Application Note 110

Flash Programming with RealView Debugger

Document number: ARM DAI 110A

Issued: April 2003

Copyright ARM Limited 2003

Application Note 110
Flash Programming with RealView Debugger

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
April 2003	A	First release

Feedback on this Application Note

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

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

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction

Unlike standard RAM based memory, Flash memory cannot be programmed directly by the debugger. This is because of the block structure of flash and the various control signals that must be generated to access the device.

The exact process needed to write to flash memory varies depending on:

- the flash type
- the specific device used
- how this device is integrated into your design.

A standard install of *RealView Debugger* (RVD) includes built-in support for programming flash memory on the following ARM based platforms:

- Evaluator7T
- Integrator/AP
- ARM Evaluation Board (AEB-1)

Support for other ARM based hardware platforms must be added to RVD if required. To allow this RVD is supplied with algorithms suitable for all four major flash types:

- AMD
- Atmel
- Intel
- SST

Note: These algorithms are designed to be run under debugger control, not to form the basis of standalone flash programming code. For more details on developing standalone flash programming code refer to ARM Application Note: 111

This application note examines how RVD can be used to program flash memory on your hardware target. The two main aims are to:

- Describe the mechanism RVD uses to program flash.
- Offer a step-by-step guide to adding support (to RVD) for flash devices on custom ARM based hardware platforms.

It should be read in association with the documentation supplied with RVD. In particular you should refer to:

- RVD 1.6.1 Users Guide - Working with Flash.
- RVD 1.6.1 Target Configuration Guide - Configuring Custom Targets

It assumes the user has access to:

- RVD v1.6.1 and a compatible JTAG debug interface (for example ARM Multi-ICE or RealView ICE)
- The *ARM Developer Suite* (ADS) or *RealView Compiler Tools* (RVCT) build tools.

2 An overview of how flash programming works in RVD

RVD enables you to program Flash Memory (that is, download programs or patch code/data). It does this using information provided in two target-specific files :

- *Board Chip Definition* (BCD) file
- *Flash Method* (FME) file

BCD files add extended target visibility to RVD. As a minimum the BCD file must specify where in your memory map flash is located and reference an appropriate FME file.

FME files are produced from a standard axf image using a special RVD utility called `pakflash`. They combine textual descriptive information (in `.ame` files) about the flash devices on your target with appropriate code algorithms for reading, writing and erasing. This code is run on the target (under debugger control) when you select options in the RVD Flash Programming Window.

Source code, `.ame` and FME files are supplied with RVD for supported targets.

- `..\flash`

  contains example `.ame` files and source files.

- `..\flash\examples`

  contains the RVD project files and the resultant prebuilt flash method files.

An example of how to program flash memory on supported targets is provided in *Section 3 Programming Supported Targets.*

If your target is not one of the supported types you must create your own versions of these files. This process is described by example in *Section 5* of this application note.

3 Programming supported targets

By default RVD is supplied with support for programming flash memory on the following ARM based platforms:

- Evaluator7T
- Integrator/AP
- ARM Evaluation Board (AEB-1)

3.1 Programming flash on an Integrator/AP development board

The ARM Integrator/AP platform includes 32MB of flash memory for user applications. This is located at address `0x24000000`.

To program this flash memory using RVD you must have specified an appropriate BCD in the connection properties window.

3.1.1 Specifying a BCD file for your connection

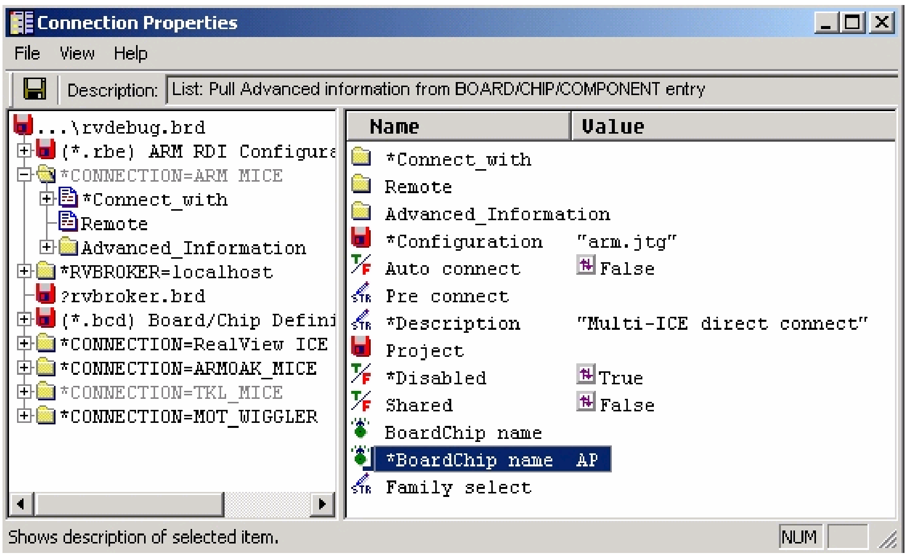
The following steps assume you are using Multi-ICE.

First make sure you are disconnected from the target (by unchecking the tickbox in the RVD Connection Control window). This ensures any changes made to your board file are applied when you reconnect to the target.

1. Open the connection properties window

2. Expand the ARM RDI configuration.

3. Expand the `Connection = Multi-ICE` folder.



Description: List: Pull Advanced information from BOARD/CHIP/COMPONENT entry

```
...\rvdebug.brd
(*.rbe) ARM RDI Configura
*CONNECTION=ARM MICE
  *Connect_with
  Remote
  Advanced_Information
*RVBROKER=localhost
?rvbroker.brd
(*.bcd) Board/Chip Defini
*CONNECTION=RealView ICE
*CONNECTION=ARMOAK_MICE
*CONNECTION=TKL_MICE
*CONNECTION=MOT_WIGGLER
```

Name	Value
*Connect_with	
Remote	
Advanced_Information	
*Configuration	"arm.jtg"
Auto connect	False
Pre connect	
*Description	"Multi-ICE direct connect"
Project	
*Disabled	True
Shared	False
BoardChip name	
*BoardChip name	AP
Family select	

Shows description of selected item. NUM

4. Highlight the entry for Boardchip name

5. Right click to select AP from the enumerated list.

3.1.2 Review the information contained in the AP BCD

At this point it is worth looking at exactly what information is stored in the Integrator/AP's BCD that enables you to program the flash memory.

