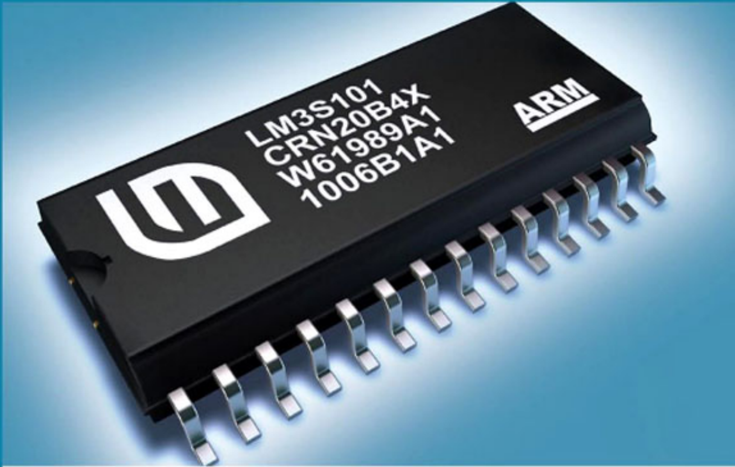
EMBEDDED TECHNOLOGY™ SERIES



The Definitive Guide to the ARM Cortex-M3

Joseph Yiu

Newnes

The Definitive Guide to the ARM Cortex-M3

This page intentionally left blank

The Definitive Guide to the ARM Cortex-M3

Joseph Yiu

AMSTERDAM • BOSTON • HEIDELBERG • LONDON • NEW YORK
OXFORD • PARIS • SAN DIEGO • SAN FRANCISCO
SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier

Newnes is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Library of Congress Cataloging-in-Publication Data

(Application submitted.)

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-0-7506-8534-4

For information on all Newnes publications
visit our Web site at www.books.elsevier.com

08 09 10 11 12 13 10 9 8 7 6 5 4 3 2 1

Typeset by Charon Tec Ltd (A Macmillan Company), Chennai, India
www.charontec.com

Printed in The United States of America

Table of Contents

Foreword

Microcontroller programmers are, by their nature, truly resourceful beings. They take a fixed design and create fantastic new products by implementing the microcontroller in a very unique way. Constantly, they demand highly efficient computing from the most frugal of system designs. The primary ingredient used to perform this alchemy is the tool chain environment, and it is for this reason that engineers from ARM's own tool chain division joined forces with CPU designers to form a team that would rationalize, simplify, and improve upon the ARM7TDMI processor design.

The result of this combination, the ARM Cortex-M3, represents an exciting development to the original ARM architecture. The device blends the best features from the 32-bit ARM architecture with the highly successful Thumb-2 instruction set design whilst adding several new capabilities. Despite these changes, the Cortex-M3 retains a simplified programmer's model that will be easily recognizable to all existing ARM aficionados.

—Wayne Lyons

Director of Embedded Solutions, ARM

Preface

This book is for both hardware and software engineers who are interested in the Cortex-M3 processor from ARM. The *Cortex-M3 Technical Reference Manual* (TRM) and the *ARMv7-M Architecture Application Level Reference Manual* already provide lots of information on this new processor, but they are very detailed and can be challenging for new starters to read.

This book is intended to be a lighter read for programmers, embedded product designers, System-on-a-Chip (SoC) engineers, electronics enthusiasts, academic researchers, and others with some experience of microcontrollers or microprocessors who are investigating the Cortex-M3 processor. The text includes an introduction to the new architecture, an instruction set summary, examples of some instructions, information on hardware features, and an overview of the processor's advanced debug system. It also provides application examples, including basic steps in software development for the Cortex-M3 processor using ARM tools as well as the GNU tool chain. This book is also targeted to those engineers who are familiar with the ARM7TDMI processor and who are migrating to the Cortex-M3 processor, because it covers the differences between the processors, and the porting of application software from the ARM7TDMI to the Cortex-M3.

Acknowledgments

I would like to thank the following people for reviewing this book or for providing me with their advice and feedback:

Alan Tringham, Dan Brook, David Brash, Haydn Povey, Gary Campbell, Kevin McDermott, Richard Earnshaw, Samin Ishtiaq, Shyam Sadasivan, Simon Axford, Simon Craske, Simon Smith, Stephen Theobald and Wayne Lyons.

I would also like to thank CodeSourcery for their technical support, and Luminary Micro for providing images for the book cover, and of course, the staff at Elsevier for their professional work towards the publication of this book.

Finally, a special thank-you to Peter Cole and Ivan Yardley for inspiring me to write this book.

Terms and Abbreviations

Abbreviation	*Meaning*
ADK	AMBA Design Kit
AHB	Advanced High-Performance Bus
AHB-AP	AHB Access Port
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ARM ARM	ARM Architecture Reference Manual
ASIC	Application Specific Integrated Circuit
ATB	Advanced Trace Bus
BE8	Byte Invariant Big Endian Mode
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DAP	Debug Access Port
DSP	Digital Signal Processor/Digital Signal Processing
DWT	Data WatchPoint and Trace
ETM	Embedded Trace Macrocell
FPB	Flash Patch and Breakpoint
FSR	Fault Status Register
HTM	CoreSight AHB Trace Macrocell
ICE	In-Circuit Emulator
IDE	Integrated Development Environment
IRQ	Interrupt Request (normally refers to external interrupts)
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
ITM	Instrumentation Trace Macrocell
JTAG	Joint Test Action Group (a standard of test/debug interfaces)
JTAG-DP	JTAG Debug Port
LR	Link Register
LSB	Least Significant Bit
LSU	Load/Store Unit
MCU	Microcontroller Unit

MMU	Memory Management Unit
MPU	Memory Protection Unit
MSB	Most Significant Bit
MSP	Main Stack Pointer
NMI	Nonmaskable Interrupt
NVIC	Nested Vectored Interrupt Controller
OS	Operating System
PC	Program Counter
PSP	Process Stack Pointer
PPB	Private Peripheral Bus
PSR	Program Status Register
SCS	System Control Space
SIMD	Single Instruction, Multiple Data
SP, MSP, PSP	Stack Pointer, Main Stack Pointer, Process Stack Pointer
SoC	System-on-a-Chip
SP	Stack Pointer
SW	Serial-Wire
SW-DP	Serial-Wire Debug Port
SWJ-DP	Serial-Wire JTAG Debug Port
SWV	Serial-Wire Viewer (an operation mode of TPIU)
TPA	Trace Port Analyzer
TPIU	Trace Port Interface Unit
TRM	Technical Reference Manual

Conventions

Various typographical conventions have been used in this book, as follows:

- Normal assembly program codes:

  ```
  MOV   R0, R1  ; Move data from Register R1 to Register R0
  ```

- Assembly code in generalized syntax; items inside < > must be replaced by read register names:

  ```
  MRS  <reg>, <special_reg>  ;
  ```

- C program codes:

  ```
  for (i=0;i<3;i++) { func1(); }
  ```

- Pseudo code:

  ```
  if (a > b) { ...
  ```

- Values:

  1. 4'hC , 0x123 are both hexadecimal values
  2. *#3* indicates item number 3 (e.g., IRQ #3 means IRQ number 3)
  3. *#immed_12* refers to 12-bit immediate data
  4. Register bits
     Typically used to illustrate a part of a value based on bit position. For example, bit[15:12] means bit number 15 down to 12.

- Register access types:

  1. R is Read only
  2. W is Write only
  3. R/W is Read or Write accessible
  4. R/Wc is Readable and clear by a Write access

References

Ref No.	Document
1	*Cortex-M3 Technical Reference Manual (TRM)* downloadable from the ARM documentation Web site at www.arm.com/documentation/ARMProcessor_Cores/index.html
2	*ARMv7-M Architecture Application Level Reference Manual* downloadable from the ARM documentation Web site at www.arm.com/products/CPUs/ARM_Cortex-M3_v7.html
3	*CoreSight Technology System Design Guide* downloadable from the ARM documentation Web site at www.arm.com/documentation/Trace_Debug/index.html
4	*AMBA Specification* downloadable from the ARM documentation Web site at www.arm.com/products/solutions/AMBA_Spec.html
5	*AAPCS Procedure Call Standard for the ARM Architecture* downloadable from the ARM documentation Web site at www.arm.com/pdfs/aapcs.pdf
6	*RVCT 3.0 Compiler and Library Guide* downloadable from the ARM documentation Web site at www.arm.com/pdfs/DUI0205G_rvct_compiler_and_libraries_guide.pdf
7	*ARM Application Note 179: Cortex-M3 Embedded Software Development* downloadable from the ARM documentation Web site at www.arm.com/documentation/Application_Notes/index.html

This page intentionally left blank

CHAPTER 1

Introduction

In This Chapter:

- What Is the ARM Cortex-M3 Processor?
- Background of ARM and ARM Architecture
- Instruction Set Development
- The Thumb-2 Instruction Set Architecture (ISA)
- Cortex-M3 Processor Applications
- Organization of This Book
- Further Readings

What Is the ARM Cortex-M3 Processor?

The microcontroller market is vast, with over 20 billion devices per year estimated to be shipped in 2010. A bewildering array of vendors, devices, and architectures are competing in this market. The requirement for higher-performance microcontrollers has been driven globally by the industry's changing needs; for example, microcontrollers are required to handle more work without increasing a product's frequency or power. In addition, microcontrollers are becoming increasingly connected, whether by Universal Serial Bus (USB), Ethernet, or Wireless Radio, and hence the processing needed to support these communications channels and advanced peripherals is growing. Similarly, general application complexity is on the increase, driven by more sophisticated user interfaces, multimedia requirements, system speed, and convergence of functionalities.

The ARM Cortex-M3 processor, the first of the Cortex generation of processors released by ARM in 2006, was primarily designed to target the 32-bit microcontroller market. The Cortex-M3 processor provides excellent performance at low gate count and comes with many new features previously available only in high-end processors. The Cortex-M3 addresses the requirements for the 32-bit embedded processor market in the following ways:

- Greater performance efficiency, allowing more work to be done without increasing the frequency or power requirements

- Low power consumption, enabling longer battery life, especially critical in portable products including wireless networking applications
- Enhanced determinism, guaranteeing that critical tasks and interrupts are serviced as quickly as possible but in a known number of cycles
- Improved code density, ensuring that code fits in even the smallest memory footprints
- Ease of use, providing easier programmability and debugging for the growing number of 8-bit and 16-bit users migrating to 32-bit
- Lower-cost solutions, reducing 32-bit-based system costs close to those of legacy 8-bit and 16-bit devices and enabling low-end, 32-bit microcontrollers to be priced at less than US$1 for the first time
- Wide choice of development tools, from low-cost or free compilers to full-featured development suites from many development tool vendors

Microcontrollers based on the Cortex-M3 processor already compete head-on with devices based on a wide variety of other architectures. Designers are increasingly looking at reducing the system cost, as opposed to the traditional device cost. As such, organizations are implementing *device aggregation,* whereby a single, more powerful device can potentially replace three or four traditional 8-bit devices.

Other cost savings can be achieved by improving the amount of code reuse across all systems. Since Cortex-M3 processor-based microcontrollers can be easily programmed using the C language and are based on a well-established architecture, application code can be ported and reused easily, reducing development time and testing costs.

It is worthwhile highlighting that the Cortex-M3 processor is not the first ARM processor to be used to create generic microcontrollers. The venerable ARM7 processor has been very successful in this market, with partners such as NXP (Philips), Texas Instruments, Atmel, OKI, and many other vendors delivering robust 32-bit Microcontroller Units (MCUs). The ARM7 is the most widely used 32-bit embedded processor in history, with over 1 billion processors produced each year in a huge variety of electronic products, from mobile phones to cars.

The Cortex-M3 processor builds on the success of the ARM7 processor to deliver devices that are significantly easier to program and debug and yet deliver a higher processing capability. Additionally, the Cortex-M3 processor introduces a number of features and technologies that meet the specific requirements of the microcontroller applications, such as nonmaskable interrupts for critical tasks, highly deterministic nested vector interrupts, atomic bit manipulation, and an optional memory protection unit. These factors make the Cortex-M3 processor attractive to existing ARM processor users as well as many new users considering use of 32-bit MCUs in their products.

The Cortex-M3 Processor vs Cortex-M3-Based MCUs

The Cortex-M3 processor is the central processing unit (CPU) of a microcontroller chip. In addition, a number of other components are required for the whole Cortex-M3 processor-based microcontroller. After chip manufacturers license the Cortex-M3 processor, they can put the Cortex-M3 processor in their silicon designs, adding memory, peripherals, input/output (I/O), and other features. Cortex-M3 processor-based chips from different manufacturers will have different memory sizes, types, peripherals, and features. This book focuses on the architecture of the processor core. For details about the rest of the chip, please check the particular chip manufacturer's documentation.

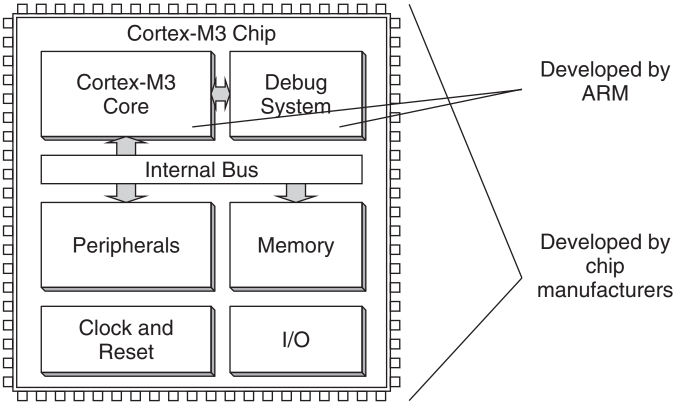


Figure 1.1 The Cortex-M3 Processor vs the Cortex-M3-Based MCU

Background of ARM and ARM Architecture

A Brief History

To help you understand the variations of ARM processors and architecture versions, let's look at a little bit of ARM history.

ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology. In 1991, ARM introduced the ARM6 processor family, and VLSI became the initial licensee. Subsequently, additional companies, including Texas Instruments, NEC, Sharp, and ST Microelectronics, licensed the

ARM processor designs, extending the applications of ARM processors into mobile phones, computer hard disks, personal digital assistants (PDAs), home entertainment systems, and many other consumer products.

Nowadays ARM partners ship in excess of 2 billion ARM processors each year. Unlike many semiconductor companies, ARM does not manufacture processors or sell the chips directly. Instead, ARM licenses the processor designs to business partners, including a majority of the world's leading semiconductor companies. Based on the ARM low-cost and power-efficient processor designs, these partners create their processors, microcontrollers, and system-on-chip solutions. This business model is commonly called *intellectual property (IP) licensing*.

In addition to processor designs, ARM also licenses systems-level IP and various software IP. To support these products, ARM has developed a strong base of development tools, hardware, and software products to enable partners to develop their own products.

Architecture Versions

Over the years, ARM has continued to develop new processors and system blocks. These include the popular ARM7TDMI processor and, more recently, the ARM1176TZ(F)-S processor, which is used in high-end applications such as smart phones. The evolution of features and enhancements to the processors over time has led to successive versions of the ARM architecture. Note that architecture version numbers are independent from processor names. For example, the ARM7TDMI processor is based on the ARMv4T architecture (the *T* is for *Thumb* instruction mode support).

The ARMv5E architecture was introduced with the ARM9E processor families, including the ARM926E-S and ARM946E-S processors. This architecture added "Enhanced" Digital Signal Processing (DSP) instructions for multimedia applications.

With the arrival of the ARM11 processor family, the architecture was extended to the ARMv6. New features in this architecture included memory system features and Single Instruction–Multiple Data (SIMD) instructions. Processors based on the ARMv6 architecture include the ARM1136J(F)-S, the ARM1156T2(F)-S, and the ARM1176JZ(F)-S.

Following the introduction of the ARM11 family, it was decided that many of the new technologies, such as the optimized Thumb-2 instruction set, were just as applicable to the lower-cost markets of microcontroller and automotive components. It was also decided that although the architecture needed to be consistent from the lowest MCU to the highest-performance application processor, there was a need to deliver processor architectures that best fit applications, enabling very deterministic and low gate count processors for cost-sensitive markets and feature-rich and high-performance ones for high-end applications.

Over the past several years, ARM extended its product portfolio by diversifying its CPU development, which resulted in the architecture version 7, or v7. In this version, the architecture design is divided into three profiles:

- The *A profile,* designed for high-performance open application platforms
- The *R profile,* designed for high-end embedded systems in which real-time performance is needed
- The *M profile,* designed for deeply embedded microcontroller-type systems

Let's look at these profiles in a bit more detail:

- A Profile (ARMv7-A): Application processors required to run complex applications such as high-end embedded operating systems (OSs), such as Symbian, Linux, and Windows Embedded, requiring the highest processing power, virtual memory system support with Memory Management Units (MMUs), and, optionally, enhanced Java support and a secure program execution environment. Example products include high-end mobile phones and electronic wallets for financial transactions.
- R Profile (ARMv7-R): Real-time, high-performance processors targeted primarily at the higher end of the real-time[1] market—those applications, such as high-end breaking systems and hard drive controllers, in which high processing power and high reliability are essential and for which low latency is important.
- M Profile (ARMv7-M): Processors targeting low-cost applications in which processing efficiency is important and cost, power consumption, low interrupt latency, and ease of use are critical, as well as industrial control applications, including real-time control systems.

The Cortex processor families are the first products developed on architecture v7, and the Cortex-M3 processor is based on one profile of the v7 architecture, called ARM v7-M, an architecture specification for microcontroller products.

This book focuses on the Cortex-M3 processor, but it is only one of the Cortex product family that uses the ARMv7 architecture. Other Cortex family processors include the Cortex-A8 (application processor), which is based on the ARMv7-A profile, and the Cortex-R4 (real-time processor), based on the ARMv7-R profile.

[1] There is always great debate as to whether we can have a "real-time" system using general processors. By definition, "real time" means that the system can get a response within a guaranteed period. In an ARM processor-based system, you may or may not able to get this response due to choice of operating system, interrupt latency, or memory latency, as well as if the CPU is running a higher-priority interrupt.

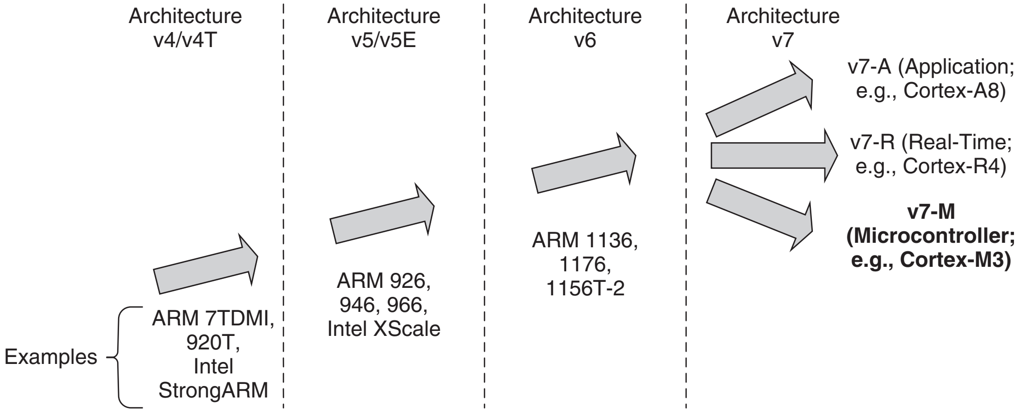

Architecture v4/v4T
Architecture v5/v5E
Architecture v6
Architecture v7
v7-A (Application; e.g., Cortex-A8)
v7-R (Real-Time; e.g., Cortex-R4)
v7-M (Microcontroller; e.g., Cortex-M3)
ARM 1136, 1176, 1156T-2
ARM 926, 946, 966, Intel XScale
Examples
ARM 7TDMI, 920T, Intel StrongARM

Figure 1.2 The Evolution of ARM Processor Architecture

The details of the ARMv7-M architecture are documented in *The ARMv7-M Architecture Application Level Reference Manual* (Ref 2). This document can be obtained via the ARM Web site through a simple registration process. The ARMv7-M architecture contains the following key areas:

- Programmer's model
- Instruction set
- Memory model
- Debug architecture

Processor-specific information, such as interface details and timing, is documented in the Cortex-M3 *Technical Reference Manual* (*TRM*) (Ref 1). This manual can be accessed freely on the ARM Web site. The Cortex-M3 *TRM* also covers a number of implementation details not covered by the architecture specifications, such as the list of supported instructions, because some of the instructions covered in the ARMv7-M architecture specification are optional on ARMv7-M devices.

Processor Naming

Traditionally, ARM used a numbering scheme to name processors. In the early days (the 1990s), suffixes were also used to indicate features on the processors. For example, with the ARM7TDMI processor, the *T* indicates Thumb instruction support, *D* indicates JTAG debugging, *M* indicates fast multiplier, and *I* indicates an embedded ICE module. Subsequently it was decided that these features should become standard features of future ARM processors; therefore, these suffixes are no longer added to the new processor family names. Instead, variations on memory interface, cache, and Tightly Coupled Memory (TCM) have created a new scheme for processor naming.

For example, ARM processors with cache and MMUs are now given the suffix "26" or "36," whereas processors with Memory Protection Units (MPUs) are given the suffix "46" (e.g., ARM946E-S). In addition, other suffixes are added to indicate synthesizable[2] (*S*) and Jazelle (*J*) technology. Table 1.1 presents a summary of processor names.

Table 1.1 ARM Processor Names

Processor Name	Architecture Version	Memory Management Features	Other Features
ARM7TDMI	ARMv4T		
ARM7TDMI-S	ARMv4T		
ARM7EJ-S	ARMv5E		DSP, Jazelle
ARM920T	ARMv4T	MMU	
ARM922T	ARMv4T	MMU	
ARM926EJ-S	ARMv5E	MMU	DSP, Jazelle
ARM946E-S	ARMv5E	MPU	DSP
ARM966E-S	ARMv5E		DSP
ARM968E-S	ARMv5E		DMA, DSP
ARM966HS	ARMv5E	MPU (optional)	DSP
ARM1020E	ARMv5E	MMU	DSP
ARM1022E	ARMv5E	MMU	DSP
ARM1026EJ-S	ARMv5E	MMU or MPU	DSP, Jazelle
ARM1136J(F)-S	ARMv6	MMU	DSP, Jazelle
ARM1176JZ(F)-S	ARMv6	MMU + TrustZone	DSP, Jazelle
ARM11 MPCore	ARMv6	MMU + multiprocessor cache support	DSP, Jazelle
ARM1156T2(F)-S	ARMv6	MPU	DSP
Cortex-M3	ARMv7-M	MPU (optional)	NVIC
Cortex-R4	ARMv7-R	MPU	DSP
Cortex-R4F	ARMv7-R	MPU	DSP + Floating point
Cortex-A8	ARMv7-A	MMU + TrustZone	DSP, Jazelle

With version 7 of the architecture, ARM has migrated away from these complex numbering schemes that needed to be decoded, moving to a consistent naming for families of processors, with Cortex its initial brand. In addition to illustrating the compatibility across processors, this

[2] A synthesizable core design is available in the form of a hardware description language (HDL) such as Verilog or VHDL and can be converted into a design netlist using synthesis software.

system removes confusion between architectural version and processor family number; for example, the ARM7TDMI is not a v7 processor but was based on the v4T architecture.

Instruction Set Development

Enhancement and extension of instruction sets used by the ARM processors has been one of the key driving forces of the architecture's evolution.

Historically (since ARM7TDMI), two different instruction sets are supported on the ARM processor: the ARM instructions that are 32-bit and Thumb instructions that are 16-bit. During program execution, the processor can be dynamically switched between the ARM state or the Thumb state to use either one of the instruction sets. The Thumb instruction set provides only a subset of the ARM instructions, but it can provide higher code density. It is useful for products with tight memory requirements.

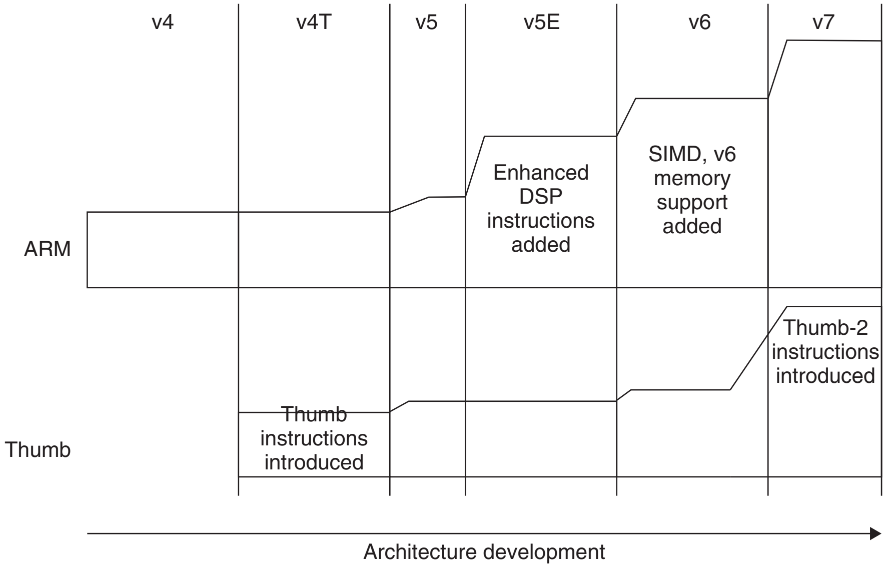

v4
v4T
v5
v5E
v6
v7
ARM
Enhanced DSP instructions added
SIMD, v6 memory support added
Thumb
Thumb instructions introduced
Thumb-2 instructions introduced
Architecture development

Figure 1.3 Instruction Set Enhancement

As the architecture version has been updated, extra instructions have been added to both ARM instructions and the Thumb instructions. Appendix II provides some information on the change of Thumb instructions during the architecture enhancements. In 2003, ARM announced the Thumb-2 instruction set, which is a new superset of Thumb instructions that contains both 16-bit and 32-bit instructions.

The details of the instruction set are provided in a document called *The ARM Architecture Reference Manual* (also known as the *ARM ARM*). This manual has been updated for the

ARMv5 architecture, the ARMv6 architecture, and the ARMv7 architecture. For the ARMv7 architecture, due to its growth into different profiles, the specification is also split into different documents. For Cortex-M3 developers, the *ARM v7-M Architecture Application Level Reference Manual* (Ref 2) covers all the required instruction set details.

The Thumb-2 Instruction Set Architecture (ISA)

The Thumb-2[3] ISA is a highly efficient and powerful instruction set that delivers significant benefits in terms of ease of use, code size, and performance. The Thumb-2 instruction set is a superset of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions. It allows more complex operations to be carried out in the Thumb state, thus allowing higher efficiency by reducing the number of states switching between ARM state and Thumb state.

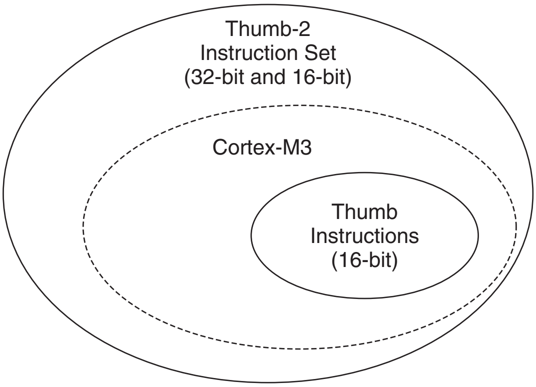


Figure 1.4 The Relationship Between the Thumb-2 Instruction Set and the Thumb Instruction Set

Focused on small memory system devices such as microcontrollers and reducing the size of the processor, the Cortex-M3 supports only the Thumb-2 (and traditional Thumb) instruction set. Instead of using ARM instructions for some operations, as in traditional ARM processors, it uses the Thumb-2 instruction set for all operations. As a result, the Cortex-M3 processor is not backward compatible with traditional ARM processors. That is, you cannot run a binary image for ARM7 processors on the Cortex-M3 processor. Nevertheless, the Cortex-M3 processor can execute almost all the 16-bit Thumb instructions, including all 16-bit Thumb instructions supported on ARM7 family processors, making application porting easy.

[3] Thumb and Thumb-2 are registered trademarks of ARM.

With support for both 16-bit and 32-bit instructions in the Thumb-2 instructions set, there is no need to switch the processor between Thumb state (16-bit instructions) and ARM state (32-bit instructions). For example, in ARM7 or ARM9 family processors, you might need to switch to ARM state if you want to carry out complex calculations or a large number of conditional operations and good performance is needed; whereas in the Cortex-M3 processor, you can mix 32-bit instructions with 16-bit instructions without switching state, getting high code density and high performance with no extra complexity.

The Thumb-2 instruction set is a very important feature of the ARMv7 architecture. Compared with the instructions supported on ARM7 family processors (ARMv4T architecture), the Cortex-M3 processor instruction set has a large number of new features. For the first time, hardware divide instruction is available on an ARM processor, and a number of multiply instructions are also available on the Cortex-M3 processor to improve data-crunching performance. The Cortex-M3 processor also supports unaligned data accesses, a feature previously available only in high-end processors.

Cortex-M3 Processor Applications

With its high performance and high code density and small silicon footprint, the Cortex-M3 processor is ideal for a wide variety of applications:

- Low-cost microcontrollers: The Cortex-M3 processor is ideally suited for low-cost microcontrollers, which are commonly used in consumer products, from toys to electrical appliances. It is a highly competitive market due to the many well-known 8-bit and 16-bit microcontroller products on the market. Its lower power, high performance, and ease-of-use advantages enable embedded developers to migrate to 32-bit systems and develop products with the ARM architecture.
- Automotive: Another ideal application for the Cortex-M3 processor is in the automotive industry. The Cortex-M3 processor has very high-performance efficiency and low interrupt latency, allowing it to be used in real-time systems. The Cortex-M3 processor supports up to 240 external vectored interrupts, with a built-in interrupt controller with nested interrupt supports and an optional memory protection unit, making it ideal for highly integrated and cost-sensitive automotive applications.
- Data communications: The processor's low power and high efficiency, coupled with Thumb-2 instructions for bit-field manipulation, make the Cortex-M3 ideal for many communications applications, such as Bluetooth and ZigBee.
- Industrial control: In industrial control applications, simplicity, fast response, and reliability are key factors. Again, the Cortex-M3 processor's interrupt feature, low interrupt latency, and enhanced fault-handling features make it a strong candidate in this area.

- Consumer products: In many consumer products, a high-performance microprocessor (or several of them) is used. The Cortex-M3 processor, being a small processor, is highly efficient and low in power and supports an MPU enabling complex software to execute while providing robust memory protection.

There are already many Cortex-M3 processor-based products available in the market, including low-end products priced as low as US$1, making the cost of ARM microcontrollers comparable to or lower than that of many 8-bit microcontrollers.

Organization of This Book

This book contains a general overview of the Cortex-M3 processor, with the rest of the contents divided into a number of sections:

Chapters 1 and 2, Introduction and Overview of the Cortex-M3
Chapters 3–6, Cortex-M3 Basics
Chapters 7–9, Exceptions and Interrupts
Chapters 10 and 11, Cortex-M3 Programming
Chapters 12–14, Cortex-M3 Hardware Features
Chapters 15 and 16, Debug Supports in Cortex-M3
Chapters 17–20, Application Development with Cortex-M3
Appendixes

Further Readings

This book does not contain all the technical details on the Cortex-M3 processor. It is intended to be a starter guide for people who are new to the Cortex-M3 processor and a supplemental reference for people using Cortex-M3 processor-based microcontrollers. To get further detail on the Cortex-M3 processor, the following documents, available from ARM (www.arm.com) and ARM partner Web sites, should cover most necessary details:

- *The Cortex-M3 Technical Reference Manual* (*TRM*) (Ref 1) provides detailed information about the processor, including programmer's model, memory map, and instruction timing.
- *The ARMv7-M Architecture Application Level Reference Manual* (Ref 2) contains detailed information about the instruction set and the memory model.
- Refer to datasheets for the Cortex-M3 processor-based microcontroller products; visit the manufacturer Web site for the datasheets on the Cortex-M3 processor-based product you plan to use.

- Refer to AMBA Specification 2.0 (Ref 4) for more detail regarding internal AMBA interface bus protocol details.
- C programming tips for Cortex-M3 can be found in the *ARM Application Note 179: Cortex-M3 Embedded Software Development* (Ref 7).

This book assumes that you already have some knowledge of and experience with embedded programming, preferably using ARM processors. If you are a manager or a student who wants to learn the basics without spending too much time reading the whole book or the *TRM,* Chapter 2 of this book is a good one to read, since it provides a summary on the Cortex-M3 processor.

CHAPTER 2

Overview of the Cortex-M3

In This Chapter:

- Fundamentals
- Registers
- Operation Modes
- The Built-In Nested Vectored Interrupt Controller
- The Memory Map
- The Bus Interface
- The Memory Protection Unit
- The Instruction Set
- Interrupts and Exceptions
- Debugging Support
- Characteristics Summary

Fundamentals

The Cortex-M3 is a 32-bit microprocessor. It has a 32-bit data path, a 32-bit register bank, and 32-bit memory interfaces. The processor has a Harvard architecture, which means it has a separate instruction bus and data bus. This allows instructions and data accesses to take place at the same time, and as a result of this the processor performance increases because data accesses do not affect the instruction pipeline. This feature results in multiple bus interfaces on Cortex-M3, each with optimized usage and the ability to be used simultaneously. However, the instruction and data buses share the same memory space (a unified memory system). In other words, you cannot get 8 GB of memory space just because you have separate bus interfaces.

For complex applications that require more memory system features, the Cortex-M3 processor has an optional MPU, and it is possible to use an external cache if it's required. Both little endian and big endian memory systems are supported.

The Cortex-M3 processor includes a number of fixed internal debugging components. These components provide debugging operation supports and features such as breakpoints and

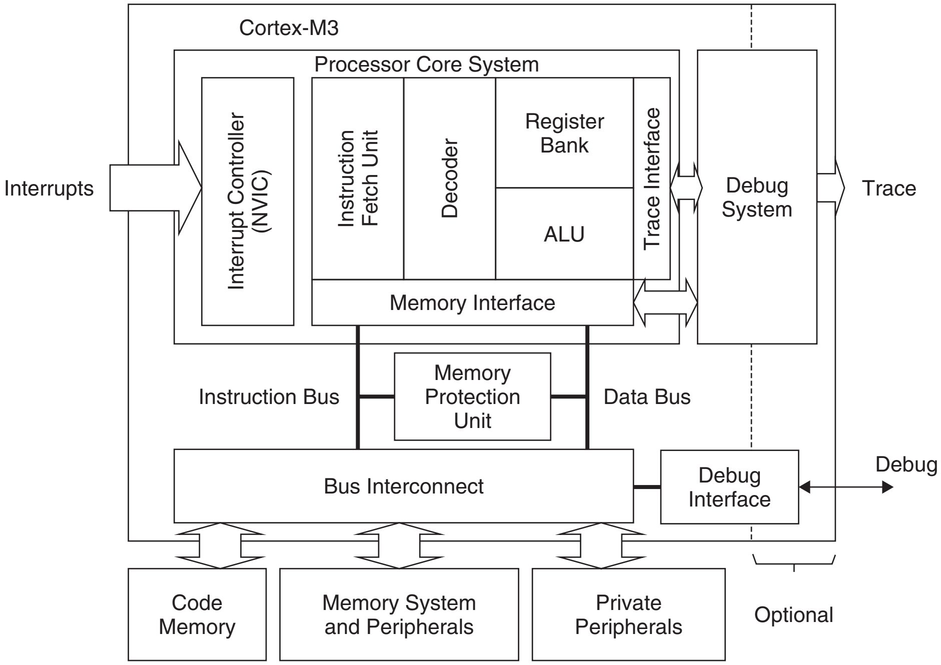


Figure 2.1 A Simplified View of the Cortex-M3

watchpoints. In addition, optional components provide debugging features such as instruction trace and various types of debugging interfaces.

Registers

The Cortex-M3 processor has registers R0 to R15. R13 (the stack pointer) is banked, with only one copy of the R13 visible at a time.

R0 to R12: General-Purpose Registers

R0 to R12 are 32-bit general-purpose registers for data operations. Some 16-bit Thumb instructions can only access a subset of these registers (low registers, R0 to R7).

R13: Stack Pointers

The Cortex-M3 contains two stack pointers, R13. They are banked so that only one is visible at a time:

- Main Stack Pointer (MSP): The default stack pointer; used by the OS kernel and exception handlers
- Process Stack Pointer (PSP): Used by user application code

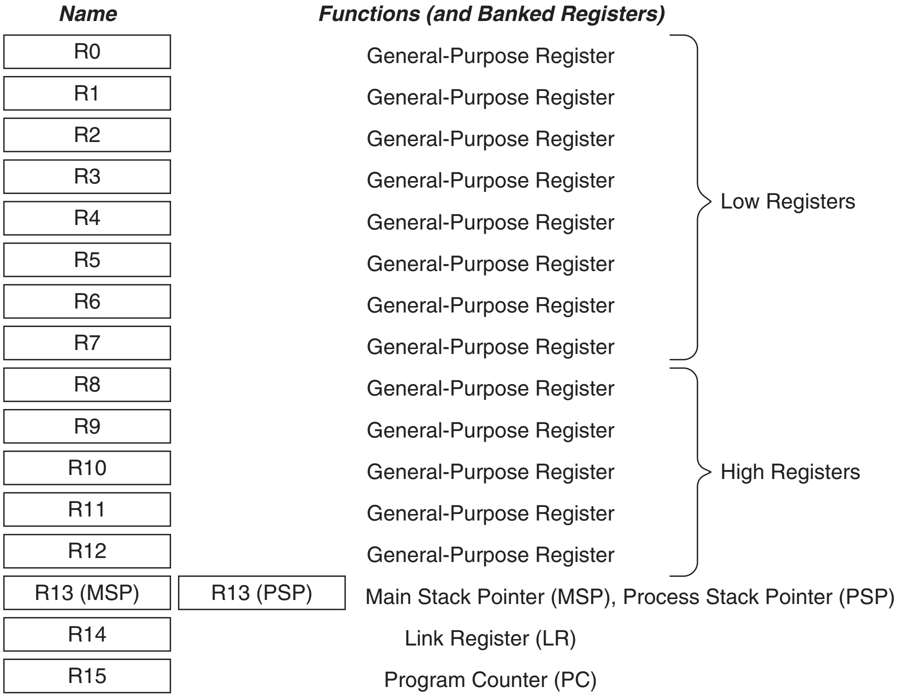


Figure 2.2 Registers in the Cortex-M3

The lowest two bits of the stack pointers are always 0, which means they are always word aligned.

R14: The Link Register

When a subroutine is called, the return address is stored in the link register.

R15: The Program Counter

The program counter is the current program address. This register can be written to control the program flow.

Special Registers

The Cortex-M3 processor also has a number of special registers:

- Program Status Registers (PSRs)
- Interrupt Mask Registers (PRIMASK, FAULTMASK, BASEPRI)
- Control Register (CONTROL)

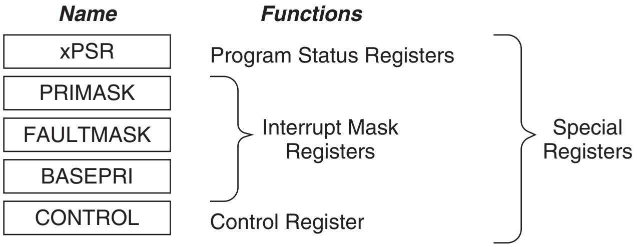


Figure 2.3 Special Registers in the Cortex-M3

These registers have special functions and can be accessed only by special instructions. They cannot be used for normal data processing.

Table 2.1 Registers and Their Functions

Register	Function
xPSR	Provide ALU flags (zero flag, carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and HardFault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

You'll find more information on these registers in Chapter 3.

Operation Modes

The Cortex-M3 processor has two modes and two privilege levels. The *operation modes* (thread mode and handler mode) determine whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler. The *privilege levels* (privileged level and user level) provide a mechanism for safeguarding memory accesses to critical regions as well as providing a basic security model.

	Privileged	***User***
When running an exception	Handle Mode	■
When running main program	Thread Mode	Thread Mode

Figure 2.4 Operation Modes and Privilege Levels in Cortex-M3

When the processor is running a main program (Thread mode), it can be in either a privileged state or a user state, but exception handlers can only be in a privileged state. When the processor exits reset, it is in Thread mode, with privileged access rights. In the privileged state, a program has access to all memory ranges (except when prohibited by MPU settings) and can use all supported instructions.

Software in the privileged access level can switch the program into the user access level using the control register. When an exception takes place, the processor will always switch back to the privileged state and return to the previous state when exiting the exception handler. A user program cannot change back to the privileged state by writing to the Control register. It has to go through an exception handler that programs the control register to switch the processor back into the privileged access level when returning to Thread mode.

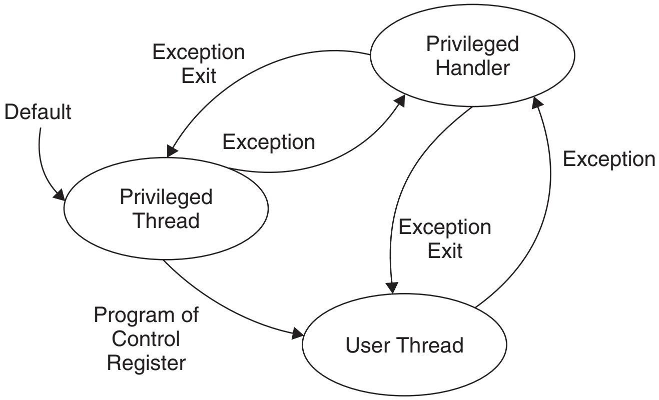


Figure 2.5 Allowed Operation Mode Transitions

The separation of privilege and user levels improves system reliability by preventing system configuration registers from being accessed or changed by some untrusted programs. If an MPU is available, it can be used in conjunction with privilege levels to protect critical memory locations such as programs and data for operating systems.

For example, with privileged accesses, usually used by the OS kernel, all memory locations can be accessed (unless prohibited by MPU setup). When the OS launches a user application, it is likely to be executed in the user access level to protect the system from failing due to a crash of untrusted user programs.

The Built-In Nested Vectored Interrupt Controller

The Cortex-M3 processor includes an interrupt controller called the *Nested Vectored Interrupt Controller (NVIC)*. It is closely coupled to the processor core and provides a number of features:

- Nested interrupt support
- Vectored interrupt support

- Dynamic priority changes support
- Reduction of interrupt latency
- Interrupt masking

Nested Interrupt Support

The NVIC provides nested interrupt support. All the external interrupts and most of the system exceptions can be programmed to different priority levels. When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level. If the priority of the new interrupt is higher than the current level, the interrupt handler of the new interrupt will override the current running task.

Vectored Interrupt Support

The Cortex-M3 processor has vectored interrupt support. When an interrupt is accepted, the starting address of the interrupt service routine (ISR) is located from a vector table in memory. There is no need to use software to determine and branch to the starting address of the ISR. Thus it takes less time to process the interrupt request.

Dynamic Priority Changes Support

Priority levels of interrupts can be changed by software during run time. Interrupts that are being serviced are blocked from further activation until the interrupt service routine is completed, so their priority can be changed without risk of accidental reentry.

Reduction of Interrupt Latency

The Cortex-M3 processor also includes a number of advanced features to lower the interrupt latency. These include automatic saving and restoring some register contents, reducing delay in switching from one ISR to another (see the discussion of tail chaining interrupts on page 152), and handling late arrival interrupts (see page 153.)

Interrupt Masking

Interrupts and system exceptions can be masked based on their priority level or masked completely using the interrupt masking registers BASEPRI, PRIMASK, and FAULTMASK. They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

The Memory Map

The Cortex-M3 has a predefined memory map. This allows the built-in peripherals, such as the interrupt controller and debug components, to be accessed by simple memory access instructions. Thus most system features are accessible in C program code. The predefined memory map also allows the Cortex-M3 processor to be highly optimized for speed and ease of integration in system-on-a-chip (SoC) designs.

Overall, the 4 GB memory space can be divided into the ranges shown in Figure 2.6.

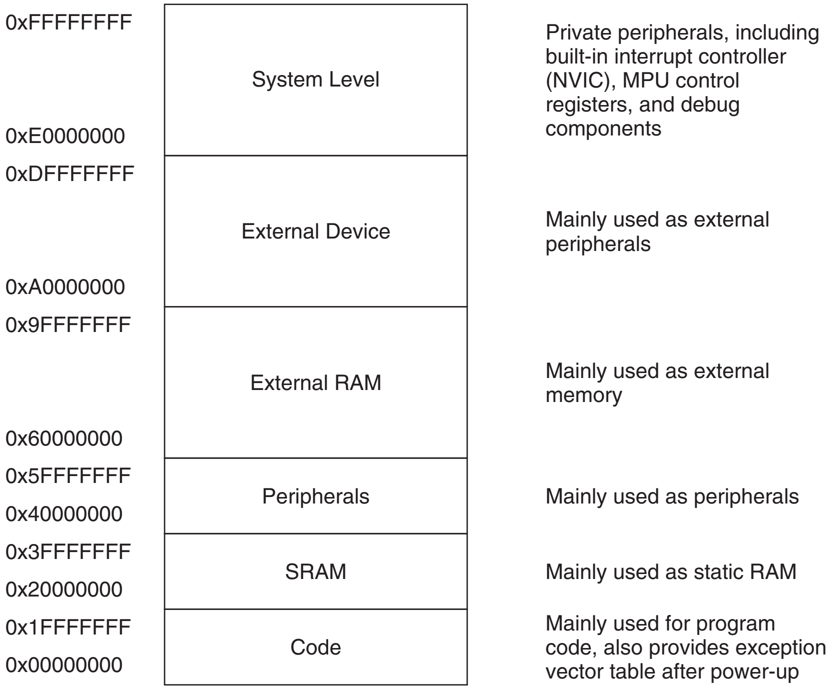

0xFFFFFFFF
System Level
Private peripherals, including built-in interrupt controller (NVIC), MPU control registers, and debug components
0xE0000000
0xDFFFFFFF
External Device
Mainly used as external peripherals
0xA0000000
0x9FFFFFFF
External RAM
Mainly used as external memory
0x60000000
0x5FFFFFFF
Peripherals
Mainly used as peripherals
0x40000000
0x3FFFFFFF
SRAM
Mainly used as static RAM
0x20000000
0x1FFFFFFF
Code
Mainly used for program code, also provides exception vector table after power-up
0x00000000

Figure 2.6 The Cortex-M3 Memory Map

The Cortex-M3 design has an internal bus infrastructure optimized for this memory usage. In addition, the design allows these regions to be used differently. For example, data memory can still be put into the CODE region, and program code can be executed from an external RAM region.

The system-level memory region contains the interrupt controller and the debug components. These devices have fixed addresses, detailed in Chapter 5 (Memory Systems) of this book. By having fixed addresses for these peripherals, you can port applications between different Cortex-M3 products much more easily.

The Bus Interface

There are several bus interfaces on the Cortex-M3 processor. They allow the Cortex-M3 to carry instruction fetches and data accesses at the same time. The main bus interfaces are:

- Code memory buses
- System bus
- Private peripheral bus

The code memory region access is carried out on the *code memory buses*, which physically consist of two buses, one called *I-Code* and another called *D-Code*. These are optimized for instruction fetches for best instruction execution speed.

The *system bus* is used to access memory and peripherals. This provides access to the SRAM, peripherals, external RAM, external devices, and part of the system-level memory regions.

The *private peripheral bus* provides access to a part of the system-level memory dedicated to private peripherals such as debugging components.

The Memory Protection Unit

The Cortex-M3 has an optional Memory Protection Unit, or MPU. This unit allows access rules to be set up for privileged access and user program access. When an access rule is violated, a fault exception is generated, and the fault exception handler will be able to analyze the problem and correct it if possible.

The MPU can be used in various ways. In common scenarios, the MPU is set up by an operating system, allowing data used by privileged code (e.g., the operating system kernel) to be protected from untrusted user programs. The MPU can also be used to make memory regions read-only, to prevent accidental erasing of data, or to isolate memory regions between different tasks in a multitasking system. Overall, it can help make embedded systems more robust and reliable.

The MPU feature is optional and is determined during the implementation stage of the microcontroller or SoC design. For more information on the MPU, refer to Chapter 13.

The Instruction Set

The Cortex-M3 supports the Thumb-2 instruction set. This is one of the most important features of the Cortex-M3 processor because it allows 32-bit instructions and 16-bit instructions to be used together for high code density and high efficiency. It is flexible and powerful yet easy to use.

In previous ARM processors, the CPU had two operation states: a 32-bit ARM state and a 16-bit Thumb state. In the ARM state, the instructions are 32-bit and can execute all supported instructions with very high performance. In the Thumb state, the instructions are 16-bit, so there is a much higher instruction code density, but the Thumb state does not have all the functionality of ARM instructions and may require more instructions to complete certain types of operations.

To get the best of both worlds, many applications have mixed ARM and Thumb codes. However, the mixed-code arrangement does not always work best. There is overhead (in terms of both execution time and instruction space) to switch between the states; and ARM code and Thumb code might need to be compiled separately in different files. This increases the complexity of software development and reduces maximum efficiency of the CPU core.

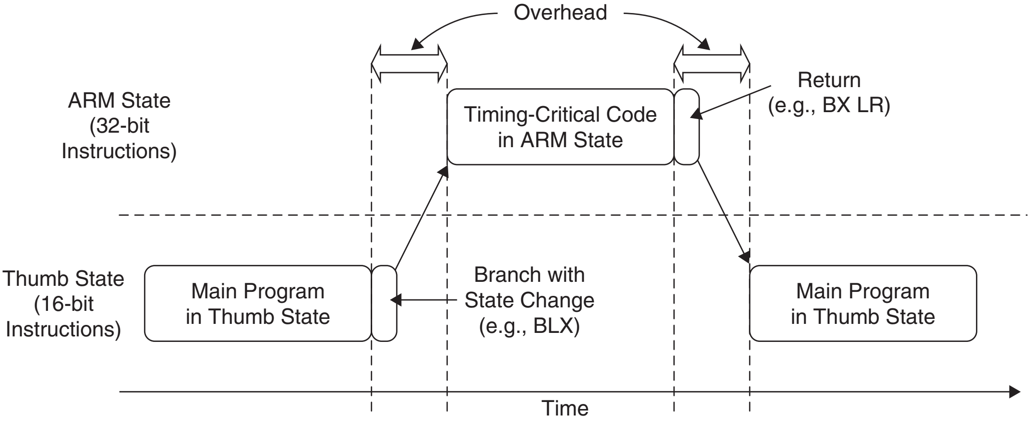


Figure 2.7 Switching Between ARM Code and Thumb Code in Traditional ARM Processors Such as the ARM7

With the introduction of the Thumb-2 instruction set, it is now possible to handle all processing requirements in one operation state. There is no need to switch between the two. In fact, the Cortex-M3 does not support ARM code. Even interrupts are now handled with the Thumb state. (Previously, the ARM core entered interrupt handlers in the ARM state.) Since there is no need to switch between states, the Cortex-M3 processor has a number of advantages over traditional ARM processors:

- No state switching overhead, saving both execution time and instruction space
- No need to separate ARM code and Thumb code source files, making software development and maintenance easier
- It's easier to get the best efficiency and performance, in turn making it easier to write software, because there is no need to worry about switching code between ARM and Thumb to try to get the best density/performance

The Cortex-M3 processor has a number of interesting and powerful instructions. Here are a few examples:

- UFBX, BFI, BFC: Bit field extract, insert, and clear instructions
- UDIV, SDIV: Unsigned and signed divide instructions
- SEV, WFE, WFI: Send-Event, Wait-For-Event, Wait-For-Interrupts; these allow the processor to handle task synchronization on multiprocessor systems or to enter sleep mode
- MSR, MRS: For access to the special registers

Since the Cortex-M3 processor supports the Thumb-2 instruction set only, existing program code for ARM needs to be ported to the new architecture. Most C applications simply need to be recompiled using new compilers that support the Cortex-M3. Some assembler codes need modification and porting to utilize the new architecture and the new unified assembler framework.

Note that not all the instructions in the Thumb-2 instruction set are implemented on the Cortex-M3. The ARMv7-M Architecture Application Level Reference Manual (Ref 2) only requires a subset of the Thumb-2 instructions to be implemented. For example, coprocessor instructions are not supported on the Cortex-M3 (external data processing engines can be added), and SIMD is not implemented on the Cortex-M3. In addition, a few Thumb instructions are not supported, such as BLX with immediate (used to switch processor state from Thumb to ARM), a couple of CPS instructions, and the SETEND instructions, which were introduced in architecture v6. For a complete list of supported instructions, refer to Appendix A of this book or the *Cortex-M3 Technical Reference Manual* (Ref 1).

Interrupts and Exceptions

The Cortex-M3 processor implements a new exception model, introduced in the ARMv7-M architecture. This exception model differs from the traditional ARM exception model, enabling very efficient exception handling. It has a number of system exceptions plus a number of external IRQs (external interrupt inputs). There is no FIQ (fast interrupt in ARM7/9/10/11) in the Cortex-M3; however, interrupt priority handling and nested interrupt support are now included in the interrupt architecture. Therefore, it is easy to set up a system that supports nested interrupts (a higher-priority interrupt can override, or preempt, a lower-priority interrupt handler) and that behaves just like the FIQ in traditional ARM processors.

The interrupt features in the Cortex-M3 are implemented in the NVIC. Aside from supporting external interrupts, the Cortex-M3 also supports a number of internal exception sources, such as system fault handling. As a result, the Cortex-M3 has a number of predefined exception types, as shown in Table 2.2.

Table 2.2 Cortex-M3 Exception Types

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (Prefetch Abort or Data Abort)
6	Usage fault	Programmable	Exceptions due to program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	System service call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
…	…	…	…
255	IRQ #239	Programmable	External interrupt #239

The number of external interrupt inputs is defined by chip manufacturers. A maximum of 240 external interrupt inputs can be supported. In addition, the Cortex-M3 also has an NMI interrupt input. When it is asserted, the NMI interrupt service routine is executed unconditionally.

Debugging Support

The Cortex-M3 processor includes a number of debugging features such as program execution controls, including halting and stepping, instruction breakpoints, data watchpoints, registers and memory accesses, profiling, and traces.

The debugging hardware of the Cortex-M3 processor is based on the CoreSight architecture. Unlike traditional ARM processors, the CPU core itself does not have a JTAG interface. Instead, a debug interface module is decoupled from the core, and a bus interface called the Debug Access Port (DAP) is provided at the core level. Via this bus interface, external debuggers can access control registers to debug hardware as well as system memory, even when the processor is running. The control of this bus interface is carried out by a Debug Port (DP) device. The DPs currently available are the SWJ-DP (supports the traditional JTAG protocol as well as the Serial Wire protocol) or the SW-DP (supports the Serial Wire protocol only). A JTAG-DP module from the ARM CoreSight product family can also be used. Chip manufacturers can choose to attach one of these DP modules to provide the debug interface.

Chip manufacturers can also include an Embedded Trace Macrocell (ETM) to allow instruction trace. Trace information is output via the Trace Port Interface Unit (TPIU), and the debug host (usually a PC) can then collect the executed instruction information via external trace-capturing hardware.

Within the Cortex-M3 processor, a number of events can be used to trigger debug actions. Debug events can be breakpoints, watchpoints, fault conditions, or external debugging request input signals. When a debug event takes place, the Cortex-M3 can either enter halt mode or execute the debug monitor exception handler.

The data watchpoint function is provided by a Data Watchpoint and Trace (DWT) unit in the Cortex-M3 processor. This can be used to stop the processor (or trigger the debug monitor exception routine) or to generate data trace information. When data trace is used, the traced data can be output via the TPIU. (In the CoreSight architecture, multiple trace devices can share one single trace port.)

In addition to these basic debugging features, the Cortex-M3 processor also provides a Flash Patch and Breakpoint (FPB) unit that can provide a simple breakpoint function or remap an instruction access from Flash to a different location in SRAM.

An Instrumentation Trace Macrocell (ITM) provides a new way for developers to output data to a debugger. By writing data to register memory in the ITM, a debugger can collect the data via a trace interface and display or process it. This method is easy to use and faster than JTAG output.

All these debugging components are controlled via the DAP interface bus on the Cortex-M3 or by a program running on the processor core, and all trace information is accessible from the TPIU.

Characteristics Summary

Why is the Cortex-M3 processor such a revolutionary product? What are the advantages of using the Cortex-M3? The benefits and advantages are summarized in this section.

High Performance

- Many instructions, including multiply, are single cycle. Also, the Cortex-M3 processor outperforms most microcontroller products.
- Separate data and instruction buses allow simultaneous data and instruction accesses to be performed.
- The Thumb-2 instruction set makes state switching overhead history. There's no need to spend time switching between the ARM state (32-bit) and the Thumb state (16-bit), so instruction cycles and program size are reduced. This feature has also simplified software development, allowing faster time to market and easier code maintenance.
- The Thumb-2 instruction set provides extra flexibility in programming. Many data operations can now be simplified using shorter code. This also means that the Cortex-M3 has higher code density and reduced memory requirements.
- Instruction fetches are 32-bit. Up to two instructions can be fetched in one cycle. As a result, there's more available bandwidth for data transfer.
- The Cortex-M3 design allows microcontroller products to operate at high clock frequency (over 100 MHz in modern semiconductor manufacturing processes). Even running at the same frequency as most other microcontroller products, the Cortex-M3 has a better clock per instruction (CPI) ratio. This allows more work per MHz, or designs can run at lower clock frequency for lower power consumption.

Advanced Interrupt-Handling Features

- The Built-In Nested Vectored Interrupt Controller (NVIC) supports up to 240 external interrupt inputs. The vectored interrupt feature greatly reduces interrupt latency because there is no need to use software to determine which IRQ handler to serve. In addition, there is no need to have software code to set up nested interrupt support.
- The Cortex-M3 processor automatically pushes registers R0–R3, R12, LR, PSR, and PC in the stack at interrupt entry and pops them back at interrupt exit. This reduces the IRQ handling latency and allows interrupt handlers to be normal C functions (as explained later, in Chapter 8).

- Interrupt arrangement is extremely flexible because the NVIC has programmable interrupt priority control for each interrupt. A minimum of eight levels of priority are supported, and the priority can be changed dynamically.
- Interrupt latency is reduced by special optimization, including late arrival interrupt acceptance and tail-chain interrupt entry.
- Some of the multicycle operations, including Load-Multiple (LDM), Store-Multiple (STM), PUSH, and POP, are now interruptible.
- On receipt of a Nonmaskable Interrupt (NMI) request, immediate execution of the NMI handler is guaranteed unless the system is completely locked up. NMI is very important for many safety-critical applications.

Low Power Consumption

- The Cortex-M3 processor is suitable for low-power designs because of the low gate count.
- It has power-saving mode support (SLEEPING and SLEEPDEEP). The processor can enter sleep mode using Wait for Interrupt (WFI) or Wait for Event (WFE) instructions. The design has separated clocks for essential blocks, so clocking circuits for most parts of the processor can be stopped during sleep.
- The fully static, synchronous, synthesizable design makes the processor easy to be manufactured using any lower power or standard semiconductor process technology.

System Features

- The system provides bit-band operation, byte-invariant big endian mode, and unaligned data access support.
- Advanced fault-handling features include various exception types and fault status registers, making it easier to locate problems.
- With the shadowed stack pointer, stack memory of kernel and user processes can be isolated. With the optional MPU, the processor is more than sufficient to develop robust software and reliable products.

Debug Supports

- Supports JTAG or Serial Wire debug interfaces
- Based on the CoreSight debugging solution; processor status or memory contents can be accessed even when the core is running

- Built-in support for six breakpoints and four watchpoints
- Optional ETM for instruction trace and data trace using DWT
- New debugging features, including fault status registers, new fault exceptions, and Flash patch operations, making debugging much easier
- ITM provides an easy-to-use method to output debug information from test code
- PC Sampler and counters inside the DWT provide code-profiling information

This page intentionally left blank

CHAPTER 3

Cortex-M3 Basics

In This Chapter:

- Registers
- Special Registers
- Operation Mode
- Exceptions and Interrupts
- Vector Tables
- Stack Memory Operations
- Reset Sequence

Registers

As we've seen, the Cortex-M3 processor has registers R0–R15 and a number of special registers. R0–R12 are general purpose, but some of the 16-bit Thumb instructions can only access R0–R7 (low registers), whereas 32-bit Thumb-2 instructions can access all these registers. Special registers have predefined functions and can only be accessed by special register access instructions.

General-Purpose Registers R0–R7

The R0–R7 general-purpose registers are also called *low registers*. They can be accessed by all 16-bit Thumb instructions and all 32-bit Thumb-2 instructions. They are all 32-bit; the reset value is unpredictable.

General-Purpose Registers R8–R12

The R8–R12 registers are also called *high registers*. They are accessible by all Thumb-2 instructions but not by all 16-bit Thumb instructions. These registers are all 32-bit; the reset value is unpredictable.

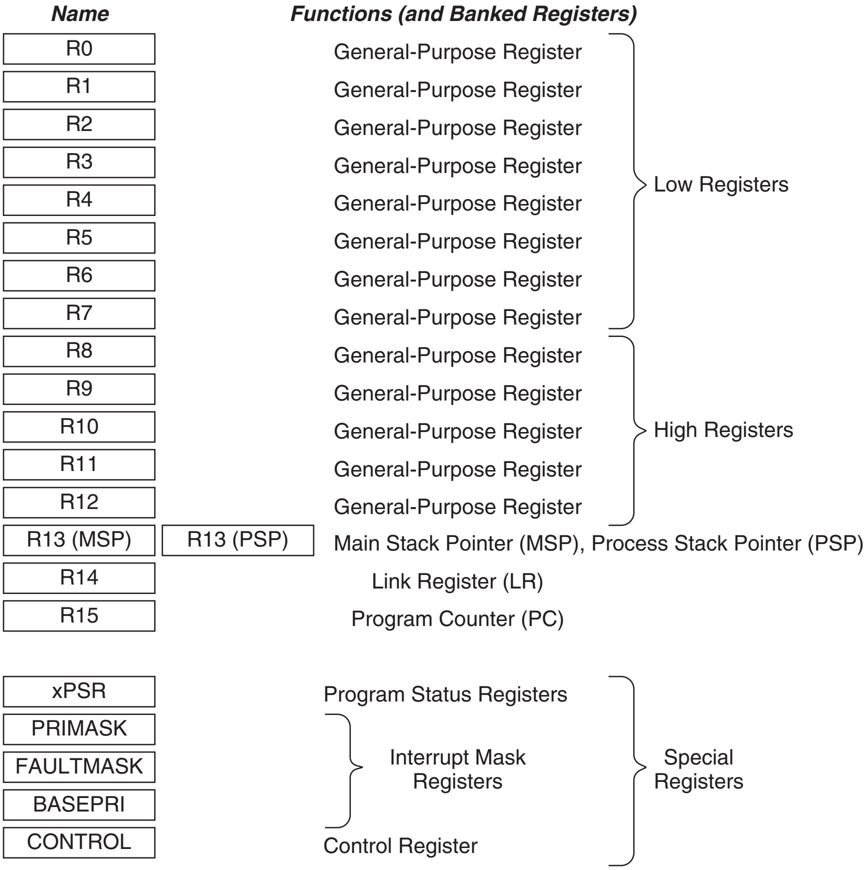


Figure 3.1 Registers in the Cortex-M3

Stack Pointer R13

R13 is the stack pointer. In the Cortex-M3 processor, there are two stack pointers. This duality allows two separate stack memories to be set up. When using the register name R13, you can only access the current stack pointer; the other one is inaccessible unless you use special instructions MSR and MRS. The two stack pointers are:

- Main Stack Pointer (MSP), or *SP_main* in ARM documentation: This is the default stack pointer; it is used by the OS kernel, exception handlers, and all application codes that require privileged access.

- Process Stack Pointer (PSP), or *SP_process* in ARM documentation: Used by the base-level application code (when not running an exception handler).

It is not necessary to use both stack pointers. Simple applications can rely purely on the MSP. The stack pointers are used for accessing stack memory processes such as PUSH and POP.

Stack PUSH and POP

Stack is a memory usage model. It is simply part of the system memory, and a pointer register (inside the processor) is used to make it work as a first-in/last-out buffer. The common use of a stack is to save register contents before some data processing and then restore those contents from the stack after the processing task is done.

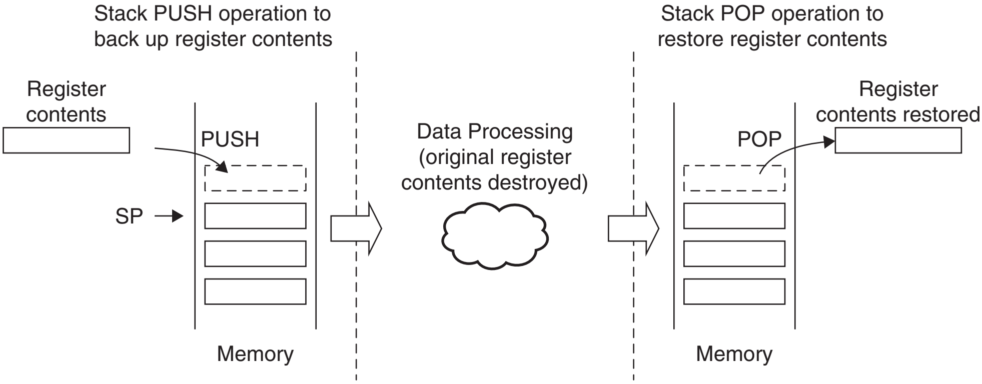

Stack PUSH operation to back up register contents
Stack POP operation to restore register contents
Register contents
PUSH
SP
Memory
Data Processing (original register contents destroyed)
POP
Register contents restored
Memory

Figure 3.2 Basic Concept of Stack Memory

When doing PUSH and POP operations, the pointer register, commonly called Stack Pointer (SP), is adjusted automatically to prevent next stack operations from corrupting previous stacked data. More details on stack operations is provided on later part of this chapter.

In the Cortex-M3, the instructions for accessing stack memory are PUSH and POP. The assembly language syntax is as follows (text after each semicolon [;] is a comment):

```
PUSH        {R0}   ; R13=R13-4, then Memory[R13] = R0
POP         {R0}   ; R0 = Memory[R13], then R13 = R13+4
```

The Cortex-M3 uses a full-descending stack arrangement. (More detail on this subject can be found in the Stack Memory Operations section of this chapter.) Therefore, the stack pointer decrements when new data is stored in the stack. PUSH and POP are usually used to save register contents to stack memory at the start of a subroutine and then restore the registers from stack at the end of the subroutine. You can PUSH or POP multiple registers in one instruction:

```
subroutine_1
      PUSH           {R0-R7, R12, R14}   ; Save registers
      …                                  ; Do your processing
      POP            {R0-R7, R12, R14}   ; Restore registers
      BX             R14                 ; Return to calling function
```

Instead of using *R13,* you can use *SP* (for *stack pointer*) in your program codes. It means the same thing. Inside program code, both the MSP and the PSP can be called *R13/SP*. However, you can access a particular one using special register access instructions (MRS/MSR).

The MSP, also called *SP_main* in ARM documentation, is the default stack pointer after power-up; it is used by kernel code and exception handlers. The PSP, or *SP_process* in ARM documentation, is typically used by Thread processes.

Since register PUSH and POP operations are always word aligned (their addresses must be 0x0, 0x4, 0x8, …), the stack pointer R13 bit 0 and bit 1 are hardwired to zero and always read as zero (RAZ).

Link Register R14

R14 is the link register (LR). Inside an assembly program, you can write it as either *R14* or *LR*. LR is used to store the return program counter when a subroutine or function is called—for example, when you're using the BL (branch and link) instruction:

```
main    ; Main program
        …
        BL  function1 ; Call function1 using Branch with Link
                      ; instruction.
                      ; PC = function1 and
                      ; LR = the next instruction in main
        …
function1
        …             ; Program code for function 1
        BX  LR        ; Return
```

Despite the fact that bit 0 of the program counter is always 0 (because instructions are word aligned or half word aligned), the LR bit 0 is readable and writable. This is because in the Thumb instruction set, bit 0 is often used to indicate ARM/Thumb states. To allow the Thumb-2 program for the Cortex-M3 to work with other ARM processors that support the Thumb-2 instruction set, this LSB is writable and readable.

Program Counter R15

R15 is the program counter. You can access it in assembler code by either R15 or PC. Due to the pipelined nature of the Cortex-M3 processor, when you read this register you will find that the value is different than the location of the executing instruction by 4. For example:

```
0x1000 :   MOV   R0, PC   ; R0 = 0x1004
```

Writing to the program counter will cause a branch (but link registers do not get updated). Since an instruction address must be half word aligned, the LSB (bit 0) of the program counter read value is always 0. However, in branching, either by writing to PC or using branch instructions, the LSB of the target address should be set to 1 because it is used to indicate the Thumb state operations. If it is 0, it can imply trying to switch to the ARM state and will result in a fault exception in the Cortex-M3.

Special Registers

The special registers in the Cortex-M3 processor include these:

- Program Status Registers (PSRs)
- Interrupt Mask Registers (PRIMASK, FAULTMASK, and BASEPRI)
- Control Register (Control)

Special registers can only be accessed via MSR and MRS instructions; they do not have memory addresses:

```
     MRS   <reg>, <special_reg>; Read special register
     MSR   <special_reg>, <reg>; write to special register
```

Program Status Registers (PSRs)

The program status registers are subdivided into three status registers:

- Application PSR (APSR)[1]
- Interrupt PSR (IPSR)
- Execution PSR (EPSR)

The three PSRs can be accessed together or separately using the special register access instructions MSR and MRS. When they are accessed as a collective item, the name *xPSR* is used.

[1] In the beginning of Cortex-M3 development, APSR was called Flags Program Status Word (FPSR), so some early versions of development tools might use the name FPSR for APSR.

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

Figure 3.3 Program Status Registers (PSRs) in the Cortex-M3

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception Number				

Figure 3.4 Combined Program Status Registers (xPSR) in the Cortex-M3

You can read the program status registers using the MRS instruction. You can also change the APSR using the MSR instruction, but EPSR and IPSR are read-only. For example:

```
MRS    r0, APSR    ; Read Flag state into R0
MRS    r0, IPSR    ; Read Exception/Interrupt state
MRS    r0, EPSR    ; Read Execution state
MSR    APSR, r0    ; Write Flag state
```

In ARM assembler, when accessing xPSR (all three program status registers as one), the symbol *PSR* is used:

```
MRS    r0, PSR    ; Read the combined program status word
MSR    PSR, r0    ; Write combined program state word
```

The descriptions for the bit fields in PSR are shown in Table 3.1.

Table 3.1 Bit Fields in Cortex-M3 Program Status Registers

Bit	Description
N	Negative
Z	Zero
C	Carry/borrow
V	Overflow
Q	Sticky saturation flag
ICI/IT	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit
T	Thumb state, always 1; trying to clear this bit will cause a fault exception
Exception Number	Indicates which exception the processor is handling

If you compare this with the Current Program Status Register (CPSR) in ARM7, you might find that some bit fields that were used in ARM7 are gone. The Mode (M) bit field is gone because the Cortex-M3 does not have the operation mode as defined in ARM-7. Thumb-bit (T) is moved to bit 24. Interrupt status (I and F) bits are replaced by the new interrupt mask registers (PRIMASKs), which are separated from PSR. For comparison, the CPSR in traditional ARM processors is shown in Figure 3.5.

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
ARM	N	Z	C	V	Q	IT	J	Resrv	GE[3:0]	IT	E	A	I	F	T	M[4:0]

Figure 3.5 Current Program Status Registers (CPSR) in Traditional ARM Processors

PRIMASK, FAULTMASK, and BASEPRI Registers

The PRIMASK, FAULTMASK, and BASEPRI registers are used to disable exceptions (see Table 3.2).

Table 3.2 Cortex-M3 Interrupt Mask Registers

Register Name	Description
PRIMASK	A 1-bit register. When this is set, it allows NMI and the hard fault exception; all other interrupts and exceptions are masked. The default value is 0, which means that no masking is set.
FAULTMASK	A 1-bit register. When this is set, it allows only the NMI, and all interrupts and fault handling exceptions are disabled. The default value is 0, which means that no masking is set.
BASEPRI	A register of up to 9 bits (depending on the bit width implemented for priority level). It defines the masking priority level. When this is set, it disables all interrupts of the same or lower level (larger priority value). Higher-priority interrupts can still be allowed. If this is set to 0, the masking function is disabled (this is the default).

The PRIMASK and BASEPRI registers are useful for temporarily disabling interrupts in timing-critical tasks. An OS could use FAULTMASK to temporarily disable fault handling when a task has crashed. In this scenario, a number of different faults might be taking place when a task crashes. Once the core starts cleaning up, it might not want to be interrupted by other faults caused by the crashed process. Therefore, the FAULTMASK gives the OS kernel time to deal with fault conditions.

To access the PRIMASK, FAULTMASK, and BASEPRI registers, the MRS and MSR instructions are used. For example:

```
MRS    r0, BASEPRI   ; Read BASEPRI register into R0
MRS    r0, PRIMASK   ; Read PRIMASK register into R0
MRS    r0, FAULTMASK ; Read FAULTMASK register into R0
MSR    BASEPRI, r0   ; Write R0 into BASEPRI register
MSR    PRIMASK, r0   ; Write R0 into PRIMASK register
MSR    FAULTMASK, r0 ; Write R0 into FAULTMASK register
```

The PRIMASK, FAULTMASK, and BASEPRI registers cannot be set in the user access level.

The Control Register

The Control register is used to define the privilege level and the stack pointer selection. This register has two bits, as shown in Table 3.3.

Table 3.3 Cortex-M3 CONTROL Register

Bit	Function
CONTROL[1]	Stack status: 1 = Alternate stack is used 0 = Default stack (MSP) is used If it is in the Thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be zero when the processor is in handler mode.
CONTROL[0]	0 = Privileged in Thread mode 1 = User state in Thread mode If in handler mode (not Thread mode), the processor operates in privileged mode.

CONTROL[1]

In the Cortex-M3, the CONTROL[1] bit is always 0 in handler mode. However, in the Thread or base level, it can be either 0 or 1.

This bit is writable only when the core is in Thread mode and privileged. In the user state or handler mode, writing to this bit is not allowed. Aside from writing to this register, another way to change this bit is to change bit 2 of the LR when in exception return. This subject is discussed in Chapter 8, where details on exceptions are described.

CONTROL[0]

The CONTRL[0] bit is writable only in a privileged state. Once it enters the user state, the only way to switch back to privileged is to trigger an interrupt and change this in the exception handler.

To access the Control register, the MRS and MSR instructions are used:

```
MRS    r0, CONTROL ; Read CONTROL register into R0
MSR    CONTROL, r0 ; Write R0 into CONTROL register
```

Operation Mode

The Cortex-M3 processor supports two modes and two privilege levels.

	Privileged	***User***
When running an exception	Handler Mode	■
When running main program	Thread Mode	Thread Mode

Figure 3.6 Operation Modes and Privilege Levels in Cortex-M3

When the processor is running in Thread mode, it can be in either the privileged or user level, but handlers can only be in the privileged level. When the processor exits reset, it is in Thread mode, with privileged access rights.

In the user access level (Thread mode), access to the System Control Space, or SCS—a part of the memory region for configuration registers and debugging components—is blocked. Furthermore, instructions that access special registers (such as MSR, except when accessing APSR) cannot be used. If a program running at the user access level tries to access SCS or special registers, a fault exception will occur.

Software in a privileged access level can switch the program into the user access level using the Control register. When an exception takes place, the processor will always switch to a privileged state and return to the previous state when exiting the exception handler. A user program cannot change back to the privileged state directly by writing to the Control register. It has to go through an exception handler that programs the Control register to switch the processor back into privileged access level when returning to Thread mode.

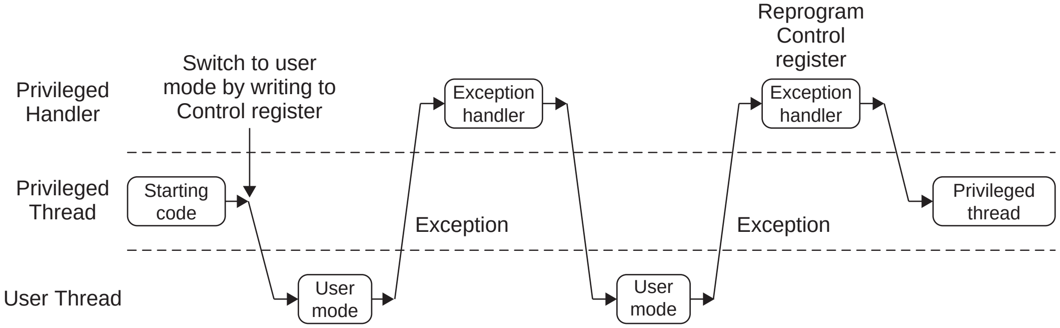


Figure 3.7 Switching of Operation Mode by Programming the Control Register or by Exceptions

The support of privileged and user access levels provides a more secure and robust architecture. For example, when a user program goes wrong, it will not be able to corrupt Control registers in the NVIC. In addition, if the MPU is present, it is possible to block user programs from accessing memory regions used by privileged processes.

You can separate the user application stack from the kernel stack memory to avoid the possibility of crashing a system caused by stack operation errors in user programs. With this arrangement, the user program (running in Thread mode) uses the PSP, and the exception handlers use the MSP. The switching of stack pointers is automatic upon entering or leaving the exception handlers. This topic is discussed in more detail in Chapter 8 of this book.

The mode and access level of the processor are defined by the Control register. When the Control regsister bit 0 is zero, the processor mode changes when an exception takes place.

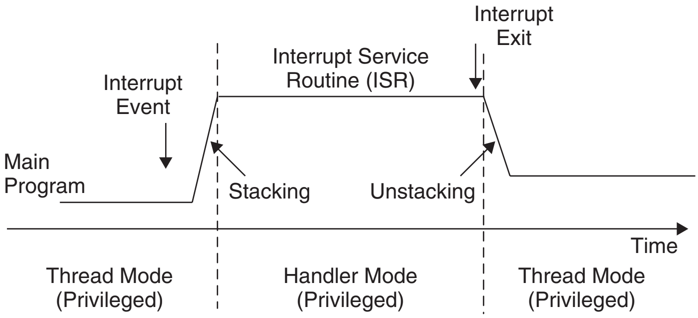


Figure 3.8 Switching Processor Mode at Interrupt

When Control register bit 0 is one (Thread running user application), both processor mode and access level change when an exception takes place.

