NXP Semiconductors
Application Note

Document Number: AN5398
Rev. 1, 03/2017

Adding Device(s) to the CodeWarrior Flash Programmer for ARMv8 Processors

Contents

1 Introduction

This document explains how to use the Flash Tool Kit to support additional flash devices on the Flash Programmer for CodeWarrior Development Studio for ARMv8 Processors by creating new programming algorithms and support files. To add support for a new flash device, you need to write a new flash programming algorithm.

This document explains how to:

- Create a flash device XML configuration file
- Create new target task flash device algorithm XML
- Create a flash device parameters XML
- Use an existing flash algorithm
- Create an external flash algorithm

2 Preliminary Background

Before you program or erase any flash device, ensure that the CPU can access the flash device. For example, you might need a different debug setup that requires modification to the debugger configuration file.

Consider the following before you begin:

- Read the flash device ID to verify the correct connection and programmability. The section QSPI algorithm example provides you with instructions.

NOTE

Many manufacturers use the same flash device algorithms, so it is likely that flashes can be programmed using the algorithms included with the CodeWarrior software.

- Check whether the new flash device can be programmed with one of the existing flash programmer algorithms.
- Refer to the section Select Flash Programming Algorithm to determine if the flash device is programmable with an algorithm already included with the CodeWarrior software.
- Follow the steps in section Creating External Flash Algorithm if the flash device cannot be programmed with an existing algorithm.

3 Flash Tool Kit (FTK) Overview

Adding a new flash device support requires few new files, including:

- xml configuration file for the new device, which describes the organization
- xml configuration file for the board, which specifies the flash it must use and, where is the RAM memory located, and.
- flash device algorithm, if none of the existing algorithms are compatible

4 Creating Flash Device using existing algorithm

In its default configuration, the CodeWarrior Flash Programmer for Power Architecture ARMv8 supports many flash devices. The configuration files are located at

```
{CodeWarrior}\CW_ARMv8\Config\flash\devices
```

To add a new device to the CodeWarrior flash programmer, you must add a new file that describes the device.

Generic flash device file format

```
<device-file>
<device>
<content>
  <device_paramteres>
      <device_type>DeviceType</device_type>
      <manufacturerid>MfgID</manufacturerid>
  </device_parameters>
        <name>NameOfFlashDevice</name>
    <sectors>
      <sector count="numberOfSectors" size="sectorSize"/>
    </sectors>
    <organizations>
           <organization depth="Capacity" width="buswidth" count="noOfDevices">
        <id>DeviceID_ForBusWidth</id>
        <algorithms>
         <algorithm>
             <fpinclude href="pathToAlgorithm"/>
         </algorithm>
        </algorithms>
       </organization>
    </organizations>
</content>
</device>
</device-file>
```

To add flash programming support for a new flash device:

- Locate the data sheet for the new device and note the following information about the flash device:
  - Device name
  - Manufacturer ID code
  - Device type
  - Number of sectors
  - Sector size
  - Capacity and bus width
  - Which device is most similar in the device configurations
- Examine the installed devices for the most similar definitions.
- 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.

The format is:

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

4.2 Manufacturer ID Code and Device ID Codes

These Manufacturer ID and Device ID are read from the flash device after a specific sequence of writes to the flash device. Although, the data sheet lists both of the ID's, only the Device ID varies among the flash devices from a given vendor, as 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.

The formats are:

```
<manufacturerid>MfgID</manufacturerid>
<id>DeviceID_ForBusWidth</id>
```

4.3 Device Type

Each device has a certain type (NOR, NAND, SD, QSPI) which can be taken directly from the data sheet.

The format is:

```
<device_type>DeviceType</device_type>
```

4.4 Number of Sectors and Sector Size

The data sheet lists the information on sector 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 1shows an example of converting a 16-bit sector map to an 8-bit map.

The formats are:

```
<sector count="sectorcount" size="sectorsize"/>
```

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

For example, consider S25FS512S. The device has 256 sectors of 0x40000 bytes each.

The configuration file will contain:

```
<sectors>
 <sector count="256" size="0x40000"/>
</sectors>
```

4.5 Options for Organization Name

The information that must be specified here, as an organization name, includes: device size, bus width, and number of devices present on 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 <*organizations*> 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 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 <*organizations*> field summarizes each possible combination of device capacity, bus width, and number of devices used.

The format is:

```
<organizations>
<organization depth=”Capacity” width=”bus_width ”count=”NoOfDevices”>
<id>0x4a54</id>
 <algorithms>
        <algorithm>
        <fpinclude href="path to algorithm xml"/>
        </algorithm>
        </algorithms>
</organization>
...
</organizations>
```

4.6 Find Most Similar Device

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

- From the data sheet for target flash devices, determine whether the bus width is 8- or 16- data bits.
- Read through the files in the configuration folder of the CodeWarrior™ Development Studio for ARMv8 installation and scan for devices from the same manufacturer with similar part names.

  For example, SL29GL01GN is similar to SL29GL01GS.

- 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.

4.7 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. These files can be found at:

```
{CodeWarrior}\CW_ARMv8\Config\flash\algorithms
```

The CodeWarrior Development Studio for ARMv8 has built-in flash programming algorithm support for a number of 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 External Flash Algorithm.

Copy the algorithm that you choose to use and rename it the same as your device configuration file (do the same with the parameters file from the “params” folder). Now, in the new algorithm xml, change the name and the parameters xml path.

For example, let’s consider we’ve created the NewDevice.xml configuration file that is similar to S25FS512S. We will make a copy of the algorithm S25FS512S.xml and rename it NewDevice.xml and a copy of the parameters file S25FS512S_QSPI_64.xml and rename it NewDevice_64.xml. Then, in the algorithm xml, we will modify the following lines as follows:

Old:

```
<name>S25FS512S</name>
<fpinclude href="algorithms/params/S25FS512S_QSPI_64.xml"/>
```

New:

```
<name>NewDevice</name>
<fpinclude href="algorithms/params/NewDevice_64.xml"/>
```

After that, modify your board initialization file from the Target Connections Configuration. Locate the following line:

```
gdb.execute("fl_device --alias qspi --name S25FS512S --address 0x40000000 --waddress
0x10000000 --wsize 0x1FFFF --geometry 8x1 --controller QSPI")
```

Replace “S25FS512S” with the name of your device (in our example, that would be NewDevice).

5 Creating External Flash Algorithm

5.1 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. Refer to QSPI algorithm example for instructions
- Many manufacturers use the same flash-device algorithms, so it is likely that flashes can be programmed using algorithms included with 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 in Creating External Flash Algorithm if the flash device cannot be programmed with an existing algorithm.

5.2 Flash Tool Kit (FTK) Overview

The Flash Took Kit helps you develop flash programming algorithms for the CodeWarrior Flash Programmer. This section provides important information needed before you begin creating a flash programming algorithm.

