Application Note 179

Cortex™-M3 Embedded Software Development

Released on: March 2007

ARM®

Copyright © 2007. All rights reserved.
ARM DAI0179B

Application Note 179
Cortex-M3 Embedded Software Development

Copyright © 2007. All rights reserved.

Release Information

Table 1 Change history

Date	Issue	Change
January 2007	A	First release (withdrawn)
March 2007	B	Second release

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means "ARM or any of its subsidiaries" as appropriate.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

1 The Cortex™-M3

This application note introduces the main features of the ARM Cortex™-M3 processor and describes different aspects of developing software for it. It also covers the migration of existing ARM projects to the Cortex-M3 platform.

The ARM Cortex-M3 is a high performance, low cost and low power 32-bit RISC processor. The Cortex-M3 processor only executes Thumb-2 instructions. It does not support the ARM instruction set. The Cortex-M3 processor is based on the ARM architecture v7-M and has an efficient Harvard 3-stage pipeline core. It also features hardware divide and low-latency *Interrupt Service Routine*(ISR) entry and exit.

As well as the CPU core, the Cortex-M3 processor includes a number of other components. These include a *Nested Vectored Interrupt Controller* (NVIC), an optional *Memory Protection Unit* (MPU), Timer, *Debug Access Port* (DAP) and optional *Embedded Trace Macrocell* (ETM). The Cortex-M3 also has a fixed memory map.

1.1 Nested Vectored Interrupt Controller (NVIC)

Depending on the implementation used by the silicon manufacturer, the NVIC can support up to 240 external interrupts with up to 256 different priority levels that can be dynamically reprioritized. It supports both level and pulse interrupt sources. The processor state is automatically saved by hardware on interrupt entry and is restored on interrupt exit. The NVIC also supports tail-chaining of interrupts.

The use of an NVIC in the Cortex-M3 means that the vector table for a Cortex-M3 is very different to previous ARM cores. The Cortex-M3 vector table contains the address of the exception handlers and ISR, not instructions as most other ARM cores do. The initial stack pointer and the address of the reset handler must be located at `0x0` and `0x4` respectively. These values are then loaded into the appropriate CPU registers at reset.

1.2 Memory Protection Unit (MPU)

The MPU is an optional component of the Cortex-M3. If included, it provides support for protecting regions of memory through enforcing privilege and access rules. It supports up to eight different regions, each of which can be split into a further eight equal-size sub-regions.

1.3 Debug Access Port (DAP)

The DAP uses an AHB-AP interface to communicate with the processor and other peripherals. There are two different supported implementations of the Debug Port, the *Serial Wire JTAG Debug Port* (SWJ-DP) and the *Serial Wire Debug Port* (SW-DP). Your Cortex-M3 implementation might contain either of these depending on the implementation used by your silicon manufacturer.

1.4 Memory map

Unlike most previous ARM cores, the overall layout of the memory map of a device based around the Cortex-M3 is fixed. This allows easy porting of software between different systems based on the Cortex-M3. The address space is split into a number of different sections. This is shown in Figure 1 on page 4 and described in Table 2 on page 4.

Copyright © 2007. All rights reserved.

