Application Note 194

Cortex™-M1 Embedded Software Development

Document number: ARM DAI 0194A
Issued: October 2007

ARM

Application Note 194
Cortex-M1 Embedded Software Development

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
October 2007	A	First release

Proprietary notice

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

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

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

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1 The Cortex-M1

The ARM Cortex™-M1 processor is the first ARM processor designed for optimal implementation in FPGAs.

As well as the CPU core, the Cortex-M1 processor includes a number of other components, some of which are optional. These include a Nested Vectored Interrupt Controller (NVIC), optional Operating System (OS) extension and Debug Access Port (DAP). The Cortex-M1 also has an architecturally defined memory map, including two optional Tightly Coupled Memories (TCMs).

The Cortex-M1 processor is based on the ARM architecture v6-M and executes a subset of the Thumb®-2 Instruction Set Architecture (ISA), consisting of all base 16-bit Thumb-2 instructions as well as new System instructions. It also features a 32-bit hardware multiplier that can be implemented as standard (faster) or small (slower) and low-latency ISR (Interrupt Service Routine) entry and exit.

1.1 Nested Vectored Interrupt Controller (NVIC)

Depending on the implementation, the NVIC can support 1, 8, 16 or 32 external interrupts with 4 different priority levels. 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 use of an NVIC in the Cortex-M1 means that the vector table for a Cortex-M1 is similar to the Cortex-M3 but very different to previous ARM cores. The Cortex-M1 vector table contains the address of the exception handlers and ISRs, whereas most previous ARM cores contained instructions. 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 Stack Pointer and Program Counter by the processor at reset.

1.2 Memory Map

Unlike most previous ARM cores, the overall layout of the memory map of a device based around the Cortex-M1 is fixed. This allows easy porting of software between different systems based on the Cortex-M1. The address space is split into a number of different sections; this is shown in Figure 1.1 and described in Table 1.1.

The Cortex-M1 has two Tightly Coupled Memories (TCM) one for instructions and one for data. Each of these can be up to 1 megabyte in size, the actual size in your device will vary depending on the implementation.

The instruction TCM is mapped at 0x0 rising to 0x100000 for a 1 megabyte TCM. The data TCM is mapped at 0x2000000 rising to 0x2100000 for a 1 megabyte TCM. Any access to the TCM address range where TCM memory is not present will result in an attempt to access external memory at that address. Therefore you can have memory from 0x0 to 0x3FFFFFF without any gaps. There is however a 1 or 2 cycle penalty to access external memory.

As the TCMs are implemented in FPGA on-chip RAM, in many cases their contents can be programmed from the FPGA's configuration flash. This can avoid the need for a separate flash device to store program and data while the device is powered down.

