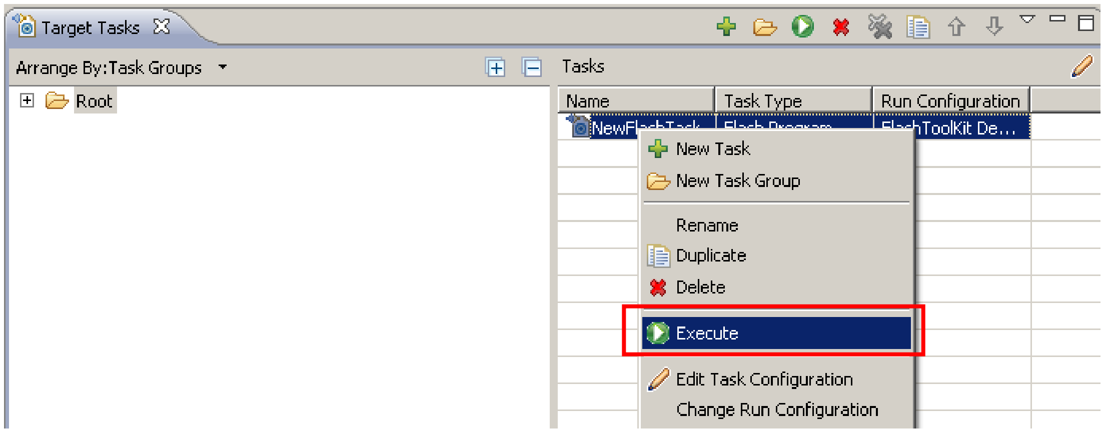
b) Save the task by pressing Ctrl + S or choosing **Save** button from toolbar.

c) Select **Window > Preferences** from the IDE menu bar. The **Preferences** dialog box appears.

d) Select C/C++ > Debug > CodeWarrior Debugger and check the Show verbose output in Target Tasks check box in the CodeWarrior Debugger panel.

e) Right-click the task and select **Execute** (Figure 20).

Figure 20. Execute the task



f) The **Flash Programmer Console** appears in the **Console** view and the task log gets printed there.

g) Check for the algorithm used in console log. The section from Listing 9 should be present.

Listing 9. Algorithm used for Erase operation

```
.
cmdwin::fl::erase all
Beginning Operation ...
-------------------------
log: Using Algorithm: flash_alg_debug.elf
.
.
```

h) Check in the log if the task is successful. The lines from Listing 10 should be present.

Listing 10. Part of Erase and Blank Check actions log

```
.
.
Erasing ...........................
Reading erase return status
Erase Command Succeeded
.
.
Blank Checking ...........
```

```
Reading blank check return status
Blank Check Succeeded
```

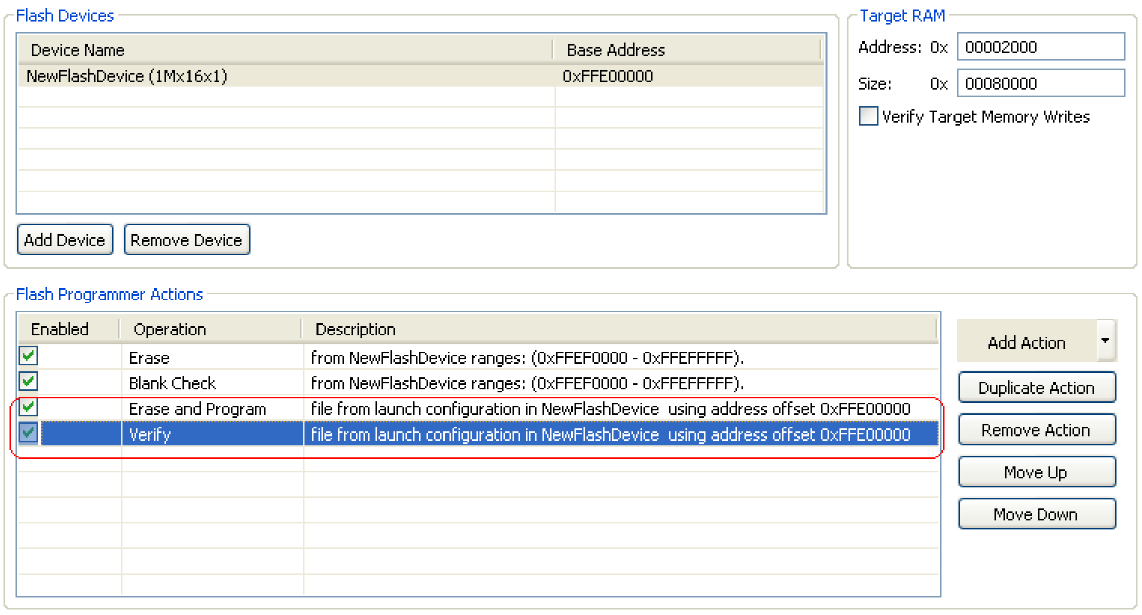
NOTE: In case the flash device cannot be erased, check if you are using the correct target task, and if the hardware connection is set up correctly.

21. Perform the Programming test:

    a) Different sized binary S-record files are available in the Flash Tool Kit delivery to check whether the flash device can be programmed. The path to the S-record files is: `{CodeWarrior}\PA\PA_Tools\FlashToolKit\TestSrecFiles`
    They are: `64k_at_0.S`, `128k_at_0.S`, `256k_at_0.S`, `1M_at_0.S`, `2M_at_0.S`, and `4M_at_0.S`. For each file, the file name specifies the size of the file. For example, file `256k_at_0.S` is 256 Kilobyte sized and is linked to the 0 start-up address.

    b) Enable Program and Verify actions (Figure 21). Save the task (Ctrl + S).

Figure 21. Enable the Program and Verify Actions



Device Name	Base Address
NewFlashDevice (1Mx16x1)	0xFFE00000

Address: 0x 00002000
Size: 0x 00080000
☐ Verify Target Memory Writes

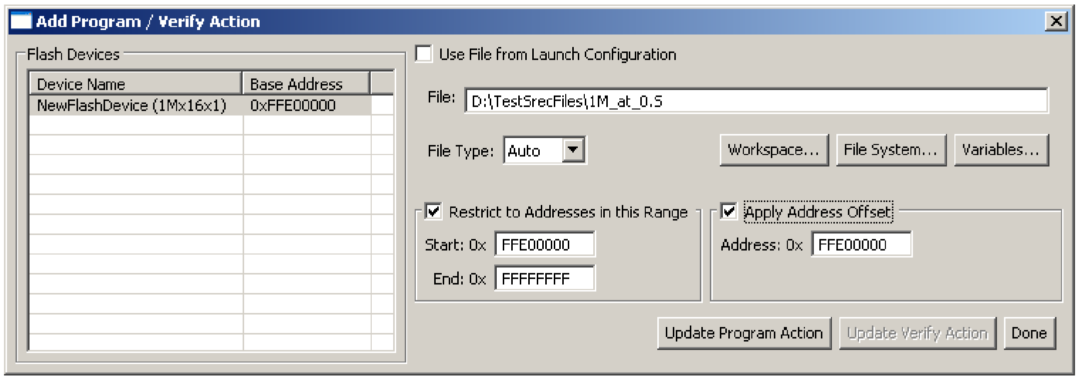
Enabled	Operation	Description
✓	Erase	from NewFlashDevice ranges: (0xFFEF0000 - 0xFFEFFFFF).
✓	Blank Check	from NewFlashDevice ranges: (0xFFEF0000 - 0xFFEFFFFF).
✓	Erase and Program	file from launch configuration in NewFlashDevice using address offset 0xFFE00000
✓	Verify	file from launch configuration in NewFlashDevice using address offset 0xFFE00000

    c) Double-click **Program** in the **Operation** column. The **Add Program / Verify Action** dialog box appears.

    d) Clear **Use File from Launch Configuration** check box and click **File System** to select a `.S` file to be programmed.

    e) Select the **Restrict to Addresses in this Range** and **Apply Address Offset** check boxes. Figure 22 shows the settings.

Figure 22. Add Program / Verify Action Dialog Box



Device Name	Base Address
NewFlashDevice (1Mx16x1)	0xFFE00000

- [ ] Use File from Launch Configuration
- File: D:\TestSrecFiles\1M_at_0.S
- File Type: Auto
- [✓] Restrict to Addresses in this Range
  - Start: 0x FFE00000
  - End: 0x FFFFFFFF
- [✓] Apply Address Offset
  - Address: 0x FFE00000

f) Figure 22 definitions:

- Restrict to Addresses in this Range: Specifies the address range of the flash device
- Apply Address Offset: Specifies the start address, where the test data is programmed in the flash; it should be the flash device start address.

g) Click **Update Program Action**.

h) Repeat steps d), e), and f) for Verify action.

i) Execute the task.

j) Check the Console view for the algorithms used in execution of the task, as shown in Listing 11.

