

Application Note **234**

Migrating from PIC Microcontrollers to Cortex™-M3

Document number: ARM DAI 0234

Issued: February 2010

Copyright ARM Limited 2010



Application Note 234

Migrating from PIC Microcontrollers to Cortex-M3

Copyright © 2010 ARM Limited. All rights reserved.

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
February 2010	A	First release

Proprietary notice

Words and logos marked with ® or © are registered trademarks or trademarks owned by ARM Limited, 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.

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	Introduction	4
1.1	Why change to Cortex-M3.....	4
1.2	Cortex-M3 products	5
1.3	References and Further Reading	5
2	Cortex-M3 Features	6
2.1	Nested Vectored Interrupt Controller (NVIC).....	6
2.2	Memory Protection Unit (MPU)	6
2.3	Debug Access Port (DAP)	6
2.4	Memory Map	6
3	PIC and Cortex-M3 Compared	7
3.1	Programmer's model	8
3.2	System control and configuration registers.....	10
3.3	Exceptions and interrupts	10
3.4	Memory	12
3.5	Debug	16
3.6	Power management.....	17
4	Migrating a software application.....	18
4.1	General considerations.....	18
4.2	Tools configuration	21
4.3	Startup	21
4.4	Interrupt handling.....	22
4.5	Timing and delays.....	23
4.6	Peripherals.....	23
4.7	Power Management.....	23
4.8	C Programming	23
5	Examples	25
5.1	Vector tables and exception handlers.....	25
5.2	Bit banding.....	26
5.3	Access to peripherals	27

1 Introduction

The ARM Cortex™-M3 is a high performance, low cost and low power 32-bit RISC processor. The Cortex-M3 processor supports the Thumb-2 instruction set – a mixed 16/32-bit architecture giving 32-bit performance with 16-bit code density. The Cortex-M3 processor is based on the ARM v7-M architecture and has an efficient Harvard 3-stage pipeline core. It also features hardware divide and low-latency ISR (Interrupt Service Routine) entry and exit.

As well as the CPU core, the Cortex-M3 processor includes a number of other components. These included a Nested Vectored Interrupt Controller (NVIC), an optional Memory Protection Unit (MPU), Timer, debug and trace ports. The Cortex-M3 has an architectural memory map.

In this document, we will refer to standard features of the PIC18 or PIC24 architecture. There are several extended versions of the architecture (e.g. dsPIC, PIC32 etc.) which support additional features (e.g. extra status flags, ability to address more memory etc.).

This document should be read in conjunction with Application Note 179 “Cortex-M3 Embedded Software Development”. That document describes many standard features and coding techniques for Cortex-M3 developers.

1.1 Why change to Cortex-M3

There are many reasons to take the decision to base a new design on a device incorporating a Cortex-M3 processor. Most, if not all, of these reasons also apply to the decision to migrate an existing product to Cortex-M3.

- **Higher performance**
While exact performance is dependent on the individual device and implementation, the Cortex-M3 processor is capable of providing 1.25DMIPS/MHz at clock speeds up to 135MHz.
- **More memory**
Since the Cortex-M3 is a full 32-bit processor, including 32-bit address and data buses, it has a 4GB address space. Within the fixed address space, up to 2GB of this is available for code execution (either in flash or RAM) and up to 2GB for RAM (either on or off chip). Significant space is also allocated for peripherals, system control registers and debug support.
- **Modern tools**
The Cortex-M3 is well supported by a wide range of tools from many suppliers. In particular, the RealView Developer Suite (RVDS) and Keil Microcontroller Developer Kit (MDK) from ARM provide full support for Cortex-M3. Models are also available to accelerate software development.
- **Can program in C**
Unlike many microcontrollers, the Cortex-M3 can be programmed entirely in C. This includes exception handling, reset and initialization as well as application software. Doing away with assembly code improves portability, maintainability and debugging and also encourages code reuse. Easier programming, improved reusability and greater availability of device libraries also reduces time-to-market.
- **More efficient interrupt handling**
The interrupt architecture of the Cortex-M3 is designed for efficient interrupt entry and exit and also to minimize interrupt latency. The integrated Nested Vectored Interrupt Controller supports hardware prioritization, pre-emption and dispatch of external and internal interrupts. The core also supports late arrival, tail-chaining and nesting with minimal software intervention.
- **Future proof**
The Cortex-M3 will meet the needs of the majority of today's microcontroller

applications but, crucially, it provides an upwards migration path to the rest of the ARM architecture family of products. Since programming is entirely in C, achieving extra performance by migrating to a higher class of ARM processor is realistic and achievable with minimal engineering effort. A single toolset also supports multiple Cortex-M architecture from multiple MC vendors.

- **Use more capable OS/scheduler**
The architecture of the Cortex-M3 provides excellent support for many standard RTOS's and schedulers. OS's can make use of the privileged "Handler" mode to provide inter-process isolation and protection. The built-in SysTick timer is ideal for system synchronization and can also function as a watchdog.
- **Better consistency between suppliers**
Using a microcontroller based on an industry-standard architecture reduces risk by ensuring that products available from different suppliers are highly consistent and standardized. The engineering effort involved in moving from one supplier to another is minimized.
- **Better debug facilities**
The Cortex-M3 supports full in-circuit debug using standard debug adapters. There is full support for breakpointing, single-stepping and program trace as well as standard instrumentation features.
- **More choices**
The Cortex-M3 architecture is implemented by many device manufacturers and supported by many tools vendors. This gives the developer significantly improved choice. The high degree of standardization across Cortex-M3 microcontrollers means that there is a large range of standard software components available.

1.2 Cortex-M3 products

See www.onarm.com for the most comprehensive list of available Cortex-M3 devices, supporting technology and development tools.

1.3 References and Further Reading

Application Note 179 – Cortex-M3 Embedded Software Development, ARM DAI0179B, ARM Ltd.

Cortex Microcontroller Software Interface Standard (see www.onarm.com).

PIC18F44J11 Datasheet, DS39932C, Microchip Technology Inc.

STM32F101T4 Datasheet, Doc ID 15058, STMicroelectronics

MPASM, MPLINK, MPLIB User's Guide, DS33014K, Microchip Technology Inc.

MPLAB C18 C Compiler User's Guide, DS51288J, Microchip Technology Inc.

STM32F10xxx Cortex-M3 Programming Manual, Doc ID 15491, STMicroelectronics.

Cortex-M3 User Guide Reference Material, ARM DUI0450A, ARM Ltd.

Cortex-M3 Technical Reference Manual, ARM DDI0337G, ARM Ltd.

2 Cortex-M3 Features

2.1 Nested Vectored Interrupt Controller (NVIC)

Depending on the silicon manufacturer's implementation, the NVIC can support up to 240 external interrupts with up to 256 different priority levels, which can be dynamically configured. 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.

