
Flash Programming with RVD



Title	Flash programming with RVD
Keywords	RVD, Flash
Abstract	This is a guide for creating the flash method file and board/chip definition file for RVD

Copyright © 2007 Samsung Electronics Co, Ltd. All Rights Reserved.

Though every care has been taken to ensure the accuracy of this document, Samsung Electronics Co, Ltd. cannot accept responsibility for any errors or omissions or for any loss occasioned to any person, whether legal or natural, from acting, or refraining from action, as the result of the information contained herein. Information in this document is subject to change at any time without obligation to notify any person of such changes.

Samsung Electronics Co, Ltd. may have patents or patent pending applications, trademarks copyrights or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give the recipient or reader any license to these patents, trademarks copyrights or other intellectual property rights.

No part of this document may be communicated, distributed, reproduced or transmitted in any form or by any means, electronic or mechanical or otherwise, for any purpose, without the prior written permission of Samsung Electronics Co, Ltd.

The document is subject to revision without further notice.

All brand names and product names mentioned in this document are trademarks or registered trademarks of their respective owners.

Contact Address:

Samsung Electronics Co., Ltd.
San#24 Nongseo-Ri, Giheung_Eup,
Yongin_City, Gyeonggi-do, Korea 449-711

Tel: (82)-(031)-209-2379
Fax: (82)-(031)-209-6494

Home Page: <http://www.samsungsemi.com>
Contact us: youngseok.oh@samsung.com

MCU Design Team,
Chipcard & Microcontroller,
System LSI Division,
Semiconductor Business
Samsung Electronics Co., Ltd.

Revision History

Date	Version	Author	Amendment
Jul.24 2007	0.1	Youngseok Oh	Initial version
Sep.11.2007	1.0	Youngseok Oh	Finished



1. Overview and Requirements

OVERVIEW

This document describes how to create FME file and BCD file for different types of microcontroller.

FME is a flash method file that includes flash erase / write / validate routines. BCD is a board / chip definition file, in which the type, property, and range of flash memory is defined.

When using RVD as a primary debugger, RVD needs a flash fusing algorithm for different types of microcontroller. If the microcontroller is supported by RVD, it is simply to choose from the list. If not, the flash fusing algorithm shall be created according to the microcontroller specification. The FME and BCD files are the outputs of the flash fusing algorithm.

REQUIEREMENTS

- RVD
- GNU Make file (version 3.8 or higher)
- pakflash utility

2. Flash Programming in RVD

PROCEDURE

When an axf file is loaded into RVD, RVD determines whether it should be located in ROM or RAM with the address information given from axf and Board/Chip definition. If the address corresponds to ROM location, RVD pops up a programming dialog box for flash programming. Users are able to command flash programming with this dialog box. Brief procedures are following.

- 1) Open axf file within the flash memory range
- 2) RVD checks the address given from axf.
 - 2-1) If the address corresponds to the flash memory range as declared in BCD, it prepares to perform flash programming by loading fme file into RAM and executing the initialization routine in fme file.
 - 2-2) While initialization, RVD calculates the number of blocks to program with the address in axf file. The size of block (or sector) of flash memory is declared in bcd and fme file.
 - 2-3) RVD notifies the number of flash blocks corresponding to axf file.
- 3) Flashing
 - 3-1) If write command is given by the user, RVD first copies one block (or sector) from flash memory to the desktop computer.
 - 3-2) RVD erases one block
 - 3-3) RVD manipulates the contents in desktop computer, for example, if the complete block has to be replaced with new contents, if a part of block has to be replaced with new contents, etc.
 - 3-4) RVD starts to transmit the contents from the desktop computer to microcontroller, and calls flash write routine
 - 3-5) Often microcontroller does not have enough buffer space, so that the desktop computer may split and transmit the contents to the microcontroller a few times. (The size of buffer is defined in fme file).

In the procedure 3), RVD calls the corresponding FME file for different types of microcontrollers. If the microcontroller is supported by RVD, the user does not need to create own FME file. If not supported, the user shall build own FME file following the procedure in the next section.

3. Building Own FME File

DEVELOPING FME

When the microcontroller is not supported by RVD, users may have to build own FME file. The FME file is obtained from axf file by means of pakflash.exe as below.