Listing 11. Programming test log

```
Downloading 0x00002800 bytes to be programmed at 0xFFEFD800
 Writing Program Function Code
 Writing the Address
 Writing the Size
 Writing the address of the buffer
 Clearing the status
 Setting up Registers
 Commanding target to run
 Programming ...
 Reading program return status
 Program Command Succeeded
    log: Programmed total of 0x00100000 bytes
    log:
    log: Program Command Succeeded
.
.
Uploading 0x00010000 bytes starting from address 0xFFEF0000
    log: Uploading 0x00010000 bytes starting from address 0xFFEF0000
    log: Verified total of 0x00100000 bytes
    log:
 Verify Command Succeeded
```

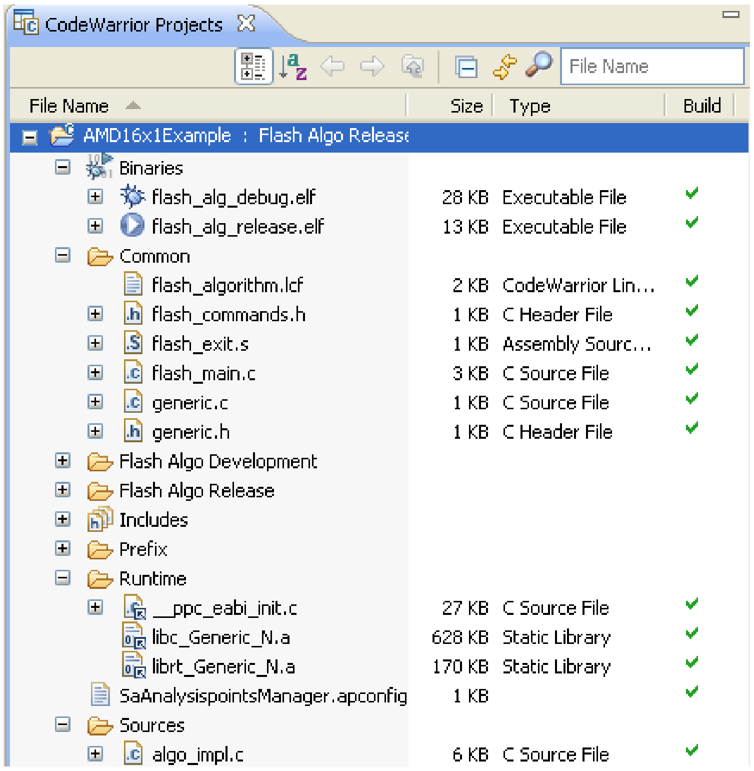
k) If all program/verify actions pass correctly, you have completed creation of a new flash programming algorithm. The new flash device can be programmed with the CodeWarrior Flash Programmer without any limitation.

7. Flash Programming Examples

7.1. Flash Programming Algorithm for AMD 16x1 Flash Devices

The AMD16x1Example project (Figure 23) illustrates how the Flash Development Kit is used to program AMD 16x1 flash algorithm.

Figure 23. Power Architecture CodeWarrior Projects View of AMD16x1Example Project



File Name	Size	Type	Build
AMD16x1Example : Flash Algo Releas			
Binaries			
flash_alg_debug.elf	28 KB	Executable File	✓
flash_alg_release.elf	13 KB	Executable File	✓
Common			
flash_algorithm.lcf	2 KB	CodeWarrior Lin...	✓
flash_commands.h	1 KB	C Header File	✓
flash_exit.s	1 KB	Assembly Sourc...	✓
flash_main.c	3 KB	C Source File	✓
generic.c	1 KB	C Source File	✓
generic.h	1 KB	C Header File	✓
Flash Algo Development			
Flash Algo Release			
Includes			
Prefix			
Runtime			
__ppc_eabi_init.c	27 KB	C Source File	✓
libc_Generic_N.a	628 KB	Static Library	✓
librt_Generic_N.a	170 KB	Static Library	✓
SaAnalysispointsManager.apconfig	1 KB		✓
Sources			
algo_impl.c	6 KB	C Source File	✓

7.1.1. Implementing ID Function for AMD 16x1 Flash Devices

The sequence for getting the Manufacturer ID and Device ID, based on the AMD flash specification, is shown in Table 5.

Table 5. ID Command Sequence for AMD Flash

Command Sequence	Cycles	Bus Cycles					
		First		Second		Third	
		Addr	Data	Addr	Data	Addr	Data

Manufacturer ID	4	555	AA	2AA	55	(BA)555	90
Device ID	6	555	AA	2AA	55	(BA)555	90

Listing 12. ID Function Sample Code for AMD Flashes

```
retval_t ID(parameter_block_t *p_pb)
{
    volatile unsigned short *baseaddress = (p_pb->base_addr).w;
    retval_t result = 0;

    /* reset */
    *(baseaddress) = (unsigned short)0xF0F0;

    /* setup for get id */
    *(baseaddress + 0x555) = (unsigned short)0xAA;
    *(baseaddress + 0x2AA) = (unsigned short)0x55;
    *(baseaddress + 0x555) = (unsigned short)0x90;

#if defined (DEBUG)
    /* get id */
    mf_id     = *(baseaddress);
    part_id   = *(baseaddress + 1);
#endif

    /* read mode again */
    *(baseaddress) = (unsigned char)0xF0;

    return result;
}
```

When using the Flash Algo Development build target, the device ID and manufacturer ID are read from the flash device and stored in the `part_id` and `mf_id` variables (Listing 12). Check these during the flash algorithm testing.

7.1.2. Implementing erase_sector Function for AMD 16x1 Flash Devices

The sequence for the `Sector Erase` command implementation, based on the AMD flash specification, is shown in Table 6.

Table 6. Sector Erase Command Sequence for AMD Flash

Command Sequence	Cycles	Bus Cycles											
		First		Second		Third		Fourth		Fifth		Sixth	
		Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Sector Erase	6	555	AA	2AA	55	555	80	555	AA	2AA	55	SA	30

See the actual encoding of the `erase_sector` function for AMD flashes in Listing 13.

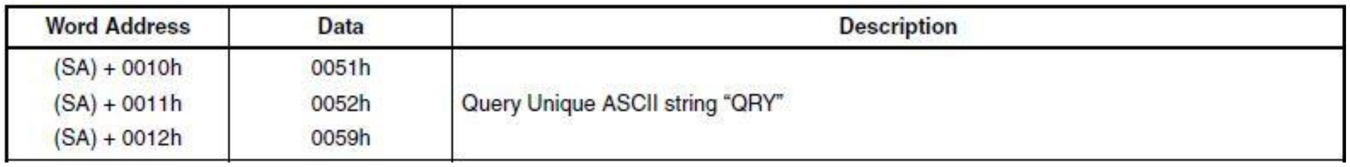
Listing 13. Function erase_sector Sample Code for AMD Flashes

```
retval_t erase_sector(parameter_block_t *p_pb, unsigned long sect_index)
{
    volatile unsigned short *sectoraddress = ((unsigned short **)(p_pb-
>items).w)[sect_index];
    volatile unsigned short read;
    retval_t result = 0;

    /* reset sector */
    *(sectoraddress) = (unsigned short)0xF0F0;
    /* identify */
    *(sectoraddress) = (unsigned short)0x9090;
    /* wait until results are ready */

    /* check if the sector is protected */
    if (* (sectoraddress+0x2)) return SECTOR_PROTECTED;

    /* first of all reset the device. The fID is no longer called in the new
     flash programmer plugin (it was used in the old AMC MWX-ICE) therefore
     the flash chip must always be brought into the read state.
    */

    /* reset sector */
    *(sectoraddress) = (unsigned short)0xF0F0;

    /* erase sector */
    *(sectoraddress + 0x555) = (unsigned short)0xAA;
    *(sectoraddress + 0x2AA) = (unsigned short)0x55;
    *(sectoraddress + 0x555) = (unsigned short)0x80;

    *(sectoraddress + 0x555) = (unsigned short)0xAA;
    *(sectoraddress + 0x2AA) = (unsigned short)0x55;
    *(sectoraddress) = (unsigned short)0x30;

    read = *(sectoraddress);

    /*
    Wait for the status value to be read from *addr or
    how_long ticks to pass.  If how_long ticks pass,
    a non-0 value will be returned.
    On the AMD chips, DQ7 is inverted until the embedded
    algorithm is completed when it flips to the correct
    value.  The parameter 'hi' will indicate whether that
    value is set or cleared.
    */
    while ((read & 0x0080) != 0x0080)
    {
        read = *(sectoraddress);
    }

    /* read mode again */
    *(sectoraddress) = (unsigned char)0xF0;
    return result;
}
```

7.1.3. Implementing erase_chip Function for AMD 16x1 Flash Devices

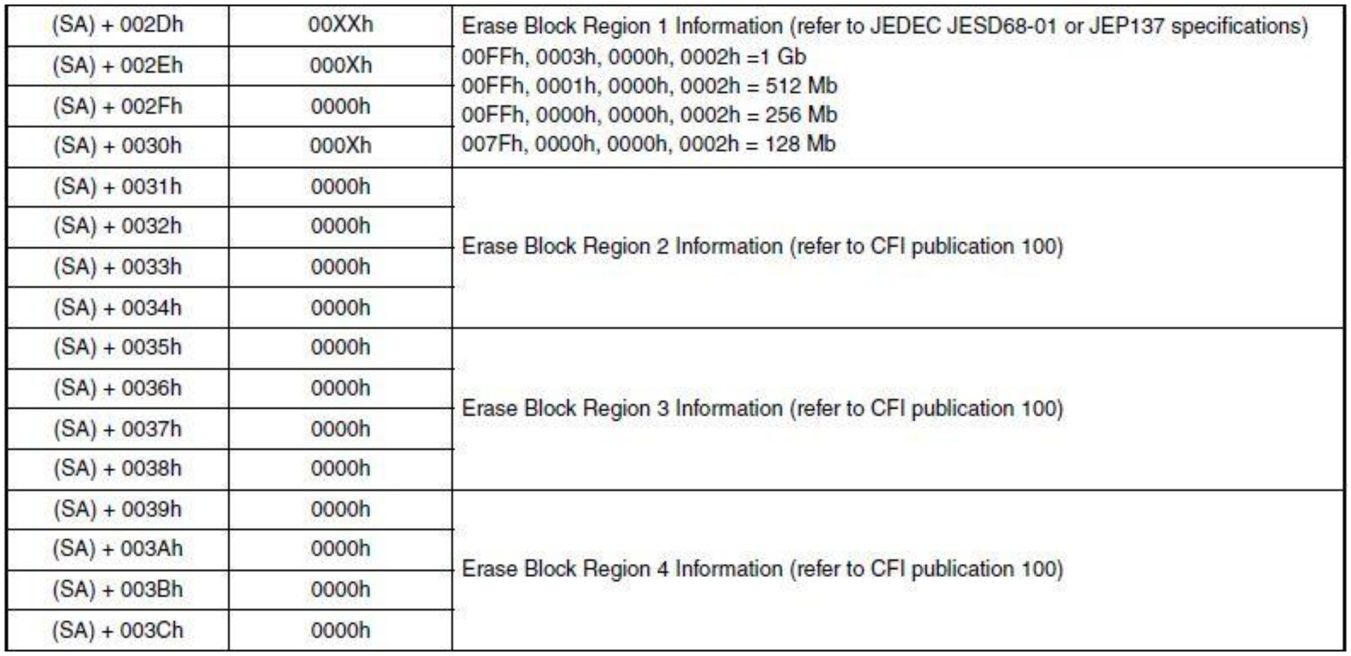
The sequence for the `Chip Erase` command, based on the AMD flash specification, is shown in Table 9. The Common Flash Interface commands are shown in Table 8Table 8 and Table 9. See the actual encoding of the `erase_chip` function for AMD flashes in the `algo_impl.c` file as shown in Listing 14.