To do this:

1. open the Connection Properties window

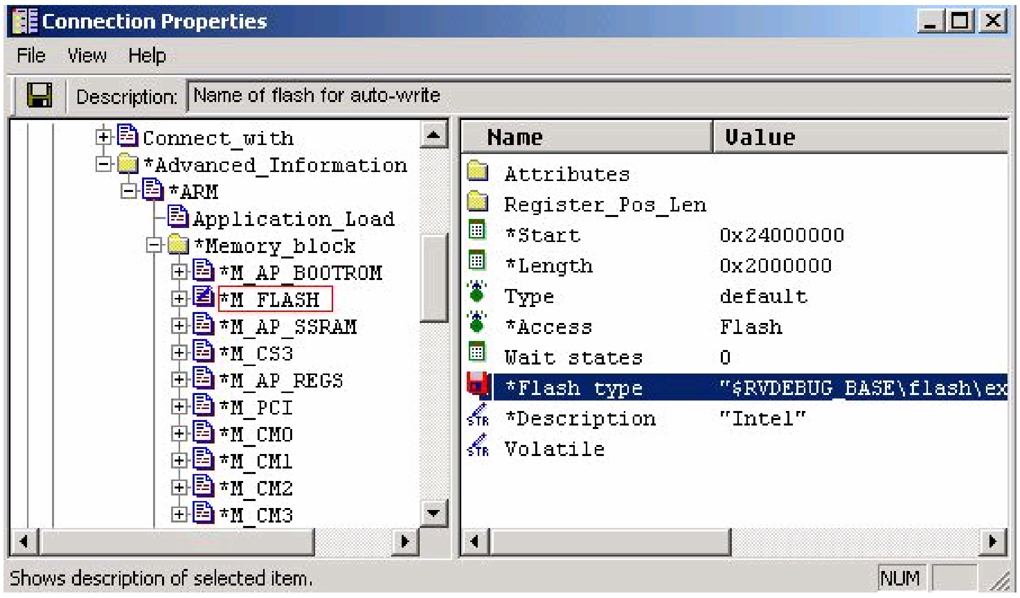
2. expand the entry for BCD files.

3. Select the AP BCD and expand the `Board=AP` folder in the right hand pane.

4. Navigate to the `Memory Block` Folder where the memory map of the AP is defined.

5. Open the `M_FLASH` folder.

Note the entries which specify the address range of the flash and reference the FME file. This is the information RVD references when you attempt to program flash memory.

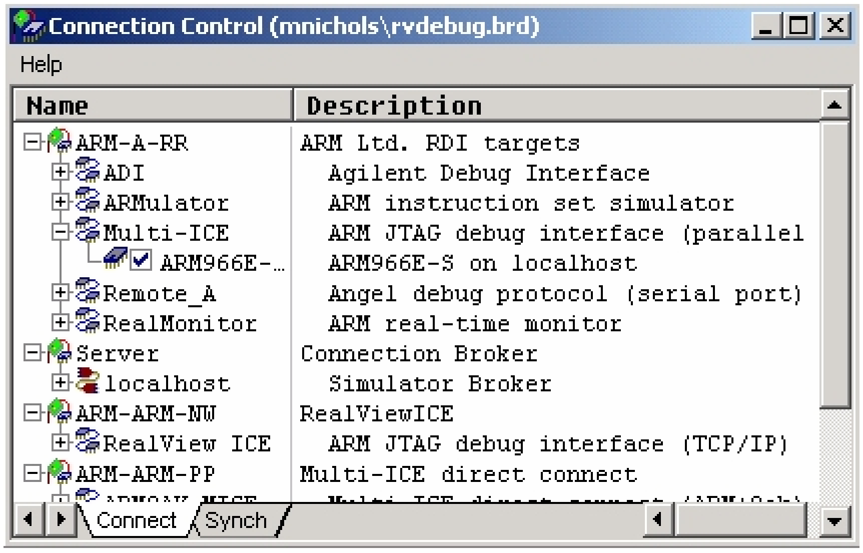


Name	Value
Attributes	
Register_Pos_Len	
*Start	0x24000000
*Length	0x2000000
Type	default
*Access	Flash
Wait states	0
*Flash type	"$RVDEBUG_BASE\flash\ex
*Description	"Intel"
Volatile	

6. Close the connection properties window without making any further changes.

3.1.3 3.1.3 Connecting to the target

Open the connection control window and connect to the target.

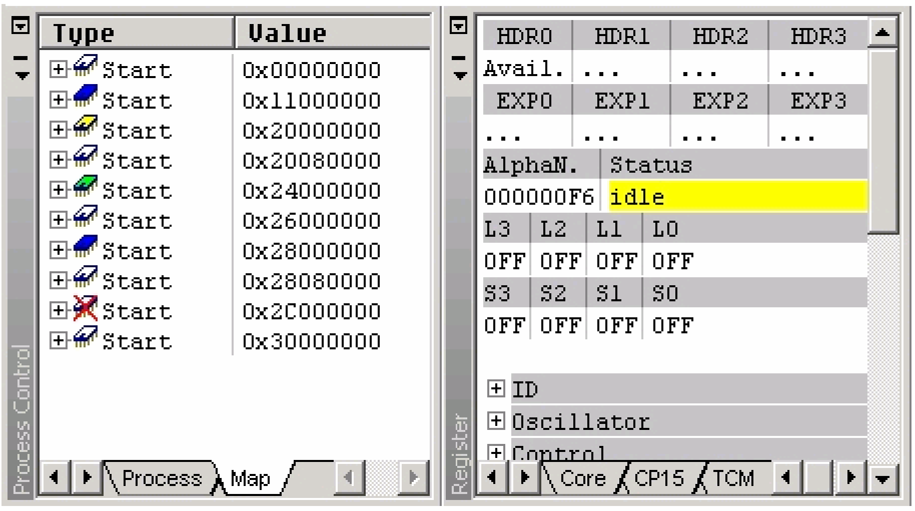


Connection Control (mnichols\rvdebug.brd)

Name	Description
ARM-A-RR	ARM Ltd. RDI targets
ADI	Agilent Debug Interface
ARMulator	ARM instruction set simulator
Multi-ICE	ARM JTAG debug interface (parallel
ARM966E-…	ARM966E-S on localhost
Remote_A	Angel debug protocol (serial port)
RealMonitor	ARM real-time monitor
Server	Connection Broker
localhost	Simulator Broker
ARM-ARM-NW	RealViewICE
RealView ICE	ARM JTAG debug interface (TCP/IP)
ARM-ARM-PP	Multi-ICE direct connect

Open a register pane in RVD. A new AP tab is visible. This shows enumerated information about the Integrator/AP board.

Open a Process Control pane in RVD and select the Map tab. This window shows the memory map for the Integrator/AP platform which has been defined in the BCD for the AP. Note the location of flash memory, highlighted in green.



Type	Value
Start	0x00000000
Start	0x11000000
Start	0x20000000
Start	0x20080000
Start	0x24000000
Start	0x26000000
Start	0x28000000
Start	0x28080000
Start	0x2C000000
Start	0x30000000

HDR0	HDR1	HDR2	HDR3
Avail.	...	...	...
EXP0	EXP1	EXP2	EXP3
...	...	...	...

