AT91 Assembler Code Startup Sequence for C Code Applications Software

Introduction

For reasons of modularity and portability most application code for the AT91 ARM-based microcontrollers is written in C. However, the startup sequence required to initialize the ARM processor and certain key peripherals is heavily dependent on the register architecture and memory mapping processor, and the memory remap operation. For this reason the C startup sequence is written in assembler.

This Application Note describes an example of the AT91 C startup sequence. It is based on the C startup sequence for the AT91 Evaluation Board working with the ARM ADS V1.1 Development Tools. Further examples of C startup sequences are available in the AT91 Library. The C startup sequence is activated on power-up and after a reset.

C-Startup Sequence

A major consideration in the design of an embedded ARM application is the layout of the memory map, in particular the memory that is situated at address 0x0. Following reset, the processor starts to fetch instructions from 0x0, therefore there must be some executable code accessible from that address. In an embedded system, this requires NVM to be present, at least initially, at address 0x0.

The simplest layout is accomplished by locating the application in ROM at address 0 in the memory map. The application can then branch to the real entry point when it executes its first instruction at the reset vector at address 0x0. But, there are disadvantages with this layout. ROM is typically narrow (8 or 16 bits) and slow compared to RAM, requiring more wait states to access it. This slows down the handling of processor exceptions, especially interrupts, through the vector table. Moreover, if the vector table is in ROM, it cannot be modified by the code.

Since RAM is normally faster and wider than ROM, it is better for the vector table and interrupt handlers if the memory at 0x0 is RAM. Although It is necessary that RAM be located at 0x0 during normal execution, if RAM is located at address 0x0 on power-up, there is not a valid instruction in the reset vector entry. Therefore, ROM must be located at 0x0 at power-up to assure that there is a valid reset vector. The changeover from reset to the normal memory map is normally accomplished by performing the remap command.

Many applications written for ARM-based systems are embedded applications that are contained in ROM and execute on reset. There are a number of factors that must be considered when writing embedded operating systems, or embedded applications that execute from reset without an operating system, including:

- Remapping ROM to RAM, to improve execution speed.
- Initializing the execution environment, such as exception vectors, stacks, I/Os.
- Initializing the application.
  - For example, copying initialization values for initialized variables from ROM to RAM and resetting all other variables to zero.
- Linking an embedded executable image to place code and data in specific locations in memory.

For an embedded application without an operating system, the code in ROM must provide a way for the application to initialize itself and start executing. No automatic initialization takes place on reset, therefore the application entry point must perform some initialization before it can call any C code.

The initialization code, located at address zero after reset, must:

- Mark the entry point for the initialization code.
- Set up exception vectors.
- Initialize the memory system.
- Initialize the stack pointer registers.
- Initialize any critical I/O devices.
- Initialize any RAM variables required by the interrupt system.
- Enable interrupts (if handled by the initialization code).
- Change processor mode if necessary.
- Change processor state if necessary.

After the environment has been initialized, the sequence continues with the application initialization and should enter the C code.

AT91 ARM Thumb

The C-startup file is the first file executed at power on and performs initialization of the microcontroller from the reset vector up to the calling of the application main routine. The main program should be a closed loop and should not return.

The ARM core begins executing instructions from address 0 at reset. For an embedded system, this means that there must be ROM at address 0 when the system is reset. Because of ROM limitations, the speed of exception handling is affected and the exception vectors cannot be modified. A common strategy is to remap ROM to RAM and copy the exception vectors from ROM to RAM after startup.

C - Startup Example

A generic start-up file is included within this Application Note and others are available in the AT91 software library. The example described is based on the AT91 Evaluation Board C-startup sequence working with ARM ADS V1.1 Development Tool and debugging in external Flash Memory. This file must be modified in order to fit the needs of the user application.

Each AT91 Evaluation Board is described in the AT91 Library inside a subdirectory of the directory\software\targets. Each of these subdirectories contains the following files:

- The <target>.h file, defines the components of the boards in C.
- The <target>.inc file, defines the components of the boards in assembly.
- One or more cstartup.s files, define standard boot for the boards according to the software development tools used: ARM SDT, ARM ADS and Green Hills MULTI® 2000 for example.

The AT91 Library provides C-Startup files that explain how to boot an AT91 part and how to branch on the main C function. The C-Startup file provides an example of how to boot an AT91 part, taking into account the part specific features, the board specific characteristics and the debug level required.

Note: The software example is delivered "As Is" without warranty or condition of any kind, either express, implied or statutory. This includes without limitation any warranty or condition with respect to merchantability or fitness for any particular purpose, or against the infringements of intellectual property rights of others.