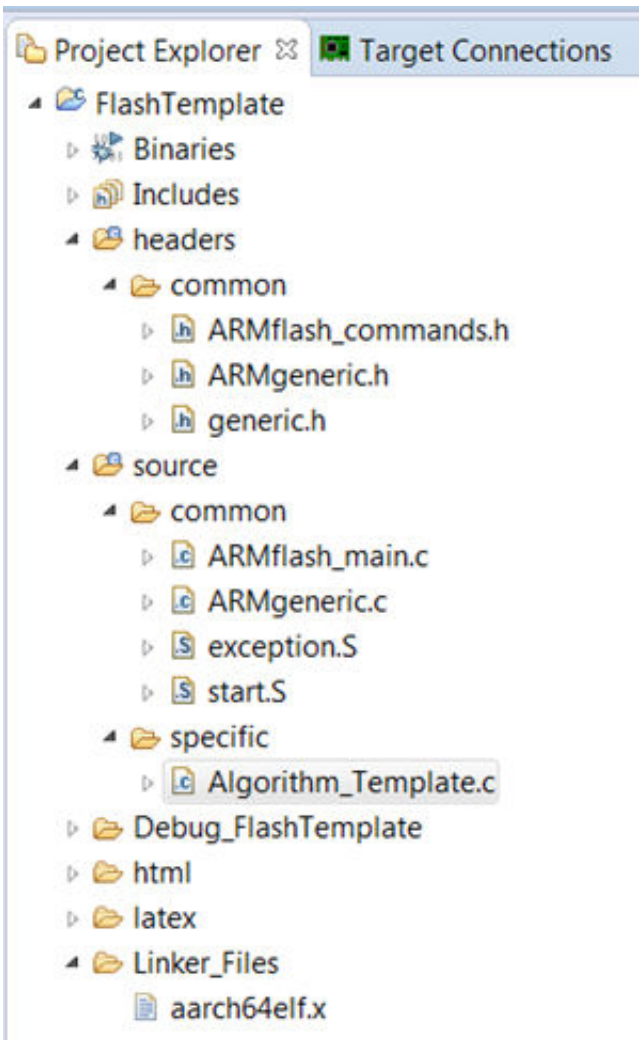


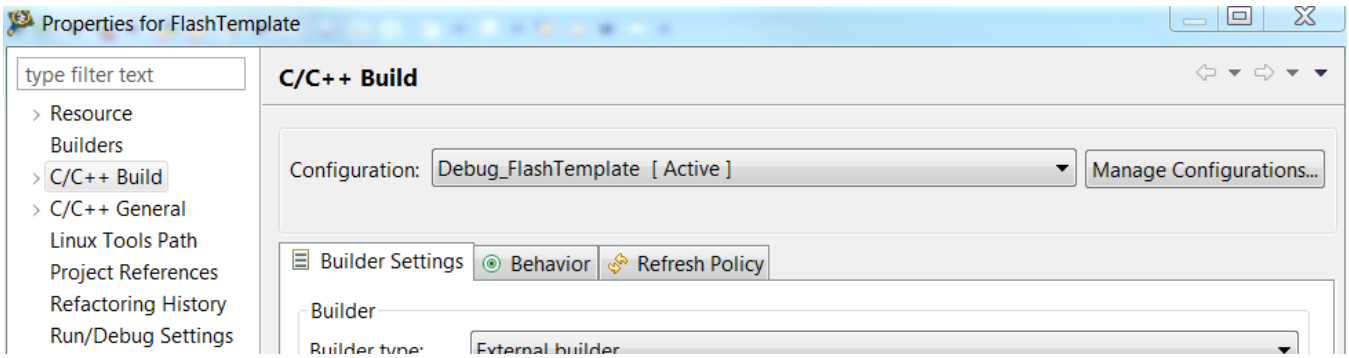
Figure 1. Target Connections

5.3 Flash Tool Kit General Structure

- ARMflash_commands.h – contains the declarations of the flash functions implemented in Algorithm_Template.c
- ARMgeneric.h – contains the declarations of the utility functions implemented in ARMgeneric.c
- generic.h – contains the definition of some macros and the parameter block structure
- ARMflash_main.c – the main function and API to the CodeWarrior Flash Programmer
- ARMgeneric.c – utility functions used for reversing endianness for different data types

- Exception.S and start.S – these files should not be modified
- Algorithm_Template.c – contains the implementation of the Flash Programmer functions.

5.4 Flash Tool Kit Build Target



The Flash Tool Kit Build Target is Debug_FlashTemplate. Building the target will generate a BIN file which will later be used to add the custom algorithm.