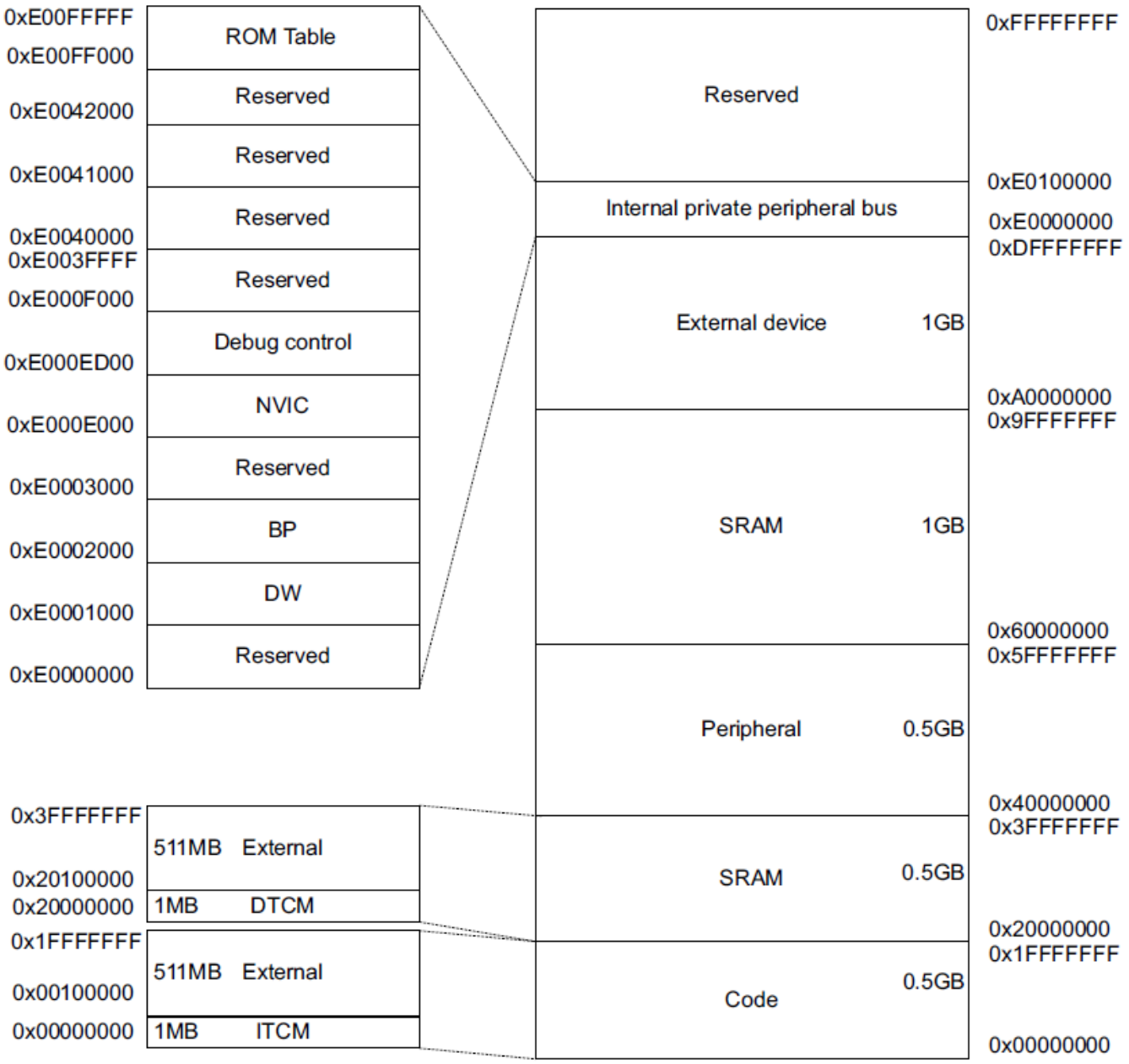

Figure 1.1 Cortex-M1 Memory Map

Table 1.1 Details of Cortex-M1 Memory Map

Memory Region	Description
Code	For instruction access to ITCM or external memory via AHB-Lite
SRAM	For data access to DTCM or external memory via AHB-Lite. Instruction fetches from this region are prevented by the processor core.
Peripheral	This is memory space for additional peripherals. Instruction fetches from this region are prevented by the processor core.
Internal Private peripheral bus	Internal address space for system devices, e.g. NVIC, Breakpoint Unit (BPU), Data Watchpoint (DW)

1.3 Debug Access Port (DAP)

The debug access port 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-M1 implementation might contain either or none of these depending on the implementation.

2 Developing Software for Cortex-M1

This section describes the different aspects of developing software for the Cortex-M1 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) 3.1 or later. A number of Cortex-M1 examples are also included in RVDS 3.1, these are located in the examples directory.