Area Definition and Entry Point for the Initialization Code

In an ARM assembly language source file, the start of a section is marked by the AREA directive. This directive names the section and sets its attributes. The attributes are placed after the name, separated by commas. The code example referenced above defines a read-only code section named reset.

An executable image must have an entry point. An embedded image that can be placed in ROM usually has an entry point at 0x0. An entry point can be defined in the initialization code by using the assembler directive ENTRY.

```
;-------------------------------------------------------------------------
;- Area Definition
;-------------------------------------------------------------------------
AREA                reset, CODE, READONLY
;-------------------------------------------------------------------------
;- Define the entry point
;-------------------------------------------------------------------------
ENTRY
```

Setup Exception Vectors

Exception Vectors are setup sequentially through the address space with branches to nearby labels or branches and links to subroutines. During the normal flow of execution through a program, the program counter increases enable the processor to handle events generated by internal or external sources. Processor exceptions occur when the normal flow of execution is diverted. Examples of such events are:

- Externally generated interrupts.
- An attempt by the processor to execute an Undefined Instruction.

It is necessary to preserve the previous processor status when handling such exceptions, so that execution of the program that was running when the exception occurred can resume when the appropriate exception routine has completed.

The initialization code must set up the required exception vectors (see Table 1). If the ROM is located at address 0, the vectors consist of a sequence of hard-coded instructions to branch to the handler for each exception. These vectors are mapped at address 0 before remap. They must be in relative addressing mode in order to guarantee a valid jump. After remap, these vectors are mapped at address 0x01000000 and can only be changed back by an internal reset or NRST assertion.

Table 1. Exception Vectors

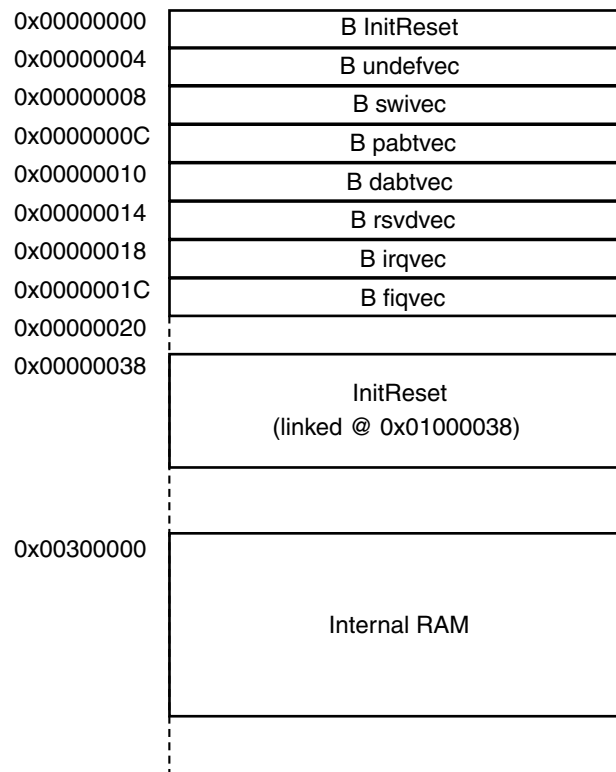| Exception	Description
Reset	Occurs when the processor reset pin is asserted. This exception is only expected to occur for signalling power-up, or for resetting as if the processor has just powered up. A soft reset can be done by branching to the reset vector (0x0000).
Undefined Instruction	Occurs if neither the processor, or any attached coprocessor, recognizes the currently executing instruction.
Software Interrupt (SWI)	This is a user-defined synchronous interrupt instruction. It allows a program running in User mode, for example, to request privileged operations that run in Supervisor mode, such as an RTOS function.
Prefetch Abort	Occurs when the processor attempts to execute an instruction that has prefetched from an illegal address.
Data Abort	Occurs when a data transfer instruction attempts to load or store data at an illegal address.
IRQ	Occurs when the processor external interrupt request pin is asserted (LOW) and the I bit in the CPSR is clear.
FIQ	Occurs when the processor external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear.