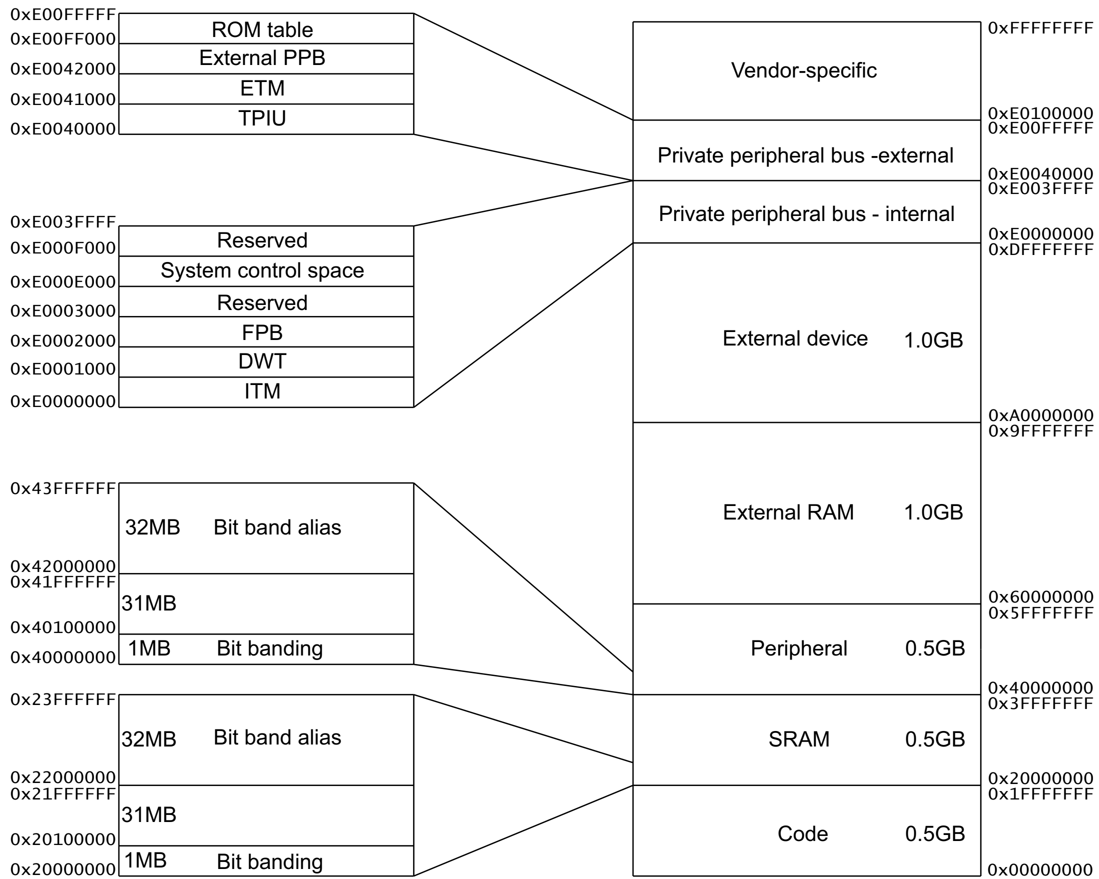

Figure 1 Cortex-M3 memory map

Table 2 Details of Cortex-M3 memory map

Memory Region	Description	Accessed via bus
code	For code memory (flash, ROM, or remapped RAM).	ICode and DCode
SRAM	For on-chip SRAM with bit-banding feature.	system
peripheral	For normal peripherals with bit-banding feature.	system
external RAM	For external memory.	system

 Copyright © 2007. All rights reserved.

Table 2 Details of Cortex-M3 memory map (continued)

Memory Region	Description	Accessed via bus
external device	Memory space for external peripherals or shared memory.	system
private peripheral	Address space for system devices such as MPU, NVIC, DAP, and other CoreSight devices.	system
vendor specific	For additional uses specified by the vendor.	.

 Copyright © 2007. All rights reserved.

2 Developing software for Cortex-M3

This section describes the different aspects of developing software for the Cortex-M3 and demonstrates how to write code to configure and use the main features of the core. The code examples in this section are designed for use with the *RealView Compilation Tools* (RVCT) 3.0 or later.

2.1 Exception handling

Writing the exception table

The easiest way to populate the vector table is to use a scatter file to place a C array of function pointers at memory address `0x0`. The C array can be used to configure the initial stack pointer, image entry point and the addresses of the exception handlers.

Example 1 Example C structure for exception handlers

```
/* Filename: exceptions.c */
typedef void(* const ExecFuncPtr)(void) __irq;

/* Place table in separate section */
#pragma arm section rodata="exceptions_area"

ExecFuncPtr exception_table[] = {
    (ExecFuncPtr)&Image$$ARM_LIB_STACKHEAP$$ZI$$Limit,
    (ExecFuncPtr)__main, /* Initial PC, set to entry point */
    NMIException,
    HardFaultException,
    MemManageException,
    BusFaultException,
    UsageFaultException,
    0, 0, 0, 0, /* Reserved */
    SVCHandler,
    DebugMonitor,
    0, /* Reserved */
    PendSVC,SysTickHandler,

    /* Configurable interrupts start here...*/
    InterruptHandler0,
    InterruptHandler1,
    InterruptHandler2

    /*
    …
    */
};

#pragma arm section
```

Note

Note that the first two items in this structure are the initial stack pointer and the image entry point. The initial stack pointer is generated using a linker defined symbol. See *Stack and heap configuration* on page 11 for details. Example 1 uses the C library entry point (`__main`) as the entry point for the image.

 Copyright © 2007. All rights reserved.

The exception table has also been placed in its own section. This has been done using `#pragma arm section rodata="exceptions_area"`. This directive instructs the compiler to place all the RO (read-only) data between `#pragma arm section rodata="exceptions_area"`and `#pragma arm section`into its own section called `exceptions_area`. This section can then be referred to in the scatter file. The exception table is then placed at the correct location in the memory map, at address `0x0`.

Writing the exception handlers

The core saves the system state when an exception occurs and restores it on return. The exception handlers do not therefore need to save or restore the system state and can be written as an ordinary ABI-compliant C function. However, we recommend that you use the `__irq` qualifier to aid clarity of code. The keyword is also used by the compiler to maintain eight-byte alignment of the stack where necessary. See *Eight byte stack alignment* on page 13 for further details.

Example 2 Simple C exception handler

```
__irq void SysTickHandler(void)
{
    printf("----- SysTick Interrupt -----");
}
```

Note

Clearing of an interrupt source must be handled by the ISR.

On the Cortex-M3, exception prioritization, nesting of exceptions, and saving of corruptible registers is handled entirely by the core to permit efficient handling. This means that interrupts remain enabled by the core on entry to every exception handler.

Placing the exception table

Because the exception table has been placed in its own section in the object it can be easily placed at `0x0` using a scatter file as shown in Example 3.

Example 3 Placing exception table in scatterfile

```
LOAD_REGION 0x00000000 0x00200000
{
    ;; Maximum of 256 exceptions (256*4 bytes == 0x400)
    VECTORS 0x0 0x400
    {
        exceptions.o (exceptions_area, +FIRST)
    }
}
```

Note

`+FIRST` is used to ensure that `exceptions_area` is placed at the very beginning of the region. It also prevents the vector table from being removed by the unused section elimination mechanism of the linker.

Copyright © 2007. All rights reserved.

Configuring the System Control Space (SCS) registers

The SCS registers are located at `0xE000E000`. As there are a large number of individual registers, it is best to use a structure to represent them. This can then be positioned in the correct memory location by adding this structure to the scatter file, using a similar method to the exception table. Example 4 shows an example structure for the SCS registers.