2.1 Exception handling

2.1.1 Writing the Vector 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 2.1 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$$ARM_LIB_STACK$$ZI$$Limit, /* Initial SP */
    (ExecFuncPtr)&__main,      /* Initial PC, set to entry point  */
    NMIException,
    HardFaultException,
    0, 0, 0, 0, 0, 0, 0,       /* Reserved */
    SVCHandler,                /* Only available with OS extensions */
    0, 0,                      /* Reserved */
    PendSVC,                   /* Only available with OS extensions */
    SysTickHandler,            /* Only available with OS extensions */

    /* Up to 32 external interrupt handlers start here...*/
    InterruptHandler0,
    InterruptHandler1,      /* Some dummy default interrupt handlers */
    InterruptHandler2,
    /*
    :
    */
};
#pragma arm section
```

Notice 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 (`Image$$ARM_LIB_STACK$$ZI$$Limit`); see section 2.2 for details. Example 2.1 uses the C library entry point (`__main`) as the entry point for the image, this is typically the reset handler.

The exception table has also been placed in its own section. This has been done using `#pragma arm section rodata="exceptions_area"`. This 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 can then be referred to in the scatter file to place the exception table at the correct location in the memory map (address 0x0).

Note *The least significant bit (bit[0]) of the addresses in the vector table must be set or a Hard Fault exception will be generated. The RVCT tool chain will normally ensure this if Thumb symbol names are used in the table.*

2.1.2 Placing the Exception Table

Because the vector table has been placed in its own section in the object it can be easily placed at 0x0 using a scatter file.

Example 2.3 Placing exception table in scatterfile

```
LOAD_REGION 0x00000000 0x00200000
{
  VECTORS 0x0 0xC0
  {
    exceptions.o (exceptions_area, +FIRST)
  }
}
```

Note *`+FIRST` is used to ensure that `exceptions_area` is placed at the very beginning of the region and to prevent the linker's unused section elimination mechanism from removing the vector table.*

2.1.3 Writing the Exception Handlers

The core saves the system state when an exception occurs and restores it on return. Therefore, the exception handlers do not need to save or restore the system state and can be written as an ordinary (ABI-compliant) C functions. However, we recommend that you use the `__irq` qualifier to aid clarity of code. See Section 2.3.2 for further details.

Example 2.2 Simple C exception handler

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

Note *Clearing of a level-triggered interrupt source must be handled by the ISR.*

On the Cortex-M1, 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. Therefore the core can take a higher priority exception if one occurs without the code having to specifically re-enable interrupts.

2.1.4 Configuring the System Control Space (SCS) registers

The SCS registers are located at 0xE000E000. It is best to use a structure to represent them to ensure that the offsets between the registers are correct and they are not re-arranged by the compiler. The structure can then be positioned in the correct memory location by adding this structure to the scatter file, using a similar method to the vector table. Example 2.4 below shows an example structure for the SCS registers.

Example 2.4 SCS Register Structure

```
typedef volatile struct {
    int MasterCtrl;
    int IntCtrlType;

    int zReserved008_00c[2];

    struct {
        int Ctrl;
        int Reload;
        int Value;
        int Calibration;
    } SysTick;

    int zReserved020_0fc[(0x100-0x20)/4];

    /* Offset 0x0100 */
    /* Additional space allocated to arrays to ensure alignment */
    struct {
        int Enable[32];
        int Disable[32];
        int Set[32];
        int Clear[32];
        int Unused_on_M1[64];
        int Priority[64];
    } NVIC;
    /* … more registers */
} SCS_t;
```

Note *The contents of the SCS registers may be different for your implementation, for example, the SysTick registers will be missing if the Operating System extension is not implemented.*

2.1.5 Configuring Individual IRQs

Each IRQ has an individual enable bit in the Interrupt Set Enable Registers, part of the NVIC registers. To enable an interrupt you need to set the corresponding bit in the Interrupt Set Enable Register. Please refer to the reference manual for the device you are using for specific details on the Interrupt Set Enable Register.

Example 2.5 shows example interrupt enable code for the SCS structure shown in Example 2.4.

Example 2.5 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);
}
```

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. There are 4 different priority levels that can be assigned to each individual interrupt. The priority levels are represented using up to 2 bits. Groups of 4 interrupt priorities are stored in each word of memory; the most significant 2 bits of each byte contain the priority setting.

The lower the assigned priority number, the higher the priority of the interrupt. Therefore 0 is the highest assignable priority and 3 the lowest.