5.5 Flash Programmer API

Parameter_block_t Structure

For the detailed description of the Parameter_block_t structure refer to 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;
} parameter_block_t;
```

Listing definitions:

- `function` – command from CodeWarrior Flash Programmer to be executed.
- `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 necessary.
- `items` – start address of the data to be transferred from the CodeWarrior Flash Programmer to the flash programming applet.

NOTE

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

5.5.1 retval_t ID(parameter_block_t* p_pb)

Function called when reading device ID.

Parameters

`parameter_block_t* p_pb`	pointer to a **parameter_block_t** structure

Returns: `retval_t` type is always returned.

5.5.2 retval_t flash_dump (parameter_block_t * p_pb)

Function called when exporting data from flash memory.

Parameters

`parameter_block_t* p_pb`	pointer to a **parameter_block_t** structure

Returns: `retval_t` type is always returned.

5.5.3 retval_t flash_erase (parameter_block_t * p_pb, uint32_t iSector)

Function called when erasing a sector from flash memory.

Parameters

`parameter_block_t* p_pb`	pointer to a **parameter_block_t** structure
`uint32_t iSector`	a 32 bit unsigned integer, representing the sector index

Returns: `retval_t` type is always returned.

5.5.4 retval_t flash_protect (parameter_block_t * p_pb, uint32_t sect_index)

Function called when protecting a sector in flash memory.

Parameters

`parameter_block_t* p_pb`	pointer to a **parameter_block_t** structure
`uint32_t sect_index`	a 32 bit unsigned integer, representing the sector index

Returns: `retval_t` type is always returned.

5.5.5 retval_t flash_unprotect (parameter_block_t * p_pb, uint32_t sect_index)

Function called when unprotecting a sector in flash memory.

Parameters

`parameter_block_t* p_pb`	pointer to a **parameter_block_t** structure
`uint32_t sect_index`	a 32 bit unsigned integer, representing the sector index

Returns: `retval_t` type is always returned.

5.5.6 retval_t flash_write (parameter_block_t * p_pb)

Function called when writing bytes into flash memory.

Parameters

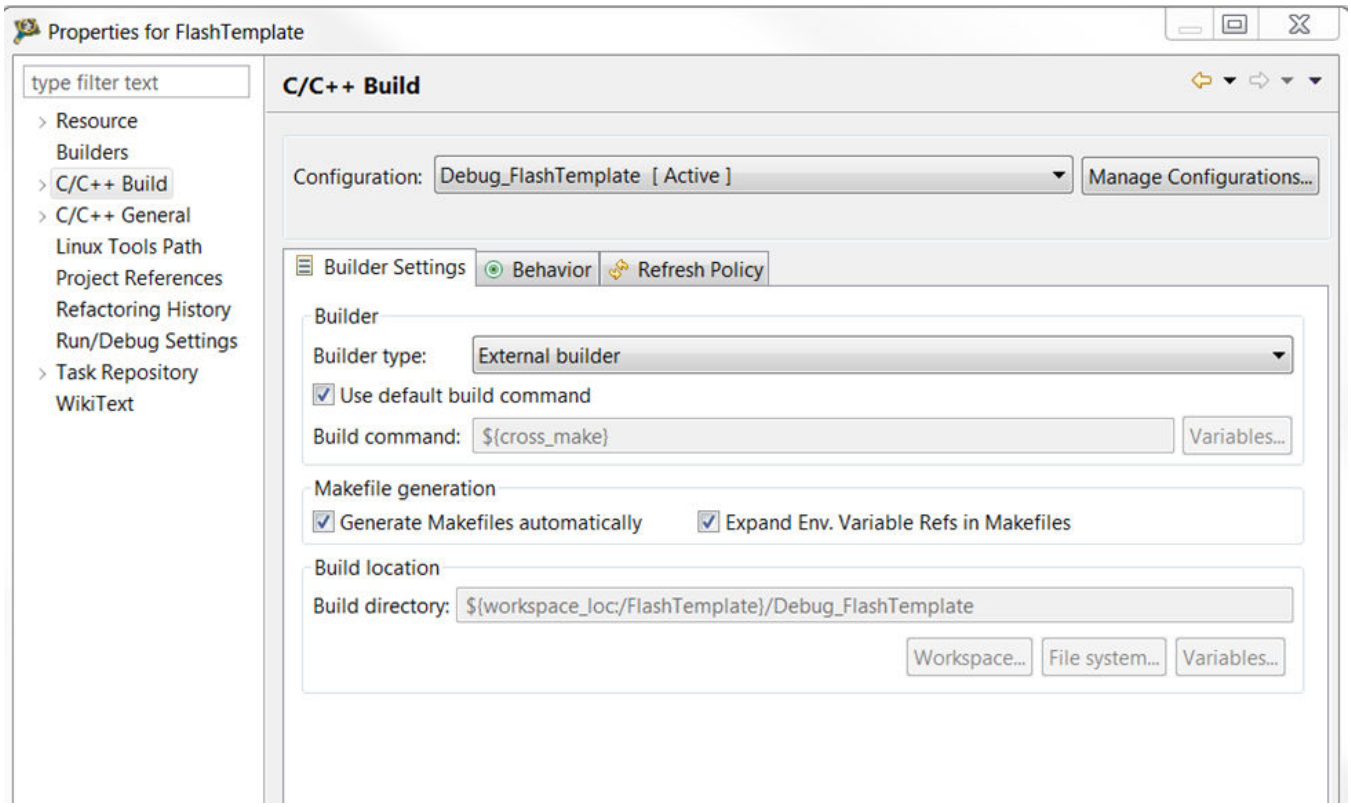
`parameter_block_t* p_pb`	pointer to a **parameter_block_t** structure

Returns: `retval_t` type is always returned.

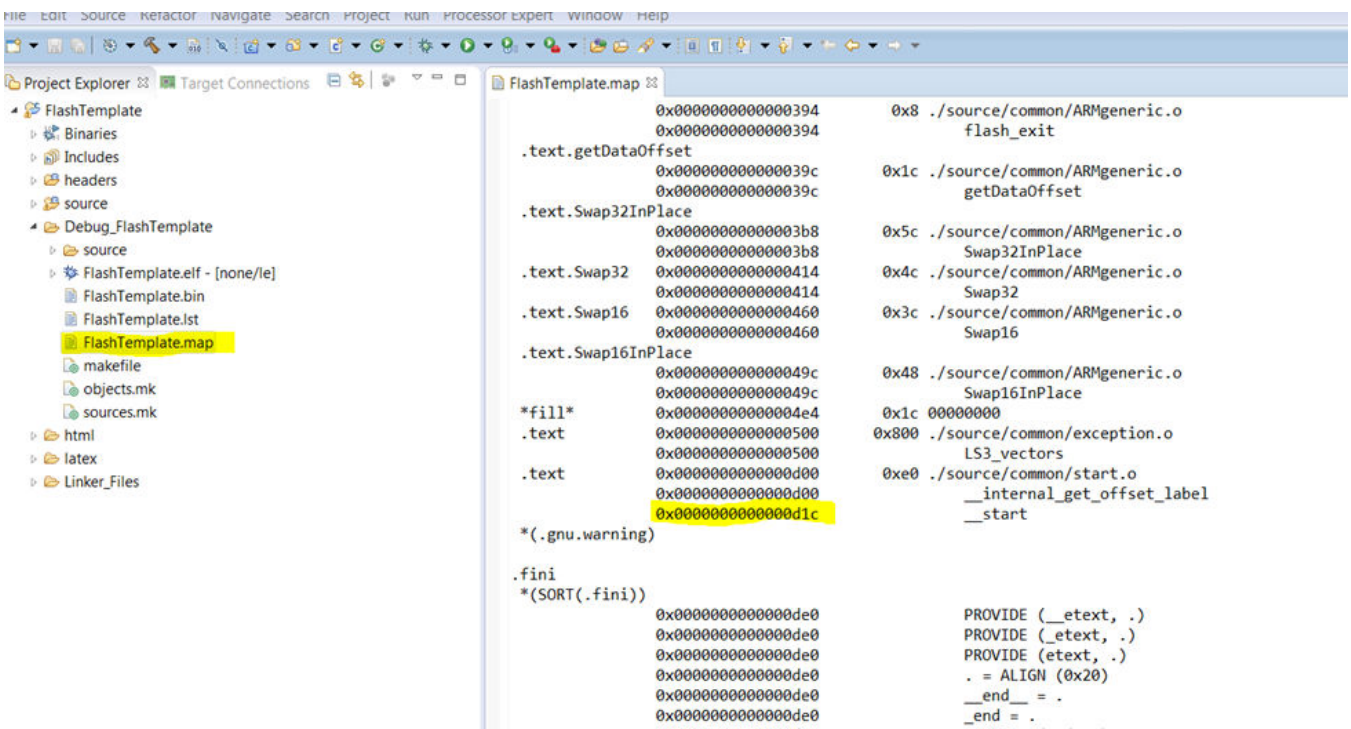
5.6 Create New Flash Programming Algorithm

In this section, step-by-step instructions show you 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.

1. Copy FlashTemplate folder from `{CodeWarrior}\CW_ARMv8\ARMv8\CodeWarrior_Examples` to a different directory from where you will import the project.
2. Import the FlashTemplate project:
   a. 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 FlashTemplate folder.
   d. The Projects list will show all available projects in the FlashTemplate folder. If you see projects other than FlashTemplate, then deselect all other projects and click Finish.
   e. Ensure Debug_FlashTemplate build target is selected (Project -> Properties -> C/C++ Build).



3. Implement the algorithms. (Note: for the purpose of this example, we will leave the algorithms blank).
4. Build the project (Project -> Build Project).
5. In the Debug_FlashTemplate directory, open the FlashTemplate.map file and scroll down to the __start address. Write it down as it will be used later when creating the xml files.



6. Also, copy the FlashTemplate.bin file into {CodeWarrior}\CW_ARMv8\Config\flash\arm\bin folder.
7. Create the parameters xml in `{CodeWarrior}\CW_ARMv8\Config\flash\algorithms\params`

```
<params_file>
    <parameters_block>
        <param name="function" size="0x4"/>
        <param name="padding1" size="0x4"/>
        <param name="base_addr" size="0x8"/>
        <param name="num_items" size="0x4" type="data_size"/>
        <param name="result_status" size="0x4" type="result"/>
        <param name="items" size="0x8" type="data_inout"/>
    </parameters_block>
</params_file>
```

This is the template for the most basic parameter file. Here is where you find the supported functions and where you can add new parameters used for the parameter block structure.

8. Create the algorithm xml in {CodeWarrior}\CW_ARMv8\Config\flash\algorithms.

```
<algorithm-file>
    <architectures>
        <architecture type="arm" address_size="64">
            <controller type="QSPI">
                <format>bin</format>
                <entry_point>0x0d9c</entry_point>
                <file>FlashTemplate</file>
                <fpinclude href="algorithms/params/FlashTemplate_64.xml"/>
                <supported_operations>
                    <operation>id</operation>
                    <operation>erase_sectors</operation>
                    <operation>program</operation>
                    <operation>dump</operation>
                    <operation>protect_sectors</operation>
                    <operation>unprotect_sectors</operation>
                </supported_operations>
            </controller>
        </architecture>
    </architectures>
</algorithm-file>
```

The architecture type and address_size are taken from the CPU. Controller type mush match the one specified in the console command. Format is always bin. Entry point is the address you've written down at step 5). The last is the path to the parameters xml.

9. Create the device configuration xml in {CodeWarrior}\CW_ARMv8\Config\flash\devices.