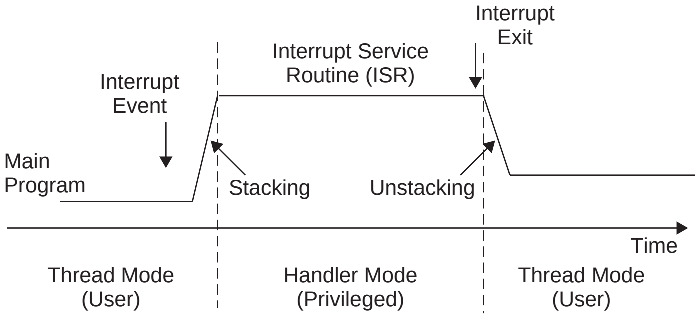


Figure 3.9 Switching Processor Mode and Privilege Level at Interrupt

Control register bit 0 is programmable only in the privileged level. For a user-level program to switch to privileged state, it has to raise an interrupt (for example, SVC, or System Service Call) and write to CONTROL[0] within the handler.

Exceptions and Interrupts

The Cortex-M3 supports a number of exceptions, including a fixed number of system exceptions and a number of interrupts, commonly called *IRQ*. The number of interrupt inputs on a Cortex-M3 microcontroller depends on the individual design. Interrupts generated by peripherals, except System Tick Timer, are also connected to the interrupt input signals. The typical number of interrupt inputs is 16 or 32. However, you might find some microcontroller designs with more (or fewer) interrupt inputs.

Besides the interrupt inputs, there is also an NMI input signal. The actual use of NMI depends on the design of the microcontroller or SoC product you use. In most cases, the NMI could be connected to a watchdog timer or a voltage-monitoring block that warns the processor when the voltage drops below a certain level. The NMI signal can be activated any time, even right after the core exits reset.

The list of exceptions found in the Cortex-M3 is shown in Table 3.4. A number of the system exceptions are fault-handling exceptions that can be triggered by various error conditions. The NVIC also provides a number of fault status registers so that error handlers can determine the cause of the exceptions.

Table 3.4 Exception Types in Cortex-M3

Exception Number	Exception Type	Priority	Function
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt
3	Hard fault	−1	All classes of fault, when the corresponding fault handler cannot be activated because it is currently disabled or masked by exception masking
4	MemManage	Settable	Memory management fault; caused by MPU violation or invalid accesses (such as an instruction fetch from a nonexecutable region)
5	BusFault	Settable	Error response received from the bus system; caused by an instruction prefetch abort or data access error
6	Usage fault	Settable	Usage fault; typical causes are invalid instructions or invalid state transition attempts (such as trying to switch to ARM state in the Cortex-M3)
7–10	–	–	Reserved
11	SVC	Settable	System service call via SVC instruction

(Continued)

Table 3.4 (Continued)

Exception Number	Exception Type	Priority	Function
12	Debug monitor	Settable	Debug monitor
13	—	—	Reserved
14	PendSV	Settable	Pendable request for System Service
15	SYSTICK	Settable	System Tick Timer
16–255	IRQ	Settable	IRQ input #0–239

More details on exception operations in the Cortex-M3 are discussed in Chapters 7–9.

Vector Tables

When an exception event takes place on the Cortex-M3 and is accepted by the processor core, the corresponding exception handler is executed. To determine the starting address of the exception handler, a vector table mechanism is used. The *vector table* is an array of word data, each representing the starting address of one exception type. The vector table is relocatable and the relocation is controlled by a relocation register in the NVIC. After reset, this relocation control register is reset to 0; therefore, the vector table is located in address 0x0 after reset.

Table 3.5 Vector Table Definition After Reset

Exception Type	Address Offset	Exception Vector
18–255	0x48–0x3FF	IRQ #2–239
17	0x44	IRQ #1
16	0x40	IRQ #0
15	0x3C	SYSTICK
14	0x38	PendSV
13	0x34	Reserved
12	0x30	Debug Monitor
11	0x2C	SVC
7–10	0x1C–0x28	Reserved
6	0x18	Usage fault
5	0x14	Bus fault
4	0x10	MemManage fault
3	0x0C	Hard fault
2	0x08	NMI
1	0x04	Reset
0	0x00	Starting value of the MSP

For example, if the reset is exception type 1, the address of the reset vector is 1 times 4 (each word is 4 bytes), which equals 0x00000004, and NMI vector (type 2) is located in 2 × 4 = 0x00000008. The address 0x00000000 is used as the starting value for the MSP.

The LSB of each exception vector indicates whether the exception is to be executed in the Thumb state. Since the Cortex-M3 can support only Thumb instructions, the LSB of all the exception vectors should be set to 1.

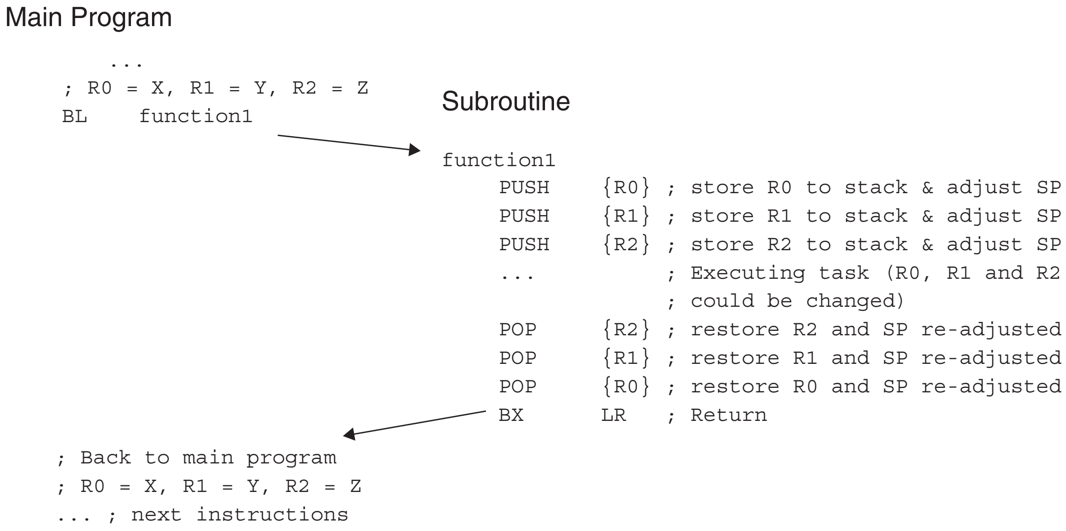
Stack Memory Operations

In the Cortex-M3, besides normal software-controlled stack PUSH and POP, the stack PUSH and POP operations are also carried out automatically when entering or exiting an exception/interrupt handler. In this section we examine the software stack operations. (Stack operations during exception handling are covered in Chapter 9.)

Basic Operations of the Stack

In general, stack operations are memory write or read operations, with the address specified by a stack pointer (SP). Data in registers are saved into stack memory by a PUSH operation and can be restored to registers later by a POP operation. The stack pointer is adjusted automatically in PUSH and POP so that multiple data PUSH will not cause old stacked data to be erased.

The function of the stack is to store register contents in memory so that they can be restored later, after a processing task is completed. For normal uses, for each store (PUSH) there must be a corresponding read (POP), and the address of the POP operation should match that of the PUSH operation (see Figure 3.10). When PUSH/POP instructions are used, the stack pointer is incremented/decremented automatically.



Main Program

```
    ...
; R0 = X, R1 = Y, R2 = Z
BL    function1
```

Subroutine

```
function1
     PUSH    {R0} ; store R0 to stack & adjust SP
     PUSH    {R1} ; store R1 to stack & adjust SP
     PUSH    {R2} ; store R2 to stack & adjust SP
     ...          ; Executing task (R0, R1 and R2
                  ; could be changed)
     POP     {R2} ; restore R2 and SP re-adjusted
     POP     {R1} ; restore R1 and SP re-adjusted
     POP     {R0} ; restore R0 and SP re-adjusted
     BX      LR   ; Return
```

```
; Back to main program
; R0 = X, R1 = Y, R2 = Z
... ; next instructions
```

Figure 3.10 Stack Operation Basics: One Register in Each Stack Operation

When program control returns to the main program, the R0–R2 contents are the same as before. Notice the order of PUSH and POP: The POP order must be the reverse of PUSH.

These operations can be simplified, thanks to PUSH and POP instructions allowing multiple load and store. In this case, the ordering of a register POP is automatically reversed by the processor (see Figure 3.11).

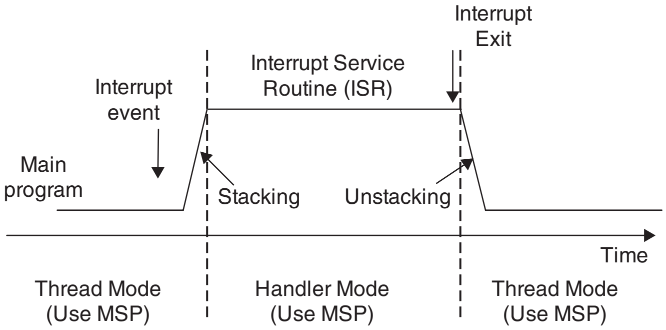


Figure 3.11 Stack Operation Basics: Multiple Register Stack Operation

You can also combine RETURN with a POP operation. This is done by pushing the LR to the stack and popping it back to PC at the end of the subroutine (see Figure 3.12).

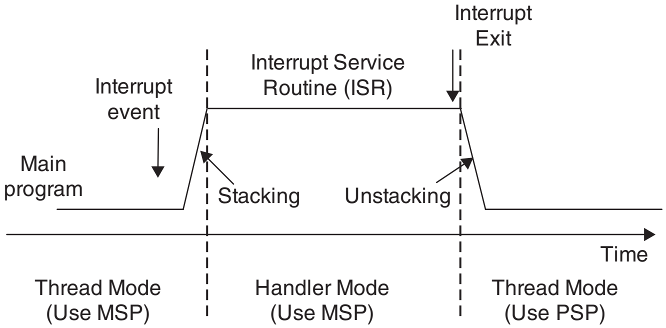


Figure 3.12 Stack Operation Basics: Combining Stack POP and RETURN

Cortex-M3 Stack Implementation

The Cortex-M3 employs a full-descending stack operation model. The stack pointer (SP) points to the last data pushed to the stack memory, and the SP decrements before a new PUSH operation. See Figure 3.13 for an example showing execution of the instruction PUSH {R0}.

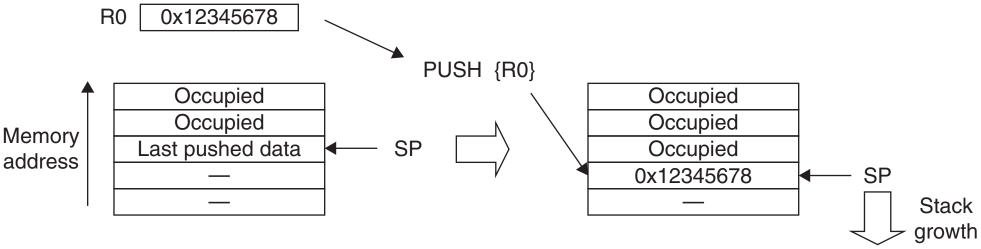


Figure 3.13 Cortex-M3 Stack PUSH Implementation

For POP operations, the data is read from the memory location pointer by SP, then the stack pointer is incremented. The contents in the memory location are unchanged but will be overwritten when the next PUSH operation takes place (see Figure 3.14).

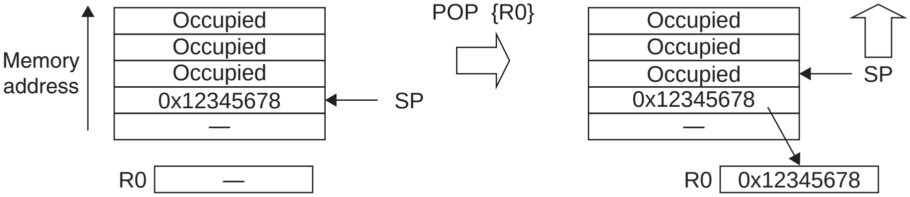


Figure 3.14 Cortex-M3 Stack POP Implementation

Since each PUSH/POP operation transfers 4 bytes of data (each register contains 1 word, or 4 bytes), the SP decrements/increments by 4 at a time, or a multiple of 4 if more than one register is pushed or popped.

In the Cortex-M3, R13 is defined as the SP. When an interrupt takes place, a number of registers will be pushed automatically, and R13 will be used as the SP for this stacking process. Similarly, the pushed registers will be restored/popped automatically when exiting an interrupt handler, and the stack pointer will also be adjusted.

The Two-Stack Model in the Cortex-M3

As mentioned before, the Cortex-M3 has two stack pointers: the Main Stack Pointer (MSP) and the Process Stack Pointer (PSP). The SP register to be used is controlled by the Control register bit 1 (CONTROL[1] in the following text).

When CONTROL[1] is 0, the MSP is used for both Thread mode and handler mode. In this arrangement, the main program and the exception handlers share the same stack memory region. This is the default setting after power-up.

When the Control register bit 1 is 1, the PSP is used in Thread mode. In this arrangement, the main program and the exception handler can have separate stack memory regions. This can prevent a stack error in a user application from damaging the stack used by the OS (assuming that the user application runs only in Thread mode and the OS kernel executes in handler mode).

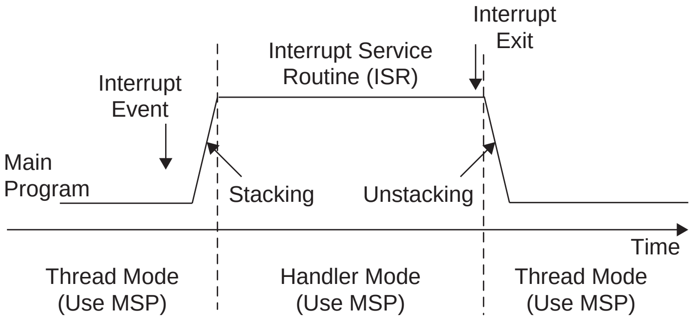


Figure 3.15 Control[1]=0: Both Thread Level and Handler Use Main Stack

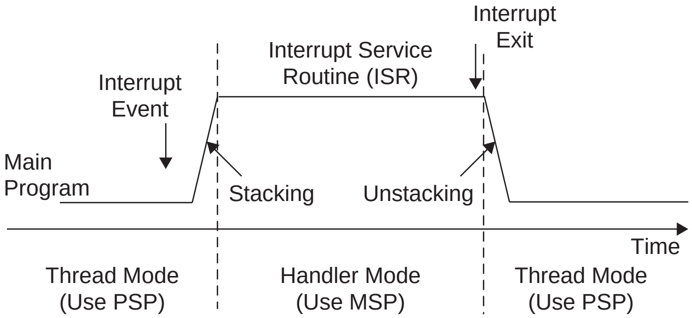


Figure 3.16 Control[1]=1: Thread Level Uses Process Stack and Handler Uses Main Stack

Note that in this situation, the automatic stacking and unstacking mechanism will use PSP; whereas stack operations inside the handler will use MSP.

It is possible to perform read/write operations directly to the MSP and PSP, without any confusion of which R13 you are referring to. Provided that you are in privileged level, you can access MSP and PSP using the MRS and MSR instructions:

```
MRS  R0, MSP    ; Read Main Stack Pointer to R0
MSR  MSP, R0    ; Write R0 to Main Stack Pointer
MRS  R0, PSP    ; Read Process Stack Pointer to R0
MSR  PSP, R0    ; Write R0 to Process Stack Pointer
```

By reading the PSP value using an MRS instruction, the OS can read data stacked by the user application (such as register contents before System Service Call, SVC). In addition, the OS can change the PSP pointer value—for example, during context switching in multitasking systems.

Reset Sequence

After the processor exits reset, it will read two words from memory:

- Address 0x00000000: Starting value of R13 (the stack pointer)

- Address 0x00000004: Reset vector (the starting address of program execution; LSB should be set to 1 to indicate Thumb state)

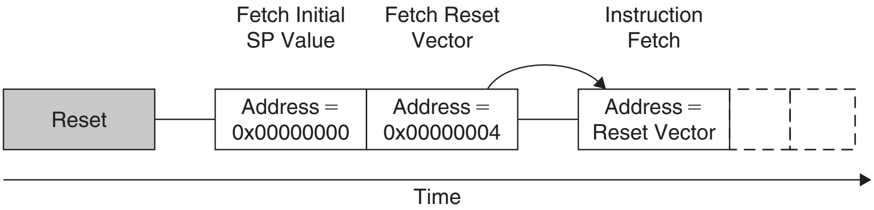


Figure 3.17 Reset Sequence

This differs from traditional ARM processor behavior. Previous ARM processors executed program code starting from address 0x0. Furthermore, the vector table in previous ARM devices was instructions (you have to put a branch instruction there so that your exception handler can be put in another location). In the Cortex-M3, the initial value for the MSP is put at the beginning of the memory map, followed by the vector table, which contains vector address values. (The vector table can be relocated to another location later, during program execution.) In addition, the contents of the vector table are address values, not branch instructions. The first item in the vector table (exception type 1) is the reset vector, which is the second piece of data fetched by the processor after reset.

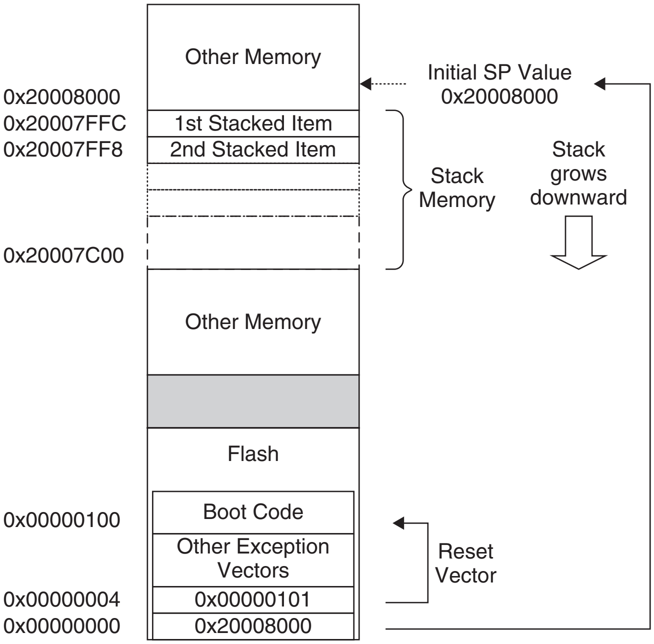


Figure 3.18 Initial Stack Pointer Value and Initial Program Counter (PC) Value Example

Since the stack operation in the Cortex-M3 is a full descending stack (stack pointer decrement before store), the initial stack pointer value should be set to the first memory after the top of the stack region. For example, if you have a stack memory range from 0x20007C00 to 0x20007FFF (1 K bytes), the initial stack value should be set to 0x20008000.

The vector table starts after the initial SP value. The first vector is the reset vector. Notice that in the Cortex-M3, vector addresses in the vector table should have their LSB set to 1 to indicate that they are Thumb code. For that reason, the previous example has 0x101 in the reset vector, whereas the boot code starts at address 0x100. After the reset vector is fetched, the Cortex-M3 can then start to execute the program from the reset vector address and begin normal operations. It is necessary to have the stack pointer initialized, because some of the exceptions (such as NMI) can happen right after reset, and the stack memory could be required for the handler of those exceptions.

Various software development tools might have different ways to specify the starting stack pointer value and reset vector. If you need more information on this topic, it's best to look at project examples provided with the development tools. Simple examples are provided in Chapter 10 and Chapter 20 of this book for ARM tools and in Chapter 19 for the GNU tool chain.

CHAPTER 4

Instruction Sets

In This Chapter:

- Assembly Basics
- Instruction List
- Instruction Descriptions
- Several Useful Instructions in the Cortex-M3

This chapter provides some insight into the instruction set in the Cortex-M3 and examples for a number of instructions. You'll also find a quick reference of the support instructions in Appendix A of this book. For complete details of each instruction, refer to the ARM v7-M Architecture Application Level Reference Manual (Ref 2).

Assembly Basics

Here we introduce some basic syntax of ARM assembly to make it easier to understand the rest of the code examples in this book. Most of the assembly code examples in this book are based on the ARM assembler tools, with the exception of those in Chapter 19, which focus on the GNU tool chain.

Assembler Language: Basic Syntax

In assembler code, the following instruction formatting is commonly used:

```
label
        opcode operand1, operand2,... ; Comments
```

The *label* is optional. Some of the instructions might have a label in front of them so that the address of the instructions can be determined using the label. Then you will find the opcode (the instruction), followed by a number of operands. Normally, the first operand is the destination of the operation. The number of operands in an instruction depends on the type of instruction, and the syntax format of the operand can also be different. For example, immediate data are usually in the form *#number*, as here:

```
    MOV R0, #0x12 ; Set R0 = 0x12 (hexadecimal)
    MOV R1, #'A'  ; Set R1 = ASCII character A
```

The text after each semicolon (;) is a comment. These comments do not affect the program operation, but they can make programs easier for humans to understand.

You can define constants using EQU and then use them inside your program code. For example:

```
NVIC_IRQ_SETEN0 EQU 0xE000E100
NVIC_IRQ0_ENABLE EQU 0x1
   ...
   LDR R0,=NVIC_IRQ_SETEN0  ; LDR here is a pseudo instruction that
                            ; convert to a PC relative load by
                            ; assembler.
   MOV R1,#NVIC_IRQ0_ENABLE ; Move immediate data to register
   STR R1, [R0]             ; Enable IRQ 0 by writing R1 to address
                            ; in R0
```

DCI can be used to code an instruction if your assembler cannot generate the exact instruction that you want and if you know the binary code for the instruction:

```
DCI 0xBE00 ; Breakpoint (BKPT 0), a 16-bit instruction
```

We can use DCB (for byte size constant values such as characters) and DCD (for word size constant values) to define binary data in your code:

```
     LDR R3,=MY_NUMBER ; Get the memory address value of MY_NUMBER
     LDR R4,[R3]       ; Get the value code 0x12345678 in R4
     ...
     LDR R0,=HELLO_TXT ; Get the starting memory address of
                       ; HELLO_TXT
     BL  PrintText     ; Call a function called PrintText to
                       ; display string
     ...
MY_NUMBER
     DCD 0x12345678
HELLO_TXT
     DCB "Hello\n",0   ; null terminated string
```

Note that the assembler syntax depends on which assembler tool you are using. Here the ARM assembler tools syntax is introduced. For syntax of other assemblers, it is best to start from the code examples provided with the tools.

Assembler Language: Use of Suffixes

In assembler for ARM processors, instructions can be followed by suffixes, as shown in Table 4.1.

Table 4.1 Suffixes in Instructions

Suffix	Description
S	Update APSR (flags); for example: `ADDS R0, R1 ; this will update APSR`
EQ , NE, LT, GT, and so on	Conditional execution; EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, etc. For example: `BEQ <Label> ; Branch if equal`

For the Cortex-M3, the conditional execution suffixes are usually used for branch instructions. However, other instructions can also be used with the conditional execution suffixes if they are inside an IF-THEN instruction block. (This concept is introduced in a later part of this chapter.) In those cases, the *S* suffix and the conditional execution suffixes can be used at the same time. Fifteen condition choices are available, as described later in this chapter.

Assembler Language: Unified Assembler Language

To support and get the best out of the Thumb-2 instruction set, the Unified Assembler Language (UAL) was developed to allow selection of 16-bit and 32-bit instructions and to make it easier to port applications between ARM code and Thumb code by using the same syntax for both. (With UAL, the syntax of Thumb instructions is now the same as for ARM instructions.)

```
ADD  R0, R1     ; R0 = R0 + R1, using Traditional Thumb syntax
ADD  R0, R0, R1 ; Equivalent instruction using UAL syntax
```

The traditional Thumb syntax can still be used. One thing you need to be careful with is that with traditional Thumb instruction syntax, some instructions change the flags in APSR, even if the *S* suffix is not used. However, when the UAL syntax is used, whether the instruction changes the flag depends on the *S* suffix. For example:

```
AND  R0, R1      ; Traditional Thumb syntax
ANDS  R0, R0, R1 ; Equivalent UAL syntax (S suffix is added)
```

With the new Thumb-2 instruction support, some of the operations can be handled by either a Thumb instruction or a Thumb-2 instruction. For example, R0 = R0 + 1 can be implemented as a 16-bit Thumb instruction or a 32-bit Thumb-2 instruction. With UAL, you can specify which instruction you want by adding suffixes:

```
ADDS   R0, #1 ; Use 16-bit Thumb instruction by default
              ; for smaller size
ADDS.N R0, #1 ; Use 16-bit Thumb instruction (N=Narrow)
ADDS.W R0, #1 ; Use 32-bit Thumb-2 instruction (W=wide)
```

The .W (wide) suffix specifies a 32-bit instruction. If no suffix is given, the assembler tool can choose either instruction but usually defaults to 16-bit Thumb code to get a smaller size.

Depending on tool support, you may also use the .N (narrow) suffix to specify a 16-bit Thumb instruction.

Again, this syntax is for ARM assembler tools. Other assemblers might have slightly different syntax. If no suffix is given, the assembler might chose the instruction for you, with the minimum code size.

In most cases, applications will be coded in C, and the C compilers will use 16-bit instructions if possible due to smaller code size. However, when the immediate data exceeds a certain range or when the operation can be better handled with a 32-bit Thumb-2 instruction, the 32-bit instruction will be used.

32-bit Thumb-2 instructions can be half word aligned. For example, you can have a 32-bit instruction located in a half word location:

```
0x1000 : LDR r0,[r1]   ; a 16-bit instructions (occupy 0x1000-0x1001)
0x1002 : RBIT.W r0     ; a 32-bit Thumb-2 instruction (occupy
                       ; 0x1002-0x1005)
```

Most of the 16-bit instructions can only access registers R0 to R7; 32-bit Thumb-2 instructions do not have this limitation. However, use of PC (R15) might not be allowed in some of the instructions. Refer to the *ARM v7-M Architecture Application Level Reference Manual* (Ref 2: section A4.6) if you need to find out more detail in this area.

Instruction List

The supported instructions are listed in Tables 4.2–4.9. The complete details of each instruction are available in the *ARM v7-M Architecture Application Level Reference Manual* (Ref 2). There is also a summary of the supported instruction sets in Appendix A.

Table 4.2 16-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (Logical AND one value with the logic inversion of another value)
CMN	Compare negative (compare one data with two's complement of another data and update flags)
CMP	Compare (compare two data and update flags)

Table 4.2 (Continued)

Instruction	Function
CPY	Copy (available from architecture v6; move a value from one high or low register to another high or low register)
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MOV	Move (can be used for register-to-register transfers or loading immediate data)
MUL	Multiply
MVN	Move NOT (obtain logical inverted value)
NEG	Negate (obtain two's complement value)
ORR	Logical OR
ROR	Rotate right
SBC	Subtract with carry
SUB	Subtract
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
REV	Reverse the byte order in a 32-bit register (available from architecture v6)
REVH	Reverse the byte order in each 16-bit half word of a 32-bit register (available from architecture v6)
REVSH	Reverse the byte order in the lower 16-bit half word of a 32-bit register and sign extends the result to 32 bits. (available from architecture v6)
SXTB	Signed extend byte (available from architecture v6)
SXTH	Signed extend half word (available from architecture v6)
UXTB	Unsigned extend byte (available from architecture v6)
UXTH	Unsigned extend half word (available from architecture v6)

Table 4.3 16-Bit Branch Instructions

Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch with link; call a subroutine and store the return address in LR
BLX	Branch with link and change state (BLX <reg> only)[1]
CBZ	Compare and branch if zero (architecture v7)
CBNZ	Compare and branch if nonzero (architecture v7)
IT	IF-THEN (architecture v7)

[1] BLX with immediate is not supported because it will always try to change to the ARM state, which is not supported in the Cortex-M3. Attempts to use BLX <*reg*> to change to the ARM state will also result in a fault exception.

Table 4.4 16-Bit Load and Store Instructions

Instruction	Function
LDR	Load word from memory to register
LDRH	Load half word from memory to register
LDRB	Load byte from memory to register
LDRSH	Load half word from memory, sign extend it, and put it in register
LDRSB	Load byte from memory, sign extend it, and put it in register
STR	Store word from register to memory
STRH	Store half word from register to memory
STRB	Store byte from register to memory
LDMIA	Load multiple increment after
STMIA	Store multiple increment after
PUSH	Push multiple registers
POP	Pop multiple registers

Table 4.5 Other 16-Bit Instructions

Instruction	Function
SVC	System service call
BKPT	Breakpoint; if debug is enabled, will enter debug mode (halted), or if debug monitor exception is enabled, will invoke the debug exception; otherwise it will invoke a fault exception
NOP	No operation
CPSIE	Enable PRIMASK (CPSIE i)/FAULTMASK (CPSIE f) register (set the register to 0)
CPSID	Disable PRIMASK (CPSID i)/ FAULTMASK (CPSID f) register (set the register to 1)

Table 4.6 32-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
ADDW	Add wide (#immed_12)
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (logical AND one value with the logic inversion of another value)
BFC	Bit field clear
BFI	Bit field insert

Table 4.6 (Continued)

Instruction	Function
CMN	Compare negative (compare one data with two's complement of another data and update flags)
CMP	Compare (compare two data and update flags)
CLZ	Count lead zero
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MLA	Multiply accumulate
MLS	Multiply and subtract
MOV	Move
MOVW	Move wide (write a 16-bit immediate value to register)
MOVT	Move top (write an immediate value to the top half word of destination reg)
MVN	Move negative
MUL	Multiply
ORR	Logical OR
ORN	Logical OR NOT
RBIT	Reverse bit
REV	Byte reserve word
REVH/REV16	Byte reverse packed half word
REVSH	Byte reverse signed half word
ROR	Rotate right register
RSB	Reverse subtract
RRX	Rotate right extended
SBFX	Signed bit field extract
SDIV	Signed divide
SMLAL	Signed multiply accumulate long
SMULL	Signed multiply long
SSAT	Signed saturate
SBC	Subtract with carry
SUB	Subtract
SUBW	Subtract wide (#immed_12)
SXTB	Sign extend byte
TEQ	Test equivalent (use as logical exclusive OR; flags are updated but result is not stored)

(Continued)

Table 4.6 (Continued)

Instruction	Function
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
UBFX	Unsigned bit field extract
UDIV	Unsigned divide
UMLAL	Unsigned multiply accumulate long
UMULL	Unsigned multiply long
USAT	Unsigned saturate
UXTB	Unsigned extend byte
UXTH	Unsigned extend half word

Table 4.7 32-Bit Load and Store Instructions

Instruction	Function
LDR	Load word data from memory to register
LDRB	Load byte data from memory to register
LDRH	Load half word data from memory to register
LDRSB	Load byte data from memory, sign extend it, and put it to register
LDRSH	Load half word data from memory, sign extend it, and put it to register
LDM	Load multiple data from memory to registers
LDRD	Load double word data from memory to registers
STR	Store word to memory
STRB	Store byte data to memory
STRH	Store half word data to memory
STM	Store multiple words from registers to memory
STRD	Store double word data from registers to memory
PUSH	Push multiple registers
POP	Pop multiple registers

Table 4.8 32-Bit Branch Instructions

Instruction	Function
B	Branch
BL	Branch and link
TBB	Table branch byte; forward branch using a table of single byte offset
TBH	Table branch half word; forward branch using a table of half word offset

Table 4.9 Other 32-Bit Instructions

Instruction	Function
LDREX	Exclusive load word
LDREXH	Exclusive load half word
LDREXB	Exclusive load byte
STREX	Exclusive store word
STREXH	Exclusive store half word
STREXB	Exclusive store byte
CLREX	Clear the local exclusive access record of local processor
MRS	Move special register to general-purpose register
MSR	Move to special register from general-purpose register
NOP	No operation
SEV	Send event
WFE	Sleep and wake for event
WFI	Sleep and wake for interrupt
ISB	Instruction synchronization barrier
DSB	Data synchronization barrier
DMB	Data memory barrier

Unsupported Instructions

A number of Thumb instructions are not supported in the Cortex-M3; they are presented in Table 4.10.

Table 4.10 Unsupported Thumb Instructions for Traditional ARM Processors

Unsupported Instruction	Function
BLX label	This is branch with link and exchange state. In a format with immediate data, BLX always changes to ARM state. Since the Cortex-M3 does not support the ARM state, instructions like this one that attempt to switch to the ARM state will result in a fault exception called *usage fault*.
SETEND	This Thumb instruction, introduced in architecture v6, switches the endian configuration during run time. Since the Cortex-M3 does not support dynamic endian, using the SETEND instruction will result in a fault exception.

A number of instructions listed in the *ARM v7-M Architecture Application Level Reference Manual* are not supported in the Cortex-M3. ARM v7-M architecture allows Thumb-2 coprocessor instructions, but the Cortex-M3 processor does not have any coprocessor support. Therefore, executing the coprocessor instructions shown in Table 4.11 will result in a fault exception (usage fault with NOCP flag in NVIC set to 1).

Table 4.11 Unsupported Coprocessor Instructions

Unsupported Instruction	Function
MCR	Move to coprocessor from ARM processor
MCR2	Move to coprocessor from ARM processor
MCRR	Move to coprocessor from two ARM register
MRC	Move to ARM register from coprocessor
MRC2	Move to ARM register from coprocessor
MRRC	Move to two ARM registers from coprocessor
LDC	Load coprocessor; load memory data from a sequence of consecutive memory addresses to a coprocessor
STC	Store coprocessor; stores data from a coprocessor to a sequence of consecutive memory addresses

Some of the Change Process State (CPS) instructions are also not supported in the Cortex-M3 (see Table 4.12). This is because the PSR definition has changed, so some bits defined in the ARM architecture v6 are not available in the Cortex-M3.

Table 4.12 Unsupported Change Process State (CPS) Instructions

Unsupported Instruction	Function
CPS<IE|ID>.W A	There is no A bit in the Cortex-M3
CPS.W #mode	There is no mode bit in the Cortex-M3 PSR

In addition, the hint instructions shown in Table 4.13 will behave as NOP in the Cortex-M3.

Table 4.13 Unsupported Hint Instructions

Unsupported Instruction	Function
DBG	A hint instruction to debug and trace system.
PLD	Preload data. This is a hint instruction for cache memory. However, since there is no cache in the Cortex-M3 processor, this instruction behaves as NOP.
PLI	Preload instruction. This is a hint instruction for cache memory. However, since there is no cache in the Cortex-M3 processor, this instruction behaves as NOP.
YIELD	A hint instruction to allow multithreading software to indicate to hardware that it is doing a task that can be swapped out to improve overall system performance.

All other undefined instructions, when executed, will cause the usage fault exception to take place.

Instruction Descriptions

Here we introduce some of the commonly used syntax for ARM assembly code. Some of the instructions have various options such as barrel shifter; these will not be fully covered in this chapter.

Assembler Language: Moving Data

One of the most basic functions in a processor is transfer of data. In the Cortex-M3, data transfers can be of one of the following types:

- Moving data between register and register
- Moving data between memory and register
- Moving data between special register and register
- Moving an immediate data value into a register

The command to move data between registers is MOV (move). For example, moving data from register R3 to register R8 looks like this:

```
MOV R8, R3
```

Another instruction can generate the negative value of the original data; it is called MVN (move negative).

The basic instructions for accessing memory are Load and Store. Load (LDR) transfers data from memory to registers, and Store transfers data from registers to memory. The transfers can be in different data sizes (byte, half word, word, and double word), as outlined in Table 4.14.

Table 4.14 Commonly Used Memory Access Instructions

Example	Description
`LDRB Rd, [Rn, #offset]`	Read byte from memory location Rn + offset
`LDRH Rd, [Rn, #offset]`	Read half-word from memory location Rn + offset
`LDR  Rd, [Rn, #offset]`	Read word from memory location Rn + offset
`LDRD Rd1,Rd2, [Rn, #offset]`	Read double word from memory location Rn + offset
`STRB Rd, [Rn, #offset]`	Store byte to memory location Rn + offset
`STRH Rd, [Rn, #offset]`	Store half word to memory location Rn + offset
`STR  Rd, [Rn, #offset]`	Store word to memory location Rn + offset
`STRD Rd1,Rd2, [Rn, #offset]`	Store double word to memory location Rn + offset

Multiple Load and Store operations can be combined into single instructions called LDM (Load Multiple) and STM (Store Multiple), as outlined in Table 4.15.

Table 4.15 Multiple Memory Access Instructions

Example	Description
`LDMIA Rd!,<reg list>`	Read multiple words from memory location specified by *Rd*. Address Increment After (IA) each transfer (16-bit Thumb instruction).
`STMIA Rd!,<reg list>`	Store multiple words to memory location specified by *Rd*. Address Increment After (IA) each transfer (16-bit Thumb instruction).
`LDMIA.W Rd(!),<reg list>`	Read multiple words from memory location specified by *Rd*. Address increment after each read (.W specified it is a 32-bit Thumb-2 instruction).
`LDMDB.W Rd(!),<reg list>`	Read multiple words from memory location specified by *Rd*. Address Decrement Before (DB) each read (.W specified it is a 32-bit Thumb-2 instruction).
`STMIA.W Rd(!),<reg list>`	Write multiple words to memory location specified by *Rd*. Address increment after each read (.W specified it is a 32-bit Thumb-2 instruction).
`STMDB.W Rd(!),<reg list>`	Write multiple words to memory location specified by *Rd*. Address Decrement Before each read (.W specified it is a 32-bit Thumb-2 instruction).

The exclamation mark (!) in the instruction specifies whether the register *Rd* should be updated after the instruction is completed. For example, if R8 equals 0x8000:

```
STMIA.W  R8!, {R0-R3} ; R8 changed to 0x8010 after store
                      ; (increment by 4 words)
STMIA.W  R8 , {R0-R3} ; R8 unchanged after store
```

ARM processors also support memory accesses with pre-indexing and post-indexing. For pre-indexing, the register holding the memory address is adjusted. The memory transfer then takes place with the updated address. For example:

```
LDR.W  R0,[R1, #offset]! ; Read memory[R1+offset], with R1
                         ; update to R1+offset
```

The use of the ! indicates the update of base register R1. The ! is optional; without it, the instruction would be just a normal memory transfer with offset from a base address. The pre-indexing memory access instructions include load and store instructions of various transfer sizes (see Table 4.16).

Table 4.16 Examples of Pre-Indexing Memory Access Instructions

Example	Description

`LDR.W   Rd,[Rn, #offset]!`
`LDRB.W  Rd,[Rn, #offset]!`
`LDRH.W  Rd,[Rn, #offset]!`
`LDRD.W  Rd1,Rd2,[Rn, #offset]!`
	Pre-indexing load instructions for various sizes (word, byte, half word, and double word)

`LDRSB.W Rd,[Rn, #offset]!`
`LDRSH.W Rd,[Rn, #offset]!`
	Pre-indexing load instructions for various sizes with sign extend (byte, half word)

Table 4.16 (Continued)

Example	Description

`STR.W   Rd,[Rn, #offset]!`
`STRB.W  Rd,[Rn, #offset]!`
`STRH.W  Rd,[Rn, #offset]!`
`STRD.W  Rd1,Rd2,[Rn, #offset]!`
	Pre-indexing store instructions for various sizes (word, byte, half word, and double word)

Post-indexing memory access instructions carry out the memory transfer using the base address specified by the register and then update the address register afterward. For example:

```
LDR.W R0,[R1], #offset ; Read memory[R1], with R1
                       ; updated to R1+offset
```

When a post-indexing instruction is used, there is no need to use the ! sign, because all post-indexing instructions update the base address register, whereas in pre-indexing you might choose whether to update the base address register or not.

Similarly to pre-indexing, post-indexing memory access instructions are available for different transfer sizes (see Table 4.17).

Table 4.17 Examples of Post-Indexing Memory Access Instructions

Example	Description

`LDR.W   Rd,[Rn], #offset`
`LDRB.W  Rd,[Rn], #offset`
`LDRH.W  Rd,[Rn], #offset`
`LDRD.W  Rd1,Rd2,[Rn], #offset`
	Post-indexing load instructions for various sizes (word, byte, half word, and double word)

`LDRSB.W Rd,[Rn], #offset`
`LDRSH.W Rd,[Rn], #offset`
	Post-indexing load instructions for various sizes with sign extend (byte, half word)

`STR.W   Rd,[Rn], #offset`
`STRB.W  Rd,[Rn], #offset`
`STRH.W  Rd,[Rn], #offset`
`STRD.W  Rd1,Rd2,[Rn], #offset`
	Post-indexing store instructions for various sizes (word, byte, half word, and double word)

Two other types of memory operation are stack PUSH and stack POP. For example:

```
     PUSH {R0, R4-R7, R9} ; Push R0, R4, R5, R6, R7, R9 into
                          ; stack memory
     POP  {R2,R3}         ; Pop R2 and R3 from stack
```

Usually a PUSH instruction will have a corresponding POP with the same register list, but this is not always necessary. For example, a common exception is when POP is used as a function return:

```
     PUSH {R0-R3, LR} ; Save register contents at beginning of
                      ; subroutine
     ....             ; Processing
     POP {R0-R3, PC}  ; restore registers and return
```

In this case, instead of popping the LR register back and then branching to the address in LR, we POP the address value directly in the program counter.

As mentioned in Chapter 3, the Cortex-M3 has a number of special registers. To access these registers, we use the instructions MRS and MSR. For example:

```
     MRS R0, PSR      ; Read Processor status word into R0
     MSR CONTROL, R1 ; Write value of R1 into control register
```

Unless you're accessing the APSR, you can use MSR or MRS to access other special registers only in privileged mode.

Moving immediate data into a register is a common thing to do. For example, you might want to access a peripheral register, so you need to put the address value into a register beforehand. For small values (8 bits or less), you can use MOV (move). For example:

```
     MOV  R0, #0x12 ; Set R0 to 0x12
```

For a larger value (over 8 bits), you might need to use a Thumb-2 move instruction. For example:

```
     MOVW.W  R0,#0x789A ; Set R0 to 0x789A
```

Or if the value is 32-bit, you can use two instructions to set the upper and lower halves:

```
     MOVW.W  R0,#0x789A ; Set R0 lower half to 0x789A
     MOVT.W  R0,#0x3456 ; Set R0 upper half to 0x3456. Now
                        ; R0=0x3456789A
```

Alternatively, you can also use LDR (a pseudo instruction provided in ARM assembler). For example:

```
     LDR  R0, =0x3456789A
```

This is not a real assembler command, but the ARM assembler will convert it into a PC relative load instruction to produce the required data. To generate 32-bit immediate data, using LDR is recommended rather than the MOVW.W and MOVT.W combination because it gives better readability and the assembler might be able to reduce the memory being used if the same immediate data are reused in several places of the same program.

LDR and ADR Pseudo Instructions

Both LDR and ADR pseudo instructions can be used to set registers to a program address value. They have different syntaxes and behaviors. For LDR, if the address is a program address value, it will automatically set the LSB to 1. For example:

```
     LDR  R0, =address1 ; R0 set to 0x4001
     ...
address1
0x4000: MOV  R0, R1 ; address1 contains program code
     ...
```

You will find that the LDR instruction will put 0x4001 into R1; the LSB is set to 1 to indicate that it is Thumb code. If *address1* is a data address, LSB will not be changed. For example:

```
     LDR R0, =address1 ; R0 set to 0x4000
     ...
address1
0x4000: DCD 0x0 ; address1 contains data
     ...
```

For ADR, you can load the address value of a program code into a register without setting the LSB automatically. For example:

```
     ADR R0, address1
     ...
address1
0x4000: MOV R0, R1 ; address1 contains program code
     ...
```

You will get 0x4000 in the ADR instruction. Note that there is no equal sign (=) in the ADR statement.

LDR obtains the immediate data by putting the data in the program code and uses a PC relative load to get the data into the register. ADR tries to generate the immediate value by adding or subtracting instructions (for example, based on the current PC value). As a result, it is not possible to create all immediate values using ADR, and the target address label must be in a close range. However, using ADR can generate smaller code sizes compared to LDR.

Assembler Language: Processing Data

The Cortex-M3 provides many different instructions for data processing. A few basic ones are introduced here. Many data operation instructions can have multiple instruction formats. For example, an ADD instruction can operate between two registers or between one register and an immediate data value:

```
     ADD   R0, R1     ; R0 = R0+R1
     ADD   R0, #0x12  ; R0 = R0 + 0x12
     ADD.W R0, R1, R2 ; R0 = R1+R2
```

These are all ADD instructions, but they have different syntaxes and binary coding.

When 16-bit Thumb code is used, an ADD instruction changes the flags in the PSR. However, 32-bit Thumb-2 code can either change a flag or keep it unchanged. To separate the two different operations, the *S* suffix should be used if the following operation depends on the flags:

```
     ADD.W  R0, R1, R2 ; Flag unchanged
     ADDS.W R0, R1, R2 ; Flag change
```

Aside from ADD instructions, the arithmetic functions that the Cortex-M3 supports include SUB (subtract), MUL (multiply), and UDIV/SDIV (unsigned and signed divide). Table 4.18 shows some of the most commonly used arithmetic instructions.

Table 4.18 Examples of Arithmetic Instructions

Instruction	Operation
ADD Rd, Rn, Rm ; Rd = Rn + Rm ADD Rd, Rm ; Rd = Rd + Rm ADD Rd, #immed ; Rd = Rd + #immed	ADD operation
ADC Rd, Rn, Rm ; Rd = Rn + Rm + carry ADC Rd, Rm ; Rd = Rd + Rm + carry ADC Rd, #immed ; Rd = Rd + #immed + ; carry	ADD with carry
ADDW Rd, Rn,#immed ; Rd = Rn + #immed	ADD register with 12-bit immediate value
SUB Rd, Rn, Rm ; Rd = Rn – Rm SUB Rd, #immed ; Rd = Rd – #immed SUB Rd, Rn,#immed ; Rd = Rn –#immed	SUBTRACT
SBC Rd, Rm ; Rd = Rd - Rm – ; carry flag SBC.W Rd, Rn, #immed ; Rd = Rn – #immed - ; carry flag SBC.W Rd, Rn, Rm ; Rd = Rn – Rm – ; carry flag	SUBTRACT with borrow (carry)
RSB.W Rd, Rn, #immed ; Rd = #immed –Rn RSB.W Rd, Rn, Rm ; Rd = Rm - Rn MUL Rd, Rm ; Rd = Rd * Rm MUL.W Rd, Rn, Rm ; Rd = Rn * Rm	Reverse subtract Multiply
UDIV Rd, Rn, Rm ; Rd = Rn /Rm SDIV Rd, Rn, Rm ; Rd = Rn /Rm	Unsigned and signed divide

The Cortex-M3 also supports 32-bit multiply instructions and multiply accumulate instructions that give 64-bit results. These instructions support signed or unsigned values (see Table 4.19).

Table 4.19 32-bit Multiply Instructions

Instruction	Operation
SMULL RdLo, RdHi, Rn, Rm ;{RdHi,RdLo} = Rn * Rm SMLAL RdLo, RdHi, Rn, Rm ;{RdHi,RdLo} += Rn * Rm	32-bit multiply instructions for signed values

Table 4.19 (Continued)

Instruction	Operation

`UMULL RdLo, RdHi, Rn, Rm`
`;{RdHi,RdLo} = Rn * Rm`
`UMLAL RdLo, RdHi, Rn, Rm`
`;{RdHi,RdLo} += Rn * Rm`
	32-bit multiply instructions for unsigned values

Another group of data processing instructions are the logical operations instructions and logical operations such as AND, ORR (or), and shift and rotate functions. Table 4.20 shows some of the most commonly used logical instructions.

Table 4.20 Logic Operation Instructions

Instruction	Operation

`AND    Rd, Rn        ; Rd = Rd & Rn`
`AND.W Rd, Rn,#immed; Rd = Rn & #immed`
`AND.W Rd, Rn, Rm    ; Rd = Rn & Rd`
	Bitwise AND

`ORR    Rd, Rn        ; Rd = Rd | Rn`
`ORR.W Rd, Rn,#immed; Rd = Rn | #immed`
`ORR.W Rd, Rn, Rm    ; Rd = Rn | Rd`
	Bitwise OR

`BIC    Rd, Rn        ; Rd = Rd & (~Rn)`
`BIC.W Rd, Rn,#immed; Rd = Rn &(~#immed)`
`BIC.W Rd, Rn, Rm    ; Rd = Rn &(~Rd)`
	Bit clear

`ORN.W Rd, Rn,#immed; Rd = Rn |(~#immed)`
`ORN.W Rd, Rn, Rm    ; Rd = Rn |(~Rd)`
	Bitwise OR NOT

`EOR    Rd, Rn        ; Rd = Rd ^ Rn`
`EOR.W Rd, Rn,#immed; Rd = Rn | #immed`
`EOR.W Rd, Rn, Rm    ; Rd = Rn | Rd`
	Bitwise Exclusive OR

The Cortex-M3 provides rotate and shift instructions. In some cases, the rotate operation can be combined with other operations (for example, in memory address offset calculation for load/store instructions). For standalone rotate/shift operations, the instructions shown in Table 4.21 are provided.

Table 4.21 Shift and Rotate Instructions

Instruction	Operation

`ASR    Rd, Rn,#immed; Rd = Rn >> immed`
`ASR    Rd, Rn        ; Rd = Rd >> Rn`
`ASR.W Rd, Rn, Rm    ; Rd = Rn >> Rm`
	Arithmetic shift right

(Continued)

Table 4.21 (Continued)

Instruction	Operation

`LSL   Rd, Rn,#immed; Rd = Rn << immed`
`LSL   Rd, Rn      ; Rd = Rd << Rn`
`LSL.W Rd, Rn, Rm  ; Rd = Rn << Rm`
	Logical shift left

`LSR   Rd, Rn,#immed; Rd = Rn >> immed`
`LSR   Rd, Rn      ; Rd = Rd >> Rn`
`LSR.W Rd, Rn, Rm  ; Rd = Rn >> Rm`
	Logical shift right

`ROR   Rd, Rn      ; Rd rot by Rn`
`ROR.W Rd, Rn, Rm  ; Rd = Rn rot by Rm`
	Rotate right

`RRX.W Rd, Rn      ; {C, Rd} = {Rn, C}`	Rotate right extended

The rotate and shift operations can also update the carry flag if the *S* suffix is used (and always update the carry flag if the 16-bit Thumb code is used). See Figure 4.1.

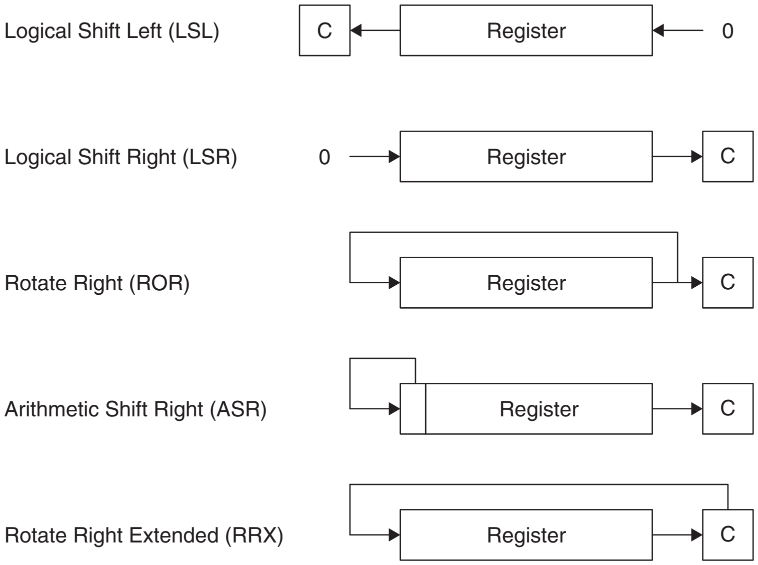


Figure 4.1 Shift and Rotate Instructions

If the shift or rotate operation shifts the register position by multiple bits, the value of the carry flag *C* will be the last bit that shifts out of the register.

Why Is There Rotate Right But No Rotate Left?

The rotate left operation can be replaced by a rotate right operation with a different rotate offset. For example, a rotate left by 4-bit operation can be written as a rotate right by 28-bit instruction, which gives the same result and takes the same amount of time to execute.

For conversion of signed data from byte or half word to word, the Cortex-M3 provides the two instructions shown in Table 4.22.

Table 4.22 Sign Extend Instructions

Instruction	Operation
`SXTB.W Rd, Rm ; Rd = signext(Rn[7:0])`	Sign extend byte data into word
`SXTH.W Rd, Rm ; Rd = signext(Rn[15:0])`	Sign extend half word data into word

Another group of data processing instructions is used for reversing data bytes in a register (see Table 4.23). These instructions are usually used for conversion between little endian and big endian data.

Table 4.23 Data Reverse Ordering Instructions

Instruction	Operation
`REV.W    Rd, Rn ; Rd = rev(Rn)`	Reverse bytes in word
`REV16.W <Rd>, <Rn> ; Rd = rev16(Rn)`	Reverse bytes in each half word
`REVSH.W <Rd>, <Rn> ; Rd = revsh(Rn)`	Reverse bytes in bottom half word and sign extend the result

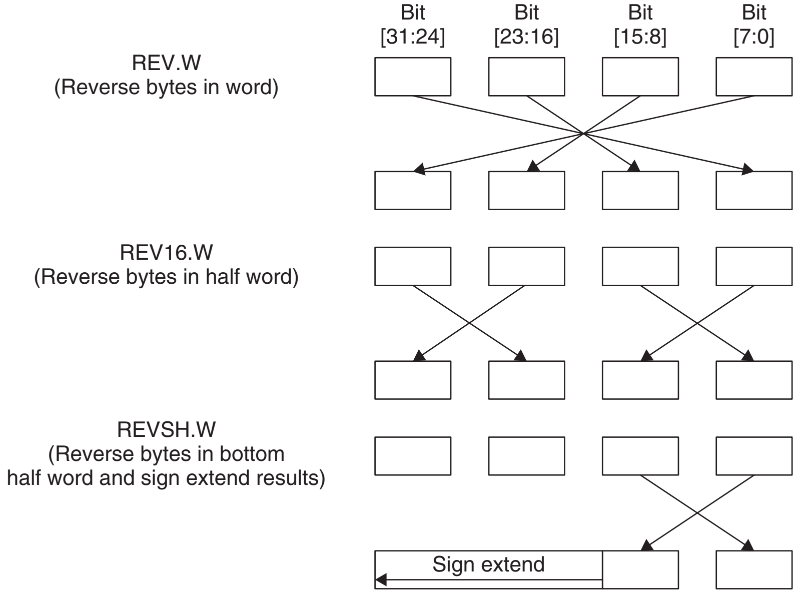


Figure 4.2 Reverse Operations

The last group of data processing instructions is for bit field processing. They include the instructions shown in Table 4.24. Examples of these instructions are provided in a later part of this chapter.

Table 4.24 Bit Field Processing and Manipulation Instructions

Instruction	Operation
`BFC.W  Rd, Rn, #<width>`	Clear bit field within a register
`BFI.W  Rd, Rn, #<lsb>, #<width>`	Insert bit field to a register
`CLZ.W  Rd, Rn`	Count leading zero
`RBIT.W Rd, Rn`	Reverse bit order in register
`SBFX.W Rd, Rn, #<lsb>, #<width>`	Copy bit field from source and sign extend it
`UBFX.W Rd, Rn, #<lsb>, #<width>`	Copy bit field from source register

Assembler Language: Call and Unconditional Branch

The most basic branch instructions are:

```
B  label ; Branch to a labeled address
BX reg   ; Branch to an address specified by a register
```

In BX instructions, the LSB of the value contained in the register determines the next state (Thumb/ARM) of the processor. In the Cortex-M3, since it is always in Thumb state, this bit should be set to 1. If it is zero, the program will cause a usage fault exception because it is trying to switch the processor into ARM state.

To call a function, the branch and link instructions should be used:

```
BL  label    ; Branch to a labeled address and save return
             ; address in LR
BLX reg      ; Branch to an address specified by a register and
             ; save return
             ; address in LR.
```

With these instructions, the return address will be stored in the link register (LR) and the function can be terminated using BX LR, which causes program control to return to the calling process. However, when using BLX, make sure that the LSB of the register is 1. Otherwise the processor will produce a fault exception because it is an attempt to switch to the ARM state.

You can also carry out a branch operation using MOV instructions and LDR instructions. For example:

```
MOV R15,  R0     ; Branch to an address inside R0
LDR R15, [R0]    ; Branch to an address in memory location
                 ; specified by R0
POP {R15}        ; Do a stack pop operation, and change the
                 ; program counter value
                 ; to the result value.
```

When using these methods to carry out branches, you also need to make sure that the LSB of the new program counter value is 0x1. Otherwise a usage fault exception will be generated because it will try to switch the processor to ARM mode, which is not allowed in the Cortex-M3.

Save the LR if You Need to Call a Subroutine

The BL instruction will destroy the current content of your LR register. So, if your program code needs the LR register later, you should save your LR before you use BL. The common method is to push the LR to stack in the beginning of your subroutine. For example:

```
main
      ...
      BL functionA
      ...
functionA
      PUSH {LR}  ; Save LR content to stack
      ...
      BL functionB
      ...
      POP  {PC}  ; Use stacked LR content to return to main
functionB
      PUSH {LR}
      ...
      POP  {PC}  ; Use stacked LR content to return to functionA
```

In addition, if the subroutine you call is a C function, you might also need to save the contents in R0–R3 and R12 if these values will be needed at a later stage. According to *AAPCS* (Ref 5), the contents in these registers could be changed by a C function.

Assembler Language: Decisions and Conditional Branches

Most conditional branches in ARM processors use flags in the Application Program Status Register (APSR) to determine whether a branch should be carried out. In the APSR, there are five flag bits; four of them are used for branch decisions (see Table 4.25).

Table 4.25 Flag bits in APSR That Can Be Used for Conditional Branches

Flag	PSR Bit	Description
N	31	Negative flag (last operation result is a negative value)
Z	30	Zero (last operation result returns a zero value)
C	29	Carry (last operation returns a carry out or borrow)
V	28	Overflow (last operation results in an overflow)

There is another flag bit at bit[27], called the *Q flag*. It is for saturation math operations and is not used for conditional branches.

Flags in ARM Processors

Often, data processing instructions change the flags in the PSR. The flags might be used for branch decisions, or they can be used as part of the input for the next instruction. The ARM processor normally contains at least the *Z, N, C,* and *V* flags, which are updated by execution of data processing instructions:

- Z (Zero) flag: This flag is set when the result of an instruction has a zero value or when a comparison of two data returns an equal result.
- N (Negative) flag: This flag is set when the result of an instruction has a negative value (bit 31 is 1).
- C (Carry) flag: This flag is for unsigned data processing—for example, in add (ADD) it is set when an overflow occurs; in subtract (SUB) it is set when a borrow did not occur (borrow is the invert of carry).
- V (Overflow) flag: This flag is for signed data processing; for example, in an add (ADD), when two positive values added together produce a negative value, or when two negative values added together produce a positive value.

These flags can also have special results when used with shift and rotate instructions. Refer to the *ARM v7-M Architecture Application Level Reference Manual* (Ref 2) for details.

Table 4.26 Conditions for Branches or Other Conditional Operations

Symbol	Condition	Flag
EQ	Equal	Z set
NE	Not equal	Z clear
CS/HS	Carry set/unsigned higher or same	C set
CC/LO	Carry clear/unsigned lower	C clear
MI	Minus/negative	N set
PL	Plus/positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N set or V set, or N clear and V clear (N == V)

Table 4.26 (Continued)

Symbol	Condition	Flag
LT	Signed less than	N set and V clear, or N clear and V set (N != V)
GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
AL	Always (unconditional)	—

With combinations of the four flags (*N, Z, C*, and *V*), 15 branch conditions are defined (see Table 4.26). Using these conditions, branch instructions can be written as, for example:

```
BEQ label ; Branch to address 'label' if Z flag is set
```

You can also use the Thumb-2 version if your branch target is further away. For example:

```
BEQ.W label ; Branch to address 'label' if Z flag is set
```

The defined branch conditions can also be used in IF-THEN-ELSE structures. For example:

```
CMP R0, R1    ; Compare R0 and R1
ITTEE GT      ; If R0 > R1 Then (first 2 statements execute
              ; if true,
              ; other 2 statements execute if false)
MOVGT R2, R0 ;      R2 = R0
MOVGT R3, R1 ;      R3 = R1
MOVLE R2, R0 ; Else R2 = R1
MOVLE R3, R1 ;      R3 = R0
```

PSR flags can be affected by the following:

- 16-bit ALU instructions
- 32-bit (Thumb-2) ALU instructions with the *S* suffix; for example, ADDS.W
- Compare (e.g., CMP) and Test (e.g., TST, TEQ)
- Write to APSR/PSR directly

Most of the 16-bit Thumb arithmetic instructions affect the *N, Z, C,* and *V* flags. With 32-bit Thumb-2 instructions, the ALU operation can either change flags or not change flags. For example:

```
ADDS.W  R0, R1, R2  ; This 32-bit Thumb-2 instruction update flag
ADD.W   R0, R1, R2  ; This 32-bit Thumb-2 instruction do not
                    ; update flag
ADDS    R0, R1      ; This 16-bit Thumb instruction update flag
ADD     R0, #0x1    ; This 16-bit Thumb instruction update flag
```

Be careful when changing an ALU instruction between Thumb and Thumb-2. Without the *S* suffix in the instruction, a Thumb instruction might update a flag, whereas a Thumb-2 instruction does not, so you can end up with different results. To make sure that the code works with different tools, you should always use the *S* suffix if the flags need to be updated for conditional operations such as conditional branches.

The CMP (Compare) instruction subtracts two values and updates the flags (just like SUBS), but the result is not stored in any registers. CMP can have the following formats:

```
      CMP R0, R1    ; Calculate R0 – R1 and update flag
      CMP R0, #0x12 ; Calculate R0 – 0x12 and update flag
```

A similar instruction is the CMN (Compare Negative). It compares one value to the negative (two's complement) of a second value; the flags are updated, but the result is not stored in any registers:

```
      CMN R0, R1    ; Calculate R0 – (-R1) and update flag
      CMN R0, #0x12 ; Calculate R0 – (-0x12) and update flag
```

The TST (Test) instruction is more like the AND instruction. It ANDs two values and updates the flags. However, the result is not stored in any register. Similarly to CMP, it has two input formats:

```
      TST R0, R1    ; Calculate R0 and R1 and update flag
      TST R0, #0x12 ; Calculate R0 and 0x12 and update flag
```

Assembler Language: Combined Compare and Conditional Branch

With ARM architecture v7-M, two new instructions are provided on the Cortex-M3 to supply a simple compare with zero and conditional branch operations. These are CBZ (Compare and Branch if Zero) and CBNZ (Compare and Branch if Nonzero).

The compare and branch instructions only support forward branches. For example:

```
      i = 5;
      while (i != 0 ){
      func1(); ; call a function
      i--;
      }
```

This can be compiled into:

```
      MOV  R0, #5         ; Set loop counter
loop1 CBZ  R0, loop1exit  ; if loop counter = 0 then exit the loop
      BL   func1          ; call a function
      SUB  R0, #1         ; loop counter decrement
      B    loop1          ; next loop
loop1exit
```

Assembler Language: Conditional Branches Using IT Instructions

The IT (IF-THEN) block is very useful for handling small conditional code. It avoids branch penalties because there is no change to program flow. It can provide a maximum of four conditionally executed instructions.

In IT instruction blocks, the first line must be the IT instruction, detailing the choice of execution, followed by the condition it checks. The first statement after the IT command must be TRUE-THEN-EXECUTE, which is always written as *ITxxx*, where *T* means THEN and *E* means ELSE. The second through fourth statements can be either THEN (true) or ELSE (false):

```
IT<x><y><z>  <cond>                        ; IT instruction (<x>, <y>,
                                           ; <z> can be T or E)
instr1<cond>                 <operands>    ; 1st instruction (<cond>
                                           ; must be same as IT)
instr2<cond or not cond> <operands>        ; 2nd instruction (can be
                                           ; <cond> or <!cond>
instr3<cond or not cond> <operands>        ; 3rd instruction (can be
                                           ; <cond> or <!cond>
instr4<cond or not cond> <operands>        ; 4th instruction (can be
                                           ; <cond> or <!cond>
```

If a statement is to be executed when <*cond* > is false, the suffix for the instruction must be the opposite of the condition. For example, the opposite of EQ is NE, the opposite of GT is LE, and so on. The following code shows an example of a simple conditional execution:

```
if (R1<R2) then
   R2=R2-R1
   R2=R2/2
else
   R1=R1-R2
   R1=R1/2
```

In assembly:

```
CMP      R1, R2         ; If R1 < R2 (less then)
ITTEE    LT             ; then execute instruction 1 and 2
                        ; (indicated by T)
                        ; else execute instruction 3 and 4
                        ; (indicated by E)
SUBLT.W  R2,R1          ; 1st instruction
LSRLT.W  R2,#1          ; 2nd instruction
SUBGE.W  R1,R2          ; 3rd instruction (notice the GE is
                        ; opposite of LT)
LSRGE.W  R1,#1          ; 4th instruction
```

You can have fewer than four conditionally executed instructions. The minimum is 1. You need to make sure the number of *T* and *E* occurrences in the IT instruction matches the number of conditionally executed instructions after the IT.

If an exception occurs during the IT instruction block, the execution status of the block will be stored in the stacked PSR (in the IT/ICI bit field). So, when the exception handler completes and the IT block resumes, the rest of the instructions in the block can continue the execution correctly. In the case of using multicycle instructions (for example, multiple load and store) inside an IT block, if an exception takes place during the execution, the whole instruction must be completed before the exception is accepted.

Assembler Language: Instruction Barrier and Memory Barrier Instructions

The Cortex-M3 supports a number of barrier instructions. These instructions are needed as memory systems get more and more complex. In some cases, if memory barrier instructions are not used, race conditions could occur.

For example, if the memory map can be switched by a hardware register, after writing to the memory switching register you should use the DSB instruction. Otherwise, if the write to the memory switching register is buffered and takes a few cycles to complete, and the next instruction accesses the switched memory region immediately, the access could be using the old memory map. In some cases, this might result in an invalid access if the memory switching and memory access happen at the same time. Using DSB in this case will make sure that the write to the memory map switching register is completed before a new instruction is executed.

There are three barrier instructions in the Cortex-M3:

- DMB
- DSB
- ISB

These instructions are described in Table 4.27.

Table 4.27 Barrier Instructions

Instruction	Description
DMB	Data Memory Barrier; ensures that all memory accesses are completed before new memory access is committed
DSB	Data Synchronization Barrier; ensures that all memory accesses are completed before next instruction is executed
ISB	Instruction Synchronization Barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

When you do a data write followed immediately by a read on a dual-port memory, if the memory write is buffered, the DMB instruction can be used to ensure the read gets the updated value.

The DSB and ISB instructions can be important for self-modifying code. For example, if a program changes its own program code, the next executed instruction should be based on the updated program. However, since the processor is pipelined, the modified instruction location might have already been fetched. Using DSB and then ISB can ensure that the modified program code is fetched again.

More detail about memory barriers can be found in the *ARM v7-M Architecture Application Level Reference Manual* (Ref 2).

Assembly Language: Saturation Operations

The Cortex-M3 supports two instructions that provide signed and unsigned saturation operations: SSAT and USAT (for signed data type and unsigned data type, respectively). Saturation is commonly used in signal processing—for example, in signal amplification. When an input signal is amplified, there is a chance that the output will be larger than the allowed output range. If the value is adjusted simply by removing the unused MSB, an overflowed result will cause the signal waveform to be completely deformed (see Figure 4.3).

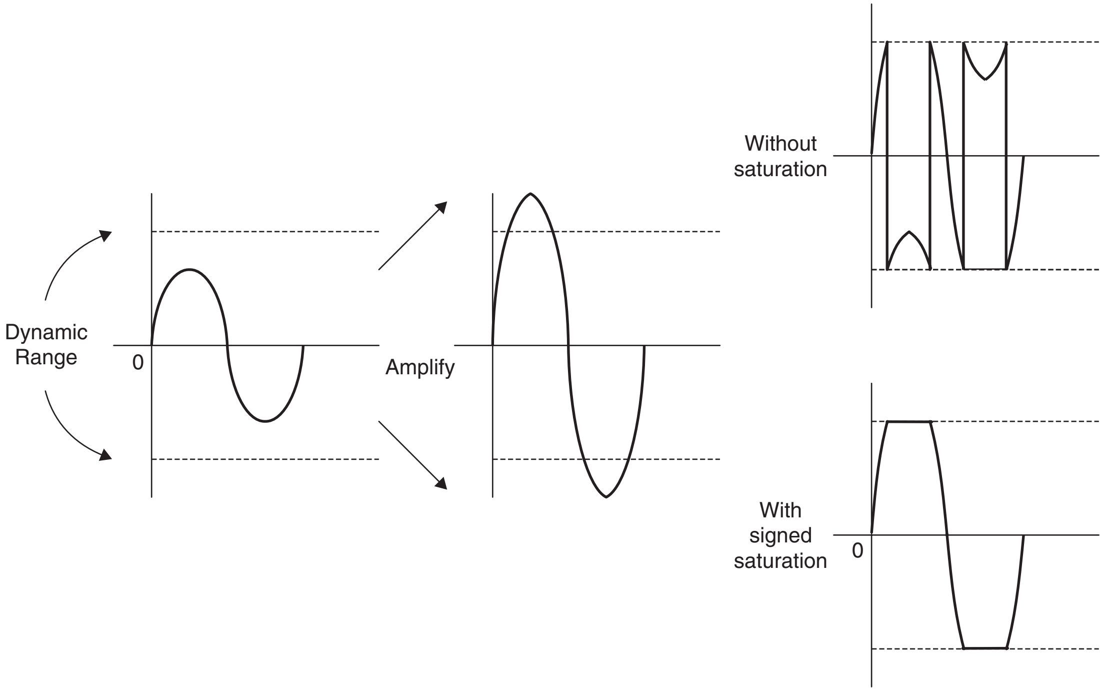


Figure 4.3 Signed Saturation Operation

The saturation operation does not prevent the distortion of the signal, but at least the amount of distortion is greatly reduced in the signal waveform.

The instruction syntax of the SSAT and USAT instructions is outlined here and in Table 4.28:

- Rn: Input value
- Shift: Shift operation for input value before saturation; optional, can be *#LSL N* or *#ASR N*
- Immed: Bit position where the saturation is carried out
- Rd: Destination register

Table 4.28 Saturation Instructions

Instruction	Description
SSAT.W <Rd>, #<immed>, <Rn>, {,<shift>}	Saturation for signed value
USAT.W <Rd>, #<immed>, <Rn>, {,<shift>}	Saturation for a signed value into an unsigned value

Besides the destination register, the Q-bit in the APSR can also be affected by the result. The Q flag is set if saturation takes place in the operation, and it can be cleared by writing to the APSR (see Table 4.29). For example, if a 32-bit signed value is to be saturated into a 16-bit signed value, the following instruction can be used:

```
SSAT.W R1, #16, R0
```

Table 4.29 Examples of Signed Saturation Results

Input (R0)	Output (R1)	Q Bit
0x00020000	0x00007FFF	Set
0x00008000	0x00007FFF	Set
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0xFFFF8000	Unchanged
0xFFFF8001	0xFFFF8000	Set
0xFFFE0000	0xFFFF8000	Set

Similarly, if a 32-bit signed value is to saturate into a 16-bit unsigned value, the following instruction can be used:

```
USAT.W R1, #16, R0
```

This will provide a saturation feature that has the properties shown in Figure 4.4.

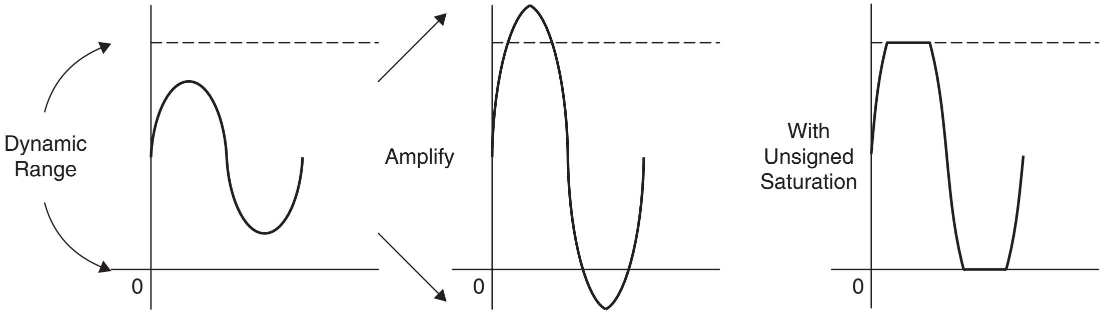


Figure 4.4 Unsigned Saturation Operation

For the preceding 16-bit saturation example instruction, the output values shown in Table 4.30 can be observed.

Table 4.30 Examples of Unsigned Saturation Results

Input (R0)	Output (R1)	Q Bit
0x00020000	0x0000FFFF	Set
0x00008000	0x00008000	Set
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0x00000000	Set
0xFFFF8001	0x00000000	Set
0xFFFFFFFF	0x00000000	Set

Saturation instructions can also be used for data type conversions. For example, they can be used to convert a 32-bit integer value to 16-bit integer value. However, C compilers might not be able to directly use these instructions, so assembler functions (or embedded/inline assembler code) for the data conversion could be required.

Several Useful Instructions in the Cortex-M3

Several useful Thumb-2 instructions from the architecture v7 and v6 are introduced here.

MSR and MRS

These two instructions provide access to the special registers in the Cortex-M3. Here is the syntax of these instructions:

```
MRS <Rn>, <SReg> ; Move from Special Register
MSR <SReg>, <Rn> ; Write to Special Register
```

where <*SReg*> could be one of the options shown in Table 4.31.

Table 4.31 Special Register Names for MRS and MSR Instructions

Symbol	Description
IPSR	Interrupt status register
EPSR	Execution status register (read as zero)
APSR[2]	Flags from previous operation
IEPSR	A composite of IPSR and EPSR
IAPSR	A composite of IPSR and APSR
EAPSR	A composite of EPSR and APSR
PSR	A composite of APSR, EPSR and IPSR
MSP	Main stack pointer
PSP	Process stack pointer
PRIMASK	Normal exception mask register
BASEPRI	Normal exception priority mask register
BASEPRI_MAX	Same as normal exception priority mask register, with conditional write (new priority level must be higher than the old level)
FAULTMASK	Fault exception mask register (also disables normal interrupts)
CONTROL	Control register

For example, the following code can be used to set up the Process Stack Pointer:

```
    LDR R0,=0x20008000 ; new value for Process Stack Pointer (PSP)
    MSR PSP, R0
```

Unless accessing the APSR, the MRS and MSR instructions can be used in privileged mode only. Otherwise the operation will be ignored, and the returned read data (if MRS is used) will be zero.

IF-THEN

The IF-THEN (IT) instructions allow up to four succeeding instructions (called an *IT block*) to be conditionally executed. They are in the following formats:

```
IT<x>         <cond>
IT<x><y>      <cond>
IT<x><y><z>   <cond>
```

where:

- <*x*> specifies the execution condition for the second instruction
- <*y*> specifies the execution condition for the third instruction
- <*z*> specifies the execution condition for the fourth instruction

[2] In older ARM Cortex-M3 documents, the APSR was called FPSR. If you are using older software development tools that were developed during the early stages of Cortex-M3 development, you might need to use the register name FPSR in your assembly code.

- <*cond*> specifies the base condition of the instruction block; the first instruction following IT executes if <*cond*> is true

Each of <*x*>, <*y*>, and <*z*> can be either *T* (THEN) or *E* (ELSE), which refers to the base condition <*cond*>, whereas <*cond*> uses traditional syntax such as EQ, NE, GT, or the like.

Here is an example of IT use:

```
if (R0 equal R1) then {
  R3 = R4 + R5
  R3 = R3 / 2
  } else {
  R3 = R6 + R7
  R3 = R3 / 2
  }
```

This can be written as:

```
  CMP   R0, R1     ; Compare R0 and R1
  ITTEE EQ         ; If R0 equal R1, Then-Then-Else-Else
  ADDEQ R3, R4, R5 ; Add if equal
  ASREQ R3, R3, #1 ; Arithmetic shift right if equal
  ADDNE R3, R6, R7 ; Add if not equal
  ASRNE R3, R3, #1 ; Arithmetic shift right if not equal
```

CBZ and CBNZ

The compare and then branch if zero/nonzero instructions are useful for looping (for example, the WHILE loop in C). The syntax is:

```
      CBZ  <Rn>, <label>
```

or:

```
      CBNZ <Rn>, <label>
```

where *label* is a forward branch address. For example:

```
while (R0 != 0) {
    function1();
    }
```

This can be written as:

```
    ...
loop
    CBZ R0, loopexit
    BL  function1
    B   loop
  loopexit
  ...
```

Flags are not affected by this instruction.

SDIV and UDIV

The syntax for signed and Unsigned divide instructions is:

```
SDIV.W <Rd>, <Rn>, <Rm>
UDIV.W <Rd>, <Rn>, <Rm>
```

The result is Rd = Rn/Rm. For example:

```
LDR    R0,=300 ; Decimal 300
MOV    R1,#5
UDIV.W R2, R0, R1
```

This will give you an R2 result of 60 (0x3C).

You can set up the DIVBYZERO bit in the NVIC Configuration Control Register so that when a divide by zero occurs, a fault exception (usage fault) takes place. Otherwise *<Rd>* will become 0 if a divide by zero takes place.

REV, REVH, and REVSH

REV reverses the byte order in a data word, and REVH reverses the byte order inside a half word. For example, if R0 is 0x12345678, in executing the following:

```
REV  R1, R0
REVH R2, R0
```

R1 will become 0x78563412, and R2 will be 0x34127856. REV and REVH are particularly useful for converting data between big endian and little endian.

REVSH is similar to REVH except that it only processes the lower half word, and then it sign extends the result. For example, if R0 is 0x33448899, running:

```
REVSH R1, R0
```

R1 will become 0xFFFF9988.

RBIT

The RBIT instruction reverses the bit order in a data word. The syntax is:

```
RBIT.W <Rd>, <Rn>
```

This instruction is very useful for processing serial bit streams in data communications. For example, if R0 is 0xB4E10C23 (binary value 1011_0100_1110_0001_0000_1100_0010_0011), executing:

```
RBIT.W R0, R1
```

R0 will become 0xC430872D (binary value 1100_0100_0011_0000_1000_0111_0010_1101).

SXTB, SXTH, UXTB, and UXTH

The four instructions SXTB, SXTH, UXTB, and UXTH are used to extend a byte or half word data into a word. The syntax of the instructions is as follows:

```
SXTB  <Rd>, <Rn>
SXTH  <Rd>, <Rn>
UXTB  <Rd>, <Rn>
UXTH  <Rd>, <Rn>
```

For SXTB/SXTH, the data are sign extended using bit[7]/bit[15] of Rn. With UXTB and UXTH, the value is zero extended to 32-bit.