Table 7. CFI Query Identification String Sequence for AMD Flash



Word Address	Data	Description
(SA) + 0010h	0051h	Query Unique ASCII string "QRY"
(SA) + 0011h	0052h	
(SA) + 0012h	0059h	

Table 8. CFI Device Geometry Definition for Regions Sequence for AMD Flash



(SA) + 002Dh	00XXh	Erase Block Region 1 Information (refer to JEDEC JESD68-01 or JEP137 specifications) 00FFh, 0003h, 0000h, 0002h =1 Gb 00FFh, 0001h, 0000h, 0002h = 512 Mb 00FFh, 0000h, 0000h, 0002h = 256 Mb 007Fh, 0000h, 0000h, 0002h = 128 Mb
(SA) + 002Eh	000Xh	
(SA) + 002Fh	0000h	
(SA) + 0030h	000Xh	
(SA) + 0031h	0000h	Erase Block Region 2 Information (refer to CFI publication 100)
(SA) + 0032h	0000h	
(SA) + 0033h	0000h	
(SA) + 0034h	0000h	
(SA) + 0035h	0000h	Erase Block Region 3 Information (refer to CFI publication 100)
(SA) + 0036h	0000h	
(SA) + 0037h	0000h	
(SA) + 0038h	0000h	
(SA) + 0039h	0000h	Erase Block Region 4 Information (refer to CFI publication 100)
(SA) + 003Ah	0000h	
(SA) + 003Bh	0000h	
(SA) + 003Ch	0000h	

Table 9. Chip Erase Command Sequence for AMD Flash

Command Sequence	Cycles	Bus Cycles											
		First		Second		Third		Fourth		Fifth		Sixth	
		Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Chip Erase	6	555	AA	2AA	55	555	80	555	AA	2AA	55	555	10

Listing 14. Function erase_chip encoding for AMD flashes

```
retval_t erase_chip(parameter_block_t *p_pb)
{
    int errors = 0;
    retval_t result = 0;
    unsigned short stat;
    unsigned short mask = (unsigned short)DQ7;
       unsigned short masked_src = (unsigned short)DQ7;
    int got_it;
    volatile unsigned short *base_addr = (p_pb->base_addr).w;

    /* reset sector */
    *(base_addr) = (unsigned short)0xF0F0;
    /* identify */
    *(base_addr) = (unsigned short)0x9090;
    /* wait until results are ready */

    *(base_addr) = (unsigned short)0xF0F0;
      /* CFI Entry command */
    *( (unsigned short *)base_addr + 0x55 ) = 0x0098;   /* write CFI entry command */

    /*
     * if the chip supports Common Flash Interface, then it answers with the Query Unique
ASCII string “QRY”
     * (0x51, 0x52, 0x59) after it receives the CFI entry command; you can read these
answers at
     * the next offsets: 0x10, 0x11, 0x12
     */

    /*
     * get the sector size and count of every four regions
     * address data   description
     * 2Dh      00xxh  Erase Block Region 1 Information
     * 2Eh      000xh  00FFh, 0003h, 0000h, 0002h = 1 Gb
     * 2Fh      0000h  00FFh, 0001h, 0000h, 0002h = 512 Mb
     * 30h      000xh  00FFh, 0000h, 0000h, 0002h = 256 Mb
     *                 007Fh, 0000h, 0000h, 0002h = 128 Mb
     * 31h      Erase Block Region 2 Information
     * 32h      Refer to CFI publication 100
     * 33h
     * 34h
     * 35h      Erase Block Region 3 Information
     * 36h      Refer to CFI publication 100
     * 37h
     * 38h
     * 39h      Erase Block Region 4 Information
     * 3Ah      Refer to CFI publication 100
     * 3Bh
     * 3Ch
     */

    /*  CFI Exit command  */
    *( (unsigned short *)base_addr + 0x000 ) = 0x00F0;   /* write cfi exit command */
    /* reset sector */
    *(base_addr) = (unsigned short)0xF0F0;
    /* identify */
    *(base_addr) = (unsigned short)0x9090;
    /* wait until results are ready */

    /*
     * you can find the total number of sectors and then for every sector you can check
     * the value from the address *(sectoraddress + 0x2) to see if that sector is protected
     */

    /* first of all reset the device. The fID is no longer called in the new
     flash programmer plugin (it was used in the old AMC MWX-ICE) therefore
     the flash chip must always be brought into the read state.
    */
    *base_addr = 0xF0F0;

       /* erase sector */
    *(base_addr + 0x555) = (unsigned short)0xAA;
    *(base_addr + 0x2AA) = (unsigned short)0x55;
    *(base_addr + 0x555) = (unsigned short)0x80;

    /* erase chip */
    *(base_addr + 0x555) = (unsigned short)0xAA;
    *(base_addr + 0x2AA) = (unsigned short)0x55;
    *(base_addr + 0x555) = (unsigned short)0x10;

       /* Wait for status operation */
    mask &= 0x0080;         /* Only dq7 flips */
    masked_src &= 0x0080;
    while ( 1 )
    {
        if ( (*base_addr & mask) == masked_src )
        {
             break;
        }
    }

    /* return to read mode */
    *base_addr = 0xf0;

    return result;
}
```

7.1.4. Implementing write Function for AMD 16x1 Flash Devices

In terms of AMD flash devices specification, the `write` function realizes the `Program` command. The sequence for the `Program` command, according to the AMD specification, is shown in Table 1010.

Table 10. Program Command Sequence for AMD Flash

Command Sequence	Cycles	Bus Cycles					
		First		Second		Third	
		Addr	Data	Addr	Data	Addr	Data
Program	4	555	AA	2AA	55	555	A0

The sequence for Common Flash Interface commands are shown in Table 8 and Table 9. They are used to check if any of the sectors involved in the writing process are protected before actualy writing the data.

See the actual encoding of the write function for AMD flashes in the `algo_impl.c` file as shown in Listing 15.

Listing 15. Sample write Function Code for AMD Flashes

```
retval_t write(parameter_block_t *p_pb)
{
    int timed_out, got_it;
    unsigned long i;
    unsigned short stat;
    retval_t errors = 0;
    unsigned short mask = (unsigned short)DQ7;
       unsigned short masked_src = (unsigned short)DQ7;
    volatile unsigned short *base_addr = (p_pb->base_addr).w;
    unsigned short *buffer = (p_pb->items).w;
    unsigned long buffer_len = p_pb->num_items;
    unsigned long how_many = buffer_len / sizeof(unsigned short);

   /* reset sector */
    *(base_addr) = (unsigned short)0xF0F0;
    /* identify */
    *(base_addr) = (unsigned short)0x9090;
    /* wait until results are ready */

    *(base_addr) = (unsigned short)0xF0F0;
       /* CFI Entry command */
    *( (unsigned short *)base_addr_align + 0x55 ) = 0x0098;   /* write CFI entry command */

    /*
     * Add code: if the chip supports Common Flash Interface, then it answers with the Query
Unique ASCII
     * string QRY(0x51, 0x52, 0x59) after it receives the CFI entry command; you can read
these answers at
     * the next offsets: 0x10, 0x11, 0x12
     */

    /*
     * Add code: get the sector size and count of every four regions
     * address data   description
     * 2Dh      00xxh  Erase Block Region 1 Information
```

```
 * 2Eh     000xh  00FFh, 0003h, 0000h, 0002h = 1 Gb
 * 2Fh      0000h  00FFh, 0001h, 0000h, 0002h = 512 Mb
   * 30h    000xh  00FFh, 0000h, 0000h, 0002h = 256 Mb
   *                            007Fh, 0000h, 0000h, 0002h = 128 Mb
   * 31h                Erase Block Region 2 Information
   * 32h                Refer to CFI publication 100
 * 33h
 * 34h
 * 35h                  Erase Block Region 3 Information
 * 36h                  Refer to CFI publication 100
 * 37h
 * 38h
 * 39h                  Erase Block Region 4 Information
 * 3Ah                  Refer to CFI publication 100
 * 3Bh
 * 3Ch
 */

/*  CFI Exit command  */
*( (unsigned short *)base_addr_align + 0x000 ) = 0x00F0;   /* write cfi exit command */

/* reset sector */
*(base_addr) = (unsigned short)0xF0F0;
/* identify */
*(base_addr) = (unsigned short)0x9090;
/* wait until results are ready */

/*
 * Add code: you can iterate through each sector that is involved in the writing process
 * and check the address *(base_addr + 0x2) to see if that sector is protected
 */

if ( buffer_len % sizeof(unsigned short) ) {
    /* we need to fill the remaining bytes with 'ff' -- this assumes
    byte accesses to DRAM will work */
    char *p = (char *)((unsigned long)buffer + buffer_len);
    *p++ = '\xff';
    how_many++ ;
}

/* first of all reset the device. The fID is no longer called in the new
 flash programmer plugin (it was used in the old AMC MWX-ICE) therefore
 the flash chip must always be brought into the read state.
*/
*base_addr = (unsigned short)0xf0f0;

for (i = 0; (i < how_many) && !errors; i++){

    unsigned short *c = (unsigned short*)((unsigned long)base_addr & ~0x1fff);
    *((c) + 0x555) = 0xaa;
    *((c) + 0x2aa) = 0x55;
    *((c) + 0x555) = 0xa0;
```

```
        *base_addr = *buffer;
             /* Wait for status operation */
          mask &= 0x0080;          /* Only dq7 flips */
          masked_src = (unsigned short)((unsigned char)DQ7 & *buffer);
          masked_src &= 0x0080;
          while ( 1 )
          {
              if ( (*base_addr & mask) == masked_src )
              {
                   break;
              }
          }

        base_addr++;
        buffer++;
    }

    /* go back to the last access */
    --base_addr;

    /* read mode again */
    *base_addr = (unsigned char)0xf0;

    return errors;
}
```

7.2. Flash Programming Algorithm for NAND eLBC – K9F5608x0D

The Algorithm and the Utility for this NAND device can be found in NAND Example folder and were made for eLBC controller and P1020RDB board. Both the algorithm and the utility were implemented in the same project, because the Utility is not common for NAND devices and is specific for each device. This project is formed of three parts and the user must complete only the files from *Sources* folder.