Example 4 SCS Register Structure

```
typedef volatile struct {
    int MasterCtrl;
    int IntCtrlType;
    int zReserved008_00c[2];
         /* Reserved space */
    struct {
        int Ctrl;
        int Reload;
        int Value;
        int Calibration;
    } SysTick;
    int zReserved020_0fc[(0x100-0x20)/4];
            /* Reserved space */

    /* Offset 0x0100 */
    struct {
        int Enable[32];
        int Disable[32];
        int Set[32];
        int Clear[32];
        int Active[64];
        int Priority[64];
    } NVIC;
    int zReserved0x500_0xcfc[(0xd00-0x500)/4];
            /* Reserved space */

    /* Offset 0x0d00 */
    int CPUID;
    int IRQcontrolState;
    int ExceptionTableOffset;
    int AIRC;
    int SysCtrl;
    int ConfigCtrl;
    int SystemPriority[3];
    int SystemHandlerCtrlAndState;
    int ConfigurableFaultStatus;
    int HardFaultStatus;
    int DebugFaultStatus;
    int MemManageAddress;
    int BusFaultAddress;
    int AuxFaultStatus;
    int zReserved0xd40_0xd90[(0xd90-0xd40)/4];
            /* Reserved space */

    /* Offset 0x0d90 */
    struct {
        int Type;
        int Ctrl;
        int RegionNumber;
        int RegionBaseAddr;
```

 Copyright © 2007. All rights reserved.

```
        int RegionAttrSize;
    } MPU;
} SCS_t;
```

Note

This register structure might not contain all of the SCS registers in your device. See the reference manual provided by the silicon manufacturer of your device.

Configuring individual IRQs

Each IRQ has an individual enable bit in the Interrupt Set Enable Registers, part of theNVIC registers. To enable an interrupt you need to set the corresponding bit in the Interrupt Set Enable Register. See the reference manual provided by the silicon manufacturer of the device you are using for specific details on the Interrupt Set Enable Register.

Example 5 shows interrupt enable code for the SCS structure shown in Example 4 on page 8.

Example 5 IRQ enable function

```
void NVIC_enableISR(unsigned isr)

{

    /* The isr argument is the number of the interrupt to enable. */

    SCS.NVIC.Enable[ (isr/32) ] = 1<<(isr % 32);
}
```

Note

Some registers in the SCS region can only be accessed from Privileged mode. Individual IRQs can be disabled by setting the appropriate bit in the Interrupt Clear Enable Registers.

Interrupt priorities

Each individual interrupt can be assigned a priority level via the Interrupt Priority Registers. Depending on the implementation, up to 256 different priority levels can be assigned to each individual interrupt. The priority levels are represented using up to 8 bits. Groups of four interrupt priorities are stored in each word of memory.

The lower the assigned priority number, the higher the priority of the interrupt. Therefore 0 is the highest priority and 255 the lowest.

2.2 Memory Protection Unit (MPU)

The Cortex-M3 includes an optional MPU. See the datasheets and other information from your silicon manufacturer to determine if an MPU is included.

Copyright © 2007. All rights reserved.

MPU register locations

The MPU registers are located at `0xE000ED90`. There are 5 basic MPU registers and a number of alias registers for each of the regions. Table 3 provides a brief overview of the MPU registers. See the *Cortex-M3 Technical Reference Manual* for further details.

Table 3 MPU Registers

Name	Address	Description
MPU type	`0xE000ED90`	Contains the number of regions in bits [15:8] and 0 for no MPU.
control register	`0xE000ED94`	Controls enabling/disabling of the MPU, the use of the default memory map for privileged accesses and whether the MPU is enabled during Hard Fault, NMI and Fault Mask handlers.
region number	`0xE000ED98`	Selects the region you want to configure.
region base address	`0xE000ED9C`	Sets or reads the base address of region.
region attribute and size	`0xE000EDA0`	Sets or reads the size and permissions of region.

There are also aliases of the region base address registers and region attribute and size register for each region. These follow the region attribute and size register directly. They are located at `0xE000EDA4` rising through memory sequentially. These are particularly useful for configuring the MPU quickly at power on, using STM instructions to write to the aliased addresses.

Configuring MPU regions

To configure an MPU region you must first select the region you want to configure. You can use one of two methods to do this. You can select the region by writing the appropriate value to the region number register. Another way to do this is to use bits 0 to 3 of the region base address register and set the VALID bit. With this second method you can also program the region base address into the region base address register at the same time.

When you have selected the region you must program the base address of the region. The base address value must be aligned to a multiple of the size of the region. So a 64KB region must be aligned on a multiple of 64KB, for example `0x00010000`, `0x00020000` and so on.

Finally, you need to configure the permissions, size and enable the region using the region attribute and size register. See the *Cortex-M3 Technical Reference Manual* for details of the register layouts.

Note

The MPU must be enabled by setting bit 0 of the MPU Control Register before any regions will be active.

 Copyright © 2007. All rights reserved.

Memory region attribute and size (types and access permissions)

The MPU supports a number of different memory types, extensions and attributes.These are configured for each region in the region attribute and size register. Table 4 provides details of the region attribute and size register.

Table 4 Region attribute and size registers

Field	Name	Description
[28]	XN	Instruction access disable bit, 1= disable instruction fetches.
[26:24]	AP	Data access permissions, allows you to configure read/write access for User and Privileged mode.
[21:19]	TEX	Type extension field, allows you to configure memory access type, for example strongly ordered, peripheral.
[18]	S	Shareable.
[17]	C	Cacheable.
[16]	B	Bufferable.
[15:8]	SRD	Sub-region disable field (see *Sub-regions*).
[5:1]	REGION SIZE	Region size of the region be configured, for example 4K, 8K.
[0]	SZENABLE	Region enable bit.

For full details about these fields see the *Cortex-M3 Technical Reference Manual*.

Sub-regions

Each memory region is divided into eight sub-regions that can be individually disabled without affecting the rest of the region. Each sub-region is one-eighth of the main region and can be disabled using the SRD fields in the region attribute and size register. The lowest bit of the SRD field disables the sub-region with the lowest address.