For example, if R0 is 0x55AA8765:

```
SXTB R1,  R0  ; R1 = 0x00000065
SXTH R1,  R0  ; R1 = 0xFFFF8765
UXTB R1,  R0  ; R1 = 0x00000065
UXTH R1,  R0  ; R1 = 0x00008765
```

BFC and BFI

BFC (Bit Field Clear) clears any number of adjacent bits in any position of a register. The syntax of the instruction is:

```
BFC.W <Rd>, <#lsb>, <#width>
```

For example:

```
LDR   R0,=0x1234FFFF
BFC.W R0, #4, #8
```

This will give R0 = 0x1234F00F.

BFI (Bit Field Insert) copies any number of bits (#width) from one register to any location (#lsb) in another register. The syntax is:

```
BFI.W <Rd>, <Rn>, <#lsb>, <#width>
```

For example:

```
LDR   R0,=0x12345678
LDR   R1,=0x3355AACC
BFI.W R1, R0, #8, #16 ; Insert R0[15:0] to R1[23:8]
```

This will give R1 = 0x335678CC.

UBFX and SBFX

UBFX and SBFX are the Unsigned and Signed Bit Field Extract instructions. The syntax of the instructions is:

```
UBFX.W <Rd>, <Rn>, <#lsb>, <#width>
SBFX.W <Rd>, <Rn>, <#lsb>, <#width>
```

UBFX extracts a bit field from a register starting from any location (specified by #lsb) with any width (specified by #width), zero extends it, and puts it in the destination register. For example:

```
      LDR    R0,=0x5678ABCD
      UBFX.W R1, R0, #4, #8
```

This will give R1 = 0x000000BC.

Similarly, SBFX extracts a bit field, but it sign extends it before putting it in a destination register. For example:

```
      LDR    R0,=0x5678ABCD
      SBFX.W R1, R0, #4, #8
```

This will give R1 = 0xFFFFFFBC.

LDRD and STRD

The two instructions LDRD and STRD transfer two words of data from or into two registers. The syntax of the instructions is:

```
      LDRD.W <Rxf>, <Rxf2> ,[Rn, #+/-offset]{!} ; Pre-indexed
      LDRD.W <Rxf>, <Rxf2> ,[Rn], #+/-offset    ; Post-indexed
      STRD.W <Rxf>, <Rxf2> ,[Rn, #+/-offset]{!} ; Pre-indexed
      STRD.W <Rxf>, <Rxf2> ,[Rn], #+/-offset    ; Post-indexed
```

where <*Rxf*> is the first destination/source register, and <*Rxf2*> is the second destination/source register.

For example, the following code reads a 64-bit value located in memory address 0x1000 into R0 and R1:

```
   LDR    R2,=0x1000
   LDRD.W R0, R1, [R2] ; This will gives R0 = memory[0x1000],
                       ; R1 = memory[0x1004]
```

Similarly, we can use STRD to store a 64-bit value in memory. In the following example, pre-indexed addressing mode is used:

```
   LDR    R2,=0x1000 ; Base address
   STRD.W R0, R1, [R2, #0x20] ; This will gives memory[0x1000] = R0,
                              ; memory[0x1004] = R1
```

TBB and TBH

TBB (Table Branch Byte) and TBH (Table Branch Halfword) are for implementing branch tables. The TBB instruction uses a branch table of byte size offset, and TBH uses a branch table of half word offset. Since the bit 0 of a program counter is always zero, the value in the branch table is multiplied by two before it's added to PC. Furthermore, because the PC value is the current instruction address plus 4, the branch range for TBB is (2 × 255) + 4 = 514, and the branch range for TBH is (2 × 65535) + 4 = 131074. Both TBB and TBH support forward branch only.

TBB has this general syntax:

```
        TBB.W [Rn, Rm]
```

where *Rn* is the base memory offset and *Rm* is the branch table index. The branch table item for TBB is located at Rn + Rm. Assuming we used PC for Rn, we can see the operation as shown in Figure 4.5.

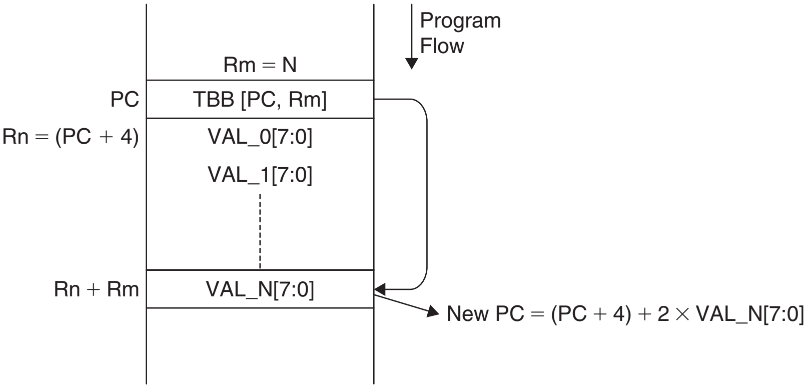


Figure 4.5 TBB Operation

For TBH instruction, the process is similar except the memory location of the branch table item is located at Rn + 2 × Rm and the maximum branch offset is higher. Again, we assume that Rn is set to PC, as shown in Figure 4.6.

If *Rn* in the table branch instruction is set to R15, the value used for *Rn* will be PC + 4 because of the pipeline in the processor. These two instructions are more likely to be used by a C compiler to generate code for switch (case) statements. Because the values in the branch table are relative to the current program counter, it is not easy to code the branch table content manually in assembler as the address offset value might not be able to be determined during assembly/compile stage, especially if the branch target is in a separate program code file. The coding syntax for calculating TBB/TBH branch table content could be dependent on the development tool. In ARM assembler (*armasm*), the TBB branch table can be created in the following way:

```
        TBB.W [pc, r0] ; when executing this instruction, PC equal
                       ; branchtable
branchtable
        DCB ((dest0 - branchtable)/2) ; Note that DCB is used because
                                      ; the value is 8-bit
        DCB ((dest1 - branchtable)/2)
        DCB ((dest2 - branchtable)/2)
        DCB ((dest3 - branchtable)/2)
dest0
        ... ; Execute if r0 = 0
dest1
        ... ; Execute if r0 = 1
```

```
dest2
      ... ; Execute if r0 = 2
dest3
      ... ; Execute if r0 = 3
```

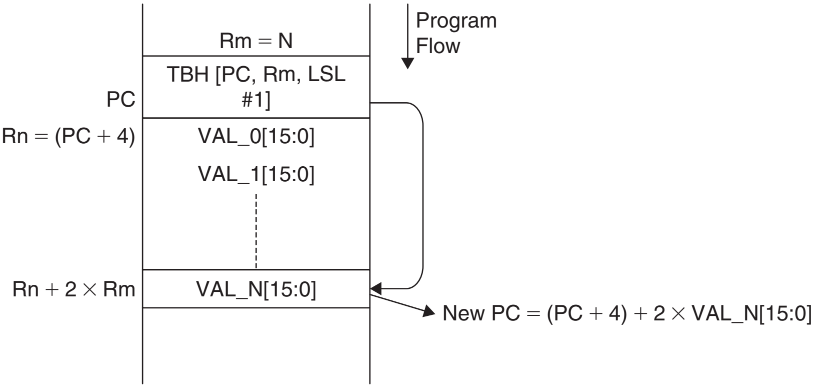


Figure 4.6 TBH Operation

When the TBB instruction is executed, the current PC value is at the address labeled as *branchtable* (because of the pipeline in the processor). Similarly, for TBH instructions, it can be used as:

```
      TBH.W [pc, r0, LSL #1]
branchtable
      DCI ((dest0 – branchtable)/2) ; Note that DCI is used because
                                    ; the value is 16-bit
      DCI ((dest1 – branchtable)/2)
      DCI ((dest2 – branchtable)/2)
      DCI ((dest3 – branchtable)/2)
dest0
      ... ; Execute if r0 = 0
dest1
      ... ; Execute if r0 = 1
dest2
      ... ; Execute if r0 = 2
dest3
      ... ; Execute if r0 = 3
```

CHAPTER 5

Memory Systems

In This Chapter:

- Memory System Features Overview
- Memory Maps
- Memory Access Attributes
- Default Memory Access Permissions
- Bit-Band Operations
- Unaligned Transfers
- Exclusive Accesses
- Endian Mode

Memory System Features Overview

The Cortex-M3 processor has a different memory architecture from that of traditional ARM processors. First, it has a predefined memory map that specifies which bus interface is to be used when a memory location is accessed. This feature also allows the processor design to optimize the access behavior when different devices are accessed.

Another feature of the memory system in the Cortex-M3 is the bit-band support. This provides atomic operations to bit data in memory or peripherals. The bit-band operations are supported only in special memory regions. This topic is covered in more detail later in this chapter.

The Cortex-M3 memory system also supports unaligned transfers and exclusive accesses. These features are part of the v7-M architecture. Finally, the Cortex-M3 supports both little endian and big endian memory configuration.

Memory Maps

The Cortex-M3 processor has a fixed memory map. This makes it easier to port software from one Cortex-M3 product to another. For example, components described in previous sections like NVIC and MPU have the same memory locations in all Cortex-M3 products. Nevertheless, the memory map definition allows great flexibility so that manufacturers can differentiate their Cortex-M3-based product from others.

Some of the memory locations are allocated for private peripherals such as debugging components. They are located in the private peripheral memory region. These debugging components include:

- Fetch Patch and BreakPoint Unit (FPB)
- Data WatchPoint and Trace Unit (DWT)
- Instrumentation Trace Macrocell (ITM)
- Embedded Trace Macrocell (ETM)
- Trace Port Interface Unit (TPIU)
- ROM Table

The details of these components are discussed in later chapters on debugging features.

The Cortex-M3 processor has a total of 4 GB of address space. Program code can be located in the Code region, the SRAM region, or the External RAM region. However, it is best to put the program code in the Code region because, with this arrangement, the instruction fetches and data accesses are carried out simultaneously on two separate bus interfaces.

The SRAM memory range is for connecting internal SRAM. Access to this region is carried out via the system interface bus. In this region, a 32 MB range is defined as a bit-band alias. Within the 32 MB bit-band alias memory range, each word address represents a single bit in the 1 Mb bit-band region. A data write access to this memory range will be converted to an atomic READ-MODIFY-WRITE operation to the bit-band region so as to allow a program to set or clear individual data bits in the memory. The bit-band operation applies only to data accesses, not instruction fetches. By putting Boolean information (single bits) in bit-band region, we can pack multiple Boolean data in a single word while still allowing it to be accessible individually via bit-band alias, thus saving memory space without the need for handling READ-MODIFY-WRITE in software. More details on bit-band alias can be found later in this chapter.

Another 0.5 GB block of address range is allocated to on-chip peripherals. Similar to the SRAM region, this region supports bit-band alias and is accessed via the system bus interface. However, instruction execution in this region is not allowed. The bit-band support in the peripheral region makes it easy to access or change control and status bits of peripherals, making it easier to program peripheral control.

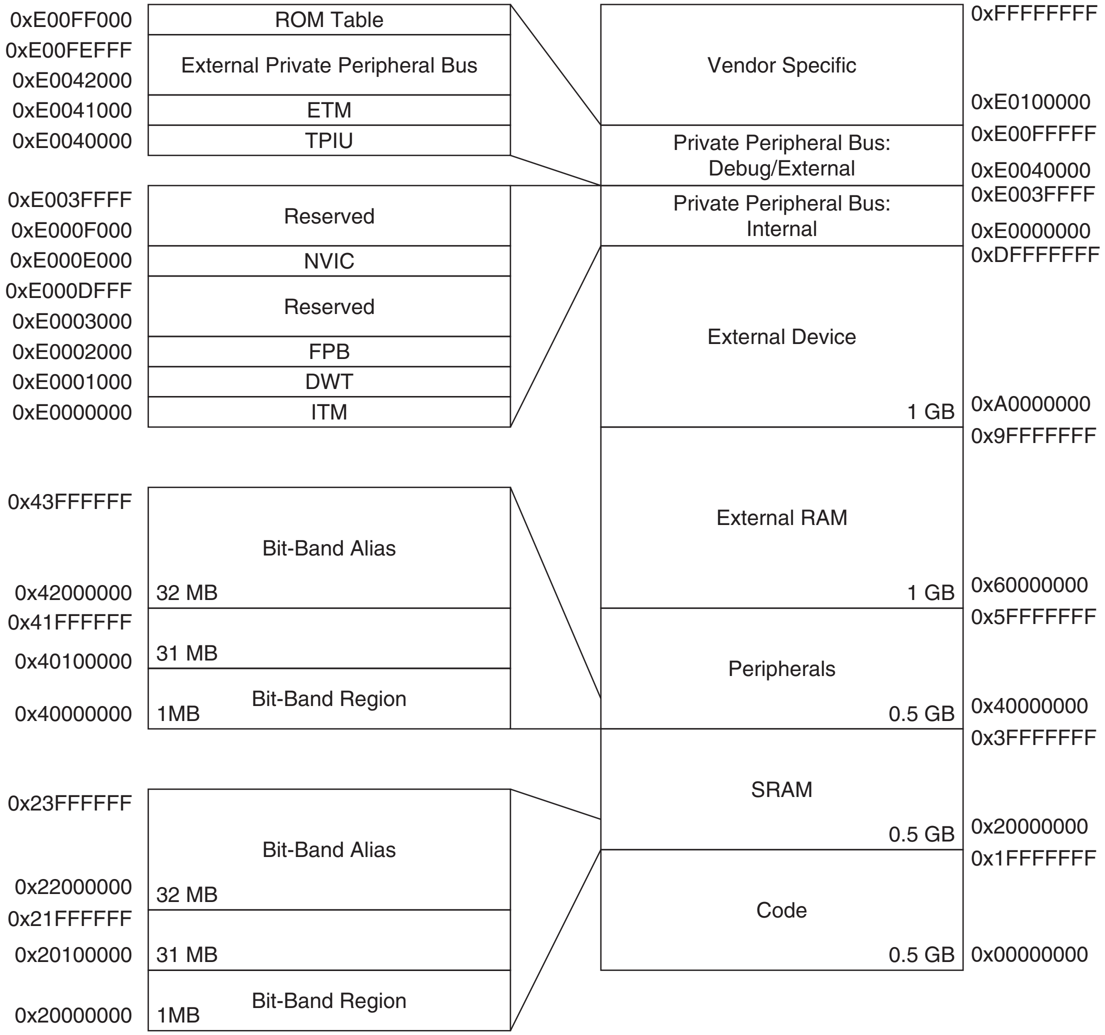


Figure 5.1 A Cortex-M3 Predefined Memory Map

Two slots of 1 GB memory space are allocated for external RAM and external devices. The difference between the two is that program execution in the external device region is not allowed, and there are some differences with the caching behaviors.

The last 0.5 GB of memory is for the system-level components, internal peripheral buses, external peripheral bus, and vendor-specific system peripherals. There are two segments of the private peripheral bus:

- AHB private peripheral bus, for Cortex-M3 internal AHB peripherals only: This includes NVIC, FPB, DWT, and ITM.

- APB private peripheral bus, for Cortex-M3 internal APB devices as well as external peripherals (external to the Cortex-M3 processor): The Cortex-M3 allows chip vendors to add additional on-chip APB peripherals on this APB private peripheral bus via an APB interface.

The NVIC is located in a memory region called the System Control Space (SCS). Besides providing interrupt control features, this region also provides the control registers for SYSTICK, MPU, and code debugging control.

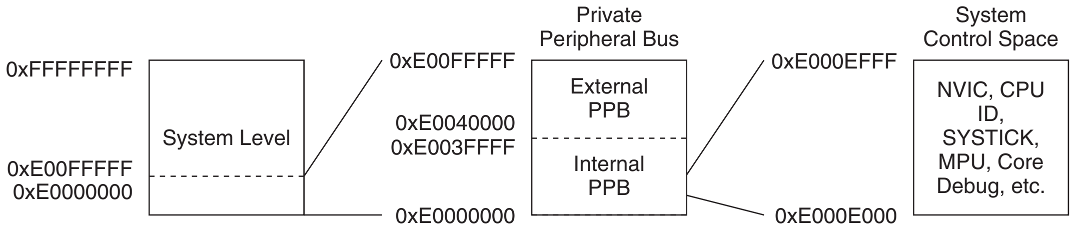


Figure 5.2 The System Control Space

The remaining unused vendor-specific memory range can be accessed via the system bus interface. However, instruction execution in this region is not allowed.

The Cortex-M3 processor also comes with an optional MPU. Chip manufacturers can decide whether to include the MPU in their products.

What we have shown in the memory map is merely a template; individual semiconductor vendors will provide detailed memory maps including the actual location and size of ROM, RAM, and peripheral memory locations.

Memory Access Attributes

The memory map shows what is included in each memory region. Aside from decoding which memory block or device is accessed, the memory map also defines the memory attributes of the access. The memory attributes you can find in the Cortex-M3 processor include these:

- Bufferable
- Cacheable
- Executable
- Sharable

The default memory attribute settings can be overridden if MPU is present and the region is programmed differently from the default. In spite of the fact that the Cortex-M3 processor does not have a cache memory or cache controller, an external cache can be added, and the cache attributes might also affect the operation of memory controllers for on-chip memory and off-chip memory, depending on the memory controllers used by the chip manufacturers:

- Code memory region (0x00000000–0x1FFFFFFF): This region is executable, and the cache attribute is WT (Write Through). You can put data memory in this region as well. If data operations are carried out for this region, they will take place via the data bus interface. Write is buffered for this region.
- SRAM memory region (0x20000000–0x3FFFFFFF): This region is intended for on-chip RAM. Write is buffered, and the cache attribute is WB-WA (Write Back, Write Allocated). This region is executable, so you can copy program code here and execute it.
- Peripheral region (0x40000000–0x5FFFFFFF): This region is intended for peripherals. The accesses are noncacheable. You cannot execute instruction code in this region (Execute Never, or XN in ARM documentation, such as the Cortex-M3 TRM).
- External RAM region (0x60000000–0x7FFFFFFF): This region is intended for either on-chip or off-chip memory. The accesses are cacheable (WB-WA), and you can execute code in this region.
- External RAM region (0x80000000–0x9FFFFFFF): This region is intended for either on-chip or off-chip memory. The accesses are cacheable (WT), and you can execute code in this region.
- External devices (0xA0000000–0xBFFFFFFF): This region is intended for external devices and/or shared memory that needs ordering/nonbuffered accesses. It is also a nonexecutable region.
- External devices (0xC0000000–0xDFFFFFFF): This region is intended for external devices and/or shared memory that needs ordering/nonbuffered accesses. It is also a nonexecutable region.
- System region (0xE0000000–0xFFFFFFFF): This region is for private peripherals and vendor-specific devices. It is nonexecutable. For the private peripheral bus memory range, the accesses are strongly ordered (noncacheable, nonbufferable). For the vendor-specific memory region, the accesses are bufferable and noncacheable.

Note that from Revision 1 of the Cortex-M3, the memory attribute for the Code region is hardwired to cacheable and nonbufferable. This cannot be overridden by MPU configuration.

Default Memory Access Permissions

The Cortex-M3 memory map has a default configuration for memory access permissions. This prevents user programs from accessing system control memory spaces such as the NVIC. The default memory access permission is used when either:

- No MPU is present
- MPU is present but disabled

If MPU is present and enabled, the access permission in the MPU setup will determine whether user accesses are allowed.

The default memory access permissions are shown in Table 5.1.

Table 5.1 Default Memory Access Permissions

Memory Region	Address	Access in User Program
Vendor specific	0xE0100000–0xFFFFFFFF	Full access
ROM Table	0xE00FF000–0xE00FFFFF	Blocked; user access results in bus fault
External PPB	0xE0042000–0xE00FEFFF	Blocked; user access results in bus fault
ETM	0xE0041000–0xE0041FFF	Blocked; user access results in bus fault
TPIU	0xE0040000–0xE0040FFF	Blocked; user access results in bus fault
Internal PPB	0xE000F000–0xE003FFFF	Blocked; user access results in bus fault
NVIC	0xE000E000–0xE000EFFF	Blocked; user access results in bus fault, except Software Trigger Interrupt Register that can be programmed to allow user accesses
FPB	0xE0002000–0xE0003FFF	Blocked; user access results in bus fault
DWT	0xE0001000–0xE0001FFF	Blocked; user access results in bus fault
ITM	0xE0000000–0xE0000FFF	Read allowed; write ignored except for stimulus ports with user access enabled
External Device	0xA0000000–0xDFFFFFFF	Full access
External RAM	0x60000000–0x9FFFFFFF	Full access
Peripheral	0x40000000–0x5FFFFFFF	Full access
SRAM	0x20000000–0x3FFFFFFF	Full access
Code	0x00000000–0x1FFFFFFF	Full access

When a user access is blocked, the fault exception takes place immediately.

Bit-Band Operations

Bit-band operation support allows a single load/store operation to access (read/write) to a single data bit. In the Cortex-M3, this is supported in two predefined memory regions called *bit-band regions*. One of them is located in the first 1 MB of the SRAM region, and the other is located in the first 1 MB of the peripheral region. These two memory regions can be accessed like normal memory, but they can also be accessed via a separate memory region called the *bit-band alias*. When the bit-band alias address is used, each individual bit can be accessed separately in the least significant bit (LSB) of each word-aligned address.

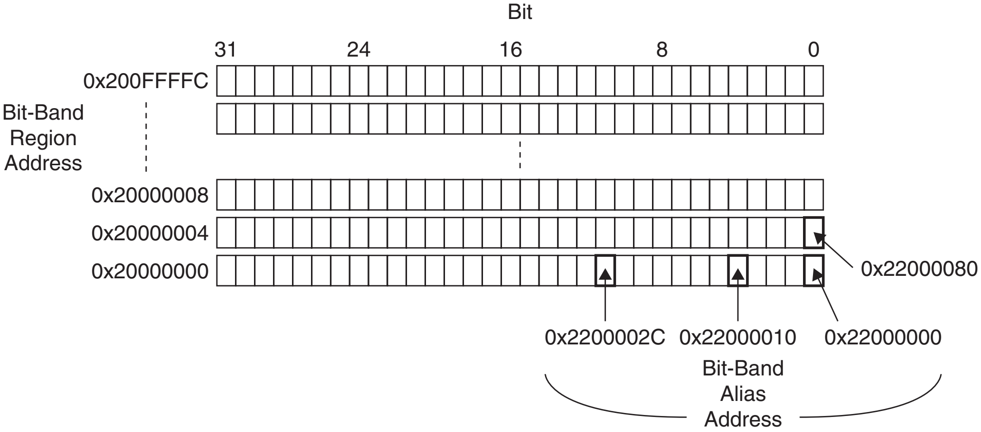


Figure 5.3 Bit Accesses to Bit-Band Region Via the Bit-Band Alias

For example, to set bit 2 in word data in address 0x20000000, instead of using three instructions to read the data, set the bit, and then write back the result, this task can be carried out by a single instruction (see Figure 5.4).

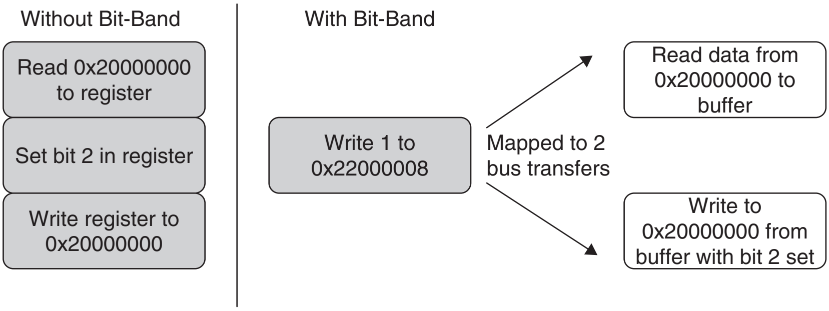

Without Bit-Band
Read 0x20000000 to register
Set bit 2 in register
Write register to 0x20000000
With Bit-Band
Write 1 to 0x22000008
Mapped to 2 bus transfers
Read data from 0x20000000 to buffer
Write to 0x20000000 from buffer with bit 2 set

Figure 5.4 Write to Bit-Band Alias

The assembler sequence for these two cases could be like the one shown in Figure 5.5.

Similarly, bit-band support can simplify application code if we need to read a bit in a memory location. For example, if we need to determine bit 2 of address 0x20000000, we use the steps outlined in Figure 5.6.

Without Bit-Band

```
LDR   R0,=0x20000000 ; Setup address
LDR   R1, [R0]       ; Read
ORR.W R1, #0x4       ; Modify bit
STR   R1, [R0] ; Write back result
```

With Bit-Band

```
LDR   R0,=0x22000008 ; Setup address
MOV   R1, #1         ; Setup data
STR   R1, [R0]       ; Write
```

Figure 5.5 Example Assembler Sequence to Write a Bit With and Without Bit-Band

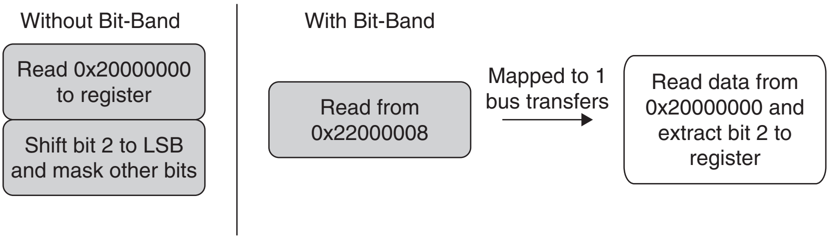

Without Bit-Band
Read 0x20000000 to register
Shift bit 2 to LSB and mask other bits
With Bit-Band
Read from 0x22000008
Mapped to 1 bus transfers
Read data from 0x20000000 and extract bit 2 to register

Figure 5.6 Read from the Bit-Band Alias

The assembler sequence for these two cases could be like the one shown in Figure 5.7.

Without Bit-Band

```
LDR    R0,=0x20000000 ; Setup address
LDR    R1, [R0]       ; Read
UBFX.W R1,R1, #2, #1  ; Extract bit[2]
```

With Bit-Band

```
LDR   R0,=0x22000008 ; Setup address
LDR   R1, [R0]       ; Read
```

Figure 5.7 Read from the Bit-Band Alias

Bit-band operation is not a new idea; in fact, a similar feature has existed for more than 30 years on 8-bit microcontrollers such as the 8051. Although the Cortex-M3 does not have special instructions for bit operation, special memory regions are defined so that data accesses to these regions are automatically converted into bit-band operations.

Note that the Cortex-M3 uses the following terms for the bit-band memory addresses:

- Bit-band region: This is a memory address region that supports bit-band operation.
- Bit-band alias: Access to the bit-band alias will cause an access (a bit-band operation) to the bit-band region. (*Note:* A memory remapping is performed.)

Within the bit-band region, each word is represented by an LSB of 32 words in the bit-band alias address range. What actually happens is that, when the bit-band alias address is accessed, the address is remapped into a bit-band address. For read operations, the word is read and the chosen bit location is shifted to the LSB of the read return data. For write operations, the written bit data is shifted to the required bit position, and a READ-MODIFY-WRITE is performed.

There are two regions of memory for bit-band operations:

- 0x20000000–0x200FFFFF (SRAM, 1 Mb)
- 0x40000000–0x400FFFFF (Peripherals, 1 Mb)

For the SRAM memory region, the remapping of the bit-band alias is shown in Table 5.2.

Table 5.2 Remapping of Bit-Band Addresses in SRAM Region

Bit-Band Region	Aliased Equivalent
0x20000000 bit[0]	0x22000000 bit[0]
0x20000000 bit[1]	0x22000004 bit[0]
0x20000000 bit[2]	0x22000008 bit[0]
…	…
0x20000000 bit[31]	0x2200007C bit[0]
0x20000004 bit[0]	0x22000080 bit[0]
…	…
0x20000004 bit[31]	0x220000FC bit[0]
…	…
0x200FFFFC bit[31]	0x23FFFFFC bit[0]

Similarly, the bit-band region of the peripheral memory region can be accessed via bit-band aliased addresses, as shown in Table 5.3.

Table 5.3 Remapping of Bit-Band Addresses in Peripheral Memory Region

Bit-Band Region	Aliased Equivalent
0x40000000 bit[0]	0x42000000 bit[0]
0x40000000 bit[1]	0x42000004 bit[0]
0x40000000 bit[2]	0x42000008 bit[0]
…	…
0x40000000 bit[31]	0x4200007C bit[0]
0x40000004 bit[0]	0x42000080 bit[0]
…	…
0x40000004 bit[31]	0x420000FC bit[0]
…	…
0x400FFFFC bit[31]	0x43FFFFFC bit[0]

Here's a simple example:

1. Set address 0x20000000 to a value of 0x3355AACC.
2. Read address 0x22000008. This read access is remapped into read access to 0x20000000. The return value is 1 (bit[2] of 0x3355AACC).
3. Write 0x0 to 0x22000008. This write access is remapped into a READ-MODIFY-WRITE to 0x20000000. The value 0x3355AACC is read from memory, bit 2 is cleared, and a result of 0x3355AAC8 is written back to address 0x20000000.
4. Now read 0x20000000. That gives you a return value of 0x3355AAC8 (bit[2] cleared).

When you access bit-band alias addresses, only the LSB (bit[0]) in the data is used. In addition, accesses to the bit-band alias region should not be unaligned. If an unaligned access is carried out to bit-band alias address range, the result is unpredictable.

Advantages of Bit-Band Operations

So, what are the uses of bit-band operations? We can use them to, for example, implement serial data transfers in general-purpose input/output (GPIO) ports to serial devices. The application code can be implemented easily because access to serial data and clock signals can be separated.

Bit-Band vs Bit-Bang

In the Cortex-M3, we use the term *bit-band* to indicate that the feature is a special memory band (region) that provides bit accesses. *Bit-band* commonly refers to driving I/O pins under software control to provide serial communication functions. The bit-band feature in the Cortex-M3 can be used for bit-banging implementations, but the definitions of these two terms are different.

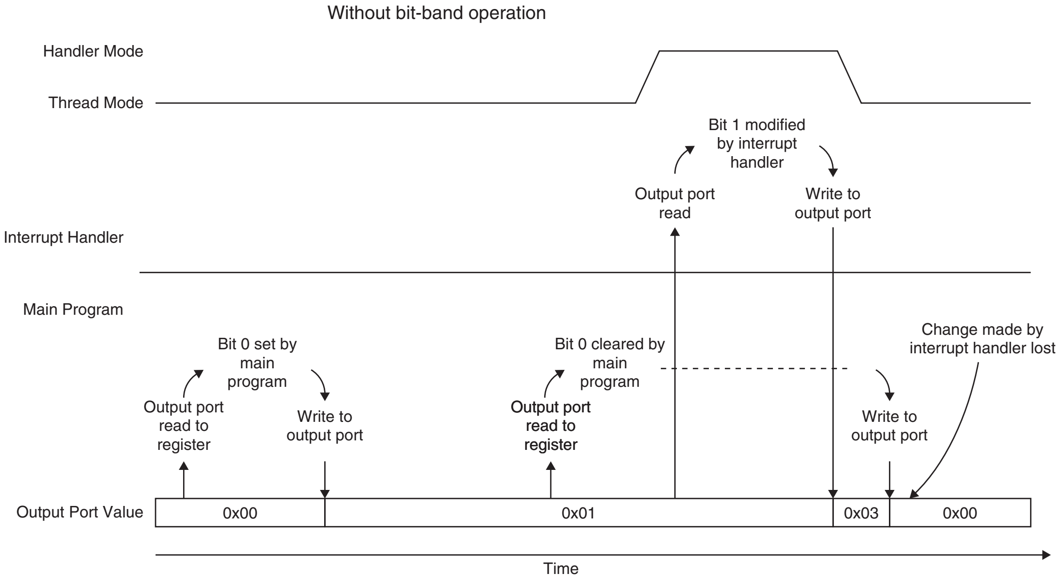
Bit-band operation can also be used to simplify branch decisions. For example, if a branch should be carried out based on 1 single bit in a status register in a peripheral, instead of:

- Reading the whole register
- Masking the unwanted bits
- Comparing and branching

you can simplify the operations to:

- Reading the status bit via the bit-band alias (get 0 or 1)
- Comparing and branching

Besides providing faster bit operations with fewer instructions, the bit-band feature in the Cortex-M3 is also essential for situations in which resources are being shared by more than one process. One of the most important advantages or properties of bit-band operation is that it is *atomic*. In other words, the READ-MODIFY-WRITE sequence cannot be interrupted by other bus activities. Without this behavior in, for example, using a software READ-MODIFY-WRITE sequence, the following problem can occur: Consider a simple output port with bit 0 used by a main program and bit 1 used by an interrupt handler. A software based READ-MODIFY-WRITE operation can cause data conflicts, as shown in Figure 5.8.



Without bit-band operation
Handler Mode
Thread Mode
Bit 1 modified by interrupt handler
Output port read
Write to output port
Interrupt Handler
Main Program
Bit 0 set by main program
Output port read to register
Write to output port
Bit 0 cleared by main program
Output port read to register
Change made by interrupt handler lost
Write to output port
Output Port Value
0x00
0x01
0x03
0x00
Time

Figure 5.8 Data Are Lost When an Exception Handler Modifies a Shared Memory Location

With the Cortex-M3 bit-band feature, this kind of race condition can be avoided because the READ-MODIFY-WRITE is carried out at the hardware level and is atomic (the two transfers cannot be pulled apart) and interrupts cannot take place between them (see Figure 5.9).

Similar issues can be found in multitasking systems. For example, if bit 0 of the output port is used by Process A and bit 1 is used by Process B, a data conflict can occur in software-based READ-MODIFY-WRITE (see Figure 5.10).

Again, the bit-band feature can ensure that bit accesses from each task are separated so that no data conflicts occur (see Figure 5.11).

Besides I/O functions, the bit-band feature can be used for storing and handling Boolean data in the SRAM region. For example, multiple Boolean variables can be packed into one single

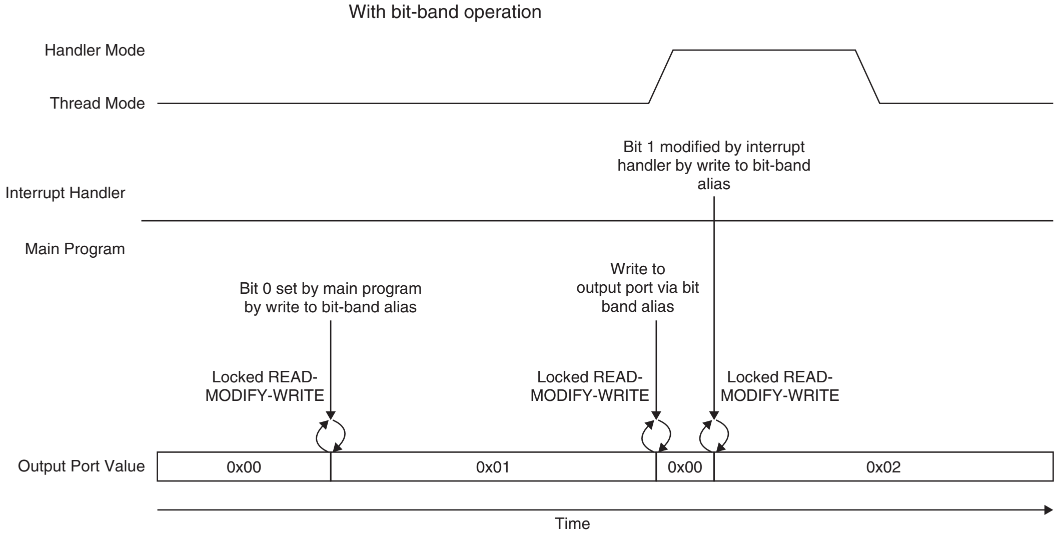


Figure 5.9 Data Loss Prevention with Locked Transfers Using the Bit-Band Feature

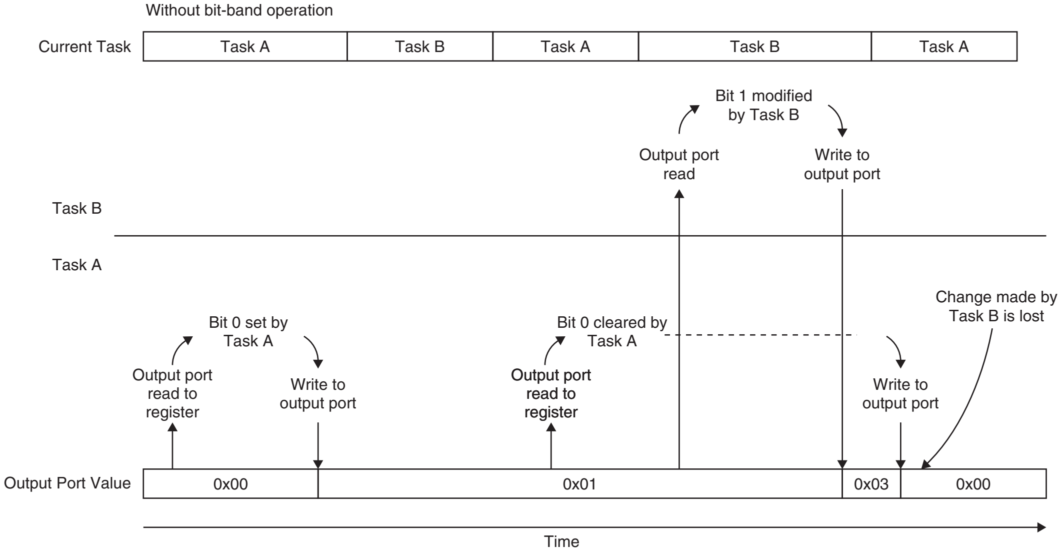


Figure 5.10 Data Are Lost When a Different Task Modifies a Shared Memory Location

memory location to save memory space, whereas the access to each bit is still completely separated when the access is carried out via the bit-band alias address range.

For SoC designers designing a bit-band-capable device, the device's memory address should be located within the bit-band memory, and the lock (HMASTLOCK) signal from the AHB interface must be checked to make sure that writable register contents will not be changed except by the bus when a locked transfer is carried out.

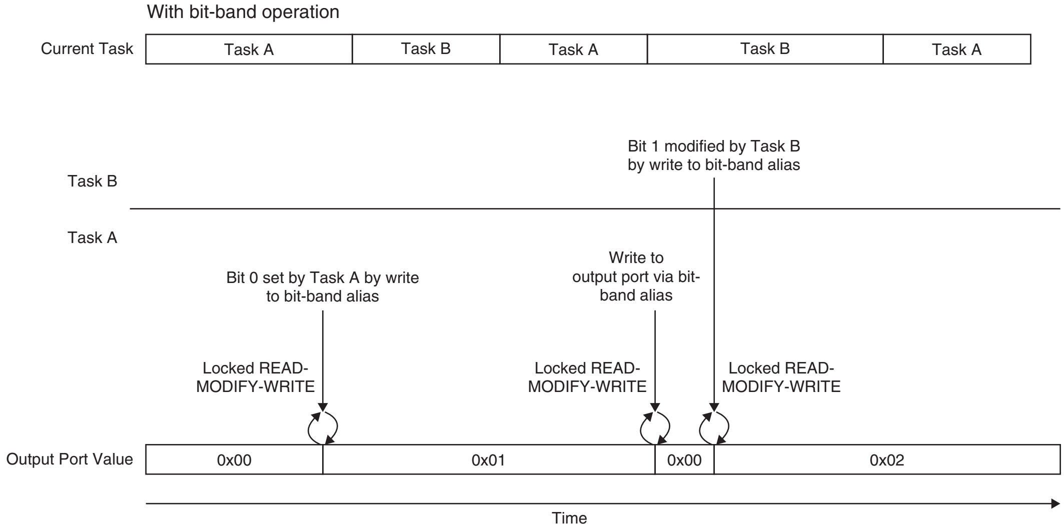


Figure 5.11 Data Loss Prevention with Locked Transfers Using the Bit-Band Feature

Bit-Band Operation of Different Data Sizes

Bit-band operation is not limited to word transfers. It can be carried out as byte transfers or half word transfers as well. For example, when a byte access instruction (LDRB/STRB) is used to access a bit-band alias address range, the accesses generated to the bit-band region will be in byte size. The same applies to half word transfers (LDRH/STRH). When you use nonword transfers to bit-band alias addresses, the address value should still be word aligned.

Bit-Band Operations in C Programs

There is no native support of bit-band operation in C compilers. For example, C compilers do not understand that the same memory can be accessed using two different addresses, and they do not know that accesses to the bit-band alias will only access the LSB of the memory location. To use the bit-band feature in C, the most simple solution is to separately declare the address and the bit-band alias of a memory location. For example:

```
#define   DEVICE_REG0        ((volatile unsigned long *)   (0x40000000))
#define   DEVICE_REG0_BIT0   ((volatile unsigned long *)   (0x42000000))
#define   DEVICE_REG0_BIT1   ((volatile unsigned long *)   (0x42000004))
   ...
   *DEVICE_REG0 = 0xAB;  // Accessing the hardware register by normal
                         // address
   ...
   *DEVICE_REG0 = *DEVICE_REG0 | 0x2;  // Setting bit 1 without using
                                       // bitband feature
   ...
   *DEVICE_REG0_BIT1 = 0x1;  // Setting bit 1 using bitband feature
                             // via the bit band alias address
```

It is also possible to develop C macros to make accessing the bit-band alias easier. For example, we could set up one macro to convert the bit-band address and the bit number into the bit-band alias address and set up another macro to access the memory location by taking the address value as a pointer:

```
// Convert bit band address and bit number into bit band alias address
#define BITBAND(addr,bitnum) ((addr & 0xF0000000)+0x2000000+((addr &
  0xFFFFF)<<5)+(bitnum <<2))

// Convert the address as a pointer
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
```

Based on the previous example, we rewrite the code as follows:

```
#define DEVICE_REG0 0x40000000
#define BITBAND(addr,bitnum) ((addr & 0xF0000000)+0x02000000+((addr &
  0xFFFFF)<<5)+(bitnum<<2))
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))

       ...
       MEM_ADDR(DEVICE_REG0) = 0xAB; // Accessing the hardware
                                     // register by normal address
       ...
       // Setting bit 1 without using bitband feature
       MEM_ADDR(DEVICE_REG0) = MEM_ADDR(DEVICE_REG0) | 0x2;

       ...
       // Setting bit 1 with using bitband feature
       MEM_ADDR(BITBAND(DEVICE_REG0,1)) = 0x1;
```

Note that when the bit-band feature is used, the variables being accessed should be declared as *volatile*. The C compilers do not know that the same data could be accessed in two different addresses, so the volatile property is used to ensure that each time a variable is accessed, the memory location is accessed instead of a local copy of the data inside the processor.

You can find further examples of bit-band accesses with C macros using ARM RealView Compiler Tools 3.0 in the *ARM Application Note 179* (Ref 7).

Unaligned Transfers

The Cortex-M3 supports unaligned transfers on single accesses. Data memory accesses can be defined as aligned or unaligned. Traditionally, ARM processors (such as the ARM7/ARM9/

ARM10) allow only aligned transfers. That means that in accessing memory, a word transfer must have address bit[1] and bit[0] equal to 0, and a half word transfer must have address bit[0] equal to 0. For example, word data can be located at 0x1000 or 0x1004, but it cannot be located in 0x1001, 0x1002, or 0x1003. For half word data, the address can be 0x1000 or 0x1002, but it cannot be 0x1001.

So, what does an unaligned transfer look like? Figures 5.12–5.16 show some examples. Assuming that the memory infrastructure is 32-bit (4 bytes) wide, an unaligned transfer can be any word size read/write such that the address is not a multiple of 4, as shown in Figures 5.12–5.14, or when the transfer is in half word size, and the address is not a multiple of 2, as in Figures 5.15 and 5.16.

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4				[31:24]
Address N	[23:16]	[15:8]	[7:0]	

Figure 5.12 Unaligned Transfer Example 1

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4			[31:24]	[23:16]
Address N	[15:8]	[7:0]		

Figure 5.13 Unaligned Transfer Example 2

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4		[31:24]	[23:16]	[15:8]
Address N	[7:0]			

Figure 5.14 Unaligned Transfer Example 3

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4				
Address N		[15:8]	[7:0]	

Figure 5.15 Unaligned Transfer Example 4

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4				[15:8]
Address N	[7:0]			

Figure 5.16 Unaligned Transfer Example 5

All the byte-size transfers are aligned on the Cortex-M3 because the minimum address step is 1 byte.

In the Cortex-M3, unaligned transfers are supported in normal memory accesses (such as LDR, LDRH, STR, and STRH instructions). There are a number of limitations:

- Unaligned transfers are not supported in Load/Store multiple instructions.
- Stack operations (PUSH/POP) must be aligned.
- Exclusive accesses (such as LDREX or STREX) must be aligned; otherwise a fault exception (usage fault) will be triggered.
- Unaligned transfers are not supported in bit-band operations. Results will be unpredictable if you attempt to do so.

When unaligned transfers are used, they are actually converted into multiple aligned transfers by the processor's bus interface unit. This conversion is transparent, so application programmers do not have to worry about it. However, when an unaligned transfer takes place, it is broken into separate transfers and as a result it takes more clock cycles for a single data access and might not be good for situations in which high performance is required. To get the best performance, it's worth making sure that data are aligned properly.

It is also possible to set up the NVIC so that an exception is triggered when an unaligned transfer takes place. This is done by setting the UNALIGN_TRP (Unaligned Trap) bit in the Configuration Control Register in the NVIC (0xE000ED14). In this way, the Cortex-M3 generates usage fault exceptions when unaligned transfers take place. This is useful during software development to test whether an application produces unaligned transfers.

Exclusive Accesses

You might have noticed that the Cortex-M3 has no SWP instruction (swap), which was used for semaphore operations in traditional ARM processors like ARM7TDMI. This is now being replaced by exclusive access operations. Exclusive accesses were first supported in architecture v6 (for example, in the ARM1136).

Semaphores are commonly used for allocating shared resources to applications. When a resource is being used by one process, it is locked to that process and cannot serve another process until the lock is released. To set up a semaphore, a memory location is defined as the lock flag to indicate whether a shared resource is locked by a process. When a process or application want to use the resource, it needs to check whether the resource has been locked first. If it is not being used, it can set the lock flag to indicate that the resource is now locked. In traditional ARM processors, the access to the lock flag is carried out by the SWP instruction. It allows the lock flag read and write to be atomic, preventing the resource from being locked by two processes at the same time.

In newer ARM processors, the read/write access can be carried out on separated buses. In such situations, the SWP instructions can no longer be used to make the memory access atomic, since the read and write in a locked transfer sequence must be on the same bus. Therefore, the locked transfers are replaced by exclusive accesses. The concept of exclusive access operation is quite simple but different from SWP; it allows the possibility that the memory location for a semaphore could be accessed by another bus master or another process running on the same processor (see Figure 5.17).

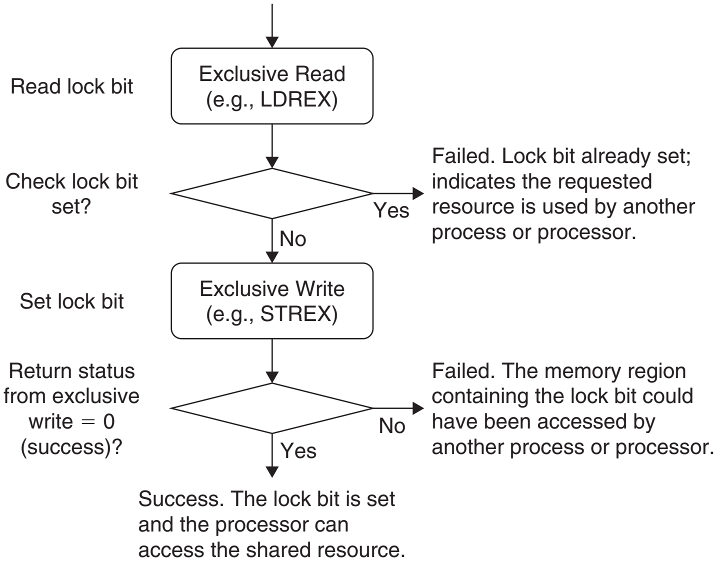

Read lock bit
Exclusive Read (e.g., LDREX)
Check lock bit set?
Yes
Failed. Lock bit already set; indicates the requested resource is used by another process or processor.
No
Set lock bit
Exclusive Write (e.g., STREX)
Return status from exclusive write = 0 (success)?
No
Failed. The memory region containing the lock bit could have been accessed by another process or processor.
Yes
Success. The lock bit is set and the processor can access the shared resource.

Figure 5.17 Using Exclusive Access in Semaphores

If the memory device has been accessed by another bus master between the exclusive read and the exclusive write, the exclusive access monitor will flag an exclusive failed through the bus system when the processor attempts the exclusive write. This will cause the return status of the exclusive write to be 1. To monitor exclusive accesses in a system with multiple bus masters (such as multiple processor designs), additional monitor hardware and bus sideband signals are needed. In the Cortex-M3 processor, the required sideband signals are available for a D-Code bus (called EXREQD and EXRESPD) and a system bus (EXREQS and EXRESPS). The instruction bus (I-Code) does not have exclusive access sideband signals.

Exclusive access instructions in the Cortex-M3 include LDREX (word), LDREXB (byte), LDREXH (half word), STREX (word), STREXB (byte), and STREXH (half word). A simple example of the syntax is:

```
LDREX    <Rxf>, [Rn, #offset]
STREX    <Rd>, <Rxf> ,[Rn, #offset]
```

where <*Rd*> is the return status of the exclusive write (0 = success, 1 = failure). Example code for exclusive accesses can be found in Chapter 10.

When exclusive accesses are used, the internal write buffers in the Cortex-M3 bus interface will be bypassed, even when the MPU defines the region as bufferable. This ensures that semaphore information on the physical memory is always up to date and coherent between bus masters. SoC designers using Cortex-M3 on multiprocessor systems should ensure that the memory system enforces data coherency when exclusive transfers occur.

Endian Mode

The Cortex-M3 supports both little endian and big endian modes. However, the supported memory type also depends on the design of the rest of the microcontroller (bus connections, memory controllers, peripherals, and so on). Make sure that you check your microcontroller datasheets in detail before developing your software. In most cases, Cortex-M3-based microcontrollers will be little endian.

The definition of big endian in the Cortex-M3 is different from the ARM7's. In the ARM7TDMI, the big endian scheme is called *word-invariant big endian*, whereas in the Cortex-M3, the big endian scheme is called *byte-invariant big endian*. (Byte-invariant big endian is supported on ARM architecture v6 and v7.) See Table 5.4.

Table 5.4 The Cortex-M3 (Byte-Invariant Big Endian): Memory View

Address, Size	Bits 31–24	Bits 23–16	Bits 15–8	Bits 7–0
0x1000, word	Data[7:0]	Data[15:8]	Data[23:16]	Data[31:24]
0x1000, half word	Data[7:0]	Data[15:8]	–	–
0x1002, half word	–	–	Data[7:0]	Data[15:8]
0x1000, byte	Data[7:0]	–	–	–
0x1001, byte	–	Data[7:0]	–	–
0x1002, byte	–	–	Data[7:0]	–
0x1003, byte	–	–	–	Data[7:0]

Note that the data transfer on the AHB bus in BE-8 mode uses the same data byte lanes as in little endian. However, the data byte inside the half word or word data is reversely ordered (see Table 5.5).

This behavior is different from ARM7TDMI, which has a different bus lane arrangement when operating in big endian mode. As mentioned, the big endian mode used in ARM7 is called *word-invariant big endian,* and the bus lane usage in the bus is as shown in Table 5.6.

In the Cortex-M3, the endian mode is set when the processor exits reset. The endian mode cannot be changed afterward. (There is no dynamic endian switching, and the SETEND

Table 5.5 Cortex-M3 (Byte-Invariant Big Endian): Data on AHB Bus

Address, Size	Bits 31–24	Bits 23–16	Bits 15–8	Bits 7–0
0x1000, word	Data[7:0]	Data[15:8]	Data[23:16]	Data[31:24]
0x1000, half word	—	—	Data[7:0]	Data[15:8]
0x1002, half word	Data[7:0]	Data[15:8]	—	—
0x1000, byte	—	—	—	Data[7:0]
0x1001, byte	—	—	Data[7:0]	—
0x1002, byte	—	Data[7:0]	—	—
0x1003, byte	Data[7:0]	—	—	—

Table 5.6 ARM7 (Word-Invariant Big Endian): Data on AHB Bus (Different from Memory View)

Address, Size	Bits 31–24	Bits 23–16	Bits 15–8	Bits 7–0
0x1000, word	Data[7:0]	Data[15:8]	Data[23:26]	Data[31:24]
0x1000, half word	Data[7:0]	Data[15:8]	—	—
0x1002, half word	—	—	Data[7:0]	Data[15:8]
0x1000, byte	Data[7:0]	—	—	—
0x1001, byte	—	Data[7:0]	—	—
0x1002, byte	—	—	Data[7:0]	—
0x1003, byte	—	—	—	Data[7:0]

instruction is not supported.) Instruction fetches are always in little endian, as are data accesses in the configuration control memory space (such as the NVIC, FPB, and the like) and the external PPB memory range (memory range from 0xE0000000 to 0xE00FFFFF is always little endian).

In case your SoC does not support big endian but one or some of the peripherals you are using contain big endian data, you can easily convert the data between little endian and big endian using some of the new instructions in the Cortex-M3. For example, REV and REVH are very useful for this kind of conversion.

This page intentionally left blank

CHAPTER 6

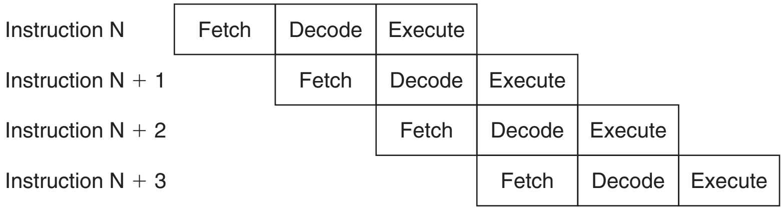
Cortex-M3 Implementation Overview

In This Chapter:

- The Pipeline
- A Detailed Block Diagram
- Bus Interfaces on the Cortex-M3
- Other Interfaces on the Cortex-M3
- The External Private Peripheral Bus
- Typical Connections
- Reset Signals

The Pipeline

The Cortex-M3 processor has a three-stage pipeline. The pipeline stages are instruction fetch, instruction decode, and instruction execution (see Figure 6.1).



Instruction N	Fetch	Decode	Execute			
Instruction N + 1		Fetch	Decode	Execute		
Instruction N + 2			Fetch	Decode	Execute	
Instruction N + 3				Fetch	Decode	Execute

Figure 6.1 The Three-Stage Pipeline in the Cortex-M3

Some people might argue that there are four stages because of the pipeline behavior in the bus interface when it accesses memory, but this stage is outside the processor, so the processor itself still has only three stages.

When running programs with mostly 16-bit instructions, you will find that the processor might not fetch instructions in every cycle. This is because the processor fetches up to two instructions (32-bit) in one go, so after one instruction is fetched, the next one is already inside the processor. In this case, the processor bus interface may try to fetch the instruction after the next or, if the buffer is full, the bus interface could be idle. Some of the instructions take multiple cycles to execute; in this case, the pipeline will be stalled.

In executing a branch instruction, the pipeline will be flushed. The processor will have to fetch instructions from the branch destination to fill up the pipeline again. However, the Cortex-M3 processor supports a number of instructions in v7-M architecture, so some of the short-distance branches can be avoided by replacing them with conditional execution codes.[1]

Due to the pipeline nature of the processor and to ensure that the program is compatible with Thumb codes, when the program counter is read during instruction execution, the read value will be the address of the instruction plus 4. This offset is constant, independent of the combination of 16-bit Thumb instructions and 32-bit Thumb-2 instructions. This ensures consistency between Thumb and Thumb-2.

Inside the instruction pre-fetch unit of the processor core, there is also an instruction buffer. This buffer allows additional instructions to be queued before they are needed. This buffer prevents the pipeline being stalled when the instruction sequence contains 32-bit Thumb-2 instructions that are not word aligned. However, this buffer does not add an extra stage to the pipeline, so it does not increase the branch penalty.

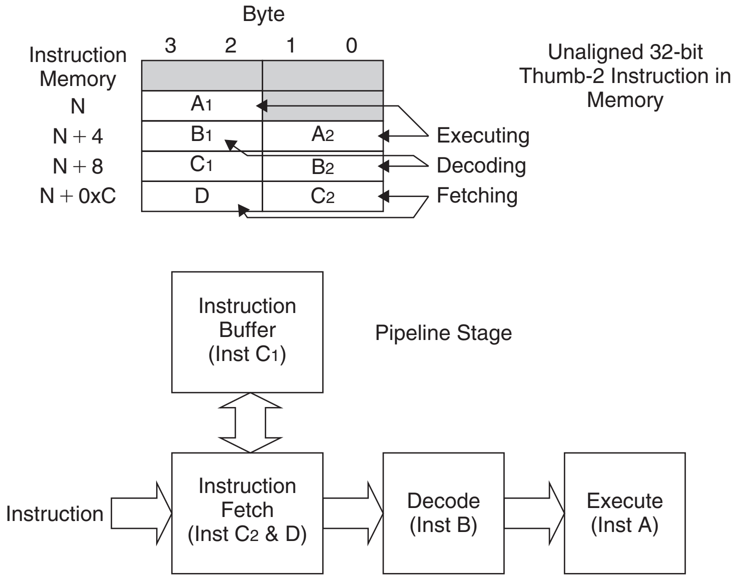


Figure 6.2 Use of a Buffer in Instruction Fetch Unit to Improve 32-Bit Instruction Handling

[1] For more information, refer to the "IF-THEN Instructions" section of Chapter 4.

A Detailed Block Diagram

The Cortex-M3 processor contains not only the processor core but also a number of components for system management, as well as debugging support components.

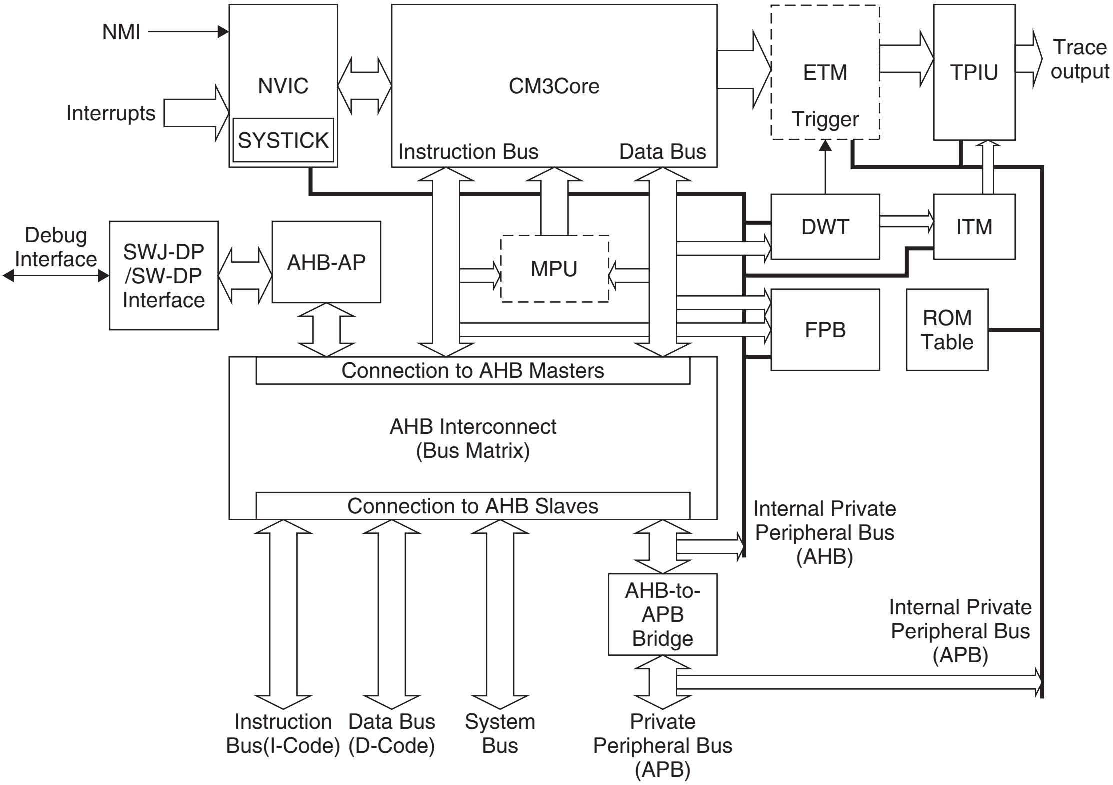


Figure 6.3 The Cortex-M3 Processor System Block Diagram

Note that the MPU and ETM blocks are optional blocks that can be included in the microcontroller system at the time of implementation.

A number of new components are shown in this diagram (see Table 6.1).

The Cortex-M3 processor is released as a processor subsystem. The CPU core itself is closely coupled to the interrupt controller (NVIC) and various debug logic blocks:

- CM3Core: The Cortex-M3 core contains the registers, ALU, data path, and bus interface.
- Nested Vectored Interrupt Controller: The NVIC is a built-in interrupt controller. The number of interrupts is customized by chip manufacturers. The NVIC is closely

Table 6.1 Block Diagram Acronyms and Definitions

Name	Description
CM3Core	Central processing core of the Cortex-M3 processor
NVIC	Nested Vectored Interrupt Controller
SYSTICK Timer	A simple timer that can be used by the operating system
MPU	Memory Protection Unit (optional)
CM3BusMatrix	Internal AHB interconnection
AHB to APB	Bus bridge to convert AHB to APB
SW-DP/SWJ-DP interface	Serial Wire/Serial Wire JTAG debug port (DP) interface; debug interface connection implemented using either Serial Wire Protocol or traditional JTAG Protocol (for SWJ-DP)
AHB-AP	AHB Access Port; converts commands from Serial Wire/SWJ interface into AHB transfers
ETM	Embedded Trace Macrocell; a module to handle instruction trace for debug (optional)
DWT	Data Watchpoint and Trace unit; a module to handle the data watchpoint function for debug
ITM	Instrumentation Trace Macrocell
TPIU	Trace Port Interface Unit; an interface block to send debug data to external trace capture hardware
FPB	Flash Patch and Breakpoint unit
ROM Table	A small lookup table that stores configuration information

coupled to the CPU core and contains a number of system control registers. It supports nested interrupt handling, which means that with the Cortex-M3, nested interrupt handling is very simple. It also comes with a vectored interrupt feature so that when an interrupt occurs, it can enter the corresponding interrupt handler routine directly, without using a shared handler to determine which interrupt has occurred.

- SYSTICK Timer: The System Tick (SYSTICK) Timer is a basic countdown timer that can be used to generate interrupts at regular time intervals, even when the system is in sleep mode. It makes OS porting between Cortex-M3 devices much easier because there is no need to change the OS's system timer code. The SYSTICK Timer is implemented as part of the NVIC.
- Memory Protection Unit: The MPU block is optional. This means that some versions of the Cortex-M3 might have the MPU and some might not. If it is included, the MPU can be used to protect memory contents by, for example, making memory regions read-only or preventing user applications from accessing privileged applications data.

- BusMatrix: A BusMatrix is used as the heart of the Cortex-M3 internal bus system. It is an AHB interconnection network, allowing transfer to take place on different buses simultaneously unless both bus masters are trying to access the same memory region. The BusMatrix also provides additional data transfer management, including a write buffer as well as bit-oriented operations (bit-band).
- AHB to APB: An AHB-to-APB bus bridge is used to connect a number of APB devices such as debugging components to the private peripheral bus in the Cortex-M3 processor. In addition, the Cortex-M3 allows chip manufacturers to attach additional APB devices to the external private peripheral bus using this APB bus.

The rest of the components in the block diagram are for debugging support and normally should not be used by application code:

- SW-DP/SWJ-DP: The Serial Wire Debug Port (SW-DP)/Serial Wire JTAG Debug Port (SWJ-DP) work together with the AHB Access Port (AHB-AP) so that external debuggers can generate AHB transfers to control debug activities. There is no JTAG scan chain inside the processor core of the Cortex-M3; most debugging functions are controlled by the NVIC registers through AHB accesses. SWJ-DP supports both the Serial Wire Protocol and the JTAG Protocol, whereas SW-DP can support only the Serial Wire Protocol.
- AHB-AP: The AHB Access Port provides access to the whole Cortex-M3 memory via a few registers. This block is controlled by the SW-DP/SWJ-DP through a generic debug interface called the Debug Access Port (DAP). To carry out debugging functions, the external debugging hardware needs to access the AHB-AP via the SW-DP/SWJ-DP to generate the required AHB transfers.
- Embedded Trace Macrocell: The ETM is an optional component for instruction trace, so some Cortex-M3 products might not have real-time instruction trace capability. Trace information is output to the trace port via TPIU. The ETM control registers are memory mapped, which can be controlled by the debugger via the DAP.
- Data Watchpoint and Trace: The DWT allows data watchpoints to be set up. When a data address or data value match is found, the match hit event can be used to generate watchpoint events to activate the debugger, generate data trace information, or activate the ETM.
- Instrumentation Trace Macrocell: The ITM can be used in several ways. Software can write to this module directly to output information to TPIU, or the DWT matching events can be used to generate data trace packets via ITM for output into a trace data stream.
- Trace Port Interface Unit: The TPIU is used to interface with external trace hardware such as trace port analyzers. Internal to the Cortex-M3, trace information is formatted

as Advanced Trace Bus (ATB) packets, and the TPIU reformats the data to allow data to be captured by external devices.

- FPB: The FPB is used to provide Flash Patch and Breakpoint functionalities. Flash Patch means that if an instruction access by the CPU matches a certain address, the address can be remapped to a different location so that a different value is fetched. Alternatively, the matched address can be used to trigger a breakpoint event. The Flash Patch feature is very useful for testing, such as adding diagnosis program code to a device that cannot be used in normal situations unless the FPB is used to change the program control.
- ROM table: A small ROM table is provided. This is simply a small lookup table to provide memory map information for various system devices and debugging components. Debugging systems use this table to locate the memory addresses of debugging components. In most cases, the memory map should be fixed to the standard memory location, as documented in the Cortex-M3 TRM, but because some of the debugging components are optional and additional components can be added, individual chip manufacturers might want to customize their chip's debugging features. In this case, the ROM table must be customized and used for debugging software to determine the correct memory map and hence detect the type of debugging components available.

Bus Interfaces on the Cortex-M3

Unless you are designing a SoC product using the Cortex-M3 processor, it is unlikely that you can directly access the bus interface signals described here. Normally the chip manufacturer will hook up all the bus signals to memory blocks and peripherals, and in a few cases, you might find that the chip manufacturer connected the bus to a bus bridge and allows external bus systems to be connected off-chip. The bus interfaces on the Cortex-M3 processor are based on AHB-Lite and APB protocols, which are documented in the AMBA Specification (Ref 4).

The I-Code Bus

The I-Code bus is a 32-bit bus based on the AHB-Lite bus protocol for instruction fetches in memory regions from 0x00000000 to 0x1FFFFFFF. Instruction fetches are performed in word size, even for Thumb instructions. Therefore, during execution, the CPU core could fetch up to two Thumb instructions at a time.

The D-Code Bus

The D-Code bus is a 32-bit bus based on the AHB-Lite bus protocol; it is used for data access in memory regions from 0x00000000 to 0x1FFFFFFF. Although the Cortex-M3 processor supports unaligned transfers, you won't get any unaligned transfer on this bus, because the bus interface on the processor core converts the unaligned transfers into aligned transfers for you. Therefore, devices (such as memory) that attach to this bus need only support AHB-Lite (AMBA 2.0) aligned transfers.

The System Bus

The system bus is a 32-bit bus based on the AHB-Lite bus protocol; it is used for instruction fetch and data access in memory regions from 0x20000000 to 0xDFFFFFFF and 0xE0100000 to 0xFFFFFFFF. As with the to the D-Code bus, all transfers are aligned.

The External Private Peripheral Bus

The External Private Peripheral bus (External PPB) is a 32-bit bus based on the APB bus protocol. This is intended for private peripheral accesses in memory regions 0xE0040000 to 0xE00FFFFF. However, since some part of this APB memory is already used for TPIU, ETM, and the ROM table, the memory region that can be used for attaching extra peripherals on this bus is only 0xE0042000 to 0xE00FF000. Transfers on this bus are word aligned.

The Debug Access Port Bus

The Debug Access Port (DAP) bus interface is a 32-bit bus based on an enhanced version of the APB specification. This is for attaching debug interface blocks such as SWJ-DP or SW-DP. Do not use this bus for other purposes. More information on this interface can be found in Chapter 15, "Debug Architecture," or in the ARM document *CoreSight Technology System Design Guide* (Ref 3).

Other Interfaces on the Cortex-M3

Apart from bus interfaces, the Cortex-M3 processor has a number of other interfaces for various purposes. These signals are unlikely to appear on the pins of the silicon chip, because they are mostly for connecting to various parts of the SoC or are unused. The details of the signals are contained in the *Cortex-M3 Technical Reference Manual (TRM)* (Ref 1). Table 6.2 contains a short summary of some of them.

The External Private Peripheral Bus

The Cortex-M3 processor has an External Private Peripheral bus (PPB) interface. The External PPB interface is based on the Advance Peripheral Bus (APB) protocol in AMBA specification 2.0. It is intended for system devices that should not be shared, such as debugging components. To support CoreSight devices, this interface contains an extra signal called PADDR31. This signal indicates the source of a transfer. If this signal is 0, it means that

Table 6.2 Miscellaneous Interface Signals

Signal Group	Function
Multiprocessor communication (TXEV, RXEV)	Simple task synchronization signals between multiple processors
Sleep signals (SLEEPING, SLEEPDEEP)	Sleep status for power management
Interrupt status signals (ETMINTNUM, ETMINTSTATE, CURRPRI)	Status of interrupt operation, for ETM operation and debug usage
Reset request (SYSRESETREQ)	Resets request output from NVIC
Lockup[2] and Halted status (LOCKUP, HALTED)	Indicate that the processor core has entered a lockup state (caused by error conditions within hard fault handler or NMI handler) or a halted state (for debug operations)
Endian input (ENDIAN)	Sets the endian of the Cortex-M3 when the core is reset
ETM interface	Connects to Embedded Trace Macrocell (ETM) for instruction trace
ITM's ATB interface	Advanced Trace Bus (ATB) is a bus protocol in ARM's CoreSight debug architecture for trace data transfer; here this interface provides trace data output from Cortex-M3's Instrumentation Trace Macrocell (ITM), which is connected to the Trace Port Interface Unit (TPIU)

the transfer is generated from software running on the Cortex-M3. If this signal is 1, it means that the transfer is generated by debugging hardware. Based on this signal, a peripheral can be designed so that only a debugger can use it, or when being used by software, only some of the features are allowed.

This bus is not intended for general use, as in peripherals. Although there is nothing to stop chip designers from designing and attaching general peripherals on this bus, users might find it a problem for programming later, due to privileged access-level management—for example, to program the device in the user state or to separate the devices from other memory regions when the MPU is used.

The External PPB does not support unaligned accesses. Since the data width of the bus is 32-bit and APB based, when you're designing peripherals for this memory region it is necessary to make sure that all register addresses in the peripheral are word aligned. In addition, when writing software accessing devices in this region, it is recommended that you make sure that all the accesses are in word size. The PPB accesses are always in little endian.

[2] More information on lockup is included in Chapter 12.

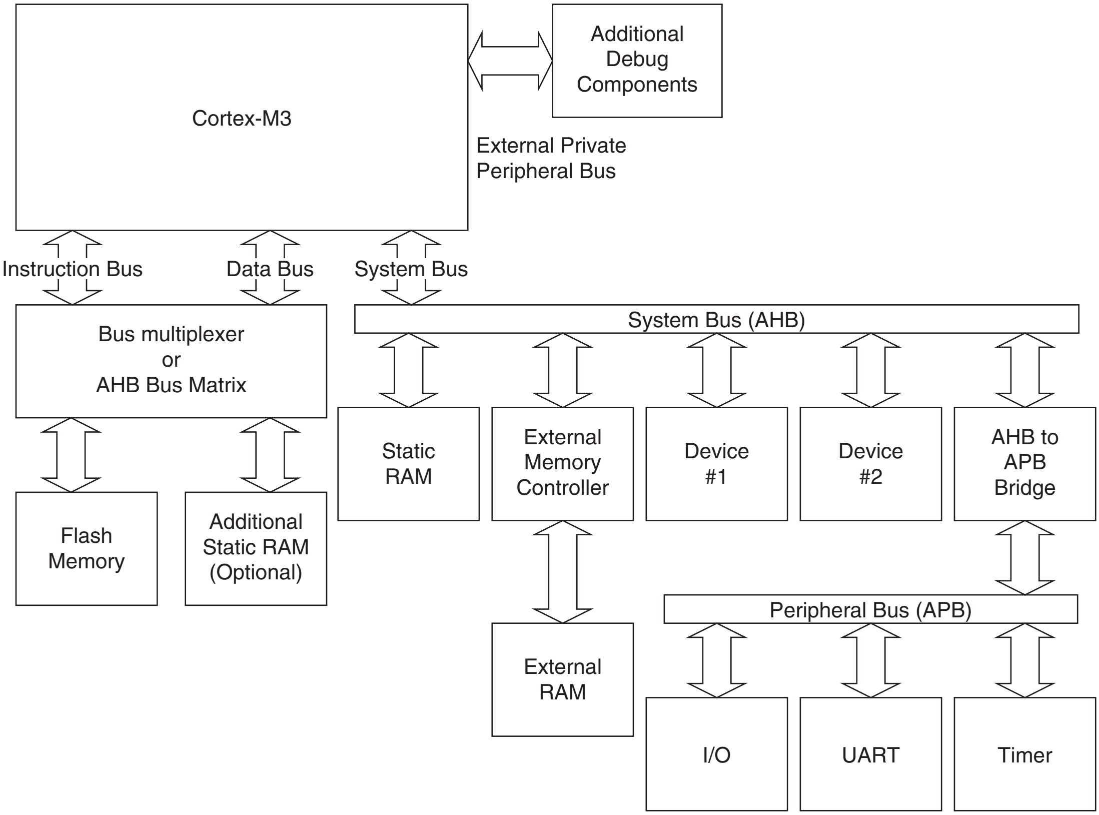


Figure 6.4 Example Cortex-M3 Bus Connections

Typical Connections

Because there are a number of bus interfaces on the Cortex-M3 processor, you might find it confusing to see how it will connect with other devices such as memory or peripherals. Figure 6.4 shows a simplified example.

Since the Code memory region can be accessed by the instruction bus (if it is an instruction fetch) and from the data bus (if it is a data access), an AHB bus switch called the Bus-Matrix[3] or an AHB bus multiplexer is needed. With the Bus-Matrix, the Flash memory and the additional SRAM memory (if implemented) can be accessed by either bus interface. The Bus-Matrix is available from ARM in the AMBA Development Kit (ADK).[4] When both data bus

[3] The Bus-Matrix required here is different from the internal BusMatrix inside the Cortex-M3. The Cortex-M3 internal bus-matrix is specially designed and cannot be used as a general AHB switch.

[4] ADK is a collection of AMBA components and example systems in VHDL/Verilog.

and instruction bus are trying to access the same memory device at the same time, the data bus access could be given higher priority for best performance.

Using the AHB Bus-Matrix, if the instruction bus and the data bus are accessing different memory devices at the same time (for example, an instruction fetch from fetch and a data bus reading data from the additional SRAM), the transfers can be carried out simultaneously. If a bus multiplexer is used, however, the transfers cannot take place at the same time, but the circuit size would be smaller. But common Cortex-M3 microcontroller designs use system bus for SRAM connection.

The main SRAM block should be connected via the system bus interface, using the SRAM memory address region. This allows data access to be carried out at the same time as instruction access. It also allows setting up of Boolean data types by using the bit-band feature.

Some microcontrollers might have an external memory interface. That requires an external memory controller because you cannot connect off chip memory devices directly to AHB. The external memory controller can be connected to the system bus of the Cortex-M3. Additional AHB devices can also be easily connected to the system bus without the need for a Bus-Matrix.

Simple peripherals can be connected to the Cortex-M3 via an AHB-to-APB bridge. This allows the use of the simpler bus protocol APB for peripherals.

The diagram shown in Figure 6.4 is just a very simple example; chip designers might choose different bus connection designs. For software/firmware development, you will only need to know the memory map.

Design blocks shown in the diagram, such as the Bus-Matrix, AHB-to-APB bus bridge, memory controller, I/O interface, timer, and UART, are all available from ARM and a number of IP providers. Since microcontrollers can have different providers for the peripherals, you need to access your microcontroller's datasheet for the correct programmer model when you're developing software for Cortex-M3 systems.

Reset Signals

The design of reset circuitry on the Cortex-M3 microcontroller or SoC is implementation specific. In the *Cortex-M3 Technical Reference Manual* (Ref 1), several reset signals are documented. However, the implemented Cortex-M3 chips will likely have only one or two reset signals, and the rest will be internally generated by reset generators designed by chip vendors. (Refer to the manufacturer datasheet for instructions on how to correctly reset their Cortex-M3-based microcontrollers.) At the Cortex-M3 processor level, you can find the reset signals shown in Table 6.3.

Table 6.3 Various Reset Types on Cortex-M3

Reset Signal	Description
Power on reset (PORESETn)	Reset that should be asserted when the device is powered up; resets both processor core and debugging system
System reset (SYSRESETn)	System reset; affects processor core, NVIC (except debug control registers), and MPU but not the debugging system
Test reset (nTRST)	Reset for debugging system

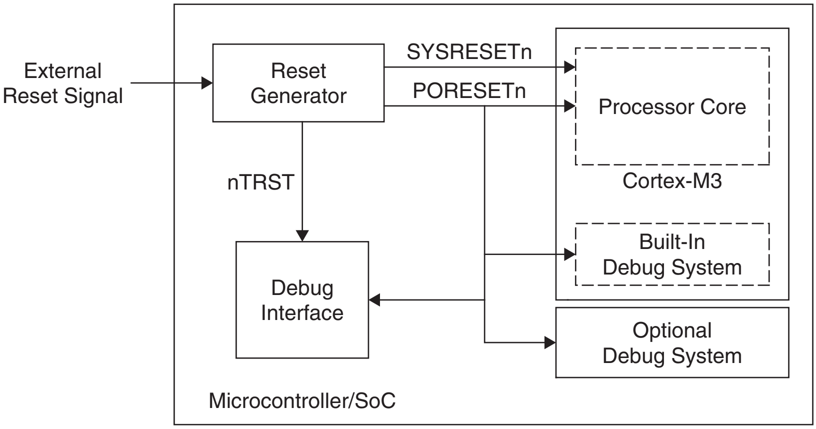


Figure 6.5 Reset Generation of Additional Internal Reset Signals in a Typical Cortex-M3 Microcontroller

This page intentionally left blank

CHAPTER 7

Exceptions

In This Chapter:

- Exception Types
- Definitions of Priority
- Vector Tables
- Interrupt Inputs and Pending Behavior
- Fault Exceptions
- SVC and PendSV

Exception Types

The Cortex-M3 provides a feature-packed exception architecture that supports a number of system exceptions and external interrupts. Exceptions are numbered 1 to 15 for system exceptions and 16 and above for external interrupt inputs. Most of the exceptions have programmable priority, and a few have fixed priority.