```
<device-file>
    <device>
        <content>
            <device_parameters>
                <device_type>spi</device_type>
                <manufacturerid>0x01</manufacturerid>
            </device_parameters>
            <sectors>
                <sector count="256" size="0x40000"/>
            </sectors>
            <organizations>
                <organization depth="64M" width="8">
                    <id>0x0102</id>
                    <algorithms>
                        <algorithm>
                            <fpinclude href="algorithms/FlashTemplate.xml"/>
                        </algorithm>
                    </algorithms>
                </organization>
            </organizations>
        </content>
    </device>
</device-file>
```

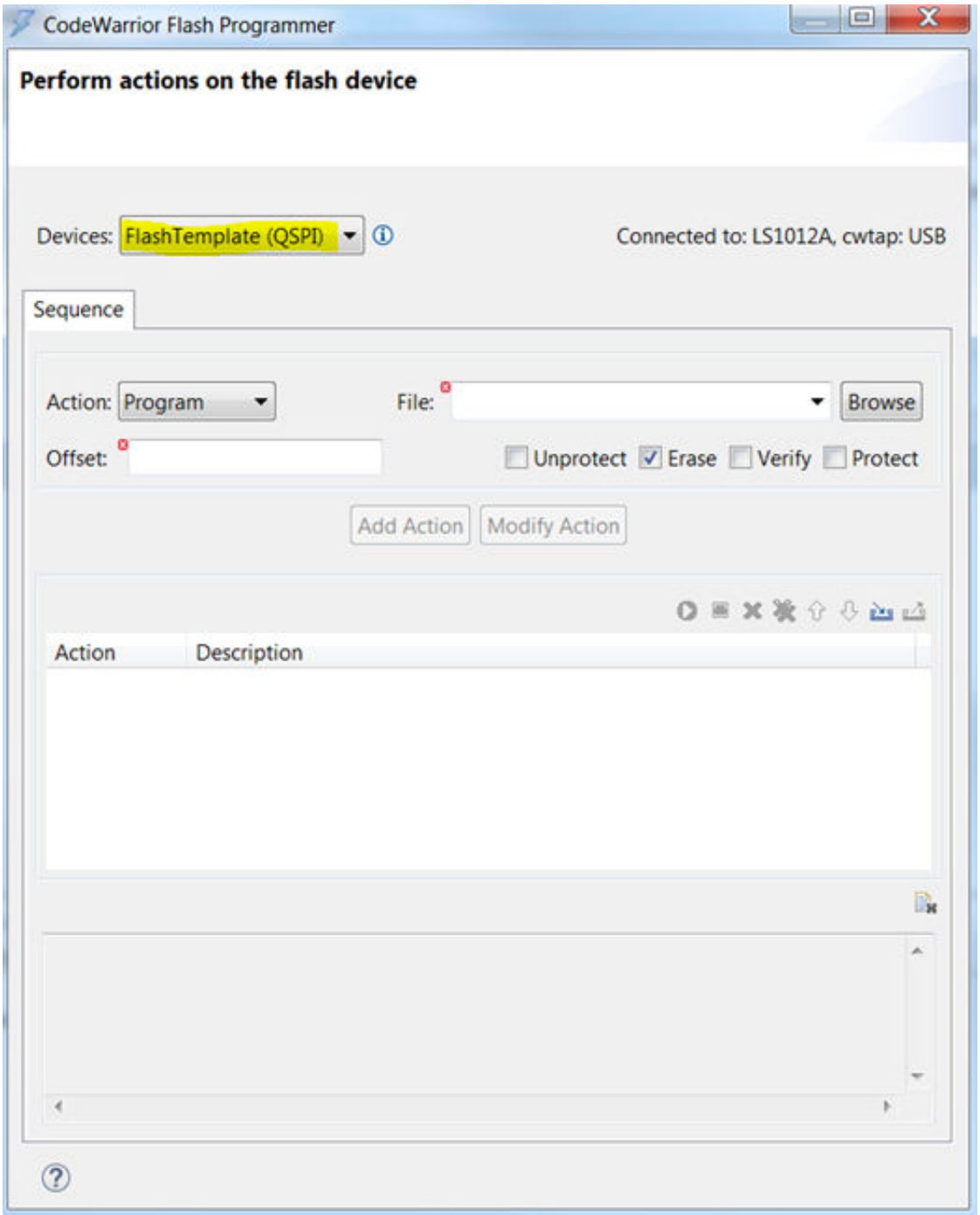
The device configuration xml has been explained in Creating Flash Device using existing algorithm.

10. In the CodeWarrior Target Connections Configuration tab, modify the target initialization file for the board you are using, as follows:

```
gdb.execute("fl_device --alias qspi --name FlashTemplate
```

Insert the name of your device configuration xml.

11. Now, in the Flash Programmer, you should be able to see your new flash device.



6 QSPI algorithm example

An overview of the QSPI algorithm, the supported commands and the parameters XML file.

6.1 Algorithm overview

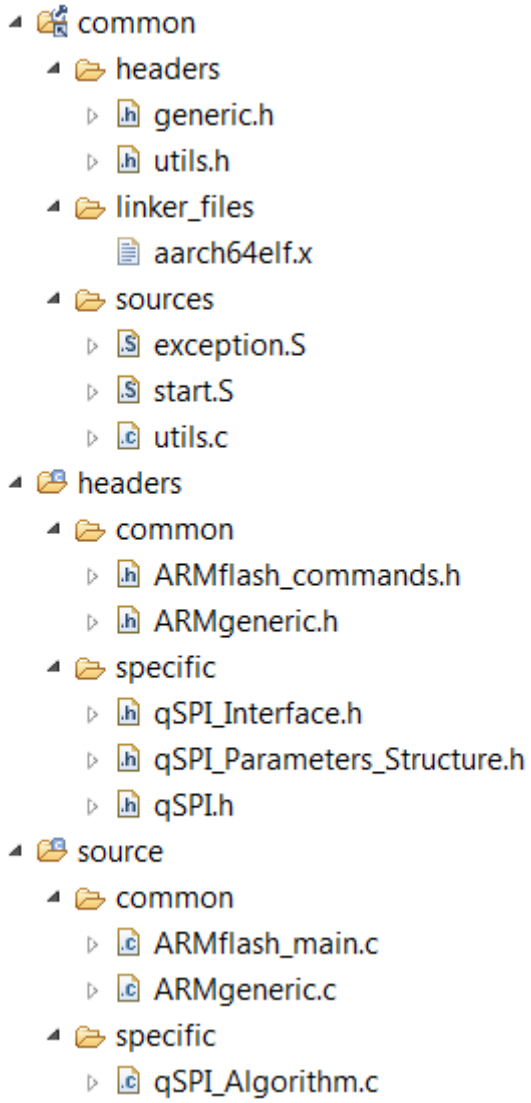


Figure 2. QSPI algorithm

- generic.h – header containing different types of operations, errors and a somewhat generic pointer type.
- utils.h – utility functions for treating data endianness, getting the offset to the parameter structure, reading/writing registers.
- aarch64elf.x – linker file.
- exception.S – exception handler code.
- start.S – calculates the entry point and other configurations.
- utils.c – implementation of the functions found in utils.h.
- ARMflash_commands.h – functions used for programming the flash (id, dump, write, erase, protect, unprotect).
- ARMgeneric.h – defines some parameters used for determining the soc configuration.
- qSPI_Interface.h – QSPI register layout.
- qSPI_Parameters_Structure.h – structure containing the parameter block.
    - *function* – the function to perform.
    - *padding1* – needed for aligning the structure to a 64-bit boundary.
    - *base_addr* - start of the flash memory where we are going to operate.
    - *num_items* – number of items in the buffer.

- *result_status* – operation status.
- *items* – pointer to a data buffer.
- *qspi_base_addr* – base address of the qspi region in the SoC memory map.
- *qspi_controller_offset* – offset to the qspi controller in memory.
- *block_protect_mask* – a mask used for determining which bits of the status register are used for block protection.
- *bytes_per_sector* – number of bytes which comprise a sector.
- *bytes_per_page* – the number of bytes of a page.
- *number_of_sectors* – how many sectors are there in the flash.
- *swap_enable* – endianness flag.
- workaround – parameter used for applying device specific configurations (like uniform sector enable).
- is_nand – flag used for identifying whether the device is QSPI Nor or QSPI Nand.
- *write_enable_cmd* – write enable command code.
- *read_id_cmd* – read id command code.
- *read_id_dummy* – read id dummy cycles.
- *read_id_length* – read id command read length.
- *read_status_register_cmd* – read status register command code.
- *read_status_register_address_length* – read status register command address length.
- *read_status_register_length* – read status register command read length.
- *write_status_register_cmd* – write status register command code.
- *write_status_register_address_length* – write status register command address length.
- *write_status_register_length* – write status register command read length.
- *read_any_register _cmd* – read any register command code.
- *read_any_register_address_length* – read any register command address length.
- *read_any_register_dummy* – read any register command dummy cycles.
- *read_any_register_length* – read any register command length.
- *write_any_register_cmd* – write any register command code.
- *write_any_register_address*_length – read any register command address length.
- *write_any_register_dummy* – read any register command dummy cycles.
- *write_any_register_length* – read any register command length.
- *page_read_cmd* – page read command code.
- *page_read_dummy1* – page read first dummy cycle.
- *page_read_address_length* – page read command address length.
- *page_read_dummy2* – page read second dummy cycle.
- *page_read_from_cache_cmd* – page read from cache command code.
- *page_read_from_cache_dummy1* – page read from cache first dummy cycle.
- *page_read_from_cache_address_length* – page read from cache address length.
- *page_read_from_cache_dummy2* – page read from cache second dummy cycle.
- *page_program_cmd* – page program command code.
- *page_program_dummy* – page program dummy cycle.
- *page_program_address_length* – page program command address length.
- *page_erase_cmd* – page erase command code.
- *page_erase_dummy* – page erase dummy cycles.
- *page_erase_address_length* – page erase command address length.
- *page_execute_cmd* – page execute command code.
- *page_execute_dummy* – page execute dummy cycles.
- *page_execute_address_length* – page execute address length.
- *die_select* – die select command code.
- *die_index* – die index.
- *padding* – used to align the parameter structure.

- qSPI.h – defines some command ids, lookup table instructions and declares some general qspi functions.
- ARMflash_main.c – the main function and logic loop.
- ARMgeneric.c – autodetect SoC function definition.
- qSPI_Algorithm.c – definitions of all the flash commands and qspi utility commands.

6.2 Functions

fID:

The id command is called by CodeWarrior Flash Programmer to read the Manufacturer and Device ID of the chip. For the fID command, the CodeWarrior Flash Programmer:

- loads the flash programming applet to the target board,
- sets the command fID, as shown in the function field,
- 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.

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,
- specifies number of blocks to be erased, as shown in the num_items field,
- specifies start-up address of each block to be erased, as shown in the items field,
- 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.

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, CodeWarrior Flash Programmer:

- loads the flash programming applet to the target board,
- sets the command fWrite, as shown in the function field ,
- specifies number of bytes to be programmed, as shown in the num_items field,
- specifies start-up address of data to be programmed, as shown in the items field,
- 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.

fDumpFlash:

The dump flash command is called by CodeWarrior Flash Programmer to read the values at a specific address and output it. For the fDumpFlash command, the CodeWarrior Flash Programmer:

- loads the flash programming applet to the target board,
- sets the command fDumpFlash, as shown in the function field,
- specifies number of bytes to be read, as shown in the num_items field,
- specifies start-up address of data to be read, as shown in the num_items field,
- 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.

fProtectSector:

The protect sector command is called by CodeWarrior Flash Programmer to protect a number of sectors in the flash. For the fProtectSector command, the CodeWarrior Flash Programmer:

- loads the flash programming applet to the target board,

- sets the command fProtectSector, as shown in the function field,
- specifies start address to be protected,
- specifies number of bytes to be protected,
- 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.

fUnprotectSector:

The unprotect sector command is called by CodeWarrior Flash Programmer to protect a number of sectors in the flash. For the fUnprotectSector command, the CodeWarrior Flash Programmer:

- loads the flash programming applet to the target board,
- sets the command fUnprotectSector, as shown in the function field,
- specifies start address to be unprotected,
- specifies number of bytes to be unprotected,
- 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.

6.3 Building the algorithm (only if the sources were modified)

In order to build the QSPI algorithm, perform the following steps:

1. Import the QSPI algorithm sources:
   a. Click **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** and navigate to the QSPI algorithm sources located at:

      `{CodeWarrior}\CW_ARMv8\ARMv8\CodeWarrior_Examples\FlashSDK\`

   d. The Projects list will show all available projects in the folder. Select the `QSPI_algorithm` project.
2. Right-click the project and select **Build Project**.
3. Copy the resulted binary in the `{CodeWarrior}\CW_ARMv8\Config\flash\arm\bin` folder.

6.4 Parameters XML

The parameter XML located in the `{CodeWarrior}\CW_ARMv8\Config\flash\algorithms\params` folder has the following structure.

QSPI algorithm example