The NVIC also incorporates a standard SysTick timer which can be used as a one-shot timer, repeating timer or system wake-up/watchdog timer.

A separate (optional) Wake-up Interrupt Controller (WIC) is also available. In low power modes, the rest of the chip can be powered down leaving only the WIC powered.

2.2 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 8 different regions which can be split into a further 8 sub-regions, each sub-region being one eighth the size of a region.

2.3 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-M3 implementation might contain either of these depending on the silicon manufacturer's implementation.

2.4 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 and is discussed further in section 3.4.1 below.;

3 PIC and Cortex-M3 Compared

Direct comparisons are necessarily difficult between these two architectures. Both are available in any number of different configurations. While the Cortex-M3 is arguably more standardized than the PIC implementations, there are still several implementation options available to individual silicon fabricators (e.g. number of interrupts and depth of priority scheme, memory protection, debug configuration etc.).

In both cases, there is a large set of devices with very different peripheral sets.

For the purposes of meaningful comparison, we have selected the PIC18 architecture and will be looking at devices like the PIC18F44J11.

On the Cortex-M3 side, we have selected the STM32F101T4 from STMicroelectronics. The features of these two devices are summarized in the table below.

	PIC18F44J11	STM32F101T4
Program memory (flash)	16 Kbytes	16 Kbytes
Data memory (RAM)	3.8 Kbytes	4 Kbytes
Max clock frequency	48 MHz	36 MHz
GPIO pins	34	26
ADC	13-channel x 10-bit	10-channel x 12-bit
Timers	2 x 8-bit, 3 x 16-bit	2 x 16-bit + SysTick
Watchdog timer	Y	Y (Two)
SPI	1	1
I2C	1	1
USART	2	2
PWM	2	N/A
Comparators	2	N/A
RTC	Y	Y
External interrupt sources	4 (+30 internal)	43 (+ 16 internal)
Interrupt prioritization	2 levels	16 levels
Vectored Interrupt Controller	N	Y
Power-saving modes	Idle/Sleep/DeepSleep	Sleep/Stop/Standby
DMA		7-channel
Debug port	ICD (In-Circuit Debug)	SWJ-DP JTAG port
Voltage Detection	Y	Y

While these two devices have been selected as they are similar in size, it is worth noting that the PIC is “large” in the family, whereas the Cortex-M3 is relatively “small” compared

to other available Cortex-M3 options. Both devices have reasonably small areas of ROM and RAM together with manageable peripheral sets.

Note that the constraints of pin count and packaging dictate that not all combinations of peripherals may be available simultaneously. This is especially true of the PIC device. The values in table indicate the maximum available set on the device.

3.1 Programmer's model

3.1.1 Register set

The two processors are quite significantly different with regards to the register set.

The PIC essentially treats internal data RAM as an extended general purpose register set and all locations support single-cycle access when using direct addressing (accessing registers outside the current bank requires an extra instruction to set the Bank Select Register first). Some of this is reserved for Special Function Registers which control system features. Most ALU instructions operate on the Working Register (W or WREG). Special registers are provided for holding pointers to access memory, stack pointer, ALU status etc.

There are many general purpose registers which are used for specific functions e.g. PRODH and PRODL which hold multiply result, FSRnH and FSRnL used for indirect memory access, STKPTR which holds the stack pointer, TOS which holds the current top-of-stack value, TBLPTR which is used for accessing program memory etc.

The fact that there is a large number of registers with special functions and the need for banked addressing means that the majority of code for PIC18 devices is written in C. Handling bank switching in assembler can be difficult.

The Cortex-M3 has 16 general purpose registers, R0-R15, all 32-bit. R0-R12 are generally available for essentially all instructions, R13 is used as the Stack Pointer, R14 as the Link Register (for subroutine and exception return) and R15 as the Program Counter. There is also a single Program Status Register (PSR) which holds current status (operating mode, ALU status etc.) – see below.

Peripheral and system control registers are memory-mapped within the System Control Space.

3.1.2 Status registers

The PIC STATUS register is an 8-bit register containing 5 ALU flags. Bits associated with interrupt status and masking are contained in separate registers.

The Cortex-M3 Program Status Register (PSR) is a single 32-bit register with several aliases, each providing a view of a different subset of the contents. From the user point of view, the Application Program Status Register (APSR) contains the ALU status flags. For operating system and exception handling use, the Interrupt Program Status Register (IPSR) contains the number of the currently executing interrupt (or zero if none is currently active). The Execution Program Status Register (EPSR) contains bits which reflect execution status and is not directly accessible.

3.1.3 Instruction set

There are several variations of the PIC instruction set. Older devices (not considered here) support 12-bit and 14-bit instruction sets. The PIC18 series support 16-bit instructions and is backwards compatible with PIC16. Although instructions are 16-bits, the ALU and memory interfaces are still 8-bit so these are regarded as 8-bit devices.

The basic PIC18F instruction set contains 77 instructions.

Later generations of PIC devices support full 16-bit operation (PIC24, dsPIC) and 32-bit operation (PIC32). The PIC18 series is one of the first for which a C compiler is available.

The Cortex-M3 supports the ARM v7-M architecture. The instruction set is a subset of the Thumb-2 instruction set, in which instructions are either 16-bits or 32-bits in size. The set contains 159 instructions (though some are functionally similar, differing only in the size and encoding).

3.1.4 Operating modes

Different operating modes, sometimes allied with the concept of “privilege”, are used by many embedded operating systems and schedulers to enforce task separation and to protect the system from rogue software.

The PIC has no concept of operating mode, nor of privilege.

The Cortex-M3 supports two modes, Thread mode (used for user processes) and Handler mode (used for handling exceptions and automatically entered when an exception is entered). Optionally, Thread mode can be configured to be “unprivileged” (sometimes called “user privilege”) and can then be prevented from carrying out certain operations. This configuration can be used to provide a degree of system protection from errant or malicious programs. At startup, Thread mode is configured to operate in privileged mode.

3.1.5 Stack

PIC devices support three stacks, two of which are not stored in “normal” memory space. These are implemented as hardware registers and this results in some limitations and can increase power consumption.

For return addresses, a 32-entry Full Ascending stack supports a maximum of 31 nested subroutines and/or interrupts. Applications cannot access this stack space directly but special registers provide access to the top word of the stack and to the stack pointer.

The instruction set allows PC to be pushed on to the stack via a PUSH instruction (the top word on the stack can then be modified via the special-purpose registers, providing a method for “push”-ing arbitrary values. The POP instruction discards the top entry. Stack underflow and overflow are automatically detected.