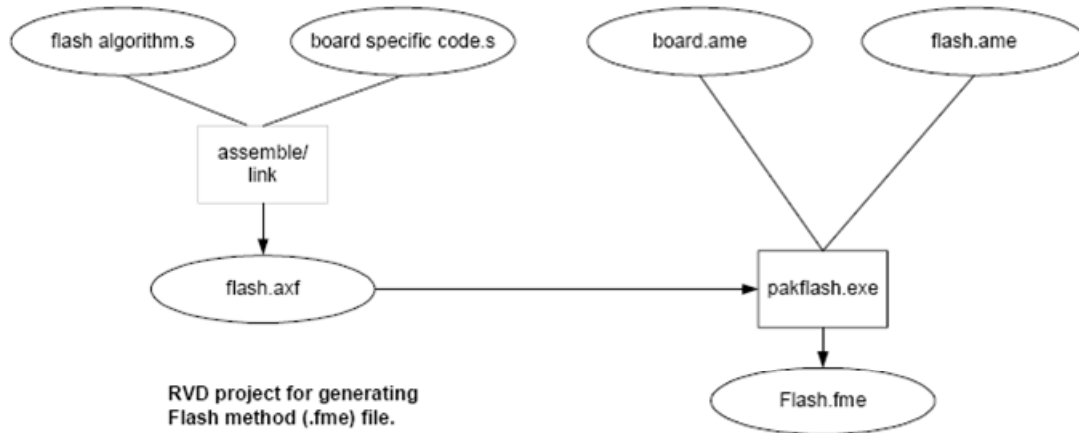


Figure 1 Generating FME file (Refer from ARM AN110)

What users have to build is flash algorithm, board specific code, and ame file. The flow described above should be managed by make file.

However, when the microcontroller or flash type is not supported by RVD, a wrapper connecting between RVD API and our own flash algorithm is necessary additionally.

A wrapper has following format.

```

EXPORT      FLASH_init
EXPORT      FLASH_erase
EXPORT      FLASH_write
EXPORT      FLASH_validate
EXPORT      FLASH_break

EXPORT      buffer

PRESERVE8
AREA FLASH, CODE, READONLY

;*****
;
; FLASH_init - Initialise any board specific memory access controls, etc.
;*****
;

FLASH_init

IMPORT RVDFlash_Init
LDR sp, =stacktop
MOV r0, r1 ; base memory address of flash device
BL RVDFlash_Init ; branch to customer function
B FLASH_break ; return

;*****
;
; FLASH_erase - erase a Flash block(s)
;*****
;

FLASH_erase
    
```

```

IMPORT RVDFLASH_Erase ; i is in r0
LDR sp, =stacktop
; STR lr, [sp, #-4]! ; preserve lr
MOV r0, r1 ; base memory address of first flash block
MOV r1, r2 ; number of flash blocks
MOV r2, r3 ; size of a single flash block in bytes
BL RVDFLASH_Erase ; branch to customer function
B FLASH_break ; return

;*****
;
; FLASH_write - write data to a Flash block
;*****
;

FLASH_write

IMPORT RVDFLASH_Write ; write with image
LDR sp, =stacktop
MOV r0, r1 ; base memory address of first flash block
MOV r1, r2 ; number of bytes to read/write
MOV r2, r4 ; offset into block (in bytes) to start read/write at
MOV r3, r5 ; address of buffer
STR r9, [sp, #-4]! ; verify flag
BL RVDFLASH_Write ; branch to customer function
B FLASH_break ; return

;*****
;
; FLASH_validate - validate a write to a Flash block
;*****
;

FLASH_validate

IMPORT RVDFLASH_Validate ; i is in r0
LDR sp, =stacktop
MOV r0, r1 ; base memory address of first flash block
MOV r1, r2 ; number of bytes to read/write
MOV r2, r4 ; offset into block (in bytes) to start read/write at
MOV r3, r5 ; address of buffer
BL RVDFLASH_Validate ; branch to customer function
B FLASH_break ; return

;*****
;
; FLASH_break - end of function entry for all Flash routines *
;*****
;

FLASH_break
nop
forever
b forever ; should never be reached.

;*****
;
;* DEFINE BUFFER FOR WRITE/VERIFY and STACK
;*****
;

AREA BUFFER, NOINIT
buffer % 16384 ; buffer for copying

AREA STACK, NOINIT
stackbottom
% 512

```

```

stacktop
% 4
END
    
```

First, export section includes RVD APIs and buffer. A buffer receives data from the desktop computer before flash programming. Special care should be taken in the variable "buffer". It has the size of 16,384bytes in the example above. It should be adjusted according to the size of SRAM of each device.