NOTE When you update this algorithm for another board, ensure to update the eLBC.h file (it is very important to have eLBC_OFFSET updated with the processor reference manual and IMMR_BASE updated with the initialization file settings).

7.3. Flash Programming Algorithm for NAND IFC – K9F5608x0D

The algorithm and utility for this NAND device can be found in NAND IFC Example folder and were made for IFC controller and P1010RDB board. Both the algorithm and the utility were implemented in the same project, because the utility is not common for NAND devices and is specific for each device. This project is formed of three parts and the user must complete only the files from *Sources* folder.

NOTE When you update this algorithm for another board, ensure to update the ifc.h file (it is very important to have the IFC_NAND_OFFSET updated with the processor reference manual and CCSRBAR updated with the initialization file settings).

7.4. Flash Programming Algorithm for SPI – S25FL128P

The algorithm and utility for this SPI device can be found in SPI Example folder and were made for eSPI controller and P1024RDB board. Both the algorithm and the utility were implemented in the same project, because the utility is not common for SPI devices and is specific for each device.
This project is formed of three parts and the user must complete only the files from *Sources* folder.

NOTE When you update this algorithm for another board, Ensure to update the device_defs.h file (it is very important to have the ESPIBASE updated with the processor reference manual and initialization file settings—this is a sum between CCSRBAR and eSPI register group address).

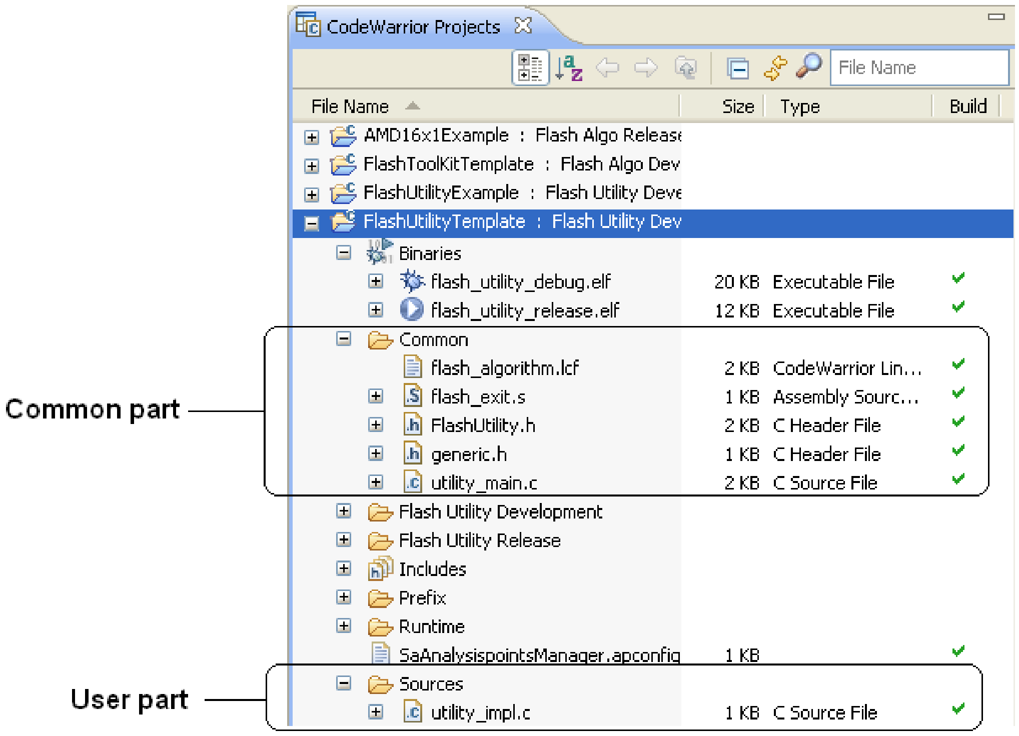
8. Creating a New Flash Utility

Some flash devices, such as NAND and SPI are not memory mapped. The memory cannot be read directly; therefore, a special utility is needed for blank check and checksum operations. This is very similar to a flash algorithm. It receives the commands through a data structure and passes back the results through the same structure.

8.1. Flash Utility Template Overview

The Flash Utility Template described in this document also helps you develop flash utilities for the CodeWarrior Flash Programmer, as shown in Figure 24. This section provides important information needed before you begin creating a flash utility.

Figure 24. Flash Utility



CodeWarrior Projects

File Name

File Name	Size	Type	Build
AMD16x1Example : Flash Algo Releas			
FlashToolKitTemplate : Flash Algo Dev			
FlashUtilityExample : Flash Utility Deve			
FlashUtilityTemplate : Flash Utility Dev			
Binaries			
flash_utility_debug.elf	20 KB	Executable File	✓
flash_utility_release.elf	12 KB	Executable File	✓
Common			
flash_algorithm.lcf	2 KB	CodeWarrior Lin...	✓
flash_exit.s	1 KB	Assembly Sourc...	✓
FlashUtility.h	2 KB	C Header File	✓
generic.h	1 KB	C Header File	✓
utility_main.c	2 KB	C Source File	✓
Flash Utility Development			
Flash Utility Release			
Includes			
Prefix			
Runtime			
SaAnalysispointsManager.apconfig	1 KB		✓
Sources			
utility_impl.c	1 KB	C Source File	✓

8.2. Flash Utility General Structure

The Flash Utility application is divided into three different sets of files:

1. Flash Utility Common Files (No Modification Needed): Contains initialization and other files. This component is common for all flash devices and you should not change it while developing the new flash programming algorithm. It consists of the following files:
   - o `flash_algorithm.lcf` file: Linker command file, which is set up according to the rules for flash programming applet allocation in physical memory.
   - o `utility_main.c`: Main function and API to the CodeWarrior Flash Utility.
   - o `FlashUtility.h`: Header file with the generic data structures and definitions used by flash utilities.
   - o `flash_exist.s`: Exit point for the flash programming applet.
   - o `generic.h`: Header file that contains general data structures and definitions used by all flash programming algorithms.
2. User Files (Implement Utility): Contains flash device specific files. This component is modified for any flash devices depending on the flash utility to be used. It consists of the following files:
   - o `Utility_impl.c`: Includes functions to implement for the flash device utility, such as **executeBlankCheck** and **executeCheckSum**.

3. Run-Time Files :
   - `ppc_eabi_init.c`: Includes flash programmer start-up initialization file.

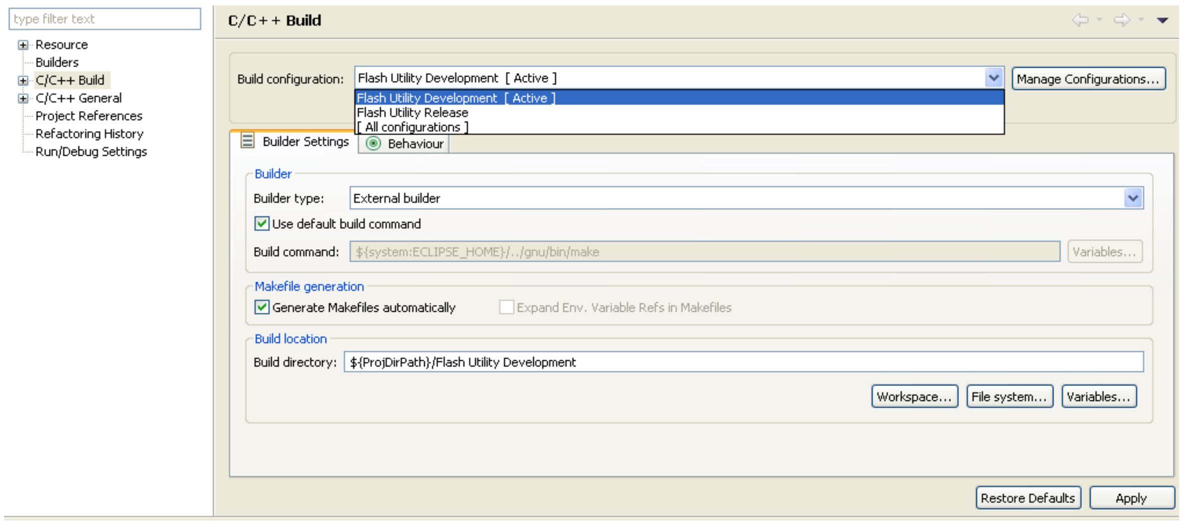
To create a new utility for flash programming, make changes only to the `utility_impl.c` file.

8.3. Flash Utility Build Targets

Several build targets are predefined in the Flash Utility Template:

- **Flash Utility Development:** Flash utility development and test application. The **ELF** executable file, created in Flash Utility Development, should be used to develop, debug, and test the new CodeWarrior Flash Programmer utility.
- **Flash Utility Release:** To create flash utility applet, the CodeWarrior Flash Programmer uses the **ELF** executable file, created in Flash Utility Release. This build target shares the flash device utility with the Flash Utility Development build target; it differs, however, because it cannot be debugged or tested (Figure 25).