Processor exception handling is controlled by a vector table. The vector table is a reserved area of 32 bytes, usually at the bottom of the memory map. It has one word of space allocated to each exception type, and one word that is currently reserved as shown in Figure 1. Because there is not enough space to contain the full code for a handler, the vector entry for each exception type contains a branch instruction or load pc instruction to continue execution with the appropriate handler.

```
;-------------------------------------------------------------------------
;- Exception vectors ( before Remap )
;-------------------------------------------------------------------------
  B     InitReset  ; reset
undefvec
  B     undefvec   ; Undefined Instruction
swivec
  B     swivec     ; Software Interrupt
pabtvec
  B     pabtvec    ; Prefetch Abort
dabtvec
  B     dabtvec    ; Data Abort
rsvdvec
  B     rsvdvec    ; reserved
irqvec
  B     irqvec     ; reserved
fiqvec
  B     fiqvec     ; reserved
```

Figure 1. Exception Vectors Mapping

Address	Vector
0x00000000	B InitReset
0x00000004	B undefvec
0x00000008	B swivec
0x0000000C	B pabtvec
0x00000010	B dabtvec
0x00000014	B rsvdvec
0x00000018	B irqvec
0x0000001C	B fiqvec
0x00000020	
0x00000038	InitReset (linked @ 0x01000038)
0x00300000	Internal RAM

External Bus Interface Initialization Table

The EBI Table is used to configure the memory controller. The EBI values depend on target, clock and external memory access time. These values are defined in an "include file" corresponding to the target, for example eb55.inc for AT91EB55 Evaluation Board.

```
;-------------------------------------------------------------------------------
;- EBI Initialization Data
;-------------------------------------------------------------------------------
InitTableEBI
  DCD           EBI_CSR_0
  DCD           EBI_CSR_1
  DCD           EBI_CSR_2
  DCD           EBI_CSR_3
  DCD           EBI_CSR_4
  DCD           EBI_CSR_5
  DCD           EBI_CSR_6
  DCD           EBI_CSR_7
  DCD           0x00000001    ; REMAP command
  DCD           0x00000006    ; 6 memory regions, standard read
PtEBIBase
  DCDEBI_BASE; EBI Base Address
```

Reset Handler

From here, the code is executed from address 0. Caution should be taken, as it is linked to 0x100 0000.

```
;-------------------------------------------------------------------------------
;- Reset Handler before Remap
;-------------------------------------------------------------------------------
InitReset
```

Speed Up the Boot Sequence

After reset, the External Bus Interface is not configured apart from the chip select 0 and the number of wait states on chip select 0 is 8. Before the remap command, the chip select 0 configuration can be modified by programming the EBI_CSR0 with exact boot memory characteristics. The base address becomes effective after the remap command, but the new number of wait states can be changed immediately. This is desirable if a boot sequence needs to be faster.

```
;-------------------------------------------------------------------------------
;- Speed up the Boot sequence
;-------------------------------------------------------------------------------
;- Load System EBI Base address and CSR0 Init Value
  ldr       r0,           PtEBIBase
  ldr       r1,           InitTableEBI     ; values (relative)
;- Speed up code execution by disabling wait state on Chip Select 0
  str       r1,       [r0]
```

Low Level Initialization

Peripherals that must be initialized before enabling interrupts should be considered as critical. If these peripherals are not initialized at this point, they might cause spurious interrupts when interrupts are enabled.

```
;-------------------------------------------------------------------------
;- Low level init
;-------------------------------------------------------------------------
        bl      __low_level_init
```

Example: Start PLL on the AT91EB55 Evaluation Board.

At reset, the AT91M55800A microcontroller starts with the slow clock oscillator (32.768 kHz) to minimize the power required to start up the system and the main oscillator is disabled. The PLL can be started by configuring the Advanced Power Management Controller to run with the main oscillator to speed up the startup sequence.

The __low_level_init function is defined in the assembly file from the AT91 software library associated to the corresponding evaluation board.