In each of RVD API, there is a function call to flash algorithm (e.g. RVDFlash_Init). Since this wrapper also defines the number of arguments and the type, C function with flash algorithm shall follow the format. The C function prototype is defined in the header file as below.

```

UINT32 RVDFlash_Init(UINT32 base_of_flash);

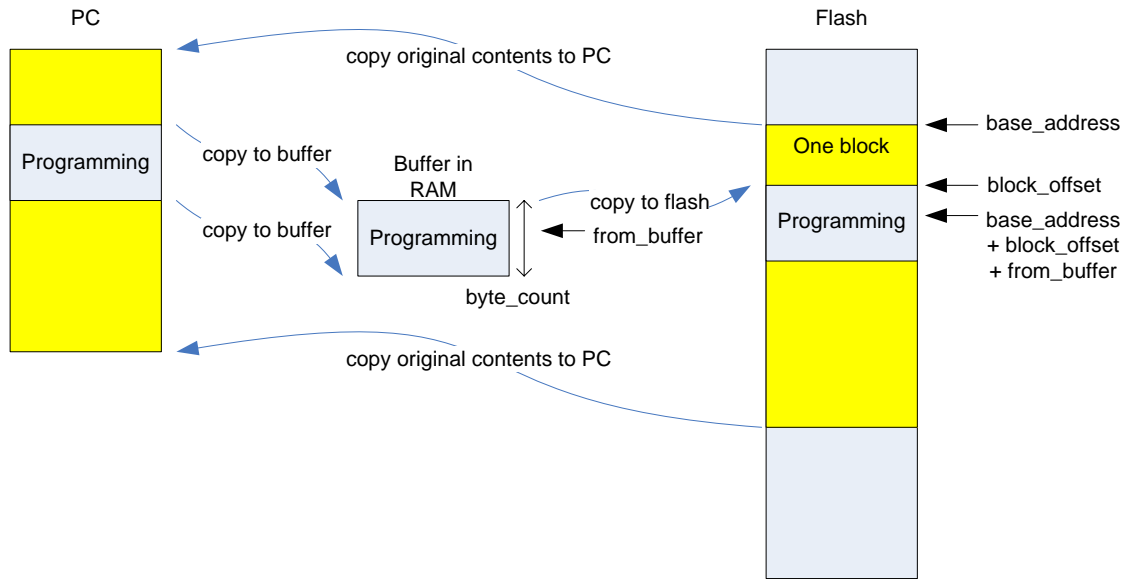
UINT32 RVDFLASH_Erase(UINT32* base_address,
UINT32 block_count,
UINT32 block_size);

UINT32 RVDFLASH_Write(UINT32* base_address,
UINT32 byte_count,
UINT32 block_offset,
                        UINT32* from_buffer,
UINT32 verify);

UINT32 RVDFLASH_Validate(UINT32 base_address,
UINT32 byte_count,
UINT32 block_offset,
                        UINT32 from_buffer);

/*****
* base_of_flash: base memory address of flash device
* base_address: base memory address of first flash block
* block_size: size of a single flash block in bytes
* block_count: number of flash blocks
* byte_count: number of bytes to read/write
* block_offset: offset into block (in bytes) to start read/write at
* from_buffer: address of buffer to copy from
* verify: 0 if no verify requested, else same value as byte_count
* return 0=ok, else error
*****/
    
```

The following figure illustrates the address and offset graphically.



Depending upon the size of buffer, the “copy to buffer” operation may occur one to several times. If it has same size as that of block, one time copy occurs. Unless otherwise, it may occur as many as the multiples. (e.g. 64k block with 1k buffer → 64 times)

The flash algorithm for AGPx is following.

Address Definition

```
// Watchdog Registers
#define WDT_OMR      (*((volatile unsigned long *) 0xFFE14068))

// Interleave Program Flash (IFC) Registers
#define IFC_PMSR    (*((volatile unsigned long *) 0xFFE04058))
#define IFC_CR      (*((volatile unsigned long *) 0xFFE04060))
#define IFC_MR      (*((volatile unsigned long *) 0xFFE04064))
#define IFC_CSR     (*((volatile unsigned long *) 0xFFE0406C))
#define IFC_SR      (*((volatile unsigned long *) 0xFFE04070))
#define IFC_IER     (*((volatile unsigned long *) 0xFFE04074))
#define IFC_IDR     (*((volatile unsigned long *) 0xFFE04078))
#define IFC_IMR     (*((volatile unsigned long *) 0xFFE0407C))

// Bit Constants
#define CE          (1 << 2)
#define SE          (1 << 1)
#define WPR         (1 << 7)
#define STANDEN    (1 << 4)
#define SPEEDMODE  (1 << 2)
#define DACCESS    (1 << 2)
#define ENDERASE   (1 << 1)
#define ENDWR      (1 << 0)
#define BUSY       (1 << 8)
```

Flash Init

```

UINT32 RVDFlash_Init(UINT32 base_of_flash)
{
    // Set unprotected mode
    IFC_MR = (0xAC << 8);
    IFC_CSR = (ENDWR | ENDERASE | DACCESS | BUSY);

    // disable watchdog
    WDT_OMR = (0x234 << 4);    // Disable watchdog
    return 0;
}
    
```

Flash erase

```

UINT32 RVDFLASH_Erase(UINT32* base_address, UINT32 block_count, UINT32 block_size)
{
    // Clear ENDERASE
    IFC_CSR = ENDERASE;
    // Find sector and erase sector
    IFC_CR = ((0x37 << 8) | SECTOR_NUM((U32_T) base_address) | IFC_SE);
    // wait
    while (!(IFC_SR & ENDERASE));
    return 0;
}
    
```