```
<params_file>
    <parameters_block>
        <param name="function" size="0x4"/>
        <param name="padding1" size="0x4"/>
        <param name="base_addr" size="0x8"/>
        <param name="num_items" size="0x4" type="data_size"/>
        <param name="result_status" size="0x4" type="result"/>
        <param name="items" size="0x8" type="data_inout"/>
        <param name="qspi_base_addr" size="0x8"/>
        <param name="qspi_controller_offset" size="0x4"/>
        <param name="bytes_per_sector" size="0x4" value="0x40000"/>
        <param name="bytes_per_page" size="0x4" value="0x200"/>
        <param name="number_of_sectors" size="0x4" value="0x100"/>
        <param name="swap_enable" size="0x1"/>
        <param name="workaround" size="0x1" value="1"/>
        <param name="is_nand" size="0x1" value ="0"/>
        <param name="block_protect_mask" size="0x1" value="0x1C"/>

        <!-- write enable -->
        <param name="write_enable_cmd" size="0x1" value="0x06"/>

        <!-- read id -->
        <param name="read_id_cmd" size="0x1" value="0x9F"/>
        <param name="read_id_dummy" size="0x1" value="0x0"/>
        <param name="read_id_length" size="0x1" value="0x08"/>

        <!-- read status register -->
        <param name="read_status_register_cmd" size="0x1" value="0x05"/>
        <param name="read_status_register_address_length" size="0x1" value="0x0"/>
        <param name="read_status_register_length" size="0x1" value="0x01"/>

        <!-- write status register -->
        <param name="write_status_register_cmd" size="0x1" value="0x01"/>
        <param name="write_status_register_address_length" size="0x1" value="0x0"/>
        <param name="write_status_register_length" size="0x1" value="0x01"/>

         <!-- read any register -->
        <param name="read_any_register_cmd" size="0x1" value="0x65"/>
        <param name="read_any_register_address_length" size="0x1" value="0x18"/>
        <param name="read_any_register_dummy" size="0x1" value="0x8"/>
        <param name="read_any_register_length" size="0x1" value="0x1"/>

         <!-- write any register -->
        <param name="write_any_register_cmd" size="0x1" value="0x71"/>
        <param name="write_any_register_address_length" size="0x1" value="0x18"/>
        <param name="write_any_register_dummy" size="0x1" value="0x0"/>
        <param name="write_any_register_length" size="0x1" value="0x1"/>

        <!-- page read (NOR) / read to cache (NAND) -->
        <param name="page_read_cmd" size="0x1" value="0x0C"/>
        <param name="page_read_dummy1" size="0x1" value="0x0"/>
        <param name="page_read_address_length" size="0x1" value="0x20"/>
        <param name="page_read_dummy2" size="0x1" value="0x08"/>

        <!-- read from cache (NAND) -->
        <param name="page_read_from_cache_cmd" size="0x1" value="0x13"/>
        <param name="page_read_from_cache_dummy1" size="0x1" value="0x8"/>
        <param name="page_read_from_cache_address_length" size="0x1" value="0x10"/>
        <param name="page_read_from_cache_dummy2" size="0x1" value="0x8"/>
```

```
        <!-- page program -->
        <param name="page_program_cmd" size="0x1" value="0x12"/>
        <param name="page_program_dummy" size="0x1" value="0x0"/>
        <param name="page_program_address_length" size="0x1" value="0x20"/>

        <!-- page erase -->
        <param name="page_erase_cmd" size="0x1" value="0xDC"/>
        <param name="page_erase_dummy" size="0x1" value="0x0"/>
        <param name="page_erase_address_length" size="0x1" value="0x20"/>

        <!-- program execute -->
        <param name="page_execute_cmd" size="0x1" value="0x0"/>
        <param name="page_execute_dummy" size="0x1" value="0x0"/>
        <param name="page_execute_address_length" size="0x1" value="0x0"/>

        <!-- die select -->
        <param name="die_select" size="0x1" value="0x0"/>
        <param name="die_index" size="0x1" value="0x0"/>

        <param name="padding" size="0x3"/>
    </parameters_block>
</params file>
```

Figure 3. Parameters XML

Each line must contain the following:

- name – must match the name of its corresponding field in the parameters structure.
- size – how many bytes it occupies, must match the size of the data type of the parameter in the parameter structure.
- type (optional; default is “in”) – input/output/data_size.
- value (optional) – the value stored in the parameter. Must be set in the xml according to the flash device manual. A value of 0 means that the operation is not supported.

NOTE

Additional padding might be needed for the alignment of the structure.

The parameter structure tries to accommodate most of the QSPI flash devices, but there might be certain cases where additional parameters might be needed. This means the algorithm has to be modified and rebuilt. Also, if the modified parameters structure exceeds the size of 0x100, it’s entry point must be modified in both the linker file and the algorithm XML.

6.5 Algorithm XML

The algorithm XML located in the `{CodeWarrior}\CW_ARMv8\Config\flash\algorithms\` folder has the following structure.