Figure 25. Flash Utility Targets



Build configuration: Flash Utility Development [Active]
Flash Utility Development [Active]
Flash Utility Release
[All configurations]

Builder type: External builder
☑ Use default build command
Build command: ${system:ECLIPSE_HOME}/../gnu/bin/make
☑ Generate Makefiles automatically
☐ Expand Env. Variable Refs in Makefiles
Build directory: ${ProjDirPath}/Flash Utility Development

8.4. Flash Utility API

The CodeWarrior Flash Programmer communicates with the flash utility applet through two different commands:

- Blank Check
- Checksum

The same memory zone defined for Flash Device Algorithm, the `parameterBlockType` structure, is used by the utility.

On the flash utility side, the commands from the CodeWarrior Flash Programmer go through the `parameterBlockType` structure, and are mapped in the memory, from the beginning of the memory buffer.
All commands from the CodeWarrior Flash Programmer are already encoded in `utility_main.c` file. This file can be used for the new flash programming algorithm without any modification. After loading the utility applet to the target board, the CodeWarrior Flash Programmer writes all parameters right in the data structure located at the beginning of the memory buffer.

For the detailed description of the `parameterBlockType` structure, see Listing 16.

Listing 16. parameterBlockType Structure Details

```
typedef volatile struct  {
  unsigned long     magicNumber; /* Magic id code used to verify image    */
  unsigned long     function; /* What function to perform ?    */
  unsigned long     result_status; /* Status of the operation */
  pointer_type    start_address;  /* start address of the operation      */
  pointer_type      end_address; /* end address of the operation         */
  unsigned long   numBlankCheckErrors; /* total number of blank check errors found    */
  unsigned long     numRecordedBlankCheckErrors;     /* number of mismatches recorded
       */
  mismatchErrorType   *mismatches;      /* address of the array of mismatches*/
  unsigned long     checksumValue; /* intput and output checksum value    */
  unsigned long     baseAddress; /* Base address of the flash */
  unsigned long     ccsrbar; /* Base address for memory mapped registers */
  unsigned long     core_type; /* Core Type - used for handle between different processor FP
controller structs */
  unsigned long     controller_offset; /* Controller Offset */
  unsigned long     controller_type; /* Controller Type */
  unsigned long     additionalParameterNum; /* Number of additional parameters sent by FP
Plugin */
  unsigned long     additionalParameterStart; /* First additional parameter */
  unsigned long     padding[7]; /* Padding for other additional parameters */ }
parameterBlockType;
```

Listing 16 definitions:

- `magicNumber`: Indicates a number written at the beginning of the flash utility parameter block. The CodeWarrior Flash Programmer reads the first location from the memory buffer upon downloading the utility. The expected value is `0xBCC5BCC5`.
- `function`: Specifies the command to be executed. It can be blank check or checksum.
- `result_status`: Contains the operation result. It can be success, fail, or unknown command.
- `start_address`: Specifies the start address for the requested command.
- `end_address`: Specifies the end address for the requested command.
- `numBlankCheckErrors`: Stores the number of errors found during the blank check operation.
- `numRecordedBlankCheckErrors`: Indicates the number of errors that have been recorded during blank check. Up to 12 errors are recorded.

- mismatches: Indicates a pointer to the mismatchErrorType structure defined in Listing 17. It contains all errors recorded.
- checksumValue: Specifies the checksum computed.
- baseAddress: Specifies the base address of the flash.
- ccsrbar: Specifies is the base address for memory mapped registers.
- core_type: Helps to handle between different processor Flash Programmer controller structures.
- controller_offset: Indicates the controller offset.
- controller_type: Gives the controller type.
- additionalParameterNum: Indicates the number of additional parameters sent by Flash Programmer Plugin.
- additionalParameterStart: Specifies the first additional parameter.
- padding[7]: Is reserved for other additional parameters.

Listing 17. mismatchErrorType Structure Details

```
typedef volatile struct {
  pointer_type    address;    /* where the error/mismatch occured */
  unsigned long   expected;
  unsigned long   actual;
} mismatchErrorType
```

Listing 17 definitions:

- address: Represents a structure of pointer_type defined in Listing 18. It contains the address where an error has been found.
- expected: Indicates the value expected to be found in flash memory.
- actual: Indicates the value actually found in flash memory.

Listing 18. pointer_type Structure Details

```
typedef union {unsigned char*        c;
               unsigned short*       w;
               unsigned long*        l;
               unsigned long long*  ll;
               void*                 e;} pointer_type;
```

The pointer_type is a somewhat generic pointer. We can use this pointer to get any data type—unsigned char, short, long, long long, or void.

The supported functions are:

BlankCheck

The BlankCheck command is called by the flash programmer to blank check a memory range. For the BlankCheck command, the CodeWarrior Flash Programmer:

- Loads the flash utility on the target board
- Sets the `BlankCheck` command as shown in the function field of Listing 16
- Sets the `start_address` and `end_address` parameters as shown in the function field of Listing 16
- Runs the flash utility applet
- Waits until flash utility stops execution
- Checks the status of the command being executed as shown in `result_status` from Listing 16
- Reads the number of errors and records the errors and their location if the status reports an error, as shown in Listing 16

Checksum

The `Checksum` command is called by the flash programmer to blank check a memory range. For the `Checksum` command, the CodeWarrior Flash Programmer:

- Loads the flash utility on the target board
- Sets the `Checksum` command as shown in the function field of Listing 16
- Sets the `start_address` and `end_address` parameters as shown in the function field of Listing 16
- Runs the flash utility applet
- Waits until flash utility stops execution
- Checks the status of the command being executed as shown in `result_status` from Listing 16
- Reads the checksum result if status is success, as shown in Listing 16

8.5. Create New Flash Utility

To create a new CodeWarrior Flash Programmer flash utility using the Flash Utility Template, for a flash device not supported by the CodeWarrior software:

1. Store the original version of Flash Utility Template available in the CodeWarrior delivery to a different working directory. The location of the FlashUtilityTemplate folder is: *`{CodeWarrior}\PA\PA_Tools\FlashToolKit\FlashUtilityTemplate`*
2. Import Flash Utility Template project:
   a) Select **File→Import** from the IDE menu bar. The **Import** dialog box appears.
   b) Select **General→Existing Projects into Workspace**. Click **Next**. The **Import Projects** page appears.
   c) Click **Browse** to choose the parent folder where you have copied the `FlashUtilityTemplate` folder, select the folder, and click OK. The **Projects** list

gets populated with the projects available in the `FlashUtilityTemplate` folder.

d) Clear the check boxes next to the projects that you do not want to import and click **Finish**. The imported project is displayed in the **CodeWarrior Projects** view in the C/C++ perspective, as shown in Figure 24.

e) Ensure that the **Flash Utility Development** build target is selected.

3. Select **Run→Debug Configurations**. The **Debug Configurations** dialog box appears.
4. Expand the **CodeWarrior Download** tree node and select the desired launch configuration.
5. Click the **Debugger** tab in the right panel.
6. Select the required target processor from the **Target Processor** drop-down list.
7. Specify required target initialization and memory configuration files for the connected hardware in the **Target initialization file** and **Memory Configuration File** text boxes. For supported Freescale Evaluation Boards, you can use the debugger config files (*.tcl), and the debugger mem files (*.mem) available with the CodeWarrior Development Studio. Check the folder: *`{CodeWarrior}\PA\PA_Support\Initialization_Files`*.
8. Modify `executeBlankCheck` function from `utility_impl.c` file. The function must read the command options, perform the blank check and put the results in parameter block. See Modify ID function in the algo_impl.c file: for an implementation example.
9. Modify `executeCheckSum` function from `utility_impl.c` file. The function must read the command options, perform the checksum, and put the results back in parameter block. See Modify ID function in the algo_impl.c file: for an implementation example.
10. Compile Flash Utility Development target. While creating and testing a new utility, use the Flash Utility Development build target of the Flash Utility Template project. Compilation will result in creation of a new `flash_utility_debug.elf` file.
11. Flash utility unit testing:

    a) To test a flash utility algorithm, define custom flash device parameters in `utility_main.c`. You can select the operation you want to perform and the address in flash to which the operation will be applied.

    b) Load the `FlashUtilityTemplateDebug.elf` file and run it on a target board. Check the tests results. As an example of the test working, see Modify ID function in the algo_impl.c file:

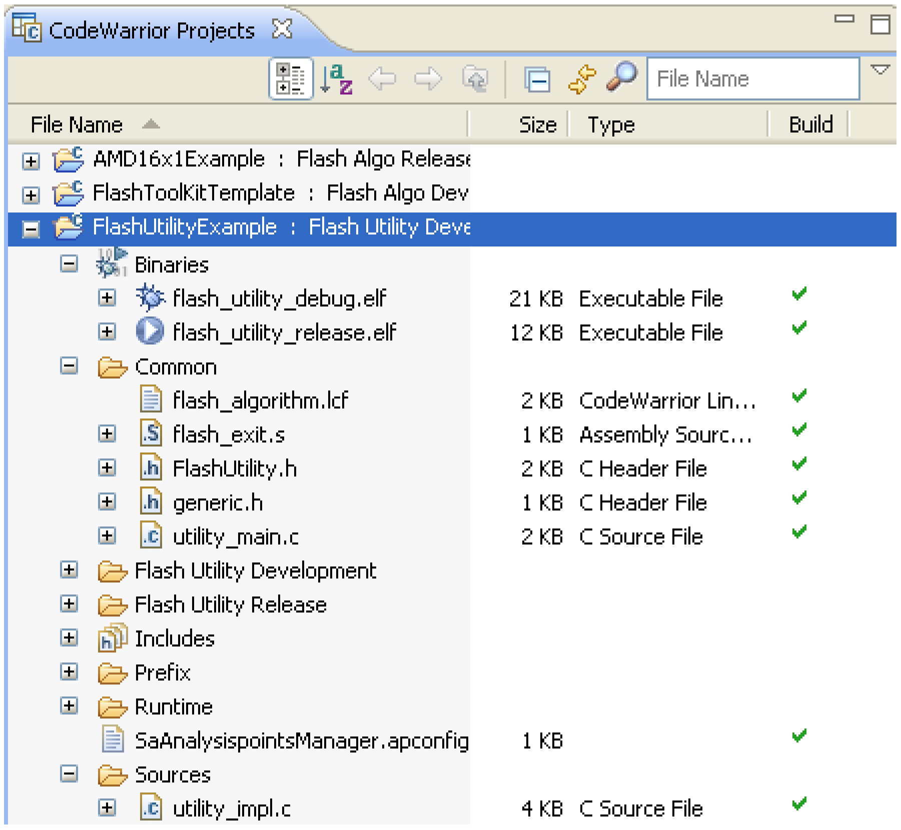
12. Compile Flash Utility Release target. Copy the resulted file in the following folder *`{CodeWarior}\PA\bin\plugins\support\Flash_Programmer\EPPC`*
13. Use the new Flash Utility in the device description XML file. Run a Flash Programmer with the new device / utility as described for the flash algorithm in Create a New Flash Programming Algorithm.