Sub-regions are useful for overlapping memory regions, for example, if you have a large region but would like different attributes for a small section of it. A sub-region in the larger region could be disabled and a second MPU region used for that sub-region to provide the required attributes.

Note

Sub-regions cannot be used on regions of size 32, 64 and 128 bytes.

2.3 Stack and heap configuration

Configuring Stack and heap

The *RealView Compilation Tools* (RVCT) provide a number of methods of configuring the location for the stack and heap. The two main methods are to either re-implement the `__user_initial_stackheap()` function or to place the stack and heap in the scatter file using specific region names.

Copyright © 2007. All rights reserved.

The tools also support two main types of stack and heap implementations, namely the one and two region models. In the one region model, the stack and heap share a single area of memory. The heap grows up from the bottom of the memory region while the stack grows down from the top.

In the two region model the heap and the stack each have their own memory region. The heap still grows upwards through memory and the stack still descends from the top of its region. Please refer to the *RVCT Developer Guide* and the *RVCT Compiler and Libraries Guide* for further information.

Note

You might want to use the MPU to place a region to detect overflows

One region model

If you are using the one region model the easiest way to place the stack and heap region in memory is in the scatter file. To do this you will need to use the special region name ARM_LIB_STACKHEAP in your scatter file with the address and size of the stack and heap region. See Example 6.

Example 6 Example one region model scatter file extract

```
;; Heap and stack share 1MB

ARM_LIB_STACKHEAP 0x20100000 EMPTY 0x100000

{
}
```

Note

`EMPTY` is used to indicate that it is intended that the region is not populated at link time.

The initial stack pointer value can then be placed using the linker-defined symbol`Image$$ARM_LIB_STACK$$ZI$$Limit` in the first entry, at `0x0`, in your vector table.

Instead of using the special region name, you can re-implement the`__user_initial_stackheap()` function. However, you must still ensure that you correctly specify the initial SP value in your vector table. See the *RVCT Developer Guide* and *RVCT Compiler and Libraries Guide* for information on `__user_initial_stackheap()`.

Two Region Model

To use the two region model you must specify two regions in the scatter file, one for the heap and one for the stack, that have special region names of `ARM_LIB_HEAP`and `ARM_LIB_STACK`. You also need to add either `IMPORT __use_two_region_memory`from assembly language or `#pragma import(__use_two_region_memory)` from C. This specifies that you want to use the two region model and not the (default) one region model. See Example 7.

Example 7 Example two region model scatter file extract

```
; Heap starts at 1MB and grows upwards

ARM_LIB_HEAP 0x20100000 EMPTY 0x100000-0x8000

{
```


Copyright © 2007. All rights reserved.

```
}

; Stack space starts at the end of the 2MB of RAM

; and grows downwards for 32KB (indicated by the negative length)

ARM_LIB_STACK 0x20200000 EMPTY -0x8000

{
}
```

The initial stack pointer value can then be placed using the linker-defined symbol `Image$$ARM_LIB_STACK$$ZI$$Limit` in the first entry, at `0x0` in your vector table.

Again, another way to place the stack and heap is by re-implementing `__user_initial_stackheap()` instead of using the special region. However, you must add the initial stack pointer to your vector table as before.

Eight byte stack alignment

The *Application Binary Interface* (ABI) for the ARM architecture requires that the stack must be eight-byte aligned on all external interfaces, such as calls between functions in different source files. However, code does not need to maintain eight-byte stack alignment internally, for example in leaf functions.

This means that when an interrupt or exception occurs the stack might not be correctly eight-byte aligned. Revision 1 and later of the Cortex-M3 silicon can automatically align the stack pointer when an exception occurs. This behavior must be enabled by setting `STKALIGN` (bit 9) in the Configuration Control Register at address `0xE000ED14`.

If you are using revision 0 of the Cortex-M3, this adjustment cannot be performed in hardware. The compiler can generate code in your IRQ handlers that correctly aligns the stack. To do this you must prefix your IRQ handlers with `__irq` and use the `--cpu=Cortex-M3-rev0` compiler switch, not `--cpu=Cortex-M3`.

Note

The `--cpu=Cortex-M3-rev0` compiler switch is only supported in RVCT 3.0 SP1 (build 586) and later versions.

2.4 Instruction set support

There are a number of instructions supported by the Cortex-M3 that the compiler cannot generate by itself, as these normally have a specific use and their function is not easily expressed in the C language. However these instructions can still be used either through the assembler, compiler intrinsics, or embedded assembly.

Note

More detail on the following instructions and the full instruction set can be found in the *RVCT Assembler Guide* or *ARM Architecture Reference Manual*.

Memory access instructions

The main memory instructions that the compiler cannot generate directly from C code are the load and store exclusive instructions, `LDREX` and `STREX`. These are used to perform exclusive memory accesses, for example to provide mutexes between different threads.

Copyright © 2007. All rights reserved.

Barrier instructions

The Cortex-M3 supports a number of barrier instructions. These can be used to ensure the completion of certain events before starting the next instruction or event.

The *Instruction Synchronization Barrier* (ISB) flushes the pipeline in the processor, so that all instructions following the ISB are fetched from cache or memory, after the instruction has been completed. It ensures that changes to the system, for example the MPU, take immediate effect.

The *Data Synchronization Barrier* (DSB) acts as a special kind of memory barrier. The DSB operation will complete when all explicit memory accesses before this instruction have completed. No instructions after the DSB will be executed until the DSB instruction has completed, that is, when all of the pending accesses have completed.