```
<algorithm-file>
    <architectures>
        <architecture type="arm" address_size="64">
            <controller type="QSPI">
                <format>bin</format>
                <entry_point>0x100</entry_point>
                <file>QSPI_64b</file>
                <fpinclude href="algorithms/params/S25FS512S_QSPI_64.xml"/>
                <supported_operations>
                    <operation>id</operation>
                    <operation>erase_sectors</operation>
                    <operation>program</operation>
                    <operation>dump</operation>
                    <operation>protect_sectors</operation>
                    <operation>unprotect_sectors</operation>
                </supported_operations>
            </controller>
        </architecture>
    </architectures>
</algorithm-file>
```

Figure 4. Algorithm XML

- *architecture*
  - *type* – the CPU type.
  - *address_size* – the size of an address in bits.
- *controller type* – the type of the flash device.
- *format* – extension of the algorithm binary. It is “bin” for every device.
- *entry point* – the entry point of the algorithm. It’s 0x100 in most cases, but if your parameters structure exceeds 0x100 in size (the sum of the sizes is bigger than 0x100), you will not be able to build the algorithm and will have to adjust the entry point accordingly. You can check it in the map file located in the same folder as your binary.
- *file* – the name of the algorithm binary.
- *fpinclude href* – path to the parameter xml.
- *supported operations* – the operations that the flash supports.

6.6 Device XML

The device XML in the `{CodeWarrior}\ CW_ARMv8\Config\flash\devices` folder is shown in the following figure.

```
<device-file>
    <device>
        <content>
            <device_parameters>
                <device_type>spi</device_type>
                <manufacturerid>0x01</manufacturerid>
            </device_parameters>
            <name>S25FS512S</name>
            <sectors>
                <sector count="256" size="0x40000"/>
            </sectors>
            <organizations>
                <organization depth="64M" width="8">
                    <id>0x0220</id>
                    <algorithm>
                        <fpinclude href="algorithms/S25FS512S.xml"/>
                    </algorithm>
                </organization>
            </organizations>
        </content>
    </device>
</device-file>
```

Figure 5. Device XML

- device_type – the type of the device.
- manufacturerid – this is the ID of the manufacturer. It is unique for all the flash devices of a certain producer.
- name – the device name.
- sector count – number of sectors, size – the size of a sector, expressed in bytes (hexadecimal).
- depth – flash size (sector count * sector size), width – the bus width of the device.
- id – device id, found in the same table as the manufacturer id.

7 Debugging the flash programmer QSPI algorithm

If the flash programmer QSPI algorithm does not behave the way it was expected to, debugging the algorithm might be necessary. This section will cover how to enter debug mode for the flash programmer algorithm, as well as the most common places to check for mistakes.

7.1 Entering debug mode

To debug the algorithm:

1. Configure the target connection to the device on which the algorithm will be tested (For details about how to configure the connection, see *ARM V8 ISA, Targeting Manual*).
2. Go to the linker file located in the algorithm (`common/linker_files/aarch64elf.x`) and modify the `__MEMORY_START` value to the `ws_address` used by the device (found in the config file located at `{CodeWarrior}\CW_ARMv8\Config\boards`).

```
def Config_Flash_Devices():
    # Add QSPI device
    gdb.execute("fl_device --alias qspi --name S25FS512S --address 0x40000000 --waddress 0x10000000
```

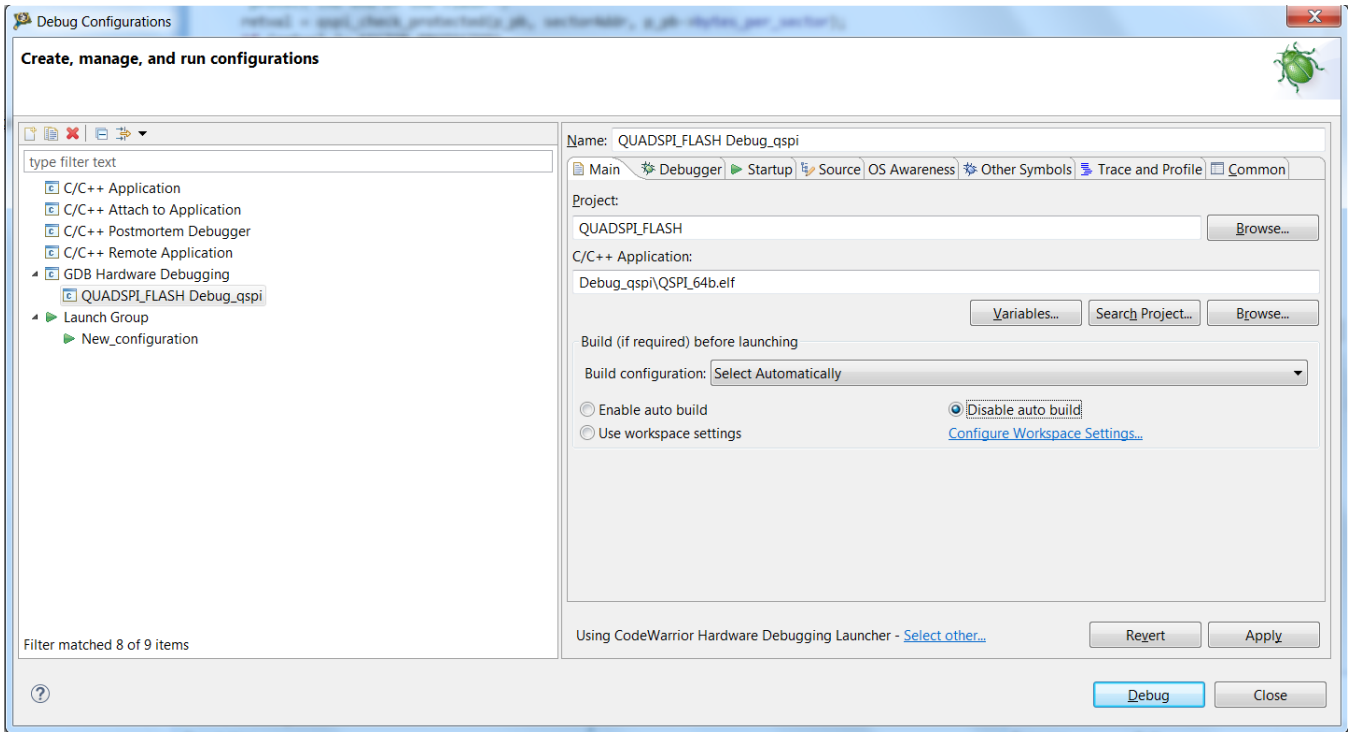
```
OUTPUT_ARCH(aarch64)
ENTRY(__start)

SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__MEMORY_START = 0x10000000);
  . = SEGMENT_START("text-segment", __MEMORY_START);