AlphaN.	Status
000000F6	idle

L3	L2	L1	L0
OFF	OFF	OFF	OFF
S3	S2	S1	S0
OFF	OFF	OFF	OFF

ID
Oscillator
Control

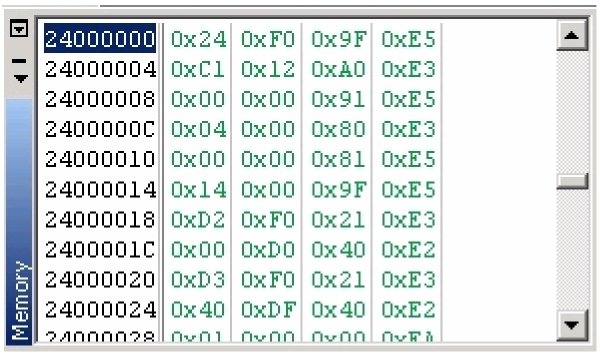
Programming the flash

When writing to an area of memory defined as flash RVD automatically invokes the flash memory control dialog. Normally this occurs when you either:

- use one of the memory/register operations from the RVD debug menu
- attempt to load an image to flash.

This example shows how to fill a small portion of flash memory with a pattern.

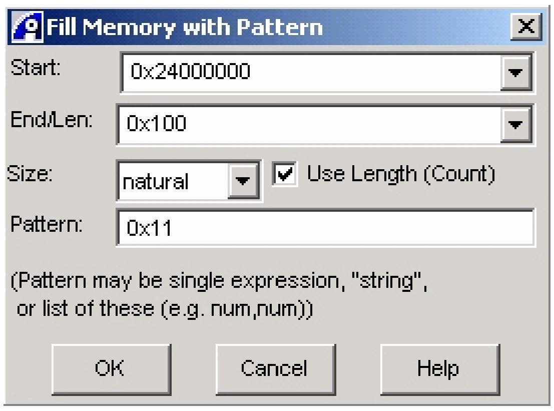
Open a memory pane in RVD and set it to view address `0x24000000` the start of flash memory.



```
24000000 0x24 0xF0 0x9F 0xE5
24000004 0xC1 0x12 0xA0 0xE3
24000008 0x00 0x00 0x91 0xE5
2400000C 0x04 0x00 0x80 0xE3
24000010 0x00 0x00 0x81 0xE5
24000014 0x14 0x00 0x9F 0xE5
24000018 0xD2 0xF0 0x21 0xE3
2400001C 0x00 0xD0 0x40 0xE2
24000020 0xD3 0xF0 0x21 0xE3
24000024 0x40 0xDF 0x40 0xE2
```

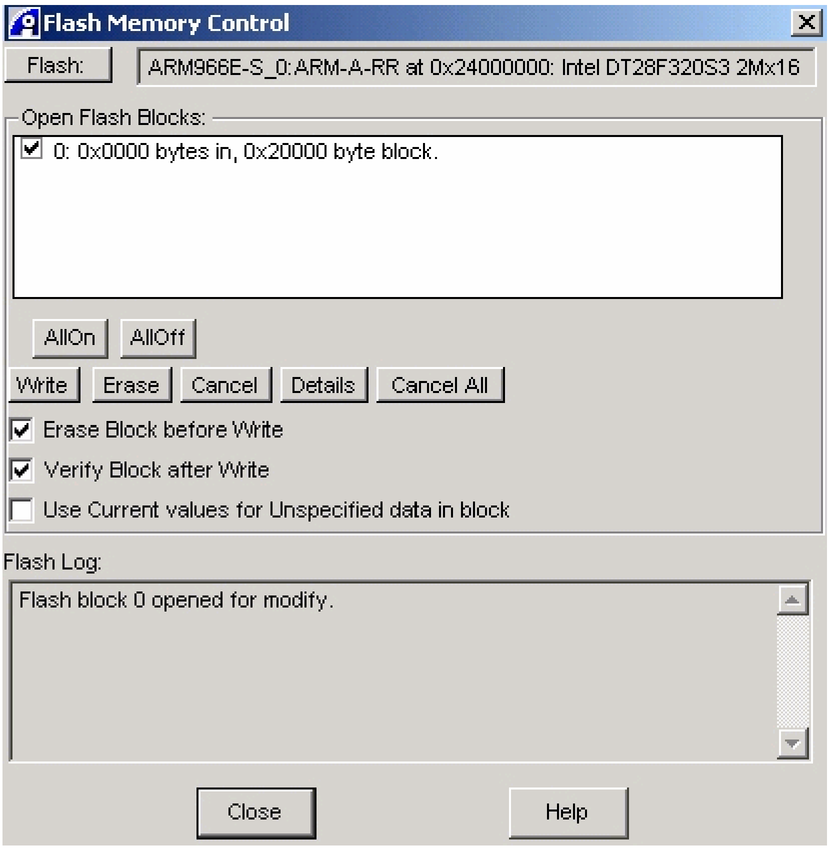
Select Debug, Memory Register Operations, Fill memory with Pattern.

Set the start address as `0x24000000`, length as `0x100`, size as natural and pattern as `0x11`.



Start: 0x24000000
End/Len: 0x100
Size: natural ☑ Use Length (Count)
Pattern: 0x11
(Pattern may be single expression, "string", or list of these (e.g. num,num))

Click OK and the Flash Memory Control Dialog appears.


Flash Memory Control

Flash: ARM966E-S_0:ARM-A-RR at 0x24000000: Intel DT28F320S3 2Mx16

Open Flash Blocks:

- [✓] 0: 0x0000 bytes in, 0x20000 byte block.

- [✓] Erase Block before Write
- [✓] Verify Block after Write
- [ ] Use Current values for Unspecified data in block

Flash Log:

```
Flash block 0 opened for modify.
```

Note that Flash block 0 has been opened for modification. Click the Write button to actually write the pattern to flash and close the window. Note the modified contents of the memory window.

4 Adding flash memory support to RVD for your own target hardware

If you are not using an ARM development board suitable BCD and FME files are not supplied with RVD. The process of generating your own BCD and FME files is described by example in *Section 5 Example Port*.

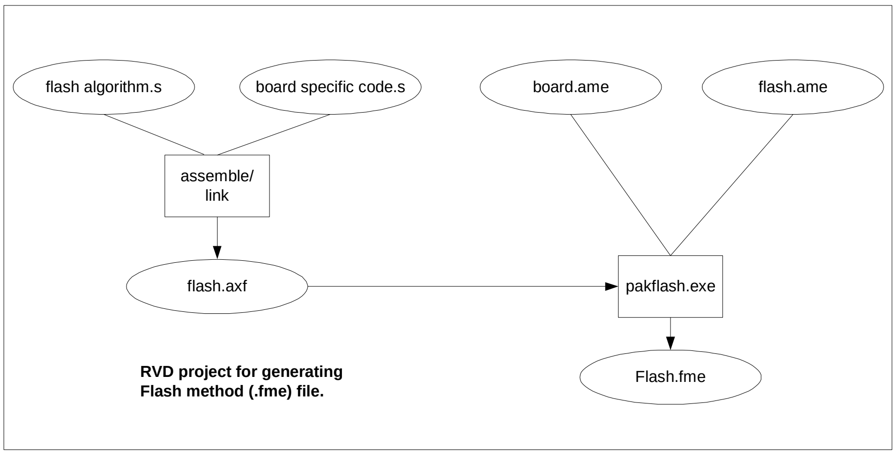
4.1 Support for ARM based development platforms provided by third parties

If you are using a standard development platform from a third party supplier it is possible that suitable BCD and FME files for use with RVD might be available for your development board. Contact your supplier to enquire about this.