The *Data Memory Barrier* (DMB) acts as a memory barrier. It has slightly different behavior to DSB. The DMB instruction will ensure that any memory accesses before the DMB have completed before any memory access from instructions following the DMB instruction are performed.

Example 8 shows a fragment of typical MPU code to show how these barrier instructions are used. This involves creating small functions using the embedded assembler, each containing a single barrier instruction. These functions can later be inlined by the linker.

Example 8 Demonstration of barrier instructions using pseudo-intrinsics

```
/* pseudo_intrinsics.c */
/* Small embedded assembly functions for barrier instructions*/
/* Link with armlink --inline ... */

__asm void __ISB(void)
{
    ISB
    BX lr
}
__asm void __DSB(void)
{
    DSB
    BX lr
}
/* scs.c - Initialize System Control Space registers */
void SCS_init(void)
{
    /* Code to configure the MPU regions inserted here
    …
    */

    /* Enable the MPU */
    SCS.MPU.Ctrl |= 1;

    /* Force Memory Writes before continuing */
    __DSB();

    /* Flush and refill pipline with updated permissions */
    __ISB();
}
```

 Copyright © 2007. All rights reserved.

Conditional execution

Unlike ARM instructions, most Thumb instructions are unconditional. Thumb-2 adds three conditional instructions in addition to the 16-bit conditional branch provided in the Thumb-1 instruction set. These are:

- A 32-bit conditional branch. This has an improved branch range of +/- 1MB.
- 16-bit *compare and branch on zero* (CBZ) and *compare and branch on non-zero* (CBNZ) instructions, with a branch range of +4 to +130 bytes. For example, this can be generated for C code to branch out of a loop when counting down to zero.
- A 16-bit *if-then* (IT) instruction. The IT instruction can be used to make up to four following instructions conditional. This is the instruction used most often in place of conditional execution, for example in a short if statement.

The assembler can automatically generate appropriate IT instructions in place of a conditionally executed instruction. This is particularly useful when porting legacy ARM-based code to the Cortex-M3.

System “hints”

There are a number of “hint” instructions that can be used to direct the core to perform an operation if it is supported by your implementation. The instructions only provide an indication to the core and do not force the core to do as instructed. Some or all the hint instructions will execute as a NOP if they are not supported by your device. Table 5 summarizes the hint instructions available in the Cortex-M3.

Table 5 Hint Instructions

Instruction	Operation	Description
WFE	Wait For Event	Indicates to the processor to enter low power mode and wait for an event before waking. This requires no software intervention when woken.
WFI	Wait For Interrupt	Indicates to the processor to enter low power mode and wait for an interrupt before waking. This requires no software intervention when woken.
SEV	Send Event	Sends an event to all processors in a multi-processor system.

2.5 Bit-banding

Bit-banding maps a complete word of memory onto a single bit in the bit-band region. For example, writing to one of the alias words will set or clear the corresponding bit in the bitband region.

This allows every individual bit in the bit-banding region to be directly accessible from a word-aligned address using a single LDR instruction. It also allows individual bits to be toggled from C without performing a read-modify-write sequence of instructions.

Address translation

The Cortex-M3 has two 32MB regions that map onto the two 1MB bit-band regions. The two regions are separate, one in the SRAM region and one in the peripheral region.

Copyright © 2007. All rights reserved.

Each bit in the bit-band region is addressed sequentially in the 32MB alias region. For example, the eighth bit in the bit-band region can be accessed using the eighth word in the 32MB alias region.

Reading and writing to the bit-banding region

When writing to the alias regions bit 0 of the 32 bit word is used to set the value at the bit-banding region. Reading from the alias address will return the value from the bit-band region in bit 0 and the other bits will be cleared.

You can also access the base bit-band region itself in the same way as normal memory, using word, half word, and byte accesses.

Using bit-banding from C code

Bit-banding can be used from C code using pre-processor macros to perform the accesses. Example 9 demonstrates how you can do this.

Example 9 Using bit-banding from C code using pre-processor macros

```
#define BITBAND_SRAM_REF 0x20000000
#define BITBAND_SRAM_BASE 0x22000000
#define BITBAND_SRAM(a,b) ((BITBAND_SRAM_BASE + (a-BITBAND_SRAM_REF)*32 \
    + (b*4))) // Convert SRAM address
#define BITBAND_PERI_REF 0x40000000
#define BITBAND_PERI_BASE 0x42000000
#define BITBAND_PERI(a,b) ((BITBAND_PERI_BASE + (a-BITBAND_PERI_REF)*32 \
    + (b*4))) // Convert PERI address

#define MAILBOX 0x20004000
#define TIMER 0x40004000
// Mailbox bit 0
#define MBX_B0 *((volatile unsigned int *)(BITBAND_SRAM(MAILBOX,0)))
// Mailbox bit 7
#define MBX_B7 *((volatile unsigned int *)(BITBAND_SRAM(MAILBOX,7)))
// Timer bit 0
#define TIMER_B0 *((volatile unsigned char *)(BITBAND_PERI(TIMER,0)))
// Timer bit 7
#define TIMER_B7 *((volatile unsigned char *)(BITBAND_PERI(TIMER,7)))

int main(void)
{
    unsigned int temp = 0;

    MBX_B0 = 1; // Word write
    temp = MBX_B7; // Word read
    TIMER_B0 = temp; // Byte write
    return TIMER_B7; // Byte read
}
```

2.6 Execution Modes

Copyright © 2007. All rights reserved.

Operating Modes

The Cortex-M3 supports Privileged and User (non-privileged) execution. Code run as Privileged has full access rights whereas code executed as User has limited access rights. The limitations include restrictions on instruction use such as MSR fields, access to memory and peripherals based on system design, and restrictions imposed by the MPU configuration.