For interrupts, a single-entry stack is used for saving status registers on interrupt entry. Both high and low priority interrupts use the same space, so special care must be taken when pre-emption is enabled. There is an option for using this space for storing context across a function call but care must be taken that no interrupts can occur in the meantime.

For application software a separate stack can be defined, used for function parameters and automatic variables. This can be located in internal or external RAM. Locating it in external RAM requires larger pointers and can be less efficient.

In contrast, the Cortex-M3 uses only “normal” memory for the stack. The Cortex-M3 supports a Full Descending stack addressed by the current stack pointer (see below). This stack can be located anywhere in RAM. Typically, for best performance, it will be located in internal SRAM. Stack size is limited only by the available RAM space.

The Cortex-M3 stack pointer is typically initialized to the word above the top of the allocated stack area. Since the stack model is Full Descending, the stack pointer is decremented before the first store, thus placing the first word on the stack at the top of the allocated region.

All stack accesses on the Cortex-M3 are word-sized.

3.1.6 Code execution

During normal sequential code execution, the PIC PC increments by 2 bytes per instruction. The Cortex-M3 PC may increment by 2 or 4 bytes depending on the size of the current instruction.

The Cortex-M3 PC is 32-bits wide, held in a single register, and is able to address an instruction anywhere in the 4GB address space. The PIC PC is 21-bits wide and is held in 3 separate 8-bit registers PCU (3 bits), PCH (8 bits) and PCL (8 bits). Only PCL is directly writeable. Updates to the higher bytes (PCU and PCH) are via the shadow registers PCLATU and PCLATH. Any operation which writes to PCL (e.g. a computed GOTO operation) will simultaneously copy the values in these shadow registers to the corresponding portions of the PC (this does not apply to CALL, RCALL and GOTO instructions).

In both processors, the Program Counter (PC) is generally accessed only indirectly i.e. via call or jump instructions. However, both support limited indirect access to the PC to support, for instance, jump tables or computed calls.

The Cortex-M3 supports loading the PC from memory and also branching to value held in a general-purpose register (using the BX or BLX instructions).

Both processors enforce alignment requirements on instructions and this places attendant restrictions on the values which can be taken by the program counter. In both cases, instructions must appear on an even byte boundary so jumps and calls must be to an even address.

The least significant bit of the PIC PC is fixed to zero to enforce instruction alignment.

In most ARM architecture processors (including the Cortex-M3), the least significant bit of a value loaded or transferred into the PC is used to control a change in operating state. Since the Cortex-M3 supports only the Thumb-2 instruction set it must operate in Thumb state at all times (for details of ARM and Thumb states, refer to the ARM Architecture Reference Manual). The programmer's model and the tools (assembler, compiler, linker) will generally ensure that this is the case at all times during normal code execution. However, if the PC is loaded from memory, bit 0 of the loaded value is used to control the state of the processor. In the Cortex-M3, this bit must always be set to 1 (to remain in Thumb state). Loading a value into the PC which has the least significant bit clear will result in a Usage Fault. Again, the tools (compiler, assembler and linker) will usually ensure that the least significant bit is always set on any function pointer.

3.2 System control and configuration registers

In both devices, the system control and configuration registers are memory-mapped. In PIC devices, it is more useful though to regard this space as part of the register set since all can be accessed in single-cycle instructions as directly-addressable memory locations with 8-bit addresses which can be embedded in a single instruction. These are different from the instructions used to access external memory if any is implemented. Many of these registers are bit-addressable.

The Cortex-M3 system control space is fixed at 0xE0000000 and above. The placement of registers in this space is fixed. All can be directly addressed using 32-bit pointers and standard memory access instructions. Since these registers do not lie in the bit-band region, they are not bit-addressable.

3.3 Exceptions and interrupts

PIC supports a number of interrupt sources, the actual number being dependent on the peripheral set in the selected device. In general, these map to 3 external interrupt sources (INT0, INT1, INT2), timer interrupts (TMR0-3), and then further interrupts associated with peripheral devices (A/D converter, UART, SSP, PSP etc.). By default, there is no priority scheme and all interrupts have the same priority with no pre-emption. Optionally, each interrupt can be assigned one of two priority levels. High- and Low-Priority interrupts are then routed to two separate vectors and high-priority interrupts can pre-empt low-priority interrupts.

PIC Low-Priority Interrupts may nest if the interrupt handler explicitly re-enables interrupts (by setting the GIEL bit). Special care must be taken to preserve registers, variables and

other system resources to ensure that the handler is re-entrant. High-Priority interrupts may not nest since they use a single set of shadow registers to store context.

The Cortex-M3 has an integrated Nested Vectored Interrupt Controller which supports between 1 and 240 separate external interrupt courses. There are up to 256 priority levels, which may be configured as a hierarchy of priority group and sub-priority. Individual implementations may configure the number of interrupts and the number of priority levels which are supported so it is important to check the manual for the target device to determine the exact configuration. The Cortex-M3 also supports an external Non-Maskable Interrupt (NMI) and several internal interrupts (e.g. HardFault, SVC etc.).

Cortex-M3 interrupts of any priority may nest. There is no need to take any special steps to write a re-entrant handler (beyond the obvious requirements not to use global variables in a non-reentrant fashion) as the hardware automatically saves sufficient system context to make this unnecessary.

3.3.1 Interrupt prioritization and pre-emption

The PIC18F44J11 supports the interrupt priority feature, though this feature is disabled on reset for compatibility with earlier and smaller devices. Each interrupts may be assigned to either High or Low Priority. There are separate vectors for each priority, with all interrupts of each priority being routed to a single vector. INT0 has fixed high priority. High and low priority interrupts can be globally enabled/disabled via single bit.

The STM32F101T4 NVIC supports 68 interrupts and 16 priority levels. There is no implementation of priority grouping.

3.3.2 External interrupts

PIC18F44J11 supports three external interrupt sources (INT0-2). All external interrupts must be routed to one of these three sources. Software must then de-multiplex the interrupt sources in the handler.

STM32F101T4 supports up to 43 external interrupts, all of which may have separately configured priority, and a Non-Maskable Interrupt (NMI).

3.3.3 Internal interrupts

PIC supports a number of internal interrupts from peripherals included in the device. Since the peripheral set varies greatly from device to device, the exact set of supported interrupts also varies. A number are associated with internal error events such as Oscillator Fail (OSCF), Bus Collision (BCL), Low-Voltage Detection (LVD).

STM32F101T4 supports the standard set of Cortex-M3 internal interrupts, with the exception of MemManageException (since there is no MPU).