Advanced Interrupt Controller Configuration

After reset, the Advanced Interrupt Controller (AIC) is not configured. The C-startup file initializes the AIC by setting up the default interrupt vectors. The default Interrupt handler functions are defined in the AT91 library. These functions can be redefined in the application code. To view interrupt default handler initialization, see Figure 2 on page 9.
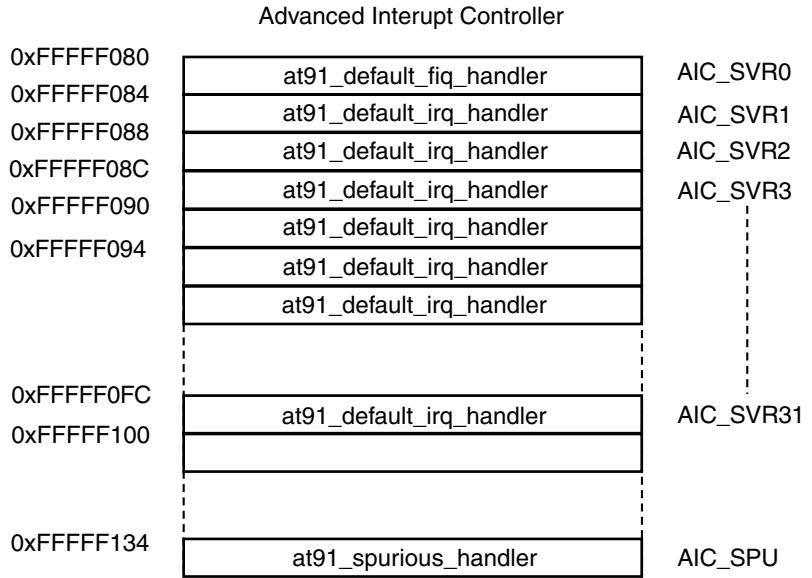
```
;-----------------------------------------------------------------------------
;- Advanced Interrupt Controller configuration
;-----------------------------------------------------------------------------
;- Set up the default vectors
;-----------------------------------------------------------------------------
;- Load the AIC Base Address and the default handler addresses
  add      r0, pc,#-(8+.-AicData)     ; @ where to read values (relative)
  ldmia    r0, {r1-r4}
;- Setup the Spurious Vector
  str      r4, [r1, #AIC_SPU]         ; r4 =spurious handler
;- Set up the default interrupt handler vectors
  str      r2, [r1, #AIC_SVR]; SVR[0] for FIQ
  add      r1, r1, #AIC_SVR
  mov      r0, #31                    ; counter
LoopAic1
  str      r3, [r1, r0, LSL #2]; SVRs for IRQs
  subs     r0, r0, #1                 ; do not save FIQ
  bhi      LoopAic1
  b        EndInitAic


;-----------------------------------------------------------------------------
;- Default Interrupt Handler
;-----------------------------------------------------------------------------
AicData
  DCD       AIC_BASE; AIC Base Address

  IMPORT    at91_default_fiq_handler
  IMPORT    at91_default_irq_handler
  IMPORT    at91_spurious_handler
PtDefaultHandler
  DCD       at91_default_fiq_handler
  DCD       at91_default_irq_handler
  DCD       at91_spurious_handler
EndInitAic
```

Figure 2. Interrupt Default Handler Initialization

Advanced Interupt Controller

Address	Handler	Register
0xFFFFF080	at91_default_fiq_handler	AIC_SVR0
0xFFFFF084	at91_default_irq_handler	AIC_SVR1
0xFFFFF088	at91_default_irq_handler	AIC_SVR2
0xFFFFF08C	at91_default_irq_handler	AIC_SVR3
0xFFFFF090	at91_default_irq_handler	
0xFFFFF094	at91_default_irq_handler	
	at91_default_irq_handler	
0xFFFFF0FC	at91_default_irq_handler	AIC_SVR31
0xFFFFF100		
0xFFFFF134	at91_spurious_handler	AIC_SPU

Copy Exception Vectors in Internal RAM

The exception vectors must be copied into internal RAM. It is important to perform this operation before remap in order to guarantee that the core is provided valid vectors during the remap operation. There are only five offsets as the vectoring is used. See Figure 3 on page 11

```
;-------------------------------------------------------------------------
;-Setup Exception Vectors in Internal RAM before Remap
;-------------------------------------------------------------------------
  b        SetupRamVectors
VectorTable
  ldr      pc, [pc, #&18]           ; SoftReset
  ldr      pc, [pc, #&18]           ; UndefHandler
  ldr      pc, [pc, #&18]           ; SWIHandler
  ldr      pc, [pc, #&18]           ; PrefetchAbortHandler
  ldr      pc, [pc, #&18]           ; DataAbortHandler
  nop                               ; Reserved
  ldr      pc, [pc,#-0xF20]         ; IRQ : read the AIC
  ldr      pc, [pc,#-0xF20]         ; FIQ : read the AIC
;- There are only 5 offsets as the vectoring is used.
  DCD      SoftReset
  DCD      UndefHandler
  DCD      SWIHandler
  DCD      PrefetchAbortHandler
  DCD      DataAbortHandler
;- Vectoring Execution function run at absolute address
SoftReset
  b        SoftReset
UndefHandler
  b        UndefHandler
SWIHandler
  b        SWIHandler
PrefetchAbortHandler
  b        PrefetchAbortHandler
DataAbortHandler
  b        DataAbortHandler

SetupRamVectors
  mov      r8, #RAM_BASE_BOOT       ; @ of the hard vector in RAM 0x300000
  add      r9, pc,#-(8+.-VectorTable) ; @ where to read values (relative)
  ldmia    r9!, {r0-r7}             ; read 8 vectors
  stmia    r8!, {r0-r7}             ; store them on RAM
  ldmia    r9!, {r0-r4}             ; read 5 absolute handler addresses
  stmia    r8!, {r0-r4}             ; store them on RAM
```