9. Flash Utility Examples

9.1. Flash Utility Example for NOR Flash Devices

The *FlashUtilityExample* project (Figure 26) implements a flash utility for the NOR flash devices.

Figure 26. FlashUtilityExample Project



File Name	Size	Type	Build
AMD16x1Example : Flash Algo Release			
FlashToolKitTemplate : Flash Algo Dev			
FlashUtilityExample : Flash Utility Deve			
Binaries			
flash_utility_debug.elf	21 KB	Executable File	✓
flash_utility_release.elf	12 KB	Executable File	✓
Common			
flash_algorithm.lcf	2 KB	CodeWarrior Lin...	✓
flash_exit.s	1 KB	Assembly Sourc...	✓
FlashUtility.h	2 KB	C Header File	✓
generic.h	1 KB	C Header File	✓
utility_main.c	2 KB	C Source File	✓
Flash Utility Development			
Flash Utility Release			
Includes			
Prefix			
Runtime			
SaAnalysispointsManager.apconfig	1 KB		✓
Sources			
utility_impl.c	4 KB	C Source File	✓

9.1.1. Implementing executeBlankCheck Function

Listing 19 shows the implementation of the function.

Listing 19. Implementation of executeBlankCheck Function

```
void executeBlankCheck()
{
        unsigned char  *endAddress;
        unsigned char  *currentAddress;
        unsigned long recordedMismatches;
        unsigned long        totalMismatches;

        // Retrieve the operating bounds from the parameter block
        currentAddress      = gParams.start_address.c;
        endAddress          = gParams.end_address.c;

        recordedMismatches = 0;
        totalMismatches        = 0;

        while ( currentAddress <= endAddress )
        {
                if ( *currentAddress != FLASH_UTILITY_ERASED_VALUE )
                {
                        totalMismatches++;

                        if ( recordedMismatches < FLASH_UTILITY_MAX_MISMATCH_ERRORS )
                        {
                                gMismatches[recordedMismatches].address.c      = currentAddress;
                                gMismatches[recordedMismatches].expected       =
FLASH_UTILITY_ERASED_VALUE;
                                gMismatches[recordedMismatches].actual         = *currentAddress;

                                recordedMismatches++;
                        }
                }

                // ensure we don't have a problem getting out of the
                // loop at 0xFFFFFFFF

                if ( currentAddress == endAddress )
                        break;

                currentAddress++;
        }

        // Update the output parameters for the flash programmer to read ...

        if ( totalMismatches != 0 )
     {
                gParams.result_status                          =
FLASH_UTILITY_STATUS_BLANKCHECK_FAIL;
     }
        else
     {
                gParams.result_status                          = FLASH_UTILITY_STATUS_SUCCESS;
     }

        gParams.numBlankCheckErrors             = totalMismatches;
        gParams.numRecordedBlankCheckErrors     = recordedMismatches;
}
```

The function reads from `start_address` to `end_address` and compares all memory values with the erased memory values. The errors equal to the value of the parameter `FLASH_UTILITY_MAX_MISMATCH_ERRORS` are stored and returned through the parameter block structure.

9.1.2. Implementing executeCheckSum Function

Listing 20 contains the implementation of the function.

Listing 20. Implementation of executeCheckSum Function

```
void executeCheckSum()
{
	unsigned char  *endAddress;
	unsigned char  *currentAddress;
	unsigned long checksumValue;

	// Retrieve the operating bounds from the parameter block
	currentAddress		= gParams.start_address.c;
	endAddress		= gParams.end_address.c;
	checksumValue		= gParams.checksumValue;

	while ( currentAddress <= endAddress )
	{
		checksumValue += *currentAddress;

		if ( currentAddress == endAddress )
			break;

		currentAddress++;
	}

	// Update the output parameters
	gParams.checksumValue = checksumValue;

	// Update the output status
	gParams.result_status = FLASH_UTILITY_STATUS_SUCCESS;
}
```

The function computes the checksum between `start_address` and `end_address`, and then returns the result through the parameter block structure.

10. Troubleshooting the Flash Programmer

It is possible that the CodeWarrior™ Flash Programmer does not recognize the flash devices on your target, or has a problem with erasing or programming. If so, use the troubleshooting techniques in this topic to ensure that basic reads and writes to flash function correctly. If you still cannot program your flash devices, contact Technical Support at:
`http://www.freescale.com/support`.

This topic explains how to configure your target flash devices to display their manufacturer and device ID codes. If the devices can display this information, then basic reads and writes to the devices are functioning correctly. This means that you are unable to program your flash due to either:

- The flash device configuration file
- The flash-programming algorithm

10.1. Theory

Current flash devices use a common method for preventing unintentional programming. A specific sequence of write cycles must precede each flash programming write cycle, to enable programming of one byte (8 bits) or word (16 bits). These preceding write cycles walk through an internal-state machine that enables the flash for one device-programming write cycle. This write-enabling process is necessary for each flash address to be programmed.
You can use the same method to configure a flash device to display its manufacturer and device ID codes:

- The manufacturer ID code is the same for all devices from each manufacturer. Common codes are `0x01` for AMD, `0x1F` for Atmel, and `0x89` for Intel.
- Device ID code is unique for each device; each device's data sheet specifies its ID code.

Reading these ID codes requires successful write and read cycles to each flash device. This indicates that the flash-programming problems exist in the CodeWarrior Flash Programmer rather than in the target hardware. Similarly, inability to read these ID codes indicates a low-level problem with reading/writing target flash. You must resolve such hardware problems for the flash programmer to work.

10.2. Practice

The explanation of this document involves using the CodeWarrior debugger and its Command Line panel to write to and to read from target flash. Before you begin, you must be running a debug session to the target.

The process uses the CodeWarrior commands, `change`, which writes to memory, and `display`, which reads from memory. The general formats are:

```
change p:<address><value><bus-width>
display p:<start_address>..<end_address><bus-width>
```

where:

- `<address>, <start_address>`, and `<end_address>` are address values that comply with the CodeWarrior default radix, or with an explicitly defined radix.
- *`<value>`* is a data value that complies with the CodeWarrior default radix, or with an explicitly defined radix.
- *`<bus_width>`* is 8bit, 16bit, 32bit, or 64bit.

These commands are used to push flash commands to the devices on your target and then to read the data that the flash presents. All flash devices use a flash-command state machine to process commands, such as ReadDevice ID, Erase Sector, and Program. See the data sheet for details of the device command sequences.

This section presents common command sets for reading manufacturer and device IDs. To use any of these examples, you must change the strings %%% to the high-order starting address of the flash on your target. For example, if the flash base address is 0xffe00000, your replacement string for %%% is ffe. The data values in these examples must remain as listed.

To use these command sets:

- Enter each of the `change` and `display` commands exactly as listed, except for substituting the correct high-order address string for %%%.
- Examine the information you obtain from the `display` command for the ID codes defined in the data sheet for your flash device.

If the manufacturer of your target flash device is not AMD, Atmel, or Intel, then compare the command sequences of your devices with the command sequences of the devices manufactured by AMD, Atmel, and Intel. Most likely, your device uses the command sequence of one of these device manufacturers. Fujitsu flash devices, for example, use the same commands as AMD devices. Sharp flash devices use the same commands as Intel devices. And even if your flash device does not use exactly the AMD, Atmel, or Intel command sequence, the command sequence should be nearly similar. You can, therefore, easily adapt the command sequences of other device manufacturers.

10.3. Using the CodeWarrior Script Files

The CodeWarrior source command reads the contents of a text file as a list of sequential CodeWarrior commands. Accordingly, you may copy the contents of any listing, below, to a text file with extension .tcl. Then, use the source command to invoke the script. See examples 1 and 2 for more information.

10.4. Before You Start

Before you can start diagnosing flash problems, gather as much of this information as possible:

- Device manufacturer, such as AMD, Atmel, or Intel
- Device part number
- Number of devices on your target
- Number of data bits (8 or 16) each flash device uses
- Starting flash address on the target

NOTE You can also troubleshoot your flash device without the information listed above. However, that would increase your efforts. The instructions below are consistent with you having all this information.

To ensure that your ID-value interrogation does not fail:

1. Writes to flash and reads from flash must occur exactly as the manufacturer defines for reading out the manufacturer and device ID codes. Ensure that you disable all address-translation and memory management features.
2. Disable all processor caches. (You must write to/read from the actual flash devices, not a cached copy of flash.)
3. Check the target schematic, to ensure that each WE# (write-enable) processor signal reaches the correct WE# pin of each target flash device. The target hardware, the target-processor configuration, or both, can disable the WE# signal.
4. Check the memory-control registers of the target processor, to ensure that flash accesses are not read-only.
5. For a 16-bit flash device, determine whether the processor's least-significant address line is connected to the flash device. If so, you can rely on the addresses of the flash data sheet. An example is the AMD AM29LV640D/AM29LV641D data sheet, which specifies this sequence for reading device and manufacturer ID codes:
   ```
   %%%0555 = AA
   %%%02AA = 55
   %%%0555 = 90
   ```
   But many processors do not have the least-significant address line connected to the flash device, as there is no reason to address individual bytes. If this is the arrangement for your target, you must compensate by shifting data sheet addresses left by one. This would change the sequence above to:
   ```
   %%%0AAA = AA
   %%%0554 = 55
   %%%0AAA = 90
   ```
6. Confirm that data bus least significant bit of each flash device connects to the least significant bit of the processor. The CodeWarrior Flash Programmer does not support reverse wired flash devices.