The priority of all the exception is configurable apart from Hard Fault, Non Maskable Interupt (NMI) and reset, which have fixed priorities of -1, -2 and -3 respectively, where -3 is the highest priority available of the three (lowest number).

2.2 Stack and Heap Configuration

2.2.1 Configuring Stack and heap

The RealView Compilation Tools (RVCT) provides 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.

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. This is the default.

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. As the stack and heap are placed in separate regions collisions can be avoided. Please refer to the RVCT Developer Guide and the RVCT Compiler and Libraries Guide for further information.

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 2.6.

Example 2.6 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$$ARM_LIB_STACKHEAP$$ZI$$Limit` in the first entry (0x0) in your vector table.

As an alternative to 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. Please refer to 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. These also 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 informs the tools that you want to use the two region model and not the (default) one region model.

Example 2.7 Example Two Region Model scatter file extract

```
; Heap starts at 1MB and grows upwards
ARM_LIB_HEAP 0x20100000 EMPTY 0x100000-0x8000
{
}
; 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, the stack and heap can be alternatively placed 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.

2.2.2 8 byte Stack alignment

The Application Binary Interface (ABI) for the ARM Architecture requires that the stack must be 8-byte aligned on all external interfaces, such as calls between functions in different source files. However, code does not need to maintain 8-byte stack alignment internally, for example in leaf functions.

This means that when an interrupt or exception occurs the stack might not be correctly 8-byte aligned. The Cortex-M1 core will automatically align the stack pointer when an exception occurs. This behavior is always enabled, however we recommend that users manually enable it so that their image is forward compatible with the Cortex-M3. It is enabled by setting STKALIGN (bit 9) in the Configuration Control Register at address 0xE000ED14.

2.3 Instruction Set Support

The Cortex-M1 executes a subset of the Thumb-2 instruction set. These includes all 16-bit Thumb instructions, Thumb-2 16-bit instructions and new System Instructions. More detail on the following instructions and the full instruction set can be found in the RVCT Assembler Guide or ARM v6-M Architecture Reference Manual.

2.3.1 Memory Access Instructions

The Cortex-M1 supports the 16-bit encodings of the load and store instructions, for example LDR/STR, LDRSH etc. However the load and store exclusive (LDREX, STREX and SWP) instructions are not supported.

2.3.2 Data Processing Instructions

The Cortex-M1 supports the 16-bit Thumb data processing instructions introduced in Architecture v6 as well as the Thumb data processing instructions supported in earlier architectures. The table 2-3 summarizes these.

Table 2-3 Data Processing Instructions

Instruction	Description
REV	Byte-Reverse Word reverses the byte order in a 32-bit register.
REV16	Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register.
REVSH	Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, andsign extends the result to 32 bits.
SXTB	Signed Extend Byte extracts an 8-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register.
SXTH	Signed Extend Halfword extracts a 16-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register.
UXTB	Unsigned Extend Byte extracts an 8-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register.
UXTH	Unsigned Extend Halfword extracts a 16-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register.

Note *All these instructions can be generated from C code by the RVCT compiler*

2.3.3 Conditional Execution

Unlike ARM instructions, most 16-bit Thumb instructions are unconditional. The only conditional instruction available is the 16-bit conditional branch e.g. BNE, BEQ etc. These can be generated by the RVCT compiler for C code.

2.3.4 Barrier Instructions

The Cortex-M1 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 ISB (Instruction Synchronization Barrier) instruction 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 take immediate effect.

The DSB (Data Synchronization Barrier) instruction 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 DMB (Data Memory Barrier) instruction 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.

These instructions can be inserted into your C code using a compiler intrinsic or used in assembly code. Example 2.8 shows how these can be used.

Example 2.8 Demonstration of barrier instructions using intrinsics

```
void foo(void)
{
    /* Force Memory Writes before continuing */
    __dsb(0xF);