Figure 3. Map to Copy Exception Vectors in Internal RAM
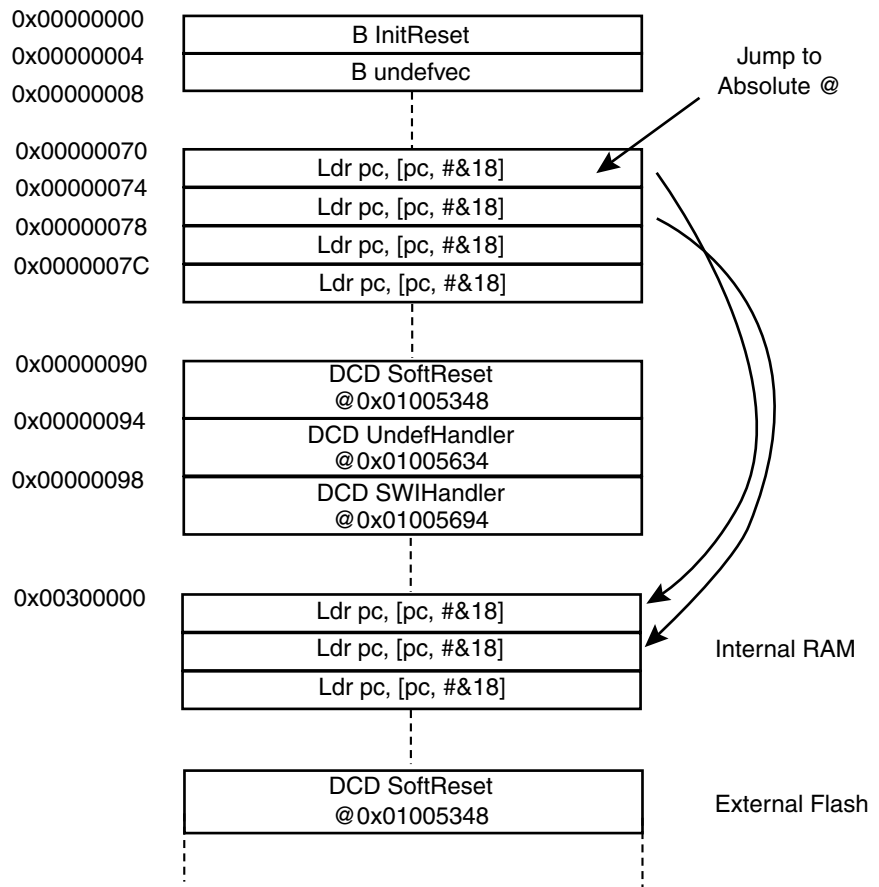
Memory Controller Initialization and Remap Command

After a reset the internal RAM of the AT91 is mapped at address 0x00300000. The memory connected to the Chip Select line 0 is mapped at address 0. When the Remap command is performed, the external memory is mapped at the address defined by the user in the Chip Select Register 0. The Internal RAM is then mapped at address 0 allowing ARM7TDMI exception vectors between 0x0 and 0x20 to be modified by the software.

```
;------------------------------------------------------------------------------
;- Memory Controller Initialization
;------------------------------------------------------------------------------
;- Copy the Image of the Memory Controller
    sub    r10, pc,#(8+.-InitTableEBI)  ; get the address of the chip
                                        ; select register image
    ldr    r12, PtInitRemap             ; get the real jump address
                                        ; ( after remap )
;- Copy Chip Select Register Image to Memory Controller and command remap
    ldmia  r10!, {r0-r9,r11}            ; load the complete image and the
                                        ; EBI base
    stmia  r11!, {r0-r9}                ; store the complete image with
                                        ; the remap command
```

```
;- Jump to ROM at its new address
  mov     pc, r12                        ; jump and break the pipeline


PtInitRemap
  DCD     InitRemap                      ; address where to jump after REMAP
;-----------------------------------------------------------------------
;- From here, the code is executed from its link address, ie. 0x100 0000.
;-----------------------------------------------------------------------
InitRemap
```

The ARM Processor Pipeline assures that the instruction "mov pc, r12" will be read before remap is executed. After remap, the next instruction is fetched in internal RAM and the instruction "mov pc, r12" is executed to perform jump in ROM at the previously loaded address in r12 (0x0100011C) thus simultaneously breaking the pipeline as shown in Figure 4. The subsequent "new" memory mapping after remap is described in Figure 5 on page 13.