10.4.1. AMD Devices

Some AMD devices are dual-mode, supporting either 8 bit or 16 bit modes. To determine the mode, check the state of the flash BYTE pin: BYTE = 0 means 8 bit mode configuration; BYTE = 1 means 16 bit mode configuration.

Listing 21 through Listing 29 provides command sequences for AMD flash devices.

10.4.2. Atmel Devices

Listing 30 through Listing 35 provides command sequences for Atmel flash devices.

10.4.3. Intel Devices

Listing 36 through Listing 41 provides command sequences for Intel flash devices. The status register read outs of these listings is not required for reading out the ID codes. However, if you enter these commands correctly, the status register results show an operation successful status or possible chip errors.
Some Intel devices are dual-mode, supporting either 8 bit or 16 bit modes. To determine the mode, check the state of the flash BYTE# pin: BYTE# = 0 means 8 bit mode configuration; BYTE# = 1 means 16 bit mode configuration.

10.5. Procedure

Perform these steps:

1. Start a CodeWarrior debugging session for your target.
2. Identify the appropriate command sequence for your device using Table 9.
   a) If your manufacturer is Fujitsu, use the AMD listing for your arrangement.
   b) If your manufacturer is Sharp, use the Intel listing for your arrangement.
   c) Otherwise, find the closest match for your device arrangement, so that you can modify the command sequence explained in Step 3.
3. Substitute the high order address string with %%% in the code of the selected listing and perform either of the following:
   a) Enter the listing commands one after another, in the Debugger Shell view, or
   b) Copy the commands, paste them into a .tcl text file, then use a source command in the Debugger Shell view to invoke the new script.
4. In the output of the display command, look for the ID codes of your flash device. The device's data sheet specifies these code values.
   a) If the output includes the ID codes, you have confirmed that flash device basic reads and writes function properly. This means that any programming problem lies with the CodeWarrior software, so you should report the matter to Freescale Technical Support: `www.freescale.com/support`.
   b) If the output does not include the ID codes, you have confirmed a low-level problem with reading from or writing to your flash devices. You must solve this problem locally for the CodeWarrior Flash Programmer to work.

Two examples follow the command sequence listings.

10.6. Command Sequence Listings

Table 9 lists various flash device arrangements and the corresponding command sequences.

Table 11. Flash Device Command Sequences

Manufacturer	Devices	See
AMD	One 8-bit device	Listing 21
	One 8-bit/16-bit device, in 8-bit mode	Listing 22
	Two 8-bit devices	Listing 23
	Two 8-bit/16-bit devices, in 8-bit mode	Listing 24
	Four 8-bit devices	Listing 25
	Four 8-bit/16-bit devices, in 8-bit mode	Listing 26
	One 16-bit device	Listing 27
	Two 16-bit devices	Listing 28
	Four 16-bit devices	Listing 29
Atmel	One 8-bit device	Listing 30
	Two 8-bit devices	Listing 31
	Four 8bit devices	Listing 32
	One 16-bit device	Listing 33
	Two 16-bit devices	Listing 34
	Four 16-bit devices	Listing 35
Intel	One 8-bit device	Listing 36
	Two 8-bit devices	Listing 37
	Four 8-bit devices	Listing 38
	One 16-bit device	Listing 39
	Two 16-bit devices	Listing 40
	Four 16-bit devices	Listing 41

Listing 21. AMD: One 8-bit Device

```
# Set device to Read state
change p:%%%00000 f0 8bit

# Get Mfg and Device ID values
change p:%%%00555 aa 8bit
change p:%%%002aa 55 8bit
change p:%%%00555 90 8bit

# Display Mfg ID value at offset
0# Display Dev ID value at offset 1
display p:%%%00000..%%%00002 8bit

# Reset device to Read state
change p:%%%00000 f0 8bit
```

Listing 22. AMD: One 8-bit/16-bit Device, in 8-bit Mode

```
# Set device to Read state
change p:%%%00000 f0 8bit

# Get Mfg and Device ID values
change p:%%%00aaa aa 8bit
change p:%%%00555 55 8bit
change p:%%%00aaa 90 8bit

# Display Mfg ID value at offset 0
# Display Dev ID value at offset 1
display p:%%%00000..%%%00002 8bit

# Reset device to Read state
change p:%%%00000 f0 8bit
```

Listing 23. AMD: Two 8-bit Devices

```
# Set devices to Read state
change p:%%%00000 f0f0 16bit

# Get Mfg and Device ID values
change p:%%%00aaa aaaa 16bit
change p:%%%00554 5555 16bit
change p:%%%00aaa 9090 16bit

# Display Mfg ID values at offsets 0, 1
# Display Dev ID values at offsets 2, 3
display p:%%%00000..%%%00004 8bit

# Reset devices to Read state
change p:%%%00000 f0f0 16bit
```

Listing 24. AMD: Two 8-bit/16-bit Devices, in 8-bit Mode

```
# Set devices to Read state
change p:%%%00000 f0f0 16bit

# Get Mfg and Device ID values
change p:%%%001554 aaaa 16bit
change p:%%%00aa8 5555 16bit
change p:%%%01554 9090 16bit

# Display Mfg ID values at offsets 0, 1
# Display Dev ID values at offsets 2, 3
display p:%%%00000..%%%00004 8bit

# Reset devices to Read state
change p:%%%00000 f0f0 16bit
```

Listing 25. AMD: Four 8-bit Devices

```
# Set devices to Read state
change p:%%%00000 f0f0f0f0 32bit

# Get Mfg and Device ID values
change p:%%%01554 aaaaaaaa 32bit
change p:%%%00aa8 55555555 32bit
change p:%%%01554 90909090 32bit

# Display Mfg ID values at offsets 0, 1, 2, 3
# Display Dev ID values at offsets 4, 5, 6, 7
display p:%%%00000..%%%00008 8bit

# Reset devices to Read state
change p:%%%00000 f0f0f0f0 32bit
```

Listing 26. AMD: Four 8-bit/16-bit Devices, in 8-bit Mode

```
# Set devices to Read state
change p:%%%00000 f0f0f0f0 32bit
# Get Mfg and Device ID values
change p:%%%02aa8 aaaaaaaa 32bit
change p:%%%01550 55555555 32bit
change p:%%%02aa8 90909090 32bit
# Display Mfg ID values at offsets 0, 1, 2, 3
# Display Dev ID values at offsets 4, 5, 6, 7
display p:%%%00000..%%%00008 8bit
# Reset devices to Read state
change p:%%%00000 f0f0f0f0 32bit
```

Listing 27. AMD: One 16-bit Device

```
# Set device to Read state
change p:%%%00000 f0f0 16bit

# Get Mfg and Device ID values
change p:%%%00aaa aaaa 16bit
change p:%%%00554 5555 16bit
change p:%%%00aaa 9090 16bit

# Display Mfg ID value at offset 0
# Display Dev ID value at offset 2
display p:%%%00000..%%%00004 16bit

# Reset device to Read state
change p:%%%00000 f0f0 16bit
```

Listing 28. AMD: Two-16 bit Devices

```
# Set devices to Read state
```

```
change p:%%%00000 f0f0f0f0 32bit

# Get Mfg and Device ID values
change p:%%%01554 aaaaaaaa 32bit
change p:%%%00aa8 55555555 32bit
change p:%%%01554 90909090 32bit

# Display Mfg ID values at offsets 0, 2
# Display Dev ID values at offsets 4, 6
display p:%%%00000..%%%00008 16bit

# Reset devices to Read state
change p:%%%00000 f0f0f0f0 32bit
```

Listing 29. AMD: Four 16-bit Devices

```
# Set devices to Read state
change p:%%%00000 f0f0f0f0f0f0f0f0 64bit

# Get Mfg and Device ID values
change p:%%%02aa8 aaaaaaaaaaaaaaaa 64bit
change p:%%%01550 5555555555555555 64bit
change p:%%%02aa8 9090909090909090 64bit

# Display Mfg ID values at offsets 0, 2, 4, 6
# Display Dev ID values at offsets 8, a, c, e
display p:%%%00000..%%%00010 16bit

# Reset devices to Read state
change p:%%%00000 f0f0f0f0f0f0f0f0 64bit
```

Listing 30. Atmel: One 8-bit Device

```
# Set device to Read state
change p:%%%05555 aa 8bit
change p:%%%02aaa 55 8bit
change p:%%%05555 f0 8bit

# Get Mfg and Device ID values
change p:%%%05555 aa 8bit
change p:%%%02aaa 55 8bit
change p:%%%05555 90 8bit

# Display Mfg ID value at offset 0
# Display Dev ID value at offset 1
display p:%%%00000..%%%00002 8bit

# Reset device to Read state
change p:%%%05555 aa 8bit
change p:%%%02aaa 55 8bit
change p:%%%05555 f0 8bit
```

Listing 31. Atmel: Two 8-bit Devices

```
# Set devices to Read state
change p:%%%0aaaa aaaa 16bit
change p:%%%05554 5555 16bit
change p:%%%0aaaa f0f0 16bit

# Get Mfg and Device ID values
change p:%%%0aaaa aaaa 16bit
change p:%%%05554 5555 16bit
change p:%%%0aaaa 9090 16bit

# Display Mfg ID values at offsets 0, 1
# Display Dev ID values at offsets 2, 3
display p:%%%00000..%%%00004 8bit

# Reset devices to Read state
change p:%%%0aaaa aaaa 16bit
change p:%%%05554 5555 16bit
change p:%%%0aaaa f0f0 16bit
```