Cortex-M3 chips can have different numbers of external interrupt inputs (from 1 to 240) and different numbers of priority levels. This is because chip designers can configure the Cortex-M3 design source code for different needs.

Exception types 1 to 15 are system exceptions (there is no exception type 0), as outlined in Table 7.1. Exceptions of type 16 or above are external interrupt inputs (see Table 7.2).

The value of the current running exception is indicated by the special register IPSR or from the NVIC's Interrupt Control State Register (the VECTACTIVE field).

Note that here the interrupt number (e.g., Interrupt #0) refers to the interrupt inputs to the Cortex-M3 NVIC. In actual microcontroller products or SoCs, the external interrupt input pin

Table 7.1 List of System Exceptions

Exception Number	Exception Type	Priority	Description
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard Fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error; occurs when AHB interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	–
11	SVCall	Programmable	System Service call
12	Debug Monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	–
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

Table 7.2 List of External Interrupts

Exception Number	Exception Type	Priority
16	External Interrupt #0	Programmable
17	External Interrupt #1	Programmable
…	…	…
255	External Interrupt #239	Programmable

number might not match the interrupt input number on the NVIC. For example, some of the first few interrupt inputs might be assigned to internal peripherals, and external interrupt pins could be assigned to the next couple of interrupt inputs. Therefore, you need to check the chip manufacturer's datasheets to determine the numbering of the interrupts.

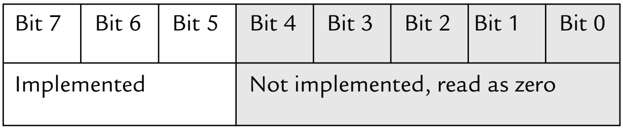
When an enabled exception occurs but cannot be carried out immediately (for instance, if a higher-priority interrupt service routine is running or if the interrupt mask register is set), it will be pended (except for some fault exceptions[1]). This means that a register (pending status) will hold the exception request until the exception can be carried out. This is different from traditional ARM processors. Previously, the devices that generate interrupts (such as IRQ/FIQ) must hold the request until they are served. Now, with the pending registers in the NVIC, an occurred interrupt will be handled even if the source requesting the interrupt de-asserts its request signal.

Definitions of Priority

In the Cortex-M3, whether and when an exception can be carried out can be affected by the priority of the exception. A higher-priority (smaller number in priority level) exception can preempt a lower-priority (larger number in priority level) exception; this is the nested exception/interrupt scenario. Some of the exceptions (reset, NMI, and hard fault) have fixed priority levels. They are negative numbers to indicate that they are higher priority than other exceptions. Other exceptions have programmable priority levels.

The Cortex-M3 supports three fixed highest-priority levels and up to 256 levels of programmable priority (a maximum of 128 levels of preemption). However, most Cortex-M3 chips have fewer supported levels—for example, 8, 16, 32, and so on. When a Cortex-M3 chip or SoC is being designed, designers can customize it to obtain the number of levels required. This reduction of levels is implemented by cutting out the LSB part of the priority configuration registers.

For example, if only 3 bits of priority level are implemented in the design, a priority-level configuration register will look like Figure 7.1.



Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented			Not implemented, read as zero				

Figure 7.1 A Priority Level Register with 3-bit Implemented

Since bit 4 to bit 0 are not implemented, they are always read as zero, and writes to these bits will be ignored. With this setup, we have possible priority levels of 0x00 (high priority), 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0, and 0xE0 (the lowest).

[1] There are a few exceptions for the exception-pending behavior. If a fault takes place and the corresponding fault handler cannot be executed immediately because a higher-priority handler is running, the hard fault handler (highest priority fault handler) might be executed instead. More details on this topic are covered later in this chapter, where we look at fault exceptions; full details can be found in the *Cortex-M3 Technical Reference Manual* and the *ARM v7-M Architecture Application Level Reference Manual*.

Similarly, if 4 bits of priority level are implemented in the design, a priority-level configuration register will look like Figure 7.2.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented				Not implemented, read as 0			

Figure 7.2 A Priority Level Register with 4-bit Implemented

If more bits are implemented, more priority levels will be available. However, more priority bits can also increase gate counts and hence power consumption. For the Cortex-M3, the minimum number of implemented priority register widths is 3 bits (eight levels).

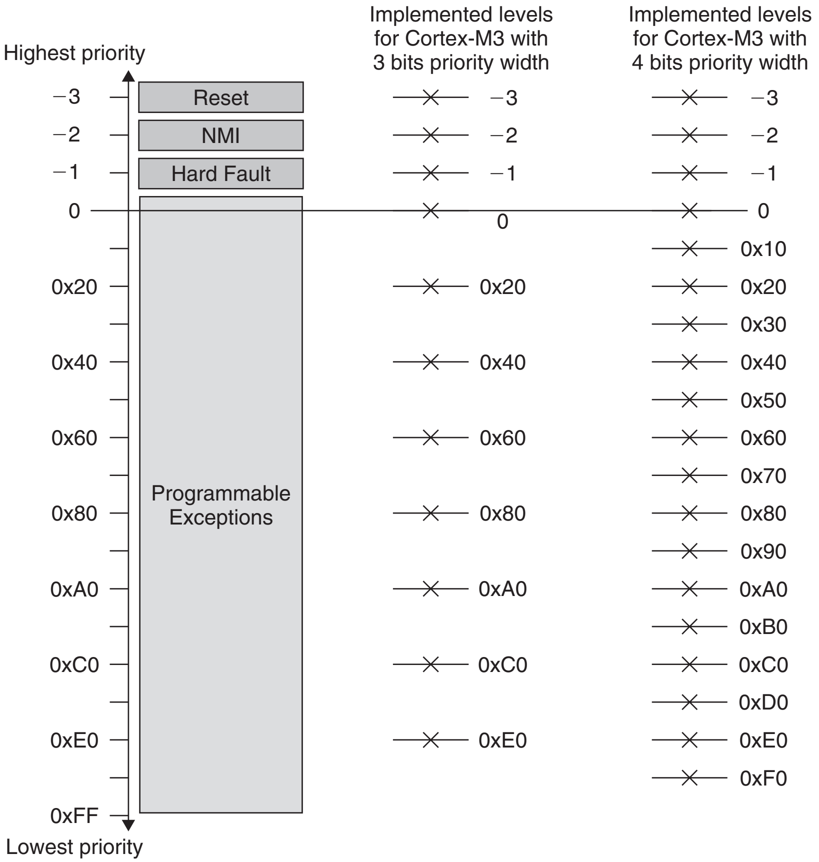


Figure 7.3 Available Priority Levels with 3-Bit or 4-Bit Priority Width

The reason for removing the LSB of the register instead of the MSB is to make it easier to port software from one Cortex-M3 device to another. In this way, a program written for devices with 4-bit priority configuration registers is likely to be able to run on devices with 3-bit priority configuration registers. If the MSB is removed instead of the LSB, you might get an inversion of priority arrangement when porting an application from one Cortex-M3 chip to another. For example, if an application uses priority level 0x05 for IRQ#0 and level 0x03 for IRQ#1, IRQ#1 should have higher priority. But when MSB bit 2 is removed, IRQ#0 will become level 0x01 and have a higher priority than IRQ#1.

Examples of available exception priority levels for devices with 3-bit, 5-bit, and 8-bit priority registers are shown in Table 7.3.

Table 7.3 Available Priority Levels for Devices with 3-bit, 5-bit, and 8-bit Priority Level Registers

Priority Level	Exception Type	Devices with 3-Bit Priority Configuration Registers	Devices with 5-Bit Priority Configuration Registers	Devices with 8-Bit Priority Configuration Registers
−3 (Highest)	Reset	−3	−3	−3
−2	NMI	−2	−2	−2
−1	Hard fault	−1	−1	−1
0, 1, … 0xFF	Exceptions with programmable priority level	0x00 0x20 … 0xE0	0x00 0x08 … 0xF8	0x00, 0x01 0x02, 0x03 … 0xFE, 0xFE

Some readers might wonder, if the priority level configuration registers are 8 bits wide, why there are only 128 preemption levels? This is because the 8-bit register is further divided into two parts: *preempt priority* and *subpriority*.

Using a configuration register in the NVIC called *Priority Group* (a part of the Application Interrupt and Reset Control register in the NVIC, see Table 7.5), the priority-level configuration registers for each exception with programmable priority levels is divided into two halves. The upper half (left bits) is the preempt priority, and the lower half (right bits) is the subpriority (see Table 7.4).

The *preempt priority level* defines whether an interrupt can take place when the processor is already running another interrupt handler. The *subpriority level* value is used only when two exceptions with same preempt priority level occur at the same time. In this case, the exception with higher subpriority (lower value) will be handled first.

As a result of the priority grouping, the maximum width of preempt priority is 7, so there can be 128 levels. When the priority group is set to 7, all exceptions with a programmable priority

Table 7.4 Definition of Preempt Priority Field and Subpriority Field in a Priority Level Register in Different Priority Group Settings

Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

Table 7.5 Application Interrupt and Reset Control Register (Address 0xE000ED0C)

Bits	Name	Type	Reset Value	Description
31:16	VECTKEY	R/W	–	Access key; 0x05FA must be written to this field to write to this register, otherwise the write will be ignored; the read-back value of the upper half word is 0xFA05
15	ENDIANNESS	R	–	Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian; this can only change after a reset
10:8	PRIGROUP	R/W	0	Priority group
2	SYSRESETREQ	W	–	Requests chip control logic to generate a reset
1	VECTCLRACTIVE	W	–	Clears all active state information for exceptions; typically used in debug or OS to allow system to recover from system error (Reset is safer)
0	VECTRESET	W	–	Resets the Cortex-M3 processor (except debug logic), but this will not reset circuits outside the processor

level will be in the same level, and no preemption between these exceptions will take place, except that hard fault, NMI, and reset, which have priority of −1, −2, and −3, respectively, can preempt these exceptions.

When deciding the effective preempt priority level and subpriority level, you must take these factors into account:

- Implemented priority-level configuration registers
- Priority group setting

For example, if the width of the configuration registers is 3 (bit 7 to bit 5 are available) and priority group is set to 5, you can have four levels of preempt priority levels (bit 7 to bit 6), and inside each preempt level there are two levels of subpriority (bit 5).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preempt priority		Sub priority					

Figure 7.4 Definition of Priority Fields in a 3-bit Priority Level Register with Priority Group Set to 5

With the setting as shown in Figure 7.4, the available priority levels are illustrated in Figure 7.5. For the same design, if the priority group is set to 0x1, there can be only eight preempt priority levels and no further subpriority levels inside each preempt level. (Bit[1:0] of preempt priority is always 0.) The definition of the priority level configuration registers is shown in Figure 7.6, and the available priority levels are illustrated in Figure 7.7.

If a Cortex-M3 device has implemented all 8 bits in the priority-level configuration registers, the maximum number of preemption levels it can have is only 128, using a priority group setting of 0. The priority fields definition is shown in Figure 7.8.

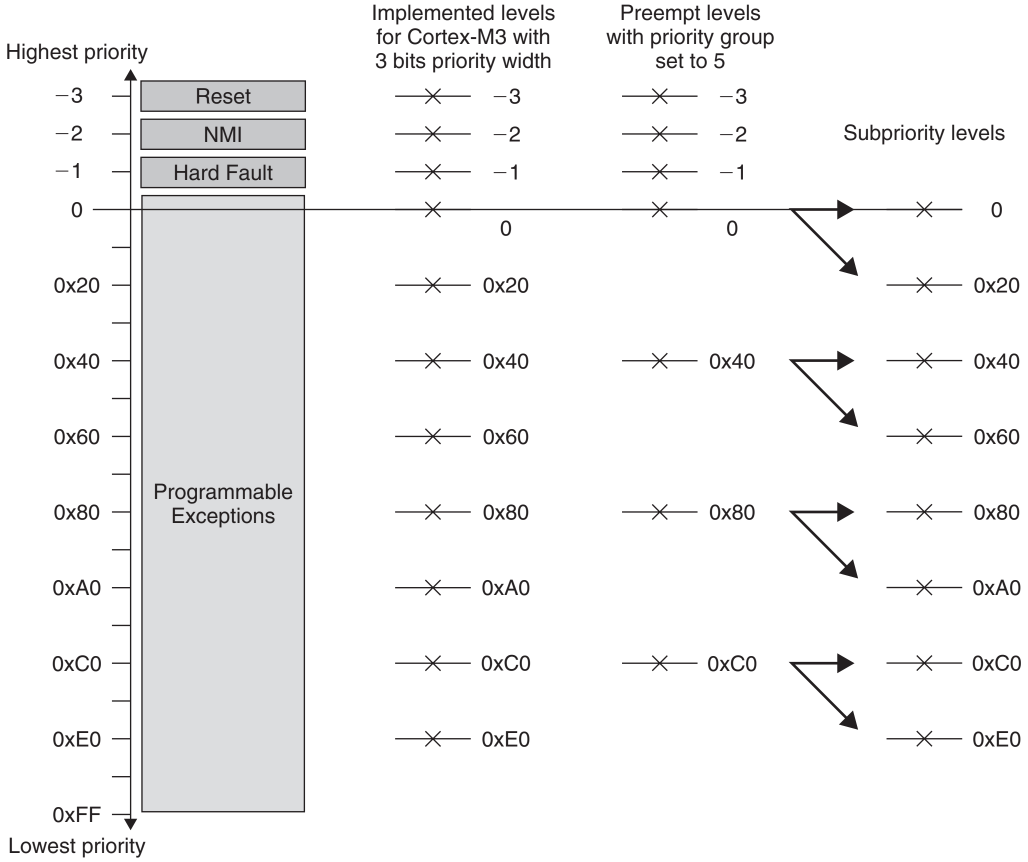


Figure 7.5 Available Priority Levels with 3-Bit Priority Width and Priority Group Set to 5

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preempt priority[5:3]			Preempt priority bit[2:0] (always 0)			Subpriority (always 0)	

Figure 7.6 Definition of Priority Fields in an 8-bit Priority Level Register with Priority Group Set to 1

When two interrupts are asserted at the same time with exactly the same preempt priority level as well as subpriority level, the interrupt with the smaller exception number has higher priority. (IRQ #0 has higher priority than IRQ #1.)

To avoid unexpected changes of priority levels for interrupts, be careful when writing to the Application Interrupt and Reset Control register (address 0xE000ED0C). In most cases, after the priority group is configured, there is no need to use this register except to generate a reset (see Table 7.5).

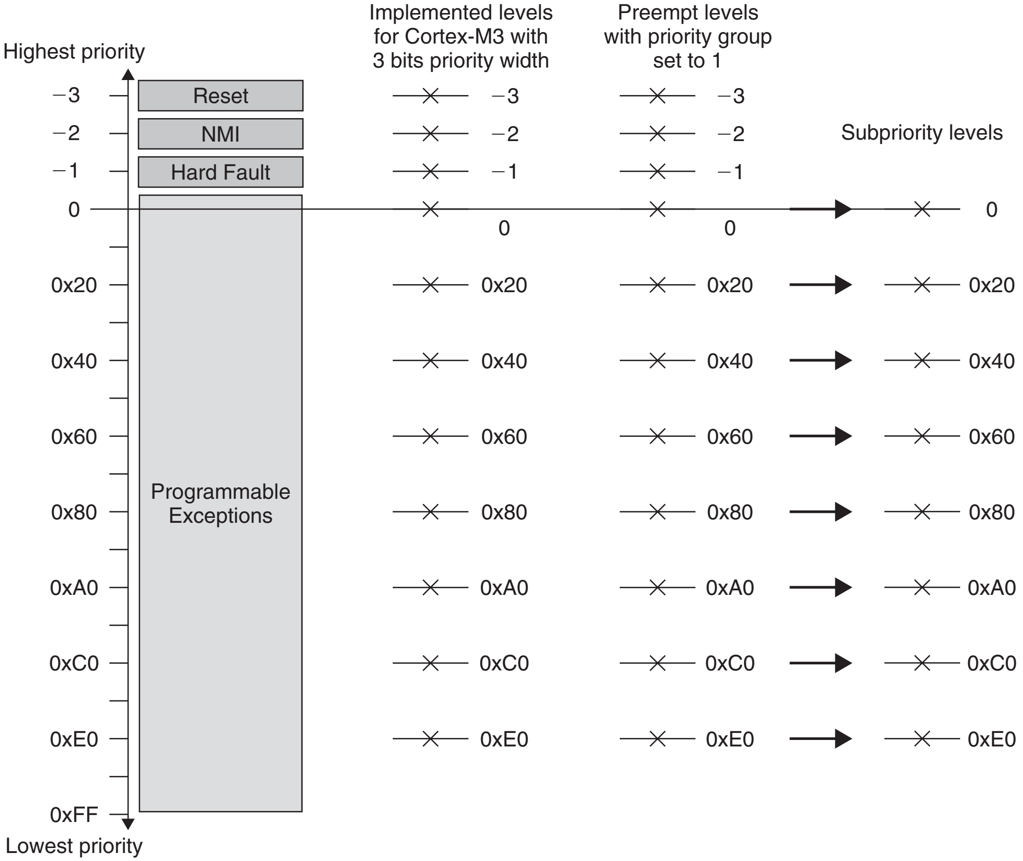


Figure 7.7 Available Priority Levels with 3-Bit Priority Width and Priority Group Set to 1

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preempt priority							Subpriority

Figure 7.8 Definition of Priority Fields in an 8-bit Priority Level Register with Priority Group Set to 0

Vector Tables

When an exception takes place and is being handled by the Cortex-M3, the processor will need to locate the starting address of the exception handler. This information is stored in the vector table. By default, the vector table starts at address zero, and the vector address is arranged according to the exception number times 4 (see Table 7.6).

Table 7.6 Exception Vector Table After Power Up

Address	Exception Number	Value (Word Size)
0x00000000	–	MSP initial value
0x00000004	1	Reset vector (program counter initial value)
0x00000008	2	NMI handler starting address
0x0000000C	3	Hard fault handler starting address
…	…	Other handler starting address

Since the address 0x0 should be boot code, usually it will either be Flash memory or ROM devices, and the value cannot be changed at run time. However, the vector table can be relocated to other memory locations in the Code or RAM region where the RAM is so that we can change the handlers during run time. This is done by setting a register in the NVIC called the *vector table offset register* (address 0xE000ED08). The address offset should be aligned to the vector table size, extended to the power of 2. For example, if there are 32 IRQ inputs, the total number of exceptions will be 32 + 16 (system exceptions) = 48. Extending it to the power of 2 makes it 64. Multiplying it by 4 makes it 256 (0x100). Therefore, the vector table offset can be programmed as 0x0, 0x100, 0x200, and so on. The vector table offset register contains the items shown in Table 7.7.

Table 7.7 Vector Table Offset Register (Address 0xE000ED08)

Bits	Name	Type	Reset Value	Description
29	TBLBASE	R/W	0	Table base in Code (0) or RAM (1)
28:7	TBLOFF	R/W	0	Table offset value from Code region or RAM region

In applications where you want to allow dynamic changing of exception handlers, in the beginning of the boot image you need to have these (at a minimum):

- Initial Main Stack Pointer value
- Reset vector

- NMI vector
- Hard fault vector

These are required because the NMI and hard fault can potentially occur during your boot process. Other exceptions cannot take place until they are enabled.

When the booting process is done, you can define a part of your SRAM as the new vector table and relocate the vector table to the new one, which is writable.

Interrupt Inputs and Pending Behavior

This section describes the behavior of IRQ inputs and pending behavior. It also applies to NMI input, except that an NMI will be executed immediately in most cases, unless the core is already executing an NMI handler, halted by a debugger, or locked up due to some serious system error.

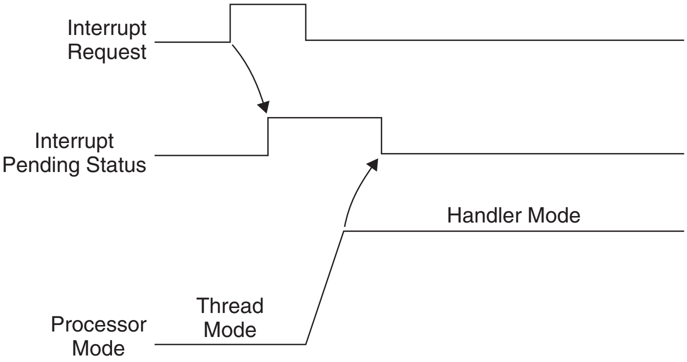


Figure 7.8 Interrupt Pending

When an interrupt input is asserted, it will be pended. Even if the interrupt source de-asserts the interrupt, the pended interrupt status will still cause the interrupt handler to be executed when the priority is allowed.

However, if the pending status is cleared before the processor starts responding to the pended interrupt (for example, because pending status register is cleared while PRIMASK/FAULTMASK is set to 1), the interrupt can be canceled (Figure 7.9). The pending status of the interrupt can be accessed in the NVIC and is writable, so you can clear a pending interrupt or use software to pend a new interrupt by setting the pending register.

When the processor starts to execute an interrupt, the interrupt becomes active and the pending bit will be cleared automatically (Figure 7.10). When an interrupt is active, you cannot start processing the same interrupt again until the interrupt service routine is terminated with an interrupt return (also called an *exception exit*, as discussed in Chapter 9). Then the

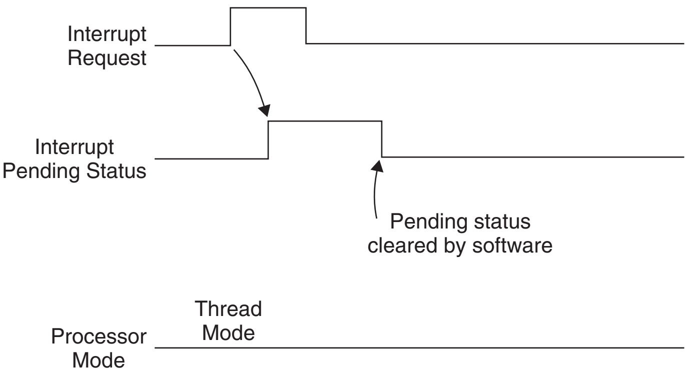


Figure 7.9 Interrupt Pending Cleared Before Processor Takes Action

active status is cleared and the interrupt can be processed again if the pending status is 1. It is possible to re-pend an interrupt before the end of the interrupt service routine.

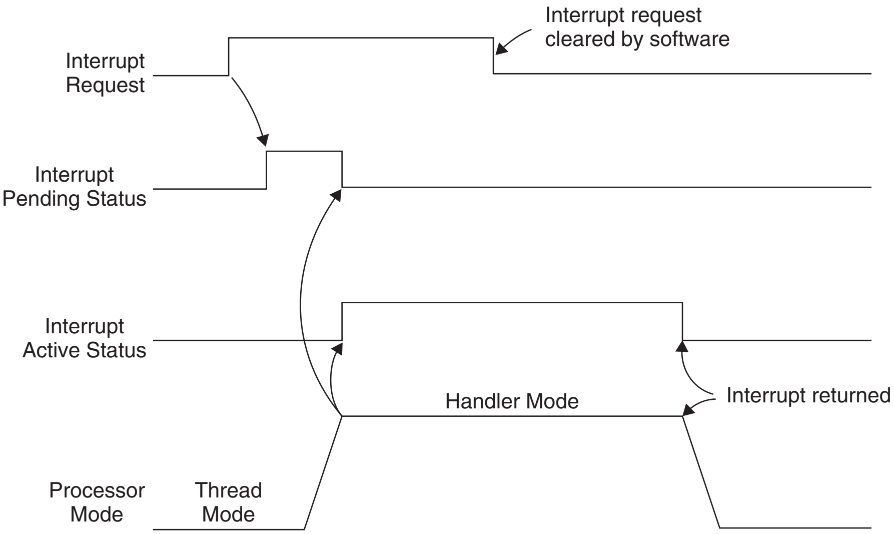


Figure 7.10 Interrupt Active Status Set as Processor Enters Handler

If an interrupt source continues to hold the interrupt request signal active, the interrupt will be pended again at the end of the interrupt service routine as shown in Figure 7.11. This is just like the traditional ARM7TDMI.

If an interrupt is pulsed several times before the processor starts processing it, it will be treated as one single interrupt request as illustrated in Figure 7.12.

If an interrupt is de-asserted and then pulsed again during the interrupt service routine, it will be pended again as shown in Figure 7.13.

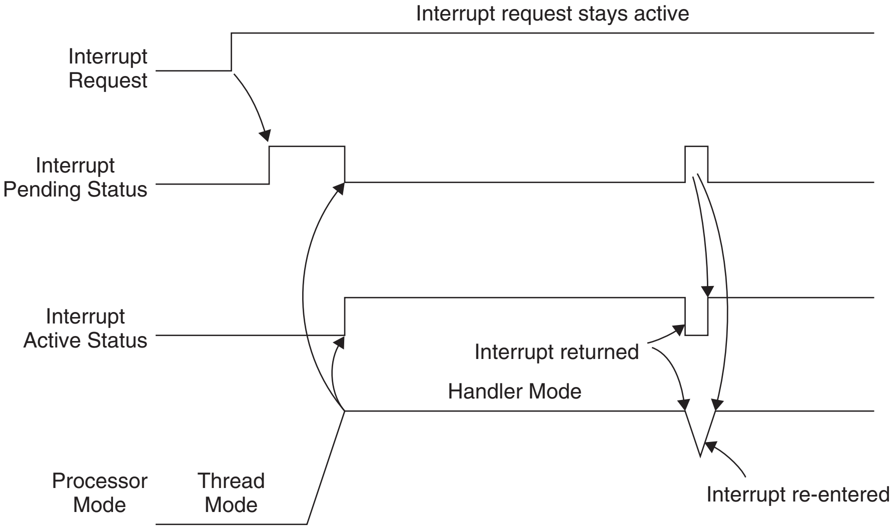


Figure 7.11 Continuous Interrupt Request Pends Again After Interrupt Exit

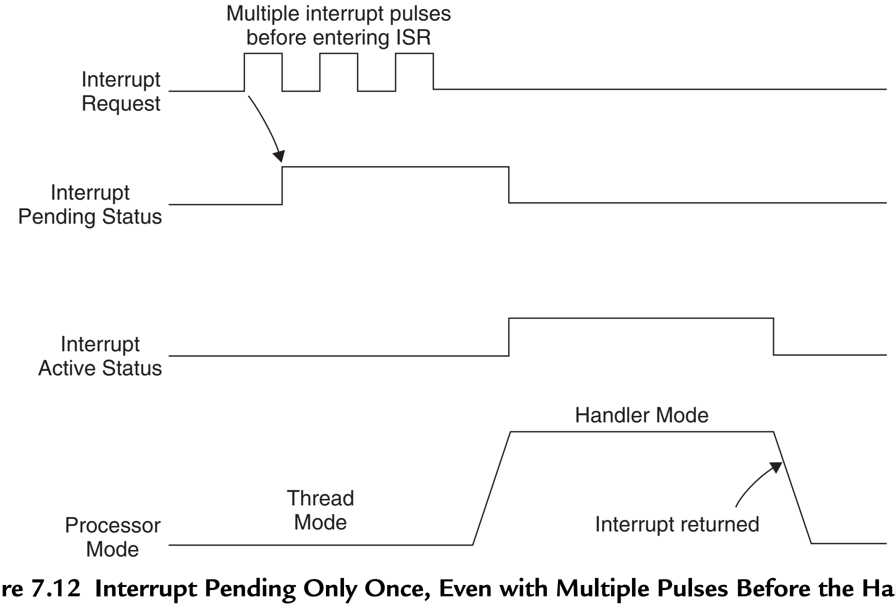


Figure 7.12 Interrupt Pending Only Once, Even with Multiple Pulses Before the Handler

Pending of an interrupt can happen even if the interrupt is disabled; the pended interrupt can then trigger the interrupt sequence when the enable is set later. As a result, before enabling an interrupt, it could be useful to check whether the pending register has been set. The interrupt source might have been activated previously and have set the pending status. If necessary, you can clear the pending status before you enable an interrupt.

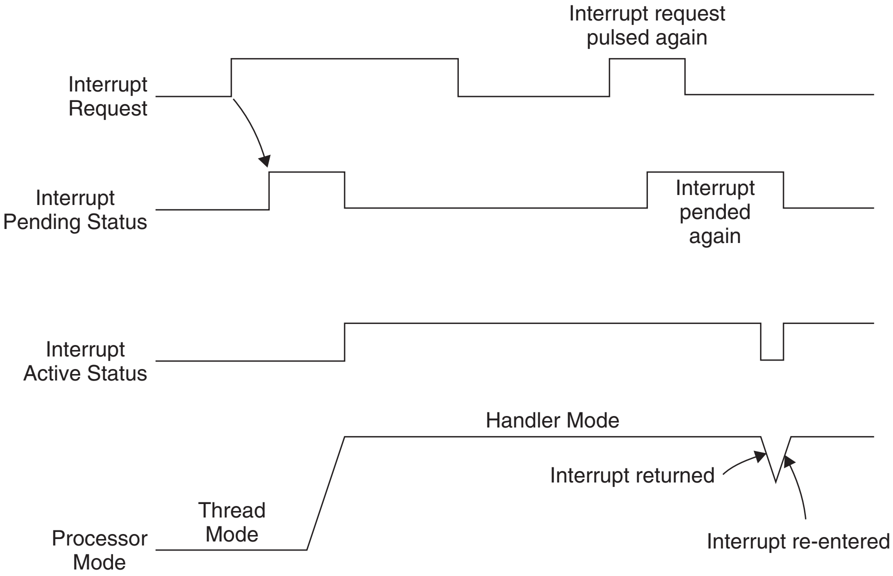


Figure 7.13 Interrupt Pending Occurs Again During the Handler

Fault Exceptions

A number of system exceptions are useful for fault handling. There are several categories of faults:

- Bus faults
- Memory management faults
- Usage faults
- Hard faults

Bus Faults

Bus faults are produced when an error response is received during a transfer on the AHB interfaces. It can happen at these stages:

- Instruction fetch, commonly called *prefetch abort*
- Data read/write, commonly called *data abort*

In the Cortex-M3, bus faults can also occur during a:

- Stack PUSH in the beginning of interrupt processing, called a *stacking error*
- Stack POP at the end of interrupt processing, called an *unstacking error*

- Reading of an interrupt vector address (vector fetch) when the processor starts the interrupt-handling sequence (a special case classified as a hard fault)

What Can Cause AHB Error Responses?

Bus faults occur when an error response is received on the AHB bus. The common causes are as follows:

- Attempts to access an invalid memory region (for example, a memory location with no memory attached)
- Device is not ready to accept a transfer (for example, trying to access SDRAM without initializing the SDRAM controller)
- Trying to carry out a transfer with a transfer size not supported by the target device (for example, doing a byte access to a peripheral register that must be accessed as a word)
- The device does not accept the transfer for various reasons (for example, a peripheral that can only be programmed at the privileged access level)

When these types of bus faults (except vector fetches) take place, and if the bus fault handler is enabled and no other exceptions with the same or higher priority are running, the bus fault handler will be executed. If the bus fault handler is enabled but at the same time the core receives another exception handler with higher priority, the bus fault exception will be pending. Finally, if the bus fault handler is not enabled or when the bus fault happens in an exception handler that has the same or higher priority than the bus fault handler, the hard fault handler will be executed instead. If another bus fault takes place when running the hard fault handler, the core will enter a lockup state.[2]

To enable the bus fault handler, you need to set the BUSFAULTENA bit in the System Handler Control and State register in the NVIC. Before doing that, make sure that the bus fault handler starting address is set up in the vector table if the vector table has been relocated to RAM.

So, how do you find out what went wrong when the processor entered the bus fault handler? The NVIC has a number of fault status registers. One of them is the Bus Fault Status Register (BFSR). From this register the bus fault handler can find out if the fault was caused by data/ instruction access or an interrupt stacking or unstacking operation.

For precise bus faults, the offending instruction can be located by the stacked program counter, and if the BFARVALID bit in BFSR is set, it is also possible to determine the memory location that caused the bus fault. This is done by reading another NVIC register

[2] More information on the lockup state is covered in Chapter 12.

called the Bus Fault Address Register (BFAR). However, the same information is not available for imprecise bus faults because by the time the processor receives the error, the processor could have already executed a number of other instructions.

Precise and Imprecise Bus Faults

Bus faults caused by data accesses can be further classified as precise or imprecise. In imprecise bus faults, the fault is caused by an already completed operation (such as a buffered write) that might have occurred a number of clock cycles ago. Precise bus faults are caused by the last completed operation—for example, a memory read is precise on the Cortex-M3 because the instruction cannot be completed until it receives the data.

The programmer's model for BFSR is as follows: It is 8 bits wide and can be accessed via byte transfer or with a word transfer to address 0xE000ED28 with BFSR in the second byte (see Table 7.8). The error indication bit is cleared when a 1 is written to it.

Table 7.8 Bus Fault Status Register (0xE000ED29)

Bits	Name	Type	Reset Value	Description
7	BFARVALID	–	0	Indicates BFAR is valid
6:5	–	–	–	–
4	STKERR	R/Wc	0	Stacking error
3	UNSTKERR	R/Wc	0	Unstacking error
2	IMPREISERR	R/Wc	0	Imprecise data access violation
1	PRECISERR	R/Wc	0	Precise data access violation
0	IBUSERR	R/Wc	0	Instruction access violation

Memory Management Faults

Memory management faults can be caused by memory accesses that violate the setup in the MPU or by certain illegal accesses (for example, trying to execute code from nonexecutable memory regions), which can trigger the fault, even if no MPU is presented.

Some of the common MPU faults include these:

- Access to memory regions not defined in MPU setup
- Writing to read-only regions
- An access in the user state to a region defined as privileged access only

When a memory management fault occurs, and if the memory management handler is enabled, the memory management fault handler will be executed. If the fault occurs at the same time a higher-priority exception takes place, the other exceptions will be handled first and the memory management fault will be pended. If the processor is already running an exception handler with same or higher priority or if the memory management fault handler is not enabled, the hard fault handler will be executed instead. If a memory management fault takes place inside the hard fault handler or the NMI handler, the processor will enter the lockup state.

Like the bus fault handler, the memory management fault handler needs to be enabled. This is done by the MEMFAULTENA bit in the System Handler Control and State register in the NVIC. If the vector table has been relocated to RAM, the memory management fault handler starting address should be set up in the vector table first.

The NVIC contains a Memory Management Fault Status Register (MFSR) to indicate the cause of the memory management fault. If the status register indicates that the fault is a data access violation (DACCVIOL bit) or an instruction access violation (IACCVIOL bit), the offending code can be located by the stacked program counter. If the MMARVALID bit in the MFSR is set, it is also possible to determine the memory address location that caused the fault from the Memory Management Address Register (MMAR) in the NVIC.

The programmer's model for the MFSR is shown in Table 7.9. It is 8 bits wide and can be accessed via byte transfer or with a word transfer to address 0xE000ED28, with the MFSR in the lowest byte. As with other fault status registers, the fault status bit can be cleared by writing 1 to the bit.

Table 7.9 Memory Management Fault Status Register (0xE000ED28)

Bits	Name	Type	Reset Value	Description
7	MMARVALID	–	0	Indicates the MMAR is valid
6:5	–	–	–	–
4	MSTKERR	R/Wc	0	Stacking error
3	MUNSTKERR	R/Wc	0	Unstacking error
2	–	–	–	–
1	DACCVIOL	R/Wc	0	Data access violation
0	IACCVIOL	R/Wc	0	Instruction access violation

Usage Faults

Usage faults can be caused by a number of things:

- Undefined instructions
- Coprocessor instructions (the Cortex-M3 processor does not support a coprocessor, but it is possible to use the fault exception mechanism to run software compiled for other Cortex processors via coprocessor emulation)

- Trying to switch to the ARM state (software can use this faulting mechanism to test whether the processor it is running on supports ARM code; since the Cortex-M3 does not support the ARM state, a usage fault takes place if there's an attempt to switch)
- Invalid interrupt return (Link Register contains invalid/incorrect values)
- Unaligned memory accesses using multiple load or store instructions

It is also possible, by setting up certain control bits in the NVIC, to generate usage faults for:

- Divide by zero
- Any unaligned memory accesses

When a usage fault occurs and if the usage fault handler is enabled, normally the usage fault handler will be executed. However, if at the same time a higher-priority exception takes place, the usage fault will be pended. If the processor is already running an exception handler with the same or higher priority or if the usage fault handler is not enabled, the hard fault handler will be executed instead. If a usage fault takes place inside the hard fault handler or the NMI handler, the processor will enter the lockup state.

The usage fault handler is enabled by setting the USGFAULTENA bit in the System Handler Control and State register in the NVIC. If the vector table has been relocated to RAM, the usage fault handler starting address should be set up in the vector table first.

The NVIC provides a Usage Fault Status Register (UFSR) for the usage fault handler to determine the cause of the fault. Inside the handler, the program code that causes the error can also be located using the stacked program counter value.

Accidentally Switching to the ARM State

One of the most common causes of usage faults is accidentally trying to switch the processor to ARM mode. This can happen if you load a new value to PC with the LSB equal to 0—for example, if you try to branch to an address in a register (BX LR) without setting the LSB, have zero in the LSB of a vector in the exception vector table, or the stacked PC value to be read by POP {PC} is modified manually, leaving the LSB cleared. When these situations happen, the usage fault exception will take place with the INVSTATE bit in the UFSR set.

The UFSR is shown in Table 7.10. It occupies 2 bytes and can be accessed by half word transfer or as a word transfer to address 0xE000ED28 with the UFSR in the upper half word. As with other fault status registers, the fault status bit can be cleared by writing 1 to the bit.

Table 7.10 Usage Fault Status Register (0xE000ED2A)

Bits	Name	Type	Reset Value	Description
9	DIVBYZERO	R/Wc	0	Indicates a divide by zero has taken place (can be set only if DIV_0_TRP is set)
8	UNALIGNED	R/Wc	0	Indicates that an unaligned access fault has taken place
7:4	–	–	–	–
3	NOCP	R/Wc	0	Attempts to execute a coprocessor instruction
2	INVPC	R/Wc	0	Attempts to do an exception with a bad value in the EXC_RETURN number
1	INVSTATE	R/Wc	0	Attempts to switch to an invalid state (e.g., ARM)
0	UNDEFINSTR	R/Wc	0	Attempts to execute an undefined instruction

Hard Faults

The hard fault handler can be caused by usage faults, bus faults, and memory management faults if their handler cannot be executed. In addition, it can also be caused by a bus fault during vector fetch (reading of a vector table during exception handling). In the NVIC there is a hard fault status register that can be used to determine whether the fault was caused by a vector fetch. If not, the hard fault handler will need to check the other fault status registers to determine the cause of the hard fault.

Details of the Hard Fault Status Register (HFSR) are shown in Table 7.11. As with other fault status registers, the fault status bit can be cleared by writing 1 to the bit.

Table 7.11 Hard Fault Status Register (0xE000ED2C)

Bits	Name	Type	Reset Value	Description
31	DEBUGEVT	R/Wc	0	Indicates hard fault is triggered by debug event
30	FORCED	R/Wc	0	Indicates hard fault is taken because of bus fault, memory management fault, or usage fault
29:2	–	–	–	–
1	VECTBL	R/Wc	0	Indicates hard fault is caused by failed vector fetch
0	–	–	–	–

Dealing with Faults

During software development, we can use the Fault Status Registers (FSRs) to determine the causes of errors in the program and correct them. A troubleshooting guide is included in Appendix E of this book for common causes of various faults. In a real running system, the situation is different. After the cause of a fault is determined, the software will have to decide what to do next. In systems that run an OS, the offending tasks or applications could be terminated. In some other cases, the system might need a reset. The requirements of fault recovery depend on the target application. Doing it properly could make the product more robust, but it is best to prevent the faults from happening in the first place. Here are some fault-handling methods:

- Reset: This can be carried out using the VECTRESET control bit in the Application Interrupt and Reset Control register in the NVIC. This will reset the processor but not the whole chip. Depending on the chip's reset design, some Cortex-M3 chips can be reset using the SYSRESETREQ in the same register. This could provide a full system reset.
- Recovery: In some cases it might be possible to resolve the problem that caused the fault exception. For example, in the case of coprocessor instructions, the problem can be resolved using coprocessor emulation software.
- Task termination: For systems running an OS, it is likely that the task that caused the fault will be terminated and restarted if needed.

The FSRs retain their status until they are cleared manually. Fault handlers should clear the fault status bit they have dealt with. Otherwise, the next time another fault takes place, the fault handler will be invoked again and could mistake that the first fault still exists and so will try to deal with it again. The FSRs use a write-to-clear mechanism (clear by writing 1 to the bits that need to be cleared).

Chip manufacturers can also include an auxiliary FSR in the chip to indicate other fault situations. The implementation of an AFSR depends on individual chip design requirements.

SVC and PendSV

SVC (System Service Call) and PendSV (Pended System Call) are two exceptions targeted at software and operating systems. SVC is for generating system function calls. For example, instead of allowing user programs to directly access hardware, an operating system may provide access to hardware via an SVC. So when a user program wants to use certain hardware, it generates the SVC exception using SVC instructions, and then the software exception handler in the operating system is executed and provides the service the user application requested. In this way, access to hardware is under the control of the OS, which can provide a more robust system by preventing the user applications from directly accessing hardware.

SVC can also make software more portable because the user application does not need to know the programming details of the hardware. The user program will only need to know the Application Programming Interface (API) function ID and parameters; the actual hardware-level programming is handled by device drivers.

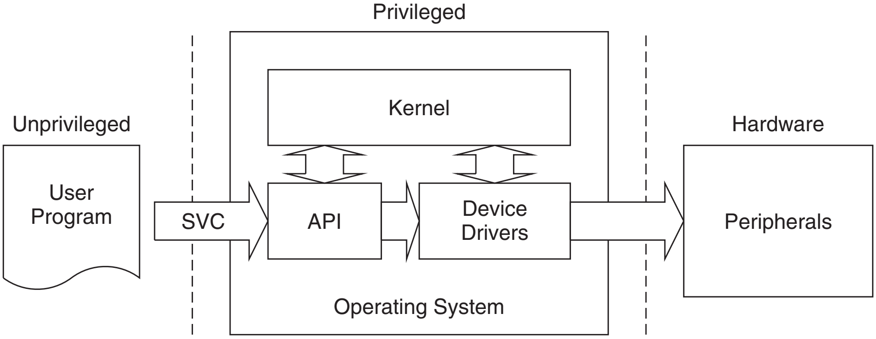


Figure 7.14 SVC as a Gateway for OS Functions

SVC is generated using the SVC instruction. An immediate value is required for this instruction, which works as a parameter-passing method. The SVC exception handler can then extract the parameter and determine what action it needs to perform. For example:

```
SVC 0x3 ; Call SVC function 3
```

When the SVC handler is executed, you can determine the immediate data value in the SVC instruction by reading the stacked Program Counter value, then reading the instruction from that address and masking out the unneeded bits. If the system uses a PSP for user applications, you might need to determine which stack was used first. This can be determined from the link register value when the handler is entered. (This topic is covered in more depth in Chapter 8).

SVC and SWI (ARM7)

If you have used traditional ARM processors (such as the ARM7), you might know that they have a software interrupt instruction (SWI). The SVC has a similar function, and in fact the binary encoding of SVC instructions is the same as SWI in ARM7. However, since the exception model has changed, this instruction is renamed to make sure that programmers will properly port software code from ARM7 to the Cortex-M3.

Due to the interrupt priority model in the Cortex-M3, you cannot use SVC inside an SVC handler (because the priority is the same as the current priority). Doing so will result in a usage fault. For the same reason, you cannot use SVC in an NMI handler or a hard fault handler.

PendSV (Pended System Call) works with SVC in the OS. Although SVC (by SVC instruction) cannot be pended (an application calling SVC will expect the required task to be done immediately), PendSV can be pended and is useful for an OS to pend an exception so that an action can be performed after other important tasks are completed. PendSV is generated by writing 1 to the NVIC PendSV pending register.

A typical use of PendSV is context switching (switching between tasks). For example, a system might have two active tasks, and context switching can be triggered by:

- Calling an SVC function
- The system timer (SYSTICK)

Let's look at a simple example of having only two tasks in a system, and a context switch is triggered by SYSTICK exceptions (see Figure 7.15).

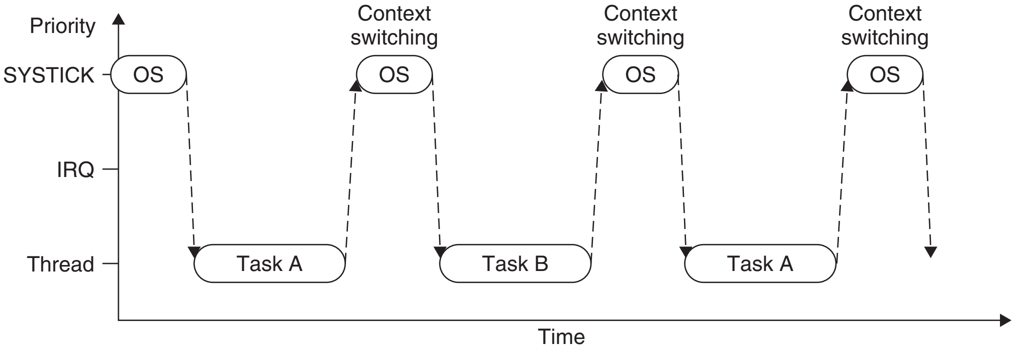


Figure 7.15 A Simple Scenario Using SYSTICK to Switch Between Two Tasks

If an interrupt request takes place before the SYSTICK exception, the SYSTICK exception will preempt the IRQ handler. In this case, the OS should not carry out the context switching. Otherwise the IRQ handler process will be delayed, and for the Cortex-M3, a usage fault could be generated if the OS tries to switch to Thread mode when an interrupt is active.

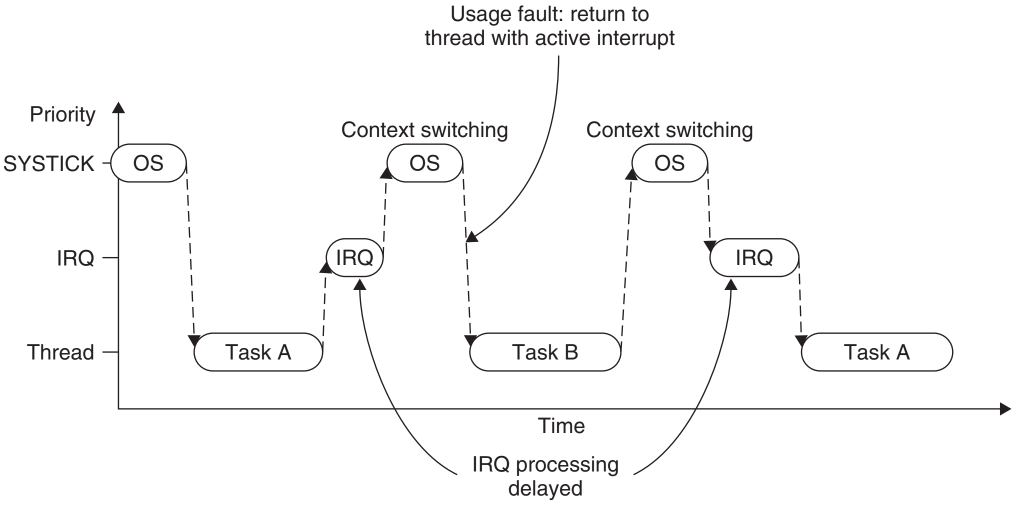


Figure 7.16 Problem with Context Switching at the IRQ

To avoid the problem of delaying the IRQ processing, some OS implementations carry out only context switching if they detect that none of the IRQ handlers are being executed. However, this can result in a very long delay for task switching, especially if the frequency of an interrupt source is close to that of the SYSTICK exception.

The PendSV exception solves the problem by delaying the context-switching request until all other IRQ handlers have completed their processing. To do this, the PendSV is programmed as the lowest priority exception. If the OS detects that an IRQ is currently active (IRQ handler running and preempted by SYSTICK), it defers the context switching by pending the PendSV exception.

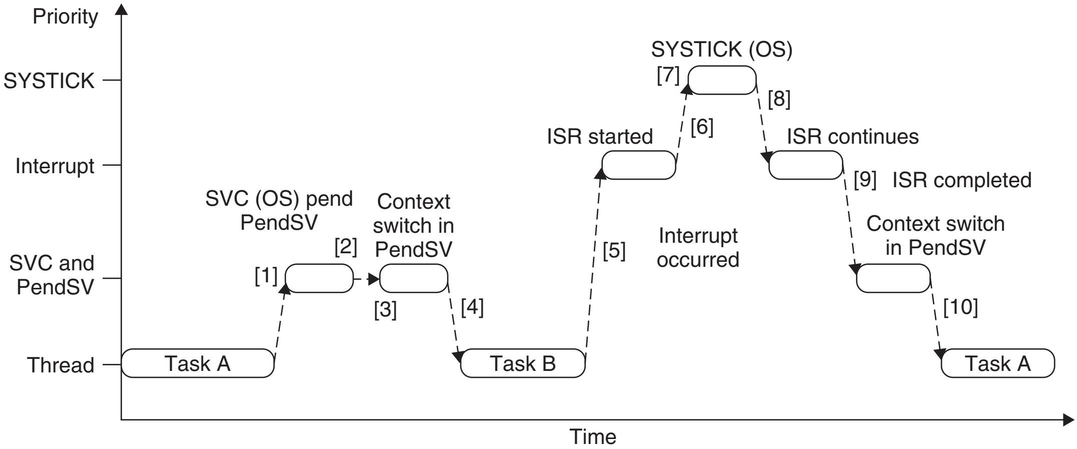


Figure 7.17 Example Context Switching with PendSV

1. Task A calls SVC for task switching (for example, waiting for some work to complete).
2. The OS receives the request, prepares for context switching, and pends the PendSV exception.
3. When the CPU exits SVC, it enters PendSV immediately and does the context switch.
4. When PendSV finishes and returns to Thread level, it executes Task B.
5. An interrupt occurs and the interrupt handler is entered.
6. While running the interrupt handler routine, a SYSTICK exception (for OS tick) takes place.
7. The OS carries out the essential operation, then pends the PendSV exception and gets ready for the context switch.
8. When the SYSTICK exception exits, it returns to the interrupt service routine.
9. When the interrupt service routine completes, the PendSV starts and does the actual context switch operations.
10. When PendSV is complete, the program returns to Thread level; this time it returns to Task A and continues the processing.

CHAPTER 8

The NVIC and Interrupt Control

In This Chapter:

- NVIC Overview
- The Basic Interrupt Configuration
- Interrupt Enable and Clear Enable
- Interrupt Pending and Clear Pending
- Example Procedures in Setting Up an Interrupt
- Software Interrupts
- The SYSTICK Timer

NVIC Overview

As we've seen, the Nested Vectored Interrupt Controller, or NVIC, is an integrated part of the Cortex-M3 processor. It is closely linked to the Cortex-M3 CPU core logic. Its control registers are accessible as memory-mapped devices. Besides control registers and control logic for interrupt processing, the NVIC also contains control registers for the MPU, the SYSTICK Timer, and debugging controls. In this chapter we'll examine the control logic for interrupt processing. MPU and debugging control logic are discussed in later chapters.

The NVIC supports 1 to 240 external interrupt inputs (commonly known as IRQs). The exact number of supported interrupts is determined by the chip manufacturers when they develop their Cortex-M3 chips. In addition, the NVIC also has a Nonmaskable Interrupt (NMI) input. The actual function of the NMI is also decided by the chip manufacturer. In some cases this NMI cannot be controlled from an external source.

The NVIC can be accessed as memory location 0xE000E000. Most of the interrupt control/status registers are accessible only in privileged mode, except the Software Trigger Interrupt register, which can be set up to be accessible in user mode. The interrupt control/status register can be accessed in word, half word, or byte transfers.

In addition, a few other interrupt-masking registers are also involved in the interrupts. They are the "special registers" covered in Chapter 3 and are accessed via MRS and MSR instructions.

The Basic Interrupt Configuration

Each external interrupt has several registers associated with it:

- Enable and clear enable registers
- Set-pending and clear-pending registers
- Priority level
- Active status

In addition, a number of other registers can also affect the interrupt processing:

- Exception-masking registers (PRIMASK, FAULTMASK, and BASEPRI)
- Vector Table Offset register
- Software Trigger Interrupt register
- Priority Group

Interrupt Enable and Clear Enable

The Interrupt Enable register is programmed via two addresses. To set the enable bit, you need to write to the SETENA register address; to clear the enable bit, you need to write to the CLRENA register address. In this way, enabling or disabling an interrupt will not affect other interrupt enable states. The SETENA/CLRENA registers are 32 bits wide; each bit represents one interrupt input.

Since there could be more than 32 external interrupts in the Cortex-M3 processor, you might find more than one SETENA and CLRENA register—for example, SETENA0, SETENA1, and so on (see Table 8.1). Only the enable bits for interrupts that exist are implemented. So if you only have 32 interrupt inputs, you will only have SETENA0 and CLRENA0. The SETENA and CLRENA registers can be accessed as word, half word, or byte. Since the first 16 exception types are system exceptions, external interrupt #0 has a start exception number of 16 (see Table 7.2).

Interrupt Pending and Clear Pending

If an interrupt takes place but cannot be executed immediately (for instance, if another higher-priority interrupt handler is running), it will be pended. The interrupt-pending status can be accessed through the Interrupt Set Pending (SETPEND) and Interrupt Clear Pending (CLRPEND) registers. Similarly to the enable registers, the pending status

Table 8.1 Interrupt Set Enable Registers and Interrupt Clear Enable Registers (0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C)

Address	Name	Type	Reset Value	Description
0xE000E100	SETENA0	R/W	0	Enable for external interrupt #0–31 bit[0] for interrupt #0 (exception #16) bit[1] for interrupt #1 (exception #17) ... bit[31] for interrupt #31 (exception #47) Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
0xE000E104	SETENA1	R/W	0	Enable for external interrupt #32–63 Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
0xE000E108	SETENA2	R/W	0	Enable for external interrupt #64–95 Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
...	–	–	–	–
0xE000E180	CLRENA0	R/W	0	Clear enable for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 ... bit[31] for interrupt #31 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status
0xE000E184	CLRENA1	R/W	0	Clear Enable for external interrupt #32–63 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status
0xE000E188	CLRENA2	R/W	0	Clear enable for external interrupt #64–95 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status
...	–	–	–	–

controls might contain more than one register if there are more than 32 external interrupt inputs.

The pending status registers can be changed, so you can cancel a current pended exception or generate software interrupts via the SETPEND register (see Table 8.2).

Table 8.2 Interrupt Set Pending Registers and Interrupt Clear Pending Registers (0xE000E200-0xE000E21C, 0xE000E280-0xE000E29C)

Address	Name	Type	Reset Value	Description
0xE000E200	SETPEND0	R/W	0	Pending for external interrupt #0–31 bit[0] for interrupt #0 (exception #16) bit[1] for interrupt #1 (exception #17) ... bit[31] for interrupt #31 (exception #47) Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
0xE000E204	SETPEND1	R/W	0	Pending for external interrupt #32–63 Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
0xE000E208	SETPEND2	R/W	0	Pending for external interrupt #64–95 Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
...	–	–	–	–
0xE000E280	CLRPEND0	R/W	0	Clear pending for external interrupt #0–31 bit[0] for interrupt #0 (exception #16) bit[1] for interrupt #1 (exception #17) ... bit[31] for interrupt #31 (exception #47) Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current pending status
0xE000E284	CLRPEND1	R/W	0	Clear pending for external interrupt #32–63 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current pending status
0xE000E288	CLRPEND2	R/W	0	Clear pending for external interrupt #64–95 Write 1 to clear bit to 1; write 0 has no effect Read value indicates the current pending status
...	–	–	–	–

Priority Levels

Each external interrupt has an associated priority-level register, which has a maximum width of 8 bits and a minimum width of 3 bits. As described in the previous chapter, each register can be further divided into preempt priority level and subpriority level based on priority group settings. The priority-level registers can be accessed as byte, half word, or word. The number of priority-level registers depends on how many external interrupts the chip contains (see Table 8.3). The priority level configuration registers details can be found in Appendix D, Table D.18.

Table 8.3 Interrupt Priority-Level Registers (0xE000E400-0xE000E4EF)

Address	Name	Type	Reset Value	Description
0xE000E400	PRI_0	R/W	0 (8-bit)	Priority-level external interrupt #0
0xE000E401	PRI_1	R/W	0 (8-bit)	Priority-level external interrupt #1
...	–	–	–	–
0xE000E41F	PRI_31	R/W	0 (8-bit)	Priority-level external interrupt #31
...	–	–	–	–

Active Status

Each external interrupt has an active status bit. When the processor starts the interrupt handler, the bit is set to 1 and cleared when the interrupt return is executed. However, during an interrupt service routine execution, a higher-priority interrupt might occur and cause a preemption. During this period, despite the fact that the processor is executing another interrupt handler, the previous interrupt is still defined as active. The active registers are 32-bit but can also be accessed using half word or byte-size transfers. If there are more than 32 external interrupts, there will be more than one active register. The active status registers for external interrupts are read-only (see Table 8.4).

Table 8.4 Interrupt Active Status Registers (0xE000E300-0xE000E31C)

Address	Name	Type	Reset Value	Description
0xE000E300	ACTIVE0	R	0	Active status for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 ... bit[31] for interrupt #31
0xE000E304	ACTIVE1	R	0	Active status for external interrupt #32–63
...	–	–	–	–

PRIMASK and FAULTMASK Special Registers

The PRIMASK register is used to disable all exceptions except NMI and hard fault. It effectively changes the current priority level to 0 (highest programmable level). This register is programmable using MRS and MSR instructions. For example:

```
MOV    R0, #1
MSR    PRIMASK, R0    ; Write 1 to PRIMASK to disable all
                      ; interrupts
```

and:

```
    MOV     R0, #0
    MSR     PRIMASK, R0    ; Write 0 to PRIMASK to allow interrupts
```

PRIMASK is useful for temporarily disabling all interrupts for critical tasks. When PRIMASK is set, if a fault takes place, the hard fault handler will be executed.

FAULTMASK is just like PRIMASK except that it changes the effective current priority level to −1 so that even the hard fault handler is blocked. Only the NMI can be executed when FAULTMASK is set.

FAULTMASK is cleared automatically upon exiting the exception handler. Both FAULTMASK and PRIMASK registers cannot be set in the user state.

The BASEPRI Special Register

In some cases you might want to disable interrupts only with priority lower than a certain level. In this case, you could use the BASEPRI register. To do this, simply write the required masking priority level to the BASEPRI register. For example, if you want to block all exceptions with priority level equal to or lower than 0x60, you can write the value to BASEPRI:

```
    MOV     R0, #0x60
    MSR     BASEPRI, R0    ; Disable interrupts with priority
                           ; 0x60-0xFF
```

To cancel the masking, just write 0 to the BASEPRI register:

```
    MOV     R0, #0x0
    MSR     BASEPRI, R0    ; Turn off BASEPRI masking
```

The BASEPRI register can also be accessed using the BASEPRI_MAX register name. It is actually the same register, but when you use it with this name it will give you a conditional write operation. (As far as hardware is concerned, BASEPRI and BASEPRI_MAX are the same register, but in the assembler code they use different register name coding.) When you use BASEPRI_MAX as a register, it can only be changed to a higher priority level; it cannot be changed to lower priority levels. For example, consider the following instruction sequence:

```
    MOV     R0, #0x60
    MSR     BASEPRI_MAX, R0    ; Disable interrupts with priority
                               ; 0x60, 0x61,..., etc
    MOV     R0, #0xF0
    MSR     BASEPRI_MAX, R0    ; This write will be ignored because
                               ; it is lower
                               ; level than 0x60
    MOV    R0, #0x40
    MSR    BASEPRI_MAX, R0   ; This write is allowed and change the
                             ; masking level to 0x40
```

To change to a lower masking level or disable the masking, the BASEPRI register name should be used. The BASEPRI/ BASEPRI_MAX register cannot be set in the user state.

As with other priority-level registers, the formatting of the BASEPRI register is affected by the number of implemented priority register widths. For example, if only 3 bits are implemented for priority-level registers, BASEPRI can be programmed as 0x00, 0x20, 0x40 … 0xC0, 0xE0.

Configuration Registers for Other Exceptions

Usage faults, memory management faults, and bus fault exceptions are enabled by the System Handler Control and State Register (0xE000ED24). The pending status of faults and active status of most system exceptions are also available from this register (see Table 8.5).

Table 8.5 The System Handler Control and State Register (0xE000ED24)

Bits	Name	Type	Reset Value	Description
18	USGFAULTENA	R/W	0	Usage fault handler enable
17	BUSFAULTENA	R/W	0	Bus fault handler enable
16	MEMFAULTENA	R/W	0	Memory management fault enable
15	SVCALLPENDED	R/W	0	SVC pended; SVCall was started but was replaced by a higher-priority exception
14	BUSFAULTPENDED	R/W	0	Bus fault pended; bus fault handler was started but was replaced by a higher-priority exception
13	MEMFAULTPENDED	R/W	0	Memory management fault pended; memory management fault started but was replaced by a higher-priority exception
12	USGFAULTPENDED	R/W	0	Usage fault pended; usage fault started but was replaced by a higher-priority exception
11	SYSTICKACT	R/W	0	Read as 1 if SYSTICK exception is active
10	PENDSVACT	R/W	0	Read as 1 if PendSV exception is active
8	MONITORACT	R/W	0	Read as 1 if debug monitor exception is active
7	SVCALLACT	R/W	0	Read as 1 if SVCall exception is active
3	USGFAULTACT	R/W	0	Read as 1 if usage fault exception is active
1	BUSFAULTACT	R/W	0	Read as 1 if bus fault exception is active
0	MEMFAULTACT	R/W	0	Read as 1 if memory management fault is active
Note: Bit 12 (USGFAULTPENDED) is not available on revision 0 of Cortex-M3.				

Be cautious when writing to this register; make sure that the active status bits of system exceptions are not changed accidentally. Otherwise, if an activated system exception has its active state cleared by accident, a fault exception will be generated when the system exception handler generates an exception exit.

Pending for NMI, the SYSTICK timer, and PendSV is programmable via the Interrupt Control and State register. In this register, quite a number of the bit fields are for debugging purposes. In most cases only the pending bits would be useful for application development (see Table 8.6).

Table 8.6 Interrupt Control and State Register (0xE000ED04)

Bits	Name	Type	Reset Value	Description
31	NMIPENDSET	R/W	0	NMI pended
28	PENDSVSET	R/W	0	Write 1 to pend system call Read value indicates pending status
27	PENDSVCLR	W	0	Write 1 to clear PendSV pending status
26	PENDSTSET	R/W	0	Write 1 to pend SYSTICK exception Read value indicates pending status
25	PENDSTCLR	W	0	Write 1 to clear SYSTICK pending status
23	ISRPREEMPT	R	0	Indicates that a pending interrupt is going to be active in the next step (for debug)
22	ISRPENDING	R	0	External interrupt pending (excluding system exceptions such as NMI for fault)
21:12	VECTPENDING	R	0	Pending ISR number
11	RETTOBASE	R	0	Set to 1 when the processor is running an exception handler; will return to Thread level if interrupt return and no other exceptions pending
9:0	VECTACTIVE	R	0	Current running interrupt service routine

Example Procedures in Setting Up an Interrupt

Here is a simple example procedure for setting up an interrupt:

1. When the system boots up, the priority group register might need to be set up. By default the priority group 0 is used (bit[7:1] of priority level is the preemption level and bit[0] is the subpriority level).

2. Copy the hard fault and NMI handlers to a new vector table location if vector table relocation is required. (In simple applications, this might not be needed.)

3. The Vector Table Offset register should also be set up to get the vector table ready (optional).
4. Set up the interrupt vector for the interrupt. Since the vector table could have been relocated, you might need to read the Vector Table Offset register, then calculate the correct memory location for your interrupt handler. This step might not be needed if the vector is hardcoded in ROM.
5. Set up the priority level for the interrupt.
6. Enable the interrupt.

The program in assembly might be something like this:

```
LDR   R0, =0xE000ED0C    ; Application Interrupt and Reset
                         ; Control Register
LDR   R1, =0x05FA0500    ; Priority Group 5 (2/6)
STR   R1, [R0]           ; Set Priority Group
...
MOV   R4,#8              ; Vector Table in ROM
LDR   R5,=(NEW_VECT_TABLE+8)
LDMIA R4!,{R0-R1}        ; Read vectors address for NMI and
                         ; Hard Fault
STMIA R5!,{R0-R1}        ; Copy vectors to new vector table
...
LDR   R0,=0xE000ED08     ; Vector Table Offset Register
LDR   R1,=NEW_VECT_TABLE
STR   R1,[R0]            ; Set vector table to new location
...
LDR   R0,=IRQ7_Handler   ; Get starting address of IRQ#7 handler
LDR   R1,=0xE000ED08     ; Vector Table Offset Register
LDR   R1,[R1]
ADD   R1, R1, #(4*(7+16)); Calculate IRQ#7 handler vector
                         ; address
STR   R0,[R1]            ; Setup vector for IRQ#7
...
LDR   R0,=0xE000E400     ; External IRQ priority base
MOV   R1, #0xC0
STRB  R1,[R0,#7]         ; Set IRQ#7 priority to 0xC0
...
LDR   R0,=0xE000E100     ; SETEN register
MOV   R1,#(1<<7)         ; IRQ#7 enable bit (value 0x1 shifted
                         ; by 7 bits)
STR   R1,[R0]            ; Enable the interrupt
```

In addition, make sure that you have enough stack memory if you allow a large number of nested interrupt levels. Since exception handlers always use the MSP, the main stack memory should contain enough space for the largest number of nesting interrupts.

If the application is stored in ROM and there is no need to change the exception handlers, we can have the whole vector table coded in the beginning of ROM in the Code region (0x00000000). This way, the vector table offset will always be 0 and the interrupt vector is already in ROM. The only steps required to set up an interrupt will be:

1. Set up the priority group, if needed.
2. Set up the priority of the interrupt.
3. Enable the interrupt.

In cases where the software needs to be able to run on a number of hardware devices, it might be necessary to determine:

- The number of interrupts supported in the design
- The number of bits in priority-level registers

The Cortex-M3 has an Interrupt Controller Type register that gives the number of interrupt inputs supported, in granularities of 32 (see Table 8.7). Alternatively, you can detect the exact number of external interrupts by performing a read/write test to interrupt configuration registers such as SETEN or priority registers.

Table 8.7 Interrupt Controller Type Register (0xE000E004)

Bits	Name	Type	Reset Value	Description
4:0	INTLINESNUM	R	–	Number of interrupt inputs in step of 32 0 = 1 to 32 1 = 33 to 64 …

To determine the number of bits implemented for interrupt priority-level registers, you can write 0xFF to one of the priority-level registers, then read it back and see how many bits are set. The minimum number is three. In that case you should get a read-back value of 0xE0.

Software Interrupts

Software interrupts can be generated in more than one way. The first one is to use the SETPEND register; the second solution is to use the Software Trigger Interrupt Register (STIR), outlined in Table 8.8.

Table 8.8 Software Trigger Interrupt Register (0xE000EF00)

Bits	Name	Type	Reset Value	Description
8:0	INTID	W	–	Writing the interrupt number sets the pending bit of the interrupt; for example, write 0 to pend external interrupt #0

System exceptions (NMI, faults, PendSV, and so on) cannot be pended using this register. By default, a user program cannot write to the NVIC; however, if it is necessary for a user program to write to this register, the bit 1 (USERSETMPEND) of the NVIC Configuration Control Register (0xE000ED14) can be set to allow user access to the NVIC's STIR.

The SYSTICK Timer

The SYSTICK Timer is integrated with the NVIC and can be used to generate a SYSTICK exception (exception type #15). In many operating systems, a hardware timer is used to generate interrupts so that the OS can carry out task management—for example, to allow multiple tasks to run at different time slots and to make sure that no single task can lock up the whole system. To do that, the timer needs to be able to generate interrupts, and if possible, it should be protected from user tasks so that user applications cannot change the timer behavior.

The Cortex-M3 processor includes a simple timer. Since all Cortex-M3 chips have the same timer, porting software between different Cortex-M3 products is simplified. The timer is a 24-bit down counter. It can use the internal clock (FCLK, the free running clock signal on the Cortex-M3 processor) or external clock (the STCLK signal on the Cortex-M3 processor). However, the source of the STCLK will be decided by chip designers, so the clock frequency might vary between products. You should check the chip's datasheet carefully when selecting a clock source.

The SYSTICK Timer can be used to generated interrupts. It has a dedicated exception type and exception vector. It makes porting operating systems and software easier because the process will be the same across different Cortex-M3 products.

The SYSTICK Timer is controlled by four registers, shown in Tables 8.9–8.12.

Table 8.9 SYSTICK Control and Status Register (0xE000E010)

Bits	Name	Type	Reset Value	Description
16	COUNTFLAG	R	0	Read as 1 if counter reaches 0 since last time this register is read; clear to 0 automatically when read or when current counter value is cleared
2	CLKSOURCE	R/W	0	0 = External reference clock (STCLK) 1 = Use core clock
1	TICKINT	R/W	0	1 = Enable SYSTICK interrupt generation when SYSTICK timer reaches 0 0 = Do not generate interrupt
0	ENABLE	R/W	0	SYSTICK timer enable

Table 8.10 SYSTICK Reload Value Register (0xE000E014)

Bits	Name	Type	Reset Value	Description
23:0	RELOAD	R/W	0	Reload value when timer reaches 0

Table 8.11 SYSTICK Current Value Register (0xE000E018)

Bits	Name	Type	Reset Value	Description
23:0	CURRENT	R/Wc	0	Read to return current value of the timer. Write to clear counter to 0. Clearing of current value also clears COUNTFLAG in SYSTICK Control and Status Register

Table 8.12 SYSTICK Calibration Value Register (0xE000E01C)

Bits	Name	Type	Reset Value	Description
31	NOREF	R	–	1 = No external reference clock (STCLK not available) 0 = External reference clock available
30	SKEW	R	–	1 = Calibration value is not exactly 10 ms 0 = Calibration value is accurate
23:0	TENMS	R/W	0	Calibration value for 10 ms.; chip designer should provide this value via Cortex-M3 input signals. If this value is read as 0, calibration value is not available

The Calibration Value register provides a solution for applications to generate the same SYSTICK interrupt interval when running on various Cortex-M3 products. To use it, just write the value in TENMS to the reload value register. This will give an interrupt interval of about 10 ms. For other interrupt timing intervals, the software code will need to calculate a new suitable value from the calibration value. However, the TENMS field might not be available in all Cortex-M3 products (the calibration input signals to the Cortex-M3 might have been tied low), so check with your manufacturer's datasheets before using this feature.

Aside from being a system tick timer for operating systems, the SYSTICK Timer can be used in a number of ways: as an alarm timer, for timing measurement, and more. Note that the SYSTICK Timer stops counting when the processor is halted during debugging.

CHAPTER 9

Interrupt Behavior

In This Chapter:

- Interrupt/Exception Sequences
- Exception Exits
- Nested Interrupts
- Tail-Chaining Interrupts
- Late Arrivals
- More on the Exception Return Value
- Interrupt Latency
- Faults Related to Interrupts

Interrupt/Exception Sequences

When an exception takes place, a number of things happen:

- Stacking (pushing eight registers' contents to stack)
- Vector fetch (reading the exception handler starting address from the vector table)
- Update of the stack pointer, link register, and program counter

Stacking

When an exception takes place, the registers PC, PSR, R0–R3, R12, and LR are pushed to the stack. If the code that is running uses the PSP, the process stack will be used; if the code that is running uses the MSP, the main stack will be used. Afterward, the main stack will always be used during the handler, so all nested interrupts will use the main stack.

The order of stacking is shown in Figure 9.1 (assuming that the SP value is *N* before the exception). Due to the pipeline nature of the AHB interface, the address and data are offset by one pipeline state.

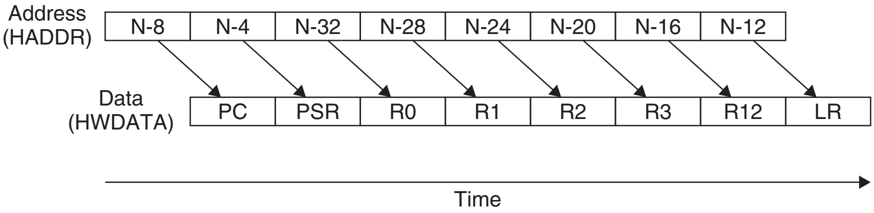


Figure 9.1 Stacking Sequence

The values of PC and PSR are stacked first so that instruction fetch can be started early (which requires modification of PC) and the IPSR can be updated early. After stacking, SP will be updated to N-32 (0 × 20), and the stacked data arrangement in the stack memory will look like Table 9.1.

Table 9.1 Stack Memory Content After Stacking and Stacking Order

Address	Data	Push Order
Old SP (N) ->	(Previously pushed data)	–
(N-4)	PSR	2
(N-8)	PC	1
(N-12)	LR	8
(N-16)	R12	7
(N-20)	R3	6
(N-24)	R2	5
(N-28)	R1	4
New SP (N-32) ->	R0	3

The reason the registers R0–R3, R12, LR, PC, and PSR are stacked is that these are caller saved registers, according to C standards (C/C++ standard Procedure Call Standard for the ARM Architecture, AAPCS, Ref 5). This arrangement allows the interrupt handler to be a normal C function, because registers that could be changed by the exception handler are saved in the stack.

The general registers (R0–R3, R12) are located at the end of the stack frame so that they can be easily accessed using SP-related addressing. As a result, it's easy to pass parameters to software interrupts using stacked registers.

Vector Fetches

While the data bus is busy stacking the registers, the instruction bus carries out another important task of the interrupt sequence: It fetches the exception vector (the starting address of the exception handler) from the vector table. Since the stacking and vector fetch are performed on separate bus interfaces, they can be carried out at the same time.

Register Updates

After the stacking and vector fetch are completed, the exception vector will start to execute. On entry of the exception handler, a number of registers will be updated:

- SP: The Stack Pointer (either the MSP or the PSP) will be updated to the new location during stacking. During execution of the interrupt service routine, the MSP will be used if the stack is accessed.
- PSR: The IPSR (the lowest part of the PSR) will be updated to the new exception number.
- PC: This will change to the vector handler as the vector fetch completes and starts fetching instructions from the exception vector.
- LR: The LR will be updated to a special value called EXC_RETURN.[1] This special value drives the interrupt return operation. The last 4 bits of the LR have a special meaning, which is covered later in this chapter.

A number of other NVIC registers will also be updated. For example, the pending status of the exception will be cleared and the active bit of the exception will be set.

Exception Exits

At the end of the exception handler, an exception exit (known as an *interrupt return* in some processors) is required to restore the system status so that the interrupted program can resume normal execution. There are three ways to trigger the interrupt return sequence; all of them use the special value stored in the LR in the beginning of the handler (see Table 9.2).

Table 9.2 Instructions that Can be Used for Triggering Exception Return

Return Instruction	Description
BX <*reg*>	If the EXC_RETURN value is still in LR, we can use the *BX LR* instruction to perform the interrupt return.
POP {PC}, or POP {...., PC}	Very often the value of LR is pushed to the stack after entering the exception handler. We can use the POP instruction, either a single POP or multiple POPs, to put the EXC_RETURN value to the program counter. This will cause the processor to perform the interrupt return.
LDR, or LDM	It is possible to produce an interrupt return using the LDR instruction with PC as the destination register.

[1] EXC_RETURN has values with bit[31:4] and are all 1 (i.e., 0xFFFFFFFX); the last 4 bits define the return information. More information on the EXC_RETURN value is covered later in this chapter.

Some microprocessor architectures use special instructions for interrupt returns (for example, *reti* in 8051). In the Cortex-M3, a normal return instruction is used so that the whole interrupt handler can be implemented as a C subroutine.

When the interrupt return instruction is executed, the following processes are carried out:

1. Unstacking: The registers pushed to the stack will be restored. The order of the POP will be the same as in stacking. The stack pointer will also be changed back.
2. NVIC register update: The active bit of the exception will be cleared. For external interrupts, if the interrupt input is still asserted, the pending bit will be set again, causing it to reenter the interrupt handler.

Nested Interrupts

Nested interrupt support is built into the Cortex-M3 processor core and the NVIC. There is no need to use assembler wrapper code to enable nested interrupts. In fact, you do not have to do anything apart from setting up the appropriate priority level for each interrupt source. First, the NVIC in the Cortex-M3 processor sorts out the priority decoding for you. So, when the processor is handling an exception, all other exceptions with the same or lower priority will be blocked. Second, the automatic hardware stacking and unstacking allow the nested interrupt handler to execute without risk of losing data in registers.

However, one thing needs to be taken care of: Make sure that there is enough space in the main stack if lots of nested interrupts are allowed. Since each exception level will use eight words of stack space and the exception handler code might require extra stack space, it might end up using more stack memory than expected.

Reentrant exceptions are not allowed in the Cortex-M3. Since each exception has a priority level assigned and, during exception handling, exceptions with the same or lower priority will be blocked, the same exception cannot be carried out until the handler is ended. For this reason, SVC instructions cannot be used inside an SVC handler, since doing so will cause a fault exception.

Tail-Chaining Interrupts

The Cortex-M3 uses a number of methods to improve interrupt latency. The first one we'll look at is *tail chaining*.

When an exception takes place but the processor is handling another exception of the same or higher priority, the exception will be pended. When the processor has finished executing the current exception handler, instead of POP, the registers go back into the stack and PUSH it back in again, skipping the unstacking and the stacking. In this way the timing gap between the two exception handlers is greatly reduced.

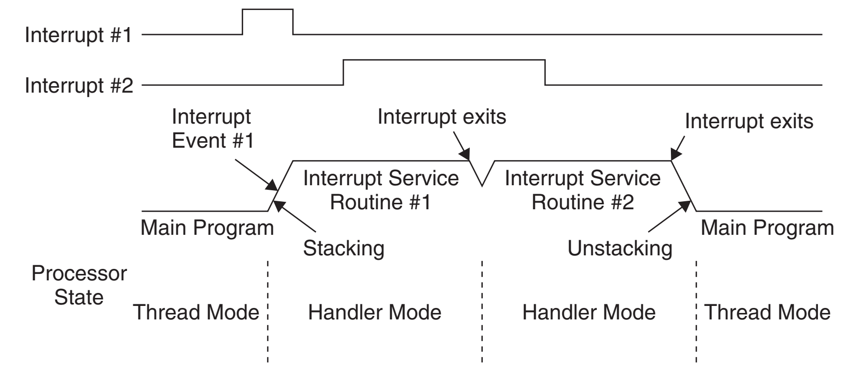


Figure 9.2 Tail Chaining of Exceptions

Late Arrivals

Another feature that improves interrupt performance is *late arrival* exception handling. When an exception takes place and the processor has started the stacking process, and if during this delay a new exception arrives with higher preemption priority, the late arrival exception will be processed first.