3.3.4 Vector table

The PIC vector table consists of three entries: Reset, High-Priority Interrupt and Low-Priority Interrupt. The vector table is located at the start of program memory, immediately following the reset vector. Each location contains executable instructions, typically branch instructions to the start of the interrupt handler. The vector table cannot be relocated.

The Cortex-M3 vector table is located, by default, at address 0x00000000. It can be relocated during initialization to a location in either Code or RAM regions.. Within the vector table, each entry contains the starting address of the corresponding handler routine. This is automatically loaded by the NVIC during interrupt execution and passed directly to the core.

3.3.5 Interrupt handlers

Interrupt handlers in the PIC architecture are responsible for preserving any registers which they corrupt. Only the status registers (STATUS, WREG and BSR) are

automatically saved. Handlers must return using RETFIE rather than a standard return instruction so must be flagged to the C compiler using #pragma keywords. High and low priority interrupt handlers must be separately flagged at compile-time so cannot be dynamically reassigned at runtime.

The Cortex-M3 supports all exception entry and exit sequences in hardware and thus allows interrupt routines to be standard C functions, compliant with the ARM Architecture Procedure Call Standard (AAPCS). Any compliant function can be installed in the vector table as a handler simply by referencing its address.

3.3.6 Interrupt optimizations

The PIC has fixed interrupt latency and does not support tail-chaining or late arrival of interrupts. Pre-emption is supported between the two available priority levels.

The Cortex-M3 supports tail-chaining and late arrival to reduce interrupt latency and minimize unnecessary context save and restore operations.

3.4 Memory

Both PIC and Cortex-M3 devices have a fixed memory map. Within these maps, areas are allocated for ROM, RAM, peripherals etc. Both also support a scheme of memory-mapped registers for system configuration and control.

Both devices support a Harvard memory architecture, with separate data and program memory interfaces. This provides for greater throughput by allowing simultaneous accesses at each of the two interfaces. Cortex-M3 supports a single, unified external address space, covering both program and data (including peripheral) regions. The PIC maintains a completely separate address space for program and data.

3.4.1 Memory map

PIC program and data memory are in separate address spaces and are addressed in different ways.

The table below shows the internal program memory map (the PIC18F44J11 does not support external program memory). Note that only 16k of the address space is populated with usable memory. Accesses to other regions will read as '0' (equivalent to a NOP instruction).

Address	Contents
0x0000	Reset Vector
0x0008	High-Priority Interrupt Vector
0x0018	Low-Priority Interrupt Vector
to 0x3FF7	On-Chip Program Memory
0x3FF8 to 0x3FFF	Configuration Words (CONFIG1 – CONFIG4)
0x4000 to 0x1FFFFFFF	Unimplemented memory (read as '0' - NOP)

Note the configuration words which are in a fixed location at the top of usable program memory. These words are read automatically by the device on reset and are used to set initial configuration. They are mainly concerned with initial configuration of clock and oscillator modes. Compiler (`#pragma config` directive) and assembly (`config` directive) tools provide mechanisms for setting these words.

PIC internal data memory is organized into up to 16 banks of 256 bytes, giving a total of 4k bytes. Banks are accessed either via a 12-bit address (using the MOVFF instruction) or via a 4-bit Bank Select Register (which specifies the bank number) coupled with an 8-bit address.

Additionally, the lower part of Bank 0 and the upper part of Bank 15 can be accessed directly (termed the “Access Bank”) bypassing the Bank Select Register. This allows for quicker access to some general purpose RAM locations and a number of the Special Purpose Registers in Bank 15. Access to the remaining Special Purpose Registers (in Bank 14 and the lower part of Bank 15) requires either using a full 12-bit address or setting the Bank Select Register.

The File Select Registers can be used to access memory indirectly using a full 12-bit address. These registers support auto-increment/decrement addressing modes.

The PIC18F44J11 data memory map is summarized in the table below. Of the 4k address space, approximately 3.8k is usable RAM, the remainder being allocated to Special Function Registers.

Bank	Bank Address	Full address	Contents
0	0x00-0x5F	0x000-0x05F	Access Bank General Purpose RAM
0	0x60-0xFF	0x060-0x0FF	Bank 0 General Purpose RAM
1	0x00-0xFF	0x100-0x1FF	Bank 1 General Purpose RAM
2-13	0x00-0xFF	0x200-0xDFF	Bank n General Purpose RAM
14	0x00-0xBF	0xE00-0xEBF	Bank 14 General Purpose RAM
14	0xC0-0xFF	0xEC0-0xEFF	Bank 14 Special Function Registers
15	0x00-0x5F	0xF00-0xF5F	Bank 15 Special Function Registers
15	0x60-0xFF	0xF60-0xFFFF	Access Bank Special Function Registers

Copying between program and data memory on PIC devices requires special operations since program memory cannot be directly accessed as data. The TBLPTR registers are provided for this purpose.

The banked memory structure limits the maximum memory size for applications and increases software overhead for data accesses. This makes PIC microcontrollers more suitable for small applications.

The Cortex-M3 memory map is summarized in the table below. This is a unified address space covering both program and data regions as well as peripherals and system control registers.

Address	Region	Address	Detail
0x0000 0000	Code	Code memory (e.g. flash, ROM etc.)	
0x2000 0000	SRAM	0x2000 0000 - 0x200F FFFF	Bit band region
		0x2200 0000 - 0x23FF FFFF	Bit band alias
0x4000 0000	Peripheral	0x4000 0000 - 0x400F FFFF	Bit band region
		0x4200 0000 - 0x43FF FFFF	Bit band alias
0x6000 0000 - 0x9FFF FFFF	External RAM	For external RAM	
0xA000 0000 - 0xDFFF FFFF	External device	External peripherals or shared memory	
0xE000 0000 - 0xE003 FFFF	Private Peripheral Bus – Internal	0xE000 0000 - 0xE000 2FFF	ITM, DWT, FPB
		0xE000 E000 - 0xE000 EFFF	System Control Space
0xE004 0000 - 0xE00F FFFF	Private Peripheral Bus – External	0xE004 0000 - 0xE004 1FFF	TPIU, ETM
		0xE004 2000 - 0xE00F EFFF	MPU, NVIC etc
0xE010 0000 to 0xFFFF FFFF	Vendor-specific	For vendor-specific use	

The memory map implemented in the STM32F101T4 is as follows. Regions which are not indicated are unimplemented.