Flash write

```

UINT32 RVDFLASH_Write(UINT32* base_address,
                      UINT32 byte_count,
                      UINT32 block_offset,
                      UINT32* from_buffer, UINT32 verify)
{
    U32_T    i;

    UINT32 *addrFlash;    // flash block address
    UINT32 *addrData;

    // get address
    addrFlash = (UINT32 *) base_address; // Flash
    addrData = (UINT32 *) from_buffer; // SRAM buffer

    // flash and verify
    for (i = 0; i < (byte_count >> 2); i++) {
        IFC_CSR = ENDWR;
        addrFlash[i + (block_offset >> 2)] = addrData[i];
        while (!(IFC_SR & ENDWR));
        //if (addrFlash[i + (block_offset >> 2)] != addrData[i]) return 1;
    }

    return 0;
}
    
```

Flash Validate

```

UINT32 RVDFLASH_Validate(UINT32 base_address, UINT32 byte_count, UINT32 block_offset,
                          UINT32 from_buffer)
{
    U32_T    i;

    UINT32 *addrFlash;    // flash block address
    UINT32 *addrData;

    // get address
    addrFlash = (UINT32 *) base_address; // Flash
    
```

```

addrData = (UINT32 *) from_buffer; // SRAM buffer

// flash and verify
for (i = 0; i < (byte_count >> 2); i++) {
    if (addrFlash[i + (block_offset >> 2)] != addrData[i]) return 1;
}

return 0;
}
    
```

Since AGP shall write in WORD (4bytes) unit, byte_count and block_offset are divided by four. It is possible to migrate the verification routine into the write routine.

Next is a makefile.

```

TARGET      = ARM
INC_PATHS   =

# CFLAGS is the default flags rule. Others are added below
CFLAGS      = -g $(INC_PATHS)
CFLAGS_SRCS =
CFLAGS_COMP = $(CC) -c $(CFLAGS)
CFLAGS_OBJS =

CARM        = -g --dwarf2
CARM_SRCS   =          source\s3f4a2f_flash.c
CARM_COMP   =          armcc -c $$SRC $(CARM) -o $$OBJ
CARM_OBJS   =          build\s3f4a2f_flash.o

CARM_CPP    = -g --dwarf2 --cpp
CARM_CPP_SRCS =
CARM_CPP_COMP =          armcc -c $$SRC $(CARM_CPP) -o $$OBJ
CARM_CPP_OBJS =

CTHUMB      = --thumb -g --dwarf2
CTHUMB_SRCS =
CTHUMB_COMP =          armcc -c $$SRC $(CTHUMB) -o $$OBJ
CTHUMB_OBJS =

CTHUMB_CPP  = --thumb -g --dwarf2 --cpp
CTHUMB_CPP_SRCS =
CTHUMB_CPP_COMP =          armcc -c $$SRC $(CTHUMB_CPP) -o $$OBJ
CTHUMB_CPP_OBJS =

AARM        = --li -g --dwarf2 --cpu ARM7TDMI-S
AARM_SRCS   =          source\b_s3f4a2f.s
AARM_COMP   =          armasm $(AARM) -o $$OBJ $$SRC
AARM_OBJS   =          build\b_s3f4a2f.o

#FLAGS end here (put all flag groups above this line)

EXTHDRS     =
HDRS        =
LDFLAGS     = --noremove --ro-base 0x300000 --rw-base 0x301000 --entry 0x300000

# LIB_DEP is list of dependents for library build ($OBJ for example)
LIB_DEP     =
# LIB_ARGS is argument line for librarian
LIB_ARGS    =

# LIBS are system libraries as name
LIBS        =
    
```

```

CC           = cl$(TARGET)
LINKER      = armlink
LIBRARIAN   =
MAKENAME    = makefile
PRINT       = pr
PAKFLASH    = pakflash
FROMELF     = fromelf
BUILD_LIB   = none.lib
POST_BUILD  =

# Project
PRJNAME     = S3F4A2F_FLASH
PROGRAM     = $(PRJNAME).axf
DISASM      = $(PRJNAME).dis

ALL_DEP     = $(PROGRAM) $(POST_BUILD)

OBJS        = $(CFLAGS_OBJS) $(CARM_OBJS) $(CARM_CPP_OBJS) $(CTHUMB_OBJS) \
              $(CTHUMB_CPP_OBJS) $(AARM_OBJS)
SRCS        = $(CFLAGS_SRCS) $(CARM_SRCS) $(CARM_CPP_SRCS) $(CTHUMB_SRCS) \
              $(CTHUMB_CPP_SRCS) $(AARM_SRCS)

# depend is for dependant rules that should not be linked in
DEPEND=
O_DEPEND    =

COMP_LINE   = $(CFLAGS_COMP)

# .c.o is default compile line if not explicitly defined
.c.o :
    $(COMP_LINE) *.c

# all is first target
all:        $(ALL_DEP)
            @+echo --- Build-all done ---

#LINK: next one is the link command
$(PROGRAM): $(OBJS) $(DEPEND) $(O_DEPEND) $(MAKENAME)
            $(LINKER) $(LDFLAGS) $(OBJS) $(LIBS) -o $(PROGRAM)
            $(FROMELF) --text -c $(PROGRAM) > $(DISASM)
            $(PAKFLASH) -d -f S3F4A2F $(PROGRAM) source\board_s3f4a2f.ame -o
            $(PRJNAME).fme
#BUILD_LIB: this is only valid if filled in at macro point
$(BUILD_LIB): $(LIB_DEP) $(OBJS) $(DEPEND) $(O_DEPEND) $(MAKENAME)
              $(LIBRARIAN) $(LIB_ARGS) $(BUILD_LIB) $(OBJS)

clean:
    rm -rf $(OBJS) $(TARGET) $(DISASM) $(BUILD_LIB)
rebuild:clean all
depend:;    updep $(MAKENAME)
index:;    ctags -wx $(HDRS) $(SRCS)
print:;    $(PRINT) $(HDRS) $(SRCS)
program:   $(PROGRAM)
tags:     $(HDRS) $(SRCS); ctags $(HDRS) $(SRCS)

# DEPEND: include files below
# Dependency summaries:
# System includes suppressed
# DEPEND: end include files
    
```