The processor supports two operation modes, Thread mode and Handler mode. Thread mode is entered on reset and normally on return from an exception. When in Thread mode, code can be executed as either Privileged or Unprivileged.

Handler mode will be entered as a result of an exception. Code in Handler mode is always executed as Privileged, therefore the core will automatically switch to Privileged mode when exceptions occur.

You can change between Privileged Thread mode and User Thread mode when returning from an exception by modifying the `EXC_RETURN` value in the link register (R14). You can also change from Privileged Thread to User Thread mode by clearing `CONTROL[0]` using an MSR instruction. However, you cannot directly change to privileged mode from unprivileged mode without going through an exception, for example an SVC.

Main and Process Stacks

The Cortex-M3 supports two different stacks, a main stack and a process stack. To support this the Cortex-M3 has two stack pointers (R13). One of these is banked out depending on the stack in use. This means that only one stack pointer at a time is visible as R13. However, both stack pointers can be accessed using the MRS and MSR instructions.

The main stack is used at reset, and is always used in Handler mode (when entering an exception handler). The process stack pointer is only available as the current stack pointer when in Thread mode. You can select which stack pointer (main or process) is used in Thread mode in one of two ways, either by using the `EXC_RETURN` value when exiting from Handler Mode or while in Thread Mode by writing to `CONTROL[1]` using an `MSR` instruction.

Note

The process stack pointer will need to be initialized by your context switch code or your initialization code.

2.7 Supervisor Calls (SVC)

As with previous ARM cores there is an instruction, SVC (formerly SWI) that generates a supervisor call. Supervisor calls are normally used to request privileged operations or access to system resources from an operating system.

The SVC instruction has a number embedded within it, often referred to as the SVC number. This is sometimes used to indicate what the caller is requesting. On previous ARM cores you had to extract the SVC number from the instruction using the return address in the link register, and the other SVC arguments were already available in R0 through R3.

On the Cortex-M3, the core saves the argument registers to the stack on the initial exception entry. A late-arriving exception, taken before the first instruction of the SVC handler executes, might corrupt the copy of the arguments still held in R0 to R3. This means that the stack copy of the arguments must be used by the SVC handler. Any return value must also be passed back to the caller by modifying the stacked register values. In order to do this, a short piece of assembly code must be implemented as the start of the SVC handler. This identifies which stack the registers were saved to, extracts the SVC number from the instruction, and passes the number and a pointer to the arguments to the main body of the handler written in C.

 Copyright © 2007. All rights reserved.

Example 10 shows an example SVC handler. This code tests the `EXC_RETURN` value set by the processor to determine which stack pointer was in use when the SVC was called. On most systems this will be unnecessary, because in a typical system design supervisor calls will only be made from user code which uses the process stack. In this case, the assembly code can consist of a single MSR instruction followed by a tailcall branch (B instruction) to the C body of the handler.

Example 10 Example SVC Handler

```
__asm void SVCHandler(void)
{
    IMPORT SVCHandler_main
    TST lr, #4
    MRSEQ r0, MSP
    MRSNE r0, PSP
    B SVCHandler_main
}
void SVCHandler_main(unsigned int * svc_args)
{
    unsigned int svc_number;
    /*
    * Stack contains:
    * r0, r1, r2, r3, r12, r14, the return address and xPSR
    * First argument (r0) is svc_args[0]
    */
    svc_number = ((char *)svc_args[6])[-2];
    switch(svc_number)
    {
        case SVC_00:
            /* Handle SVC 00 */
            break;
        case SVC_01:
            /* Handle SVC 01 */
            break;
        default:
            /* Unknown SVC */
            break;
    }
}
```

Example 11 shows how you can have different declarations for a number of SVCs. `__svc` is a compiler keyword that replaces a function call with an SVC instruction containing the specified number.

Example 11 Example of calling an SVC from C code

```
#define SVC_00 0x00
#define SVC_01 0x01
void __svc(SVC_00) svc_zero(const char *string);
void __svc(SVC_01) svc_one(const char *string);
int call_system_func(void)
{
    svc_zero("String to pass to SVC handler zero");
    svc_one("String to pass to a different OS function");
}
```

 Copyright © 2007. All rights reserved.

2.8 System Timer (SysTick)

About the SysTick

The SCS also includes a system timer (SysTick) that can be used by an operating system to ease porting from another platform. The SysTick can be polled by software or can be configured to generate an interrupt. The SysTick interrupt has its own entry in the vector table and therefore can have its own handler.

The SysTick is configured through the four registers described in Table 6.

Table 6 SysTick Registers

Name	Address	Description
SysTick Control and Status	`0xE000E010`	basic control of SysTick e.g. enable, clock source, interrupt or poll
SysTick Reload Value	`0xE000E014`	value to load Current Value register when 0 is reached
SysTick Current Value	`0xE000E018`	the current value of the count down.
SysTick Calibration Value	`0xE000E01C`	might contain the number of ticks to generate a 10ms interval and other information, depending on the implementation

Configuring SysTick

To configure the SysTick you need to load the SysTick Reload Value register with the interval required between SysTick events. The timer interrupt or `COUNTFLAG` bit (in the SysTick Control and Status register) is activated on the transition from 1 to 0, therefore it activates every n+1 clock ticks. If a period of 100 is required 99 should be written to the SysTick Reload Value register. The SysTick Reload Value register supports values between 1 and `0x00FFFFFF`.