Address	Region	Address	Detail
0x0000 0000 – 0x1FFF FFFF	Code	0x0000 0000 – 0x007F FFFF	Flash or system memory alias
		0x0800 0000 – 0x0801 EFFF	Flash memory
		0x1FFF F000 – 0x1FFF F7FF	System memory
		0x1FFF F800 – 0x1FFF F80F	Option bytes
0x2000 0000 – 0x201F FFFF	SRAM	0x2000 0000 – 0x2000 FFFF	SRAM (bit banded)
		0x2200 0000 – 0x201F FFFF	Bit band alias region for SRAM
0x4000 0000	Peripheral	0x4000 0000 – 0x4002 33FF	Peripherals (bit banded)
		0x4200 0000 – 0x43FF FFFF	Bit band alias for peripherals
0xE000 0000 – 0xE00F FFFF	Internal peripherals	0xE000 0000 – 0xE000 2FFF	ITM, DWT, FPB
		0xE000 E000 – 0xE000 EFFF	System Control Space
		0xE004 0000 – 0xE004 1FFF	TPIU, ETM
		0xE004 2000 – 0xE00F EFFF	SysTick, NVIC etc

The system may be configured (via hardware signals sensed at reset) to boot from the “System memory” region. This is typically used to hold a simple boot loader which is capable of downloading a program and programming the flash.

Since the amount of SRAM implemented on the STM32F101T4 is wholly contained within the bit band region, all SRAM on this device is bit-addressable. Likewise, all peripherals in the Peripheral region are bit-addressable.

3.4.2 Memory protection

PIC supports protection of program memory at block resolution. Block size varies from device to device but is otherwise fixed. This prevents reprogramming of the on-chip flash memory. There is no protection scheme for data memory.

The Cortex-M3 supports an optional Memory Protection Unit (MPU). When implemented, this allows access to memory to be partitioned into regions. Access to each region may then be restricted based on the current operating mode. This allows software developers to implement memory access schemes aimed at providing a degree of protection to the system from errant or malicious software applications.

When porting from PIC to Cortex-M3 there is no need to use any of the memory protection features offered by the Cortex-M3. At reset, the MPU is disabled and the default memory map as described above applies.

However, if the Cortex-M3 device incorporates caches and/or write buffers (these are not part of the architecture and, if present, are implemented externally) the cache and buffer policies are normally generated from the MPU configuration. In this case, it may be desirable for performance reasons to configure and enable the MPU.

The STM32F101T4 does not implement an MPU.

3.4.3 Access types and endianness

PIC instructions are little-endian. Since PIC data memory is 8-bits wide and is only accessed in bytes, endianness is not relevant.

The Cortex-M3 is a 32-bit processor and all internal registers are 32-bit. Memory transfers of 8-bit bytes, 16-bit halfwords and 32-bit words are supported. In the case of bytes and halfwords, the programmer needs to specify whether the loaded value is to be treated as signed or unsigned. In the case of signed values, the loaded value is sign-extended to create a 32-bit signed value in the destination register; in the case of unsigned values, the upper part of the register is cleared to zero.

The Cortex-M3 also has instructions which transfer doublewords and also Load and Store Multiple instructions which transfer multiple words in a single instruction to and from a contiguous block of memory.

Cortex-M3 instructions are always little-endian. Data memory accesses default to little-endian but the processor can be configured to access data in a big-endian format via a configuration pin which is sampled on reset. It is not possible to change endianness following reset. Note that registers in the System Control Space and accesses to any system peripherals are always little-endian regardless of the configuration.

3.4.4 Bit banding

All internal RAM contents on PIC devices can be bit-addressed. Bit Set, Bit Clear, Bit Test and Bit Toggle instructions support this addressing mode.

The Cortex-M3 provides bit access to two 1MB regions of memory, one within the internal SRAM region and the other in the peripheral region. A further 32MB of address space is reserved for this purpose and each word within these regions aliases to a specific bit within the corresponding bit-band region. Reading from the alias region returns a word containing the value of the corresponding bit; writing to bit 0 of a word in the alias region results in an atomic read-modify-write of the corresponding bit within the bit-band region.

3.5 Debug

On the PIC, limited debug functionality for on-board devices is available via the MPLAB ICD 3 unit. This communicates with the target device via a standard connector and allows simple breakpoint, single-step, data watch etc.

More complex debug facilities are available via standard header boards which connect to the target hardware as a replacement for the PIC device. Debug on the PIC device requires a small monitor program which executes on the target. This does not consume resources while the device is running but does occupy some program memory space. When debugging, it will also require some stack space.

Cortex-M3 devices are debugged via a standard JTAG or Serial-Wire Debug (SWD) connector. A simple, standardized external connector is required to interface to the host system.

In addition, the uVision simulator from Keil and the MPLAB IDE provide software simulation of target devices. In the case of uVision, this can include simulating external components at board level.

Both devices provide support for program debug and trace, though different external hardware may be required to support trace when using the PIC device. In the case of uVision, full instruction trace can be captured using the Keil ULINK-Pro trace analyzer (or a third-party equivalent).

3.6 Power management

The PIC devices supports RUN, IDLE and SLEEP modes.

In RUN mode, the processor is fully-clocked though it is possible to select a number of different clock speeds (Primary, Secondary, Internal).

In IDLE mode, the main processor clock is turned off while peripherals remain clocked (with the same options for clock source and speed).

In SLEEP mode both peripherals and processor are powered down and unclocked.

Sleep modes are entered on execution of the SLEEP instruction (with the exact mode selected via configuration bits). Exit from sleep modes is via a watchdog timeout, interrupt or reset.

Architecturally, the Cortex-M3 supports Sleep and Deep Sleep modes. The manner in which these modes are supported on an actual device and the power-savings which are possible in each is dependent on the device. Sleep modes and power-saving features can be further extended by individual microcontroller vendors by using additional control registers.

The STM device supports three power-saving configurations: Sleep, Stop and Standby.

In Sleep mode (corresponding to the architectural Sleep mode), the processor is stopped (though state is retained completely) while peripherals continue to operate. Interrupts or other external events cause the processor to restart.

Stop mode (corresponding to the architectural Deep Sleep mode) achieves lowest possible power consumption while retaining the state of SRAM and registers. All external clocks are stopped. The device is woken by an external interrupt, low voltage detector or RTC alarm.

In Standby mode, everything except the internal watchdog, RTC and Wakeup Interrupt Controller is powered down. Register state and RAM contents will be lost. Exit from Standby mode is via external reset, watchdog reset, external WKUP pin or RTC alarm.

Sleep mode can be entered in several ways:

- **Sleep-now**
The Wait-For-Interrupt (WFI) or Wait-For-Event (WFE) instructions cause the processor to enter Sleep mode immediately. Exit is on detection of an interrupt or debug event.
- **Sleep-on-exit**
Setting the SLEEPONEXIT bit within the System Control Register (SCR) causes the processor to enter Sleep mode when the last pending ISR has exited. In this case, the exception context is left on the stack so that the exception which wakes the processor can be processed immediately.