System LSI Division, Semiconductor Business

```

build\s3f4a2f_flash.o: source\s3f4a2f_flash.c
    armcc -c source\s3f4a2f_flash.c \
        $(CARM) -o build\s3f4a2f_flash.o
build\b_s3f4a2f.o: source\b_s3f4a2f.s
    armasm $(AARM) -o \
        build\b_s3f4a2f.o source\b_s3f4a2f.s
    
```

LDFLAGS requires special care. Since FME file is executed in RAM area, ro-base and entry shall be defined in RAM section, which is 0x300000 for AGP series. (see LDFLAGS)

Next is to create AME file. In AME file, the properties of flash memory are defined.

```

[BOARD=S3F4A2F]
proc_name=ARM
# uses S3F4A2F BOOT BLOCK
from_flash=S3F4A2F_01
width=4
reloc.pc_rel=True
reloc.start_addr=0x0

[FLASH=S3F4A2F_01]
flash_name="SAMSUNG S3F4A2F FLASH"
no_erase = False
# This Flash has 16 16K blocks = 256K
block.1={count=0x10:size=0x4000}
    
```

In the AME file, a board is first defined and flash is defined later. In width, the size of accessible unit is defined. A Word is for AGP. AGP3 (in this example) has 16 blocks (sectors) with 16kbytes of one block (sector).

Finally, the project structure looks like following tree.

```

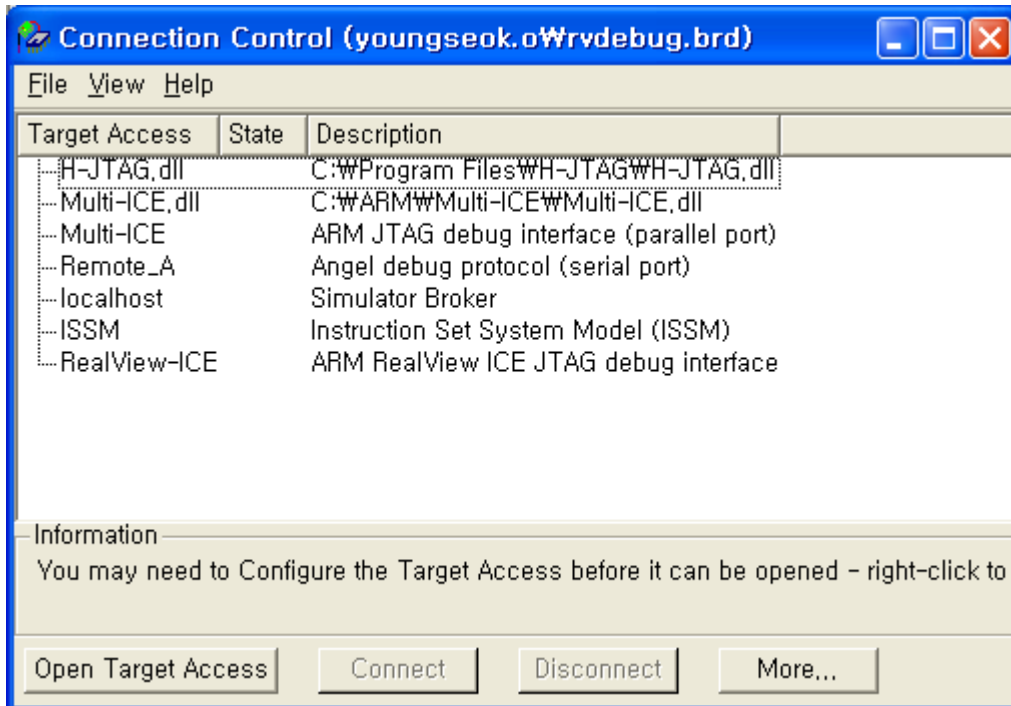
flash_downloader_agp3
├── build
└── source
    
```

In the root, makefile and pakflash are located, and FME file is created. In the source folder, AME and flash algorithm are located. In the build, object codes are created.