If you want to use the SysTick to generate an event at a timed interval, for example 1ms, you can use the SysTick Calibration Value Register to scale your value for the Reload register. The SysTick Calibration Value Register is a read-only register that contains the number of pulses for a period of 10ms, in the `TENMS` field (bits 0 to 23). This register also has a `SKEW` bit (30) that is used to indicate that the calibration for 10ms in the `TENMS` section is not exactly 10ms due to small variations in clock frequency. Bit 31 is used to indicate if the reference clock is provided.

The Control and Status Register allows you to select between polling the timer by reading `COUNTFLAG` (bit 16), or by the SysTick generating an interrupt.

By default the SysTick is configured for polling mode. In this mode, user code must read `COUNTFLAG` to ascertain if the SysTick event had occurred. This is indicated by `COUNTFLAG` being set. Reading of the Control and Status register clears the `COUNTFLAG` bit. To configure the SysTick to generate an interrupt you must set `TICKINT` (bit 1 of the SysTick Control and Status register) HIGH. You will also need to enable the appropriate interrupt in the NVIC. You must set `CLKSOURCE` (bit 2) to 1 to select the core clock and zero select the external reference clock.

The Timer is enabled by setting bit 0 of the SysTick Status and Control register.

2.9 RVCT 3.0 Options

Copyright © 2007. All rights reserved.

Compiler and Assembler Options

When building code you will need to specify the correct CPU on your compiler and assembler command lines, as by default the RVCT 3.0 tools will build code for an ARMv4T core.

To do this you must add `--cpu name` to your command line, where `name` is either `Cortex- M3` or `Cortex-M3-rev0`. You do not need to add `--thumb` to your command line as the Cortex-M3 is a Thumb-2 only core. The compiler will automatically generate Thumb-2 instructions and the assembler will only accept valid Thumb-2 instructions.

Linker Options

You do not need to specify any specific options to the linker to generate a suitable image for a Cortex-M3. The linker can obtain the required information from the object files specified on the linker command line.

However, we recommend that you use a scatter file (`--scatter filename`) as the memory map of the Cortex-M3 cannot easily be represented using command-line options available in the linker. You must also specify an entry point for the image using the `--entry` switch. Normally this will be the same entry point that you have specified as your reset handler in the exception table.

Copyright © 2007. All rights reserved.

3 Moving Existing ARM Projects to the Cortex-M3

This section discusses how to migrate an existing ARM project to a new Cortex-M3 platform. An example would be the migration of an ARM7TDMI-based application to the Cortex M-3 platform.

The best strategy is usually to build up the functionality gradually, starting with a minimal version of the project. You can do this easily using comments or preprocessor macros to remove sections of the functionality. This also allows the functions to be easily reintroduced later in the migration process, one at a time, to allow for thorough testing of each platform-dependent function.

3.1 General code modifications

Most platform-independent sections of your code will usually work correctly on the Cortex-M3 without modification. However, there are certain features of the code that you might need to modify and update for the new target.

Modifications to C code

When migrating a project from an ARM7 core to the Cortex-M3, you must recompile all of your C code for Thumb-2 with an appropriate `--cpu` option as described in *Compiler and Assembler Options* on page 20. This includes any third-party libraries.

Legacy Thumb-1 code is binary compatible with the Cortex-M3, and such code can be run on the new processor. However, in RVCT 3.0 the linker cannot protect you from accidentally linking object files and libraries that contain some ARM instructions. It is recommended that all code is recompiled for Thumb-2 if at all possible. If you need to use legacy objects or libraries, you must manually check that no ARM instructions are included in the linked image. If you have third-party libraries that are targeted at an ARM7, you might need to contact your supplier for a Thumb-2 version of their library.

The source code itself might also require some minor changes. In particular, statechanging pragma directives (`#pragma arm` and `#pragma thumb`) no longer apply and must be removed. Inline assembly code does not support compilation for Thumb-2, therefore such code must be rewritten using C, C++ or embedded assembly code.

Modifications to assembly code

Special care should be taken when porting assembly code.

If present, directives that cause assembly of ARM instructions (`ARM` or `CODE32`) must be removed or changed to THUMB in every case. Most existing code should then assemble without difficulty if you ensure that you provide the correct `--cpu` option on the assembler command line. However, some rare instructions that are supported by the ARM instruction set and not the Thumb-2 instruction set might need modification.

If any `CODE16` directives are present, be aware that these will assemble without warning if changed to `THUMB`, although there might be subtle differences in behavior. This is because `CODE16` assembles according to the legacy Thumb-1 syntax rules. For example, under the `CODE16` syntax rules many instructions without an `S` suffix will encode to flag-setting variants. Under the syntax rules of the `THUMB` directive the `S` suffix must be explicitly specified.

The assembler will insert `IT` instructions as necessary for instructions that are to be conditionally executed. For example, a single `ADDSNE r0, r0, r1` instruction (followed by an unconditional instruction) would become: `IT NE` followed by `ADDS r0, r0, r1`.

Copyright © 2007. All rights reserved.

Be careful of core-specific and architecture-specific instructions. These include:

- Co-processor instructions.
- State or mode changes and other accesses to the PSR.
- The SWP instruction, which has been replaced by LDREX and STREX.
- Some addressing modes that are not supported by LDM and STM instructions.
- Some other instructions that have additional restrictions in Thumb-2, for example LDR and STR instructions with a register offset and an immediate shift.

You might need to rewrite these portions of your code to cater for the programmer's model of the Cortex-M3. The assembler will generate a warning or error if it finds any incompatibilities. Remember that most status and control registers are memory-mapped in the Cortex-M3 and that the supported modes are very different from an ARM7TDMI. Any code which changes state or mode must be modified if appropriate or removed. Likewise, code which accesses co-processors must be removed, unless this will be emulated by a handler for the Usage Fault exception.