Listing 32. Atmel: Four 8-bit Devices

```
# Set devices to Read state
change p:%%%15554 aaaaaaaa 32bit
change p:%%%0aaa8 55555555 32bit
change p:%%%15554 f0f0f0f0 32bit

# Get Mfg and Device ID values
change p:%%%15554 aaaaaaaa 32bit
change p:%%%0aaa8 55555555 32bit
change p:%%%15554 90909090 32bit

# Display Mfg ID values at offsets 0, 1, 2, 3
# Display Dev ID values at offsets 4, 5, 6, 7
display p:%%%00000..%%%00008 8bit

# Reset devices to Read state
change p:%%%15554 aaaaaaaa 32bit
change p:%%%0aaa8 55555555 32bit
change p:%%%15554 f0f0f0f0 32bit
```

Listing 33. Atmel: One 16-bit Device

```
# Set device to Read state
change p:%%%0aaaa 00aa 16bit
change p:%%%05554 0055 16bit
change p:%%%0aaaa 00f0 16bit

# Get Mfg and Device ID values
change p:%%%0aaaa 00aa 16bit
change p:%%%05554 0055 16bit
change p:%%%0aaaa 0090 16bit
```

```
# Display Mfg ID value at offset 0
# Display Dev ID value at offset 2
display p:%%%00000..%%%00004 16bit

# Reset device to Read state
change p:%%%0aaaa 00aa 16bit
change p:%%%05554 0055 16bit
change p:%%%0aaaa 00f0 16bit
```

Listing 34. Atmel: Two 16-bit Devices

```
# Set devices to Read state
change p:%%%15554 00aa00aa 32bit
change p:%%%0aaa8 00550055 32bit
change p:%%%15554 00f000f0 32bit

# Get Mfg and Device ID values
change p:%%%15554 00aa00aa 32bit
change p:%%%0aaa8 00550055 32bit
change p:%%%15554 00900090 32bit

# Display Mfg ID values at offsets 0, 2
# Display Dev ID values at offsets 4, 6
display p:%%%00000..%%%00008 16bit

# Reset devices to Read state
change p:%%%15554 00aa00aa 32bit
change p:%%%0aaa8 00550055 32bit
change p:%%%15554 00f000f0 32bit
```

Listing 35. Atmel: Four 16-bit Devices

```
# Set devices to Read state
change p:%%%02998 00aa00aa00aa00aa 64bit
change p:%%%01550 0055005500550055 64bit
change p:%%%02aa8 00f000f000f000f0 64bit

# Get Mfg and Device ID values
change p:%%%02998 00aa00aa00aa00aa 64bit
change p:%%%01550 0055005500550055 64bit
change p:%%%02aa8 0090009000900090 64bit

# Display Mfg ID values at offsets 0, 2, 4, 6
# Display Dev ID values at offsets 8, a, c, e
display p:%%%00000..%%%00010 16bit

# Reset devices to Read state
change p:%%%02998 00aa00aa00aa00aa 64bit
change p:%%%01550 0055005500550055 64bit
change p:%%%02aa8 00f000f000f000f0 64bit
```

Listing 36. Intel: One 8-bit Device

```
# Set device to Read state
# and clear status register
change p:%%%00000 ff 8bit
change p:%%%00000 50 8bit

# Get Mfg and Device ID values
change p:%%%00000 90 8bit

# Display Mfg ID value at offset 0
# Display Dev ID value at offset 1
display p:%%%00000..%%%00002 8bit

# Read and display status register
change p:%%%00000 70 8bit
display p:%%%00000..%%%00001 8bit

# Reset device to Read state
change p:%%%00000 ff 8bit
```

Listing 37. Intel: Two 8-bit Devices

```
# Set devices to Read state
# and clear status registers
change p:%%%00000 ffff 16bit
change p:%%%00000 5050 16bit

# Get Mfg and Device ID values
change p:%%%00000 9090 16bit

# Display Mfg ID values at offsets 0, 1
# Display Dev ID values at offsets 2, 3
display p:%%%00000..%%%00004 8bit

# Read and display status registers
change p:%%%00000 7070 16bit
display p:%%%00000..%%%00002 8bit

# Reset devices to Read state
change p:%%%00000 ffff 16bit
```

Listing 38. Intel: Four 8-bit Devices

```
# Set devices to Read state
# and clear status registers
change p:%%%00000 ffffffff 32bit
change p:%%%00000 50505050 32bit

# Get Mfg and Device ID values
change p:%%%00000 90909090 32bit
```

```
# Display Mfg ID values at offsets 0, 1, 2, 3
# Display Dev ID values at offsets 4, 5, 6, 7
display p:%%%00000..%%%00008 8bit

# Read and display status registers
change p:%%%00000 70707070 32bit
display p:%%%00000..%%%00004 8bit

# Reset devices to Read state
change p:%%%00000 ffffffff 32bit
```

Listing 39. Intel: One 16-bit Device

```
# Set device to Read state
# and clear status register
change p:%%%00000 ffff 16bit
change p:%%%00000 5050 16bit

# Get Mfg and Device ID values
change p:%%%00000 9090 16bit

# Display Mfg ID value at offset 0
# Display Dev ID value at offset 2
display p:%%%00000..%%%00004 16bit

# Read and display status register
change p:%%%00000 7070 16bit
display p:%%%00000..%%%00002 16bit

# Reset device to Read state
change p:%%%00000 ffff 16bit
```

Listing 40. Intel: Two 16-bit Devices

```
# Set devices to Read state
# and clear status registers
change p:%%%00000 ffffffff 32bit
change p:%%%00000 50505050 32bit

# Get Mfg and Device ID values
change p:%%%00000 90909090 32bit

# Display Mfg ID values at offsets 0, 2
# Display Dev ID values at offsets 4, 6
display p:%%%00000..%%%00007 16bit

# Read and display status registers
change p:%%%00000 70707070 32bit
display p:%%%00000..%%%00003 16bit

# Reset devices to Read state
change p:%%%00000 ffffffff 32bit
```

Listing 41. Intel: Four 16-bit Devices

```
# Set devices to Read state
# and clear status registers
change p:%%%00000 ffffffffffffffff 64bit
change p:%%%00000 5050505050505050 64bit

# Get Mfg and Device ID values
change p:%%%00000 9090909090909090 64bit

# Display Mfg ID values at offsets 0, 2, 4, 6
# Display Dev ID values at offsets 8, a, c, e
display p:%%%00000..%%%00008 16bit

# Read and display status registers
change p:%%%00000 7070707070707070 64bit
display p:%%%00000..%%%00004 16bit

# Reset devices to Read state
change p:%%%00000 ffffffffffffffff 64bit
```

10.7. Examples

Following are example flash interrogations that are common for some target boards that use both AMD and Intel devices.

10.7.1. One 16-bit AMD Device

Freescale derivatives M5208EVBE, M52277EVB, M5282EVB, and M5235EVB all use AMD 16x1 devices. These devices have Manufacturer ID `0x01`.
The commands have been executed on a M5235EVB board. For AMD, one device with 16-bit access instructs the use of Listing 27.

Perform these steps:

1. Copy this code into text file, `check_flash.tcl`, making the code a script.
2. Substitute the string FFE with all instances of %%%.
3. Use the source command to invoke the script.

Listing 42 shows the resulting code.

NOTE The comment lines in Listing 42 and Listing 43 are for clarification. The CodeWarrior source command discards comment lines, so you will not see such comments in your Debugger Shell view.

Listing 42. Example One Results

```
# Set device to Read state
change p:FFE00000 f0f0 16bit

# Get Mfg and Device ID values
change p:FFE00aaa aaaa 16bit
change p:FFE00554 5555 16bit
change p:FFE00aaa 9090 16bit

# Display Mfg ID value at offset 0
# Display Dev ID value at offset 2
display p:FFE00000..%%%00004 16bit

ffe00000  $0001 $2245 $0000   .. "E ..

# Reset device to Read state
change p:FFE00000 f0f0 16bit
```

The results of display line show the manufacturer ID code 0x1 and the device ID code 0x2245. This confirms basic read/write functionality of the flash devices.

10.7.2. One 16-bit Intel Device

Freescale derivatives M5329EVBE, M5373EVB, M5475EVB, and M5485EVB all use AMD 16x1 devices. These devices have Manufacturer ID *0x89*.

The commands have been executed on a M5329EVBE board. For Intel, one device with 16-bit access instructs the use of Listing 39.

Perform these steps:

1. Copy this code into text file, `check_flash.tcl`, making the code a script.
2. Substitute the string 000 with all instances of %%%.
3. Use the source command to invoke the script.

Listing 43 shows the resulting code.

Listing 43. Example Two Results

```
# Set device to Read state
# and clear status register
change p:00000000 ffff 16bit
change p:00000000 5050 16bit

# Get Mfg and Device ID values
change p:00000000 9090 16bit

# Display Mfg ID value at offset 0
# Display Dev ID value at offset 2
display p:00000000..%%%00004 16bit

  0  $0089 $88C3 $0001   .. .. ..

# Read and display status register
change p:00000000 7070 16bit
display p:00000000..%%%00002 16bit

  0  $0080 $0080   .. ..

# Reset device to Read state
change p:00000000 ffff 16bit
```

The result of the first display line shows the manufacturer ID code 0x0089 and the device ID code 0x88C3. This confirms basic read/write functionality of the flash devices.

10.8. Summary

For most flash devices in use today, programming involves state machine like cycles of multiple writes that must proceed the final write cycle. To diagnose flash programming failures, you must determine whether the cause of the failure is in target hardware or flash programming software. The general method of document lets you make this determination through simple, low level writes and reads, without the use of expensive and complicated logic analyzers. If these reads and writes fail, the problem most likely is on the target. If these reads and writes succeed, the problem most likely is in the flash programming software, so contact Technical Support at http://www.freescale.com/support.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4349

6 May 2013