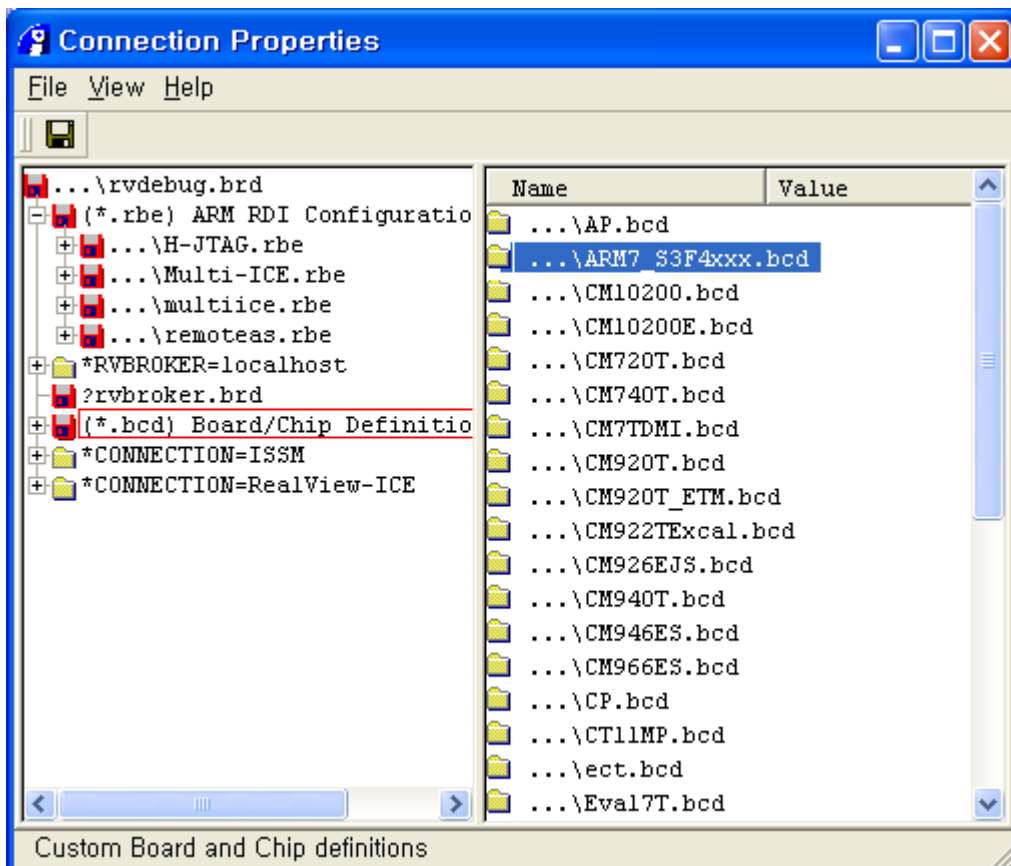
4. Creating BCD File

BCD file shall be created in RVD. After executing RVD,

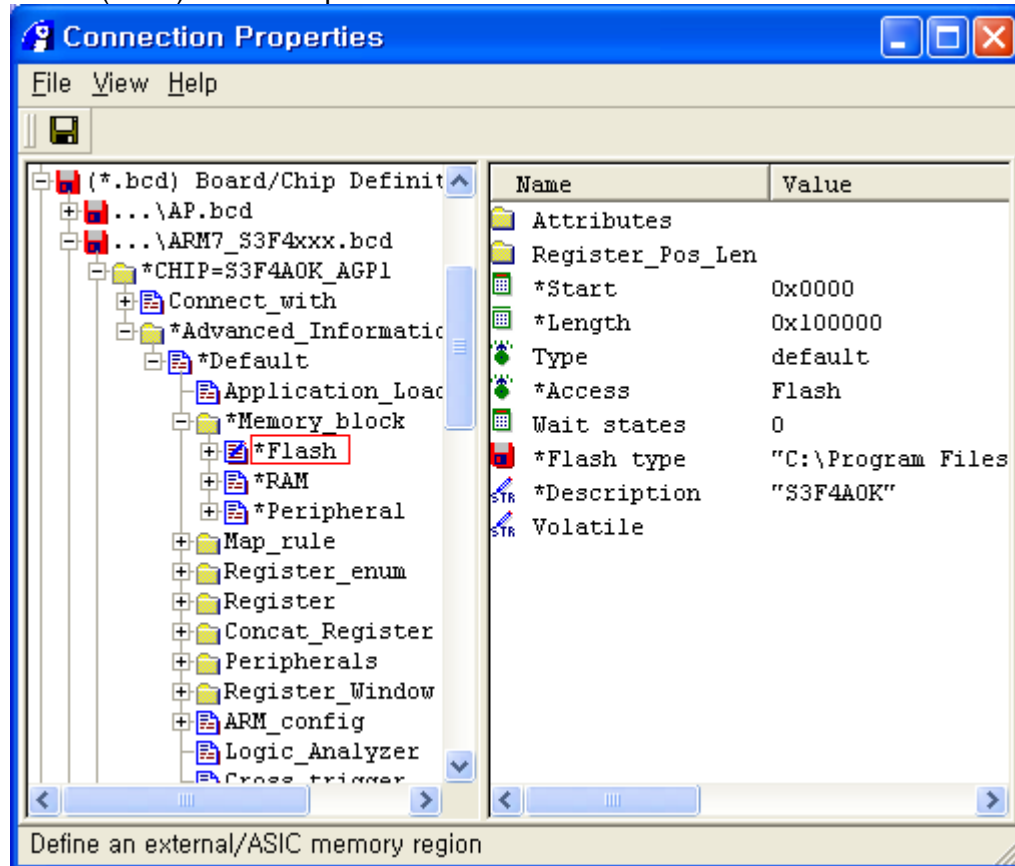
Select Target>connection to target



Select File>properties



Select (*.bcd) Board/Chip Definition



By creating new group step by step, users are allowed to create *ARM7_S3F4xxx.bcd* and *CHIP=S3F4A0K_AGP1*.