4.2 Unsupported development boards and custom hardware

If your development platform is not supported or you are working with your own custom hardware then you must create your own BCD and FME file.

The figure below shows how an RVD project can be used to generate an FME file.



4.2.1 Flash Algorithm code

ARM provides support code, in the form of flash algorithms, for most of the major flash types. The following files can be found in the RVD `..\flash directory`:

`f_amd_sst_arm.s`

`f_atmel_arm.s`

`f_intel_arm.s`

You must provide all other files.

Note: If you are not using one of these flash types you must provide your own flash programming algorithms. Integrating this code with RVD is discussed in Appendix C.

4.2.2 Board specific code

Board specific code for your target must `INCLUDE` the appropriate flash algorithm code and perform any board specific operations (for example: unlock operations) that might be necessary to access flash. A basic example (with comments) is supplied with this application note. Alternatively you can edit one of the supported target board files supplied in the RVD `..\flash` directory. For example:

`b_IntegratorAP.s`

`b_evaluator7t.s`

4.2.3 Board-level .ame file

A board level `.ame` file must contain a text description of the type and configuration of flash device(s) used on your target hardware. Examples of existing board level `.ame` files for supported ARM targets are in the RVD `..\flash directory`. For example:

`board_intel_arm.ame`

`board_sst_eval7T.ame`

4.2.4 Flash-level .ame file

The flash-level `.ame` file is used to describe the flash devices themselves. Examples of existing flash-level ame files used on supported ARM targets are in the RVD `..\flash` directory. For example:

`flash_amd.ame`

`flash_atmel.ame`

Refer to Appendix A for details on the format of `.ame` files.

5 Example Port

We have chosen to use the ARM Evaluator 7T board as an example platform to port to. This contains a single SST 39VF400A 4 Mbit Multi-Purpose Flash device.

Note this is a somewhat academic exercise, as support for this platform is already included in RVD. However the steps provided assume 'from scratch' generation of new BCD and FME files and are exactly the same as those that are needed for any target.

The Appendices contain a more in depth guide on editing the assembler source and descriptive `.ame` files to match your target.

The process of adding flash support can be split into three stages:

1. Gathering information about your target.
2. Creating an RVD project and building an FME file
3. Generating a BCD

5.1 Gathering information about your target

To provide flash support for your target you need the following information:

- A copy of the datasheet for the flash device used on your target.
- A memory map of your development board showing where flash and RAM are located
- Details of any board specific code which might be pertinent.

The Evaluator7T board used for this application note has an SST39VF400A flash device fitted. The device is comprised of 256K 16bit words. (that is: 512K bytes). The sector size is 2K 16bit words. (that is: 4K bytes).

The memory map for the Evaluator7T board is defined in the ARM Evaluator-7T Board Users Guide (ARM DUI 0134A) as follows.

Address range	Size	Description
`0x00000000` to `0x0003FFFF`	256KB	SRAM bank
`0x00040000` to `0x0007FFFF`	256KB	SRAM bank
`0x01800000` to `0x0187FFFF`	512KB	Flash
`0x03FE0000` to `0x03FE1FFF`	8KB	Internal SRAM

Note: the bottom of flash contains the bootstrap loader, Angel and other utilities and must not be edited by the user. Address `0x01820000` to `0x0187FFFF` are available for application code and data.

No board specific code (for example: unlock codes for specific memory regions) is required to access flash on the Evaluator7T.

5.2 Creating an RVD project and building an FME file

This section assumes some familiarity with the process of managing projects with RVD. Please refer to the Managing Projects chapter of the RVD version 1.6.1 User Guide for more details.

5.2.1 Locate the source and `.ame` files required

Create a suitable directory on your hard drive and copy in the appropriate source and `.ame` files specified below.

- `b_flashwrapper.s` (board specific code).
- `f_amd_sst_arm.s` (flash algorithm code)

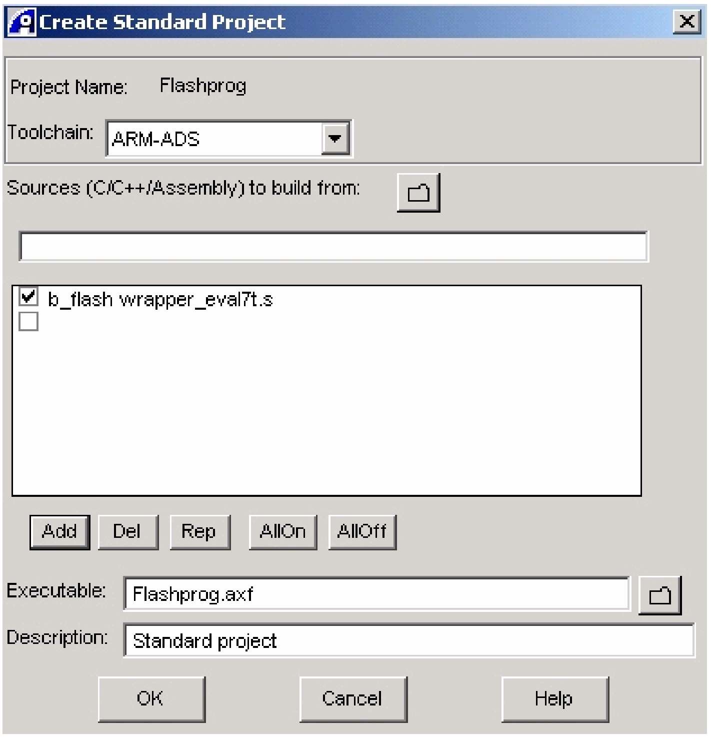
The first is supplied with this application note and the second should be copied from the RVD `../flash` directory.

- `board_sst_eval7T.ame` (board-level `.ame` file)
- `flash_sst.ame` (flash-level `.ame` file)

Both the above files are supplied with this application note. These are needed by the `pakflash` utility called in the projects post link step.

5.2.2 Create a standard RVD project

Create a standard RVD project and add the board-level assembler source. The project can be built using either the ADS or RVCT tool chain.