    /* Flush and refill pipeline with updated permissions */
    __isb(0xF);
}
```

2.3.5 System Instructions

There are three system control instructions available on the Cortex-M1. The MRS and MSR instructions allow the contents of special registers (e.g. the Application Program Status Register, APSR) to be transferred to and from the ARM core integer registers.

The CPS instruction is also supported to allow the enabling (CPSIE) and disabling (CPSID) of interrupts. The __disable_irq() and__enable_irq() intrinsics can be used to insert CPS instructions from your C code or used in assembly code.

2.3.6 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. They will execute as a NOP if they are not supported by your device. Again these instructions can be inserted into your C code using a compiler intrinsic or used in assembly code. Table 2-4 summarizes the hint instructions available in the Cortex-M1.

Table 2-4 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 executes as a NOP on the Cortex-M1.
WFI	Wait For Interrupt	Indicates to the processor to enter low power mode and wait for an interrupt before waking. This executes as a NOP on the Cortex-M1.
SEV	Send Event	Sends an event to all processors in a multi-processor system. This executes as a NOP on the Cortex-M1.

Note *The Cortex-M1 does not support sleep modes. Therefore WFI and WFE execute as NOP instructions.*

2.4 Execution Modes

2.4.1 Operating Modes

The processor supports two operation modes, Thread mode and Handler mode. Thread mode is entered on reset and is used to execute system initialization and application code.

Handler mode will be entered as a result of an exception. On return from the exception the core will return to Thread mode.

Note *The Cortex-M1, unlike the Cortex-M3 and earlier ARM cores, does not support an unprivileged mode.*

2.4.2 Main and Process Stacks

Note *The Process Stack is only supported if the Operating System extension is implemented*

The Cortex-M1 can support two different stacks, a main stack and a process stack. To do this the Cortex-M1 has two stack pointers (R13), one of which is banked out depending

on the stack in use. This means that only one stack pointer is visible as R13 at a time. 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 initialization code or your context switch code.*

2.5 Supervisor Calls (SVC)

Note *The SVC instruction is only supported if the Operating System extension is implemented*

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-M1, 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 at the start of the SVC handler. This identifies which stack the registers were saved to and passes the correct stack pointer to the C code as an argument. The C code then extracts the SVC number from the instruction and processes it accordingly

Example 2-8 below 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 2.8 Example SVC Handler

```
__asm void SVCHandler(void)
{
    IMPORT SVCHandler_main
    MOVS   r1, #4
    MOV    r0, lr
    TST    r0, r1
    BEQ    Main_Stack
    MRS    r0, PSP
    B      SVCHandler_main
Main_Stack
    MRS    r0, MSP
    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 2.9 shows how you can declare 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 2.9 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");
}
```

2.6 System Timer (SysTick)

Note *The System Timer is only supported if the Operating System extension is implemented*

2.6.1 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 2-7.

Table 2-7 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.

2.6.2 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 0x1 and 0x00FFFFFF.

The Control and Status Register allows you to select between polling the timer by reading COUNTFLAG (bit 16) and 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 should set TICKINT (bit 1 of the SysTick Control and Status register) HIGH. You will also need to enable the appropriate interrupt in the NVIC.

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

2.7 RVCT 3.1 Options

2.7.1 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 3.1 tools will build code for an ARM architecture v4T core.

To do this you must add `--cpu` *`name`* to your command line, where *`name`* is either `Cortex-M1, --cpu=Cortex-M1.os_extension` or `--cpu=Cortex-M1.no_os_extension`. You do not need to add `--thumb` to your command line as the Cortex-M1 is a Thumb only core. The compiler will automatically generate Thumb instructions and the assembler will only accept valid instructions.

Example compiler command line:

```
armcc –c ––debug –O2 –Ospace ––cpu Cortex-M1 file.c –o file.o
```

2.7.2 Linker Options

You do not need to specify any specific options to the linker to generate a suitable image for a Cortex-M1. The linker can obtain the required information from the object files specified on the linker command line. This allows the linker to automatically link the Cortex-M1 C runtime libraries.

However, we recommend that you use a scatter file (`--scatter` *`filename`*) as the memory map of the Cortex-M1 cannot easily be represented using the linker command-line options. You must also specify an entry point for the image using the `--entry` switch. Normally this will be the same entry point you have specified for your reset handler in the exception table.

A `-–cpu` option can be supplied to the linker to ensure that it only links in object files that are compatible with the Cortex-M1. The linker will then generate an error message if it loads any objects that are not compatible with the Cortex-M1.

Example Compiler command line:

```
armlink -–cpu Cortex-M1 –-scatter Cortex_M1.scat file1.o file2.o -–entry
__main –o image.axf
```

3 Moving Existing ARM Projects to the Cortex-M1

This section discusses the steps required to migrate an existing ARM project, for example an ARM7/9 based application, to a new Cortex-M1 platform.

As with any migration of a project to a new target 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 original 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-M1 without modification. However, there are certain features of the code that you might need to modify and update for the new target.

3.1.1 Modifications to C code

When migrating a project from an ARM7/9 core to the Cortex-M1, you must recompile all of your C code for Thumb with an appropriate `--cpu` option as described in Section 2.9.1. This includes any third-party libraries.

Legacy Thumb code is binary compatible with the Cortex-M1, and such code can be run on the new processor. However, in RVCT 3.1 the linker cannot protect you from accidentally linking object files and libraries that contain some ARM instructions. The linker can however check the build attributes of object files linked into your image and abort the link step if any incompatible build attributes are identified. To enable this feature add `--cpu Cortex-M1` to your linker command line.

It is recommended that all C code is recompiled for the Cortex-M1 if possible. If you need to use legacy objects or libraries, you must manually check that no ARM or unsupported Thumb-2 instructions are included in the linked image. If you have third-party libraries that are targeted at an ARM7/9, you should contact your supplier for a Cortex-M1 version of their library.

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

3.1.2 Changes to startup code

The startup code typically 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 might need one or more of the memory barrier instructions after writing to these registers to ensure that the changes take effect immediately.

For all Cortex-M1 projects, you must create the new vector table as described in section 2.1 and add the initial stack pointer and address of your reset handler at 0x0 and 0x4 respectively.

Note *The address of the reset handler at 0x4 must have its LSB set.*

3.1.3 Changes to Exception Handling

Your exception handlers must be adapted for the Cortex-M1.

Traditional ARM 7/9 processors require a top-level handler written in assembler language to handle re-entrancy. This can normally just be removed because re-entrancy is handled by the Cortex-M1 automatically. In most cases you simply need to populate the vector table with the address of the appropriate ISR functions written in C. If your top-level assembler handler performed additional work, you might need to split some of this into separate functions which can be called from the C handlers. Remember to mark your IRQ handlers using the `__irq` keyword for clarity.

The Cortex-M1 has no FIQ input. Any peripheral that signals an FIQ on the ARM7/9 project must be moved to a high-priority vectored interrupt, or the Cortex-M1's NMI signal. You might need to check that the handler for this kind of interrupt does not expect to use the banked FIQ registers, as the Cortex-M1 does not have the extra banked registers. Therefore if these are used they will now need to be stacked as for another normal IRQ handler and their values will not be maintained across interrupts.

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-M1, 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 to disable interrupts temporarily in your handler, they must be re-enabled first and then an additional 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-M1, and such code might need to be rewritten to take account of this.

3.1.4 Modifications to remaining assembly code

Special care should be taken when porting assembly code. The Thumb instruction set is a subset of the ARM instruction set which is primarily targeted for code generation by C compilers and not hand written assembly. Therefore there are a number of restrictions that make writing Thumb assembly code more involved.

When converting from ARM to Thumb assembler the main differences that will require attention are:

- In some Thumb data processing instructions the destination register (Rd) is always also the first source register (Rn).
- Most Thumb instructions cannot be conditionally executed. Therefore sections of ARM code that use conditional execution will need to be written to use conditional branch instructions instead.
- The PC relative branch instructions have a reduced branch range. The conditional branch has a range of +/- 256 bytes, the unconditional branch has a range of +/- 2KB and the BL has a range of +/- 4MB

- Only a subset of the ARM instruction set is available in Thumb, therefore some ARM instructions will need to be replaced with a sequence of Thumb instructions.