Exploring the tree down to *Memory_block*, users can define *Start*, *Length*, *Access*, *Flash type*, and *Description*.

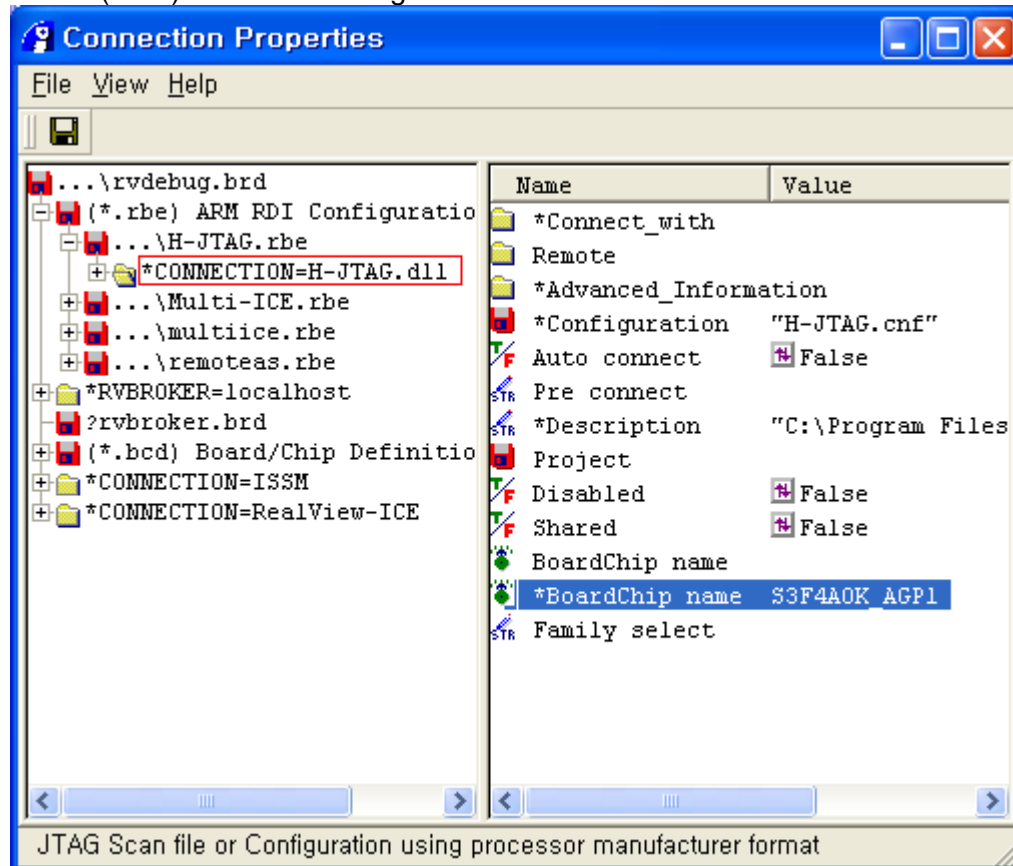
- Start: Flash memory start address
- Length: Entire size (e.g. 1Mbytes for AGP1)
- Access: Flash memory
- Flash type: designate the location of FME file
- Description: Chip name

In this way, RAM can be declared, too.

Note: When the values are customized, "*" appears at the beginning of each of strings.