Create Standard Project

Project Name: Flashprog

Toolchain: ARM-ADS

Sources (C/C++/Assembly) to build from:

☑ b_flash wrapper_eval7t.s

Executable: Flashprog.axf

Description: Standard project

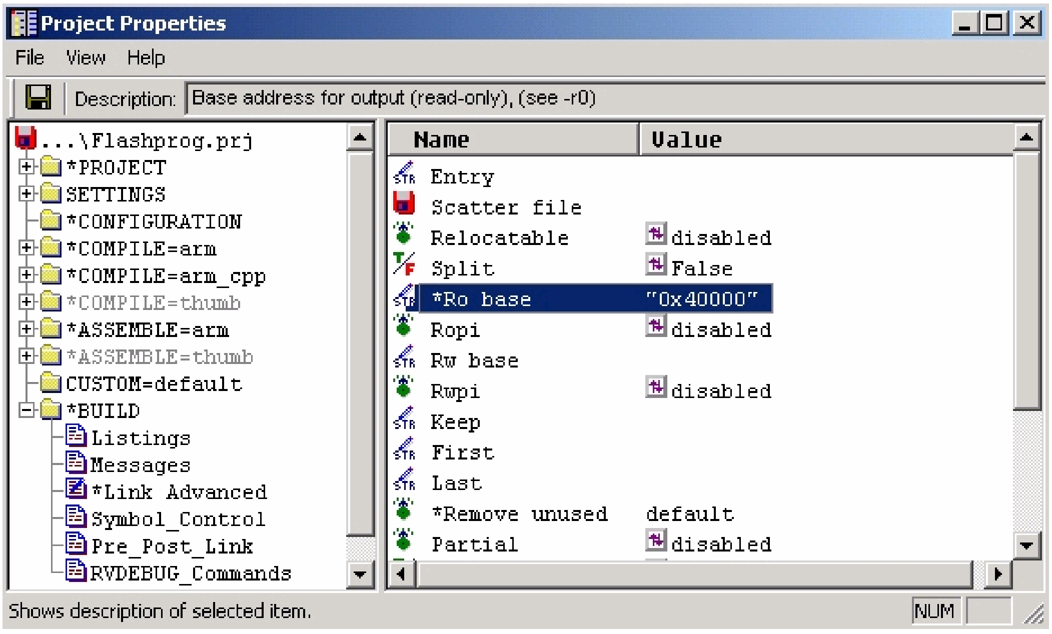
Note: `b_flashwrapper.s` INCLUDES `f_amd_sst_arm.s` and so does not need to be added manually.

5.2.3 Project Build folder settings

In the project build folder:

- Use application to name the axf you wish to generate. (for example:`Evaluator.axf`)
- Under link advanced set the `RoBase = 0x40000` (Evaluator SRAM)

  This value must specify an area of RAM on your target.


Project Properties
File View Help
Description: Base address for output (read-only), (see -r0)
Shows description of selected item.
NUM

```
...\Flashprog.prj
  *PROJECT
  SETTINGS
  *CONFIGURATION
  *COMPILE=arm
  *COMPILE=arm_cpp
  *COMPILE=thumb
  *ASSEMBLE=arm
  *ASSEMBLE=thumb
  CUSTOM=default
  *BUILD
    Listings
    Messages
    *Link_Advanced
    Symbol_Control
    Pre_Post_Link
    RVDEBUG_Commands
```

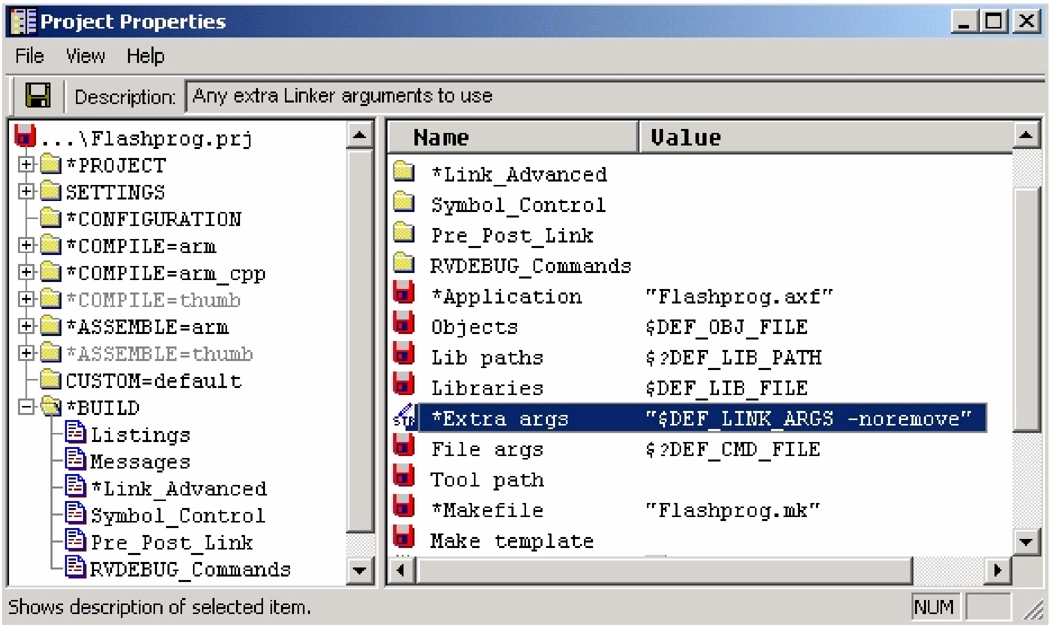
Name	Value
Entry	
Scatter file	
Relocatable	disabled
Split	False
*Ro base	"0x40000"
Ropi	disabled
Rw base	
Rwpi	disabled
Keep	
First	
Last	
*Remove unused	default
Partial	disabled

5.2.4 Adding "Extra args"

You must add an "Extra args" line to the build folder:

`$DEF_LINK_ARGS -noremove`

This prevents the linker removing the uninit RAM buffer.


Project Properties
File View Help
Description: Any extra Linker arguments to use
Shows description of selected item.
NUM

```
...\Flashprog.prj
  *PROJECT
  SETTINGS
  *CONFIGURATION
  *COMPILE=arm
  *COMPILE=arm_cpp
  *COMPILE=thumb
  *ASSEMBLE=arm
  *ASSEMBLE=thumb
  CUSTOM=default
  *BUILD
    Listings
    Messages
    *Link_Advanced
    Symbol_Control
    Pre_Post_Link
    RVDEBUG_Commands
```

Name	Value
*Link_Advanced	
Symbol_Control	
Pre_Post_Link	
RVDEBUG_Commands	
*Application	"Flashprog.axf"
Objects	$DEF_OBJ_FILE
Lib paths	$?DEF_LIB_PATH
Libraries	$DEF_LIB_FILE
*Extra args	"$DEF_LINK_ARGS -noremove"
File args	$?DEF_CMD_FILE
Tool path	
*Makefile	"Flashprog.mk"
Make template	

5.3 Calling Pakflash

Under the Pre_Post_Link folder of the project include the following post link step.

```
'$RVDEBUG_BASE\mwbin\pakflash' -f  ARM_EVAL7T debug\Evaluator.axf
board_sst_eval7T.ame -o ARM_EVAL7T.FME
```

The format is:

```
pakflash.exe -f flashname b_boardname.axf board_boardname.ame -o
ARM_EVAL7T.FME
```

where:

`flashname` = is specified in the board-level `.ame` file. For example: `[BOARD=ARM_EVAL7T]`

`b_boardname.axf` = is the axf produced by the project.

`b_boardname.ame` = the board-level `.ame` file. For example: `board_sst_eval7T.ame`

Now build the project. There should be no errors or warnings.

This produces an axf linked to run from address `0x40000`. (Evaluator SDRAM).The post link step then calls the RVD utility `pakflash` to produce an FME file. This combines the text information from the `.ame` file: `board_sst_eval7T.ame` with the axf to produce the FME output file.