In addition, Deep Sleep mode can be entered by setting the SLEEPDEEP bit in the SCR. On entry to Sleep mode, if this bit is set, the processor indicates to the external system that deeper sleep is possible.

4 Migrating a software application

The majority of software on both PIC18 and STM32 devices will be written in high-level languages, almost all in C. We will therefore ignore any migration relating to changes in instruction set as this can be handled by a recompilation.

We must also recognize that the “size” of PIC devices often requires a distinctive style of software which does not lend itself easily to porting to another architecture.

This section is written with reference to the C18 C compiler from Microchip and the Keil Microcontroller Developer Kit from ARM. Other tools may differ from these in significant ways.

4.1 General considerations

4.1.1 Operating mode

The Cortex-M3 will reset into Thread mode, executing as privileged. Handler mode (always privileged) is automatically entered when handling any exceptions which occur.

Since PIC does not support more than multiple operating modes and has no concept of privilege, leaving the Cortex-M3 in this configuration is the simplest option and is often sufficient.

In order to take advantage of the protection offered by the privileged execution Thread mode can be configured to be unprivileged by setting CONTROL[0]. Unprivileged execution is prohibited from carrying out some system operations, e.g. masking interrupts.

4.1.2 Stack configuration

On PIC devices, the return stack is automatically initialized and is not accessible to the application software. If an application stack is required (and it almost always will be when programming in C) it is initialized via a STACK directive in the linker control script. Allocating a stack region of up to 256 bytes is simple since it can be located within a single bank of memory. C18 supports stacks larger than this by combining more than one contiguous memory bank into a single region. However, this can mean that space available for data variables is limited.

The Cortex-M3 takes the initial value for the Main Stack Pointer (SP_main) from the first word in the vector table. This must be initialized to an area of RAM. Ideally this should be internal SRAM for best performance. Unless configured otherwise (see below) the Cortex-M3 will use this single stack pointer in both Thread and Handler modes. This is the simplest configuration to use when migrating from the 8051 which only supports a single stack pointer.

For applications which require an OS, the Cortex-M3 can be configured to use the separate Process Stack Pointer (SP_process) when in Thread Mode. This is done by writing to the CONTROL[1] bit. Setting this configuration allows separate stacks to be used for normal execution and exception handling. This would normally be handled by an OS kernel – in most simple applications, there is no need to use the PSP.

Since the Cortex-M3 is programmed entirely in C, stack usage is likely to be much higher than for the same program running on PIC. Sufficient stack space must therefore be provided. When allocating stack space, ensure that you take account of any usage required by exceptions.

4.1.3 Memory map

Unless an MPU is present and enabled, the default memory map described above is used. When migrating an application from PIC it is not usually necessary to implement any memory map configuration. In this case the MPU can be safely left disabled.

Microcontrollers using a Cortex-M3 processor can be built with many different memory devices. Usually, there will be some internal Flash or ROM (mapped to the CODE region) and internal SRAM (in the SRAM region). Any peripherals will normally be mapped to the Peripheral region. There may also be some external RAM.

Consult the manual for your chosen device to determine exactly what memories have been implemented and how they are mapped.

In any event, the system control registers and standard core peripherals (such as the SysTick timer etc) will be located in the standard location.

4.1.4 Code and data placement

1. Code

When coding for PIC devices, it is common to write non-relocatable code. Indeed, the assembler produces absolute executable files by default (ARM compilers and assemblers do not do this – a link stage is always required). Directives in C and assembly language source files fix the placement in memory at compile-time. This is extremely rare when coding for Cortex-M3 devices. Essentially, all code and object files are relocatable and the placement of code and data is decided at link time.

Both linkers take a list of object files and a script which controls the code and data placement.

Since PIC devices differ considerably in the amount and type of memory which they support and also the location of special registers, the PIC linker (MPLINK) control scripts are different for every PIC device. The generic script then needs to be modified by the developer to add application-specific placement rules.

When writing code containing GOTO or CALL instructions, unless page selection instructions are also included in the source, it will be necessary to split these up so that the maximum range of the instructions is not exceeded. This is not necessary when coding for ARM since the linker will automatically add long-branch support code where necessary.

The ARM linker takes its control input from a “scatter control file” – this can be generated automatically from project setting if using the Keil-MDK development tools. Specifying explicit sections is not usually necessary when linking for Cortex-M3 devices. Providing information about available ROM sections is often sufficient to allow the linker to place all objects. Code is normally placed in the Code region and this is normally populated with non-volatile memory of some kind e.g. flash. It is possible to place code in the SRAM region at run-time but performance will be slightly degraded as the core is optimized to fetch instructions using the ICODE bus from the Code region.

2. Data

Data memory in PIC devices is divided into banks. It is difficult, therefore, to create data segments which are larger than a single bank.

The PIC linker tries to place all variables in a single 256-byte section. Arrays larger than 256 bytes need special sections creating in the linker script.

There are no such restrictions when coding for Cortex-M3 devices. The only restriction is the absolute size of the RAM devices available.

3. Peripherals

All PIC peripherals which are included in the device are accessed and controller via SFRs in fixed locations (though the location and layout may vary from device to device). Additional memory-mapped peripherals may only be used with larger devices which support external memory interfaces. are not generally used so do not need locations defining at link time. The locations of these registers are generally supplied via included

header files when coding in assembly or C and mirrored in the standard linker configuration script for the target device.

In contrast, Cortex-M3 devices have a small set of architectural peripherals (e.g. the SysTick timer, the NVIC etc) which are accessed via registers in the System Control Space. The location of this is fixed. Other memory-mapped peripherals can be located absolutely at compile-time (via fixed addresses in source code) but this is not recommended. Instead, they are usually defined in relocatable data segments which are then placed by the linker at absolute addresses – these addresses are supplied to the linker in the scatter control file.

4.1.5 Data types and alignment

When programming in a high-level language, the natural data type of the underlying machine is not necessarily important. The C compiler will take care of mapping high-level data types to low-level arithmetic operations and storage units.

Both compilers support a variety of types, as listed in the table.

Type	Cortex-M3	PIC	Notes
char	8-bit signed	8-bit signed	Char is signed by default in both architectures
short	16-bit	16-bit	
int	32-bit	16-bit	
short long	N/A	24-bit	
long	32-bit	32-bit	
long long	64-bit	N/A	No 64-bit type for PIC
float	32-bit	32-bit	
double	64-bit	32-bit	
long double	64-bit	N/A	

The Cortex-M3 is a 32-bit architecture and, as such, handles 32-bit types very efficiently. In particular, 8-bit and 16-bit types are less efficiently manipulated, although they will save on data memory if this is an issue.