Also beware of code which uses the value of the PC in address arithmetic. Because Thumb-2 uses mixed 16- and 32-bit instructions, the value of the PC is always the address of the current instruction plus 4 when used in a data processing operation.

3.2 Changes to startup code

The startup code consists of the reset handler of your application, together with any initialization functions that set up the environment and peripherals before the main body of your application can run. This is specific to a particular core and target.

If your system is simple, it might be sufficient to specify the C library entry point (the `__main()` function) as your reset handler in the vector table, and perform additional initialization from the `main()` function in your own code. However, if there are peripherals that require critical initialization, you might need to write a short assembly code function to act as your initial reset handler before branching to `__main()`. Also be aware that code accessing some devices, for example the MPU, might need one or more of the memory barrier instructions after writing to these registers to ensure that the changes take effect immediately. If your project previously targeted a platform supporting an MPU or MMU, the associated code must be revised.

For all Cortex-M3 projects, you must create the new vector table as described in *Exception handling* on page 6 and add the initial stack pointer and address of your reset handler at `0x0` and `0x4`respectively.

3.3 Changes to Exception Handling

Your exception handlers must be adapted for the Cortex-M3.

You will not normally need a low-level handler in assembly language, because reentrancy is handled by the core. If your low-level handler performs additional work, you might need to split some of this into separate functions which can be called from the new handlers.

Remember to mark your IRQ handlers using the `__irq` keyword for clarity and to ensure that the compiler can maintain the stack alignment for Cortex-M3 revision 0 hardware.

The Cortex-M3 has no FIQ input. Any peripheral that signals an FIQ on the ARM7TDMI project must be moved to a high-priority vectored interrupt, or to the NMI signal of the Cortex-M3. You might need to check that the handler for this kind of interrupt does not expect to use the banked FIQ registers, as these will now need to be stacked as for another normal IRQ handler.

 Copyright © 2007. All rights reserved.

Finally, you must write a new initialization function to configure the NVIC including the interrupt priorities. Interrupts can then be enabled before entering your main application code.

Critical sections and exception behavior

On the Cortex-M3, exception prioritization, nesting of exceptions, and saving of corruptible registers is handled entirely by the core to provide very efficient handling and minimize interrupt latency. This means that interrupts remain enabled by the core on entry to every exception handler. In addition, if interrupts are disabled when returning from an exception they will not be automatically re-enabled by the processor. It is not possible to perform an atomic enabling of interrupts and return from an exception. If you have disabled interrupts temporarily in your handler, they must be re-enabled first and then a separate instruction used to return. Exceptions might therefore occur immediately before the exception return.

These features of the exception model might impact on critical sections in the code, depending on the system design. Critical sections are those that require interrupts to be disabled for the duration of their execution so that they are executed as an uninterruptible block, for example the context switching code in an operating system. Certain legacy code might make assumptions that interrupts will be disabled on entry to exception handlers and will only be enabled explicitly by the code once any critical sections have been completed. These assumptions do not hold under the new exception model of the Cortex-M3, and such code might need to be rewritten to take account of this.

3.4 Retargeting for new device

You must also perform additional steps to retarget the build on to the Cortex-M3, for example, defining the new memory map in the scatter file. This includes placement of structures over registers for the System Control Space, and the addition of library stack and heap regions as necessary. If your project uses a timer, you might want to modify the code to use the SysTick functionality provided by the SCS. See *Configuring SysTick* on page 19.

3.5 Exploiting new features of the Cortex-M3

Once the original project has been retargeted to the new platform, you might wish to modify the project to take advantage of the new features provided by the Cortex-M3. Most of the new instructions available in ARMv7M will be used automatically if you recompile your code for Thumb-2. However, certain features of the Cortex-M3 require manual code changes in order to take advantage of them. These include:

- Modifying the initialization code to enable the MPU.
- Making use of the sleep modes provided by the core to reduce power consumption.
- Exploiting bit-banding to improve the performance of bit modifications, if any of your data or peripherals are located in an appropriate bit-band region.

See *Bit-banding* on page 15 for a description of how to use the bit-banding functionality. Details of how to configure the MPU can be found in *Memory Protection Unit (MPU)* on page 9, and full details of the MPU registers are in the *Cortex-M3 Technical Reference Manual*. This also provides details of the sleep modes supported by the core.

Copyright © 2007. All rights reserved.

4 Debugging with the Cortex-M3

The Cortex-M3 provides sophisticated debug capabilities. These include:

- A flash patch and breakpoint unit to allow a debugger to set up hardware breakpoints over a flash or ROM region.
- A data watchpoint and trace unit to provide support for watchpoints and ETM triggers.
- An optional ETM for instruction and data tracing.
- An *Instrumentation Trace Macrocell* (ITM) that allows printf-style debugging using application-driven tracing of events, for example from an operating system.

In addition, the structure of the processor means that debug accesses are performed through a *Debug Access Port* (DAP). This is connected to the rest of the system, including the core, through the bus matrix. This means that you can carry out certain debug operations while the core itself is still running, if this is supported by your debugger. For example, you do not need to stop the core to be able to read from or write to external memory. This functionality is supported by RVD 3.0 when connecting to a hardware target. However, RVD 3.0 does not support the trace features of the Cortex-M3.

RVDS 3.0 also includes an *Instruction Set System Model* (ISSM) of the Cortex-M3. This provides simulation of the core, NVIC and other key features of the processor. It also models one UART peripheral and three additional timer peripherals. However, the model only simulates the breakpoint and watchpoint debug logic. The trace components are not modeled.

 Copyright © 2007. All rights reserved.