Note: If you want you can call `pakflash.exe` manually from the command line.

5.4 Checking your FME file with Dispflash

You can check the contents of an FME file using a utility called `dispflash.exe` (a copy is provided with this Application note).

This can be called from the command line using the following syntax:

```
dispflash.exe ARM_EVAL7T.FME
```

The output produced is:

```
#=====================================================
Flash 'Silicon Storage Tech 39VF400A' for processor 'ARM' (Little
Endian)
Width=2 with erase value of 0xFFFF
1 Block Groups:
  (0) 128 blocks with byte size 4096/0x1000
Routine Code PC-rel. Can load at 0x40000
Init routine 0x0030 bytes from start
Erase routine 0x007C bytes from start
Erase then Write routine 0x010C bytes from start
Write routine 0x0110 bytes from start
Validate routine 0x015C bytes from start
Breakpoint routine 0x0188 bytes from start
Separate Data image (vars) 0x05CC bytes from start
RAM Buffer at 0x401CC (page 0) with byte size of 1024
0 Locked blocks
0 Memory Regions to Restore
0 Registers to Restore
#=====================================================
```

5.5 Creating a Board Chip Definition (BCD) file

By default RVD looks for BCD files in the `..\etc` directory. There are a number of examples provided for various ARM development boards.

You can create a new BCD from within RVD by copying an existing BCD. For more details on board files refer to the section on Creating a *.BCD file in *RVD version 1.6 Target Configuration Guide.*

This application note does not describe all aspects of Extended Target Visibility (ETV), for example: enumeration of registers. It deals with those required to program the flash.

The steps needed to create a new minimal BCD for the Evaluator7T are as shown in this section.

5.5.1 Copy and rename an existing BCD

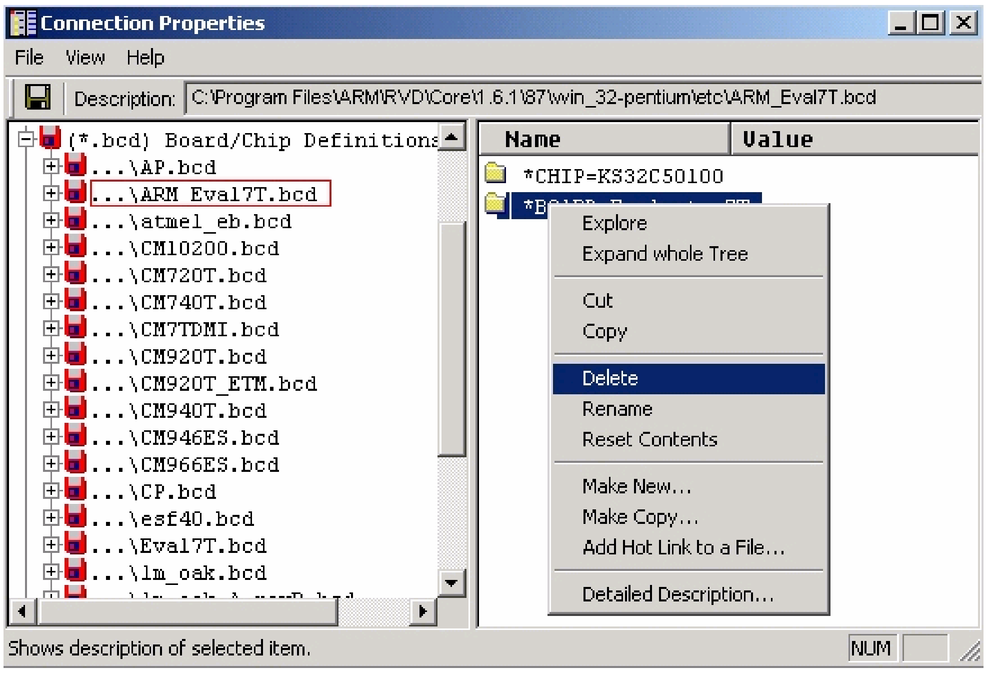
Locate an existing BCD file (for example: `Eval7T.BCD`) in the RVD `../etc` directory to provide a basis for your new file. Make a copy of this file and rename it `ARM_EVAL7T.BCD`.

5.5.2 Delete any existing Board/Chip definitions

1. Open the Connection Properties window in RVD.

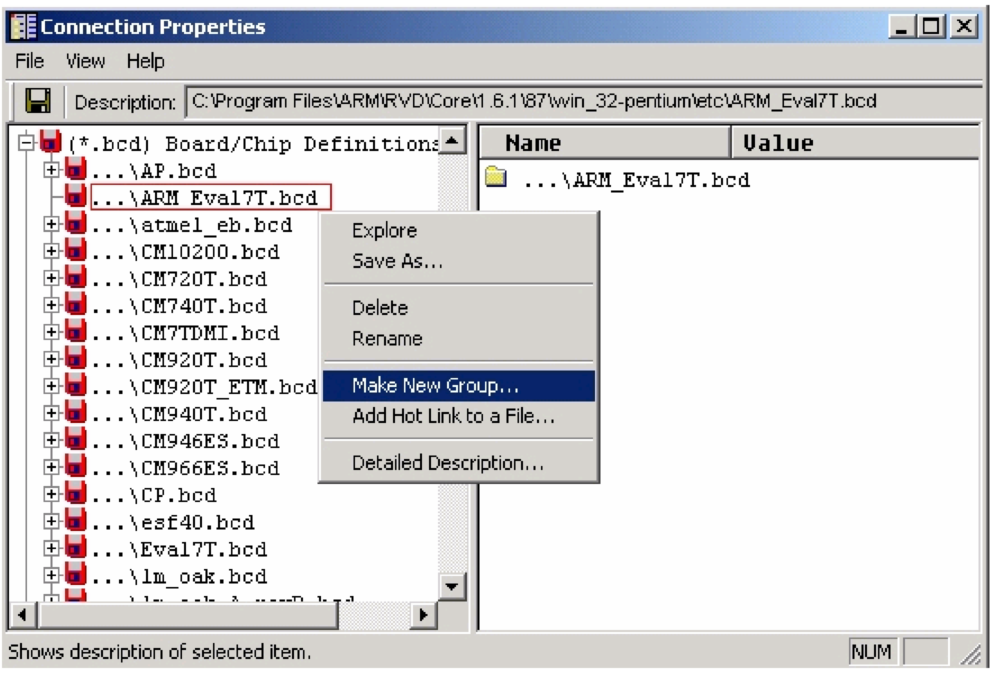
2. Select the new Board/Chip definition entry for the `ARM_EVAL7T` you have just created.