For example, if Exception #1 (lower priority) takes place a few cycles before Exception #2 (higher priority), the processor will behave as shown in Figure 9.3, such that Handler #2 is executed as soon as the stacking completes.

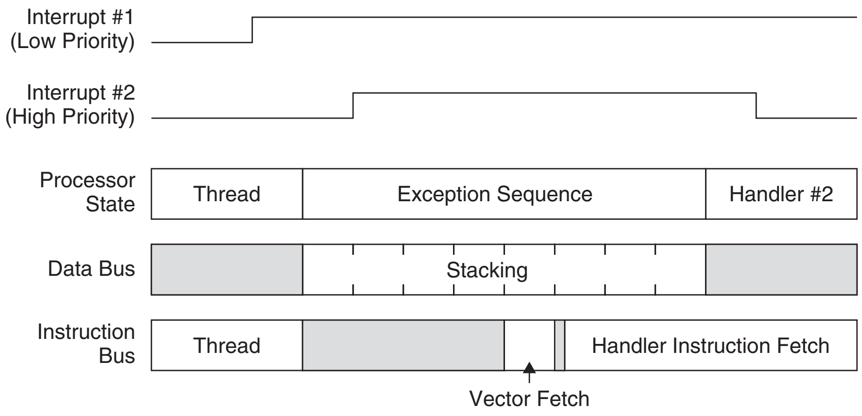


Figure 9.3 Late Arrival Exception Behavior

More on the Exception Return Value

When entering an exception handler, the LR is updated to a special value called EXC_RETURN, with the upper 28 bits all set to 1. This value, when loaded into the PC at the end of the exception handler execution, will cause the processor to perform an exception return sequence.

The instructions that can be used to generate exception returns are:

- POP/LDM
- LDR with PC as a destination
- BX with any register

The EXC_RETURN value has bit [31:4] all set to 1, and bit[3:0] provides information required by the exception return operation (see Table 9.3). When the exception handler is entered, the LR value is updated automatically, so there is no need to generate these values manually.

Table 9.3 Description of Bit Fields in EXC_RETURN Value

Bits	31:4	3	2	1	0
Descriptions	0xFFFFFFF	Return mode (Thread/handler)	Return stack	Reserved; must be 0	Process state (Thumb/ARM)

Bit 0 indicates the process state being used after the exception return. Since the Cortex-M3 supports only the Thumb state, bit 0 must be 1.

The valid values (for the Cortex-M3) are shown in Table 9.4.

Table 9.4 Allowed EXC_RETURN Values on Cortex-M3

Value	Condition
0xFFFFFFF1	Return to handler mode
0xFFFFFFF9	Return to Thread mode and on return use the main stack
0xFFFFFFFD	Return to Thread mode and on return use the process stack

If the thread is using the MSP (main stack), the value of LR will be set to 0xFFFFFFF9 when it enters an exception, and 0xFFFFFFF1 when a nested exception is entered, as shown in Figure 9.4. If the thread is using PSP (process stack), the value of LR would be 0xFFFFFFFD when entering the first exception and 0xFFFFFFF1 for entering a nested exception, as shown in Figure 9.5.

As a result of the EXC_RETURN number format, you cannot perform interrupt returns to an address in the 0xFFFFFFF0–0xFFFFFFFF memory range. However, since this address is in a nonexecutable region anyway, it is not a problem.

Interrupt Latency

The term *interrupt latency* refers to the delay from the start of the interrupt request to the start of interrupt handler execution. In the Cortex-M3 processor, if the memory system has zero

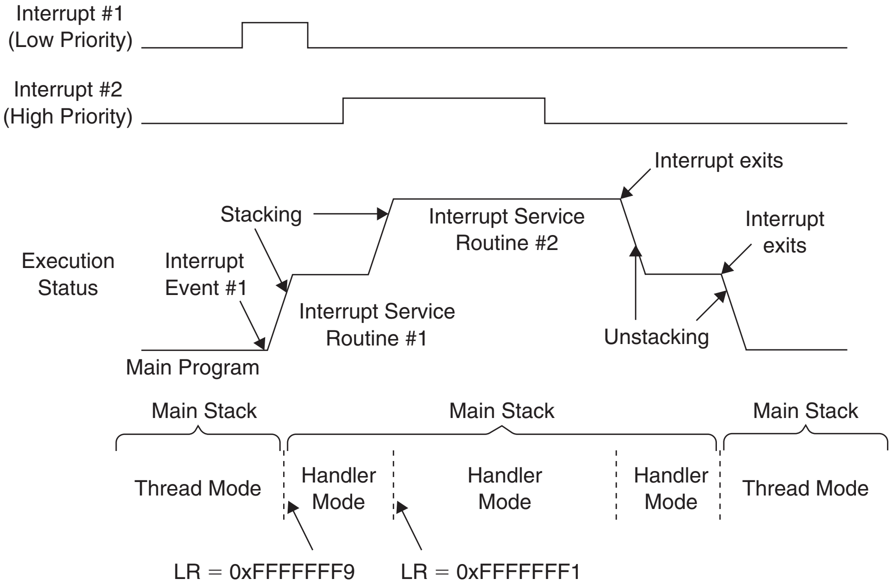


Figure 9.4 LR Set to EXC_RETURN at Exception (Main Stack Used in Thread Mode)

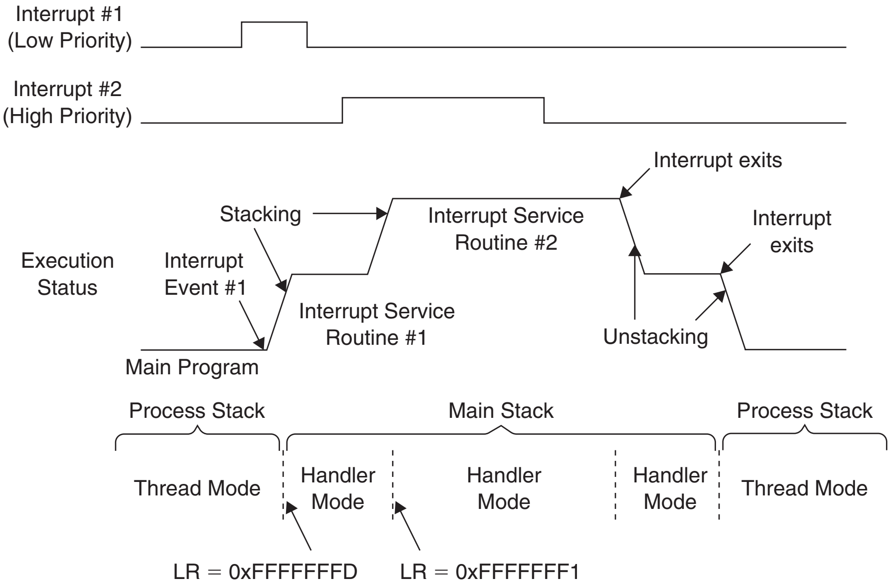


Figure 9.5 LR Set to EXC_RETURN at Exception (Process Stack Used in Thread Mode)

latency, and provided that the bus system design allows vector fetch and stacking to happen at the same time, the interrupt latency can be as low as 12 cycles. This includes stacking the registers, vector fetch, and fetching instructions for the interrupt handler. However, this depends on memory access wait states and a few other factors.

For tail-chaining interrupts, since there is no need to carry out stacking operations, the latency of switching from one exception handler to another exception handler can be as low as six cycles.

When the processor is executing a multicycle instruction such as divide, the instruction could be abandoned and restarted after the interrupt handler completes. This also applies to load double (LDRD) and store double (STRD) instructions.

To reduce exception latency, the Cortex-M3 processor allows exceptions in the middle of multiple load and store instructions (LDM/STM). If the LDM/STM instruction is executing, the current memory accesses will be completed, and the next register number will be saved in the stacked xPSR (ICI bits). After the exception handler completes, the multiple load/store will resume from the point at which the transfer stopped. There is a corner case: If the multiple load/store instruction being interrupted is part of an IF-THEN (IT) instruction block, the load/store instruction will be cancelled and restarted when the interrupt is completed. This is because the ICI bits and IT execution status bits share the same space in the EPSR.

In addition, if there is an outstanding transfer on the bus interface, such as a buffered write, the processor will wait until the transfer is completed. This is necessary to ensure that a bus fault handler preempts the correct process.

Of course, the interrupt could be blocked if the processor is already executing another exception handler of the same or higher priority or if the interrupt mask register was masking the interrupt request. In these cases, the interrupt will be pended and will not be processed until the blocking is removed.

Faults Related to Interrupts

Various faults can be caused by exception handling. Let’s take a look at these now.

Stacking

If a bus fault takes place during stacking, the stacking sequence will be terminated and the bus fault exception will be triggered or pended. If the bus fault is disabled, the hard fault hander will be executed. Otherwise, if the bus fault handler has higher priority than the original exception, the bus fault handler will be executed; if not, it will be pended until the original exception is completed. This scenario, called a *stacking error,* is indicated by the STKERR (bit 4) in the Bus Fault Status register (0xE000ED29).

If the stacking error is caused by an MPU violation, the memory management fault handler will be executed and the MSTKERR (bit 4) in the Memory Management Fault Status register (0xE000ED28) will be set to indicate the problem. If the memory management fault is disabled, the hard fault handler will be executed.

Unstacking

If a bus fault takes place during unstacking (an interrupt return), the unstacking sequence will be terminated and the bus fault exception will be triggered or pended. If the bus fault is disabled, the hard fault handler will be executed. Otherwise, if the bus fault handler has higher priority than the current priority of the executing task (the core could already be executing another exception in a nested interrupt case), the bus fault handler will be executed. This scenario, called an *unstacking error*, is indicated by the UNSTKERR (bit 3) in the Bus Fault Status register (0xE000ED29).

Similarly, if the stacking error is caused by an MPU violation, the memory management fault handler will be executed and the MUNSTKERR (bit 3) in the Memory Management Fault Status register (0xE000ED28) will be set to indicate the problem. If the memory management fault is disabled, the hard fault handler will be executed.

Vector Fetches

If a bus fault or memory management fault takes place during a vector fetch, the hard fault handler will be executed. This is indicated by VECTTBL (bit 1) in the Hard Fault Status register (0xE000ED2C).

Invalid Returns

If the EXC_RETURN number is invalid or does not match the state of the processor (as in using 0xFFFFFFF1 to return to Thread mode), it will trigger the usage fault. If the usage fault handler is not enabled, the hard fault handler will be executed instead. The INVPC bit (bit 2) or INVSTATE (bit 1) bit in the Usage Fault Status register (0xE000ED2A) will be set, depending on the actual cause of the fault.

This page intentionally left blank

CHAPTER 10

Cortex-M3 Programming

In This Chapter:

- Overview
- The Interface Between Assembly and C
- A Typical Development Flow
- The First Step
- Producing Outputs
- Using Data Memory
- Using Exclusive Access for Semaphores
- Using Bit Band for Semaphores
- Working with Bit Field Extract and Table Branch

Overview

The Cortex-M3 can be programmed using either assembler or C. There might be compilers for other languages, but most people will use assembler, C, or a combination of the two in their projects. Because a lot of information on the way to do programming depends on the tool chain and silicon chips you use, this book will not focus on the details of compiling a program or how to download the program to your circuit board. A little bit of this information is covered in Chapters 19 and 20.

Using Assembly

For small projects, it is possible to develop the whole application in assembly language. Using assembler, you might be able to get the best optimization you want. However, it might increase your development time, and it could be easy to make mistakes. In addition, handling complex data structures or function library management can be extremely difficult in assembler. Yet even when the C language is used in a project, in many situations part of the program is implemented in assembly language:

- Functions that cannot be implemented in C, such as special register accesses and exclusive accesses
- Timing-critical routines
- Tight memory requirements, causing part of the program to be written in assembly to get the smallest memory size

Using C

C has the advantage of being portable and easier for implementing complex operations, compared to assembly language. Since it's a generic computer language, C does not specify how the processor is initialized. For these areas, tool chains can have different approaches. The best way to get started is to look at example codes. For users of ARM C compiler products such as RealView Development Suite (RVDS) or KEIL RealView Microcontroller Development Kit, a number of Cortex-M3 program examples are already included in the installation. For users of the GNU tool chain, Chapter 19 of this book provides a simple C example based on the CodeSourcery GNU tool chain for ARM.

Use of the C language can often speed up application development, but in many cases low-level system control will still require assembly code. Most ARM C compilers allow you to include assembly code, called *inline assembler*. This code is often necessary for many projects.

In the ARM compiler, you can add assembly code inside the C program. Traditionally, inline assembler is used, but the inline assembler in RealView C Compiler does not support Thumb-2 instructions. Starting with RealView C Compiler version 3.0, a new feature called the Embedded Assembler is included, and it supports Thumb-2 instructions. For example, you can insert assembly functions in your C programs this way:

```
__asm void SetFaultMask(unsigned int new_value)
{
  // Assembly code here
  MSR FAULTMASK, new_value // Write new value to FAULTMASK
  BX LR                    // Return to calling program
}
```

Detailed descriptions of Embedded Assembler in RealView C Compiler can be found in the *RVCT 3.0 Compiler and Library Guide* (Ref 6).

For the Cortex-M3, Embedded Assembler is useful for tasks such as accessing special registers (MRS and MSR instructions; for example, setting up stack memory) or when it is necessary to use instructions that cannot be generated using C (for example, sleep [WFI and WFE], exclusive accesses, and memory barrier operations).

Previously in ARM processors, because there is a Thumb state and an ARM state, the code for different states has to be compiled differently. In the Cortex-M3 there is no such need; because everything is in the Thumb state, project file management is much simpler.

When you're developing applications in C, it is recommended that you use the double word stack alignment function (configured by the STKALIGN bit in the NVIC Configuration Control register). This can be set in the startup code. For example:

```
#define NVIC_CCR ((volatile unsigned long *)(0xE000ED14))
*NVIC_CCR = *NVIC_CCR | 0x200; /* Set STKALIGN */
```

Using this feature ensures that the system conforms to Procedure Call Standards for the ARM Architecture (AAPCS). Additional information on this subject is covered in Chapter 12.

The Interface Between Assembly and C

In various situations, assembly code and the C program interact. For example:

- When embedded assembly (or inline assembler, in the case of the GNU tool chain) is used in C program code
- When C program code calls a function or subroutine implemented in assembler in a separate file
- When an assembly program calls a C function or subroutine

In these cases, it is important to understand how parameters and return results are passed between the calling program and the function being called. The mechanisms of these interactions are specified in the ARM Architecture Procedure Call Standard (AAPCS, Ref 5).

For simple cases, when a calling program needs to pass parameters to a subroutine or function, it will use registers R0 to R3, where R0 is the first parameter, R1 is the second, and so on. Similarly, R0 is used for returning a value at the end of a function. R0–R3 and R12 can be changed by a function or subroutine, whereas the contents of R4–R11 should be restored to the previous state before entering the function, usually handled by stack PUSH and stack POP.

To make them easier to understand, the examples in this book do not strictly follow AAPCS practices. If a C function is called by an assembly code, the effect of a possible register change to R0–R3 and R12 will need to be taken into account. If the contents of these registers are needed at a later stage, these registers might need to be saved on the stack and restored after the C function completes. Since the example codes mostly only call assembly functions or subroutines that affect a few registers or restore the register contents at the end, it's not necessary to save registers R0–R3 and R12.

A Typical Development Flow

Various software programs are available for developing Cortex-M3 applications. The concepts of code generation flow in terms of these tools are similar. For the most basic uses, you will need assembler, a C compiler, a linker, and binary file generation utilities. For ARM solutions, the RealView Development Suite (RVDS) or RealView Compiler Tools (RVCT) provide a file generation flow, as shown in Figure 10.1. The scatter-loading script is optional but often required when the memory map becomes more complex.

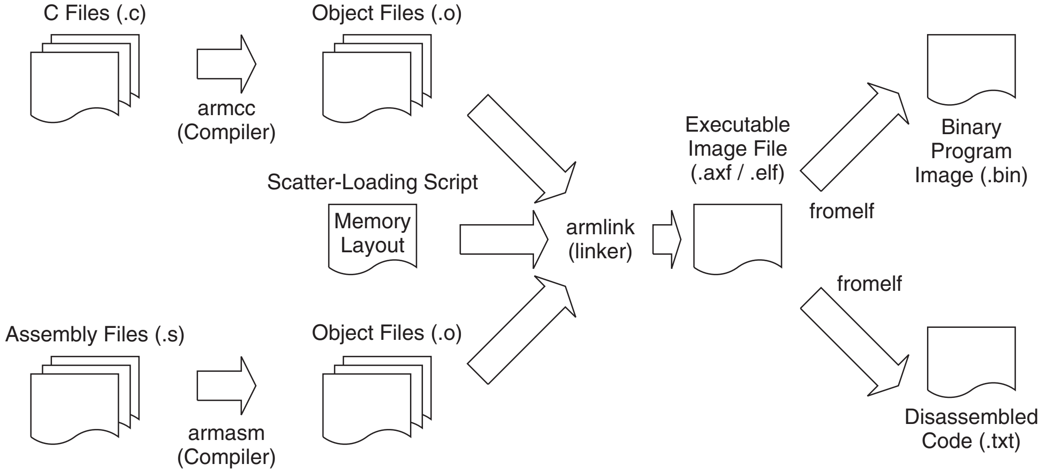


Figure 10.1 Example Flow Using ARM Development Tools

Aside from these basic tools, RVDS also contains a large number of utilities, including an Integrated Development Environment (IDE) and debuggers. Please visit the ARM Web site (www.arm.com) for details.

The First Step

This chapter reviews a few examples in assembly language. In most cases you will be programming in C, but by looking into some assembler examples, we can gain a better understanding of how to use the Cortex-M3 processor. The examples here are based on ARM assembler tools (armasm). For other assembler tools, the file format and instruction syntax might need to be modified. In addition, some development tools will actually do the startup code for you, so you might not need to worry about creating your assembly startup code.

The first simple program can be something like this:

```
STACK_TOP   EQU   0x20002000        ; constant for SP starting value

            AREA  |Header Code|, CODE
            DCD    STACK_TOP  ; Stack top
            DCD    Start      ; Reset vector
            ENTRY             ; Indicate program execution start here
Start       ; Start of main program
            ; initialize registers
            MOV   r0,  #10    ; Starting loop counter value
            MOV   r1,  #0     ; starting result
            ; Calculated 10+9+8+...+1
loop
            ADD   r1,  r0     ; R1=R1 + R0
            SUBS  r0,  #1     ; Decrement R0, update flag ("S" suffix)
            BNE   loop        ; If result not zero jump to loop
            ; Result is now in R1
deadloop
            B     deadloop    ; Infinite loop
            END               ; End of file
```

This simple program contains the initial SP value, the initial PC value, and setup registers and then does the required calculation in a loop.

Assuming you are using ARM tools, this program can be assembled using:

```
$> armasm --cpu cortex-m3 -o test1.o test1.s
```

The *-o* option specifies the output filename. The test1.o is an object file. We then need to use a linker to create an executable image (ELF). This can be done by:

```
$> armlink --rw_base 0x20000000 --ro_base 0x0 --map -o test1.elf
  test1.o
```

Here, *—ro-base 0x0* specifies that the read-only region (program ROM) starts at address 0x0; *—rw-base* specifies that the read/write region (data memory) starts at address 0x20000000. (In this example test1.s, we did not have any RAM data defined.) The *--map* option creates an image map, which is useful for understanding the memory layout of the compiled image.

Finally, we need to create the binary image:

```
$> fromelf --bin --output test1.bin test1.elf
```

For checking that the image looks like what we wanted, we can also generate a disassembled code list file by:

```
$> fromelf -c --output test1.list test1.elf
```

If everything works fine, you can then load your ELF image or binary image into your hardware or instruction set simulator for testing.

Producing Outputs

It is always more fun when you can connect your microcontroller to the outside world. The simplest way to do that is to turn on/off the LEDs. However, this practice is quite limiting because it can only represent very limited information. One of the most common output methods is to send text messages to a console. In embedded product development, this task is often handled by a UART interface connecting to a personal computer. For example, a computer running a Windows[1] system with the Hyper-Terminal program acting as a console can be a handy way to produce outputs.

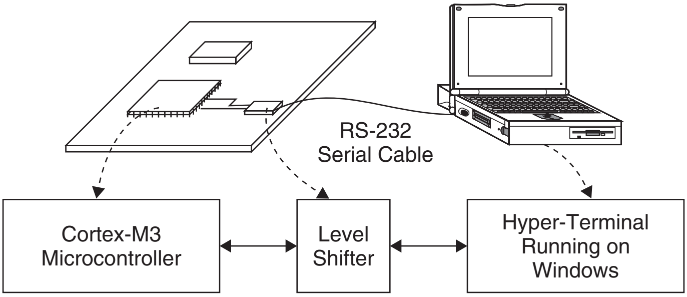


Figure 10.2 A Low-Cost Test Environment for Outputting Text Messages

The Cortex-M3 processor does not contain a UART interface, but most Cortex-M3 microcontrollers come with UART provided by the chip manufacturers. The specification of the UART can differ among various devices, so we won't attempt to cover the topic in this book. Our next example assumes that a UART is available and has a status flag to indicate whether the transmit buffer is ready for sending out new data. A level shifter is needed in the connection because RS-232 has a different voltage level than the microcontroller I/O pins.

UART is not the only solution to output text messages. A number of features are implemented on the Cortex-M3 processor to help output debugging messages:

- Semihosting: Depending on the debugger and code library support, *semihosting* (outputting *printf* messages via a debug probe device) can be done via debug register in the NVIC. (More information on this topic is covered in Chapter 15.) In these cases, you can use *printf* within your C program and the output will be displayed on the console/standard output (STDOUT) of the debugger software.

[1] Windows and Hyper-Terminal are trademarks of Microsoft Corporation.

- Instrumentation trace: If the Cortex-M3 microcontroller provides a trace port and an external Trace Port Analyzer (TPA) is available, instead of using UART to output messages, we can use the Instrumentation Trace Module (ITM). The trace port works much faster than UART and can offer more data channels.
- Instrumentation trace via Serial Wire Viewer: Alternatively, the Cortex-M3 processor (revision 1 and later) also provides a Serial Wire Viewer (SWV) operation mode on the Trace Port Interface Unit (TPIU). This interface allows outputs from ITM to be captured using low-cost hardware instead of a TPA. However, the bandwidth provided with the SWV mode is limited, so it is not ideal for large amounts of data.

The "Hello World" Example

Before we try to write a "Hello world" program, we should figure out how to send one character through the UART. The code used to send a character can be implemented as a subroutine, which can be called by other message output codes. If the output device changes, we only need to change this subroutine and all the text messages can be output by a different device. This modification is usually called *retargeting*.

A simple routine to output a character could be something like this:

```
UART0_BASE     EQU     0x4000C000
UART0_FLAG     EQU     UART0_BASE+0x018
UART0_DATA     EQU     UART0_BASE+0x000

Putc           ; Subroutine to send a character via UART
               ; Input R0 = character to send
               PUSH  {R1,R2, LR}     ; Save registers
               LDR   R1,=UART0_FLAG
PutcWaitLoop
               LDR   R2,[R1]         ; Get status flag
               TST   R2, #0x20       ; Check transmit buffer full flag
                                     ; bit
               BNE   PutcWaitLoop    ; If busy then loop
               LDR   R1,=UART0_DATA ; otherwise
               STRB  R0, [R1]        ; Output data to transmit buffer
               POP   {R1,R2, PC}     ; Return
```

The register addresses and bit definitions here are just examples; you might need to change the value for your device. In addition, some UART might require a more complex status-checking process before the character is output to the transmit buffer. Furthermore, another subroutine call (*Uart0Initialize* in the following example) is required to initialize the UART, but this depends on the UART specification and will not be covered here. (An example of UART initialization in C for Luminary Micro LM3S811 devices is covered in Chapter 20.)

Now we can use this subroutine to build a number of functions to display messages:

```
Puts         ; Subroutine to send string to UART
             ; Input R0 = starting address of string.
             ; The string should be null terminated
             PUSH {R0 ,R1, LR} ; Save registers
             MOV R1, R0        ; Copy address to R1, because R0 will
                               ; be used
PutsLoop                      ; as input for Putc
             LDRB  R0,[R1],#1  ; Read one character and increment
                               ; address
             CBZ   R0, PutsLoopExit ; if character is null, goto end
             BL    Putc         ; Output character to UART
             B     PutsLoop     ; Next character
PutsLoopExit
             POP   {R0, R1, PC} ; Return
```

With this subroutine, we are ready for our first "Hello world" program:

```
STACK_TOP    EQU   0x20002000       ; constant for SP starting value
UART0_BASE   EQU   0x4000C000
UART0_FLAG   EQU   UART0_BASE+0x018
UART0_DATA   EQU   UART0_BASE+0x000
             AREA  | Header Code|, CODE
             DCD   STACK_TOP ; Stack Pointer initial value
             DCD   Start ; Reset vector
             ENTRY
Start        ; Start of main program
             MOV   r0, #0       ; initialize registers
             MOV   r1, #0
             MOV   r2, #0
             MOV   r3, #0
             MOV   r4, #0
             BL    Uart0Initialize ; Initialize the UART0
             LDR   r0,=HELLO_TXT ; Set R0 to starting address of string
             BL    Puts
deadend
             B     deadend            ; Infinite loop
             ;-------------------------------
             ; subroutines
             ;-------------------------------
Puts         ; Subroutine to send string to UART
             ;Input R0 = starting address of string.
             ; The string should be null terminated
             PUSH {R0 ,R1, LR} ; Save registers
             MOV R1, R0        ; Copy address to R1, because R0 will
                               ; be used
```

```
PutsLoop                      ; as input for Putc
            LDRB   R0,[R1],#1 ; Read one character and increment
                              ; address
            CBZ  R0, PutsLoopExit ; if character is null, goto end
            BL   Putc         ; Output character to UART
            B    PutsLoop     ; Next character
PutsLoopExit
            POP  {R0, R1, PC} ; Return
            ;--------------------------------
Putc        ; Subroutine to send a character via UART
            ; Input R0 = character to send
            PUSH  {R1,R2, LR}    ; Save registers
            LDR   R1,=UART0_FLAG
PutcWaitLoop
            LDR   R2,[R1]        ; Get status flag
            TST   R2, #0x20      ; Check transmit buffer full flag bit
            BNE   PutcWaitLoop   ; If busy then loop
            LDR   R1,=UART0_DATA ; otherwise
            STR   R0, [R1]       ; Output data to transmit buffer
            POP   {R1,R2, PC}    ; Return
            ;--------------------------------
Uart0Initialize
            ; Device specific, not shown here
            BX    LR    ; Return
            ;--------------------------------
HELLO_TXT
            DCB   “Hello world\n”,0          ; Null terminated Hello
                                             ; world string
            END                 ; End of file
```

The only thing you need to add to this code is the details for the *Uart0Initialize* subroutine.

It will also be useful to have subroutines that output register values as well. To make things easier, they can all be based on *Putc* and *Puts* subroutines we have already done. The first subroutine is to display hexadecimal values:

```
PutHex  ; Output register value in hexadecimal format
        ; Input R0 = value to be displayed
        PUSH  {R0-R3,LR}
        MOV   R3, R0   ; Save register value to R3 because R0 is used
                       ; for passing input parameter
         MOV   R0,#‘0’ ; Starting the display with “0x”
         BL    Putc
         MOV   R0,#‘x’
         BL    Putc
         MOV   R1, #8  ; Set loop counter
         MOV   R2, #28 ; Rotate offset
```

```
PutHexLoop
           ROR    R3, R2       ; Rotate data value left by 4 bits
                               ; (right 28)
           AND    R0, R3,#0xF  ; Extract the lowest 4 bit
           CMP    R0, #0xA     ; Convert to ASCII
           ITE    GE
           ADDGE  R0, #55      ; If larger or equal 10, then convert
                               ; to A-F
           ADDLT  R0, #48      ; otherwise convert to 0-9
           BL     Putc         ; Output 1 hex character
           SUBS   R1, #1       ; decrement loop counter
           BNE    PutHexLoop   ; if all 8 hexadecimal character been
                               ; display then
           POP    {R0-R3,PC}   ; return, otherwise process next 4-bit
```

This subroutine is useful for outputting register values. However, sometimes we also want to output register values in decimal. This sounds like a rather complex operation, but in the Cortex-M3 it is easy because of the hardware multiply and divide instructions. One of the other main problems is that during calculation, we will get output characters in reverse order, so we need to put the output results in a text buffer first, wait until the whole text is ready to display, and then use the *Puts* function to display the whole result. In this example, a part of the stack memory is used as the text buffer:

```
PutDec     ; Subroutine to display register value in decimal
            ; Input R0 = value to be displayed.
            ; Since it is 32 bit, the maximum number of character
            ; in decimal format, including null termination is 11
            PUSH   {R0-R5, LR}     ; Save register values
            MOV    R3, SP          ; Copy current Stack Pointer to R3
            SUB    SP, SP, #12     ; Reserved 12 bytes as text buffer
            MOV    R1, #0          ; Null character
            STRB   R1,[R3, #-1]!   ; Put null character at end of text
                                   ; buffer,pre-indexed
            MOV    R5, #10         ; Set divide value
PutDecLoop
            UDIV   R4, R0, R5   ; R4 = R0 / 10
            MUL    R1, R4, R5   ; R1 = R4 * 10
            SUB    R2, R0, R1   ; R2 = R0 - (R4 * 10) + remainder
            ADD    R2, #48      ; convert to ASCII (R2 can only be 0-9)
            STRB   R2,[R3, #-1]! ; Put ascii character in text
                                 ; buffer, pre-indexed
            MOVS   R0, R4       ; Set R0 = Divide result and set Z flag
                                ; if R4=0
            BNE    PutDecLoop   ; If R0(R4) is already 0, then there
                                ; is no more digit
            MOV    R0, R3      ; Put R0 to starting location of text
                               ; buffer
            BL     Puts        ; Display the result using Puts
            ADD    SP, SP, #12 ; Restore stack location
            POP    {R0-R5, PC} ; Return
```

With various features in the Cortex-M3 instruction set, the processing to convert values into decimal format display can be implemented in a very short subroutine.

Using Data Memory

Back to our first example: When we were doing the linking stage, we specified the read/write memory region. How do we put data there? The method is to define a data region in your assembly file. Using the same example from the beginning, we can store the data in the data memory at 0x20000000 (the SRAM region). The location of the data section is controlled by a command-line option when you run the linker:

```
STACK_TOP   EQU   0x20002000    ; constant for SP starting value
            AREA  | Header Code|, CODE
            DCD   STACK_TOP     ; SP initial value
            DCD   Start         ; Reset vector
            ENTRY
Start       ; Start of main program
            ; initialize registers
            MOV   r0, #10       ; Starting loop counter value
            MOV   r1, #0        ; starting result
            ; Calculated 10+9+8+...+1
loop
            ADD   r1, r0        ; R1 = R1 + R0
            SUBS  r0, #1        ; Decrement R0, update flag ("S" suffix)
            BNE   loop          ; If result not zero jump to loop
            ; Result is now in R1
            LDR   r0,=MyData1   ; Put address of MyData1 into R0
            STR   r1,[r0]       ; Store the result in MyData1
deadloop
            B     deadloop      ; Infinite loop

            AREA  | Header Data|, DATA
            ALIGN 4
MyData1     DCD   0             ; Destination of calculation result
MyData2     DCD   0
            END                 ; End of file
```

During the linking stage, the linker will put the DATA region into read/write memory, so the address for *MyData1* will be 0x20000000 in this case.

Using Exclusive Access for Semaphores

Exclusive access instructions are used for semaphore operations—for example, to make sure that a resource is used by only one task. For instance, let's say that a data variable *DeviceALocked* in memory can be used to indicate that Device A is being used. If a task wants to use Device A, it should check the status by reading the variable *DeviceALocked*. If it is zero, it can write a 1 to *DeviceALocked* to lock the device. After it's finished using the device, it can then clear the *DeviceALocked* to zero so that other tasks can use it.

What will happen if two tasks try to access Device A at the same time? In that case, possibly both tasks will read the variable *DeviceALocked*, and both will get zero. Then both of them will try writing back 1 to the variable *DeviceALocked* to lock the device, and we'll end up with both tasks believing that they have exclusive access to Device A. That is where exclusive accesses are used. The STREX instruction has a return status, which indicates whether the exclusive store has been successful. If two tasks try to lock a device at the same time, the return status will be 1 (exclusive failed) and the task can then know that it needs to retry the lock.

Chapter 5 provided some background on the use of exclusive accesses. The flowchart in that discussion is shown in Figure 10.3.

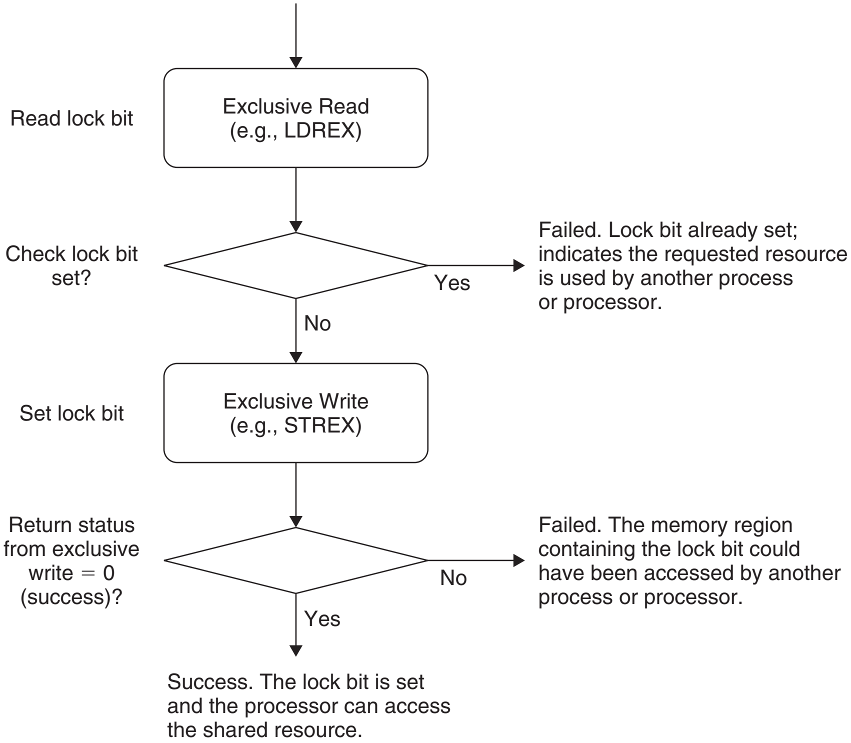

Read lock bit
Exclusive Read
(e.g., LDREX)
Check lock bit set?
Yes
Failed. Lock bit already set; indicates the requested resource is used by another process or processor.
No
Set lock bit
Exclusive Write
(e.g., STREX)
Return status from exclusive write = 0 (success)?
No
Failed. The memory region containing the lock bit could have been accessed by another process or processor.
Yes
Success. The lock bit is set and the processor can access the shared resource.

Figure 10.3 Using Exclusive Access for Semaphore Operations

The operation can be carried out by the following assembly code. Note that the data write operation of STREX will not be carried out if the exclusive monitor returns a fail status, preventing a lock bit being set when the exclusive access fails:

```
LockDeviceA
            ; A simple function to try to lock Device A
            ; Output R0 : 0 = Success, 1 = failed
            ; If successful, value of 1 will be written to variable
            ; DeviceALocked
            PUSH  {R1, R2, LR}
TryToLockDeviceA
            LDR      R1,=DeviceALocked     ; Get the lock status
            LDREX    R2,[R1]
            CMP      R2,#0                 ; Check if it is locked
            BNE      LockDeviceAFailed
DeviceAIsNotLocked
            MOV      R0,#1                 ; Try to write 1 to
                                           ; DeviceALocked
            STREX    R2,R0,[R1]            ; Exclusive write
            CMP      R2, #0
            BNE      LockDeviceAFailed     ; STREX Failed
LockDeviceASucceed
            MOV      R0,#0                 ; Return success status
            POP      {R1, R2, PC}          ; Return
LockDeviceAFailed
           MOV       R0,#1                 ; Return fail status
           POP       {R1, R2, PC}          ; Return
```

If the return status of this function is 1 (exclusive failed), the application tasks should wait a bit and retry later. In single-processor systems, the common cause of an exclusive access failing is an interrupt occurring between the exclusive load and the exclusive store. If the code is run in privileged mode, this situation can be prevented by setting an interrupt mask register such as PRIMASK for a short time to increase the chance of getting the resource locked successfully.

In multiprocessor systems, aside from interrupts, the exclusive store could also fail if another processor has accessed the same memory region. To detect memory accesses from different processors, the bus infrastructure requires exclusive access monitor hardware to detect whether there is an access from a different bus master to a memory between the two exclusive accesses. However, in most low-cost Cortex-M3 microcontrollers, there is only one processor, so this monitor hardware is not required.

With this mechanism, we can be sure that only one task can have access to certain resources. If the application cannot gain the lock to the resource after a number of times, it might need to quit with a timeout error. For example, a task that locked a resource might have crashed and the lock remained set. In these situations, the OS should check which task is using the resource. If the task has completed or terminated without clearing the lock, the OS might need to unlock the resource.

If the process has started an exclusive access using LDREX and then found that the exclusive access is no longer needed, it can use the CLREX instruction to clear the local record in the exclusive access monitor. The syntax is:

```
CLREX.W
```

For the Cortex-M3 processor, all exclusive memory transfers must carry out sequentially. However, if the exclusive access control code has to be reused on other ARM Cortex processors, the Data Memory Barrier (DMB) instruction might need to be inserted between exclusive transfers to ensure correct ordering of the memory accesses.

Using Bit-Band for Semaphores

It is possible to use the bit-band feature to carry semaphore operations, provided that the memory system supports locked transfers or only one bus master is present on the memory

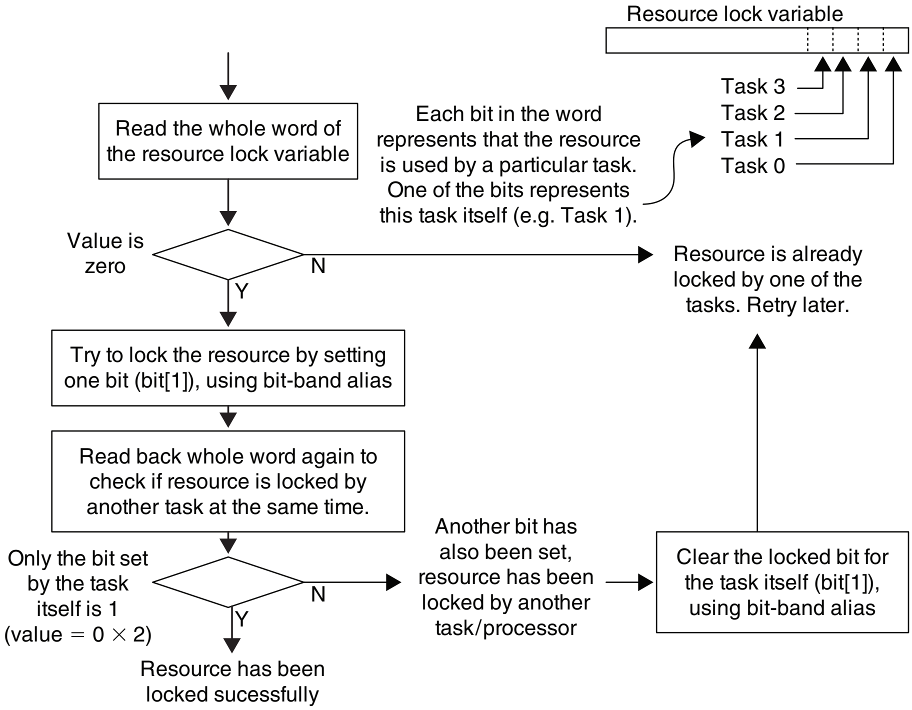

Resource lock variable
Task 3
Task 2
Task 1
Task 0
Read the whole word of the resource lock variable
Each bit in the word represents that the resource is used by a particular task. One of the bits represents this task itself (e.g. Task 1).
Value is zero
N
Y
Resource is already locked by one of the tasks. Retry later.
Try to lock the resource by setting one bit (bit[1]), using bit-band alias
Read back whole word again to check if resource is locked by another task at the same time.
Only the bit set by the task itself is 1 (value = 0 × 2)
N
Y
Another bit has also been set, resource has been locked by another task/processor
Clear the locked bit for the task itself (bit[1]), using bit-band alias
Resource has been locked sucessfully

Figure 10.4 Using Bit-Band as a Semaphore Control

bus. With bit band, it is possible to carry out the semaphore in C code, but the operation is different from using exclusive access. To use bit band as a resource allocation control, a memory location (such as word data) with a bit-band memory region is used, and each bit of this variable indicates that the resource is used by a certain task.

Since the bit-band alias writes are locked READ-MODIFY-WRITE transfers (the bus master cannot be switched to another one between the transfers), provided that all tasks only change the lock bit representing themselves, the lock bits of other tasks will not be lost, even if two tasks try to write to the same memory location at the same time. Unlike using exclusive accesses, it is possible for a resource to be "locked" simultaneously by two tasks for a short period of time until one of them detects the conflict and releases the lock.

Using bit band for semaphores can work only if all the tasks in the system change only the lock bit they are assigned to using the bit-band alias. If any of the tasks change the lock variable using a normal write, the semaphore can fail because, if another task sets a lock bit just before the write to the lock variable, the previous lock bit set by the other task will be lost.

Working with Bit Field Extract and Table Branch

We examined the Unsigned Bit Field Extract (UBFX) and Table Branch (TBB/TBH) instructions in Chapter 4. These two instructions can work together to form a very powerful branching tree. This capability is very useful in data communication applications where the data sequence can have different meanings with different headers. For example, let's say that the following decision tree based on Input A is to be coded in assembler (see Figure 10.5):

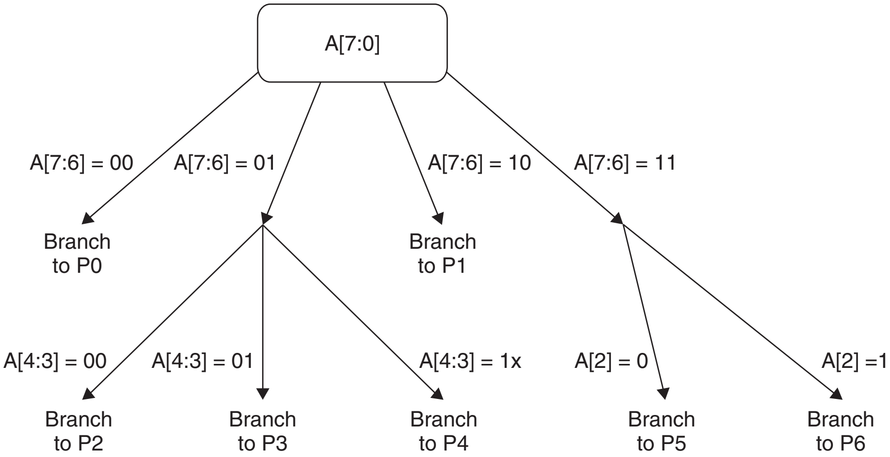


Figure 10.5 Bit Field Decoder: Example use of Bit Field Extract (UBFX) and Table Branch (TBB) Instructions

```
DecodeA
    LDR    R0,=A            ; Get the value of A from memory
    LDR    R0,[R0]
    UBFX   R1, R0, #6, #2   ; Extract bit[7:6] into R1
    TBB    [PC, R1]
BrTable1
    DCB   ((P0      -BrTable1)/2) ; Branch to P0       if A[7:6] = 00
    DCB   ((DecodeA1-BrTable1)/2) ; Branch to DecodeA1 if A[7:6] = 01
    DCB   ((P1      -BrTable1)/2) ; Branch to P1       if A[7:6] = 10
    DCB   ((DecodeA2-BrTable1)/2) ; Branch to DecodeA1 if A[7:6] = 11
DecodeA1
    UBFX   R1, R0, #3, #2 ; Extract bit[4:3] into R1
    TBB    [PC, R1]
BrTable2
    DCB    ((P2     -BrTable2)/2) ; Branch to P2    if A[4:3] = 00
    DCB    ((P3     -BrTable2)/2) ; Branch to P3    if A[4:3] = 01
    DCB    ((P4     -BrTable2)/2) ; Branch to P4    if A[4:3] = 10
    DCB    ((P4     -BrTable2)/2) ; Branch to P4    if A[4:3] = 11
DecodeA2
    TST    R0, #4 ; Only 1 bit is tested, so no need to use UBFX
    BEQ    P5
    B      P6
P0  ...    ; Process 0
P1  ...    ; Process 1
P2  ...    ; Process 2
P3  ...    ; Process 3
P4  ...    ; Process 4
P5  ...    ; Process 5
P6  ...    ; Process 6
```

This code completes the decision tree in a short assembler code sequence. If the branch target is larger, the instruction TBH will have to be used instead of TBB.

CHAPTER 11

Exceptions Programming

In This Chapter:

- Using Interrupts
- Exception/Interrupt Handlers
- Software Interrupts
- Example with Exception Handlers
- Using SVC
- SVC Example: Use for Output Functions
- Using SVC with C

Using Interrupts

Interrupts are used in almost all embedded applications. In the Cortex-M3 processor, the interrupt controller NVIC handles a number of processing tasks for you, including priority checking and stacking/unstacking of registers. However, a number of tasks have to be prepared when an interrupt is used:

- Stack setup
- Vector table setup
- Interrupt priority setup
- Enable the interrupt

Stack Setup

For simple application development, you can use the MSP for the whole program. That way you need to reserve memory that's just large enough and set the MSP to the top of the stack. When determining the stack size required, besides checking the stack level that could be used by the software, you also need to check how many levels of nested interrupts can occur.

For each level of nested interrupts, you need at least eight words of stack. The processing inside interrupt handlers might need extra stack space as well.

Since the stack operation in the Cortex-M3 is full descending, it is common to put the stack initial value at the end of the static memory so that the free space in the SRAM is not fragmented.

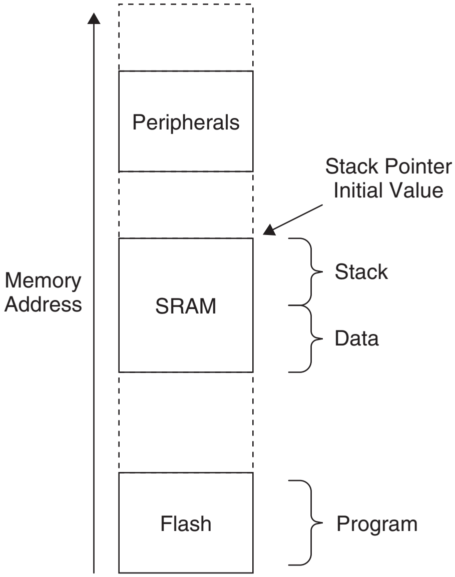


Figure 11.1 A Simple Memory Usage Example

For applications that use separate stacks for user code and kernel code, the main stack should have enough memory for the nested interrupt handlers as well as the stack memory used by the kernel code. The process stack should have enough memory for the user application code plus one level of stacking space (eight words). This is because stacking from the user thread to the first level of the interrupt handler uses the process stack.

Vector Table Setup

For simple applications that have fixed interrupt handlers, the vector table can be coded in ROM. In this case there is no need to set up the vector table during run time. However, in many applications, it is necessary to change the interrupt handlers for different situations. Then you will need to relocate the vector table to a writable memory.

Before the vector table is relocated, you might need to copy the existing vector table content to the new vector table location. This includes vector addresses for fault handlers, the NMI, system calls, and so on. Otherwise, invalid vector addresses will be fetched by the processor if these exceptions take place after the vector table relocation.

After the necessary vector table items are set up and the vector table is relocated, we can add new vectors to the vector table. For example:

```
      ; Subroutine for setting vector of an exception based on
      ; exception type
      ; (For IRQs add 16 : IRQ #0 = exception type 16)
SetVector
      ; Input R0 = exception type
      ; Input R1 = vector address value
      PUSH {R2, LR}
      LDR  R2,=0xE000ED08       ; Vector table offset register
      LDR  R2, [R2]
      STR  R1, [R2, R0, LSL #2] ; Write vector to VectTblOffset+
                                ; ExcpType*4
      POP  {R2, PC}             ; Return
```

Interrupt Priority Setup

By default, after a reset all exceptions with programmable priority are in priority level 0. For hard fault exceptions and NMI, the priority levels are −1 and −2, respectively. To program priority-level registers, we can take advantage of the fact that the registers are byte addressable, making the coding easier. For example:

```
      ; Setting IRQ #4 priority to 0xC0
      LDR  R0, =0xE000E400 ; External Interrupt Priority Reg starting
                           ; address
      LDR  R1, =0xC0       ; Priority level
      STRB R1, [R0, #4]    ; Set IRQ #4 priority (Byte write)
```

In the Cortex-M3, the width of interrupt priority configuration registers is specified by chip manufacturers. The minimum width is 3 bits and the maximum is 8 bits. You can determine the implemented width by writing 0xFF to one of the priority configuration registers and reading it back. For example:

```
      ; Determine the implemented priority width
      LDR    R0,=0xE000E400 ; Priority Configuration register for
                            ; external interrupt #0
      LDR    R1,=0xFF
      STRB   R1,[R0]        ; Write 0xFF (note : byte size write)
      LDRB   R1,[R0]        ; Read back (e.g. 0xE0 for 3-bits)
      RBIT   R2, R1         ; Bit reverse R2 (e.g. 0x07000000 for
                            ; 3-bits)
      CLZ    R1, R2         ; Count leading zeros (e.g. 0x5 for 3-bits)
      MOV    R2, #8
      SUB    R2, R2, R1   ; Get implemented width of priority
                          ; (e.g. 8-5=3 for 3-bits)
      MOV    R1, #0x0
      STRB   R1,[R0]        ; Restore to reset value (0x0)
```

If your application needs to be portable, it is best to use priority levels 0x00, 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0, and 0xE0 only. This is because all Cortex-M3 devices should have these priority levels.

Do not forget to set up the priority for system exceptions and fault handler exceptions as well. If it is necessary for some of the important interrupts to have higher priority than other system exceptions or fault handlers, you'll need to reduce the priority level of these system exceptions and fault handlers so that the important interrupts can preempt these handlers.

Enable the Interrupt

After the vector table and interrupt priority are set up, it's time to enable the interrupt. However, two steps might be required before you actually enable the interrupt:

1. If the vector table is located in a memory region that is write buffered, a Data Synchronization Barrier (DSB) instruction might be needed to ensure that the vector table memory is updated. In most cases the memory write should be completed within a few clock cycles. However, if your software needs to be portable between different Cortex-M3 products, this step ensures that the core will get the updated vector if the interrupt takes place immediately after being enabled.
2. An interrupt might already be pended or asserted beforehand, so it might be needed to clear the pending status. For example, signal glitches during power-up might have accidentally triggered some interrupt generation logic. In addition, in some peripherals such as UART, noise from the UART receiver before connection might be mistaken as data and can cause an interrupt to be pended. Therefore, it might be necessary to check and clear the pending status of an interrupt before enabling it.

Inside the NVIC, two separate register addresses are used for enabling and disabling interrupts. This duality ensures that each interrupt can be enabled or disabled without affecting or losing the other interrupt enable status. Otherwise, through software-based READ-MODIFY-WRITE, changes in enable register status carried out by interrupt handlers could be lost. To set an enable, the software needs to compute the correct bit location in the SETEN registers in the NVIC and write a 1 to it. Similarly, to clear an interrupt, the software needs to write a 1 to the corresponding bit in the CLREN registers:

```
        ; A subroutine to enable an IRQ based on IRQ number
EnableIRQ
        ; Input R0 = IRQ number
        PUSH   {R0-R2, LR}
        AND.W  R1, R0, #0x1F  ; Generate enable bit pattern for
                              ; the IRQ
        MOV    R2, #1
        LSL    R2, R2, R1      ; Bit pattern = (0x1 << (N & 0x1F))
        AND.W  R1, R0, #0xE0   ; Generate address offset if IRQ number
                               ; is above 31
        LSR    R1, R1, #3      ; Address offset = (N/32)*4 (Each word
                               ; has 32 IRQ enable)
        LDR    R0,=0xE000E100  ; SETEN register for external interrupt
                               ; #31-#0
        STR    R2, [R0, R1]    ; Write bit pattern to SETEN register
        POP    {R0-R2, PC}     ; Restore registers and Return
```

Likewise, we can write another subroutine for disabling IRQ:

```
        ; A subroutine to Disable an IRQ based on IRQ number
DisableIRQ
        ; Input R0 = IRQ number
        PUSH   {R0-R2, LR}
        AND.W  R1, R0, #0x1F   ; Generate Disable bit pattern for
                               ; the IRQ
        MOV    R2, #1
        LSL    R2, R2, R1      ; Bit pattern = (0x1 << (N & 0x1F))
        AND.W  R1, R0, #0xE0   ; Generate address offset if IRQ number
                               ; is above 31
        LSR    R1, R1, #3      ; Address offset = (N/32)*4 (Each word
                               ; has 32 IRQ enable)
        LDR    R0,=0xE000E180  ; CLREN register for external interrupt
                               ; #31-#0
        STR    R2, [R0, R1]    ; Write bit pattern to CLREN register
        POP    {R0-R2, PC}     ; Restore registers and Return
```

Similar subroutines can be developed for setting and clearing IRQ pending status registers.

Accessing NVIC Interrupt Registers

Most registers in the NVIC can be accessed using word, half word, or byte transfers. Selecting the right transfer size can make your program development easier. For example, priority-level registers are best programmed with byte transfers. In this way there is no need to worry about accidentally changing the priority of other exceptions.

Exception/Interrupt Handlers

In the Cortex-M3, interrupt handlers can be programmed completely in C, whereas in ARM7, an assembly handler is commonly used to ensure that all registers are saved and, in cases of systems with nested interrupt support, the processor needs to switch to a different mode to prevent losing information. These steps are not required in the Cortex-M3, making programming much easier.

In assembler, a simple exception handler might look like this:

```
irq1_handler
        ; Process IRQ request
        ...
        ; Deassert IRQ request in peripheral
        ...
        ; Interrupt return
        BX   LR
```

In many cases, the interrupt handler requires more than R0–R3 and R12 to process the interrupt, so we might need to save some other registers as well. The following example saves all registers that are not saved during the stacking process, but if some of the registers are not used by the exception handler, they can be omitted from the saved register list:

```
irq1_handler
        PUSH  {R4-R11, LR} ; Save all registers that are not saved
                           ; during stacking
        ; Process IRQ request
        ...
        ; Deassert IRQ request in peripheral (optional)
        ...
        POP   {R4-R11, PC} ; Restore registers and Interrupt return
```

Since POP is one of the instructions that can start interrupt returns, we can combine the register restore and interrupt return in the same instruction.

Depending on the design of a peripheral, it might be necessary for an exception handler to program the peripheral to deassert the exception request. If the exception request from the peripheral to the NVIC is a pulse signal, then there is no need for the exception handler to clear the exception request. Otherwise, the exception handler will need to clear the exception request so that it won't get pended again immediately after exception exit. In traditional ARM processors, a peripheral has to maintain its interrupt request until it is served, because the interrupt controllers designed for previous ARM cores do not have the pending memory.

With the Cortex-M3, if a peripheral generates interrupt requests in the form of pulses, the NVIC can store the request as a pending request status. Once the processor enters the exception handler, the pending status is cleared automatically. In this way, the exception handler does not have to program the peripheral to clear the interrupt request.

Software Interrupts

There are various ways to trigger an interrupt:

- External interrupt input
- Setting an interrupt pending register in the NVIC (see Chapter 8)
- Via the Software Trigger Interrupt Register (STIR) in the NVIC (see Chapter 8)

In most cases, some of the interrupts are unused and can be used as software interrupts. Software interrupts can work similarly to SVC, allowing accesses to system services. However, by default user programs cannot access the NVIC, except that they can access the NVIC's STIR only if the USERSETMPEND bit in the NVIC Configuration Control register is set (see Table D.17 in Appendix D).

Unlike the SVC, software interrupts are not precise. In other words, the interrupt preemption does not necessarily happen immediately, even when there is no blocking from interrupt mask registers or other interrupt service routines. As a result, if the instruction immediately following the write to the NVIC STIR depends on the result of the software interrupt, the operation could fail because the software interrupt could invoke after the instruction is executed.

To solve this problem, use the DSB instruction. For example:

```
        MOV     R0, #SOFTWARE_INTERRUPT_NUMBER
        LDR     R1,=0xE000EF00 ; NVIC Software Interrupt Trigger
                               ; Register address
        STR     R0, [R1]       ; Trigger software interrupt
        DSB                    ; Data synchronization barrier
        ...
```

However, there is still another possible problem: If the interrupt mask register is set or if the program code generating the software interrupt is an exception handler itself, there could be a chance that the software interrupt cannot execute. Therefore, the program code generating the software interrupt should check to see whether the software interrupt has been executed. This can be done by having a software flag set by the software interrupt handler.

Finally, setting USERSETMPEND can lead to another problem. After this is set, user programs can trigger any software interrupt except system exceptions. As a result, if the USERSETMPEND is used and the system contains untrusted user programs, exception handlers will need to check whether the exception is allowed, because it could have been triggered from user programs. Ideally, if a system contains untrusted user programs, it is best to provide system services only via SVC.

Example with Exception Handlers

In Chapter 7, we mentioned that the starting vector table should contain a reset vector, an NMI vector, and a hard fault vector, since the NMI and hard fault handler can take place without any exception enabling. After the program starts, we can then relocate the vector table to a different place in the SRAM. Depending on the application, relocation of the vector table might not be necessary. In the following example, we put the newly relocated vector table in the beginning of the SRAM, and then the data variables follow after it:

```
STACK_TOP      EQU   0x20002000   ; constant for SP starting value
NVIC_SETEN     EQU   0xE000E100   ; Set enable registers base address
NVIC_VECTTBL   EQU   0xE000ED08   ; Vector Table Offset Register
```

```
NVIC_AIRCR     EQU   0xE000ED0C   ; Application Interrupt and Reset
                                  ; Control Register
NVIC_IRQPRI    EQU   0xE000E400   ; Interrupt Priority Level register

               AREA  | Header Code|, CODE
               DCD   STACK_TOP    ; SP initial value
               DCD   Start        ; Reset vector
               DCD   Nmi_Handler  ; NMI handler
               DCD   Hf_Handler   ; Hard fault handler
               ENTRY
Start          ; Start of main program
               ; initialize registers
               MOV   r0, #0        ; initialize registers
               MOV   r1, #0
               ...

               ; Copy old vector table to new vector table
               LDR   r0,=0
               LDR   r1,=VectorTableBase
               LDMIA r0!,{r2-r5} ; Copy 4 words
               STMIA r1!,{r2-r5}

               DSB   ; Data synchronization barrier.

               ; Set vector table offset register
               LDR   r0,=NVIC_VECTTBL
               LDR   r1,=VectorTableBase
               STR   r1,[r0]

               ...
               ; Setup Priority group register
               LDR   r0,=NVIC_AIRCR
               LDR   r1,=0x05FA0500    ; Priority group 5
               STR   R1,[r0]

               ; Setup IRQ 0 vector
               MOV   r0, #0     ; IRQ#0
               LDR   r1, =Irq0_Handler
               BL    SetupIrqHandler

               ; Setup priority
               LDR   r0,=NVIC_IRQPRI
               LDR   r1,=0xC0   ; IRQ#0 priority
               STRB  r1,[r0,#0] ; Set IRQ0 priority at offset=0.
                                ; Note : Byte store
                                ;(IRQ#1 will have offset = 1)
               DSB   ; Data synchronization barrier. Make sure
                     ; everything ready before enabling interrupt
```

```
           MOV   r0, #0     ; select IRQ#0
           BL    EnableIRQ

           ...
           ;------------------------
           ; functions
SetupIrqHandler
           ; Input R0 = IRQ number
           ;       R1 = IRQ handler
           PUSH    {R0, R2, LR}
           LDR     R2,=NVIC_VECTTBL ; Get vector table offset
           LDR     R2,[R2]
           ADD     R0, #16      ; Exception number = IRQ number + 16
           LSL     R0, R0, #2   ; Times 4 (each vector is 4 bytes)
           ADD     R2, R0       ; Find vector address
           STR     R1,[R2]      ; store vector handler
           POP     {R0, R2, PC} ; Return
EnableIRQ

           ; Input R0 = IRQ number
           PUSH    {R0 - R3, LR}
           AND     R1, R0, #0x1F ; Get lower 5 bit to find bit pattern
           MOV     R2, #1
           LSL     R2, R2, R1    ; Bit pattern in R2
           BIC     R0, #0x1F
           LSR     R0, #3        ; word offset. (IRQ number can be
                                 ; higher than 32)
           LDR     R1, =NVIC_SETEN
           STR     R2,[R1, R0]   ; Set enable bit
           POP     {R0 - R3, PC} ; Return
           ;------------------------
           ; Exception handlers
Hf_Handler

           ...               ; insert your code here
           BX      LR        ; Return
Nmi_Handler
           ...               ; insert your code here
           BX      LR        ; Return
Irq0_Handler
           ...               ; insert your code here
           BX      LR        ; Return
           ;------------------------
           AREA    | Header Data|, DATA
           ALIGN   4
           ; Relocated vector table
VectorTableBase    SPACE 256   ; Number of bytes
VectorTableEnd                 ; (256 / 4 = upto 64 exceptions)
```

```
MyData1      DCD    0      ; Variables
MyData2      DCD    0

             END           ; End of file
```

This is a slightly long example. Let's start from the end, the data region first.

In the data memory region (almost the end of the program), we define a space of 256 bytes as a vector table (SPACE 256). This allows up to 64 exception vectors to be stored here. You might want to change the size if you want less or more space for the vector table. The other software variables follow the vector table space, so the variable *MyData1* is now in address 0x20000100.

In the beginning of the code, we defined a number of address constants for the rest of the program. So, instead of using numbers, we can use these constant names to make the program easier to understand.

The initial vector table now contains the reset vector, the NMI vector, and the hard fault handler vector. The preceding example code illustrates how to set up the exception vectors and does not contain actual NMI, hard fault, or IRQ handlers. Depending on the actual application, these handlers will have to be developed. The example uses BX LR as exception return, but that could be replaced by other valid exception return instructions.

After the initialization of registers, we copy the vector handlers to the new vector table in the SRAM. This is done by one multiple load and one multiple store instruction. If more vectors need to be copied, we can simply add extra load/store multiple instructions or increase the number of words to be copied for each pair of load and store instructions.

After the vector table is ready, we can relocate the vector table to the new one in the SRAM. However, to ensure that the transfer of the vector handler is complete, the DSB instruction is used.

We then need to set up the rest of the interrupt setting. The first one is the priority group setup. This needs to be done only once. In the example, two subroutines called *SetupIrqHandler* and *EnableIRQ* have been developed to make it easier to set up interrupts. Using the same code and simply changing the NVIC_SETEN to NVIC_CLREN, we can also add a similar function called *DisableIRQ*. After the handler and priority level have been set up, the IRQ can then be enabled.

Using SVC

SVC is a common way to allow user applications to access the API in an OS. This is because the user applications only need to know what parameters to pass to the OS; they don't need to know the memory address of API functions.

SVC instructions contain a parameter, which is 8-bit immediate data inside the instruction. The value is required for using the SVC instruction. For example:

```
        SVC 3 ; Call system service number 3
```

Inside the SVC handler, it will need to extract the parameter back from the instruction. To do this, the procedures illustrated in Figure 11.2 can be used.

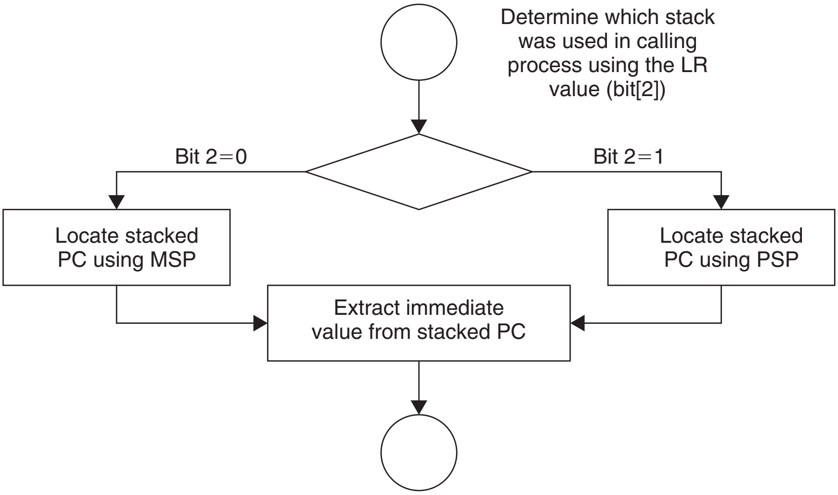

Determine which stack was used in calling process using the LR value (bit[2])
Bit 2=0
Bit 2=1
Locate stacked PC using MSP
Locate stacked PC using PSP
Extract immediate value from stacked PC

Figure 11.2 One Way to Extract the SVC Parameter

Here's some simple assembly code to do this:

```
svc_handler
       TST     LR, #0x4       ; Test EXC_RETURN number in LR bit 2
       ITE     EQ             ; if zero (equal) then
       MRSEQ   R0, MSP        ; Main Stack was used, put MSP in R0
       MRSNE   R0, PSP        ; else, Process Stack was used, put PSP
                              ; in R0
       LDR     R1,[R0,#24]    ; Get stacked PC from stack
       LDRB    R0,[R1,#-2]    ; Get the immediate data from the
                              ; instruction
       ; Now the immediate data is in R0
       ...
       BX      LR             ; Return to calling function
```

Once the calling parameter of the SVC is determined, the corresponding SVC function can be executed. An efficient way to branch into the correct SVC service code is to use table branch instructions such as TBB and TBH. However, if the table branch instruction is used, unless it is certain that the SVC calling parameter contains a correct value, you should do a value check on the parameter to prevent invalid SVC calling from crashing the system.

Since an SVC call cannot request another SVC service via the exception mechanism, the SVC handler should directly call another SVC function (for example, BL).

SVC Example: Use for Output Functions

Previously we developed various subroutines for output functions. Sometimes it is not good enough to use BL to call the subroutines—for example, when the functions are in different object files so that we might not be able to find out the address of the subroutines or when the branch address range is too large. In these cases, we might want to use SVC to act as an entry point for the output functions. For example:

```
      LDR    R0,=HELLO_TXT
      SVC    0         ; Display string pointed to by R0
      MOV    R0,#'A'
      SVC    1         ; Display character in R0
      LDR    R0,=0xC123456
      SVC    2         ; Display hexadecimal value in R0
      MOV    R0,#1234
      SVC    3         ; Display decimal value in R0
```

To use SVC, we need to set up the SVC handler. We can modify the function that we have done for IRQ. The only difference is that this function takes an exception type as input (SVC is exception type 11). In addition, this time we have further optimized the code to use the Thumb-2 instruction features:

```
SetupExcpHandler
        ; Input R0 = Exception number
        ;       R1 = Exception handler
        PUSH  {R0, R2, LR}
        LDR   R2,=NVIC_VECTTBL ; Get vector table offset
        LDR   R2,[R2]
        STR.W R1,[R2, R0, LSL #2] ; store vector handler in [R2+R0<<2]
        POP   {R0, R2, PC} ; Return
```

For *svc_handler*, the SVC calling number can be extracted as in the previous example, and the parameter passed to the SVC can be accessed by reading from the stack. In addition, the decision branches to reach various functions are added:

```
svc_handler
       TST     LR, #0x4      ; Test EXC_RETURN number in LR bit 2
       ITTEE   EQ            ; if zero (equal) then
       MRSEQ   R1, MSP       ; Main Stack was used, put MSP in R0
       MRSNE   R1, PSP       ; else, Process Stack was used, put PSP
                             ; in R0
       LDR     R0,[R1,#0]    ; Get stacked R0 from stack
       LDR     R1,[R1,#24]   ; Get stacked PC from stack
       LDRB    R1,[R1,#-2]   ; Get the immediate data from the
                             ; instruction
       ; Now the immediate data is in R1, input parameter is in R0
       PUSH    {LR}              ; Store LR to stack
       CBNZ    R1,svc_handler_1
       BL      Puts              ; Branch to Puts
       B       svc_handler_end
svc_handler_1
       CMP     R1,#1
       BNE     svc_handler_2
       BL      Putc              ; Branch to Putc
       B       svc_handler_end
svc_handler_2
       CMP     R1,#2
       BNE     svc_handler_3
       BL      PutHex            ; Branch to PutHex
       B       svc_handler_end
svc_handler_3
       CMP     R1,#3
       BNE     svc_handler_4
       BL      PutDec            ; Branch to PutDec
       B       svc_handler_end
svc_handler_4
       B       error             ; input not known
       ...
svc_handler_end
       POP     {PC}              ; Return
```

The *svc_handler* code should be put together with the outputting functions so that we can ensure that they are within the allowed branch range.

Notice that instead of the current contents of the register bank, the stacked register contents are used for parameter passing. This is because if a higher-priority interrupt takes place when the SVC is executed, the SVC will start after other interrupt handlers (tail chaining), and the contents of R0–R3 and R12 might be changed by the interrupt handler. This is caused by the characteristic that unstacking is not carried out if there is tail chaining of interrupts. For example:

1. A parameter is put in R0 as a parameter.
2. SVC is executed at the same time a higher-priority interrupt takes place.
3. Stacking is carried out, and R0–R3, R12, LR, PC, and xPSR are saved to the stack.
4. The interrupt handler is executed. R0–R3 and R12 can be changed by the handler. This is acceptable because these registers will be restored by hardware unstacking.
5. The SVC handler tail chains the interrupt handler. When SVC is entered, the contents in R0–R3 and R12 can be different from the value when SVC is called. However, the correct parameter is stored in the stack and can be accessed by the SVC handler.

Make the Most of the Addressing Modes

From the code examples of the *SetupIrqHandler* and *SetupExcpHandler* routines, we find that the code can be shortened a lot if we utilize the addressing mode feature in the Cortex-M3. In *SetupIrqHandler*, the destination address of the IRQ vector is calculated, and then the store is carried out:

```
SetupIrqHandler
   PUSH  {R0, R2, LR}
   LDR   R2,=NVIC_VECTTBL ; Get vector table offset        ; Step 1
   LDR   R2,[R2]                                           ; Step 2
   ADD   R0, #16       ; Exception number = IRQ number + 16 ; Step 3
   LSL   R0, R0, #2    ; Times 4 (each vector is 4 bytes)   ; Step 4
   ADD   R2, R0        ; Find vector address                ; Step 5
   STR   R1,[R2]       ; store vector handler               ; Step 6
   POP   {R0, R2, PC} ; Return
```

In *SetupExcpHandler*, the operation Step 4–6 are reduced to just one step:

```
SetupExcpHandler
         PUSH    {R0, R2, LR}
         LDR     R2,=NVIC_VECTTBL ; Get vector table offset
         LDR     R2,[R2]
         STR.W   R1,[R2, R0, LSL #2] ; store vector handler in
                                     ; [R2+R0<<2]
         POP    {R0, R2, PC} ; Return
```

In general, we can reduce the number of instructions required if the data address is like one of these:

- Rn + 2N*Rm
- Rn +/− immediate_offset

For the *SetupIrqHandler* routine, the shortest code we can get is this:

```
SetupIrqHandler
         PUSH    {R0, R2, LR}
         LDR     R2,=NVIC_VECTTBL ; Get vector table offset ; Step 1
         LDR     R2,[R2]                                    ; Step 2
         ADD     R2, #(16*4)      ; Get IRQ vector start    ; Step 3
         STR.W   R1,[R2, R0, LSL #2] ; Store vector handler ; Step 4
         POP     {R0, R2, PC} ; Return
```

Using SVC with C

In most cases, an assembler handler code is needed for parameter passing to SVC functions. This is because the parameters should be passed by the stack, not by registers, as explained earlier. If the SVC handler is to be developed in C, a simple assembly wrapper code can be used to obtain the stacked register location and pass it on to the SVC handler. The SVC handler can then extract the SVC number and parameters from the stack pointer information. Assuming that RealView Development Suite (RVDS) or KEIL RealView Microcontroller Development Kit is used, the assembler wrapper can be implemented with Embedded Assembler:

```
// Assembler wrapper for extracting stack frame starting location.
// Starting of stack frame is put into R0 and then branch to the
// actual SVC handler.
__asm void svc_handler_wrapper(void)
{
  IMPORT svc_handler
  TST     LR, #4
  ITE     EQ
  MRSEQ   R0, MSP
  MRSNE   R0, PSP
  B       svc_handler
} // No need to add return (BX LR) because return of svc_handler
  // should return execution to SVC calling program directly
```

The rest of the SVC handler can then be implemented in C using R0 as input (stack frame starting location), which is used to extract the SVC number and passing parameters (R0–R3):

```
// SVC handler in C, with stack frame location as input parameter
// and use it as a memory pointer pointing to an array of arguments.
// svc_args[0] = R0 , svc_args[1] = R1
// svc_args[2] = R2 , svc_args[3] = R3
// svc_args[4] = R12, svc_args[5] = LR
// svc_args[6] = Return address (Stacked PC)
// svc_args[7] = xPSR
void svc_handler(unsigned int * svc_args)
{
  unsigned int svc_number;
  unsigned int svc_r0;
  unsigned int svc_r1;
  unsigned int svc_r2;
  unsigned int svc_r3;

  svc_number = ((char *) svc_args[6])[-2]; // Memory[(Stacked PC)-2]
  svc_r0     = ((unsigned long) svc_args[0]);
  svc_r1  = ((unsigned long) svc_args[1]);
  svc_r2  = ((unsigned long) svc_args[2]);
  svc_r3  = ((unsigned long) svc_args[3]);
printf (“SVC number = %xn”, svc_number);
printf (“SVC parameter 0 = %x\n”, svc_r0);
printf (“SVC parameter 1 = %x\n”, svc_r1);
printf (“SVC parameter 2 = %x\n”, svc_r2);
printf (“SVC parameter 3 = %x\n”, svc_r3);

return;
}
```

Note that SVC cannot return results to the calling program in the same way as in normal C functions. Normal C functions return values by defining the function with a data type such as *unsigned int func( )* and use *return* to pass the return value, which actually puts the value in register R0. If an SVC handler put return values in register R0 to R3 when exiting the handler, the register values would be overwritten by the unstacking sequence. Therefore, if an SVC has to return results to a calling program, it must directly modify the stack frame so that the value can be loaded into the register during unstacking.

To call an SVC inside a C program for ARM RealView Development Suite (RVDS) or KEIL RealView Microcontroller Development Kit (RV-MDK), we can use the _ _*svc compiler* keyword. For example, if four variables are to be passed to an SVC function number 3, an SVC named *call_svc_3* can be declared as:

```
void __svc(0x03) call_svc_3(unsigned long svc_r0, unsigned long
svc_r1, unsigned long svc_r2, unsigned long svc_r3);
```

This will then allow the C program code to call the system call by:

```
int main(void)
{
  unsigned long p0, p1, p2, p3; // parameters to pass to SVC handler
  . . .
  call_svc_3(p0, p1, p2, p3); // call SVC number 3, with parameters
                              // p0, p1, p2, p3 pass to the SVC
  . . .
  return;
}
```

Detailed information on using the _ _*svc* keyword in RealView Development Suite or RealView C Compiler can be found in the *RVCT 3.0 Compiler and Library Guide* (Ref 6).

For users of the GNU tool chain, since there is no _ _*svc* keyword in GCC, the SVC has to be accessed by inline assembler. For example, if the SVC call number 3 is needed with one input variable and it returns one variable via register R0 (according to the *AAPCS*, Ref 5, the first passing variable will use register R0), the following inline Assembler code can be used to call the SVC:

```
int MyDataIn = 0x123;

__asm __volatile ("mov R0, %0\n"
                  "svc 3     \n" : "" : "r" (MyDataIn) );
```

This inline assembler code can be broken down into the following parts, with input data specified by *r* (*MyDataIn*) and no output field (indicated as "" in the preceding code):

```
         __asm ( assembler_code : output_list : input_list )
```

More examples using inline Assembler in the GNU tool chain can be found in Chapter 19 of this book. For complete details on passing parameters to or from inline Assembler, refer to the GNU tool chain documentation.

This page intentionally left blank

CHAPTER 12

Advanced Programming Features and System Behavior

In This Chapter:

- Running a System with Two Separate Stacks
- Double-Word Stack Alignment
- Nonbase Thread Enable
- Performance Consideration
- Lockup Situations

Running a System with Two Separate Stacks

One of the important features of v7-M architecture is the capability to allow the user application stack to be separated from the privileged/kernel stack. If the optional MPU is implemented, it could be used to block user applications from accessing kernel stack memory so that they cannot crash the kernel by memory corruption.

Typically, a robust system based on the Cortex-M3 has the following properties:

- Exception handlers using MSP
- Kernel code invoked by a SYSTICK exception at regular intervals, running in the privileged access level for task scheduling and system management
- User applications running as threads with the user access level (nonprivileged); these applications use PSP
- Stack memory for kernel and exception handlers is pointed to by the MSP, and the stack memory is restricted to privileged accesses only if the MPU is available
- Stack memory for user applications is pointed to by the PSP

Assume that the system memory has an SRAM memory. We could set up the MPU so that the SRAM is divided into two regions for user and privileged access. Each region is used by application data as well as stack memory space. Since stack operation in the Cortex-M3 is full descending, the initial value of stack pointers needs to be pointed to the top of the regions.