- Coprocessor instructions are not supported. These are commonly used to setup caches, MMU and for VFP. These are not supported on the Cortex-M1 and therefore these instructions should be removed.
- The majority of the Thumb data processing instructions are limited to using the low registers (r0 to r7), you will need to look at the register use in your assembler code to ensure that the data processing instructions are using the lower registers. The CMP, MOV and some variants of the ADD and SUB instructions can access the high registers, you may wish to use a high register for loop counters and store values not currently being used.
- All the Thumb data processing instructions that use the low registers set the ALU flags.
- The range of constants that can be embedded into a Thumb instruction is much smaller. If you use the `=<value>` syntax the assembler will automatically insert an appropriate instruction or DCD literal pool as required.
- The inline barrel shifter is not supported – Instructions that use the inline barrel shifter will need to be re-written into an instruction and separate shift operation. For example ADD r2,r2,r0,LSL #4, would need to be split into separate shift and add operations.
- There are less addressing modes supported for load and store instructions. The supported modes are
    - 8 bit PC relative immediate (LDR <rt>, <label>),
    - Register with 5 bit offset (LDR<Rt>, [<Rn>{,#<imm5>}],
    - Register with offset in a second register (LDR <Rt>,[<Rn>,<Rm>]).

  The immediate specified in these instruction will be shifted depending on the size of access being performed.
- The commonly used effective NOP instruction MOV r0,r0 should be replaced with NOP or MOV r8,r8. This is because the MOV r0,r0 instruction may be assembled to an instruction that affects the flags. However a NOP or MOV of a high register (e.g. r8) will not affect the flags and is therefore safe.

If present, directives that cause assembly of ARM instructions (`ARM` or `CODE32`) must be removed or changed to `THUMB` in every case. As a consequence your assembly file will then require updating for the differences between the Thumb and ARM instruction sets.

If any `CODE16` directives are present, be aware that these will assemble without warning if changed to `THUMB`, however there might be subtle differences in behavior. This is because `CODE16` assembles according to the legacy Thumb-1 syntax rules but `THUMB` uses the new Unified Assembler Language (UAL) syntax. 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.

You might need to rewrite these portions of your code to cater for the programmer's model of the Cortex-M1. 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-M1 and that the supported modes are very different from an ARM7/9; any code which changes state or mode must be modified if appropriate or removed. Likewise, code which accesses coprocessors must be removed.

3.2 Retargeting for new peripherals

You must also perform additional steps to retarget the build on to the Cortex-M1, for example, defining the new memory map in the scatter file. This includes placement of structures over registers for the SCS (System Control Space), and the addition of 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 Section 2.8.2.

4 Debugging with the Cortex-M1

The Cortex-M1 provides debug capabilities. These include:

- BreakPoint Unit.
- A data watchpoint unit to provide support for watchpoints.

In addition, the structure of the processor means that debug accesses are performed through a DAP (Debug Access Port). 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.

RVDS 3.1 also includes an ISSM (Instruction Set System Model) of the Cortex-M1. This provides instruction-level simulation of the core, NVIC and other key features of the processor. It also models one UART peripheral and three additional timer peripherals.

5 Forward compatibility with Cortex-M3 processor

The Cortex-M1 processor implements a forward binary compatible subset of the instruction set and features provided by Cortex-M3 processor. Software, including system level code, can be easily moved from Cortex-M1 processors to Cortex-M3 processors providing increased performance and a simple migration path from FPGA to ASIC without the need for recompilation.

To ensure a smooth transition, ARM recommends that code designed to operate on both processor architectures obey the following rules and configure the Configuration Control Register (CCR) appropriately:

- Use word transfers only to access all registers in the NVIC and System Control Space (SCS)
- Treat all unused SCS registers and bit fields on Cortex-M1 as do-not-modify
- As soon as possible after reset, manually configure the following fields in the CCR:
    - STKALIGN bit to one
    - UNALIGN_TRP bit to one
    - Leave all other bits in the CCR register as their original value.