Cortex-M3 devices, in common with other ARM architecture devices, normally require that data be aligned on natural boundaries. For instance, 32-bit objects should be aligned on 32-bit (word) boundaries and so on. The core is capable of supporting accesses to unaligned types (and this is useful when porting from architectures which do not have such strict alignment requirements) but there is a small speed penalty associated with this (since the memory interface is required to make multiple word accesses).

Since data memory is always accessed as bytes, PIC devices have no such restrictions.

4.1.6 Storage classes

1. Static

PIC compilers support the ability to share storage for automatic variables between functions. These are placed in static RAM locations and initialized on each entry to the function. The compiler will attempt to overlay storage for these functions, provided that they can never be active at the same time. Such functions cannot be re-entrant.

Function parameters can also be declared as static and will be allocated fixed storage in data memory. Again, such functions cannot be re-entrant.

The advantage in each case is the potential for smaller code (due to the ease of access to the variables) and smaller data footprint (due to the possibility of overlaying data segments).

While ARM compilers support the static keyword, it is not applicable to parameters and the concept of overlaying data objects is not supported.

2. Banked

Due to the banked nature of the data memory in the PIC architecture, data items can be declared as near or far. Near items are located within the Access RAM area and can therefore be accessed directly with an 8-bit address. Items declared as far are in banked memory and will therefore require the Bank Selection Register to be set in order to access them.

Near/far can also be applied to program memory objects. Far objects are located at addresses above 64k. in the context of the PIC18F46 device, this is not relevant as the maximum size of program memory is less than this threshold.

In Cortex-M3 devices, there is no concept of near/far for either program or data objects since all pointers can address all of available memory.

3. RAM/ROM location

Since PIC devices enforce strict isolation between program and data address spaces, data which is to be placed in ROM must be explicitly marked using the rom qualifier. Similarly, pointers to data objects in program memory must be marked using the same keyword.

This introduces complications when using standard library functions to access, for instance, string constants held in program memory (the default). There are several versions of functions like strcpy, depending on whether source and data strings are located in program or data memory. Care must be taken to use the correct variant.

Since the Cortex-M3 address space is unified, this technique is not necessary.

4.2 Tools configuration

When using larger PIC devices, addressing extended program and data memory becomes problematic since pointers are 16-bits in size. Therefore, in addition to the “near” and “far” qualifiers for objects and pointers, PIC compilers, support various memory configurations when building applications. “Small” models limit pointers to 16-bits, while “large” models allow 24-bit pointers. There comes with a code size and speed penalty.

When building for Cortex-M3, there is no need for this since all pointers are 32-bits and can address the entire memory map.

4.3 Startup

PIC18 devices begin execution at the reset vector, which is located at address 0x0000. The return stack pointer is automatically initialized to the bottom of the internal stack region. The software stack pointer, if one is used, is initialized via the linker control file (see 4.1.2).

Before starting execution, the PIC will load a set of system configuration values from a fixed location in program memory (the exact location varies from device to device – the default location will be specified in the template linker script for the device). These values are set via #pragma statements in C or CONFIG directives in assembler language source files.

The PIC startup code automatically initializes the C runtime environment before calling `main()`.

Cortex-M3 devices take the initial value for the Main Stack Pointer from the first word of the vector table (at address 0x00000000) and then begin execution by jumping to the address contained in the reset vector (at address 0x00000004).

Note that Cortex-M3 devices do not support any mechanism for auto-configuration (as described above for PIC) so all components will be in their reset state as described in the manual.

The C startup code will initialize the runtime environment and then call `main()`.

4.4 Interrupt handling

One significant difference between the two architectures is the balance between hardware and software in interrupt dispatch.

PIC18 devices (with the priority scheme enabled) have two interrupt vectors. All high-priority interrupts are assigned to one, all low priority interrupts to the other. Each interrupt handler is responsible for determining which of the possible sources has raised an interrupt (the Interrupt Flag bits must be individually checked to work this out). Once the source has been identified, a specific handler can be involved to handle the event.

The NVIC on Cortex-M3 devices handles all this automatically. All interrupt sources have separate vectors defined in the vector table. In parallel with saving context on the stack, the NVIC reads the relevant vector address from the table and directs program execution directly to the start of the correct handler.

1. Writing interrupt handlers

Interrupt handler routines for PIC devices must be identified in C source code using a `#pragma` directive.

When writing interrupt handlers in PIC assembler, there are constraints (e.g. cannot e.g. access arrays using calculated index, call other functions, perform complex math, access ROM variables etc). These must be followed for correct operation.

If the interrupt priority scheme is not used only one handler is required; two are required if the priority scheme is used.

Cortex-M3 interrupt handlers are standard C functions and there are no special C coding rules for them.

2. Vector table generation

The interrupt handlers must be installed by putting a GOTO instruction at the vector location. Typically this is done with a short section of inline assembler within the C application. If the priority scheme is used, two vectors are required.

On Cortex-M3 devices, the vector table is typically defined by an array of C function pointers. This is then located at the correct address by the linker.

3. Interrupt configuration

In order to enable a PIC interrupt, the following must be set correctly:

- Specific Interrupt Enable bit
- Global Interrupt Enable bit
- Global High/Low Priority Interrupt Enable bit (if priority is used)
- Peripheral Interrupt Enable bit (for peripheral interrupts)

On Cortex-M3 devices, the NVIC must be configured appropriately:

- Interrupt Set Enable register
- PRIMASK must be clear to enable interrupts

This is achieved using CMSIS intrinsic functions `__enable_irq()` and `NVIC_EnableIRQ()`. See the CMSIS documentation for more detail. CMSIS functions are also available for setting and clearing pending interrupts, checking the current active interrupt and configuring priority.

4.5 Timing and delays

It is common, when programming for PIC devices, to use NOP instructions as a way of consuming time. The execution time of NOP instructions is easily determined from the system clock speed. When developing for Cortex-M3 devices this cannot be relied on as the pipeline is free to “fold out” NOP instructions from the instruction stream. When this happens, they do not consume time at all.

Deterministic delays in Cortex-M3 systems must therefore make use of a timer peripheral.

4.6 Peripherals

4.6.1 Standard peripherals

On the Cortex-M3 it is usual to define a structure covering the System Control Space. All of the system control registers are located in this region. The standard peripherals (NVIC, SysTick etc.) are all controlled via registers in this area.

4.6.2 Custom peripherals

Custom peripherals on Cortex-M3 microcontrollers are generally memory-mapped within the defined peripheral region in the memory map. The usual method of access to these is to define C structures which describe the relevant registers. These can then be located at absolute addresses at link time via the scatter control file.

4.7 Power Management

ANSI C cannot generate the WFI and WFE instructions directly. Instead, you should use the CMSIS intrinsic functions `__WFE()` and `__WFI()` in your source code. If suitable device-driver functions are provided by the vendor, then these should be used in preference.