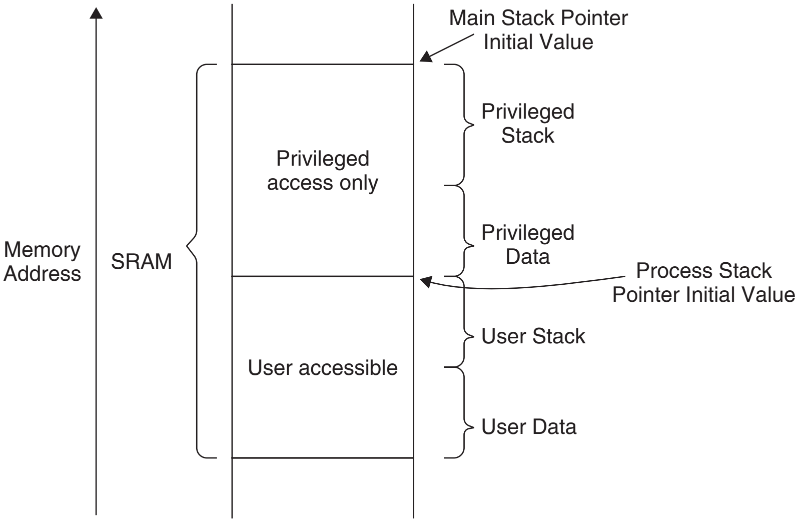


Figure 12.1 Example Memory Use with Privileged Data and User Application Data

After power-up, only the MSP is initialized (by fetching address 0x0 in the power-up sequence). Additional steps are required to set up a completely robust two-stack system. For applications in assembly code, it can simply be:

```
; Start at privileged level (this code locates in user
; accessible memory)
BL      MpuSetup    ; Setup MPU regions and enable memory
                    ; protection
LDR     R0,=PSP_TOP ; Setup Process SP to top of process stack
MSR     PSP, R0
BL      SystickSetup; Setup Systick and systick exception to
                    ; invoke OS kernel at regular intervals
MOV    R0, #0x3     ; Setup CONTROL register so that user
                    ; program use PSP,
MSR     CONTROL, R0 ; and switch current access level to user
B       UserApplicationStart  ; Now we are in user access
                              ; level. Start user code
```

This arrangement is fine for assembler, but for C programs, switching stack pointers in the middle of a C function can cause loss of local variables (because in C functions or subroutines, local variables may be put onto stack memory). The Cortex-M3 TRM (Ref 1) suggests that we use an ISR like SVC to invoke the kernel, then change the stack pointer by modifying the EXC_RETURN value.

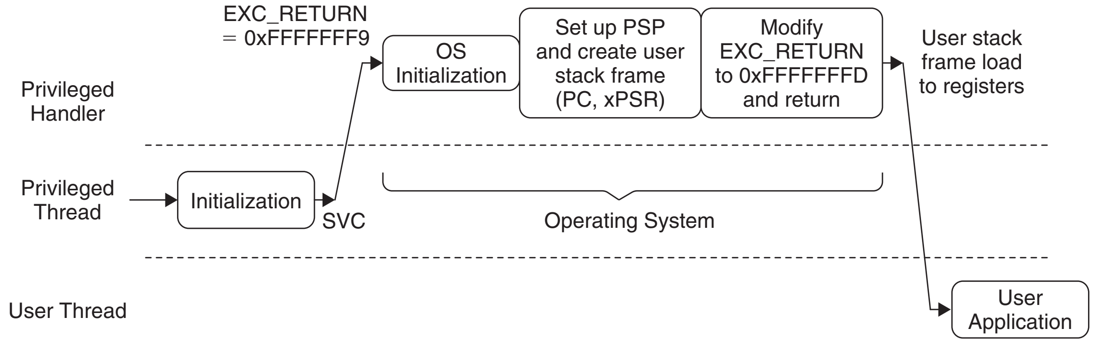


Figure 12.2 Initialization of Multiple Stacks in a Simple OS

In most cases, EXC_RETURN modification and stack switching are included in the operating system. After the user application starts, the SYSTICK exception can be used regularly to invoke the operating system for system management and possibly arrange context switching if needed.

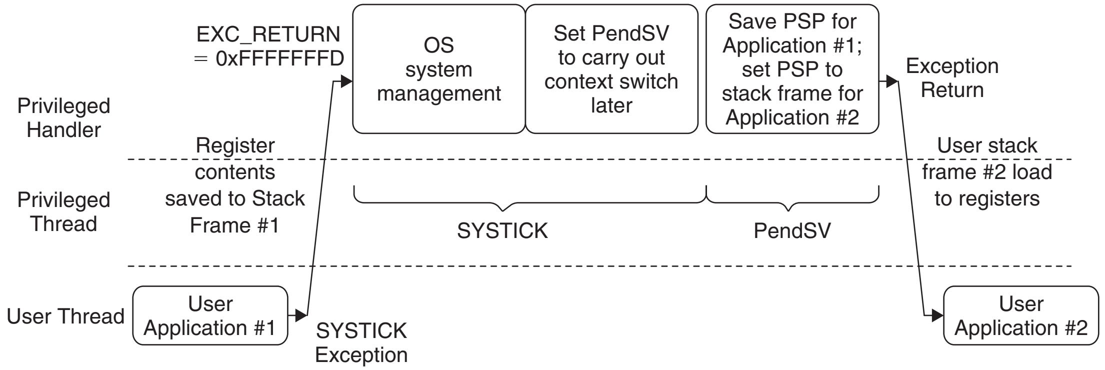

EXC_RETURN
= 0xFFFFFFFD
Privileged Handler
OS system management
Set PendSV to carry out context switch later
Save PSP for Application #1; set PSP to stack frame for Application #2
Exception Return
Register contents saved to Stack Frame #1
User stack frame #2 load to registers
Privileged Thread
SYSTICK
PendSV
User Thread
User Application #1
SYSTICK Exception
User Application #2

Figure 12.3 Context Switching in a Simple OS

Note that context switching is carried out in PendSV (a low-priority exception) to prevent context switching at the middle of an interrupt handler.

However, many applications do not require an operating system, but it is still helpful to use separate stacks for different sections of application code as a way to improve reliability. One possible way to handle this is to start Cortex-M3 with the MSP pointed to a process stack region. This way the initialization is done with the process stack region but using MSP. Before starting the user application, the following code is executed:

```
; Start at privileged level, MSP point to User stack
MpuSetup();     // Setup MPU regions and enable memory protection
SystickSetup(); // Setup Systick and systick exception for routine
                // system management code
SwitchStackPointer();  // Call an assembly subroutine to switch SP
  /*; ------Inside SwitchStackPointer -----
  PUSH  {R0, R1, LR}
  MRS   R0, MSP       ; Save current stack pointer
  LDR   R1, =MSP_TOP  ; Change MSP to new location
  MSR   MSP, R1
  MSR   PSP, R0       ; Store current stack pointer in PSP
  MOV   R0, #0x3
  MSR   CONTROL, R0   ; Switch to user mode, and use PSP as
                      ; current stack
  POP   {R0, R1, PC}  ; Return
  ; ------ Back to C program -----*/
; Now we are in User mode, using PSP and the local variables
; still here
UserApplicationStart();    // Start application code in user mode
```

Double-Word Stack Alignment

In applications that conform to AAPCS[1] it is necessary to ensure that the stacking of registers at exception handling are aligned to the primitive data size (1, 2, 4, or 8 bytes). This is a configurable option on the Cortex-M3 processor. To enable this feature, the STKALIGN bit in the NVIC Configuration Control register needs to be set (see Table D.17 in Appendix D). For example, this can be done in assembly language:

```
LDR      R0,=0xE000ED14  ; Set R0 to be address of NVIC CCR
LDR      R1, [R0]
ORR.W    R1, R1, #0x200  ; Set STKALIGN bit
STR      R1, [R0]        ; Write to NVIC CCR
```

or in C language:

```
#define NVIC_CCR ((volatile unsigned long *)(0xE000ED14))
*NVIC_CCR = *NVIC_CCR | 0x200; /* Set STKALIGN in NVIC */
```

When the STKALIGN bit is set during exception stacking, bit 9 of the stacked xPSR is used to indicate whether a stack pointer adjustment has been made to align the stacking.

[1] Procedure Call Standard for the ARM Architecture (AAPCS) (Ref 5). An advisory note has been published on the ARM Web site regarding SP alignment and AAPCS; see www.arm.com/pdfs/ABI-Advisory-1.pdf.

When unstacking, the SP adjustment checks bit 9 of the stacked xPSR and adjusts the SP accordingly.

To prevent stack data corruption, the STKALIGN bit must not be changed within an exception handler; this can cause a mismatch of stack pointer location before and after the exception.

This feature is available from Cortex-M3 revision 1 onward. Early Cortex-M3 products based on revision 0 do not have this feature. This feature should be used if the AAPCS conformation is required. Also, this feature is recommended when the application (or part of it) is developed in C and when the program contains data that is double-word size.

Nonbase Thread Enable

In the Cortex-M3 it is possible to switch a running interrupt handler from privileged level to user access level. This is needed when the interrupt handler code is part of a user application and should not be allowed to have privileged access. This feature is enabled by the Nonbase Thread Enable (NONBASETHRDENA) bit in the NVIC Configuration Control register.

Use This Feature with Caution

Due to the need to manually adjust the stack and modify stacked data, this feature should be avoided in normal application programming. If it is necessary to use this feature, it must be done very carefully, and the system designer must ensure that the interrupt service routine is terminated correctly. Otherwise, it could cause some interrupts with the same or lower priority levels to be masked.

To use this feature, an exception handler redirection is involved. The vector in the vector table points to a handler running in privileged mode but located in user mode accessible memory:

```
redirect_handler
        PUSH     {LR}
        SVC      0    ; A SVC function to change from privileged to
                      ; user mode
        BL       User_IRQ_Handler
        SVC      1    ; A SVC function to change back from user to
                      ; privileged mode
        POP      {PC} ; Return
```

The SVC handler is divided into three parts:

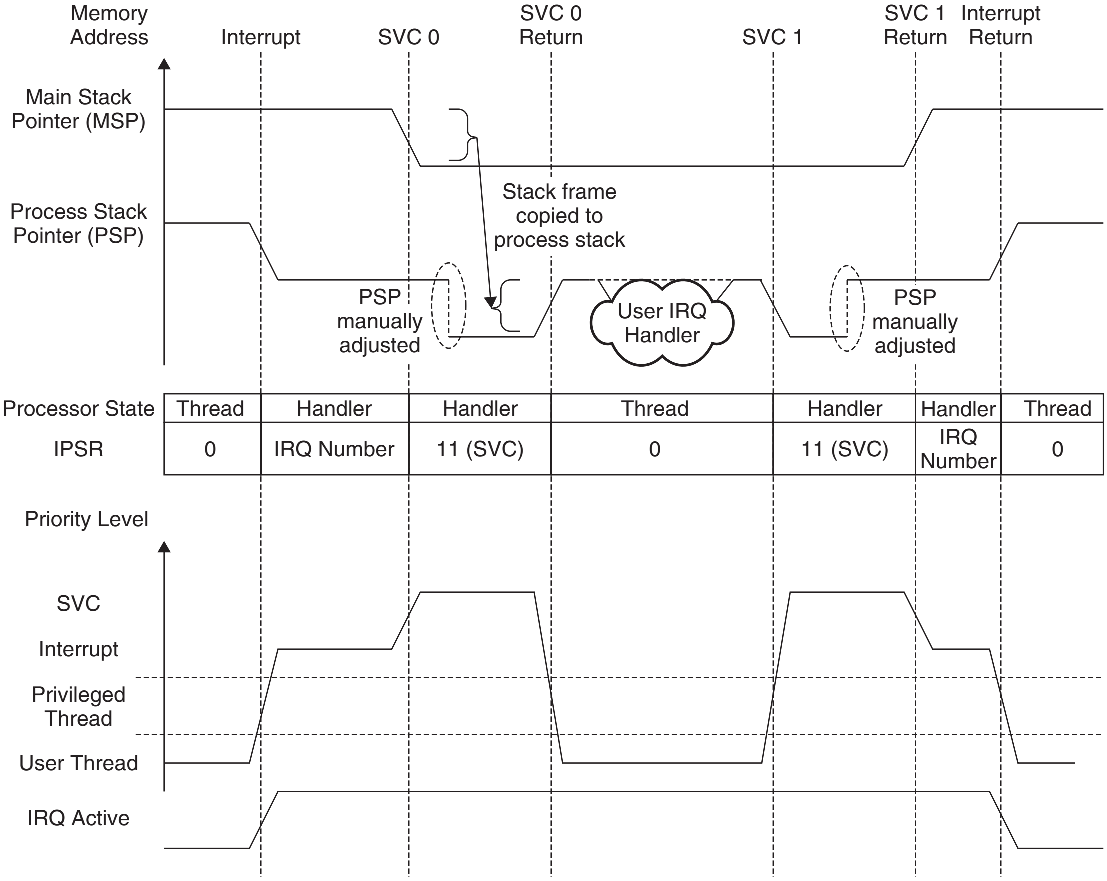
- Determine the parameter when calling SVC.
- SVC service #0 enables the nonbase Thread enable, adjusts the user stack and EXC_RETURN value, and returns to the redirect handler in user mode, using the process stack.

- SVC service #1 disables the nonbase Thread enable, restores the user stack pointer position, and returns to the redirect handler in privileged mode, using the main stack.

```
svc_handler
       TST     LR, #0x4        ; Test EXC_RETURN bit 2
       ITE     EQ              ; if zero then
       MRSEQ   R0, MSP         ; Get correct stack pointer to R0
       MRSNE   R0, PSP
       LDR     R1,[R0, #24]    ; Get stacked PC
       LDRB    R0,[R1, #-2]    ; Get parameter at stacked PC – 2
       CBZ     r0, svc_service_0
       CMP     r0, #1
       BEQ     svc_service_1
       B.W     Unknown_SVC_Request

svc_service_0  ; Service to switch handler from privileged mode to
               ; user mode
       MRS     R0, PSP         ; Adjust PSP
       SUB     R0, R0, #0x20   ; PSP = PSP + 0x20
       MSR     PSP, R0
       MOV     R1, #0x20       ; Copy stack frame from main stack to
                               ; process stack

svc_service_0_copy_loop
       SUBS    R1, R1, #4
       LDR     R2,[SP, R1]
       STR     R2,[R0, R1]
       CMP     R1, #0
       BNE     svc_service_0_copy_loop
       STRB    R1,[R0, #0x1C]  ; Clear stacked IPSR of user stack to 0
       LDR     R0, =0xE000ED14 ; Set Non-base thread enable in CCR
       LDR     r1,[r0]
       ORR     r1, #1
       STR     r1,[r0]
       ORR     LR, #0xC ; Change LR to return to thread, using PSP
       BX      LR

svc_service_1  ; Service to switch handler back from user mode to
               ; privileged mode
       MRS     R0, PSP         ; Update stacked PC in privileged
                               ; stack so that it
       LDR     R1,[R0, #0x18]  ; return to the instruction after 2nd
                               ; SVC in redirect
       STR     R1,[SP, #0x18]  ; handler
       MRS     R0, PSP         ; Adjust PSP back to what it was
                               ; before 1st SVC
```

```
ADD     R0, R0, #0x20
MSR     PSP, R0
LDR     R0, =0xE000ED14  ; Clear Non-base thread enable in CCR
LDR     r1,[r0]
BIC     r1, #1
STR     r1,[r0]
BIC     LR, #0xC        ; Return to handler mode, using main
                        ; stack
BX      LR
```

The SVC services are used because the only way you can change the IPSR is via an exception return. Other exceptions such as software-triggered interrupts could be used, but they are not recommended because they are imprecise and could be masked, which means that there is a possibility that the required stack copying and switch operation is not carried out immediately. The sequence of the code is illustrated in Figure 12.4, which shows the stack pointer changes and the current exception priority.



Processor State	Thread	Handler	Handler	Thread	Handler	Handler	Thread
IPSR	0	IRQ Number	11 (SVC)	0	11 (SVC)	IRQ Number	0

Figure 12.4 Operation of Nonbase Thread Enable

In the diagram, the manual adjustment of the PSP inside the SVC services is highlighted by circles made of dotted lines.

Performance Considerations

To get the best out of the Cortex-M3, a few aspects need to be considered. First, we need to avoid memory wait states. During the design stage of the microcontroller or SoC, the designer should optimize the memory system design to allow instruction and data accesses to be carried out at the same time, and use 32-bit memories if possible. For developers, the memory map should be arranged so that program code is executed from the Code region and majority of data accesses is done via the system bus. This way data accesses can be carried out at the same time as instruction fetches.

Second, the interrupt vector table should also be put into the Code region if possible. Thus vector fetch and stacking can be carried out at the same time. If the vector table is located in the SRAM, extra clock cycles might result in interrupt latency because both vector fetch and stacking could share the same system bus (unless the stack is located in the Code region, which uses a D-Code bus).

If possible, avoid using unaligned transfers. An unaligned transfer might take two or more AHB transfers to complete and will slow program performance, so plan your data structure carefully. In assembly language with ARM tools, you can use the ALIGN directive to ensure that a data location is aligned.

Most of you might be using C language for development, but for those who are using assembly, you can use a few tricks to speed up parts of the program:

1. Use memory access instruction with offset. When multiple memory locations in a small region are to be accessed, instead of writing:

```
LDR      R0, =0xE000E400  ; Set interrupt priority #3,#2,#1,#0
LDR      R1, =0xE0C02000  ; priority levels
STR      R1,[R0]
LDR      R0, =0xE000E404  ; Set interrupt priority #7,#6,#5,#4
LDR      R1, =0xE0E0E0E0  ; priority levels
STR      R1,[R0]
```

you can reduce the program code to:

```
LDR      R0, =0xE000E400  ; Set interrupt priority #3,#2,#1,#0
LDR      R1, =0xE0C02000  ; priority levels
STR      R1,[R0]
LDR      R1,=0xE0E0E0E0   ; priority levels
STR      R1,[R0,#4]       ; Set interrupt priority #7,#6,#5,#4
```

The second store uses an offset of the first address and hence reduces the number of instructions.

2. Combine multiple memory accesses into load/store multiple instructions (LDM/STM). The preceding example can be further reduced by using STM instruction:

```
LDR      R0,=0xE000E400  ; Set interrupt priority base
LDR      R1,=0xE0C02000  ; priority levels #3,#2,#1,#0
LDR      R2,=0xE0E0E0E0  ; priority levels #7,#6,#5,#4
STMIA    R0, {R1, R2}
```

3. Use IT instruction blocks to replace small conditional branches. Since the Cortex-M3 is a pipelined processor, a branch penalty happens when a branch operation is taken. If the conditional branch operation is used to skip a few instructions, this can be replaced by the IT instruction block, which might save a few clock cycles.

4. If an operation can be carried out by either two Thumb instructions or a single Thumb-2 instruction, the Thumb-2 instruction method should be used because it gives a shorter execution time, despite the fact that the memory size is the same.

Lockup Situations

When an error condition occurs, the corresponding fault handler will be triggered. If another fault takes place inside the usage fault/bus fault/memory management fault handler, the hard fault handler will be triggered. However, what if we get another fault inside the hard fault handler? In this case, a lockup situation will take place.

What Happens During Lockup?

During lockup, the program counter will be forced to 0xFFFFFFFX and will keep fetching from that address. In addition, an output signal called LOCKUP from the Cortex-M3 will be asserted to indicate the situation. Chip designers might use this signal to trigger a reset at the system reset generator.

Lockup can take place when:

- Faults occur inside the hard fault handler (double fault)
- Faults occur inside the NMI handler
- Bus faults occur during the reset sequence (initial SP or PC fetch)

For double-fault situations, it is still possible for the core to respond to an NMI and execute the NMI handler. But after the handler completes it will return to the lockup state, with the program counter restored to 0xFFFFFFFX. In this case, the system locks up and the current priority level is held at −1. If an NMI occurs, the processor will still preempt and execute the NMI handler because the NMI has a higher priority (−2) than the current priority level (−1).

When the NMI is complete and returns to the lockup state, the current exception priority is returned to −1.

Normally, the best way to exit a lockup is to perform a reset. Alternatively, for a system with a debugger attached, it is possible to halt the core, change the PC to a different value, and start the program execution from there. In most cases this might not be a good idea, since a number of registers, including the interrupt system, might need reinitialization before the system can be returned to normal operation.

You might wonder why we do not simply reset the core when a lockup takes place. You might want to do that in a live system, but during software development we should first try to find out the cause of the problem. If we reset the core immediately, we might not be able to analyze what went wrong, because registers will be reset and hardware status will be changed. In most Cortex-M3 microcontrollers, a watchdog timer can be used to reset the core if it enters the lockup state.

Note that a bus fault that occurs during stack when entering a hard fault handler or NMI handler does not cause lockup, but the bus fault handler will be pended.

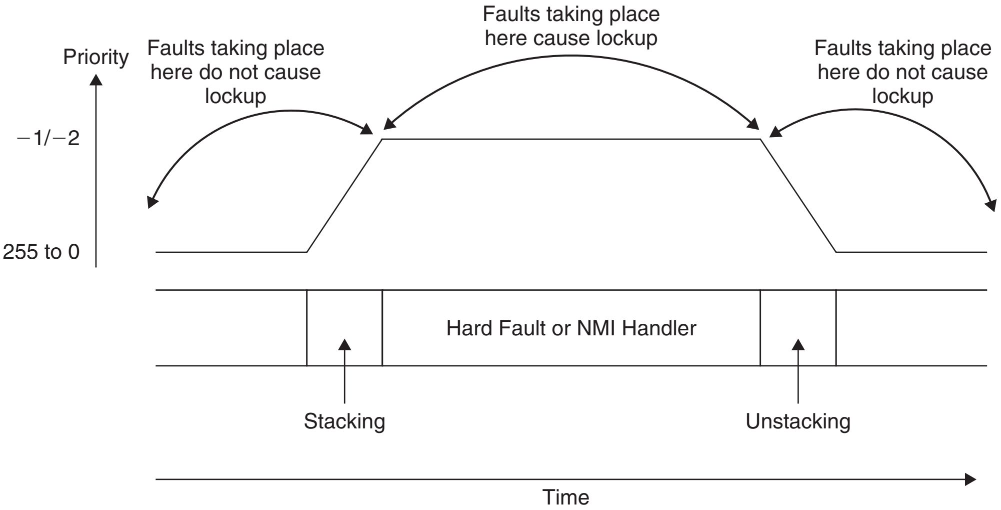

Priority
Faults taking place here do not cause lockup
Faults taking place here cause lockup
Faults taking place here do not cause lockup
−1/−2
255 to 0
Hard Fault or NMI Handler
Stacking
Unstacking
Time

Figure 12.5 Only a Fault Occurring During a Hard Fault or NMI Handler Will Cause Lockup

Avoiding Lockup

It is important to take extra care to prevent lockup problems when you're developing an NMI or hard fault handler. For example, we can avoid unnecessary stack accesses in a hard fault handler unless we know that the memory is functioning correctly and the stack pointer is still valid. In developing complex systems, one of the possible causes of a bus fault or memory fault is stack pointer corruption. If we start the hard fault handler with something like this:

```
hard_fault_handler
       PUSH    {R4-R7,LR}   ; Bad idea unless you are sure that the
                            ; stack is safe to use!
       ...
```

and if the fault was caused by a stack error, we could enter lockup in our hard fault handler straight away. In general, when programming hard fault, bus fault, and memory management fault handlers, it might be worth checking whether the stack pointer is in valid range before we carry out more stack operations. For coding NMI handlers, we can try to reduce risk caused by stack operation by using R0–R3 and R12 only, since they are already stacked.

One approach for developing hard fault and NMI handlers is to carry out only the essential tasks inside the handlers, and the rest of the tasks, such as error reporting, can be pended using a separate exception such as PendSV or a software interrupt. This helps to ensure that the hard fault handler or NMI is small and robust.

Furthermore, we should ensure that the NMI and hard fault handler code will not try to use SVC instructions. Since SVC always has lower priority than hard fault and NMI, using SVC in these handlers will cause lockup. This might look simple, but when your application is complex and you call functions from different files in your NMI and hard fault handler, you might accidentally call a function that contains an SVC instruction. Therefore, before you develop your software, you need to carefully plan the SVC implementation.

This page intentionally left blank

CHAPTER 13

The Memory Protection Unit

In This Chapter:

- Overview
- MPU Registers
- Setting Up the MPU
- Typical Setup

Overview

The Cortex-M3 design includes an optional Memory Protection Unit (MPU). Including the MPU in the microcontrollers or SoC products provides memory protection features, which can make the developed products more robust. The MPU needs to be programmed and enabled before use. If the MPU is not enabled, the memory system behavior is the same as though no MPU is present.

The MPU can improve the reliability of an embedded system by:

- Preventing user applications from corrupting data used by the operating system
- Separating data between processing tasks by blocking tasks from accessing others' data
- Allowing memory regions to be defined as read-only so that vital data can be protected
- Detecting unexpected memory accesses (for example, stack corruption)

In addition, the MPU can also be used to define memory access characteristics such as caching and buffering behaviors for different regions.

The MPU sets up the protection by defining the memory map as a number of regions. Up to eight regions can be defined, but it is also possible to define a default background memory map for privileged accesses. Accesses to memory locations that are not defined in the MPU regions or not permitted by the region settings will cause the memory management fault exception to take place.

MPU regions can be overlapped. If a memory location falls on two regions, the memory access attributes and permission will be based on the highest-numbered region. For example, if a transfer address is within the address range defined for region 1 and region 4, the region 4 settings will be used.

MPU Registers

The MPU contains a number of registers. The first one is the MPU Type register (see Table 13.1).

Table 13.1 MPU Type Register (0xE000ED90)

Bits	Name	Type	Reset Value	Description
23:16	IREGION	R	0	Number of instruction regions supported by this MPU; because ARM v7-M architecture uses a unified MPU, this is always 0
15:8	DREGION	R	0 or 8	Number of regions supported by this MPU; in the Cortex-M3 this is either 0 (MPU not present) or 8 (MPU present)
0	SEPARATE	R	0	This is always 0 as the MPU is unified

The MPU Type register can be used to determine whether the MPU is fitted. If the DREGION field is read as 0, the MPU is not implemented (see Table 13.2).

Table 13.2 MPU Control Register (0xE000ED94)

Bits	Name	Type	Reset Value	Description
2	PRIVDEFENA	R/W	0	Privileged default memory map enable. When set to 1 and if the MPU is enabled, the default memory map will be used for privileged accesses as a background region. If this bit is not set, the background region is disabled and any access not covered by any enabled region will cause a fault.
1	HFNMIENA	R/W	0	If set to 1, it enables the MPU during the hard fault handler and NMI handler; otherwise, the MPU is not enabled for the hard fault handler and NMI.
0	ENABLE	R/W	0	Enables the MPU if set to 1.

By using PRIVDEFENA and if no other regions are set up, privileged programs will be able to access all memory locations, and only user programs will be blocked. However, if other MPU regions are programmed and enabled, they can override the background region. For example, for two systems with similar region setups but only one with PRIVDEFENA set to 1 (the right-hand side in Figure 13.1), the one with PRIVDEFENA set to 1 will allow privileged access to background regions.

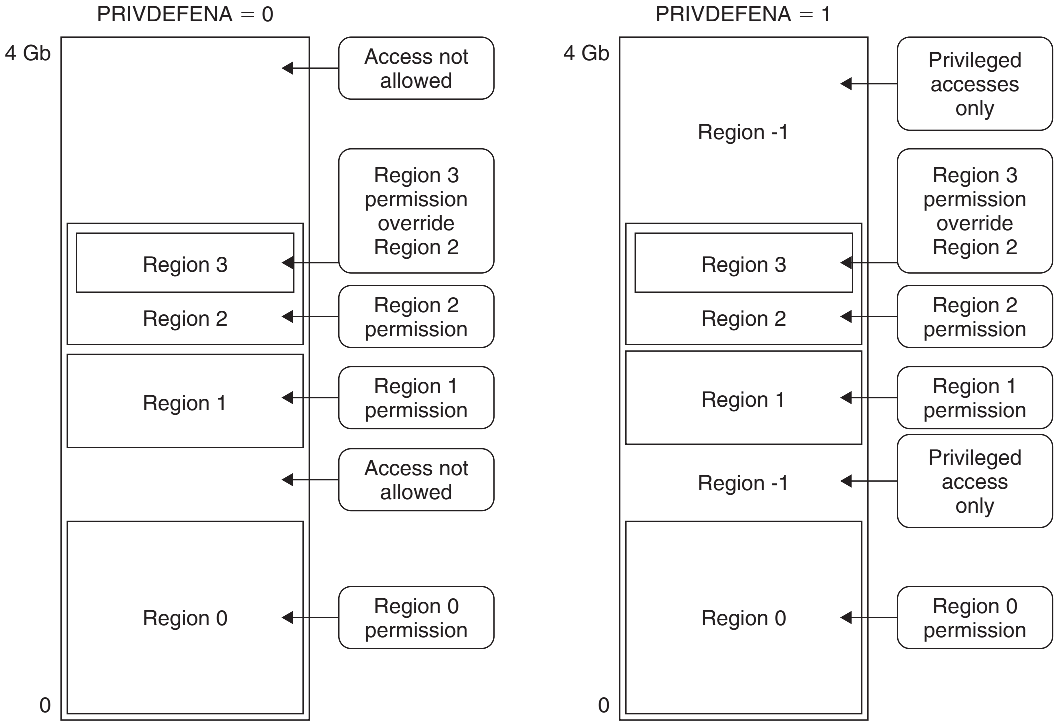


Figure 13.1 The Effect of PRIVDEFENA

Setting the enable bit in the MPU Control register is usually the last step in the MPU setup code. Otherwise the MPU might generate faults by accident before the region configuration is done. In some situations, it might be worth clearing the MPU Enable at the start of the MPU configuration routine to make sure that the MPU faults won't be triggered by accident during setup of MPU regions.

Table 13.3 MPU Region Number Register (0xE000ED98)

Bits	Name	Type	Reset Value	Description
7:0	REGION	R/W	—	Select the region that is being programmed. Since eight regions are supported in the Cortex-M3 MPU, only bit[2:0] of this register is implemented.

Before each region is set up, write to this register to select the region to be programmed (see Table 13.3).

Using the VALID and REGION fields in the MPU Region Base Address register (see Table 13.4), we can skip the step of programming the MPU Region Number register. This might reduce the complexity of the program code, especially if the whole MPU setup is defined in a lookup table.

Table 13.4 MPU Region Base Address Register (0xE000ED9C)

Bits	Name	Type	Reset Value	Description
31:N	ADDR	R/W	—	Base address of the region; N is dependent on the region size—for example, a 64 k size region will have a base address field of [31:16].
4	VALID	R/W	—	If this is 1, the REGION defined in bit[3:0] will be used in this programming step; otherwise, the region selected by the MPU Region Number register is used.
3:0	REGION	R/W	—	This field overrides the MPU Region Number register if VALID is 1; otherwise it is ignored. Since eight regions are supported in the Cortex-M3 MPU, the region number override is ignored if the value of the REGION field is larger than 7.

We also need to define the memory address and properties of each region. This is controlled by the MPU Region Base Attribute and Size register (see Table 13.5).

Table 13.5 MPU Region Base Attribute and Size Register (0xE000EDA0)

Bits	Name	Type	Reset Value	Description
31:29	Reserved	—	—	—
28	XN	R/W	—	Instruction Access Disable (1 = Disable instruction fetch from this region; an attempt to do so will result in a memory management fault)
27	Reserved	—	—	—
26:24	AP	R/W	—	Data Access Permission field
23:22	Reserved	—	—	—
21:19	TEX	R/W	—	Type Extension field
18	S	R/W	—	Shareable
17	C	R/W	—	Cacheable
16	B	R/W	—	Bufferable
15:8	SRD	R/W	—	Subregion disable
7:6	Reserved	—	—	
5:1	REGION SIZE	R/W	—	MPU Protection Region size
0	SZENABLE	R/W	—	Region enable

The REGION SIZE field (5-bit) in the MPU Region Base Attribute and Size register determines the size of the region (see Table 13.6).

Table 13.6 Encoding of REGION Field for Different Memory Region Sizes

REGION Size	Size	REGION Size	Size
b00000	Reserved	b10000	128 KB
b00001	Reserved	b10001	256 KB
b00010	Reserved	b10010	512 KB
b00011	Reserved	b10011	1 MB
b00100	32 Byte	b10100	2 MB
b00101	64 Byte	b10101	4 MB
b00110	128 Byte	b10110	8 MB
b00111	256 Byte	b10111	16 MB
b01000	512 Byte	b11000	32 MB
b01001	1 KB	b11001	64 MB
b01010	2 KB	b11010	128 MB
b01011	4 KB	b11011	256 MB
b01100	8 KB	b11100	512 MB
b01101	16 KB	b11101	1 GB
b01110	32 KB	b11110	2 GB
b01111	64 KB	b11111	4 GB

The subregion disable field (bit [15:8] of the MPU Region Base Attribute and Size register) is used to divide a region into eight equally sized subregions and define each of them as enable or disable. If a subregion is disabled and it was overlapped with another region, the access rules for the other region are applied. If the subregion is disabled and it does not overlap any other region, access to this memory range will result in a memory management fault. Subregions cannot be used if the region size is 128 bytes or less.

The Data Access Permission (AP) field (bit[26:24]) defines the access permission of the region (see Table 13.7).

Table 13.7 Encoding of AP Field for Various Access Permission Configurations

AP Value	Privileged Access	User Access	Description
000	No access	No access	No access
001	Read/Write	No access	Privileged access only
010	Read/Write	Read only	Write in a user program generates a fault
011	Read/Write	Read/Write	Full access
100	Unpredictable	Unpredictable	Unpredictable
101	Read only	No access	Privileged read only
110	Read only	Read only	Read only
111	Read only	Read only	Read only

The XN (Execute Never) field (bit [28]) decides whether an instruction fetch from this region is allowed. When this field is set to 1, all instructions fetched from this region will generate a memory management fault when they enter the execution stage.

The TEX, S, B, and C fields (bit [21:16]) are more complex. Despite the fact that the Cortex-M3 processor does not have cache, its implementation follows ARM v7-M architecture, which can support external cache and more advanced memory systems. Therefore, the region access properties can be programmed to support different types of memory management models.

In v6 and v7 architecture, the memory system can have two cache levels: inner cache and outer cache. They can have different caching policies. Since the Cortex-M3 processor itself does not have a cache controller, the cache policy only affects write buffering in the internal bus matrix and possibly the memory controller (see Table 13.8).

Table 13.8 [S] Indicates That Shareability Is Determined by the *S* Bit Field (Shared by Multiple Processors)

TEX	C	B	Description	Region Shareability
b000	0	0	Strongly ordered (transfers carry out and complete in programmed order)	Shareable
b000	0	1	Shared device (write can be buffered)	Shareable
b000	1	0	Outer and inner write-through; no write allocate	[S]
b000	1	1	Outer and inner write-back; no write allocate	[S]
b001	0	0	Outer and inner non-cacheable	[S]
b001	0	1	Reserved	Reserved
b001	1	0	Implementation defined	—
b001	1	1	Outer and inner write-back; write and read allocate	[S]
b010	0	0	Nonshared device	Not shared
b010	0	1	Reserved	Reserved
b010	1	X	Reserved	Reserved
b1BB	A	A	Cached memory; BB = outer policy, AA = inner policy	[S]

When TEX[2] is 1, the cache policy for outer cache and inner cache are as shown in Table 13.9.

Table 13.9 Encoding of Inner and Outer Cache Policy When Most Significant Bit of TEX is Set to 1

Memory Attribute Encoding (AA and BB)	Cache Policy
00	Noncacheable
01	Write back, write and read allocate
10	Write through, no write allocate
11	Write back, no write allocate

For more information on cache behavior and cache policy, refer to the *ARM Architecture Application Level Reference Manual* (Ref 2).

Setting Up the MPU

The MPU register might look complicated, but as long as you have a clear idea of the memory regions that are required for your application, it should not be difficult. Typically, you will need to have the following memory regions:

- Program code for privileged programs (for example, OS kernel and exception handlers)
- Program code for user programs
- Data memory for privileged programs within Code region (data + stack)
- Data memory for user programs within Code region (data + stack)
- Data memory for privileged and user programs in other memory regions (e.g., SRAM)
- System device region (usually privileged access only; for example, NVIC and MPU registers)
- Other peripherals

For Cortex-M3 products, most memory regions can be set up with TEX = b000, C = 1, B = 1. System devices such as the NVIC should be strongly ordered, and peripheral regions can be programmed as shared devices (TEX=b000, C= 0, B = 1). However, if you want to make sure that any bus faults occurring in the region are precise bus faults, you should use strong ordering (TEX = b000, C = 0, B = 0) so that write buffering is disabled. However, doing so can reduce system performance.

A simple flow for an MPU setup routine might look like the diagram shown in Figure 13.2.

Before the MPU is enabled and if the vector table is relocated to RAM, remember to set up the fault handler for the memory management fault in the vector table, and enable the memory management fault in the System Handler Control and State register. They are needed to allow the memory management fault handler to be executed if an MPU violation takes place.

For a simple case of only four required regions, a simple MPU setup code (without the region checking and enabling) might look like this:

```
LDR    R0,=0xE000ED98 ; Region number register
MOV    R1,#0          ; Select region 0
STR    R1, [R0]
LDR    R1,=0x00000000 ; Base Address = 0x00000000
STR    R1, [R0, #4]   ; MPU Region Base Address Register
LDR    R1,=0x0307002F ; R/W, TEX=0,S=1,C=1,B=1, 16MB, Enable=1
```

```
STR     R1, [R0, #8]    ; MPU Region Attribute and Size Register
MOV     R1,#1           ; Select region 1
STR     R1, [R0]
LDR     R1,=0x08000000 ; Base Address = 0x08000000
STR     R1, [R0, #4]    ; MPU Region Base Address Register
LDR     R1,=0x0307002B ; R/W, TEX=0,S=1,C=1,B=1, 4MB, Enable=1
STR     R1, [R0, #8]    ; MPU Region Attribute and Size Register
MOV     R1,#2           ; Select region 2
STR     R1, [R0]
```

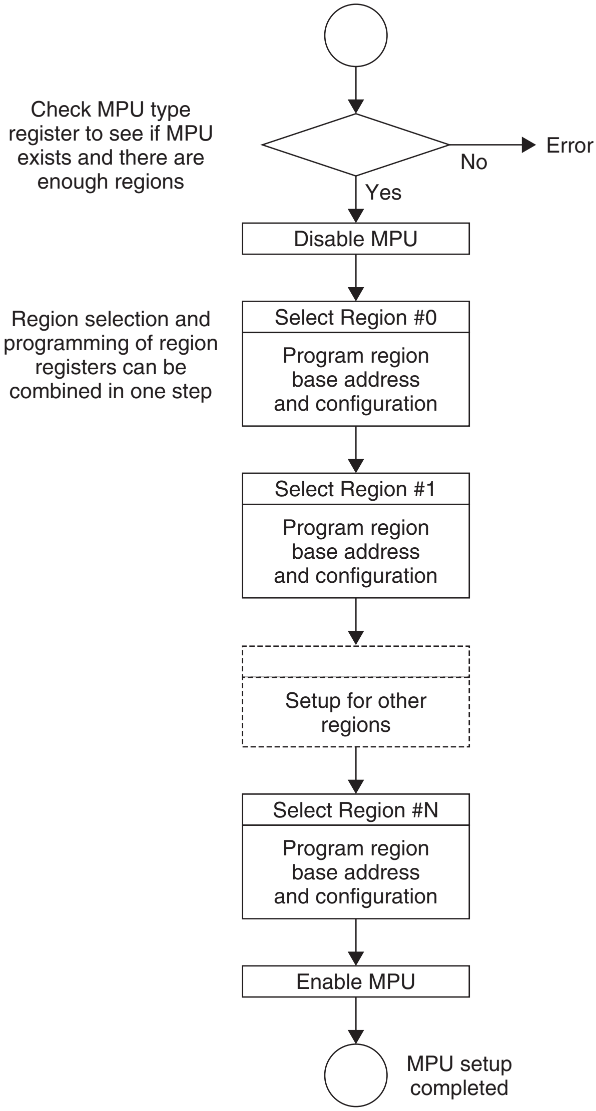

Check MPU type register to see if MPU exists and there are enough regions
No
Error
Yes
Disable MPU
Region selection and programming of region registers can be combined in one step
Select Region #0
Program region base address and configuration
Select Region #1
Program region base address and configuration
Setup for other regions
Select Region #N
Program region base address and configuration
Enable MPU
MPU setup completed

Figure 13.2 Example Steps to Set Up the MPU

```
LDR     R1,=0x40000000 ; Base Address = 0x40000000
STR     R1, [R0, #4]   ; MPU Region Base Address Register
LDR     R1,=0x03050039 ; R/W, TEX=0,S=1,C=0,B=1, 512MB, Enable=1
STR     R1, [R0, #8]   ; MPU Region Attribute and Size Register
MOV     R1,#3          ; Select region 3
STR     R1, [R0]
LDR     R1,=0xE0000000 ; Base Address = 0xE0000000
STR     R1, [R0, #4]   ; MPU Region Base Address Register
LDR     R1,=0x03040027 ; R/W, TEX=0,S=1,C=0,B=0, 1MB, Enable=1
STR     R1, [R0, #8]   ; MPU Region Attribute and Size Register
MOV     R1,#1          ; Enable MPU
STR     R1, [R0,#-4]   ; MPU Control register
                       ; (0xE000ED98-4=0xE000ED94)
```

This provides four regions:

- Privileged code: 0x00000000–0x00FFFFFF (16 MB), full access, cacheable
- Privileged data: 0x08000000–0x0803FFFF (4 MB), full access, cacheable
- Peripheral: 0x40000000–0x5FFFFFFF (0.5 GB), full access, shared device
- System control: 0xE0000000–0xE00FFFFF (1 MB), privileged access, strongly ordered, XN

By combining region selection and writing to the base address register, we can shorten the code to this:

```
LDR     R0,=0xE000ED9C ; Region Base Address register
LDR     R1,=0x00000010 ; Base Address = 0x00000000, region 0,
                       ; valid=1
STR     R1, [R0, #0]   ; MPU Region Base Address Register
LDR     R1,=0x0307002F ; R/W, TEX=0,S=1,C=1,B=1, 16MB, Enable=1
STR     R1, [R0, #4]   ; MPU Region Attribute and Size Register
LDR     R1,=0x08000011 ; Base Address = 0x08000000, region 1,
                       ; valid=1
STR     R1, [R0, #0]   ; MPU Region Base Address Register
LDR     R1,=0x0307002B ; R/W, TEX=0,S=1,C=1,B=1, 4MB, Enable=1
STR     R1, [R0, #4]   ; MPU Region Attribute and Size Register
LDR     R1,=0x40000012 ; Base Address = 0x40000000, region 2,
                       ; valid=1
STR     R1, [R0, #0]   ; MPU Region Base Address Register
LDR     R1,=0x03050039 ; R/W, TEX=0,S=1,C=0,B=1, 512MB, Enable=1
STR     R1, [R0, #4]   ; MPU Region Attribute and Size Register
LDR     R1,=0xE0000013 ; Base Address = 0xE0000000, region 3,
                       ; valid=1
STR     R1, [R0, #0]   ; MPU Region Base Address Register
LDR     R1,=0x03040027 ; R/W, TEX=0,S=1,C=0,B=0, 1MB, Enable=1
```

```
      STR    R1, [R0, #4]    ; MPU Region Attribute and Size Register
      MOV    R1,#1           ; Enable MPU
      STR    R1, [R0,#-8]    ; MPU Control register
                             ; (0xE000ED9C-8=0xE000ED94)
```

We've shortened the code quite a bit. However, you can make further enhancements to create even faster setup code. This is done using MPU aliased register addresses (see Table D.33 in Appendix D). The aliased register addresses follow the MPU Region Attribute and Size registers and are aliased to the MPU Base Address register and the MPU Region Attribute and Size register. They produce a continuous address of 8 words, making it possible to use load/store multiple (LDM and STM) instructions:

```
      LDR    R0,=0xE000ED9C   ; Region Base Address register
      LDR    R1,=MPUconfig  ; Table of predefined MPU setup variables
      LDMIA  R1!, {R2, R3, R4, R5}  ; Read 4 words from table
      STMIA  R0!, {R2, R3, R4, R5}  ; write 4 words to MPU
      LDMIA  R1!, {R2, R3, R4, R5}  ; Read next 4 words from table
      STMIA  R0!, {R2, R3, R4, R5}  ; write next 4 words to MPU
      B      MPUconfigEnd
      ALIGN  4     ; This is needed to make sure the following table
                   ; is word aligned
MPUconfig          ; so that we can use load multiple instruction
      DCD     0x00000010 ; Base Address = 0x00000000, region 0,
                         ; valid=1
      DCD     0x0307002F ; R/W, TEX=0,S=1,C=1,B=1, 16MB, Enable=1
      DCD     0x08000011 ; Base Address = 0x08000000, region 1,
                         ; valid=1
      DCD     0x0307002B ; R/W, TEX=0,S=1,C=1,B=1, 4MB, Enable=1
      DCD     0x40000012 ; Base Address = 0x40000000, region 2,
                         ; valid=1
      DCD     0x03050039 ; R/W, TEX=0,S=1,C=0,B=1, 512MB, Enable=1
      DCD     0xE0000013 ; Base Address = 0xE0000000, region 3,
                         ; valid=1
      DCD     0x03040027 ; R/W, TEX=0,S=1,C=0,B=0, 1MB, Enable=1
MPUconfigEnd
      LDR    R0,=0xE000ED94 ; MPU Control register
      MOV    R1,#1          ; Enable MPU
      STR    R1, [R0]
```

This solution, of course, can be used only if all the required information is known beforehand. Otherwise, a more generic approach has to be used. One way to handle this is to use a subroutine (*MpuRegionSetup*) that can set up a region based on a number of input parameters and then call it several times to set up different regions:

```
MpuSetup   ; A subroutine to setup the MPU by calling subroutines that
           ; setup regions
      PUSH  {R0-R6,LR}
      LDR   R0,=0xE000ED94 ; MPU Control Register
      MOV   R1,#0
      STR   R1,[R0]        ; Disable MPU
      ; --- Region #0 ---
      LDR   R0,=0x00000000 ; Region 0: Base Address  = 0x00000000
      MOV   R1,#0x0        ; Region 0: Region number = 0
      MOV   R2,#0x17       ; Region 0: Size          = 0x17 (16MB)
      MOV   R3,#0x3        ; Region 0: AP            = 0x3 (full
                                                            access)
      MOV   R4,#0x7        ; Region 0: MemAttrib     = 0x7
      MOV   R5,#0x0        ; Region 0: Sub R disable = 0
      MOV   R6,#0x1        ; Region 0: {XN, Enable}  = 0,1
      BL    MpuRegionSetup
      ; --- Region #1 ---
      LDR   R0,=0x08000000 ; Region 1: Base Address  = 0x08000000
      MOV   R1,#0x1        ; Region 1: Region number = 1
      MOV   R2,#0x15       ; Region 1: Size          = 0x15 (4MB)
      MOV   R3,#0x3        ; Region 1: AP            = 0x3 (full
                                                            access)
      MOV   R4,#0x7        ; Region 1: MemAttrib     = 0x7
      MOV   R5,#0x0        ; Region 1: Sub R disable = 0
      MOV   R6,#0x1        ; Region 1: {XN, Enable}  = 0,1
      BL    MpuRegionSetup
      ...                  ; setup for region #2 and #3
      ; --- Region #4-#7 Disable ---
      MOV   R0,#4
      BL    MpuRegionDisable
      MOV   R0,#5
      BL    MpuRegionDisable
      MOV   R0,#6
      BL    MpuRegionDisable
      MOV   R0,#7
      BL    MpuRegionDisable
      LDR   R0,=0xE000ED94 ; MPU Control Register
      MOV   R1,#1
      STR   R1,[R0]        ; Enable MPU
      POP  {R0-R6,PC}      ; Return


MpuRegionSetup
      ; MPU region setup subroutine
      ; Input R0 : Base Address
      ;       R1 : Region number
      ;       R2 : Size
      ;       R3 : AP (access permission)
      ;       R4 : MemAttrib ({TEX[2:0], S, C, B})
      ;       R5 : Sub region disable
      ;      R6 : {XN,Enable}
      PUSH  {R0-R1, LR}
      BIC   R0, R0, #0x1F   ; Clear unused bits in address
      BFI   R0, R1, #0, #4  ; Insert region number to R0[3:0]
      ORR   R0, R0, #0x10   ; Set valid bit
      LDR   R1,=0xE000ED9C  ; MPU Region Base Address Register
      STR   R0,[R1]         ; Set base address reg

      AND   R0, R6, #0x01   ; Get Enable bit
      UBFX  R1, R6, #1, #1  ; Get XN bit
      BFI   R0, R1, #28, #1 ; Insert XN to R0[28]
      BFI   R0, R2, #1 , #5 ; Insert Region Size field (R2[4:0]) to
                            ; R0[5:1]
      BFI   R0, R3, #24, #3 ; Insert AP fields (R3[2:0]) to R0[26:24]
      BFI   R0, R4, #16, #6 ; Insert memattrib field (R4[5:0]) to
                            ; R0[21:16]
      BFI   R0, R5, #8,  #8 ; Insert subregion disable (SRD) fields to
                            ; R0[15:8]
      LDR   R1,=0xE000EDA0  ; MPU Region Base Size and Attribute
                            ; Register
      STR   R0,[R1]         ; Set base attribute and size reg
      POP   {R0-R1, PC}     ; Return

MpuRegionDisable
      ; Subroutine to disable unused region
      ; Input R0 : Region number
      PUSH  {R1, LR}
      AND   R0, R0, #0xF    ; Clear unused bits in Region Number
      ORR   R0, R0, #0x10   ; Set valid bit
      LDR   R1,=0xE000ED9C  ; MPU Region Base Address Register
      STR   R0,[R1]
      MOV   R0, #0
      LDR   R1,=0xE000EDA0  ; MPU Region Base Size and Attribute
                            ; Register
      STR   R0,[R1]         ; Set base attribute and size reg to 0
                            ; (disabled)
      POP   {R1, PC}        ; Return
```

In this example, we included a subroutine that is used to disable a region that is not used. This is necessary if you do not know whether a region has been programmed previously. If an unused region is previously programmed to be enabled, it needs to be disabled so that it doesn't affect the new configuration.

In addition, the example shows the application of the Bit Field Insert (BFI) instruction in the Cortex-M3. This can greatly simplify bit-field merging operations.

Typical Setup

In typical applications, the MPU is used when there is a need to prevent user programs from accessing privileged process data and program regions. When developing the setup routine for the MPU, you need to consider a number of regions:

1. Code region:
    - Privileged code, including a starting vector table
    - User code
2. SRAM region:
    - Privileged data, including the main stack
    - User data, including the process stack
    - Privileged bit-band alias region
    - User bit-band alias region
3. Peripherals:
    - Privileged peripherals
    - User peripherals
    - Privileged peripheral bit-band alias region
    - User peripheral bit-band alias region
4. System Control Space (NVIC and debug components):
    - Privileged accesses only

From this list we have identified 11 regions, more than the eight regions supported by the Cortex-M3 MPU. However, we can define the privileged regions by means of a background region (PRIVDEFENA set to 1), so there are only five user regions to set up, leaving three spare MPU regions. The unused regions might still be used for setting up additional regions in external memory, to protect read-only data, or to completely block some part of the memory if necessary.

Example Use of the Subregion Disable

In some cases we might have some peripherals accessible by user programs, and a few should be protected to be privileged accesses only, resulting in fragmentation of user-accessible peripheral memory space. In this kind of scenario, we could do one of these things:

- Define multiple user regions
- Define privileged regions inside the user peripheral region
- Use subregion disable within the user region

The first two methods can use up available regions very easily. With the third solution, using the subregion disable feature, we can easily set up access permission to separate peripheral blocks without using extra regions. For example:

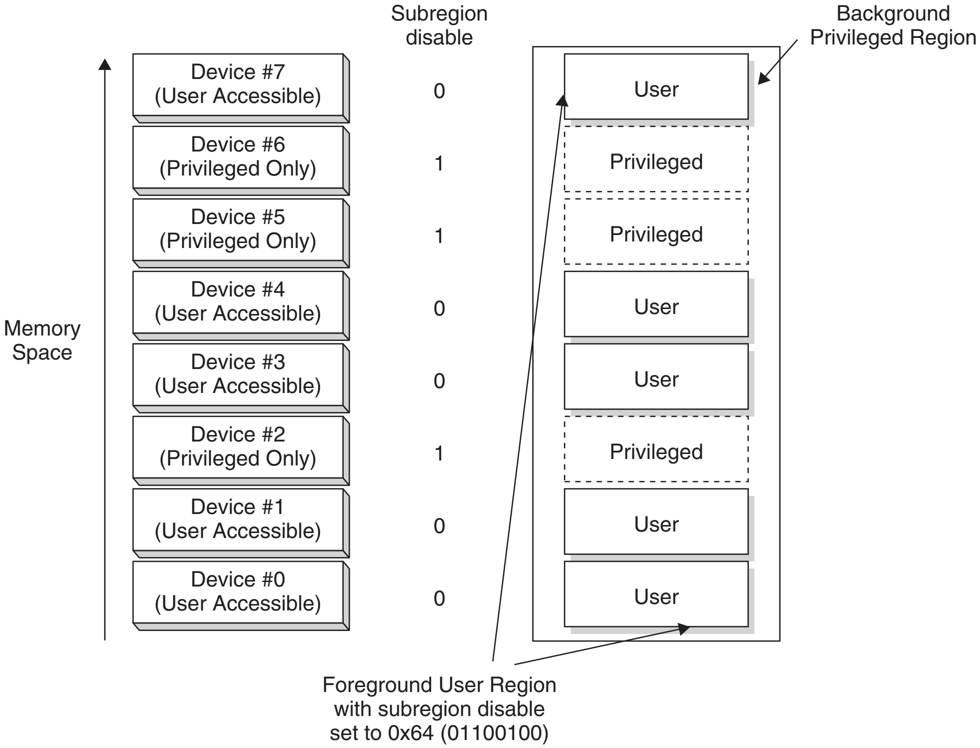

Subregion disable
Background Privileged Region
Memory Space
Device #7 (User Accessible)
Device #6 (Privileged Only)
Device #5 (Privileged Only)
Device #4 (User Accessible)
Device #3 (User Accessible)
Device #2 (Privileged Only)
Device #1 (User Accessible)
Device #0 (User Accessible)
0
1
1
0
0
1
0
0
User
Privileged
Privileged
User
User
Privileged
User
User
Foreground User Region with subregion disable set to 0x64 (01100100)

Figure 13.3 Using Subregion Disable to Control Access Rights to Separated Peripherals

The same techniques can be applied to memory regions as well. However, it is more likely that peripherals will have a fragmented privilege setup.

Let's assume that the memory regions in Table 13.10 will be used. After the required regions are defined, we can create the MPU setup code. To make the code easier to understand

and modify, we used the function we created earlier to develop the completed MPU setup example:

```
MpuSetup  ; A subroutine to setup the MPU by calling subroutines that
          ; setup regions
      PUSH  {R0-R6,LR}
      LDR   R0,=0xE000ED94 ; MPU Control Register
      MOV   R1,#0
      STR   R1,[R0]        ; Disable MPU
      ; --- Region #0 ---    User program
      LDR   R0,=0x00004000 ; Region 0: Base Address  = 0x00004000
      MOV   R1,#0x0        ; Region 0: Region number = 0
      MOV   R2,#0x0D       ; Region 0: Size          = 0x0D (16KB)
      MOV   R3,#0x3        ; Region 0: AP            = 0x3 (full
                                                          access)
      MOV   R4,#0x2        ; Region 0: MemAttrib     = 0x2 (TEX=0,
                                                          S=0, C=1,
                                                          B=0)
      MOV   R5,#0x0        ; Region 0: Sub R disable = 0
      MOV   R6,#0x1        ; Region 0: {XN, Enable}  = 0,1
      BL    MpuRegionSetup
      ; --- Region #1 ---    User data
      LDR   R0,=0x20000000 ; Region 1: Base Address  = 0x20000000
      MOV   R1,#0x1        ; Region 1: Region number = 1
      MOV   R2,#0x0B       ; Region 1: Size          = 0x0B (4KB)
      MOV   R3,#0x3        ; Region 1: AP            = 0x3 (full
                                                          access)
      MOV   R4,#0xB        ; Region 1: MemAttrib     = 0xB (TEX=1,
                                                          S=0, C=1,
                                                          B=1)
      MOV   R5,#0x0        ; Region 1: Sub R disable = 0
      MOV   R6,#0x1        ; Region 1: {XN, Enable}  = 0,1
      BL    MpuRegionSetup
      ; --- Region #2 ---    User bit band
      LDR   R0,=0x22000000 ; Region 2: Base Address  = 0x22000000
      MOV   R1,#0x2        ; Region 2: Region number = 2
      MOV   R2,#0x10       ; Region 2: Size          = 0x10 (128KB)
      MOV   R3,#0x3        ; Region 2: AP            = 0x3 (full
                                                          access)
      MOV   R4,#0xB        ; Region 2: MemAttrib     = 0xB (TEX=1,
                                                          S=0, C=1,
                                                          B=1)
      MOV   R5,#0x0        ; Region 2: Sub R disable = 0
      MOV   R6,#0x1        ; Region 2: {XN, Enable}  = 0,1
      BL    MpuRegionSetup
      ; --- Region #3 ---    User Peripherals
```

```
LDR   R0,=0x40000000 ; Region 3: Base Address  = 0x40000000
MOV   R1,#0x3        ; Region 3: Region number = 3
MOV   R2,#0x13       ; Region 3: Size          = 0x13 (1MB)
MOV   R3,#0x3        ; Region 3: AP            = 0x3 (full
                                                      access)
MOV   R4,#0x1        ; Region 3: MemAttrib     = 0x1 (TEX=0,
                                                      S=0,C=0,B=1)
MOV   R5,#0x9B       ; Region 3: Sub R disable = 0x9B (from
                                                       previous
                                                       example)
MOV   R6,#0x3        ; Region 3: {XN, Enable}  = 1,1
BL    MpuRegionSetup
; --- Region #4 ---    User peripheral bit band
LDR   R0,=0x42000000 ; Region 4: Base Address  = 0x42000000
MOV   R1,#0x4        ; Region 4: Region number = 4
MOV   R2,#0x18       ; Region 4: Size          = 0x18 (32MB)
MOV   R3,#0x3        ; Region 4: AP            = 0x3 (full
                                                      access)
MOV   R4,#0x1        ; Region 4: MemAttrib     = 0x1 (TEX=0,
                                                      S=0, C=0,
                                                      B=1)
MOV   R5,#0x9B       ; Region 4: Sub R disable = 0x64 (from
                                                       previous
                                                       example)
MOV   R6,#0x3        ; Region 4: {XN, Enable}  = 1,1
BL    MpuRegionSetup
; --- Region #5 ---    External RAM
LDR   R0,=0x60000000 ; Region 5: Base Address  = 0x60000000
MOV   R1,#0x5        ; Region 5: Region number = 5
MOV   R2,#0x17       ; Region 5: Size          = 0x17 (16MB)
MOV   R3,#0x3        ; Region 5: AP            = 0x3 (full
                                                      access)
MOV   R4,#0xB        ; Region 5: MemAttrib     = 0xB (TEX=0,
                                                      S=0,C=1,B=1)
MOV   R5,#0x0        ; Region 5: Sub R disable = 0
MOV   R6,#0x1        ; Region 5: {XN, Enable}  = 0,1
BL    MpuRegionSetup
; --- Region #6 ---    Not used, make sure it is disabled
MOV   R0,#6
BL    MpuRegionDisable
; --- Region #7 ---    Not used, make sure it is disabled
MOV   R0,#7
BL    MpuRegionDisable
LDR   R0,=0xE000ED94 ; MPU Control Register
MOV   R1,#5
```

```
STR    R1,[R0]          ; Enable MPU with Privileged Default
                        ; memory map enabled
POP    {R0-R6,PC}
```

Table 13.10 Memory Region Arrangement for MPU Setup Example Code

Address	Description	Size	Type	Memory Attributes (C, B, A, S, XN)	MPU Region
0x00000000–0x00003FFF	Privileged program	16 k	Read only	C, –, A, –, –	Background
0x00004000–0x00007FFF	User program	16 k	Read only	C, –, A, –, –	Region #0
0x20000000–0x20000FFF	User data	4 k	Full access	C, B, A, –, –	Region #1
0x20001000–0x20001FFF	Privileged data	4 k	Privileged accesses	C, B, A, –, –	Background
0x22000000–0x2201FFFF	User data bit-band alias	128 k	Full access	C, B, A, –, –	Region #2
0x22020000–0x2203FFFF	Privileged data bit-band alias	128 k	Full access	C, B, A, –, –	Background
0x40000000–0x400FFFFF	User peripherals	1 M	Full access	–, B, –, –, XN	Region #3
0x40040000–0x4005FFFF	Privileged peripherals within user peripheral region	128 k	Privileged accesses	–, B, –, –, XN	Disabled subregions in Region #3
0x42000000–0x43FFFFFF	User peripherals bit-band alias	32 M	Full access	–, B, –, –, XN	Region #4
0x42800000–0x42BFFFFF	Privileged peripherals bit-band alias within user region	4 M	Privileged accesses	–, B, –, –, XN	Disabled subregion in Region #4
0x60000000–0x60FFFFFF	External RAM	16 M	Full access	C, B, A, –, –	Region #5
0xE0000000 – 0xF00FFFFF	NVIC, debug, and Private Peripheral Bus	1 M	Privileged accesses	–, –, –, –, XN	Background

This page intentionally left blank

CHAPTER 14

Other Cortex-M3 Features

In This Chapter:

- The SYSTICK Timer
- Power Management
- Multiprocessor Communication
- Self-Reset Control

The SYSTICK Timer

The SYSTICK register in the NVIC was covered briefly in Chapter 8. As we saw, the SYSTICK timer is a 24-bit down counter. Once it reaches zero, the counter loads the reload value from the RELOAD register. It does not stop until the enable bit in the SYSTICK Control and Status register is cleared.

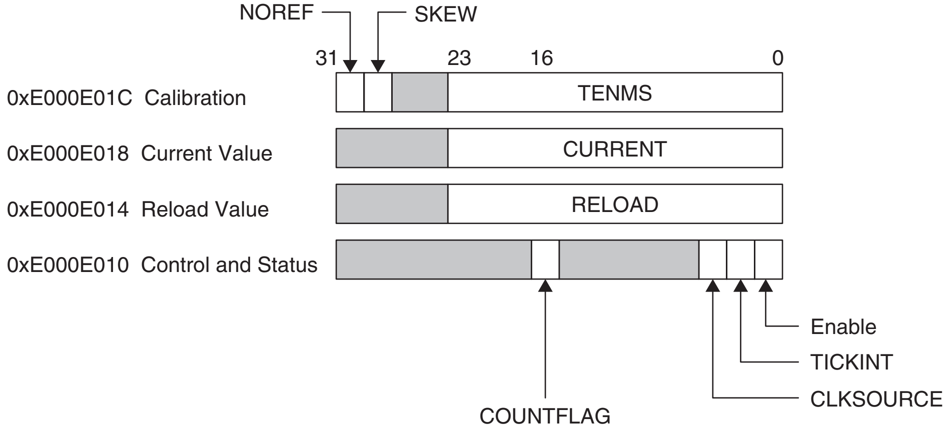


Figure 14.1 SYSTICK Registers in the NVIC

The Cortex-M3 processor allows two different clock sources for the SYSTICK counter. The first one is the core free-running clock (not from the system clock HCLK, so it does not stop when the system clock is stopped). The second one is an external reference clock. This clock signal must be at least two times slower than the free-running clock because this signal is sampled by the free-running clock. Because a chip designer might decide to omit this external reference clock in the design, it might not be available. To determine whether the external clock source is available, you should check bit[31] of the SYSTICK Calibration register. The chip designer should connect this pin to an appropriate value base on the design.

When the SYSTICK timer changes from 1 to 0, it will set the COUNTFLAG bit in the SYSTICK Control and Status register. The COUNTFLAG can be cleared by one of the following:

- Read of the SYSTICK Control and Status register by the processor
- Clear of the SYSTICK counter value by writing any value to the SYSTICK Current Value register

The SYSTICK counter can be used to generate SYSTICK exceptions at regular intervals. This is often necessary for the OS, for task and resources management. To enable SYSTICK exception generation, the TICKINT bit should be set. In addition, if the vector table has been relocated to SRAM, it would be necessary to set up the SYSTICK exception handler in the vector table:

```
; Setup SYSTICK exception handler
MOV      R0,#0xF          ; Exception type 15
LDR      R1,=systick_handler ; address of exception handler
LDR      R2,=0xE000ED08    ; Vector table offset register
LDR      R2,[R2]
STR      R1, [R2, R0, LSL #2]  ; Write vector to
                               ; VectTblOffset+ExcpType*4
```

A simple code to set up the SYSTICK could be:

```
; Enable SYSTICK timer operation and enable SYSTICK interrupt
LDR    R0,=0xE000E010   ; SYSTICK control and status register
MOV    R1, #0
STR    R1, [R0]         ; Stop counter to prevent interrupt
                        ; triggered accidentally
LDR    R1,=0x3FF        ; Trigger every 1024 cycles (since
                        ; counter decrement from
                        ; 1023 to 0,total of 1024 cycles, the
                        ; value 0x3FF is used).
STR    R1, [R0,#4]      ; Write reload value to reload register
                        ; address
STR    R1, [R0,#8]      ; Write any value to current value
                        ; register to clear
                        ; current value to 0 and clear COUNTFLAG
```

```
    MOV    R1, #0x7          ; Clock source = core clock, Enable
                             ; Interrupt, Enable
                             ; SYSTICK counter
    STR    R1, [R0]          ; Start counter
```

The SYSTICK counter provides a simple way to allow timing calibration information to be accessed. The top level of the Cortex-M3 processor has a 24-bit input to which a chip designer can input a reload value that can be used to generate a 10 ms time interval. This value can be accessed by the SYSTICK calibration register. However, this option is not necessarily available, so you'll need to check the device's datasheet to see if you can use this feature.

The SYSTICK counter can also be used as an alarm timer that starts a certain task after a number of clock cycles. For example, if a task has to be started to execute after 300 clock cycles, we could set up the task at the SYSTICK exception handler and program the SYSTICK timer so that the task will be executed when the 300 cycle count is reached:

```
      LDR   r0,=15             ; Setup SYSTICK handler
      LDR   r1,=SysTickAlarm ; SYSTICK Exception handler name
      BL    SetupExcpHandler

      LDR   R0,=0xE000E010     ; SYSTICK base
      MOV   R1, #0             ; Disable SYSTICK during programming
      STR   R1, [R0]
      STR   R1, [R0,#0x8]      ; Clear current value
      LDR   R1, =(300-12)      ; Set Reload value : Minus 12 because of
                               ; exception latency
      STR   R1,[R0,#0x4]

      LDR   R4,=SysTickFired ; A data variable in RAM
      MOV   R5, #0             ; Setup the software flag to zero
      STR   R5, [R4]
      MOV   R1, #0x7           ; Use internal clock, enable SYSTICK
                               ; exception,
      STR   R1, [R0]           ; Start counting

      LDR   R4,=SysTickFired

WaitLoop
      LDR   R5, [R4]   ; Wait until Software flag is set by SYSTICK
                       ; handler
      CMP   R5, #0
      BEQ   WaitLoop
      ...                      ; SysTickFired set, main program
                               ; continue on other tasks
```

The example code on the last page uses a subroutine called SetupExcpHandler to set the SYSTICK vector. This is used only when the vector table is writable (for example, relocated to SRAM):

```
SetupExcpHandler     ; Subroutine for setting exception vector
      ; Input R0 : Exception number
      ;       R1 : Exception vector
      PUSH  {R2, LR}
      LDR   R2,=0xE000ED08   ; Vector Table offset
      LDR   R2, [R2]
      STR   R1, [R2, R0, LSL #2] ; Address = vector table offset + 4
                                 ; x Exception number
      POP   {R2, PC}
```

The counter starts with an initial value of zero because it was manually cleared from the main program. It then immediately reloads to 288 (300 – 12). We subtract 12 from the count because this is the number of clock cycles for minimum exception latency. However, if another exception with the same or a higher priority is running when the SYSTICK counter reaches zero, the start of the exception could be delayed.

Note that the subtraction of 12 cycles from the reload value in this example is required for only one-shot alarm timer usage. For periodic counting usage, the reload value should be the number of clock cycles per period minus one.

Since the SYSTICK counter does not stop automatically, we need to stop it within the SYSTICK handler. Furthermore, there's a chance that the SYSTICK exception could have been pended again if it was delayed by processing of other exceptions, so a number of steps must be carried out if the SYSTICK exception is a one-off processing:

```
SysTickAlarm                  ; SYSTICK exception handler
      PUSH  {LR}
      LDR   R0,=0xE000E010    ; SYSTICK base
      MOV   R1, #0
      STR   R1,[R0]           ; Disable further SYSTICK exception
      LDR   R0,=0xE000ED04
      LDR   R1,=0x02000000    ; Clear SYSTICK pend bit in case it has
                              ; been pended again
      STR   R1, [R0]
      ...                     ; Execute required processing task
      LDR   R2,=SysTickFired  ; Setup software flag so that main
                              ; program knows tasks
      LDR   R1, [R2]          ; has been carried out.
      ADD   R1, #1
      STR   R1, [R2]
      POP   {PC}  ; Exception return
```

In the final step of the SYSTICK exception handler, we set a software variable called *SysTickFired* so that the main program knows the required task has been carried out.

Power Management

The Cortex-M3 provides sleep modes as a power management feature. During sleep mode, the system clock can be stopped, but the free-running clock input should still be running to allow the processor to be woken by an interrupt. The two sleep modes are:

- Sleep: Indicated by the SLEEPING signal from the Cortex-M3 processor
- Deep sleep: Indicated by the SLEEPDEEP signal from the Cortex-M3 processor

To decide which sleep mode will be used, the NVIC System Control register has a bit field called SLEEPDEEP (see Table 14.1). The actions of SLEEPING and SLEEPDEEP depend on the particular MCU implementation. In some implementations, the action will be the same in both cases.

Table 14.1 System Control Register (0xE000ED10)

Bits	Name	Type	Reset Value	Description
4	SEVONPEND	R/W	0	Send Event on Pending; wakes up from WFE if a new interrupt is pended, regardless of whether the interrupt has priority higher than the current level
3	Reserved	–	–	–
2	SLEEPDEEP	R/W	0	Enable SLEEPDEEP output signal when entering sleep mode
1	SLEEPONEXIT	R/W	0	Enable SleeponExit feature
0	Reserved	–	–	–

The sleep modes are invoked by WFI or WFE instructions. WFI stands for *Wait-For-Interrupt*, and WFE stands for *Wait-For-Events*. Events can be interrupts, a previously triggered interrupt, or an external event signal pulse via the RXEV signal. Inside the processor there is a latch for events, so a past event can wake up a processor from WFE.

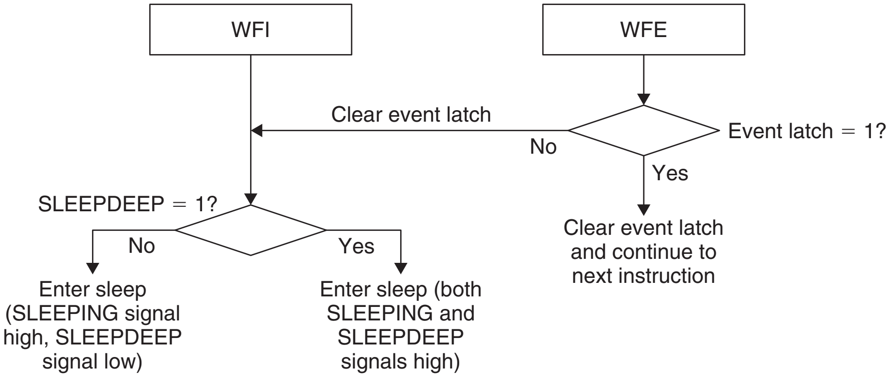

WFI
WFE
Clear event latch
No
Event latch = 1?
Yes
SLEEPDEEP = 1?
No
Yes
Clear event latch and continue to next instruction
Enter sleep (SLEEPING signal high, SLEEPDEEP signal low)
Enter sleep (both SLEEPING and SLEEPDEEP signals high)

Figure 14.2 Sleep Operations

What exactly happens when the processor enters sleep mode depends on the chip design. The common case is that some of the clock signals can be stopped to reduce power consumption. However, the chip can also be designed to shut down part of the chip to further reduce power, or it is also possible that a design can shut down the chip completely and all the clock signals will be stopped. In a case where the chip is shut down completely, the only way to wake the system from sleep is via a system reset.

To wake the processor from WFI sleep, the interrupt will have to be higher priority than the current priority level (if it is an executing interrupt) and higher than the level set by the BASEPRI register or mask registers (PRIMASK and FAULTMASK). If an interrupt is not going to be accepted due to priority level, it will not wake up a sleep caused by WFI.

The situation for WFE is slightly different. If the interrupt triggered during sleep has lower or equal priority than the mask registers or BASEPRI registers and if the SEVONPEND is set, it could still wake the processor from sleep. The rules of waking the Cortex-M3 processor from sleep modes are summarized in Table 14.2.

Table 14.2 WFI and WFE Wake Up Behavior

WFI Behavior	Wake Up	IRQ Execution
`IRQ with BASEPRI`		
`  IRQ priority >  BASEPRI`	`Y`	`Y`
`  IRQ priority =< BASEPRI`	`N`	`N`
`IRQ with BASEPRI and PRIMASK`		
`  IRQ priority >  BASEPRI`	`Y`	`N`
`  IRQ priority =< BASEPRI`	`N`	`N`
WFE Behavior	**Wake Up**	**IRQ Execution**
`IRQ with BASEPRI, SEVONPEND = 0`		
`  IRQ priority >  BASEPRI`	`Y`	`Y`
`  IRQ priority =< BASEPRI`	`N`	`N`
`IRQ with BASEPRI, SEVONPEND = 1`		
`  IRQ priority >  BASEPRI`	`Y`	`Y`
`  IRQ priority =< BASEPRI`	`Y`	`N`
`IRQ with BASEPRI and PRIMASK, SEVONPEND = 0`		
`  IRQ priority >  BASEPRI`	`N`	`N`
`  IRQ priority =< BASEPRI`	`N`	`N`
`IRQ with BASEPRI & PRIMASK, SEVONPEND = 1`		
`  IRQ priority >  BASEPRI`	`Y`	`N`
`  IRQ priority =< BASEPRI`	`Y`	`N`

Another feature of sleep mode is that it can be programmed to go back to sleep automatically after the interrupt routine exit. In this way we can make the core sleep all the time unless an

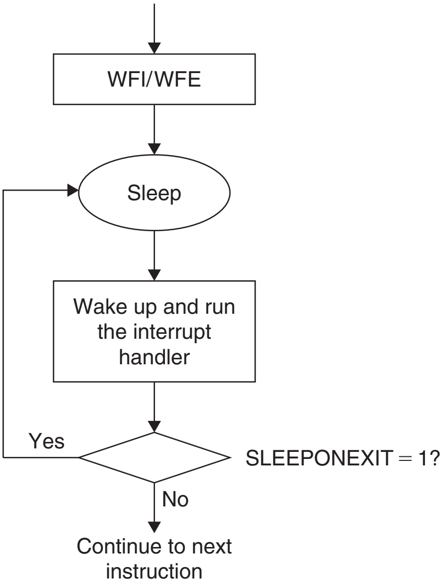

WFI/WFE
Sleep
Wake up and run the interrupt handler
Yes
SLEEPONEXIT = 1?
No
Continue to next instruction

Figure 14.3 Example Use of the SleepOnExit Feature

interrupt needs to be served. To use this feature, we need to set the SLEEPONEXIT bit in the System Control register.

Multiprocessor Communication

The Cortex-M3 comes with a simple multiprocessor communication interface for synchronizing tasks. The processor has one output signal, called TXEV (Transmit Event), for sending out events and an input signal, called RXEV (Receive Event), for receiving events. For a system with two processors, the event communication signal connection can be implemented as shown in Figure 14.4.

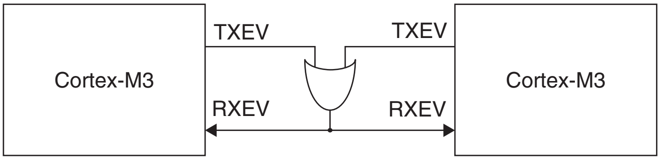


Figure 14.4 Event Communication Connection in a Two-Processor System

As mentioned in the previous section on Power Management, the processor can enter sleep when the WFE instruction is executed and continue the instruction execution when an external event is received. If we use an instruction call SEV (Send Event), one processor can wake up another processor that is in sleep mode and make sure both processors start executing a task at the same time.

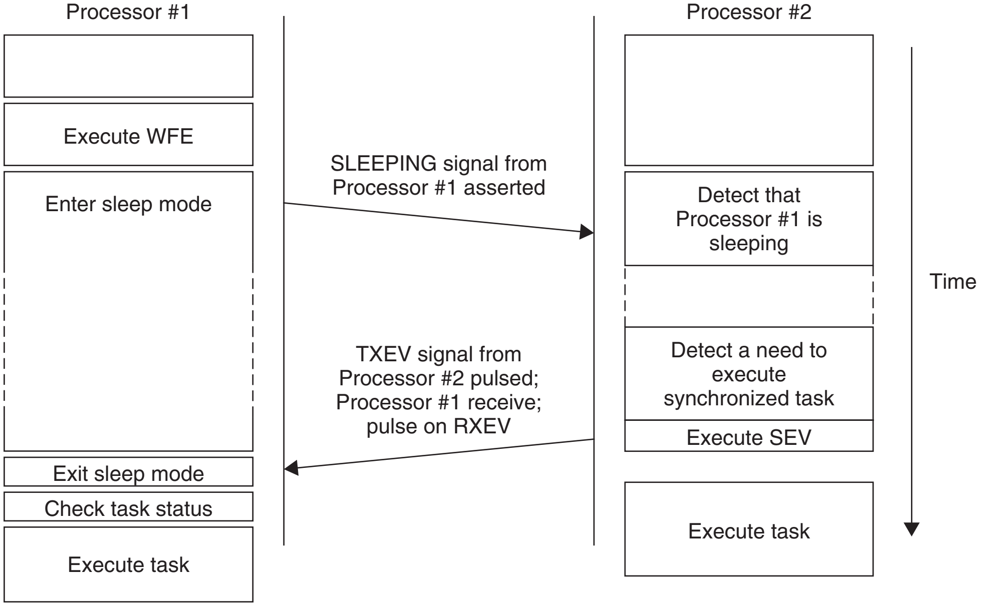

Processor #1
Processor #2
Execute WFE
Enter sleep mode
Exit sleep mode
Check task status
Execute task
SLEEPING signal from Processor #1 asserted
TXEV signal from Processor #2 pulsed; Processor #1 receive; pulse on RXEV
Detect that Processor #1 is sleeping
Detect a need to execute synchronized task
Execute SEV
Execute task
Time

Figure 14.5 Using Event Signals to Synchronize Tasks

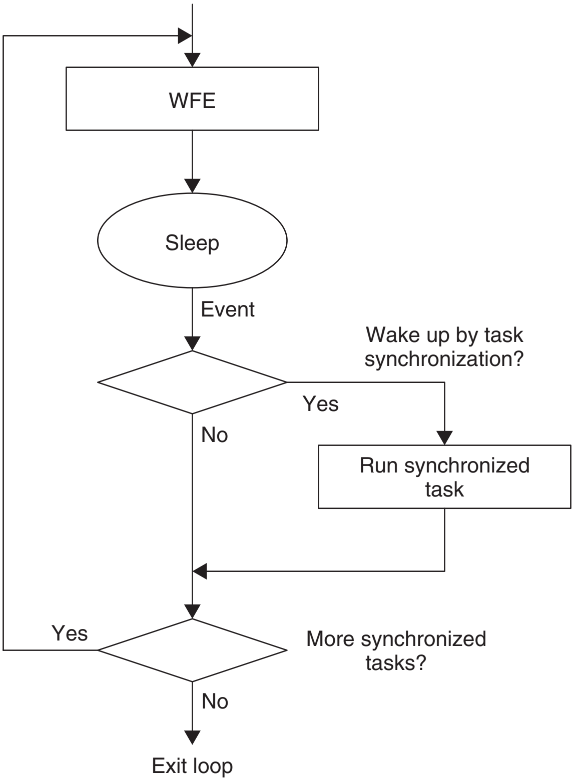


Figure 14.6 Example Use of the WFE Feature

Using this feature, we can make both processors start executing a task at the same time (possibly with a couple of clock cycles' difference, depending on actual chip implementation). The number of processors invoked can be any number, but it requires that one processor act as a master to generate the event pulse to other processors.