3. Delete any existing board or chip definitions that were part of the original BCD.

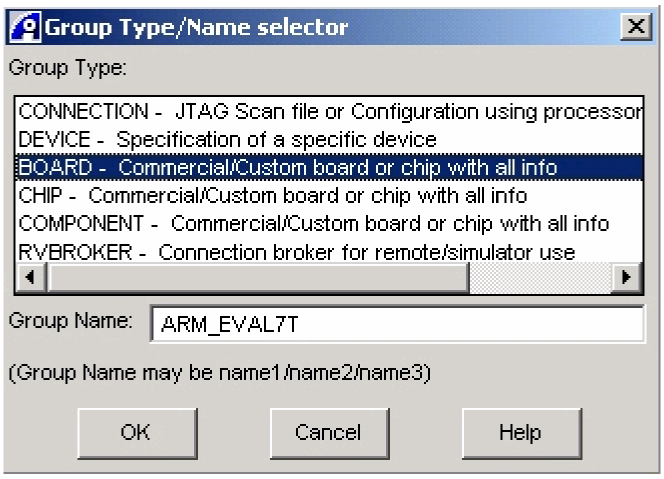


5.5.3 Creating a new board group.

1. Right click on the new BCD entry and select Make New Group.



2. Create a new BOARD group entry called ARM_EVAL7T.


Group Type/Name selector
Group Type:
CONNECTION - JTAG Scan file or Configuration using processor
DEVICE - Specification of a specific device
BOARD - Commercial/Custom board or chip with all info
CHIP - Commercial/Custom board or chip with all info
COMPONENT - Commercial/Custom board or chip with all info
RVBROKER - Connection broker for remote/simulator use
Group Name:
ARM_EVAL7T
(Group Name may be name1/name2/name3)
OK
Cancel
Help

Note: This name must match the BOARD= entry used in your board-level .ame file.

5.5.4 Describe the Evaluator7T memory map.

1. Select the new `ARM_EVAL7T` BCD entry

2. Expand the `BOARD=ARM_EVAL7T` folder in the right hand pane.

3. Navigate to the `Memory Block` folder in the `Advanced_Information` tree. This is where you can describe your target runtime memory system to RVD.

Note: You must set up definitions for all areas of memory you need to access on your target. RVD aborts accesses to undefined areas of memory when using a BCD file.

4. Open the `Memory Block` folder and select the default entry.

5. Right click and select `Make Copy` to create the following folders.

Create folder `Eval_App_Flash`

Specify the following attributes (all others can be left as default).

`Start = 0x01820000`

`Length = 0x60000`

`Access = Flash` (right click and select from enumerated list)

`Flash type:` full path and name of your FME file.

`Description = "Application Flash 384K"`

Create folder `SRAM_bank1`

Specify the following attributes (all others can be left as default).

`Start = 0x0`

`Length = 0x40000`

`Access = RAM` (right click and select from enumerated list)

`Description = "SRAM bank1 256K"`

Create folder `SRAM_bank2`

Specify the following attributes (all others can be left as default).

`Start = 0x40000`

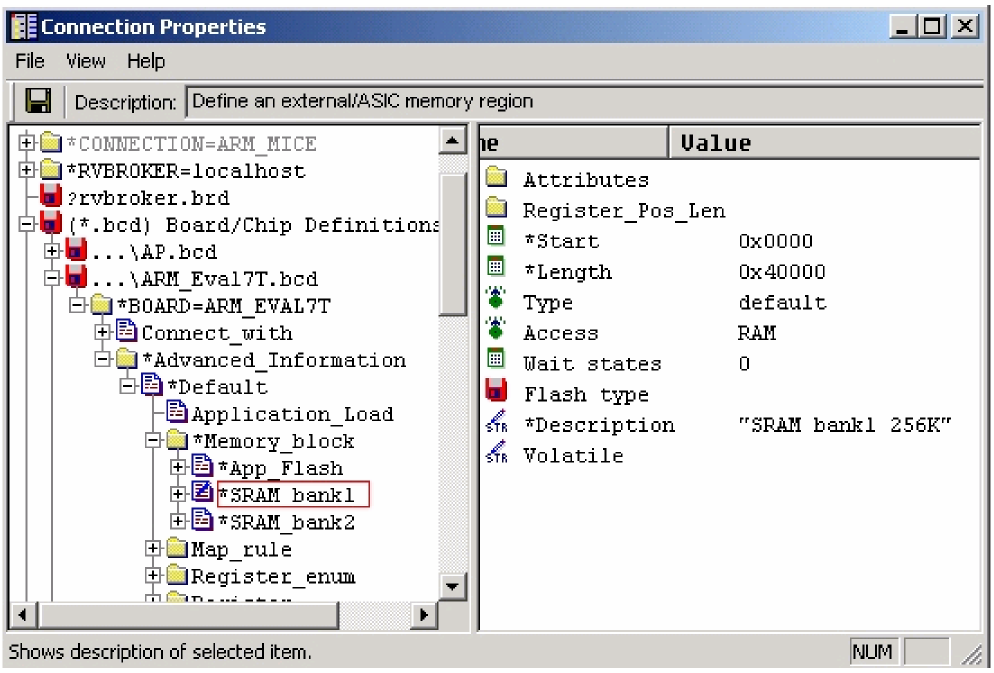
`Length = 0x40000`

`Access = RAM` (right click and select from enumerated list)

`Description = "SRAM bank2 256K"`

6. Now deleted the default entry.

7. The Connection Properties window Memory Block group will look as shown:



Connection Properties

File View Help

Description: Define an external/ASIC memory region

```
*CONNECTION=ARM_MICE
*RVBROKER=localhost
?rvbroker.brd
(*.bcd) Board/Chip Definitions
  ...\AP.bcd
  ...\ARM_Eval7T.bcd
    *BOARD=ARM_EVAL7T
      Connect_with
      *Advanced_Information
        *Default
          Application_Load
          *Memory_block
            *App_Flash
            *SRAM_bank1
            *SRAM_bank2
          Map_rule
          Register_enum
```

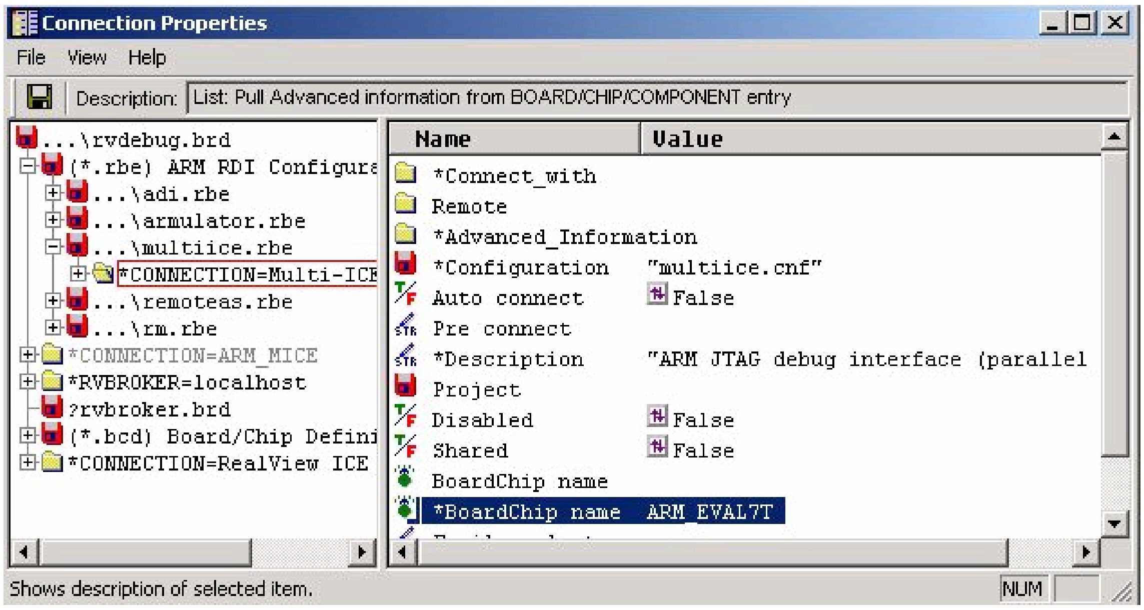
Name	Value
Attributes	
Register_Pos_Len	
*Start	0x0000
*Length	0x40000
Type	default
Access	RAM
Wait states	0
Flash type	
*Description	"SRAM bank1 256K"
Volatile	

Shows description of selected item.