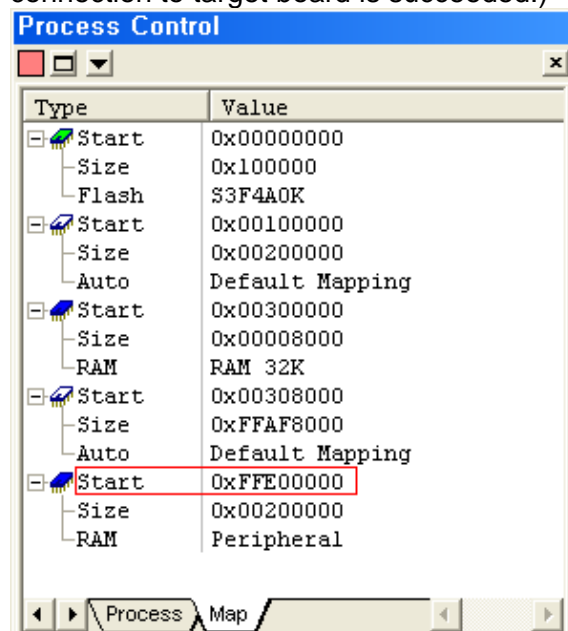
After creating BCD, users shall make a decision of *BoardChip* name that will be used for RVD.

Select (*.rbe) ARM RDI Configuration



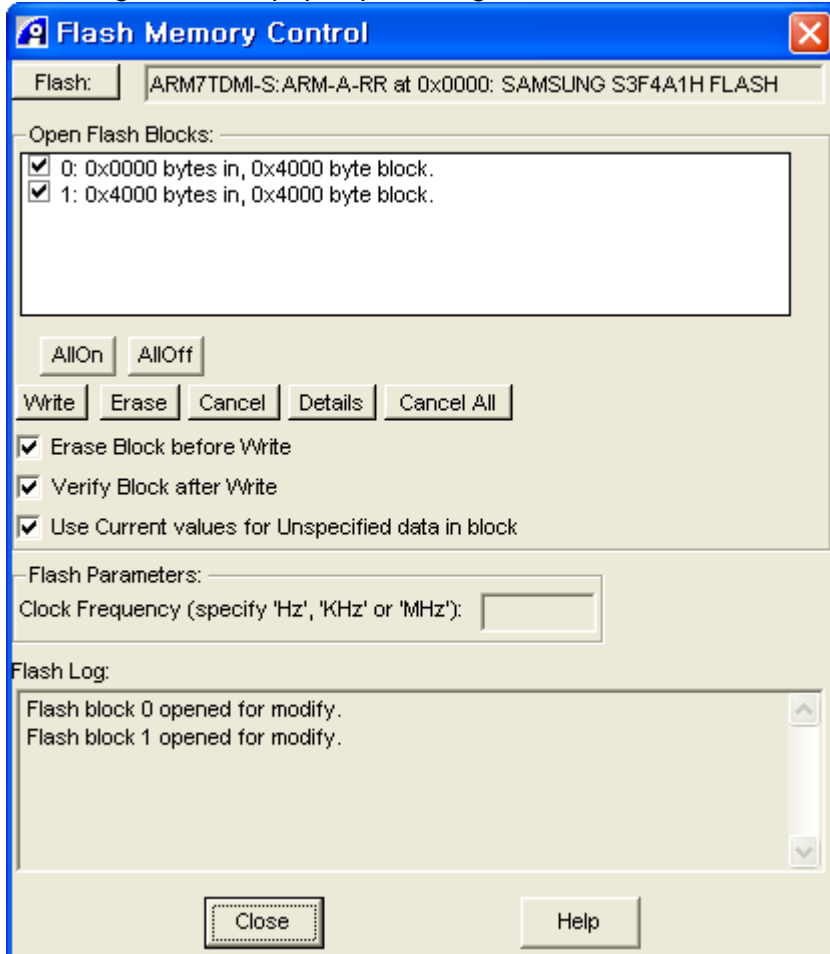
RDI connection shows currently available connections (e.g. H-JTAG, Multi-ICE, etc.) By clicking *BoardChip* name, a list of available BCD will be listed. Then, select S3F4A0K_AGP1 which was just created.

When it is completed, users are able to see the memory section as below (when the connection to target board is succeeded.)



5. Programming Flash

Choosing an axf file pops up a dialog box as below.



Users can command to write or erase in this window. When erase is done, following messages appear in the log:

```

Loading Flash routine...
Initializing Flash routine...
Beginning Flash programming...
Erasing block 0 at address 0x0000
Erasing block 1 at address 0x4000
Flash programming complete.
    
```

When write is performed, following messages appear.

```

Loading Flash routine...
Initializing Flash routine...
Beginning Flash programming...
Erasing block 0 at address 0x0000
Writing to Flash block 0...
Erasing block 1 at address 0x4000
Writing to Flash block 1...
Flash programming complete.
Opened Flash closed (last block closed).
    
```

Note that this programming method is often unstable, so it produces following messages.

Error V003D (Vehicle): Failed writing cache to Flash Error: Failed downloading flash routine to memory (verify failed)

In this case, repeat the write action again.

6. References

ARM website: <http://www.arm.com>

Application note 110: Flash programming with RealView Debugger

RVD Target Configuration Guide