When the WFE instruction is executed, it first checks the local event latch. If the latch is not set, the core will enter sleep mode. If the latch is set, it will be cleared and the instruction execution will continue without entering sleep mode. The local event latch can be set by previously occurring exceptions and by the SEV instruction. So, if you execute an SEV and then execute a WFE, the processor will not enter sleep and will simply continue on to the next instruction, with the event latch cleared by WFE.

It is also important to note that the processor could also be woken by other events, such as interrupt and debugging events. Therefore, before starting the required synchronized task it is often necessary to check whether the wake-up was caused by task synchronization.

In most Cortex-M3 based products, there will be only one processor, and the RXEV input is likely tied to 0.

Self-Reset Control

The Cortex-M3 provides two self-reset control features. The first one is the VECTRESET control bit in the NVIC Application Interrupt and Reset Control register (bit [0]):

```
    LDR    R0,=0xE000ED0C   ; NVIC AIRCR address
    LDR    R1,=0x05FA0001   ; Set VECTRESET bit (05FA is a write
                            ; access key)
    STR    R1,[R0]

deadloop

    B      deadloop         ; a deadloop is used to ensure no other
                            ; instructions
                            ; follow the reset is executed
```

Writing to this bit will reset the Cortex-M3 processor, excluding the debug logic. This does not reset any circuit outside the Cortex-M3 processor. For example, if the SoC contains a UART, writing to this bit does not reset the UART or any peripherals outside the Cortex-M3.

The second reset feature is the SYSRESETREQ bit in the same NVIC register. It allows the Cortex-M3 processor to assert a reset request signal to the system's reset generator. Since the system reset generator is not part of Cortex-M3 design, the implementation of this reset feature depends on the chip design. Therefore, it is necessary to carefully check the chip's specification because this feature might not exist in some chips!

Here's an example code using the SYSRESTREQ:

```
    LDR    R0,=0xE000ED0C   ; NVIC AIRCR address
    LDR    R1,=0x05FA0004   ; Set SYSRESETREQ bit (05FA is a write
                            ; access key)
    STR    R1,[R0]

deadloop

    B      deadloop         ; a deadloop is used to ensure no other
                            ; instructions
                            ; follow the reset is executed
```

In most cases, when the SYSRESETREQ bit is set, the system reset signal of the Cortex-M3 processor (SYSRESETn) will be asserted by the reset generator. Depending on the chip design, it might or might not reset the other parts of the chip, such as peripherals. Normally this should not reset the debug logic of the Cortex-M3.

Note that the delay from assertion of SYSRESETREQ to actual reset from the reset generator can also be an issue. Due to delay in the reset generator, you might find the processor still accepting interrupts after the reset request is set. If you want to stop the core from accepting interrupts before running this code, you can set the FAULTMASK using the MSR instruction.

CHAPTER 15

Debug Architecture

In This Chapter:

- Debugging Features Overview
- CoreSight Overview
- Debug Modes
- Debugging Events
- Breakpoint in the Cortex-M3
- Accessing Register Content in Debug
- Other Core Debugging Features

Debugging Features Overview

The Cortex-M3 processor provides a comprehensive debugging environment. Based on the nature of operations, the debugging features can be classified into two groups:

1. Invasive debugging:

   - Program halt and stepping
   - Hardware breakpoints
   - Breakpoint instruction
   - Data watchpoint on access to data address, address range, or data value
   - Register value accesses (both read or write)
   - Debug monitor exception
   - ROM-based debugging (Flash patch)

2. Noninvasive debugging:
    - Memory accesses (memory contents can be accessed even when the core is running)
    - Instruction trace (via the optional Embedded Trace Module)
    - Data trace
    - Software trace (via the Instrumentation Trace Module)
    - Profiling (via the Data Watchpoint and Trace Module)

A number of debugging components are included in the Cortex-M3 processor. The debugging system is based on the CoreSight debug architecture, allowing a standardized solution to access debugging controls, gather trace information, and detect debugging system configuration.

CoreSight Overview

The CoreSight debug architecture covers a wide area, including the debugging interface protocol, debugging bus protocol, control of debugging components, security features, trace data interface, and more. The *CoreSight Technology System Design Guide* (Ref 3) is a useful document for getting an overview of the architecture. In addition, a number of sections in the *Cortex-M3 Technical Reference Manual* (Ref 1) are descriptions of the debugging components in Cortex-M3 design. These components are normally used only by debugger software, not by application code. However, it is still useful to briefly review these items so that we can have a better understanding of how the debugging system works.

Processor Debugging Interface

Unlike traditional ARM7 or ARM9, the debugging system of the Cortex-M3 processor is based on the CoreSight Debug Architecture. Traditionally, ARM processors provide a JTAG interface, allowing registers to be accessed and memory interface to be controlled. In the Cortex-M3, the control to the debug logic on the processor is carried out via a bus interface called the Debug Access Port (DAP), which is similar to APB in AMBA. The DAP is controlled by another component that converts JTAG or Serial-Wire into the DAP bus interface protocol.

Since the internal debug bus is similar to APB, it is easy to connect multiple debugging components, resulting in a very scalable debugging system. In addition, by separating the debug interface and debug control hardware, the actual interface type used on the chip can become transparent; hence the same debugging tasks can be carried out no matter what debugging interface you use.

The actual debugging functions in the Cortex-M3 processor core are controlled by the NVIC and a number of other debugging components, such as the FPB, the DWT, and the ITM. The NVIC contains a number of registers to control the core debugging actions, such as halt and stepping, while the other blocks support features such as watchpoints, breakpoints, and debug message outputs.

The Debug Host Interface

CoreSight technology supports a number of interface types for connection between the debug host and the SoC. Traditionally this has always been JTAG. Now, since the processor debugging interface has been changed to a generic bus interface, by putting a different interface module between the debug host and the processor's debug interface we can come up with different chips that have different debug host interfaces, without redesigning the debug interface on the processor.

Currently Cortex-M3 systems support two types of debug host interface: The first one is the well-known JTAG interface, and the second one is a new interface protocol called Serial-Wire (SW). The SW interface reduces the number of signals to two. Several types of debug host interface modules (called Debug Port, or DP) are available from ARM. The debugger hardware is connected to one side of a DP, and the other side is connected to the DAP interface on the processor.

Why Serial-Wire?

The Cortex-M3 is targeted at the low-cost microcontroller market in which most devices have very low pin counts. For example, some of the low-end versions are in 28-pin packages. Despite the fact that JTAG is a very popular protocol, using four pins to debug is a lot for a 28-pin device. Therefore, SW is an attractive solution because it can reduce the number of debug pins to two.

DP Module, AP Module, and DAP

The connection from external debugging hardware to the debug interface in the Cortex-M3 processor is divided into multiple stages (see Figure 15.1).

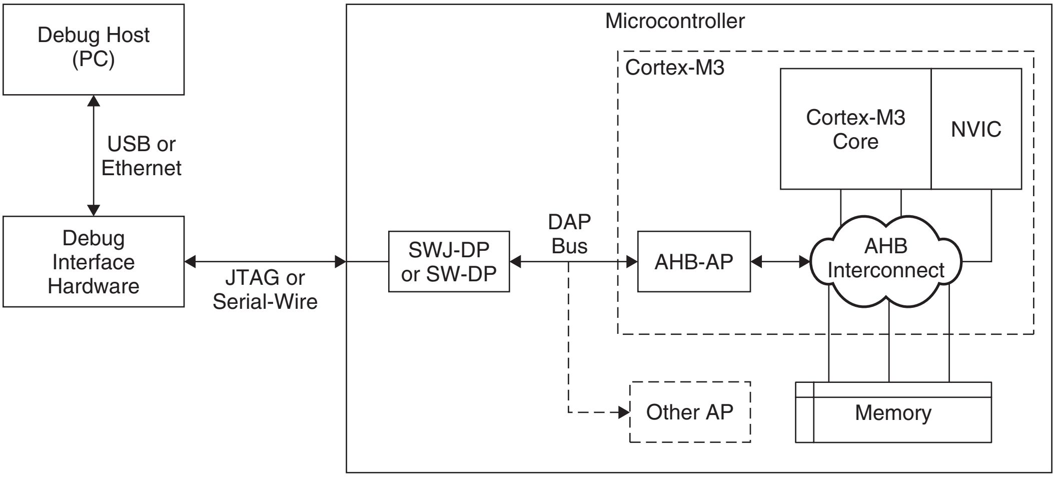


Figure 15.1 Connection from the Debug Host to the Cortex-M3

The DP interface module (normally either SWJ-DP or SW-DP) first converts the external signals into a generic 32-bit debug bus (a DAP bus in the diagram). SWJ-DP supports both JTAG and SW, and SW-DP supports SW only. In the ARM CoreSight product series there is also a JTAG-DP, which only supports the JTAG protocol; chip manufacturers can choose to use one of these DP modules to suit their needs. The address of the DAP bus is 32-bit, with the upper 8 bits of the address bus used to select which device is being accessed. Up to 256 devices can be attached to the DAP bus. Inside the Cortex-M3 processor, only one of the device addresses is used, so you can attach 255 more Access Port (AP) devices to the DAP bus if needed.

After passing through the DAP interface in the Cortex-M3 processor, an AP device called AHB-AP is connected. This acts as a bus bridge to convert commands into AHB transfers, which are inserted into the internal bus network inside the Cortex-M3. This allows the memory map of the Cortex-M3, including the debug control registers in the NVIC, to be accessed.

In the CoreSight product series, several types of AP devices are available, including an APB-AP and a JTAG-AP. The APB-AP can be used to generate APB transfers, and the JTAG-AP can be used to control traditional JTAG-based test interfaces such as the debug interface on ARM7.

Trace Interface

Another part of the CoreSight architecture concerns tracing. In the Cortex-M3, there can be three types of trace sources:

- Instruction trace: Generated by the Embedded Trace Macrocell (ETM)
- Data trace: Generated by DWT

- Debug message: Generated by ITM (provides message output such as *printf* in the debugger GUI)

During tracing, the trace results, in the form of data packets, are output from the trace sources like ETM, using a trace data bus interface called Advanced Trace Bus (ATB). Based on the CoreSight architecture, if a SoC contains multiple trace sources (e.g., multiprocessors), the ATB data stream can be merged using ATB merger hardware (in the CoreSight architecture this hardware is called *ATB funnel*). The final data stream on the chip can then be connected to a Trace Port Interface Unit (TPIU) and exported to external trace hardware. Once the data reach the debug host (for example, a PC), the data stream can then be converted back into multiple data streams.

Despite the Cortex-M3 having multiple trace sources, its debugging components are designed to handle trace merging so that there is no need to add ATB funnel modules. The trace output interface can be connected directly to a special version of the TPIU designed for the Cortex-M3. The trace data are then captured by external hardware and collected by the debug host (e.g., a PC) for analysis.

CoreSight Characteristics

The CoreSight-based design has a number advantages:

- The memory content and peripheral registers can be examined even when the processor is running.
- Multiple processor debug interfaces can be controlled with a single piece of debugger hardware. For example, if JTAG is used, only one TAP controller is required, even when there are multiple processors on the chip.
- Internal debugging interfaces are based on simple bus design, making it scalable and easy to develop additional test logic for other parts of the chip or SoC.
- It allows multiple trace data streams to be collected in one trace capture device and separated back into multiple streams on the debug host.

The debugging system used in the Cortex-M3 processor is slightly different from the standard CoreSight implementation:

- Trace components are specially designed in the Cortex-M3. Some of the ATB interface is 8 bits wide in the Cortex-M3, whereas in CoreSight the width is 32 bits.
- The debug implementation in the Cortex-M3 does not support TrustZone.[1]

[1] TrustZone is an ARM technology that provides security features to embedded products.

- The debug components are part of the system memory map, whereas in standard CoreSight systems, a separate bus (with a separate memory map) is used for controlling debug components. For example, the conceptual system connection in a CoreSight system can be like the one shown in Figure 15.2.

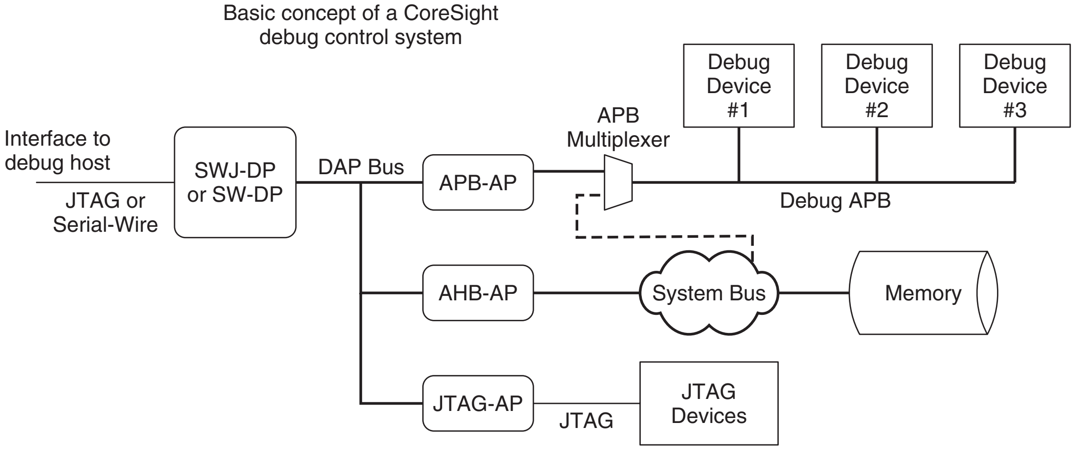


Figure 15.2 Design Concept of a CoreSight System

In the Cortex-M3, the debugging devices share the same system memory map (see Figure 15.3).

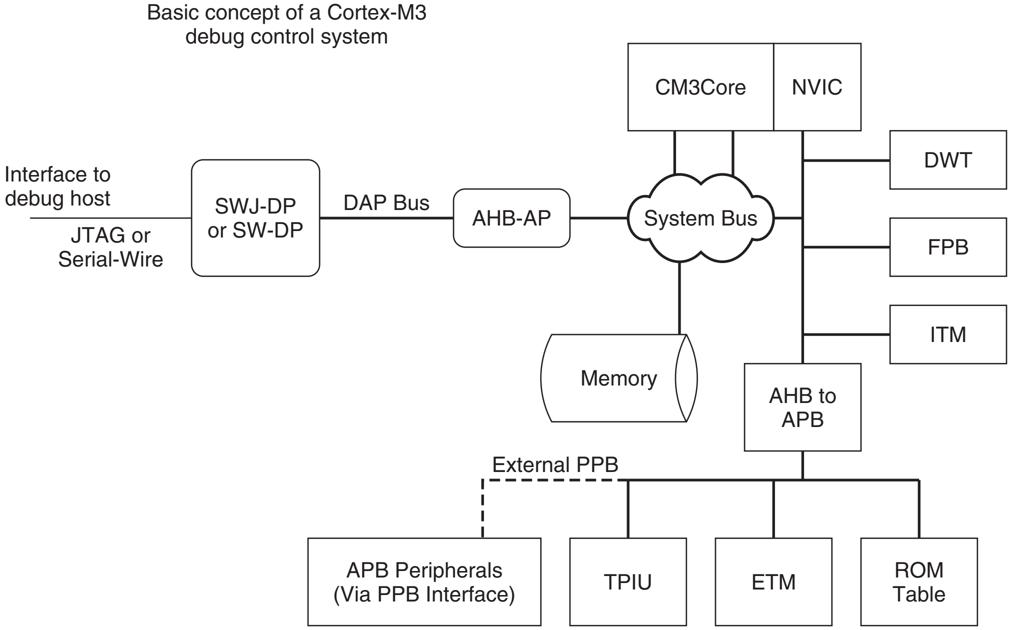


Figure 15.3 The Debug System in the Cortex-M3

Additional information about the CoreSight Debug Architecture can be found in the *CoreSight Technology System Design Guide* (Ref 3).

Debug Modes

There are two types of debug operation modes in the Cortex-M3. The first one is *halt,* whereby the processor stops program execution completely. The second one is the *debug monitor exception,* whereby the processor executes an exception handler to carry out the debugging tasks while still allowing higher-priority exceptions to take place. Debug monitor is exception type 12 and its priority is programmable. It can be invoked by means of debug events as well as by manually setting the pending bit. In summary:

1. Halt mode:
    - Instruction execution is stopped
    - SYSTICK counter is stopped
    - Supports single-step operations
    - Interrupts can be pended and can be invoked during single stepping or be masked so that external interrupts are ignored during stepping
2. Debug monitor mode:
    - Processor executes exception handler type 12 (debug monitor)
    - SYSTICK counter continues to run
    - New arrive interrupts may or may not preempt, depending on the priority of the debug monitor and the priority of the new interrupt
    - If the debug event takes place when a higher-priority interrupt is running, the debug event will be missed
    - Supports single-step operations
    - Memory contents (for example, stack memory) could be changed by the debug monitor handler during stacking and handler execution

The reason for having a debug monitor is that in some electronic systems, stopping a processor for a debugging operation can be infeasible. For example, in automotive engine control or hard disk controller applications, the processor should continue to serve interrupt requests during debugging, to ensure safety of operations or to prevent damage to the device being tested. With a debug monitor, the debugger can stop and debug the Thread level application and lower-priority interrupt handlers while higher-priority interrupts and exceptions can still be executed.

To enter halt mode, the C_DEBUGEN bit in the NVIC Debug Halting Control and Status Register (DHCSR) must be set. This bit can only be programmed via the DAP, so you cannot halt the Cortex-M3 processor without a debugger. After C_DEBUGEN is set, the core can be halted by setting the C_HALT bit in DHCSR. This bit can be set by either the debugger or by the software running on the processor itself.

The bit field definition of DHCSR differs between read operations and write operations. For write operations, a debug key value must be used on bit 31 to bit 16. For read operations, there is no debug key and the return value of the upper half word contains the status bits (see Table 15.1).

Table 15.1 Debug Halting Control and Status Register (0xE000EDF0)

Bits	Name	Type	Reset Value	Description
31:16	KEY	W	—	Debug key; value of 0xA05F must be written to this field to write to this register, otherwise the write will be ignored
25	S_RESET_ST	R	—	Core has been reset or being reset; this bit is clear on read
24	S_RETIRE_ST	R	—	Instruction is completed since last read; this bit is clear on read
19	S_LOCKUP	R	—	When this bit is 1, the core is in a locked-up state
18	S_SLEEP	R	—	When this bit is 1, the core is in sleep mode
17	S_HALT	R	—	When this bit is 1, the core is halted
16	S_REGRDY	R	—	Register read/write operation is completed
15:6	Reserved	—	—	Reserved
5	C_SNAPSTALL	R/W	0*	Use to break a stalled memory access
4	Reserved	—	—	Reserved
3	C_MASKINTS	R/W	0*	Mask interrupts while stepping; can only be modified when the processor is halted
2	C_STEP	R/W	0*	Single step the processor; valid only if C_DEBUGEN is set
1	C_HALT	R/W	0*	Halt the processor core; valid only if C_DEBUGEN is set
0	C_DEBUGEN	R/W	0*	Enable halt mode debug
* The control bit in DHCSR is reset by power on reset. System reset (for example, by the NVIC's Application Interrupt and Reset Control register) does not reset the debug controls.				

In normal situations, the DHCSR is used only by the debugger. Application codes should not change DHCSR contents to avoid causing problems to debugger tools.

- The control bit in DHCSR is reset by power-on reset. System reset (for example, by the NVIC's Application Interrupt and Reset Control register) does not reset the debug controls. For debugging using debug monitor, a different NVIC register, the NVIC's Debug Exception and Monitor Control register, is used to control the debug activities (see Table 15.2). Aside from the debug monitor control bits, the Debug Exception and Monitor Control register contains the trace system enable bit (TRCENA) and a number of Vector Catch (VC) control bits. The VC feature can be used only with halt mode debugging. When a fault (or core reset) takes place and the corresponding VC control bit is set, the halt request will be set and the core will halt as soon as the current instruction completes.

Table 15.2 Debug Exception and Monitor Control Register (0xE000EDFC)

Bits	Name	Type	Reset Value	Description
24	TRCENA	R/W	0*	Trace system enable; to use DWT, ETM, ITM and TPIU, this bit must be set to 1
23:20	Reserved	—	—	Reserved
19	MON_REQ	R/W	0	Indication that the debug monitor is caused by a manual pending request rather than hardware debug events
18	MON_STEP	R/W	0	Single step the processor; valid only if MON_EN is set
17	MON_PEND	R/W	0	Pend the monitor exception request; the core will enter monitor exceptions when priority allows
16	MON_EN	R/W	0	Enable the debug monitor exception
15:11	Reserved	—	—	Reserved
10	VC_HARDERR	R/W	0*	Debug trap on hard faults
9	VC_INTERR	R/W	0*	Debug trap on interrupt/exception service errors
8	VC_BUSERR	R/W	0*	Debug trap on bus faults
7	VC_STATERR	R/W	0*	Debug trap on usage fault state errors
6	VC_CHKERR	R/W	0*	Debug trap on usage fault-enabled checking errors (e.g., unaligned, divide by zero)
5	VC_NOCPERR	R/W	0*	Debug trap on usage fault, no coprocessor errors
4	VC_MMERR	R/W	0*	Debug trap on memory management fault
3:1	Reserved	–	–	Reserved
0	VC_CORERESET	R/W	0*	Debug trap on core reset

* The control bit in DHCSR is reset by power on reset. System reset (for example, by the NVIC's Application Interrupt and Reset Control register) does not reset the debug controls.

- The TRCENA control bit and VC control bits in DEMCR are reset by power-on reset. System reset does not reset these bits. The control bits for monitor mode debug, however, are reset by power-on reset as well as system reset.

Debugging Events

The Cortex-M3 can enter debug mode (both halt or debug monitor exception) for a number of possible reasons. For halt mode debugging, the processor will enter halt mode if conditions resemble those shown in Figure 15.4.

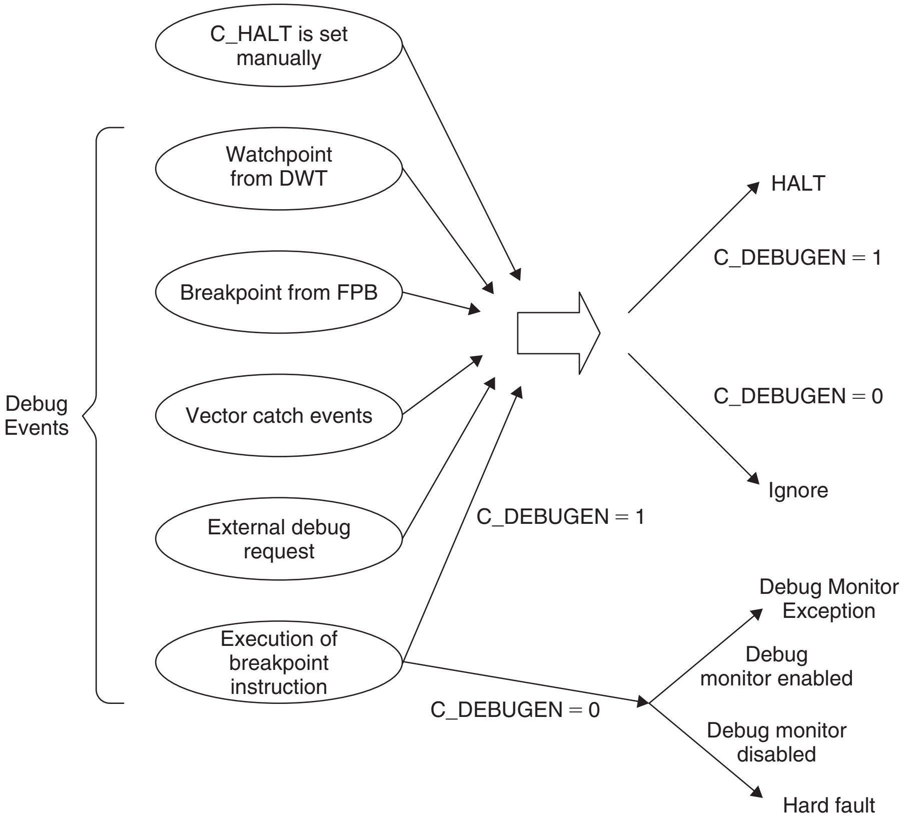


Figure 15.4 Debugging Events for Halt Mode Debugging

The external debug request is from a signal called EDBGREQ on the Cortex-M3 processor. The actual connection of this signal depends on the microcontroller or SoC design. In some cases this signal could be tied low and never occur. However, this can be connected to accept debug events from additional debug components (chip manufacturers can add extra debug components to the SoC) or, if the design is a multiprocessor system, it could be linked to debug events from another processor.

After debugging is completed, the program execution can be returned to normal by clearing the C_HALT bit.

Similarly, for debugging with the debug monitor exceptions, a number of debug events can cause a debug monitor to take place (see Figure 15.5).

For debug monitor, the behavior is a bit different from halt mode debugging. This is because the debug monitor exception is just one type of exception and can be affected by the current priority of the processor if it is running another exception handler.

After debugging is completed, the program execution can be returned to normal by carrying out an exception return.

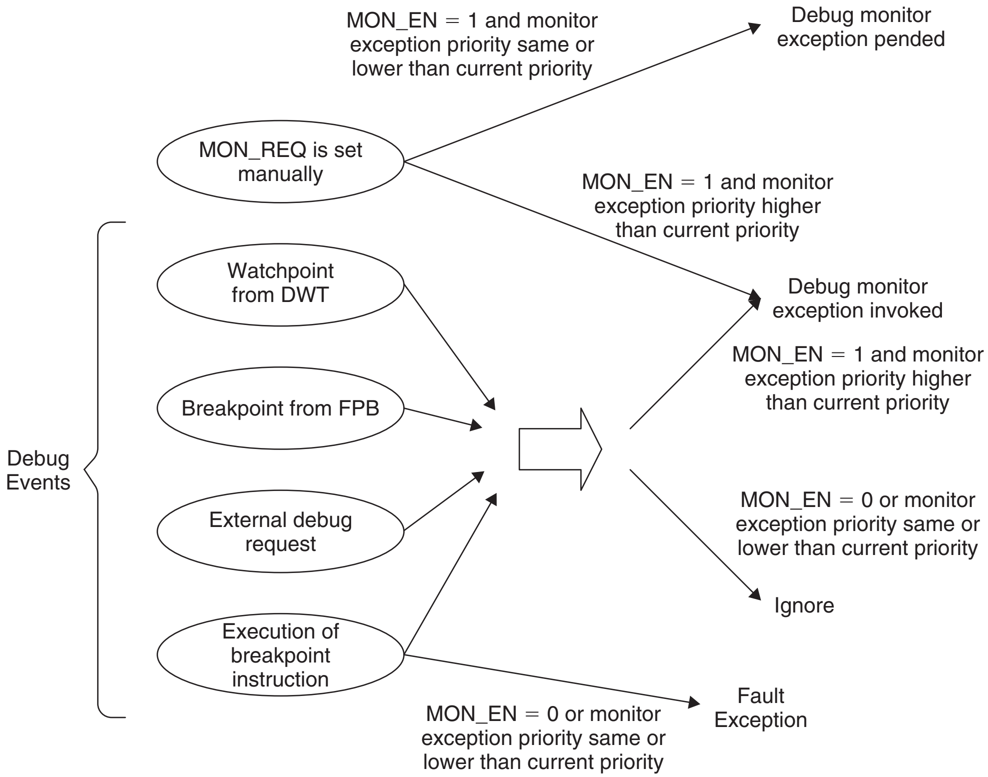

MON_EN = 1 and monitor exception priority same or lower than current priority
Debug monitor exception pended
MON_REQ is set manually
MON_EN = 1 and monitor exception priority higher than current priority
Watchpoint from DWT
Debug monitor exception invoked
MON_EN = 1 and monitor exception priority higher than current priority
Breakpoint from FPB
Debug Events
External debug request
MON_EN = 0 or monitor exception priority same or lower than current priority
Ignore
Execution of breakpoint instruction
Fault Exception
MON_EN = 0 or monitor exception priority same or lower than current priority

Figure 15.5 Debugging Events for Debug Monitor Exceptions

Breakpoint in the Cortex-M3

One of the most commonly used debug features in most microcontrollers is the breakpoint feature. In the Cortex-M3, two types of breakpoint mechanisms are supported:

- Breakpoint instruction
- Breakpoint using address comparators in the FPB

The breakpoint instruction (*BKPT immed8*) is a 16-bit Thumb instruction with encoding 0xBExx. The lower 8 bits depend on the immediate data given following the instruction. When this instruction is executed, it generates a debug event and can be used to halt the processor core if C_DBGEN is set or, if debug monitor is enabled, it can be used to trigger the debug monitor exception. Since the debug monitor is one type of exception with programmable priority, it can only be used in Thread or exception handlers with priority lower than itself. As a result, if debug monitor is used for debugging, the BKPT instructions should not be used in exception handlers such as NMI or hardfault, and the debug monitor can only be pended and executed after the exception handler is completed.

When the debug monitor exception returns, it is returned to the address of the BKPT instruction, not the address after the BKPT instruction. This is because in normal use of breakpoint instructions, the BKPT is used to replace a normal instruction, and when the breakpoint is hit and the debug action is carried out, the instruction memory is restored to the original instruction and the rest of the instruction memory is unaffected.

If the BKPT instruction is executed with C_DEBUGEN = 0 and MON_EN = 0, it will cause the processor to enter a hard fault exception, with DEBUGEVT in the Hard Fault Status Register (HFSR) set to 1, and BKPT in the Debug Fault Status Register (DFSR) also set to 1.

The FPB unit can be programmed to generate breakpoint events even if the program memory cannot be altered. However, it is limited to six instruction addresses and two literal addresses. More information about FPB is covered in the next chapter.

Accessing Register Content in Debug

Two more registers are included in the NVIC to provide debug functionality. They are the Debug Core Register Selector Register (DCRSR) and the Debug Core Register Data Register (DCRDR) (see Tables 15.3 and 15.4). These two registers allow the debugger to access processors' registers. The register transfer feature can be used only when the processor is halted.

Table 15.3 Debug Core Register Selector Register (0xE000EDF4)

Bits	Name	Type	Reset Value	Description
16	REGWnR	W	—	Direction of data transfer: Write = 1, Read = 0
15:5	Reserved	—	—	-
4:0	REGSEL	W	—	Register to be accessed: 00000 = R0 00001 = R1 ... 01111 = R15 10000 = xPSR/flags 10001 = MSP (Main Stack Pointer) 10010 = PSP (Process Stack Pointer) 10100 = Special registers: [31:24] Control [23:16] FAULTMASK [15:8] BASEPRI [7:0] PRIMASK Other values are reserved

Table 15.4 Debug Core Register Data Register (0xE000EDF8)

Bits	Name	Type	Reset Value	Description
31:0	Data	R/W	—	Data register to hold register read result or to write data into selected register

To use these registers to read register contents, the following procedure must be followed:

1. Make sure the processor is halted.
2. Write to the DCRSR with bit 16 set to 0, indicating it is a read operation.
3. Poll until the S_REGRDY bit in DHCSR (0xE000EDF0) is 1.
4. Read the DCRSR to get the register content.

Similar operations are needed for writing to a register:

1. Make sure the processor is halted.
2. Write data value to the DCRDR.
3. Write to the DCRSR with bit 16 set to 1, indicating it is a write operation.
4. Poll until the S_REGRDY bit in DHCSR (0xE000EDF0) is 1.

The DCRSR and the DCRDR registers can only transfer register values during halt mode debug. For debugging using a debug monitor handler, the contents of some of the register can be accessed from the stack memory; the others can be accessed directly within the monitor exception handler.

The DCRDR can also be used for semihosting if suitable function libraries and debugger support are available. For example, when an application executes a *printf* statement, the text output could be generated by a number of *putc* (put character) function calls. The *putc* function calls can be implemented as functions that store the output character and status to the DCRDR and then trigger debug mode. The debugger can then detect the core halt and collect the output character for display. This operation, however, requires the core to halt, whereas the semihosting solution using ITM does not have this limitation.

Other Core Debugging Features

The NVIC also contains a number of other features for debugging. These include the following:

- External debug request signal: The NVIC provides an external debug request signal that allows the Cortex-M3 processor to enter debug mode via an external event such as debug status of other processors in a multiprocessor system. This feature is very useful for debugging a multiprocessor system. In simple microcontrollers, this signal is likely to be tied low.

- Debug Fault Status register: Due to the various debug events available on the Cortex-M3, a DFSR is available for the debugger to determine the debug event that has taken place.
- Reset control: During debugging, the processor core can be restarted using the VECTRESET control bit in the NVIC Application Interrupt and Reset Control register (0xE000ED0C). Using this reset control, the processor can be reset without affecting the debug components in the system.
- Interrupt masking: This feature is very useful during stepping. For example, if you need to debug an application but do not want the code to enter the interrupt service routine during the stepping, the interrupt request can be masked. This is done by setting the C_MASKINTS bit in the Debug Halting Control and Status register (0xE000EDF0).
- Stalled bus transfer termination: If a bus transfer is stalled for a very long time, it is possible to terminate the stalled transfer by an NVIC control register. This is done by setting the C_SNAPSTALL bit in the Debug Halting Control and Status register (0xE000EDF0). This feature can be used only by a debugger during halt.

CHAPTER 16

Debugging Components

In This Chapter:

- Introduction
- Trace Components: Data Watchpoint and Trace
- Trace Components: Instrumentation Trace Macrocell
- Trace Components: Embedded Trace Macrocell
- Trace Components: Trace Port Interface Unit
- The Flash Patch and Breakpoint Unit
- The AHB Access Port
- ROM Table

Introduction

The Cortex-M3 processor comes with a number of debugging components used to provide debugging features such as breakpoint, watchpoint, Flash patch, and trace. If you are an application developer, there might be a chance that you'll never need to know the details about these debugging components, because they are normally used only by debugger tools. This chapter will introduce you to the basics of each debug component. If you want to know details about things such as the actual programmer's model, refer to the *Cortex-M3 Technical Reference Manual* (Ref 1).

All the debug trace components, as well as the FPB, can be programmed via the Cortex-M3 Private Peripheral Bus (PPB). In most cases, the components will only be programmed by the debugging host. It is not recommended for applications to try accessing the debug components (except stimulus port registers in the ITM), because this could interfere with the debugger's operation.

The Trace System in the Cortex-M3

The Cortex-M3 trace system is based on the CoreSight architecture. Trace results are generated in the form of packets, which can be of various lengths (in terms of number of bytes). The trace components transfer the packets using Advanced Trace Bus (ATB) to the Trace Port Interface Unit (TPIU), which formats the packets into Trace Interface Protocol. The data are then captured by an external trace capture device such as a Trace Port Analyzer (TPA).

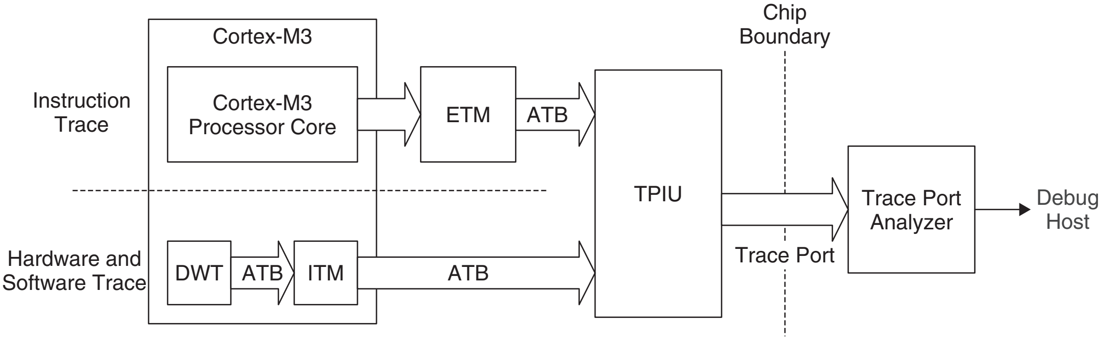


Figure 16.1 The Cortex-M3 Trace System

There are up to three trace sources in a standard Cortex-M3 processor: ETM, ITM, and DWT. Note that the ETM in the Cortex-M3 is optional, so some Cortex-M3 products do not have instruction trace capability. During operation, each trace source is assigned a 7-bit ID value (ATID), which is transferred along the trace packets during merging in the ATB so that the packets can be separated back into multiple trace streams when they reach the debug host.

Unlike many other standard CoreSight components, the debug components in the Cortex-M3 processor include the functionality of merging ATB streams, whereas in standard CoreSight systems, ATB packet merger, called *ATB funnel,* is a separate block.

Before using the trace system, the Trace Enable (TRCENA) bit in the Debug Exceptions and Monitor Control Register (DEMCR) must be set to 1 (see Table 15.2 or D.37). Otherwise the trace system will be disabled. In normal operations that do not require tracing, clearing the TRCENA bit can disable some of the trace logic and reduce power consumption.

Trace Components: Data Watchpoint and Trace

The DWT has a number of debugging functionalities:

1. It has four comparators, each of which can be configured as follows:
    - Hardware watchpoint (generates a watchpoint event to processor to invoke debug modes such as halt or debug monitor)
    - ETM trigger (causes the ETM to emit a trigger packet in the instruction trace stream)
    - PC sampler event trigger

- Data address sampler trigger
- The first comparator can also be used to compare against the clock cycle counter (CYCCNT) instead of comparing to a data address

2. Counters for counting the following:
   - Clock cycles (CYCCNT)
   - Folded instructions
   - Load Store Unit (LSU) operations
   - Sleep cycles
   - Cycles per instruction (CPI)
   - Interrupt overhead
3. PC sampling at regular intervals
4. Interrupt events trace

When used as a hardware watchpoint or ETM trigger, the comparator can be programmed to compare either data addresses or program counters. When programmed as other functions, it compares the data addresses.

Each of the comparators has three corresponding registers:

- COMP (compare) register
- MASK register
- FUNCTION control register

The COMP register is a 32-bit register that the data address (or program counter value, or CYCCNT) compares to. The MASK register determines whether any bit in the data address will be ignored during the compare (see Table 16.1).

Table 16.1 Encoding of the DWT Mask Registers

MASK	Ignore Bit
0	All bits are compared
1	Ignore bit [0]
2	Ignore bit [1:0]
3	Ignore bit [2:0]
…	…
15	Ignore bit [14:0]

The comparator's FUNCTION register determines its function. To avoid unexpected behavior, the MASK register and the COMP register should be programmed before this register is set. If the comparator's function is to be changed, you should disable the comparator by setting FUNCTION to 0 (disable), then program the MASK and COMP registers, and then enable the FUNCTION register in the last step.

The rest of the DWT counters are typically used for profiling the application codes. They can be programmed to emit events (in the form of trace packets) when the counter overflows. One typical application is to use the CYCCNT register to count the number of clock cycles required for a specific task, for benchmarking purposes.

The TRCENA bit in the DEMCR must be set to 1 before the DWT is used. If the DWT is being used to generate a trace, the DWTEN bit in the ITM Control register should also be enabled.

Trace Components: Instrumentation Trace Macrocell

The ITM has the following functionalities:

- Software can directly write console messages to ITM stimulus ports and output them as trace data.
- The DWT can generate trace packets and output them via the ITM.
- The ITM can generate timestamp packets that are inserted into a trace stream to help the debugger find out the timing of events.

Since the ITM uses a trace port to output data, if the microcontroller or SoC does not have TPIU support, the traced information cannot be output. Therefore, it is necessary to check whether the microcontroller or SoC has all the required features before you use the ITM. In the worst case, if these features are not available you can still use the NVIC debug register or a UART to output console messages.

To use the ITM, the TRCENA bit in the DEMCR must be set to 1. Otherwise the ITM will be disabled and ITM registers cannot be accessed.

In addition, there is also a lock register in the ITM. You need to write the access key 0xC5ACCE55 (CoreSight ACCESS) to this register before programming the ITM. Otherwise, all write operations to the ITM will be ignored.

Finally, the ITM itself is another Control register to control the enabling of individual features. The Control register also contains the ATID field, which is an ID value for the ITM in the ATB. This ID value must be unique from the IDs for other trace sources so that the debug host receiving the trace packet can separate the ITM's trace packets from other trace packets.

Software Trace with the ITM

One of the main uses of the ITM is to support debug message output (such as *printf*). The ITM contains 32 stimulus ports, allowing different software processes to output to different ports, and the messages can be separated later at the debug host. Each port can be enabled or disabled by the Trace Enable register and can be programmed (in groups of eight ports) to allow or disallow user processes to write to it.

Unlike UART-based text output, using the ITM to output does not cause much delay for the application. A FIFO buffer is used inside the ITM, so writing output messages can be buffered. However, it is still necessary to check whether the FIFO is full before you write to it.

The output messages can be collected at the trace port interface or the Serial-Wire Interface (SWV) on the TPIU. There is no need to remove code that generates the debug messages from the final code, because if the TRCENA control bit is low, the ITM will be inactive and debug messages will not be output. You can also switch on the output message in a "live" system and use the Trace Enable register in the ITM to limit which ports are enabled so that only some of the messages can be output.

Hardware Trace with ITM and DWT

The ITM is used in output of hardware trace packets. The packets are generated from the DWT, and the ITM acts as a trace packet merging unit. To use DWT trace, you need to enable the DWTEN bit in the ITM Control register; the rest of the DWT trace settings still need to be programmed at the DWT.

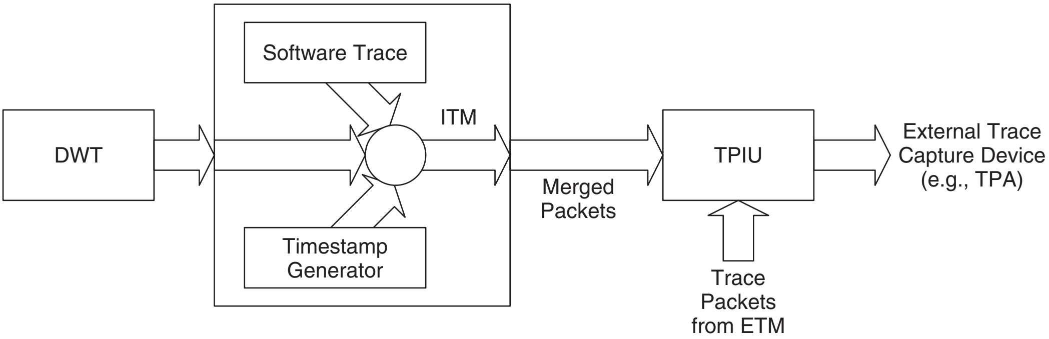


Figure 16.2 Merging of Trace Packets on the ITM and TPIU

ITM Timestamp

ITM has a timestamp feature that allows trace capture tools to find out timing information by inserting delta timestamp packets into the traces when a new trace packet enters the

FIFO inside the ITM. The timestamp packet is also generated when the timestamp counter overflows.

The timestamp packets provide the time difference (delta) with previous events. Using the delta timestamp packets, the trace capture tools can then establish the timing of when each packet is generated and hence reconstruct the timing of various debug events.

Trace Components: Embedded Trace Macrocell

The ETM block is used for providing instruction traces. It is optional and might not be available on some Cortex-M3 products. When it is enabled and when trace operation starts, it generates instruction trace packets. A FIFO buffer is provided in the ETM to allow enough time for the trace stream to be captured.

To reduce the amount of data generated by the ETM, it does not always output exactly what address the processor has reached/executed. It usually outputs information about program flow and outputs full addresses only if needed (e.g., if a branch has taken place). Since the debugging host should have a copy of the binary image, it can then reconstruct the instruction sequence the processor has carried out.

The ETM also interacts with other debugging components such as the DWT. The comparators in the DWT can be used to generate trigger events in the ETM or to control the trace start/stop.

Unlike the ETM in traditional ARM processors, the Cortex-M3 ETM does not have its own address comparators, because the DWT can carry out the comparison for ETM. Furthermore, since the data trace functionality is carried out by the DWT, the ETM design in the Cortex-M3 is quite different from traditional ETM for other ARM cores.

To use the ETM in the Cortex-M3, the following setup is required (handled by debug tools):

1. The TRCENA bit in the Debug Exceptions and Monitor Control Register (DEMCR) must be set to 1 (see Table 15.2 or D.37).
2. The ETM needs to be unlocked so that its control registers can be programmed. This can be done by writing the value 0xC5ACCE55 to the ETM LOCK_ACCESS register.
3. The ATB ID register (ATID) should be programmed to a unique value so that the trace packet output via the TPIU can be separated from packets from other trace sources.
4. The NIDEN input signal of the ETM must be set to high. The implementation of this signal is device specific. Refer to the datasheet from your chip's manufacturer for details.
5. Program the ETM control registers for trace generation.

Trace Components: Trace Port Interface Unit

The TPIU is used to output trace packets from the ITM, DWT, and ETM to the external capture device (for example, a trace port analyzer). The Cortex-M3 TPIU supports two output modes:

- Clocked mode, using up to 4-bit parallel data output ports
- Serial-Wire Viewer (SWV) mode, using single-bit SWV output[1]

In clocked mode, the actual number of bits being used on the data output port can be programmed to different sizes. This will depend on the chip package as well as the number of signal pins available for trace output in the application. The maximum trace port size supported by the chip can be determined from one of the registers in the TPIU. In addition, the speed of trace data output can also be programmed.

In SWV mode, the SWV protocol is used. This reduces the number of output signals, but the maximum bandwidth for trace output will also be reduced.

To use the TPIU, the TRCENA bit in the DEMCR must be set to 1, and the protocol (mode) selection register and trace port size control registers need to be programmed by the trace capture software.

The Flash Patch and Breakpoint Unit

The FPB has two functions:

- Hardware breakpoint (generates a breakpoint event to the processor to invoke debug modes such as halt or debug monitor)
- Patch instruction or literal data from Code memory space to SRAM

The FPB contains eight comparators:

- Six instruction comparators
- Two literal comparators

[1] Not available on early versions of Cortex-M3 products that are based on Cortex-M3 revision 0.

What Are Literal Loads?

When we program in assembler language, very often we need to set up immediate data values in a register. When the value of the immediate data is large, the operation cannot be fitted into one instruction space. For example:

```
LDR   R0, =0xE000E400    ; External Interrupt Priority Register
                         ; starting address
```

Since no instruction has an immediate value space of 32, we need to put the immediate data in a different memory space, usually after the program code region, and then use a PC relative load instruction to read the immediate data into the register. So what we get in the compiled binary code will be something like this:

```
LDR   R0, [PC, #<immed_8>*4]
        ; immed_8 = (address of literal value - PC)/4
...
; literal pool
...
DCD   0xE000E400
...
```

or with Thumb-2 instructions:

```
LDR.W R0, [PC, #+/- <offset_12>]
        ; offset_12 = address of literal value - PC
...
; literal pool
...
DCD 0xE000E400
...
```

Since we are likely to use more than one literal value in our code, the assembler or compiler will usually generate a block of literal data, it is commonly called literal pool.

In Cortex-M3, the literal load are data read operation carried out on the data bus (D-CODE bus or System bus depending on memory location).

The FPB has a Flash Patch control register that contains an enable bit to enable the FPB. In addition, each comparator comes with a separate enable bit in its comparator control register. Both of the enable bits must be set to 1 for a comparator to operate.

The comparators can be programmed to remap addresses from Code space to the SRAM memory region. When this function is used, the REMAP register needs to be programmed to provide the base address of the remapped contents. The upper three bits of the REMAP register (bit[31:29]) is hardwired to 3′b001, which limited the remap base address location to be within 0x20000000 to 0x3FFFFF80, which is always within the SRAM memory region.

When the instruction address or the literal address hits the address defined by the comparator, the read access is remapped to the table pointed to by the REMAP register.

Using the remap function, it is possible to create some "what if" test cases in which the original instruction or a literal value is replaced by a different one; even the program code is in ROM or Flash memory. An example use is to allow execution of a program or subroutine in the SRAM region by patching program ROM in the Code region so that a branch to the test program or subroutine can take place. This makes it possible to debug a ROM-based device.

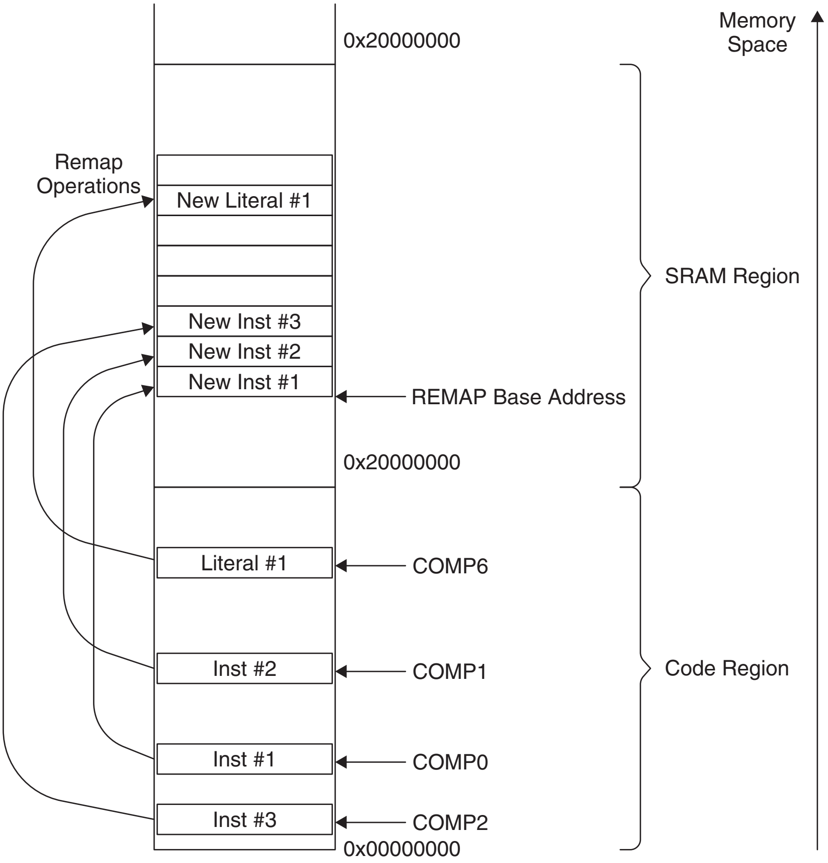


Figure 16.3 Flash Patch: Remap of Instructions and Literal Read

Alternatively, the six instruction address comparators can be used to generate breakpoints as well as to invoke halt mode debug or debug monitor exceptions.

The AHB Access Port

The AHB-AP is a bridge between the debug interface module (SWJ-DP or SW-DP) and the Cortex-M3 memory system. For the most basic data transfers between the debug host and the Cortex-M3 system, three registers in the AHB-AP are used:

- Control and Status Word (CSW)
- Transfer Address Register (TAR)
- Data Read/Write (DRW)

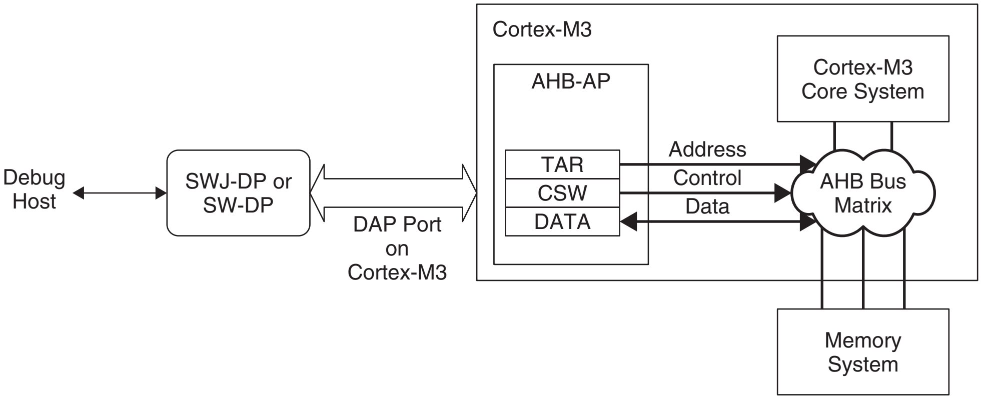


Figure 16.4 Connection of the AHB-AP in the Cortex-M3

The CSW register can control the transfer direction (read/write), transfer size, transfer types, and so on. The TAR register is used to specify the transfer address, and the DRW register is used to carry out the data transfer operation (transfer starts when this register is accessed).

The data register DRW represents exactly what is shown on the bus. For half word and byte transfers, the required data will have to be manually shifted to the correct byte lane by debugger software. For example, if you want to carry out a data transfer of half word size to address 0x1002, you need to have the data on bit [31:16] of the DRW register. The AHB-AP can generate unaligned transfers, but it does not rotate the result data based on address offset. So the debugger software will have to either rotate the data manually or split an unaligned data access into several accesses if needed.

Other registers in the AHB-AP provide additional features. For example, the AHB-AP provides four banked registers and an automatic address increment function so that access to memory within close range or sequential transfers can be speeded up.

In the CSW register, there is one bit called MasterType. This is normally set to 1 so that hardware receiving the transfer from AHB-AP knows that it is from the debugger. However, the debugger can pretend to be the core by clearing this bit. In this case the transfer received by the device attached to the AHB system should behave as though it is accessed by the processor. This is useful for testing peripherals with FIFO that can behave differently when accessed by the debugger.

ROM Table

The ROM table is used to allow auto detection of debug components inside a Cortex-M3 chip. The Cortex-M3 processor is the first product based on ARM v7-M architecture. It has a defined memory map and includes a number of debug components. However, in newer Cortex-M devices or if the chip designers modified the default debug components, the memory map for the debug devices could be different. To allow debug tools to detect the components in the debug system, a ROM table is included; it provides information on the NVIC and debug block addresses.

The ROM table is located in address 0xE00FF000. Using contents in the ROM table, the memory locations of system and debug components can be calculated. The debug tool can then check the ID registers of the discovered components and determine what is available on the system.

For the Cortex-M3, the first entry in the ROM table (0xE00FF000) should contain the offset to the NVIC memory location. (The default value in the ROM table's first entry is 0xFFF0F003; bit[1:0] means that the device exists and there is another entry in the ROM table following. The NVIC offset can be calculated as 0xE00FF000 + 0xFFF0F000 = 0xE000E000.)

The default ROM table for the Cortex-M3 is shown in Table 16.2. However, since chip manufacturers can add, remove, or replace some of the optional debug components with other CoreSight debug components, the value you find on your Cortex-M3 device could be different.

Table 16.2 Cortex-M3 Default RAM Table Values

Address	Value	Name	Description
0xE00FF000	0xFFF0F003	NVIC	Points to the NVIC base address at 0xE000E000
0xE00FF004	0xFFF02003	DWT	Points to the DWT base address at 0xE0001000
0xE00FF008	0xFFF03003	FPB	Points to the FPB base address at 0xE0002000
0xE00FF00C	0xFFF01003	ITM	Points to the ITM base address at 0xE0000000
0xE00FF010	0xFFF41003 / 0xFFF41002	TPIU	Points to the TPIU base address at 0xE0040000
0xE00FF014	0xFFF42003 / 0xFFF42002	ETM	Points to the ETM base address at 0xE0041000

(Continued)

Table 16.2 (Continued)

Address	Value	Name	Description
0xE00FF018	0	End	End-of-table marker
0xE00FFFCC	0x1	MEMTYPE	Indicates that system memory can be accessed on this memory map
0xE00FFFD0	0	PID4	Peripheral ID space; reserved
0xE00FFFD4	0	PID5	Peripheral ID space; reserved
0xE00FFFD8	0	PID6	Peripheral ID space; reserved
0xE00FFFDC	0	PID7	Peripheral ID space; reserved
0xE00FFFE0	0	PID0	Peripheral ID space; reserved
0xE00FFFE4	0	PID1	Peripheral ID space; reserved
0xE00FFFE8	0	PID2	Peripheral ID space; reserved
0xE00FFFEC	0	PID3	Peripheral ID space; reserved
0xE00FFFF0	0	CID0	Component ID space; reserved
0xE00FFFF4	0	CID1	Component ID space; reserved
0xE00FFFF8	0	CID2	Component ID space; reserved
0xE00FFFFC	0	CID3	Component ID space; reserved

The lowest two bits (LSB) of the value indicate whether the device exists. In normal cases, the NVIC, DWT, and FPB should always be there, so the last two bits are always 1. However, the TPIU and the ETM could be taken out by the chip manufacturer and might be replaced with other debugging components from the CoreSight product family.

The upper part of the value indicates the address offset from the ROM table base address. For example:

```
NVIC address = 0xE00FF000 + 0xFFF0F000 = 0xE000E000 (truncated to
  32-bit)
```

For debug tool development, it is necessary to determine the address of debug components from the ROM table. Some Cortex-M3 devices might have a different setup of the debug component connection that can result in different base addresses. By calculating the correct device address from this ROM table, the debugger can determine the base address of the provided debug component, and then from the component ID of those components the debugger can determine the type of debug components that are available.

CHAPTER 17

Getting Started with Cortex-M3 Development

In This Chapter:

- Choosing a Cortex-M3 Product
- Differences Between Cortex-M3 Revision 0 and Revision 1
- Development Tools

Choosing a Cortex-M3 Product

Aside from memory, peripheral options, and operation speed, a number of other factors make one Cortex-M3 products different from another. The Cortex-M3 design supplied by ARM contains a number of features that are configurable, such as:

- Number of external interrupts
- Number of interrupt priority levels (width of priority-level registers)
- With MPU or without MPU
- With ETM or without ETM
- Choice of debug interface (Serial-Wire, JTAG, or both)

In most projects, the features and specification of the microcontroller will certainly affect your choice of Cortex-M3 product. For example:

- Peripherals: For many applications, peripheral support is the main criterion. More peripherals might be good, but this also affects the microcontroller's power consumption and price.
- Memory: Cortex-M3 microcontrollers can have Flash memory from several kilobytes to several megabytes. In addition, the size of the internal memory might also be important. Usually these factors will have a direct impact on the price.

- Clock speed: The Cortex-M3 design from ARM can easily reach more than 100 MHz, even in 0.18 um processes. However, manufacturers might specify a lower operation speed due to limitations of memory access speed.
- Footprint: The Cortex-M3 can be available in many different packages, depending on the chip manufacturer's decision. Many Cortex-M3 devices are available in low pin count packages, making them ideal for low-cost manufacturing environments.

Differences Between Cortex-M3 Revision 0 and Revision 1

Early versions of Cortex-M3 products were based on revision 0 of the Cortex-M3 processor. Products based on Cortex-M3 revision 1 were available since the third quarter of 2006. When this book is published, all new Cortex-M3 based products should be based on revision 1. It could be important to know whether the chip you are using is revision 0 or revision 1, because there are a number of changes and improvements in the second release.

Changes visible in the programmer's model and development features include these:

- From revision 1, the stacking of registers when an exception occurs can be configured such that it is forced to begin from a double word aligned memory address. This is done by setting the STKALIGN bit in the NVIC Configuration Control register.
- For that reason, the NVIC Configuration Control register has the STKALIGN bit.
- Revision 2 includes the new AUXFAULT (Auxiliary Fault) status register (optional).
- Additional features include data value matching added to the DWT.
- ID register value changes due to the revision fields update.

Changes invisible to end users include:

- The memory attribute for Code memory space is hardwired to cacheable, allocated, nonbufferable, and nonshareable. This affects the I-Code AHB and the D-Code AHB interface but not the system bus interface.
- Supports bus multiplexing operation mode between I-Code AHB and D-Code AHB. Under this operation mode, the I-Code and D-Code bus can be merged using a simple bus multiplexer (previous solution is using an ADK Bus Matrix component). This can lower the total gate count.
- Added new output port for connection to the AHB Trace Macrocell (HTM, a CoreSight debug component from ARM) for complex data trace operations.
- Debug components or debug control registers can be accessed even during system reset; only during power-on reset are those registers inaccessible.

- The TPIU has SWV operation mode support. This allows trace information to be captured with low-cost hardware.
- In revision 1, the VECTPENDING field in the NVIC Interrupt Control and Status register can be affected by the C_MASKINTS bit in the NVIC Debug Halting Control and Status register. If C_MASKINTS is set, the VECTPENDING value could be zero if the mask is masking a pending interrupt.
- The JTAG-DP debug interface module has been changed to the SWJ-DP module (see the next section, “Revision 1 Change: Moving from JTAG-DP to SWJ-DP”). Chip manufacturers can continue to use JTAG-DP, which is still a product in the CoreSight product family.

Since revision 0 of the Cortex-M3 does not have a double word stack alignment feature in its exception sequence, some compiler tools, such as ARM RealView Development Suite (RVDS) and the KEIL RealView Microcontroller Development Kit, have special options to allow software adjustment of stacking, which allows the developed application to be EABI compliant. This could be important if it has to work with other EABI-compliant development tools.

To determine which revision of the Cortex-M3 processor is used inside the microcontroller or SoC, you can use the CPU ID Base Register in the NVIC. The last 4 bits of this register contain the revision number, as shown in Table 17.1.

Table 17.1 CPU ID Base Register (0xE000ED00)

	Implementer [31:24]	Variant [23:20]	Constant [19:16]	PartNo [15:4]	Revision [3:0]
Revision 0 (r0p0)	0x41	0x0	0xF	0xC23	0x0
Revision 1 (r1p0)	0x41	0x0	0xF	0xC23	0x1
Revision 1 (r1p1)	0x41	0x1	0xF	0xC23	0x1

Individual debug components inside the Cortex-M3 processor also carry their own ID registers, and the revision field might also be different between revision 0 and revision 1.

Revision 1 Change: Moving from JTAG-DP to SWJ-DP

The JTAG-DP provided in some earlier Cortex-M3 products is replaced with the SWJ-DP. The Serial-Wire JTAG Debug Port (SWJ-DP) combines the function of the SW-DP and the JTAG-DP, and with automatic protocol detection. Using this component, a Cortex-M3 device can support debugging with both SW and JTAG interfaces.

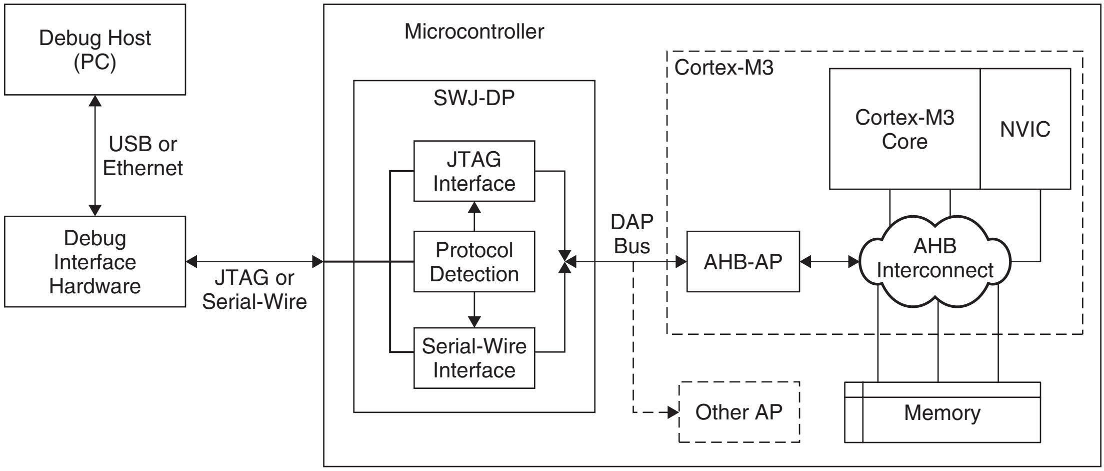


Figure 17.1 SWJ-DP: Combining JTAG-DP and SW-DP Functionalities

Development Tools

To start using the Cortex-M3, you'll need a number of tools. Typically they will include:

- A compiler and/or assembler: Software to compile your C or assembler application codes. Almost all C compiler suites come with an assembler.
- Instruction set simulator: Software to simulate the instruction execution for debugging in early stages of software development.
- In-circuit emulator (ICE) or debug probe: A hardware device to connect your debug host (usually a PC) to the target circuit. The interface can be either JTAG or SW.
- A development board: A circuit board that contains the microcontroller.
- Trace capture: An optional hardware and software package for capturing instruction traces or output from DWT and ITM modules and outputs them to human-readable format.
- An embedded operating system: An operating system running on the microcontroller. This is optional; many applications do not require an OS.

C Compiler

A number of C compiler suites and development tools are already available for the Cortex-M3 (see Table 17.2).

The GNU C Compiler from CodeSourcery provides a free solution. At this writing, the main GNU C Compiler (GCC) does not have Cortex-M3 support; however, this support will be merged into the main GCC in the near future. You can also get evaluation versions of some commercial tools such as RealView-MDK.

Table 17.2 Examples of Development Tools Supporting Cortex-M3

Company	Product[1]
ARM (www.arm.com)	The Cortex-M3 is supported from RealView Development Suite 3.0 (RVDS). RealView-ICE (RVI) version 1.5 is available for connecting debug target to debug environment. Note that older products such as ADS and SDT do not support the Cortex-M3.
KEIL (an ARM company; www.keil.com)	The Cortex-M3 is supported in RealView Microcontroller Development Kit (RealView-MDK). The ULINK(TM) USB-JTAG adapter is available for connecting debug target to debug IDE.
CodeSourcery (www.codesourcery.com)	GNU Tool Chain for ARM Processors is now available at www.codesourcery.com/gnu_toolchains/arm/. It is based on GNU C Compiler 4.1.0 and supports the Cortex-M3.
Rowley Associates (www.rowley.co.uk)	CrossWorks for ARM is a GNU C Compiler-based development suite supporting the Cortex-M3 (www.rowley.co.uk/arm/index.htm).
IAR Systems (www.iar.com)	IAR Embedded Workbench for ARM and Cortex provides a C/C++ compiler and debug environment. (v4.40 or above). A KickStart kit is also available, based on the Luminary Micro LM3S102 microcontroller, including debugger and a J-Link Debug Probe for connecting the target board to debug IDE.
Lauterbach (www.lauterbach.com)	JTAG debugger and trace utilities are available from Lauterbach.

Embedded Operating System Support

Many applications require an OS. Many OSs are developed for the embedded market. Currently, a number of these OSs are supported on the Cortex-M3 (see Table 17.3).

Table 17.3 Examples of Embedded Operating Systems Supporting Cortex-M3

Company	Product[2]
FreeRTOS (www.freertos.org)	FreeRTOS
Express Logic (www.expresslogic.com)	ThreadX(TM) RTOS
Micrium (www.micrium.com)	µC/OS-II
Accelerated Technology (www. Acceleratedtechnology.com)	Nucleus
Pumpkin Inc. (www.pumpkininc.com)	Salvo RTOS
CMX Systems (www.cmx.com)	CMX-RTX
Keil (www.keil.com)	ARTX-ARM
Segger (www.segger.com)	embOS
IAR Systems (www.iar.com)	IAR PowerPac for ARM

[1] Product names are registered trademarks of the companies listed on the left-hand side of the table.

[2] Product names are registered trademarks of the companies listed on the left-hand side of the table.

This page intentionally left blank

CHAPTER 18

Porting Applications from the ARM7 to the Cortex-M3

In This Chapter:

- Overview
- System Characteristics
- Assembly Language Files
- C Program Files
- Precompiled Object Files
- Optimization

Overview

For many engineers, porting existing program code to new architecture is a typical task. With the Cortex-M3 products starting to emerge on the market, many of us have to face the challenge of porting ARM7TDMI (referred to as ARM7 in the following text) code to the Cortex-M3. This chapter evaluates a number of aspects involved in porting applications from the ARM7 to the Cortex-M3.

There are several areas to consider when you're porting from the ARM7 to the Cortex-M3:

- System characteristics
- Assembly language files
- C language files
- Optimization

Overall, low-level code such as hardware control, task management, and exception handlers requires the most changes, whereas application codes normally can be ported with minor modification and recompiling.

System Characteristics

There are a number of system characteristic differences between ARM7-based systems and Cortex-M3 based systems (for example, memory map, interrupts, MPU, system control, and operation modes.)

Memory Map

The most obvious target of modification in porting programs between different microcontrollers is their memory map differences. In the ARM7, memory and peripherals can be located in almost any address, whereas the Cortex-M3 processor has a predefined memory map. Memory address differences are usually resolved in compile and linking stages. Peripheral code porting could be more time consuming because the programmer model for the peripheral could be completely different. In that case, device driver codes might need to be completely rewritten.

Many ARM7 products provide a memory remap feature so that the vector table can be remapped to the SRAM after boot-up. In the Cortex-M3, the vector table can be relocated using the NVIC register so that memory remapping is no longer needed. Therefore, the memory remap feature might be unavailable in many Cortex-M3 products.

Big endian support in the ARM7 is different from such support in the Cortex-M3. Program files can be recompiled to the new big endian system, but hardcoded lookup tables might need to be converted during the porting process.

In ARM720T, and some later ARM processors like ARM9, a feature called high vector is available, which allows the vector table to be located to 0xFFFF0000. This feature is for supporting Windows CE and is not available in the Cortex-M3.

Interrupts

The second target is the difference in the interrupt controller being used. Program code to control the interrupt controller, such as enabling or disabling interrupts, will need to be changed. In addition, new code is required for setting up interrupt priority levels and vector addresses for various interrupts.

The interrupt return method is also changed. This requires modification of interrupt return in assembler code or, if C language is used, it might be necessary to make adjustments on compile directives.

Enable and disable of interrupts, previously done by modifying CPSR, must be replaced by setting up the interrupt mask register.

In the Cortex-M3, some registers are automatically saved by the stacking and unstacking mechanism. Therefore, some of the software stacking operations could be reduced or removed. However, in the case of the FIQ handler, traditional ARM cores have separate registers for FIQ (R8-R11). Those registers can be used by the FIQ without the need to push them into the stack. However, in the Cortex-M3, these registers are not stacked automatically, so when an FIQ handler is ported to the Cortex-M3, either the registers being used by the handler must be changed or a stacking step will be needed.

Code for nest interrupt handling can be removed. In the Cortex-M3, the NVIC has built-in nested interrupt handling.

There are also differences in error handling. The Cortex-M3 provides various fault status registers so that the cause of faults can be located. In addition, new fault types are defined in the Cortex-M3 (for example, stacking and unstacking faults, memory management faults, and hard faults). Therefore, fault handlers will need to be rewritten.

MPU

The MPU programming model is another system block that needs new program code set up. Microcontroller products based on the ARM7TDMI/ARM7TDMI-S do not have MPUs, so moving the application code to the Cortex-M3 should not be a problem. However, products based on the ARM720T have a Memory Management Unit (MMU), which has different functionalities to the MPU in Cortex-M3. If the application needs to use the MMU (as in a virtual memory system), it cannot be ported to the Cortex-M3.

System Control

System control is another key area to look into when you're porting applications. The Cortex-M3 has built-in instructions for entering sleep mode. In addition, the system controller inside Cortex-M3 products is likely to be completely different from that of the ARM7 products, so function code that involves system management features will need to be rewritten.

Operation Modes

In the ARM7 there are seven operation modes; in the Cortex-M3 these have been changed to difference exceptions (see Table 18.1).

Table 18.1 Mapping of ARM7TDMI Exceptions and Modes to Cortex-M3

Modes and Exceptions in the ARM7	Corresponding Modes and Exceptions in the Cortex-M3
Supervisor (default)	Privileged, Thread
Supervisor (software interrupt)	Privileged, SVC
FIQ	Privileged, interrupt
IRQ	Privileged, interrupt
Abort (prefetch)	Privileged, bus fault exception
Abort (data)	Privileged, bus fault exception
Undefined	Privileged, usage fault exception
System	Privileged, Thread
User	User access (nonprivileged), Thread

The FIQ in the ARM7 can be ported as a normal IRQ in the Cortex-M3 because in the Cortex-M3, we can set up the priority for a particular interrupt to be highest; thus it will be able to preempt other exceptions, just like the FIQ in the ARM7. However, due to the difference between banked FIQ registers in the ARM7 and the stacked registers in the Cortex-M3, the registers being used in the FIQ handler must be changed, or the registers used by the handler must be saved to the stack manually.

FIQ and NMI

Many engineers might expect the FIQ in the ARM7 to be directly mapped to the NMI in the Cortex-M3. In some applications it is possible, but a number of differences between the FIQ and the NMI need special attention when you're porting applications using the NMI as an FIQ.

First, the NMI cannot be disabled, whereas on the ARM7, the FIQ can be disabled by setting the F-bit in the CPSR. So it is possible in the Cortex-M3 for an NMI handler to start right at boot-up time, whereas in the ARM7, the FIQ is disabled at reset.

Second, you cannot use SVC in an NMI handler on the Cortex-M3, whereas you can use SWI in an FIQ handler on the ARM7. During execution of an FIQ handler on the ARM7, it is possible for other exceptions to take place (except IRQ, because the I-bit is set automatically when the FIQ is served). However, on the Cortex-M3, a fault exception inside the NMI handler can cause the processor to lock up.

Assembly Language Files

Porting assembly files depends on whether the file is for ARM state or Thumb state.

Thumb State

If the file is for Thumb state, the situation is much easier. In most cases the file can be reused without a problem. However, a few Thumb instructions in the ARM7 are not supported in the Cortex-M3:

- Any code that tries to switch to ARM state
- SWI is replaced by SVC (note that the usage model is changed as well)

Finally, make sure that the program accesses the stack only in full descending stack operations. It is possible, though uncommon, to implement a different stacking model differently (for example, full ascending) in ARM7TDMI.

ARM State

The situation for ARM code is more complicated. There are several scenarios:

- Vector table: In the ARM7, the vector table starts from address 0x0 and consists of branch instructions. In the Cortex-M3, the vector table contains the initial value for the stack pointer and reset vector address, followed by addresses of exception handlers. Due to these differences, the vector table will need to be completely rewritten.
- Register initialization: In the ARM7, it is often necessary to initialize different registers for different modes. For example, there are banked stack pointers (R13), a link register (R14), and a Saved Program Status Register (SPSR) in the ARM7. Since the Cortex-M3 has a different programmer's model, the register initialization code will have to be changed. In fact, the register initialization code on the Cortex-M3 will be much simpler because there is no need to switch the processor into a different mode.
- Mode switching and state switching code: Since the operation mode definition in the Cortex-M3 is different from that of the ARM7, the code for mode switching needs to be removed. The same applies to ARM/Thumb state switching code.
- Interrupt enabling and disabling: In the ARM7, interrupts can be enabled or disabled by clearing or setting the I-bit in the CPSR. In the Cortex-M3, this is done by clearing or setting an interrupt mask register such as PRIMASK or FAULTMASK. Furthermore, there is no F-bit in the Cortex-M3 because there is no FIQ input.
- Coprocessor accesses: There is no coprocessor support on the Cortex-M3, so this kind of operation cannot be ported.
- Interrupt handler and interrupt return: In the ARM7, the first instruction of the interrupt handler is in the vector table, which normally contains a branch instruction to the actual interrupt handler. In the Cortex-M3, this step is no longer needed. For interrupt returns, the ARM7 relies on manual adjustment of the return program counter. In the Cortex-M3, the correctly adjusted program counter is saved into the stack and the interrupt return is triggered by loading EXC_RETURN into the program counter. Instructions such as MOVS and SUBS should not be used as interrupt returns on the Cortex-M3. Due to these differences, interrupt handlers and interrupt return codes need modification during porting.
- Nested interrupt support code: In the ARM7, when a nested interrupt is needed, usually the IRQ handler will need to switch the processor to system mode and re-enable the interrupt. This is not required in the Cortex-M3.
- FIQ handler: If an FIQ handler is to be ported, you might need to add an extra step to save the contents of R8–R11 to stack memory. In the ARM7, R8–R12 are banked, so

the FIQ handler can skip the stack push for these registers. However, on the Cortex-M3, R0–R3 and R12 are saved onto the stack automatically, but R8–R11 are not.

- Software interrupt (SWI) handler: The SWI is replaced with an SVC. However, when porting an SWI handler to an SVC, the code to extract the passing parameter for the SWI instruction needs to be updated. The calling SVC instruction address can be found in the stacked PC, which is different from the SWI in the ARM7, where the program counter address has to be determined from the Link Register.
- SWAP instruction (SWP): There is no swap instruction in the Cortex-M3. If the swap instruction was used for semaphores, the exclusive access instructions should be used as replacement. This requires rewriting the semaphores code. If the instruction was used purely for data transfers, this can be replaced by multiple memory access instructions.
- Access to CPSR, SPSR: The CPSR in the ARM7 is replaced with xPSR in the Cortex-M3, and the SPSR has been removed. If the application would like to access the current values of processor flags, the program code can be replaced with read access to the APSR. If an exception handler would like to access the PSR before the exception takes place, it can find the value in the stack memory because the value of xPSR is automatically saved to the stack when an interrupt is accepted. So there is no need for an SPSR in the Cortex-M3.
- Conditional execution: In the ARM7, conditional execution is supported for many ARM instructions, whereas most Thumb-2 instructions do not have the condition field inside the instruction coding. When porting these codes to the Cortex-M3, in some cases we can use the IF-THEN instruction block; otherwise we might need to insert branches to produce conditionally executed codes. One potential issue with replacing conditional execution code with IT instruction blocks is that it could increase the code size and, as a result, could cause minor problems, such as load/store operations in some part of the program could exceed the access range of the instruction.
- Use of the Program Counter value in code that involves calculation using the current program counter: In running ARM code on the ARM7, the read value of the PC during an instruction is the address of the instruction plus 8. This is because the ARM7 has three pipeline stages and, when reading the PC during the execution stage, the program counter has already incremented twice, 4 bytes at a time. When porting code that processes the PC value to the Cortex-M3, since the code will be in Thumb, the offset of the program counter will only be 4.
- Use of the R13 value: In the ARM7, the stack pointer R13 has 32 bits; in the Cortex-M3 processor, the lowest 2 bits of the stack pointer are always forced to zero. Therefore, in the unlikely case that R13 is used as a data register, the code has to be modified because the lowest 2 bits would be lost.