4.8 C Programming

- Don't bother with static parameters or overlays. Cortex-M3 does not suffer from data RAM size constraints like PIC.
- Unless memory size is a constraint, don't bother with small data types. They are less efficient on Cortex-M3.
- There is no need to declare objects as “near” or “far”.
- There is no need to specify which “bank” variables are located in.
- No need to designate interrupt handlers using any special keywords. Though it is regarded as good programming practice to declare interrupt handlers using the `__irq` keyword when using the ARM/Keil tools to develop for Cortex-M3.
- It is unlikely that you will encounter any data alignment problems but the `__packed` keyword should be used to resolve any which do arise.

- There is no need to specify any particular memory model.
- Any inline assembler in your PIC source will need to be re-written. Typically it should be rewritten in C rather than translated to inline ARM assembler – this will be easier and more portable.
- Any #pragma directives will need to be either removed or replaced. Those associated with placement code or data in C source code must be removed. Those marking interrupt handlers should also be removed and optionally the functions marked with __irq instead.
- There is no facility in Cortex-M3 for specifying any initial conditions as with the PIC configuration words which are automatically loaded on reset. Any code associated with this should be removed. All setup of Cortex-M3 core and peripherals needs to be performed in software following reset.

5 Examples

5.1 Vector tables and exception handlers

5.1.1 In assembler

The examples below show definition of vector tables and placeholders for exception handlers when writing in assembly code. Note that it is possible to avoid C startup for Cortex-M3 systems completely.

PIC

```
; include standard device header file
#include p18f452.inc

; the following code will be located at
; 0x0000 - reset vector
org 0x0000
goto Main ; jump to main entry point

; the following code will be located at
; 0x00088 - high priority interrupt vector
org 0x00088
goto int_hi ; jump to handler

; the following code will be located at
; 0x0018 - low priority interrupt vector
org 0x0018
goto int_lo ; jump to handler

Main

; write your main program here.
;
;

int_hi

;
; write your high priority
; interrupt service routine here

    retfie ;use retfie to return

int_lo

;
; write your low priority
; interrupt service routine here.
    retfie ;use retfie to return

end
```

Cortex-M3

```
; Vector Table Mapped to Address 0 at Reset
AREA    RESET, DATA, READONLY
EXPORT  __Vectors

__Vectors
DCD     __initial_sp      ; Initial SP
DCD     Reset_Handler    ; Reset Handler
DCD     NMI_Handler      ; NMI Handler
DCD     HardFault_Handler
DCD     MemManage_Handler
DCD     BusFault_Handler
DCD     UsageFault_Handler
DCD     0, 0, 0, 0       ; Reserved
DCD     SVC_Handler
DCD     DebugMon_Handler
DCD     0                 ; Reserved
DCD     PendSV_Handler
DCD     SysTick_Handler

; External Interrupts from here
DCD     InterruptHandler0
DCD     InterruptHandler1
DCD     InterruptHandler2

; etc

AREA    |.text|, CODE, READONLY
; Reset Handler

Reset_Handler    PROC
    EXPORT  Reset_Handler
    IMPORT  __main
    LDR     R0, =__main
    BX      R0
    ENDP

InterruptHandler0
; write your IRQ0 handler here
;
    BX     lr

END
```

5.1.2 In C

These two examples show how the same is achieved when coding in C.

PIC

```
#include <p18cxxx.h>
void low_isr(void);
void high_isr(void);

/*
 * This will be located at the low-priority
 * interrupt vector at 0x0018.
 */
#pragma code low_vector=0x18
void interrupt_at_low_vector(void)
{
    _asm
        GOTO low_isr
    _endasm
}

/* return to the default code section
 * and declare interrupt handler
 */
#pragma code
#pragma interruptlow low_isr
void low_isr (void)
{
    /* write handler here */
}

/*
 * This will be located at the hi-priority
 * interrupt vector at 0x0018.
 */
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
    _asm
        GOTO high_isr
    _endasm
}

/* return to the default code section
 * and declare interrupt handler
 */
#pragma interrupt high_isr
void high_isr (void)
{
    /* write handler here */
}
```

Cortex-M3

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

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

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

    /* Configurable interrupts from here */
    InterruptHandler0,
    InterruptHandler1,
    InterruptHandler2
    /*
     * etc.
     */
};

/* One example exception handler */
#pragma arm section
void SysTickHandler(void)
{
    printf("---- SysTick Interrupt ----");
}

In Scatter Control File:

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

5.2 Bit banding

As described above, both devices support bit access to certain areas of memory. In both cases, bit accesses are atomic.

The PIC supports this through a direct bit addressing mode in many instructions. Individual bits within many of the Special Function Registers and within the internal RAM memory can be addressed like this. Instructions are implemented to set, clear, test and toggle individual bits.

Cortex-M3 devices support bit access via a different method entirely. Within, for example, the SRAM region of the memory map, 1MB is designated as the “bit band region”. A second 32MB region, called the bit band alias region, is used to access the bits within the bit band region. Bit 0 of each word in the alias region is mapped within the memory system to a single bit within the bit band region. Bits can be read and written. Reading or writing any bit other than bit 0 in a word in the alias region has no effect.

A simple formula converts from bit address to aliased word address.

```
word_addr = bit_band_base
           + (byte_offset x 32)
           + (bit_number x 4)
```

C macros can then be easily defined to automate this process. For example

```
#define BITBAND_SRAM(a,b) ((BITBAND_SRAM_BASE \
                           + (a - BITBAND_SRAM_REF) * 32 \
                           + (b * 4)))
```

A similar macro can be defined for the peripheral region.

Individual bits can then be accessed using sequences like this.

```
#define MAILBOX    0x20004000
#define MBX_B7     *((volatile unsigned int *) \
                   (BITBAND_SRAM(MAILBOX,7)))

a = MBX_B7;
```

5.3 Access to peripherals

There are device-specific header files for all available PIC devices. These are included with most, if not all, development tools for PIC. These header files define all registers and systems constants e.g. available memory size etc.

Similarly, header files are usually provided for Cortex-M3 devices. You can obtain these either from the device supplier or use those included with many development tools. Keil MDK-ARM includes header files for most common devices.

Developers for Cortex-M3 platforms should be aware of the Cortex Microcontroller Software Interface Standard (CMSIS). This defines a standard software application interface for many standard peripherals (e.g. SysTick, NVIC) and system functions (e.g. enable/disable interrupts) on Cortex-M3 platforms. This covers the set of standard peripherals and core functions. Most Cortex-M3 device manufacturers supply additional CMSIS-compliant header files which provide definitions for all device-specific functions and peripherals.