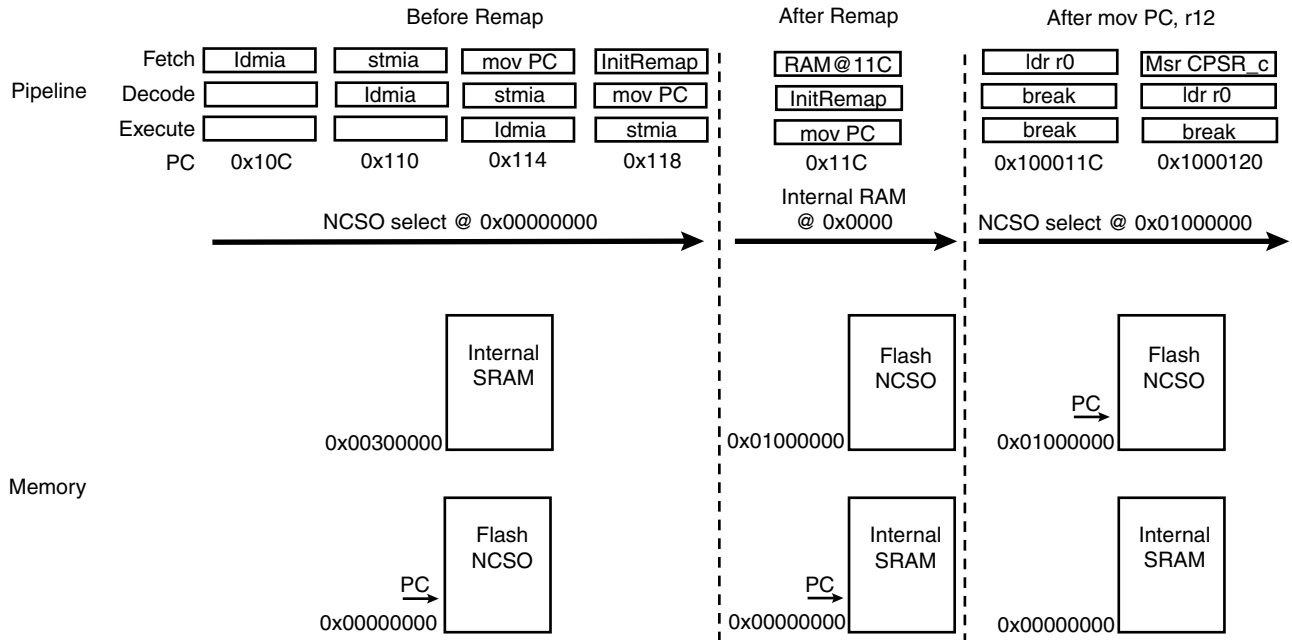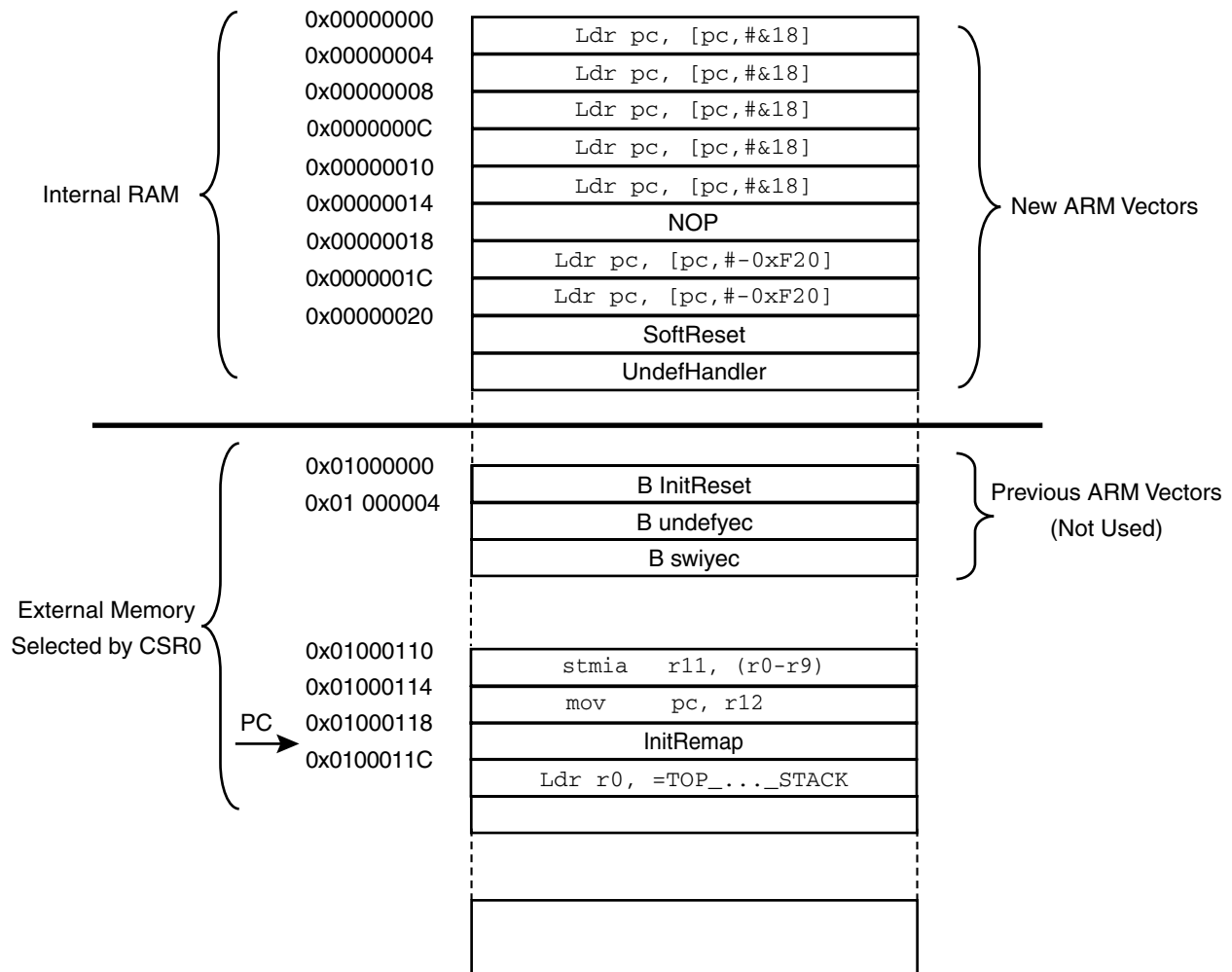**Figure 4.** ARM Core Pipeline During Remap Command