5.5.5 Specify the new BCD in the connection properties for Multi-ICE or RVICE.

Close the RVD connection properties window saving the changes you have made to your board file.

Open the Connection Control Window and select the new BCD you have created as the board chip file for your connection.



Connection Properties

File View Help

Description: List: Pull Advanced information from BOARD/CHIP/COMPONENT entry

```
...\rvdebug.brd
(*.rbe) ARM RDI Configura
  ...\adi.rbe
  ...\armulator.rbe
  ...\multiice.rbe
    *CONNECTION=Multi-ICE
  ...\remoteas.rbe
  ...\rm.rbe
*CONNECTION=ARM_MICE
*RVBROKER=localhost
?rvbroker.brd
(*.bcd) Board/Chip Defini
*CONNECTION=RealView ICE
```

Name	Value
*Connect_with	
Remote	
*Advanced_Information	
*Configuration	"multiice.cnf"
Auto connect	False
Pre connect	
*Description	"ARM JTAG debug interface (parallel
Project	
Disabled	False
Shared	False
BoardChip name	
*BoardChip name	ARM_EVAL7T

Shows description of selected item.

You can now see the memory map you have defined in RVD's process control pane when you connect to the target.

RVD now automatically displays the flash memory control dialog and accesses the new FME file you have created when you attempt to write to flash memory.

6 Appendix A - Format of `.ame` files

This appendix defines the minimum information that must be specified in board-level and flash-level `.ame` files used to describe your targets flash memory.

6.1 Board-Level `.ame` files

Example:

```
#=====================================================
[BOARD=IntegratorAP]
proc_name=ARM
from_flash=DT28F320S3_AP
width=0x0004
reloc.start_addr=0x28000000
reloc.pc_rel=True
[INCLUDE] $RVDEBUG_INSTALL\flash\flash_intel.ame
#=====================================================
```

`BOARD` - defines board file name.

`proc_name` - always `"ARM".`

`from_flash` - references the relevant flash device in the lower level `.ame` file.

`width` - Resultant data width when writing to the flash. `[4=32bit, 2=16bit, 1=8bit]`

In this example there are two 16bit flash devices in parallel across the data bus. Therefore the `width = 4` so we write to the bottom 16bits of each flash simultaneously. This entry should match the entry defined in the board-level assembler code.

`reloc.start_addr` - specifies the address to run the flash programming code from. It should match the RO base address used in your RVD project link options. *Note*: This does not have to be free 'scratch' memory. By default RVD saves the contents of affected RAM and restores them after the flash operation has completed.

`reloc.pc_rel` - indicates PC relative code (normally `True`)

`INCLUDE` - specifies the relevant flash `.ame` file (with path)

6.2 Flash-Level `.ame` files

Example:

```
#=====================================================
[FLASH=DT28F320S3_AP]
flash_name="Intel DT28F320S3 2Mx16 x2 x4"
no_erase=False
width=4
# Each Flash has 64 blocks(32K-halfwords x 2) as main blocks
block.1={count=64:size=128K}
block.2={count=64:size=128K}
block.3={count=64:size=128K}
block.4={count=64:size=128K}
proc_name="ARM"
#=====================================================
```

`FLASH` - defines a name for a specific flash device.

`flash_name` - text description displayed by RVD

`no_erase` - always be set to `False` by default

Setting to `True` might improve performance in some NOR style flashes that support auto-erase.

`width` - width of the flash device. `[4=32bit, 2=16bit, 1=8bit]`

This entry is overridden by the board-level `.ame` file.

`block` - list of block groups.

> **`size`** - size in bytes of each block. Can be specified in decimal, hex or K.
>
> **`count`** - defines the number of blocks in each group.

`proc_name` - always `"ARM".`

7 Appendix B - Board-level assembly code for supported flash types

For targets that use one of the four main flash types supported by RVD (AMD, Atmel, Intel and SST) sample board-level assembly code (complete with comments) is provided in the associated file: `b_flashwrapper.s`.

This code must be included as part of an RVD project to produce an FME for your target. You have to provide board and flash-level ame files and select the appropriate flash algorithm for your flash type. (The process is described in Section 5 of this application note).

Note1: This code is designed to be linked to reside in a contiguous area of RAM on your target. (So only an RO base should be specified in the linker options).

Note2: This does not have to be free 'scratch' memory. By default RVD saves the contents of affected RAM and restores them after the flash operation has completed.

The areas of `b_flashwrapper.s` that are designed to be user editable are:

Equates.

You must set `P_WIDTH` and `WIDTH` to match your target.

Flash_init routine

The `FLASH_init` label is exported, enabling the routine to be called directly by RVD.

Executing this function ultimately results in a branch to a label `FLASH_break`. RVD automatically places a breakpoint at this point enabling the debugger to halt the target while the Flash Memory control dialog is displayed.

An area is marked for you to insert your own code if required. For example: you might want to include code to disable watchdogs, enable chip selectors/enables to allow writes for example. Note the RVD API rules in the comments, which specify that only `R6`,`R7` and `R8` are available as scratch registers.

Buffer size

`Buffer` defines an area of uninitialised RAM which RVD can use to store data before writing it to flash. Size is not flash dependant and therefore does not usually need editing. However for large flash memory devices an increased RAM buffer size might improve write speeds.

8 Appendix C - Adding support for other flash devices

If your target uses a flash device that cannot be programmed using the standard flash algorithms provided with RVD, you must provide your own flash programming code.

The following files are provided with this application note:

- `rvd2apcs.s`
- `flash.h`

The first is an assembly wrapper intended to acts as an interface between the RVD API and your flash programming code.

The second is a 'C' header file containing prototypes for the functions called by RVD.

These files must be included as part of an RVD project to produce an FME for your target. You have to provide board and flash-level ame files to describe your target to RVD. (The process is described in section 5 of this application note).

Note1: This code is designed to be linked to reside in a contiguous area of RAM on your target. (so only an RO base should be specified in the linker options).

Note2: This does not have to be free 'scratch' memory. By default RVD saves the contents of affected RAM and restores them after the flash operation has completed.