  .header         : { *(.header) }
  .header         : { *(.header_utils) }
  .start          : { *(.start) }
```

3. Go to the code area you wish to debug and place an assembler halt command (`asm ("hlt 1")`) (this will act as a breakpoint in the algorithm).

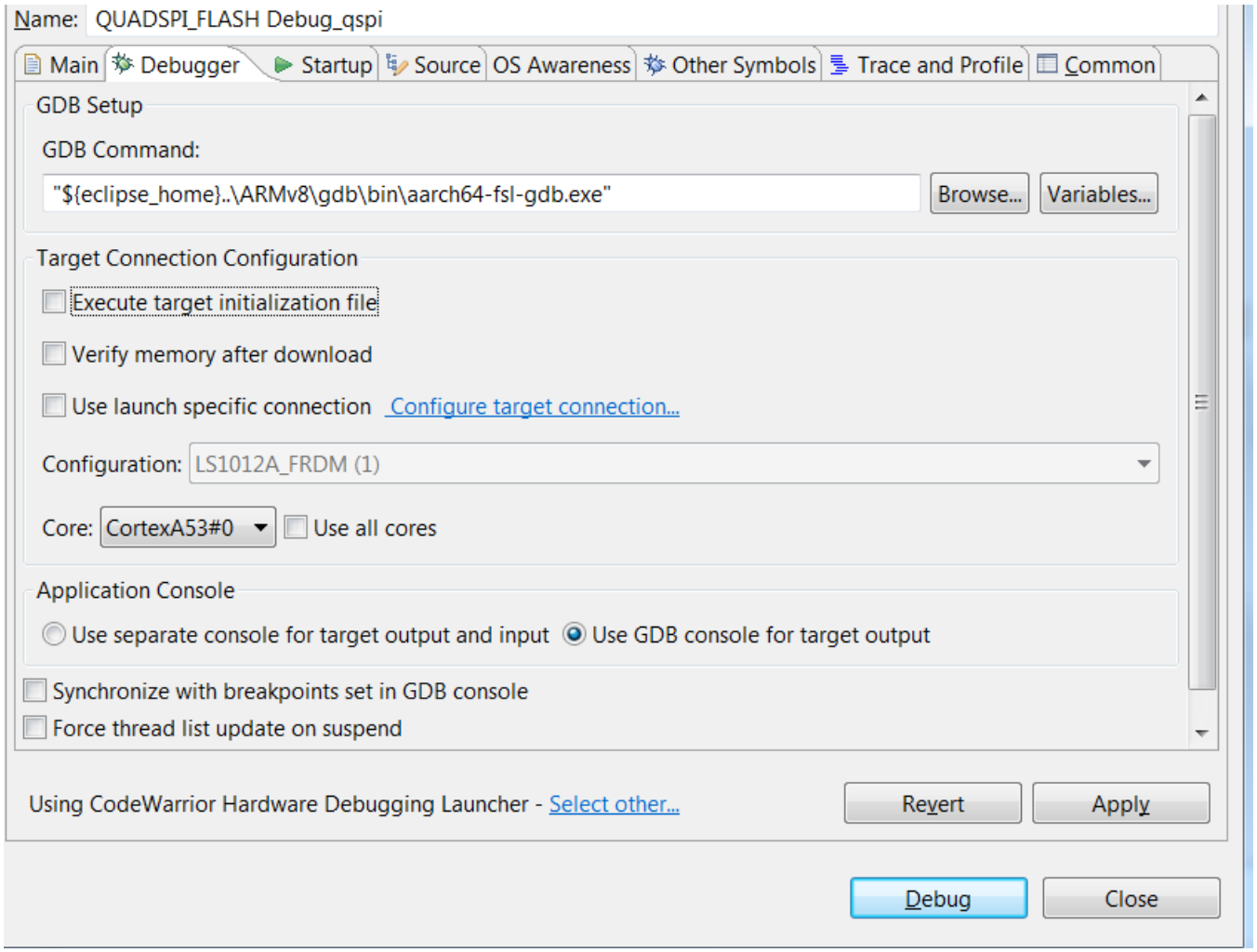
```
retval_t
read_id(parameter_block_t* p_pb)
{
    volatile uint32_t auxVal = 0;

    /* get a pointer to the parameter structure data buffer */
    volatile uint8_t* buffer = p_pb->items.c;