Figure 5. Memory Map After Remap

Internal RAM

Address	
0x00000000	Ldr pc, [pc,#&18]
0x00000004	Ldr pc, [pc,#&18]
0x00000008	Ldr pc, [pc,#&18]
0x0000000C	Ldr pc, [pc,#&18]
0x00000010	Ldr pc, [pc,#&18]
0x00000014	NOP
0x00000018	Ldr pc, [pc,#-0xF20]
0x0000001C	Ldr pc, [pc,#-0xF20]
0x00000020	SoftReset
	UndefHandler

New ARM Vectors

External Memory Selected by CSR0

Address	
0x01000000	B InitReset
0x01 000004	B undefyec
	B swiyec
0x01000110	stmia r11, (r0-r9)
0x01000114	mov pc, r12
PC → 0x01000118	InitRemap
0x0100011C	Ldr r0, =TOP_...._STACK

Previous ARM Vectors (Not Used)

Initialize Stack Registers

Fast Interrupt, Interrupt, Abort, Undefined and Supervisor Stack are located at the top of internal memory in order to speed up the exception handling context. User (Application, C) Stack is located at the top of the external memory.

The initialization code initializes the stack pointer registers. Depending on the interrupts and exceptions desired, some or all of the following stack pointers may require initialization:

- Supervisor stack must always be initialized.
- IRQ stack must be initialized if IRQ interrupts are used. It must be initialized before interrupts are enabled.
- FIQ stack must be initialized if FIQ interrupts are used. It must be initialized before interrupts are enabled.
- Abort-status stack must be initialized for Data and Prefetch Abort handling.
- Undefined Instruction stack must be initialized for Undefined Instruction handling.

Generally, Abort-status and Undefined Instruction stacks are not used in a simple embedded system. However, it might be preferable to initialize them for debugging purposes.