    /* get a pointer to the QSPI controller address */
    QuadSPI_Type *QuadSPI = (QuadSPI_Type*)(intptr_t)p_pb->qspi_controller_offset;
    asm("hlt 1");
    /* initialize the lookup table */
    retval_t rc = qspi_init(p_pb);
    if (rc)
        return rc;
```

Above is an example that wants to debug the id command.

4. Right-click the project and select **Build Project**.
5. Copy the created binary in the `{CodeWarrior}\ CW_ARMv8\Config\flash\arm\bin` folder.
6. Connect to the board.
7. Issue a command that will trigger the breakpoint (in the example above, it would be “fl_id”).
8. Disconnect.
9. Enter debugging mode:
    a. Create a new Debug Configuration (**Run -> Debug Configurations**).
    b. Create a new GDB Hardware Debugging configuration.
    c. Make sure the project and elf file are correct and disable auto build.

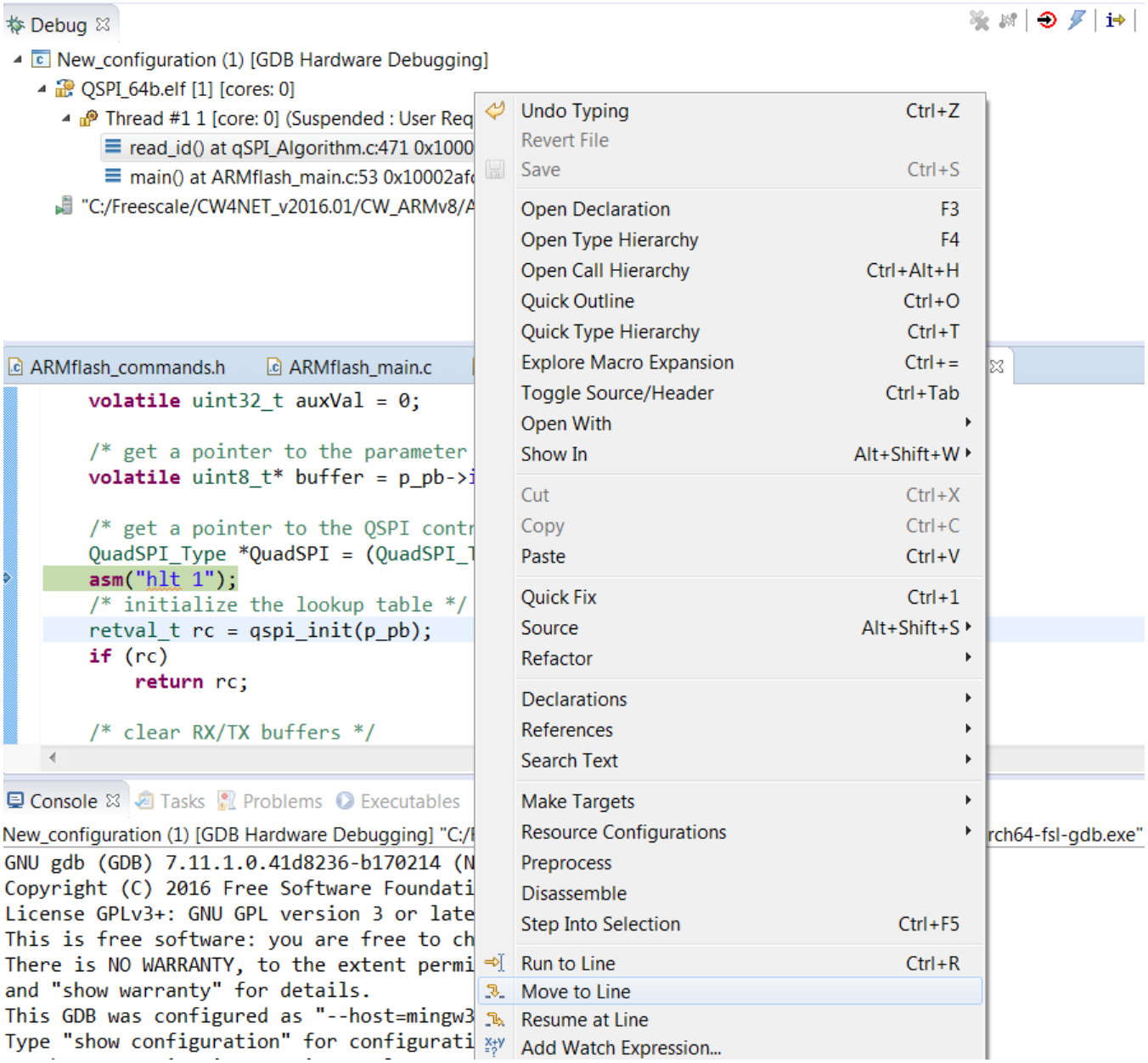


d. In the **Debugger** tab, uncheck the **Execute target initialization file** checkbox.



e. In the **Startup** tab, uncheck **Reset and Delay** and **Load image**, and make sure **Load symbols** is checked.

10. Your PC will point to the asm("`hlt 1`") instruction. To continue debugging, right-click the next line and select Move to Line.



11. Now, you can step through the code, inspect registers, view memory.

7.2 Debugging process

Debugging the application might prove to be a bit challenging. The most common places to check for errors are the following:

1. The parameter structure: most frequent errors are because of incorrect parameter values. Most frequently, there are two reasons for this:
   - Incorrect value provided in the XML
   - Incorrect padding inserted

The first step is to go to the “Variables” tab and check the values in “p_pb”. If they are not correct, first check if the values in the XML are correct. If they are correct in the XML but not in the parameter structure, it means there is a padding problem. Check the addresses in the “Expressions” tab to see if there are any incorrect addresses (current type size + current address =/= next address). If there are such cases, it means there should be extra padding there (in both the XML and the algorithm parameter structure).

Expression	Type	Value	Address
p_gParams	parameter_block_t *	0x10003ef0	0x10003ec0
function	uint32_t	268451600	0x10003ef0
padding1	uint32_t	0	0x10003ef4
base_addr	pointer_t	{...}	0x10003ef8
num_items	uint32_t	268435456	0x10003f00
result_status	retval_t	0	0x10003f04
items	pointer_t	{...}	0x10003f08
qspi_base_addr	pointer_t	{...}	0x10003f10
qspi_controller_offset	uint32_t	0	0x10003f18
bytes_per_sector	uint32_t	0	0x10003f1c
bytes_per_page	uint32_t	0	0x10003f20
number_of_sectors	uint32_t	6	0x10003f24
swap_enable	uint8_t	36 '$'	0x10003f28
workaround	uint8_t	0 '\0'	0x10003f29
is_nand	uint8_t	0 '\0'	0x10003f2a
block_protect_mask	uint8_t	0 '\0'	0x10003f2b
write_enable_cmd	uint8_t	12 '\f'	0x10003f2c
read_id_cmd	uint8_t	0 '\0'	0x10003f2d
read_id_dummy	uint8_t	0 '\0'	0x10003f2e
read_id_length	uint8_t	0 '\0'	0x10003f2f
read_status_register_cn	uint8_t	36 '$'	0x10003f30
read_status_register_ac	uint8_t	0 '\0'	0x10003f31
read_status_register_le	uint8_t	0 '\0'	0x10003f32
write_status_register_cr	uint8_t	0 '\0'	0x10003f33
write_status_register_a	uint8_t	36 '$'	0x10003f34
write_status_register_le	uint8_t	0 '\0'	0x10003f35

2. The QuadSPI LUT registers were not written correctly. This can be a consequence of incorrect values stored in the parameter structure, or it could be because some commands need extra parameters that are not yet covered by our structure and XMLs. The LUT registers can be checked in Peripherals -> Device Registers -> QuadSPI.

QuadSPI_LUT0	0x03041808	0x0	RW	0x1550310	Look-up Table register
QuadSPI_LUT1	0x081c0024	0x0	RW	0x1550314	Look-up Table register
QuadSPI_LUT2	0x00000000	0x0	RW	0x1550318	Look-up Table register
QuadSPI_LUT3	0x00000000	0x0	RW	0x155031c	Look-up Table register
QuadSPI_LUT4	0x9f04081c	0x0	RW	0x1550320	Look-up Table register
QuadSPI_LUT5	0x00000000	0x0	RW	0x1550324	Look-up Table register
QuadSPI_LUT6	0x00000000	0x0	RW	0x1550328	Look-up Table register
QuadSPI_LUT7	0x00000000	0x0	RW	0x155032c	Look-up Table register
QuadSPI_LUT8	0x0c042008	0x0	RW	0x1550330	Look-up Table register
QuadSPI_LUT9	0x080c801c	0x0	RW	0x1550334	Look-up Table register
QuadSPI_LUT10	0x00000000	0x0	RW	0x1550338	Look-up Table register
QuadSPI_LUT11	0x00000000	0x0	RW	0x155033c	Look-up Table register
QuadSPI_LUT12	0x12042008	0x0	RW	0x1550340	Look-up Table register
QuadSPI_LUT13	0x40200000	0x0	RW	0x1550344	Look-up Table register
QuadSPI_LUT14	0x00000000	0x0	RW	0x1550348	Look-up Table register
QuadSPI_LUT15	0x00000000	0x0	RW	0x155034c	Look-up Table register
QuadSPI_LUT16	0xdc042008	0x0	RW	0x1550350	Look-up Table register
QuadSPI_LUT17	0x00000000	0x0	RW	0x1550354	Look-up Table register
QuadSPI_LUT18	0x00000000	0x0	RW	0x1550358	Look-up Table register
QuadSPI_LUT19	0x00000000	0x0	RW	0x155035c	Look-up Table register
QuadSPI_LUT20	0x00000000	0x0	RW	0x1550360	Look-up Table register
QuadSPI_LUT21	0x00000000	0x0	RW	0x1550364	Look-up Table register
QuadSPI_LUT22	0x00000000	0x0	RW	0x1550368	Look-up Table register
QuadSPI_LUT23	0x00000000	0x0	RW	0x155036c	Look-up Table register
QuadSPI_LUT24	0x06040000	0x0	RW	0x1550370	Look-up Table register
QuadSPI_LUT25	0x00000000	0x0	RW	0x1550374	Look-up Table register
QuadSPI_LUT26	0x00000000	0x0	RW	0x1550378	Look-up Table register
QuadSPI_LUT27	0x00000000	0x0	RW	0x155037c	Look-up Table register
QuadSPI_LUT28	0x0504011c	0x0	RW	0x1550380	Look-up Table register
QuadSPI_LUT29	0x00000000	0x0	RW	0x1550384	Look-up Table register
QuadSPI_LUT30	0x00000000	0x0	RW	0x1550388	Look-up Table register
QuadSPI_LUT31	0x00000000	0x0	RW	0x155038c	Look-up Table register
QuadSPI_LUT32	0x01040120	0x0	RW	0x1550390	Look-up Table register
QuadSPI_LUT33	0x00000000	0x0	RW	0x1550394	Look-up Table register
QuadSPI_LUT34	0x00000000	0x0	RW	0x1550398	Look-up Table register
QuadSPI_LUT35	0x00000000	0x0	RW	0x155039c	Look-up Table register
QuadSPI_LUT36	0x65041808	0x0	RW	0x15503a0	Look-up Table register
QuadSPI_LUT37	0x080c011c	0x0	RW	0x15503a4	Look-up Table register
QuadSPI_LUT38	0x00000000	0x0	RW	0x15503a8	Look-up Table register
QuadSPI_LUT39	0x00000000	0x0	RW	0x15503ac	Look-up Table register
QuadSPI_LUT40	0x71041808	0x0	RW	0x15503b0	Look-up Table register

8 Revision history

This sections summarizes revisions to this document.

Table 1. Revision history

Revision	Date	Section	Description
0	02/2017		Initial public release.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “ typicals ,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions .

NXP, the NXP logo, Freescale, the Freescale logo, and QorIQ are trademarks of are trademarks of NXP B.V. All other product or service names are the property of their respective owners. ARM, Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2017 NXP B.V.

.

Document Number AN5398
Revision 1, 03/2017