The user stack pointer can be set up when changing to User mode to start executing the application.

```
;--------------------------------------------------------------------------------
;- Stack sizes definition
;--------------------------------------------------------------------------------
IRQ_STACK_SIZE          EQU     (3*8*4)         ; 3 words
FIQ_STACK_SIZE          EQU     (3*4)           ; 3 words
ABT_STACK_SIZE          EQU     (1*4)           ; 1 word
UND_STACK_SIZE          EQU     (1*4)           ; 1 word
```

Assuming that the IRQ_ENTRY/IRQ_EXIT macro is used, Interrupt Stack requires 3 words x 8 priority level x 4 bytes when using the vectoring. The Interrupt Stack must be adjusted depending on the interrupt handlers. Fast Interrupt requires 3 words x 4 bytes without priority level. Other stacks are defined by default to save one word each. The system stack size is not defined and is limited by the free internal SRAM. User stack size is not defined and is limited by the free external SRAM.

```
;--------------------------------------------------------------------------------
;- Top of Stack Definition
;--------------------------------------------------------------------------------
TOP_EXCEPTION_STACK     EQU     RAM_LIMIT       ; Defined in part
TOP_APPLICATION_STACK   EQU     EXT_SRAM_LIMIT  ; Defined in Target


;--------------------------------------------------------------------------------
;- Setup stack for each mode
;--------------------------------------------------------------------------------
  ldr                   r0, =TOP_EXCEPTION_STACK
;- Set up Fast Interrupt Mode and set FIQ Mode Stack
  msr                   CPSR_c, #ARM_MODE_FIQ:OR:I_BIT:OR:F_BIT
  mov                   r13, r0                 ; Init stack FIQ
  sub                   r0, r0, #FIQ_STACK_SIZE
;- Set up Interrupt Mode and set IRQ Mode Stack
  msr                   CPSR_c, #ARM_MODE_IRQ:OR:I_BIT:OR:F_BIT
  mov                   r13, r0                 ; Init stack IRQ
  sub                   r0, r0, #IRQ_STACK_SIZE
;- Set up Abort Mode and set Abort Mode Stack
  msr                   CPSR_c, #ARM_MODE_ABORT:OR:I_BIT:OR:F_BIT
  mov                   r13, r0                 ; Init stack Abort
  sub                   r0, r0, #ABT_STACK_SIZE
;- Set up Undefined Instruction Mode and set Undef Mode Stack
  msr                   CPSR_c, #ARM_MODE_UNDEF:OR:I_BIT:OR:F_BIT
  mov                   r13, r0                 ; Init stack Undef
  sub r0, r0, #UND_STACK_SIZE
;- Set up Supervisor Mode and set Supervisor Mode Stack
  msr                   CPSR_c, #ARM_MODE_SVC:OR:I_BIT:OR:F_BIT
  mov                   r13, r0                 ; Init stack Sup
```

Change Processor Mode and Enable Interrupts

The initialization code can now enable interrupts if necessary, by clearing the interrupt disable bits in the CPSR. This is the earliest point that it is safe to enable interrupts. At this stage the processor is still in Supervisor mode. If the application runs in User mode, change to User mode and initialize the User mode stack.

```
;-------------------------------------------------------------------------
;- Setup Application Operating Mode and Enable the interrupts
;-------------------------------------------------------------------------
  msr             CPSR_c, #ARM_MODE_USER        ; set User mode
  ldr             r13, =TOP_APPLICATION_STACK   ; Init stack User
```

Initialize Software Variable and Branch to Main Function

The next task is to initialize the data memory by entering a loop that writes zeroes into allocations used for data storage. This may seem superfluous, but there are two reasons for this:

1. In C language, any non-initialized variable is supposed to contain zero as an initial value.

2. This makes the program behavior reproducible, even if not all variables are initialized explicitly.
   - The table of initial values for the initialized variable (in the C language sense) is copied to the location in RAM where the variables are positioned.
   - The linker puts the initial values in the same order as the variables in RAM, so a mere block copy is sufficient for this initialization.

The initial values for any initialized variables must be copied from ROM to RAM. All other variables must be initialized to zero.

When the compiler compiles a function called main(), it generates a reference to the symbol __main to force the linker to include the basic C run-time system from the ANSI C library.The library initialization code called at __main performs the copying and initialization. The function main() should be a closed loop and should not return.

```
;-------------------------------------------------------------------------
;- Branch on C code Main function (with interworking)
;-------------------------------------------------------------------------
  IMPORT          __main

  ldr             r0, =__main
  bx              r0

  END
```

ATMEL®

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 487-2600

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 2-40-18-18-18
FAX (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-42-53-60-00
FAX (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
TEL (44) 1355-803-000
FAX (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
TEL (49) 71-31-67-0
FAX (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/ High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-76-58-30-00
FAX (33) 4-76-58-34-80

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

ARM POWERED®

Printed on recycled paper.

2644A–ATARM–06/02   0M