For the rest of the ARM program code, we can try to compile it as Thumb/Thumb-2 and see if further modifications are needed. For example, some of the pre-index and post-index memory access instructions in the ARM7 are not supported in the Cortex-M3 and have to be recoded into multiple instructions. Some of the code might have long branch range or large immediate data values that cannot be compiled as Thumb code and so must be modified to Thumb-2 code manually.

C Program Files

Porting C program files is much easier than porting assembly files. In most cases, application code in C can be recompiled for the Cortex-M3 without a problem. However, there are still a few areas that potentially need modification:

- Inline assemblers: Some C program code might have inline assembly code that needs modification. This code can be easily located via the *__asm* keyword. If RVDS/RVCT 3.0 or later is used, it should be changed to Embedded Assembler.
- Interrupt handler: In the C program you can use *__irq* to create interrupt handlers that work with the ARM7. Due to the difference between the ARM7 and the Cortex-M3 interrupt behaviors such as saved registers and interrupt returns, depending on development tools being used, the __irq keyword might need to be removed. (However, in RVDS 3.0 and RVCT 3.0, support for the Cortex-M3 is added to the *__irq*, and use of the *__irq* directive is recommended for reasons of clarity.)

Precompiled Object Files

Most C compilers will provide precompiled object files for various function libraries and startup code. Some of those (such as startup code for traditional ARM cores) cannot be used on the Cortex-M3 due to the difference in operation modes and states. A number of them will have source code available and can be recompiled using Thumb-2 code. Refer to your tool vendor documentation for details.

Optimization

After getting the program to work with the Cortex-M3, you might be able to further improve it to obtain better performance and lower memory use. A number of areas should be explored:

- Use of the Thumb-2 instruction: For example, if a 16-bit Thumb instruction transfers data from one register to another and then carries a data processing operation on it, it might be possible to replace the sequence with a single Thumb-2 instruction. This can reduce the number of clock cycles required for the operation.

- Bit band: If peripherals are located in bit-band regions, access to control register bits can be greatly simplified by accessing the bit via a bit-band alias.
- Multiply and divide: Routines that require divide operations, such as converting values into decimals for display, can be modified to use the divide instructions in the Cortex-M3. For multiplication of larger data, the multiple instructions in the Cortex-M3 such as UMULL, SMULL, MLA, MLS, UMLAL, and SMLAL can be used to reduce complexity of the code.
- Immediate data: Some of the immediate data that cannot be coded in Thumb instructions can be produced using Thumb-2 instructions.
- Branches: Some of the longer distance branches that cannot be coded in Thumb code (usually ending up with multiple branch steps) can be coded with Thumb-2 instructions.
- Boolean data: Multiple Boolean data (either 0 or 1) can be packed into a single byte/half word/word in bit-band regions to save memory space. They can then be accessed via the bit-band alias.
- Bit-field processing: The Cortex-M3 provides a number of instructions for bit-field processing, including UBFX, SBFX, BFI, BFC, and RBIT. They can simplify many program codes for peripheral programming, data packet formation, or extraction and serial data communications.
- IT instruction block: Some of the short branches might be replaceable by the IT instruction block. By doing that we could avoid wasting clock cycles when the pipeline is flushed during branching.
- ARM/Thumb state switching: In some situations, ARM developers divide code into various files so that some of them can be compiled to ARM code and others compiled to Thumb code. This is usually needed to improve code density where execution speed is not critical. With Thumb-2 features in the Cortex-M3, this step is no longer needed, so some of the state switching overhead can be removed, producing short code, less overhead, and possibly fewer program files.

CHAPTER 19

Starting Cortex-M3 Development Using the GNU Tool Chain

In This Chapter:

- Background
- Getting the GNU Tool Chain
- Development Flow
- Examples
- Accessing Special Registers
- Using Unsupported Instructions
- Inline Assembler in the GNU C Complier

Background

Many people use the GNU tool chain for ARM product development, and a number of development tools for ARM are based on the GNU tool chain. The GNU tool chain supports the Cortex-M3 and is currently freely available from CodeSourcery (www.codesourcery.com). The main GNU C Compiler development will include support for the Cortex-M3 in the near future.

This chapter introduces only the most basic steps in using the GNU tool chain. Detailed uses of the tool chain are available on the Internet and are outside the scope of this book.

Assembler syntax for GNU assembler (AS in the GNU tool chain) is a bit different from ARM assembler. These differences include declarations, compile directives, comments, and the like. Therefore, assembly codes for ARM RealView Development tools need modification before being used with the GNU tool chain.

Getting the GNU Tool Chain

The compiled version of the GNU tool chain can be downloaded from www.codesourcery.com/gnu_toolchains/arm/. A number of binary builds are available. For the most simple uses, let's select one with EABI[1] and without a specific embedded OS as the target platform. The tool chain is available for various development platforms such as Windows and Linux. The examples shown in this chapter should work with either version.

Development Flow

As with ARM tools, the GNU tool chain contains a compiler, an assembler, and a linker. The tools allow projects that contain source code in both C and assembly language (see Figure 19.1).

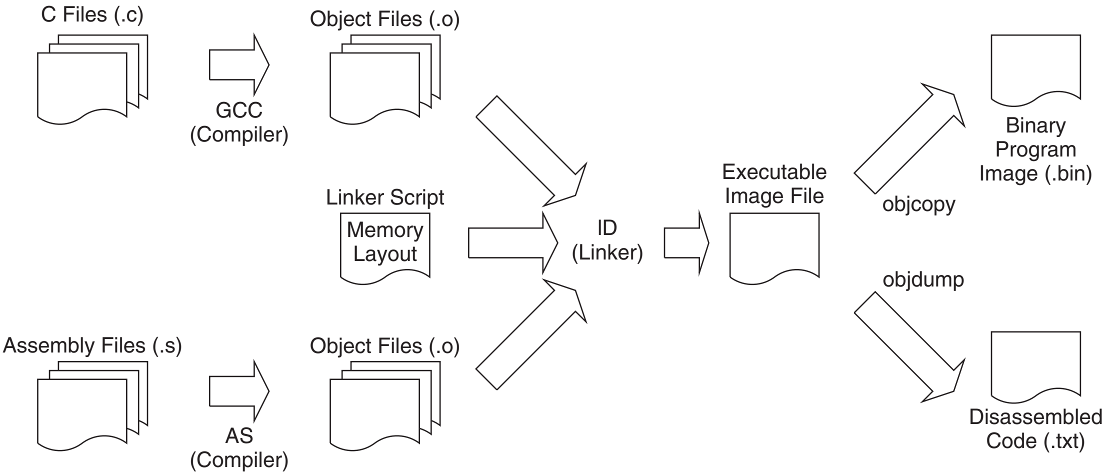


Figure 19.1 Example Development Flow Based on the GNU Tool Chain

There are versions of the tool chain for different application environments (Symbian, Linux, EABI, and so on). The filenames of the programs usually have a prefix, depending on your tool chain target options. For example, if the EABI environment is used, the GCC command could be *arm-xxxx-eabi-gcc*. The following examples use the commands from the CodeSourcery GNU ARM Tool Chain shown in Table 19.1.

Table 19.1 Command Name of CodeSourcery Tool Chain

Function	Command (EABI Version)
Assembler	*arm-none-eabi-as*
C Compiler	*arm-none-eabi-gcc*
Linker	*arm-none-eabi-ld*
Binary image generator	*arm-none-eabi-objcopy*
Disassembler	*arm-none-eabi-objdump*

Notice how command names of tool chains from other vendors differ.

[1] Embedded Application Binary Interface (EABI) for the ARM architecture—executables must conform to this specification in order for them to be used with various development tool sets.

The linker script in the development flow is optional but often required when the memory map becomes more complex.

Examples

Let's look at a few examples using the GNU tool chain.

Example 1: The First Program

For a start, let's try a simple assembly program that we covered in Chapter 10 that calculates 10 + 9 + 8 .. +1:

```
========== example1.s ==========
/* define constants */
      .equ      STACK_TOP, 0x20000800
      .text
      .global _start
      .code 16
      .syntax unified
      /* .thumbfunc  */
      /* .thumbfunc is only needed with CodeSourcery GNU tool chain
       prior to 2006Q3-26*/
_start:
      .word STACK_TOP, start
      .type start, function
       /* Start of main program */
start:
      movs  r0, #10
      movs  r1, #0
      /* Calculate 10+9+8...+1 */
loop:
      adds  r1, r0
      subs  r0, #1
      bne   loop
      /* Result is now in R1 */
deadloop:
      b     deadloop
      .end
========== end of file ==========
```

- The *.word* directive here helps us define the starting stack pointer value as 0x20000800 and the reset vector as start.
- *.text* is a predefined directive indicating that it is a program region that needs to be assembled.

- *.global* allows the label *_start* to be shared with other object files if needed.
- *.code 16* indicates that the program code is in Thumb.
- *.syntax unified* indicates that the unified assembly language syntax is used.
- *_start* is a label indicating the starting point of the program region.
- *start* is a separate label indicating the reset handler.
- *.type start, function* declares that the symbol *start* is a function. This is necessary for all the exception vectors in the vector table. Otherwise the assembler will set the LSB of the vector to zero.
- *.end* indicates the end of this program file.

Unlike ARM assembler, labels in GNU assemblers are followed by a colon (:). Comments are quoted with /* and */, and directives are prefixed by a period (.).

Notice that the reset vector (start) is defined as a function (.type start function) within thumb code (.code 16). The reason for this is to force the LSB of the reset vector to 1 to indicate that it starts in Thumb state. Otherwise, the processor will try starting in ARM mode, resulting in a hard fault. To assemble this file, we can use *as*, as in the following command:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
```

This creates the object file example1.o. The options *-mcpu* and *-mthumb* define the instruction set to be used. The linking stage can be done by *ld* as follows:

```
$> arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
```

Then the binary file can be created using Object Copy (*objcopy*) as follows:

```
$> arm-none-eabi-objcopy -Obinary example1.out example1.bin
```

We can examine the output by creating a disassembled code listing file using Object Dump (*objdump*):

```
$> arm-none-eabi-objdump -S example1.out > example1.list
```

which looks like:

```
example1.out:     file format elf32-littlearm
Disassembly of section .text:

00000000 <_start>:
   0: 0800        lsrs  r0, r0, #32
   2: 2000        movs  r0, #0
   4: 0009        lsls  r1, r1, #0
      ...
```

```
00000008 <start>:
   8: 200a        movs  r0, #10
   a: 2100        movs  r1, #0

0000000c <loop>:
   c: 1809        adds  r1, r1, r0
   e: 3801        subs  r0, #1
  10: d1fc        bne.n c <loop>

00000012 <deadloop>:
  12: e7fe        b.n   12 <deadloop>
```

Example 2: Linking Multiple Files

As mentioned before, we can create multiple object files and link them together. Here we have an example of two assembly files, example2a.s and example2b.s; example2a.s contains the vector table only, and example2b.s contains the program code. The *.global* is used to pass the address from one file to another:

```
========== example2a.s ==========
/* define constants */

      .equ     STACK_TOP, 0x20000800
      .global vectors_table
      .global start
      .global nmi_handler
      .code 16
      .syntax unified

vectors_table:
      .word STACK_TOP, start, nmi_handler, 0x00000000
      .end
========== end of file ==========

========== example2b.s ==========
/* Main program */
      .text
      .global _start
      .global start
      .global nmi_handler
      .code 16
      .syntax unified
      .type start, function
      .type nmi_handler, function
_start:
      /* Start of main program */
```

```
start:
      movs  r0, #10
      movs  r1, #0
      /* Calculate 10+9+8...+1 */
loop:
      adds  r1, r0
      subs  r0, #1
      bne   loop
      /* Result is now in R1 */
deadloop:
      b     deadloop
      /* Dummy NMI handler for illustration */
nmi_handler:
      bx    lr
      .end
========== end of file ==========
```

To create the executable image, the following steps are used:

1. Assemble example2a.s:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2a.s -o example2a.o
```

2. Assemble example2b.s:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2b.s -o example2b.o
```

3. Link the object files to a single image. Note that the order of the object files in the command line will affect the order of the objects in the final executable image:

```
$> arm-none-eabi-ld -Ttext 0x0 -o example2.out example2a.o example2b.o
```

4. The binary file can then be generated:

```
$> arm-none-eabi-objcopy -Obinary example2.out example2.bin
```

5. As in the previous example, we generate a list file to check that we have a correctly assembled image:

```
$> arm-none-eabi-objdump -S example2.out > example2.list
```

As the number of files increases, the compile process can be simplified using a UNIX *makefile*. Individual development suites may also have built-in facilities to make the compile process easier.

Example 3: A Simple "Hello World" Program

To be a bit more ambitious, let's now try the "Hello World" program. (*Note:* We skipped the UART initialization here; you need to add your own UART initialization code to try this example. A example of UART initialization in C language is provided in Chapter 20.)

```
========== example3a.s ==========
/* define constants */
      .equ      STACK_TOP, 0x20000800
      .global vectors_table
      .global _start
      .code 16
      .syntax unified
vectors_table:
      .word STACK_TOP, _start
      .end
========== end of file ==========
```

```
========== example3b.s ==========
      .text
      .global _start
      .code 16
      .syntax unified
      .type _start, function
_start:
      /* Start of main program */
      movs    r0, #0
      movs    r1, #0
      movs    r2, #0
      movs    r3, #0
      movs    r4, #0
      movs    r5, #0

      ldr    r0,=hello
      bl     puts
      movs    r0, #0x4
      bl      putc
deadloop:
      b     deadloop
hello:
      .ascii  "Hello\n"
      .byte   0
      .align

puts: /* Subroutine to send string to UART */
      /* Input r0 = starting address of string */
      /*  The string should be null terminated */
      push  {r0, r1, lr} /* Save registers */
      mov   r1, r0       /* Copy address to R1, because */
                         /* R0 will be used as input for */
                         /* putc */
```

```
putsloop:
      ldrb.w r0,[r1],#1   /* Read one character and increment address */
      cbz   r0, putsloopexit /* if character is null, goto end */
      bl    putc
      b     putsloop
putsloopexit:
      pop   {r0, r1, pc}  /* return */

.equ  UART0_DATA, 0x4000C000
.equ  UART0_FLAG, 0x4000C018

putc: /* Subroutine to send a character via UART */
      /* Input R0 = character to send */
      push  {r1, r2, r3, lr}  /* Save registers */
      LDR   r1,=UART0_FLAG
putcwaitloop:
      ldr   r2,[r1]        /* Get status flag */
      tst.w   r2, #0x20    /* Check transmit buffer full flag bit */
      bne   putcwaitloop   /* If busy then loop   */
      ldr   r1,=UART0_DATA    /* otherwise output data to transmit
      buffer */
      str   r0, [r1]
      pop   {r1, r2, r3, pc}  /* Return */
      .end
========== end of file ==========
```

In this example we used *.ascii* and *.byte* to create a null terminated string. After defining the string, we used *.align* to ensure that the next instruction will start in the right place. Otherwise the assembler might put the next instruction in an unaligned location.

To compile the program, create the binary image, and disassemble outputs, the following steps can be used:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example3a.s -o example3a.o
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example3b.s -o example3b.o
$> arm-none-eabi-ld -Ttext 0x0 -o example3.out example3a.o example3b.o
$> arm-none-eabi-objcopy -Obinary example3.out example3.bin
$> arm-none-eabi-objdump -S example3.out > example3.list
```

Example 4: Data in RAM

Very often we will data store in SRAM. The following simple example shows the required setup:

```
========== example4.s ==========
      .equ      STACK_TOP, 0x20000800
      .text
      .global _start
      .code 16
      .syntax unified
_start:
      .word STACK_TOP, start
      .type start, function
       /* Start of main program */
start:
      movs  r0, #10
      movs  r1, #0
      /* Calculate 10+9+8...+1 */
loop:
      adds  r1, r0
      subs  r0, #1
      bne   loop
      /* Result is now in R1 */
      ldr   r0,=Result
      str   r1,[r0]
deadloop:
      b     deadloop

/* Data region */
      .data
Result:
      .word 0
      .end
========== end of file ==========
```

In the program, the *.data* directive is used to create a data region. Inside this region, a *.word* directive is used to reserved a space labeled *Result*. The program code can then access this space using the defined label *Result*.

To link this program, we need to tell the linker where the RAM is. This can be done using the *-Tdata* option, which sets the data segment to the required location:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example4.s -o example4.o
$> arm-none-eabi-ld -Ttext 0x0 -Tdata 0x20000000 -o example4.out
  example4.o
$> arm-none-eabi-objcopy -Obinary –R .data example4.out example4.bin
$> arm-none-eabi-objdump -S example4.out > example4.list
```

Also notice that the *–R .data* option is used in running *objcopy* in this example. This prevents the data memory region from being included in the binary output file.

Example 5: C Only, Without Assembly File

One of the main components in the GNU tool chain is the C compiler. In this example, the whole executable—even the reset vector and stack pointer initial value— is coded using C.

In addition, a linker script is needed to put the segments in place. First, let's look at the C program file:

```
========== example5.c ==========
#define STACK_TOP 0x20000800
#define NVIC_CCR ((volatile unsigned long *)(0xE000ED14))
// Declare functions
void myputs(char *string1);
void myputc(char mychar);
int  main(void);
void nmi_handler(void);
void hardfault_handler(void);
// Define the vector table
__attribute__ ((section("vectors")))
void (* const VectorArray[])(void) = {
  STACK_TOP,
  main,
  nmi_handler,
  hardfault_handler
  };

// Start of main program
int main(void)
{
const char *helloworld[]="Hello world\n";
*NVIC_CCR = *NVIC_CCR | 0x200; /* Set STKALIGN in NVIC */
myputs(*helloworld);
while(1);
return(0);
}

// Functions
void myputs(char *string1)
{
char mychar;
int j;
j=0;
do {
  mychar = string1[j];
  if (mychar!=0) {
    myputc(mychar);
    j++;
    }
  } while (mychar != 0);
return;
}
```

```
void myputc(char mychar)
{
#define UART0_DATA ((volatile unsigned long *)(0x4000C000))
#define UART0_FLAG ((volatile unsigned long *)(0x4000C018))

// Wait until busy flag is clear
while ((*UART0_FLAG & 0x20) != 0);
// Output character to UART
*UART0_DATA = mychar;
return;
}

//Dummy handlers
void nmi_handler(void)
{
  return;
}
void hardfault_handler(void)
{
  return;
}
========== end of file ==========
```

The vector table is defined using the *__attribute__* code. This file does not say where the vector table is; that's the job of the linker script. A simple linker script can be something like the following simple.ld:

```
========== simple.ld ==========
/* MEMORY command : Define allowed memory regions       */
/* This part define various memory regions that the     */
/* linker is allowed to put data into.  This is an      */
/* optional feature, but useful because the linker can */
/* warn you when your program is too big to fit.        */
MEMORY
  {
  /* ROM is a readable (r), executable region (x)       */
  rom (rx)    : ORIGIN = 0, LENGTH = 2M

  /* RAM is a readable (r), writable (w) and            */
  /* executable region (x)                              */
  ram (rwx)   : ORIGIN = 0x20000000, LENGTH = 4M
  }

/* SECTION command : Define mapping of input sections   */
/* into output sections.                                */
```

```
SECTIONS
  {
  . = 0x0;          /* From 0x00000000 */
  .text : {
    *(vectors)      /* Vector table */
    *(.text)        /* Program code */
    *(.rodata)      /* Read only data */
    }
  . = 0x20000000;   /* From 0x20000000 */
  .data : {
    *(.data)        /* Data memory */
    }
  .bss : {
    *(.bss)         /* Zero-filled run time allocate data memory */
    }
  }
========== end of file ==========
```

The memory map information is then passed on to the compiler during the compile stage:

```
$> arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb example5.c -nostartfiles
  -T simple.ld -o example5.o
```

The output object file can then be linked, again, using the linker script:

```
$> arm-none-eabi-ld -T simple.ld -o example5.out example5.o
```

In this case we only have one source file, so the linking stage can be omitted. Finally, the binary and disassembled list file can be generated:

```
$> arm-none-eabi-objcopy -Obinary example5.out example5.bin
$> arm-none-eabi-objdump -S example5.out   > example5.list
```

In this example we used a compiler option called *-nostartfiles*. This prevents the C compiler from inserting startup library functions into the executable image. One of the reasons for doing this is to reduce the size of the program image. However, the main reason to use this option is that the startup library code of the GNU tool chain is dependent on the suppliers of the distributions. Some of them might not be suitable for the Cortex-M3; they might be compiled for traditional ARM processors such as the ARM7 (using ARM code instead of Thumb code).

But in many cases, depending on the applications and libraries used, it would be necessary to use the startup library to carry out initialization processes such as initialization of the data regions (for example, regions of data that should be initialized to zero before running the application). The next example shows a simple setup for this.

Example 6: C Only, with Standard C Startup Code

In normal situations, the standard C library startup code is automatically included in the output when a C program is compiled. This ensures that run-time libraries are initialized correctly. The C library startup code is provided by the GNU tool chain; however, the setup might vary between different tool chain providers. The following example is based on CodeSourcery GNU ARM Tool Chain version 2006q3-26. For this version, you need to contact CodeSourcery support to get the correct startup code object file, armv7m-crt0.o, because this version provides an incorrect startup code compiled in ARM code rather than Thumb code. This problem is fixed in version 2006q3-27 or after. Versions of the GNU tool chain from different vendors can have different startup code implementations and different filenames. Check the documentation from your tool chain to determine the best arrangement for defining the startup code.

Before we compile the C source code, the C program in Example 5 requires several small modifications. By default, the startup code armv7m-crt0 already contains a vector table, and it has the NMI handler and hard fault handler names defined as *_nmi_isr* and *_fault_isr*, respectively. As a result, we need to remove our vector table from the C code and rename the NMI and hard fault handlers:

```
========== example6.c ==========
// Declare functions
void myputs(char *string1);
void myputc(char mychar);
int  main(void);
void _nmi_isr(void);
void _fault_isr(void);

// Start of main program
int main(void)
{
const char *helloworld[]={“Hello world\n”};

myputs(*helloworld);
while(1);
return(0);
}

// Functions
void myputs(char *string1)
{
char mychar;
int j=0;
do {
  mychar = string1[j];
  if (mychar!=0) {
    myputc(mychar);
    j++;
    }
  } while (mychar != 0);

return;
}

void myputc(char mychar)
{
#define UART0_DATA ((volatile unsigned long *)(0x4000C000))
#define UART0_FLAG ((volatile unsigned long *)(0x4000C018))

// Wait until busy flag is clear
while ((*UART0_FLAG & 0x20) != 0);
// Output character to UART
*UART0_DATA = mychar;
return;
}

//Dummy handlers
void _nmi_isr(void)
{
  return;
}
void _fault_isr(void)
{
  return;
}
========== end of file ==========
```

A number of linker scripts are already included in the CodeSourcery installation. They can be located in the codesourcey/sourcery g++/arm-none-eabi/lib directory. In the following example, the file lm3s8xx-rom.ld is used. This linker script supports the Luminary Micro LM3S8XX series devices.

Aside from the current directory, when the C program code is located, a library subdirectory called *lib* is also created in the current directory. This makes the library search path setup easier. The startup code object file armv7m-crt0.o and the required linker script are copied to this lib directory, and in the following examples, the *-L lib* option defines directory *lib* as the library search path.

Now we can compile the C program:

```
$> arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb example6.c -L lib –T
  lm3s8xx-rom.ld -o example6.out
```

This creates and links the output object file as example6.out. Since there is only one object file, the binary file can be directly generated:

```
$> arm-none-eabi-objcopy -Obinary example6.out example6.bin
```

Generation of disassembly code is the same as in the previous example:

```
$> arm-none-eabi-objdump -S example6.out   > example6.list
```

Accessing Special Registers

The CodeSourcery GNU ARM tool chain supports access to special registers. The names of the special registers must be in lowercase. For example:

```
      msr    control, r1
      mrs    r1, control
      msr    apsr, R1
      mrs    r0, psr
```

Using Unsupported Instructions

If you are using another GNU ARM tool chain, there might be cases in which the GNU assembler you are using does not support the assembly instruction that you wanted. In this situation, you can still insert the instruction in form of binary data using *.word*. For example:

```
.equu  DW_MSR_CONTROL_R0, 0x8814F380

        ...
        MOV   R0, #0x1
.word   DW_MSR_CONTROL_R0  /* This set the processor in user mode */
        ...
```

Inline Assembler in the GNU C Compiler

As in the ARM C Compiler, the GNU C Compiler supports an inline assembler. The syntax is a little bit different:

```
        __asm ("    inst1  op1, op2... \n"
               "    inst2  op1, op2... \n"
               ...
               "    inst   op1, op2... \n"
               : output_operands           /* optional */
               : input_operands            /* optional */
```

```
                : clobbered_register_list   /* optional */
                );
```

For example, a simple code to enter sleep mode looks like this:

```
void Sleep(void)
{  // Enter sleep mode using Wait-For-Interrupt
  __asm (
    "WFI\n"
    );
}
```

If the assembler code needs to have an input variable and an output variable—for example, divide a variable by 5 in the following code—it can be written as:

```
unsigned int DataIn, DataOut;  /* variables for input and output */
...
__asm  ("mov   r0, %0\n"
        "mov   r3, #5\n"
        "udiv  r0, r0, r3\n"
        "mov   %1, r0\n"
        : "=r" (DataOut) : "r" (DataIn) : "cc", "r3" );
```

With this code, the input parameter is a C variable called *DataIn* (*%0* first parameter), and the code returns the result to another C variable called *DataOut* (*%1* second parameter). The inline assembler code manually modifies register *r3* and changes the condition flags *cc* so that they are listed in the clobbered register list.

For more examples of inline assembler, refer to the GNU tool chain documentation *GCC-Inline-Assembly-HOWTO* on the Internet.

CHAPTER 20

Getting Started with the KEIL RealView Microcontroller Development Kit

In This Chapter:

- Overview
- Getting Started with μVision
- Outputting the “Hello World” Message Via UART
- Testing the Software
- Using the Debugger
- The Instruction Set Simulator
- Modifying the Vector Table
- Stopwatch Example with Interrupts

Overview

Various commercial development platforms are available for the Cortex-M3. One of the popular choices is the KEIL RealView Microcontroller Development Kit (RealView MDK). The RealView MDK contains various components:

- μVision
- Integrated Development Environment (IDE)
- Debugger
- Simulator
- RealView Compilation Tools from ARM
  1. C/C++ Compiler
  2. Assembler
  3. Linker
- RTX Real-Time Kernel
- Detailed startup code for microcontrollers

- Flash programming algorithms
- Program examples

For learning about the Cortex-M3 with RealView MDK, it is not necessary to have Cortex-M3 hardware. The µVision environment contains an instruction set simulator that allows testing of simple programs that do not require a development board.

RealView MDK can also be used with other tool chains, such as:

- GNU ARM Compiler
- ARM Development Suite (ADS)

A free evaluation CD-ROM for the KEIL tool can be requested from the KEIL Web site (www.keil.com). This version is also included in the Luminary Micro Stellaris Evaluation Kit[1] (www.luminarymicro.com).

Getting Started with µVision

A number of examples are provided with the RealView MDK, including some examples for the Luminary Micro Stellaris microcontroller products. These examples provide a powerful set of device driver libraries that are ready to use. It's easy to modify the provided examples to start developing your application, or you can develop your project from scratch. The following examples illustrate how this is done. The examples shown in this chapter are based on the v3.03 beta and on Luminary Micro LM32811 devices.

After installing the RealView MDK, you can start the µVision from the program menu. After installation, the µVision might start with a default project for a traditional ARM processor. We can close the current project and start a new one by selecting **New Project** in the pull-down menu (see Figure 20.1).

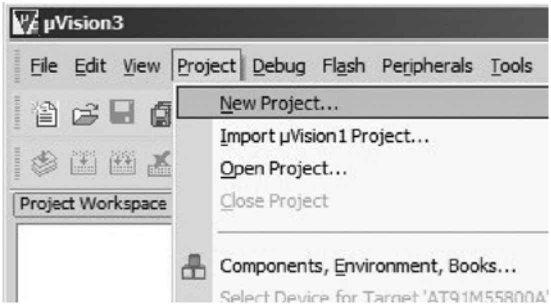
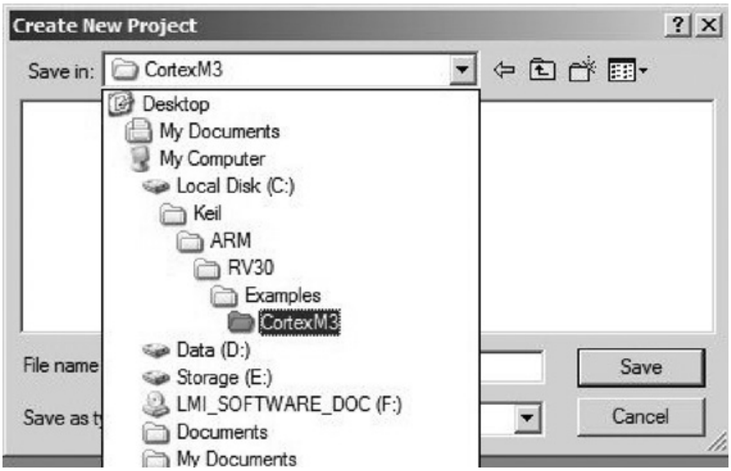


Figure 20.1 Selecting a New Project from the Program Menu

[1] Stellaris is a registered trademark of Luminary Micro.

Here a new project directory called CortexM3 is created (see Figure 20.2).

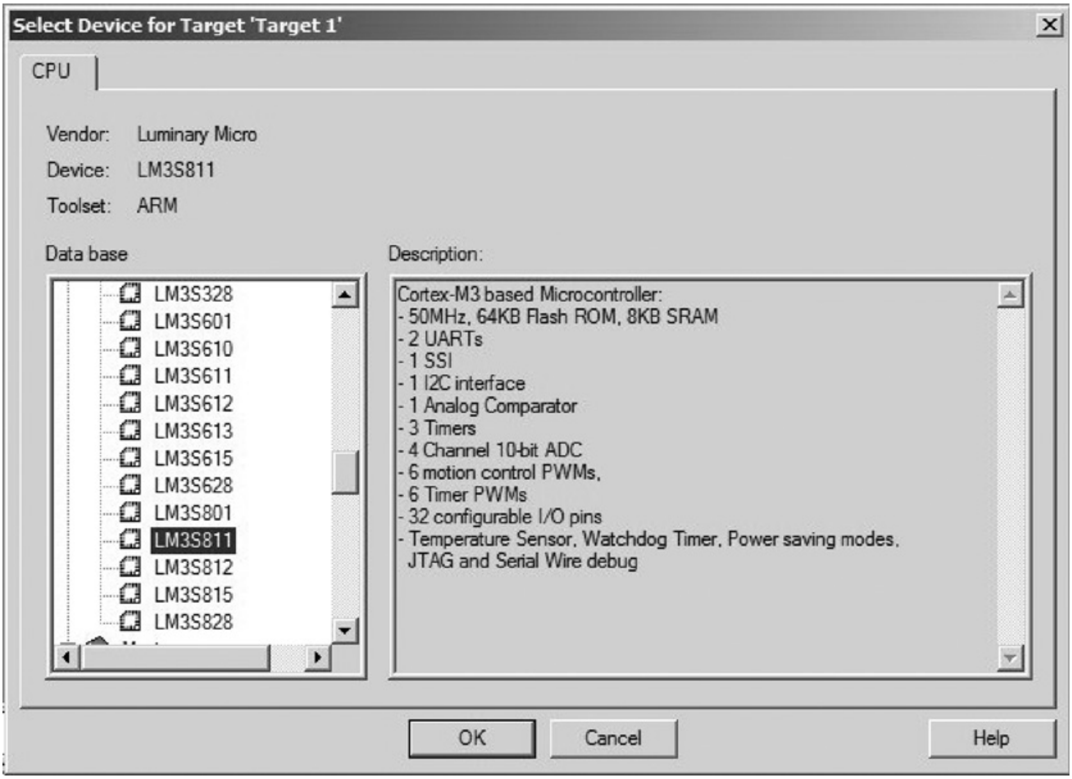


Create New Project

Save in: CortexM3

Figure 20.2 Choosing the CortexM3 Project Directory

Now we need to select the targeted device for this project. In this example, the LM3S811 is selected (see Figure 20.3).



Select Device for Target 'Target 1'

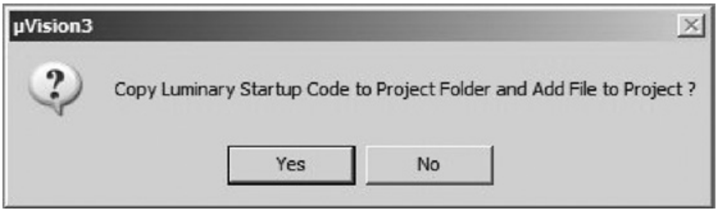
Vendor: Luminary Micro
Device: LM3S811
Toolset: ARM

Description:

Cortex-M3 based Microcontroller:
- 50MHz, 64KB Flash ROM, 8KB SRAM
- 2 UARTs
- 1 SSI
- 1 I2C interface
- 1 Analog Comparator
- 3 Timers
- 4 Channel 10-bit ADC
- 6 motion control PWMs,
- 6 Timer PWMs
- 32 configurable I/O pins
- Temperature Sensor, Watchdog Timer, Power saving modes, JTAG and Serial Wire debug

Figure 20.3 Selecting the LM3S811 Device

The software will then ask if you would like to use the default startup code. In this case, we select **Yes** (see Figure 20.4).



Copy Luminary Startup Code to Project Folder and Add File to Project ?

Figure 20.4 Choosing to Use the Default Startup Code

Now we have a project called Hello with only one file, called Startup.s (see Figure 20.5).

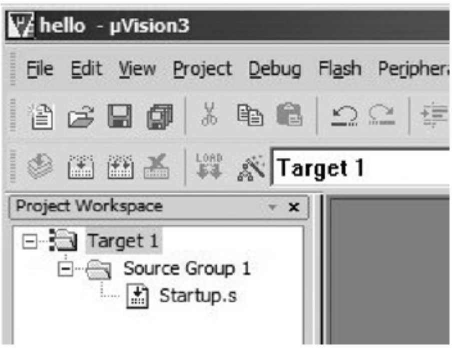


Figure 20.5 Project Created with the Default Startup Code

We can create a new C program file containing the main program (see Figure 20.6).

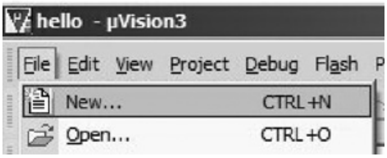
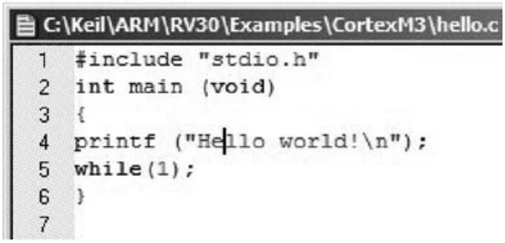


Figure 20.6 Creating a New C Program File

A text file is created and saved as hello.c (see Figure 20.7).



```
#include "stdio.h"
int main (void)
{
printf ("Hello world!\n");
while(1);
}

```

Figure 20.7 A Hello World C Example

Now we need to add this file to our project by right-clicking **Source Group 1** (see Figure 20.8).

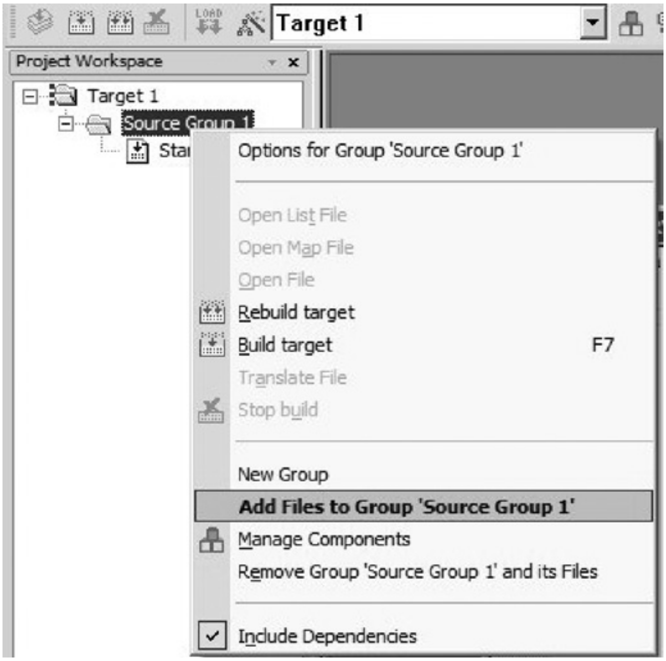


Figure 20.8 Adding the Hello world C Example to the Project

Renaming the Target and File Groups

The target name Target 1 and file group name Source Group 1 can be renamed to give a clearer meaning. This is done by clicking **Target 1** and **Source Group 1** in the project workspace and editing the names from there.

Select the **hello.c** that we created and then close the Add File window. Now the project contains two files (see Figure 20.9).

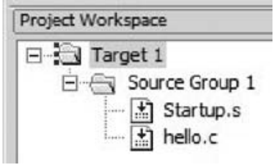
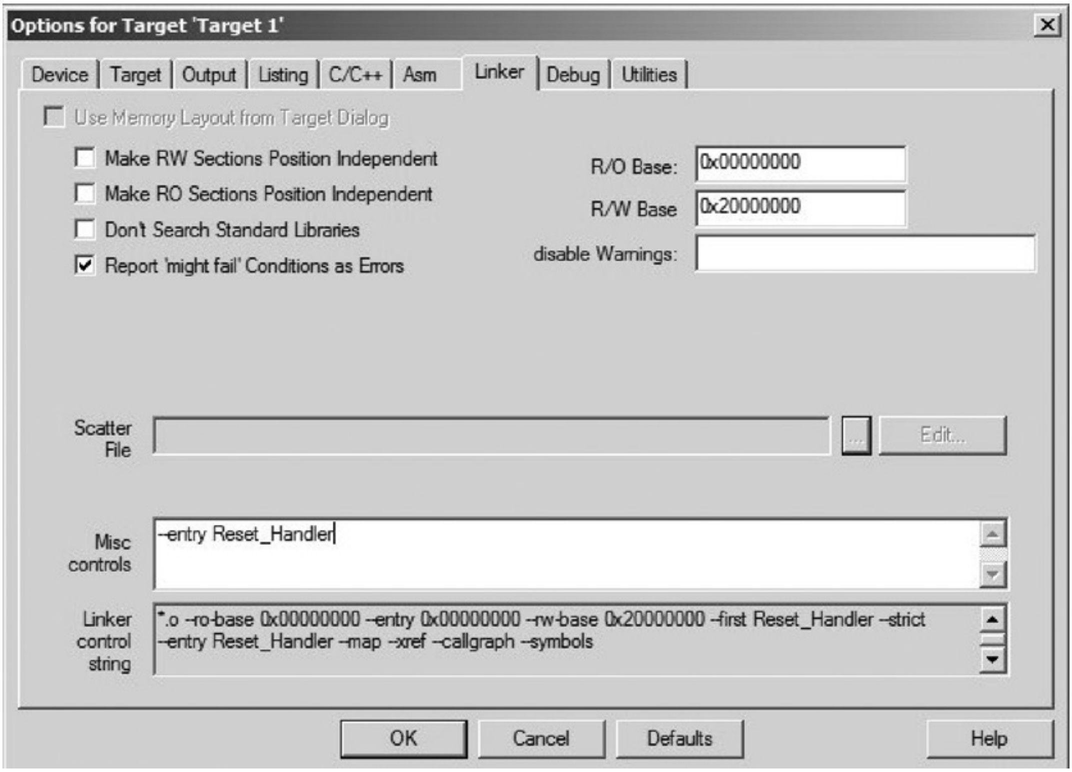


Figure 20.9 Project Window After the Hello World C Example is Added

We also need to set a linker setting to define the entry point of the program. We do this by adding—**entry Reset_Handler** in the Misc Controls box (see Figure 20.10). This option defines the starting point of the program. *Reset_Handler* is an instruction address that can be found in the Startup.s.



Options for Target 'Target 1'

□ Use Memory Layout from Target Dialog

□ Make RW Sections Position Independent

□ Make RO Sections Position Independent

□ Don't Search Standard Libraries

☑ Report 'might fail' Conditions as Errors

R/O Base: 0x00000000

R/W Base 0x20000000

disable Warnings:

Scatter File

Misc controls: --entry Reset_Handler

Linker control string:

```
*.o --ro-base 0x00000000 --entry 0x00000000 --rw-base 0x20000000 --first Reset_Handler --strict
--entry Reset_Handler --map --xref --callgraph --symbols
```

Figure 20.10 Defining the Entry Point in the Project

We can now compile the program. Right-click **Target 1** and select **Build target** (see Figure 20.11).

You should see the compilation success message in the output window (see Figure 20.12).

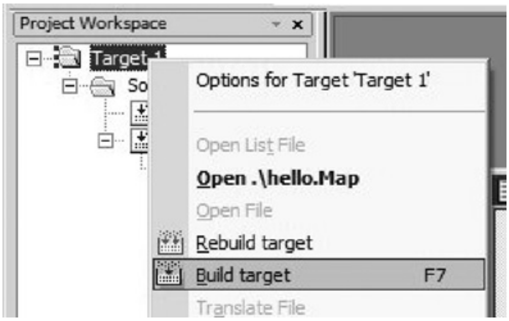
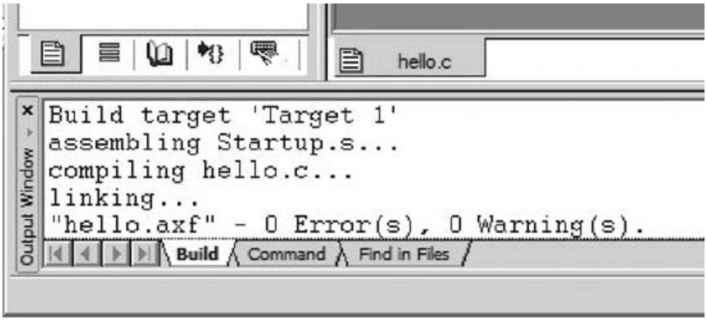


Figure 20.11 Starting the Compilation



```
Build target 'Target 1'
assembling Startup.s...
compiling hello.c...
linking...
"hello.axf" - 0 Error(s), 0 Warning(s).
```

Figure 20.12 Compilation Result in the Output Window

Outputting the "Hello World" Message Via UART

In the program code we created, we used the *printf* function in the standard C library. Since the C library does not know about the actual hardware we are using, if we want to output the text message using real hardware such as the UART on a chip, we need additional code.

As mentioned earlier in the book, the implementation of output to actual hardware is often referred to as *retargeting*. Besides creating text output, the retargeting code might also include functions for error handling and program termination. In this example, only the text output retargeting is covered.

For the following code, the "Hello world" message is output to UART 0 of the LM3S811 device. The target system used is the Luminary Micro LM3S811 evaluation board. The board has a 6 MHz crystal as a clock source, and an internal Phase Locked Loop (PLL) module that can step up the clock frequency to 50 MHz after a simple setup process. The baud rate setting is 115200 and is output to HyperTerminal running on a Windows PC.

To retarget the *printf* message, we need to implement the *fputc* function. In the following code we have created a *fputc* function that calls the *sendchar* function, which carries out the UART control:

```
========== hello.c ===========
#include "stdio.h"
#define CR     0x0D      // Carriage return
#define LF     0x0A      // Linefeed

void Uart0Init(void);
void SetClockFreq(void);
int  sendchar(int ch);

// Comment out the following line to use 6MHz clock
#define CLOCK50MHZ
```

```
// Register addresses
#define SYSCTRL_RCC   ((volatile unsigned long *)(0x400FE060))
#define SYSCTRL_RIS   ((volatile unsigned long *)(0x400FE050))
#define SYSCTRL_RCGC1 ((volatile unsigned long *)(0x400FE104))
#define SYSCTRL_RCGC2 ((volatile unsigned long *)(0x400FE108))
#define GPIOPA_AFSEL  ((volatile unsigned long *)(0x40004420))

#define UART0_DATA ((volatile unsigned long *)(0x4000C000))
#define UART0_FLAG ((volatile unsigned long *)(0x4000C018))
#define UART0_IBRD ((volatile unsigned long *)(0x4000C024))
#define UART0_FBRD ((volatile unsigned long *)(0x4000C028))
#define UART0_LCRH ((volatile unsigned long *)(0x4000C02C))
#define UART0_CTRL ((volatile unsigned long *)(0x4000C030))
#define UART0_RIS  ((volatile unsigned long *)(0x4000C03C))

int main (void)
{
SetClockFreq(); // Setup clock setting (50MHz/6MHz)
Uart0Init();    // Initialize Uart0

printf (“Hello world!\n”);
while (1);
}

void SetClockFreq(void)
{
#ifdef         CLOCK50MHZ
// Set BYPASS, clear USRSYSDIV and SYSDIV
*SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF83FFFFF) | 0x800 ;
// Clr OSCSRC, PWRDN and OEN
*SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFCFCF);
// Change SYSDIV, set USRSYSDIV     and Crystal value
*SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF87FFC3F) | 0x01C002C0;
// Wait until PLLLRIS is set
while ((*SYSCTRL_RIS & 0x40)==0); // wait until PLLLRIS is set
// Clear bypass
*SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFF7FF) ;
#else
// Set BYPASS, clear USRSYSDIV and SYSDIV
*SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF83FFFFF) | 0x800 ;
#endif
return;
}

void Uart0Init(void)
{
```

```
*SYSCTRL_RCGC1 = *SYSCTRL_RCGC1 | 0x0003; // Enable UART0 & UART1
                                          // clock
*SYSCTRL_RCGC2 = *SYSCTRL_RCGC2 | 0x0001; // Enable PORTA clock

*UART0_CTRL = 0; // Disable UART
#ifdef        CLOCK50MHZ
*UART0_IBRD = 27; // Program baud rate for 50MHz clock
*UART0_FBRD = 9;
#else
*UART0_IBRD = 3; // Program baud rate for 6MHz clock
*UART0_FBRD = 17;
#endif
*UART0_LCRH = 0x60; // 8 bit, no parity
*UART0_CTRL = 0x301; // Enable TX and RX, and UART enable
*GPIOPA_AFSEL = *GPIOPA_AFSEL | 0x3; // Use GPIO pins as UART0

return;
}

/* Output a character to UART0 (used by printf function to output
data) */
int sendchar (int ch)  {
  if (ch == '\n')  {
   while ((*UART0_FLAG & 0x8));   // Wait if it is busy
   *UART0_DATA = CR;              // output extra CR to get correct
  }                               // display on HyperTerminal
  while ((*UART0_FLAG & 0x8));    // Wait if it is busy
  return (*UART0_DATA = ch);      // output data
}
/* Retargetting code for text output */
int fputc(int ch, FILE *f) {
  return (sendchar(ch));
}
========== end of file ===========
```

The *SetupClockFreq* routine sets the system clock to 50 MHz. The setup sequence is device dependent. The subroutine can also be used to set the clock frequency to 6 MHz if the CLOCK50 MHZ compile directive is not set.

The UART initialization is carried out inside the *Uart0Init* subroutine. The setup process includes setting up the baud rate generator to provide a baud rate of 115200; configuring the UART to 8-bit, no parity, and 1 stop bit; and switching the GPIO port to alternate function because the UART pins are shared with GPIO port A. Before accessing the UART and the GPIO, the clocks for these blocks must be turned on. This is done by writing to SYSCTRL_RCGC1 and SYSCTRL_RCGC2.

The retargeting code is carried out by *fputc*, a predefined function name for character outputs. This function calls the *sendchar* function to output the character to the UART. The *sendchar* function outputs an extra carriage return character as a new line is detected. This is needed to get the text output correct on HyperTerminal; otherwise the new text in the next line will overwrite the previous line of text.

After the hello.c program is modified to include the retargeting code, the program is compiled again.

Testing the Software

If you've got the Luminary Micro LM3S811 evaluation board, you can try out the example by downloading the compile program into Flash and getting the "Hello world" message display output from the HyperTerminal. Assuming that you have set up the software drivers that come with the evaluation board, you can download and test the program by following these steps.

First, set up the Flash download option. This can be accessed from the pull-down menu, as shown in Figure 20.13.

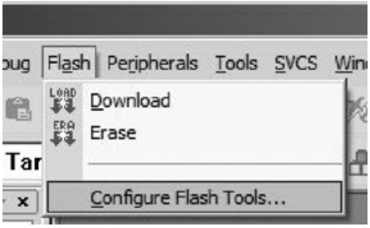
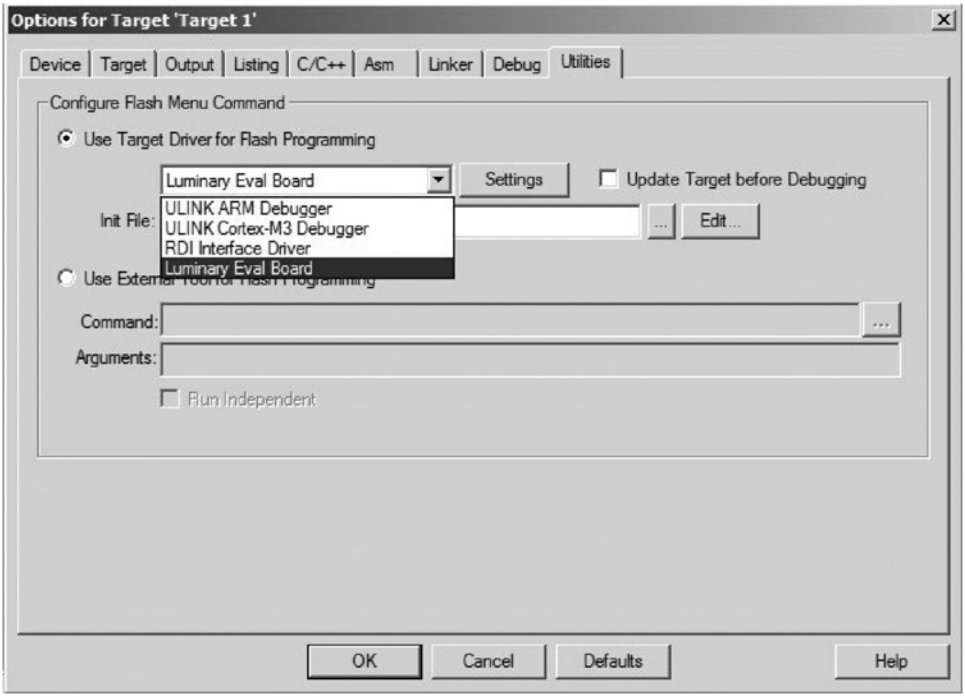


Figure 20.13 Setting Up Flash Programming Configuration

Inside this menu, we select the **Luminary Evaluation Board** as the download target (see Figure 20.14).

Then we can download the program to the Flash on a chip by selecting **Download** in the pull-down menu (see Figure 20.15).

The message shown in Figure 20.16 will appear, indicating that the download is complete. *Note:* If you have the board already running with HyperTerminal, you might need to close HyperTerminal, disconnect the USB cable from the PC, and reconnect before programming the Flash.



● Use Target Driver for Flash Programming

Luminary Eval Board

□ Update Target before Debugging

Init File:

Figure 20.14 Selecting Flash Programming Driver

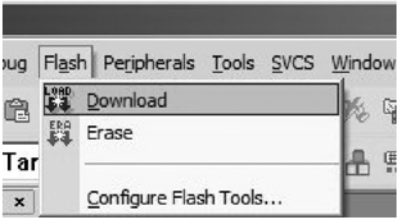
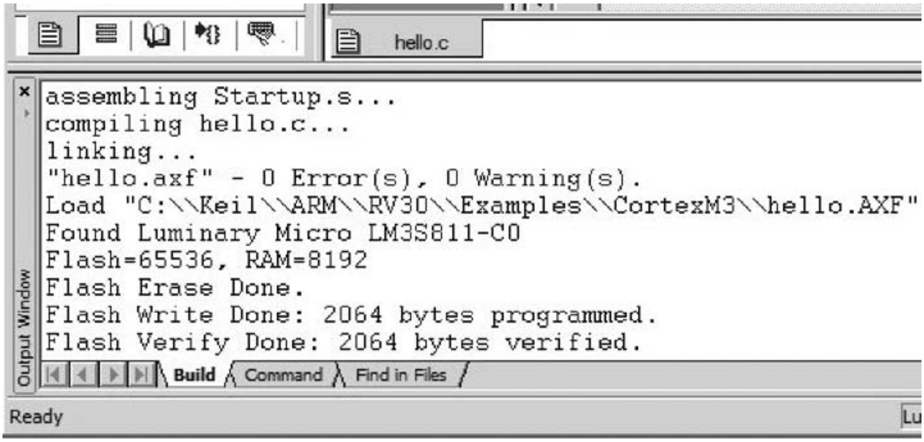


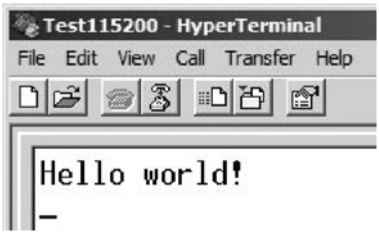
Figure 20.15 Starting the Download Process



```
assembling Startup.s...
compiling hello.c...
linking...
"hello.axf" - 0 Error(s), 0 Warning(s).
Load "C:\\Keil\\ARM\\RV30\\Examples\\CortexM3\\hello.AXF"
Found Luminary Micro LM3S811-C0
Flash=65536, RAM=8192
Flash Erase Done.
Flash Write Done: 2064 bytes programmed.
Flash Verify Done: 2064 bytes verified.
```

Figure 20.16 Report of the Download Process in the Output Window

After the programming is completed, you can start HyperTerminal and connect to the board using the Virtual COM Port driver (via USB connection) and get the text display from the program running on the microcontroller (see Figure 20.17).

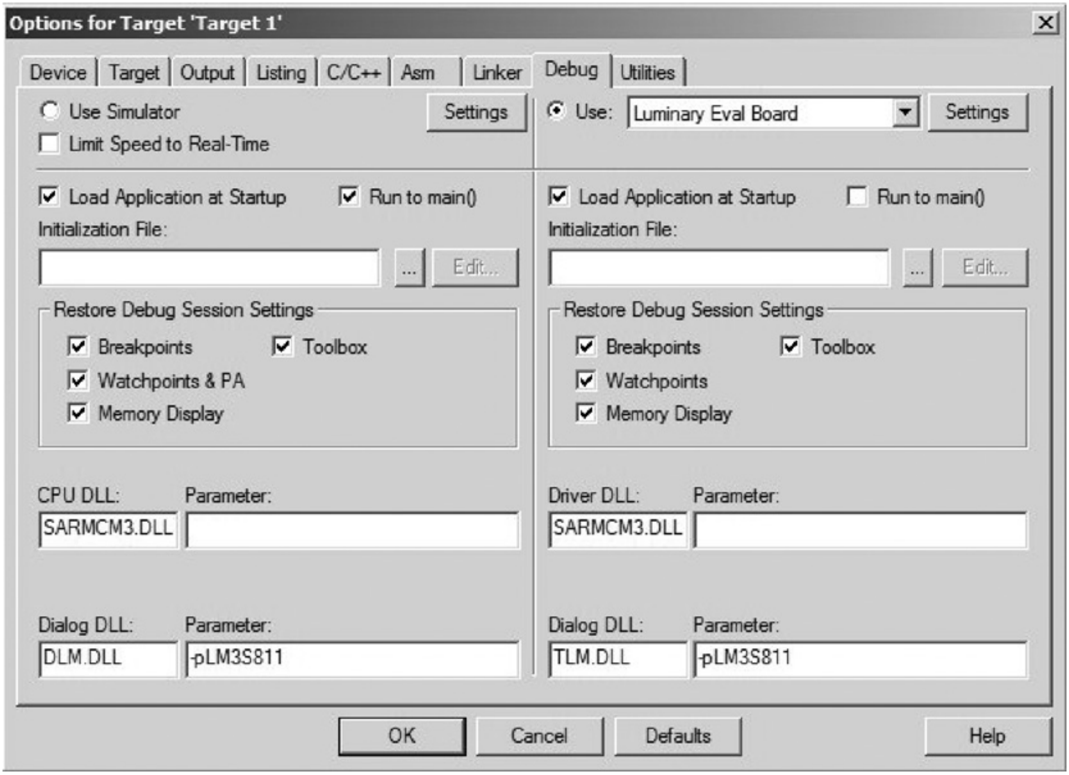


```
Hello world!
_
```

Figure 20.17 Output of Hello World Example from HyperTerminal console

Using the Debugger

You can use the debugger in µVision to connect to the Luminary Evaluation Board to debug your application. By right-clicking the project **Target 1** and selecting **Options**, we can access the debug option. In this case we select to use the **Luminary Eval Board** for debugging (see Figure 20.18).



○ Use Simulator
□ Limit Speed to Real-Time

☑ Load Application at Startup ☑ Run to main()
Initialization File:

Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox
☑ Watchpoints & PA
☑ Memory Display

CPU DLL: SARMCM3.DLL Parameter:

Dialog DLL: DLM.DLL Parameter: -pLM3S811

● Use: Luminary Eval Board

☑ Load Application at Startup □ Run to main()
Initialization File:

Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox
☑ Watchpoints
☑ Memory Display

Driver DLL: SARMCM3.DLL Parameter:

Dialog DLL: TLM.DLL Parameter: -pLM3S811

Figure 20.18 Configuring to Use LuminaryMicro Evaluation Board with µVision Debugger

We can then start the debug session from the pull-down menu (see Figure 20.19). *Note:* If you have the board already running with HyperTerminal, you might need to close HyperTerminal, disconnect the USB cable from the PC, and reconnect before starting the debug session.

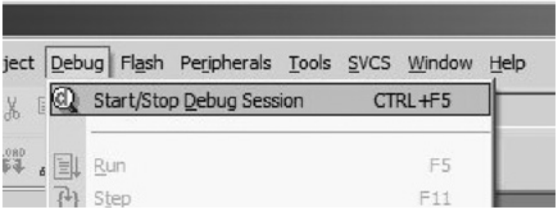
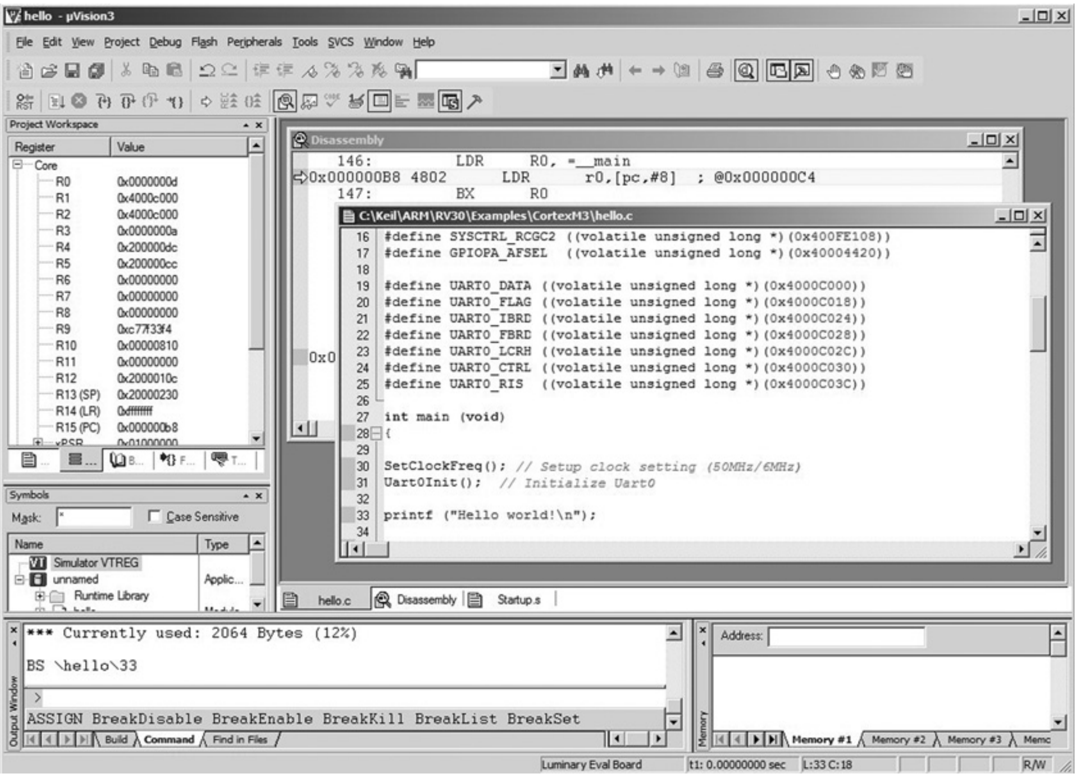


Figure 20.19 Starting a Debugger Session in µVision

When the debugger starts, the IDE provides a register view to display register contents. You can also get the disassemble code window and see the current instruction address. In Figure 20.20 we can see that the core is halted at the *Reset_Handler*.



Register	Value
Core	
R0	0x0000000d
R1	0x4000c000
R2	0x4000c000
R3	0x0000000a
R4	0x200000dc
R5	0x200000cc
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0xc77f33f4
R10	0x00000810
R11	0x00000000
R12	0x2000010c
R13 (SP)	0x20000230
R14 (LR)	0xffffffff
R15 (PC)	0x000000b8
xPSR	0x01000000

Name	Type
Simulator VTREG	
unnamed	Applic...
Runtime Library	

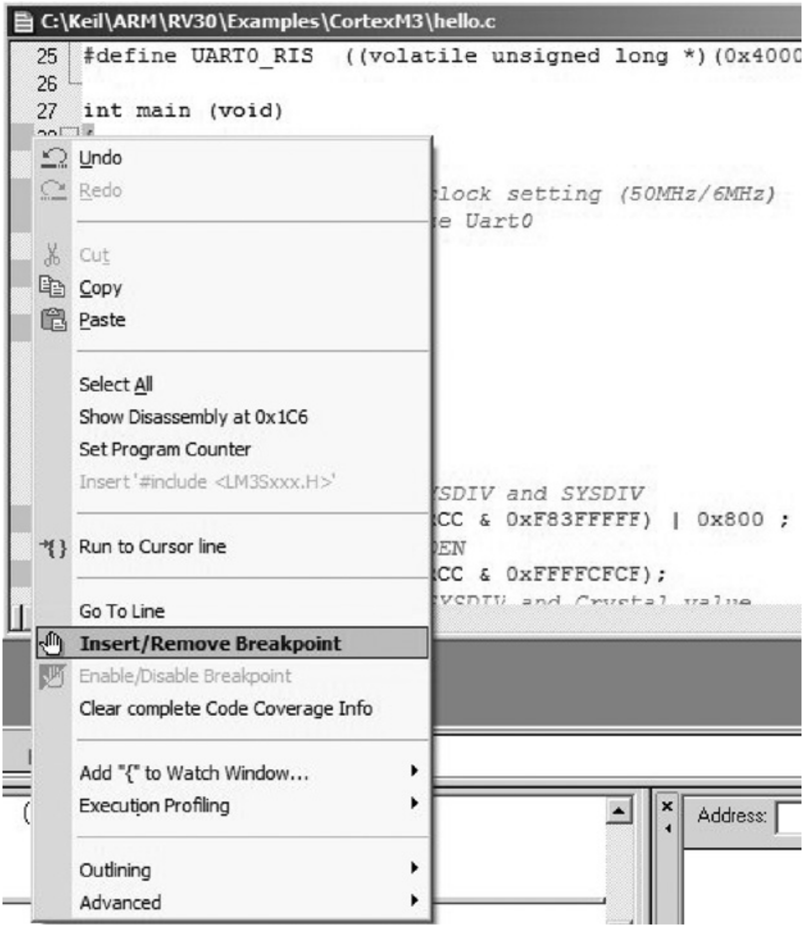
```
   146:          LDR     R0, =__main
0x000000B8 4802      LDR       r0,[pc,#8]  ; @0x000000C4
   147:          BX      R0
```

```
#define SYSCTRL_RCGC2 ((volatile unsigned long *)(0x400FE108))
#define GPIOPA_AFSEL  ((volatile unsigned long *)(0x40004420))

#define UART0_DATA ((volatile unsigned long *)(0x4000C000))
#define UART0_FLAG ((volatile unsigned long *)(0x4000C018))
#define UART0_IBRD ((volatile unsigned long *)(0x4000C024))
#define UART0_FBRD ((volatile unsigned long *)(0x4000C028))
#define UART0_LCRH ((volatile unsigned long *)(0x4000C02C))
#define UART0_CTRL ((volatile unsigned long *)(0x4000C030))
#define UART0_RIS  ((volatile unsigned long *)(0x4000C03C))

int main (void)
{

SetClockFreq(); // Setup clock setting (50MHz/6MHz)
Uart0Init();  // Initialize Uart0

printf ("Hello world!\n");
```

```
*** Currently used: 2064 Bytes (12%)

BS \hello\33

>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet
```

Figure 20.20 µVision Debug Environment

For testing, a breakpoint is set to stop the program execution at the beginning of main. This can be done by right-clicking the program code window and selecting **Insert/Remove Breakpoint** (see Figure 20.21). *Note*: We could also use the *Run to main( )* feature in the debug option to get the program execution stop at the beginning of main.



C:\Keil\ARM\RV30\Examples\CortexM3\hello.c

```
#define UART0_RIS  ((volatile unsigned long *)(0x4000C

int main (void)
```

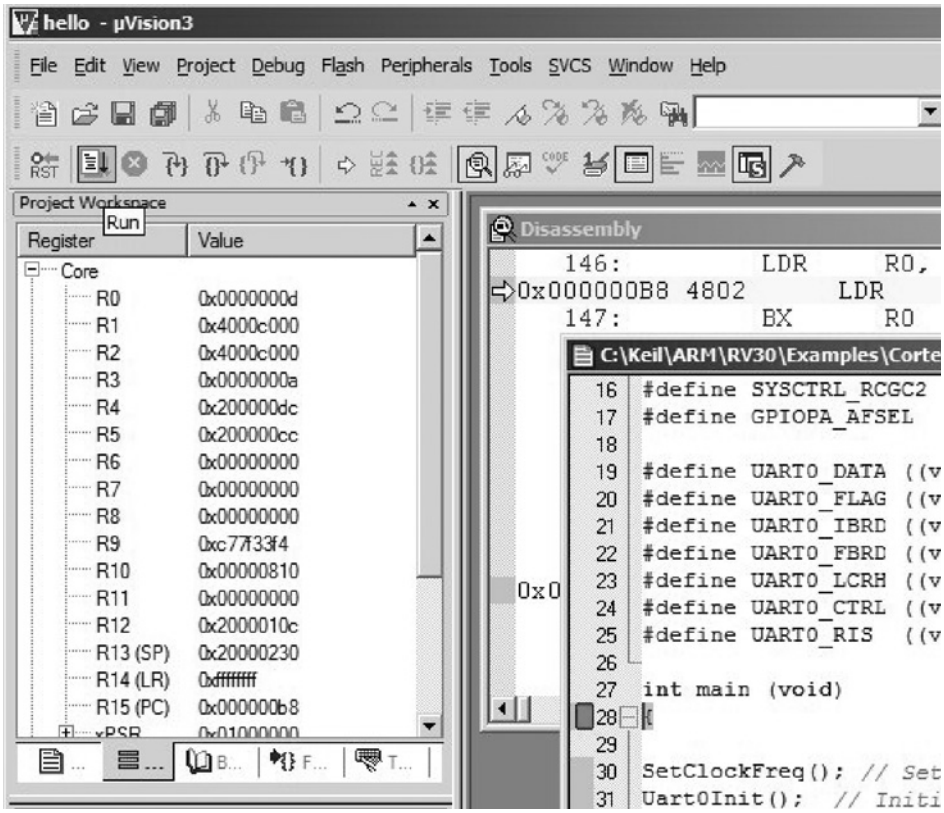
```
lock setting (50MHz/6MHz)
e Uart0
SDIV and SYSDIV
CC & 0xF83FFFFF) | 0x800 ;
EN
CC & 0xFFFFCFCF);
```

Figure 20.21 Insert or Remove Breakpoint

The program execution can then be started using the **Run** button on the tool bar (see Figure 20.22).

The program execution is then started, and it stops when it gets to the start of the main program (see Figure 20.23).

We can then use the stepping control of the tool bar to test our application and examine the results using the register window.


hello - µVision3
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
RST
Project Workspace
Run
Disassembly
C:\Keil\ARM\RV30\Examples\Corte
B...
F...
T...

Register	Value
Core	
R0	0x0000000d
R1	0x4000c000
R2	0x4000c000
R3	0x0000000a
R4	0x200000dc
R5	0x200000cc
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0xc77f33f4
R10	0x00000810
R11	0x00000000
R12	0x2000010c
R13 (SP)	0x20000230
R14 (LR)	0xffffffff
R15 (PC)	0x000000b8
xPSR	0x01000000

```
   146:          LDR      R0,
0x000000B8 4802       LDR
   147:          BX       R0
```

```
#define SYSCTRL_RCGC2
#define GPIOPA_AFSEL

#define UART0_DATA  ((v
#define UART0_FLAG  ((v
#define UART0_IBRD  ((v
#define UART0_FBRD  ((v
#define UART0_LCRH  ((v
#define UART0_CTRL  ((v
#define UART0_RIS   ((v

int main (void)
{

SetClockFreq(); // Set
Uart0Init();   // Initi
```

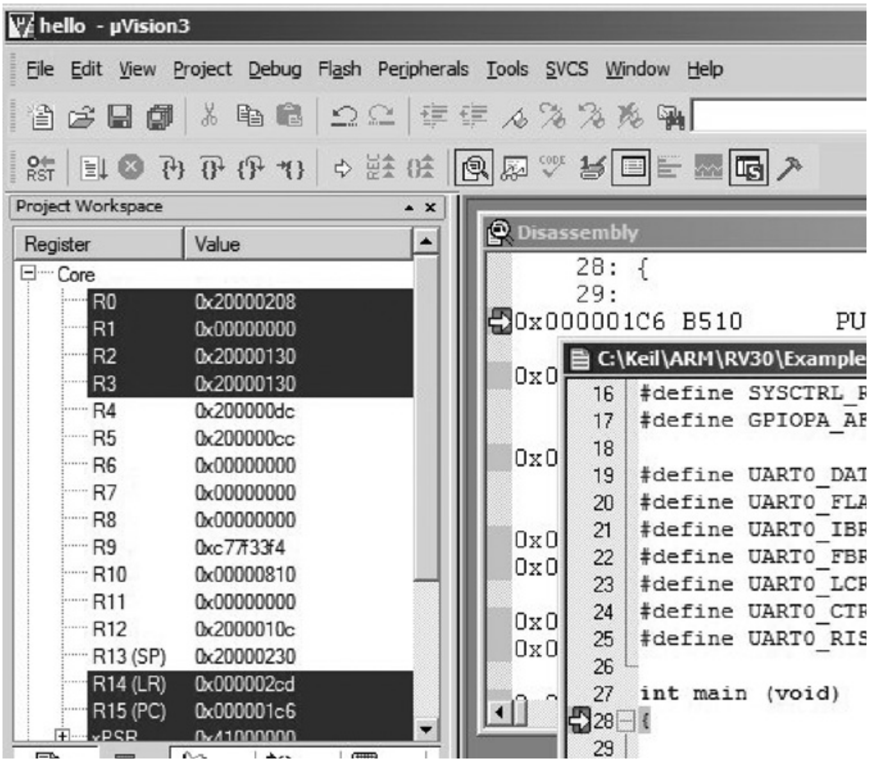
Figure 20.22 Starting Program Execution Using the RUN Button

The Instruction Set Simulator

The µVision IDE also comes with an instruction set simulator that can be used for debugging applications. The operation is similar to using the debugger with hardware and is a useful tool for learning the Cortex-M3. To use the instruction set simulator, change the debug option of the project to **Use Simulator** (see Figure 20.24). Note that the simulator cannot simulate all hardware peripheral behaviors, so the UART interface code might not simulate correctly.

When using the simulator for debugging, you might also need to adjust the memory setting of the simulation. This is done by accessing the **Memory Map** option after starting the debugging session (see Figure 20.25).

For example, you might need to add the UART memory address range to the memory map (see Figure 20.26). Otherwise you will get an abort exception in the simulation when you try to access the UART.

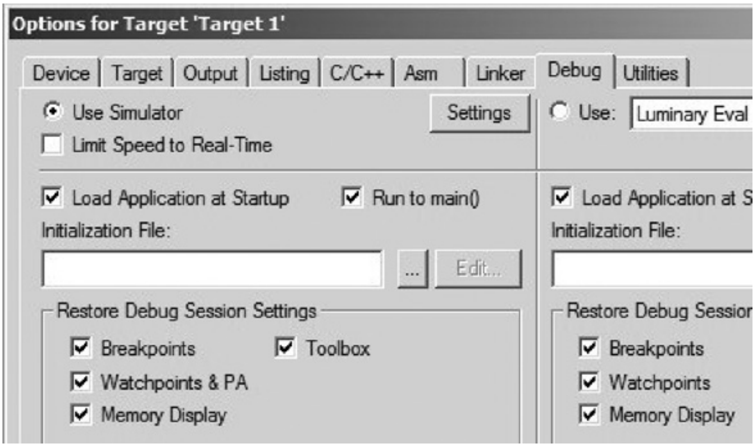


Register	Value
Core	
R0	0x20000208
R1	0x00000000
R2	0x20000130
R3	0x20000130
R4	0x200000dc
R5	0x200000cc
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0xc77f33f4
R10	0x00000810
R11	0x00000000
R12	0x2000010c
R13 (SP)	0x20000230
R14 (LR)	0x000002cd
R15 (PC)	0x000001c6
xPSR	0x41000000

```
    28: {
    29:
0x000001C6 B510       PU
```

```
#define SYSCTRL_F
#define GPIOPA_AF

#define UART0_DAT
#define UART0_FLA
#define UART0_IBF
#define UART0_FBF
#define UART0_LCF
#define UART0_CTF
#define UART0_RIS

int main (void)
{
```

Figure 20.23 Program Execution Halted at Beginning of Main When a Break Point is Hit



● Use Simulator

□ Limit Speed to Real-Time

☑ Load Application at Startup ☑ Run to main()

Initialization File:

Restore Debug Session Settings

☑ Breakpoints ☑ Toolbox

☑ Watchpoints & PA

☑ Memory Display

○ Use: Luminary Eval

☑ Load Application at S

Initialization File:

Restore Debug Session

☑ Breakpoints

☑ Watchpoints

☑ Memory Display

Figure 20.24 Selecting Simulator as Debugging Target

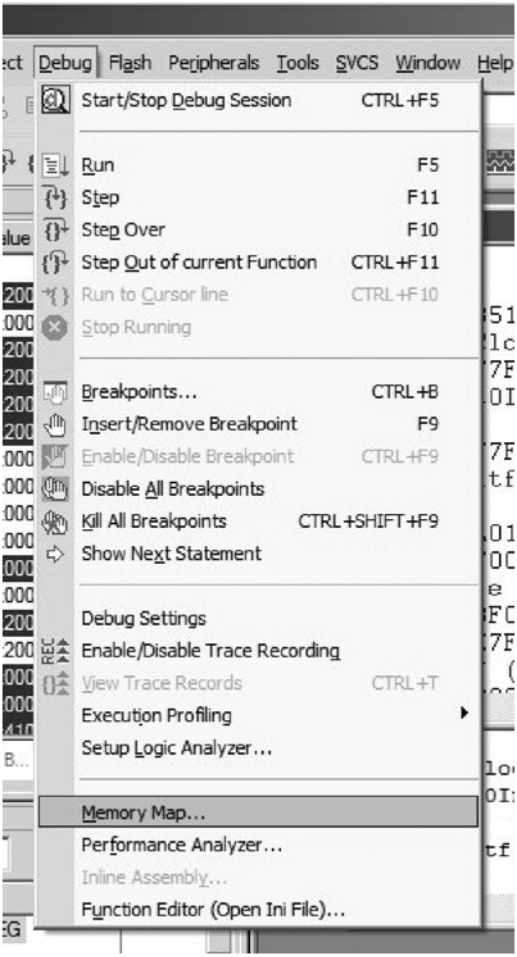
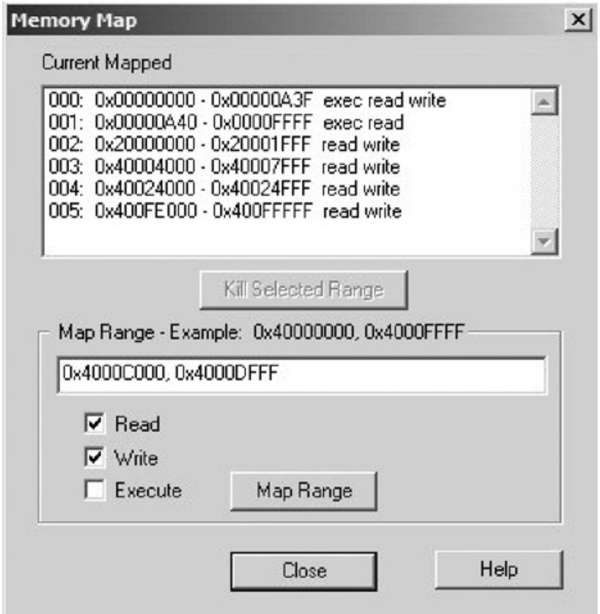

Debug
Flash
Peripherals
Tools
SVCS
Window
Help
Start/Stop Debug Session CTRL+F5
Run F5
Step F11
Step Over F10
Step Out of current Function CTRL+F11
Run to Cursor line CTRL+F10
Stop Running
Breakpoints... CTRL+B
Insert/Remove Breakpoint F9
Enable/Disable Breakpoint CTRL+F9
Disable All Breakpoints
Kill All Breakpoints CTRL+SHIFT+F9
Show Next Statement
Debug Settings
Enable/Disable Trace Recording
View Trace Records CTRL+T
Execution Profiling
Setup Logic Analyzer...
Memory Map...
Performance Analyzer...
Inline Assembly...
Function Editor (Open Ini File)...

Figure 20.25 Accessing the Memory Map Option

Modifying the Vector Table

In the previous example, the vector table is defined inside the file Startup.s, which is a standard startup code the tool prepares automatically. This file contains the vector table, a default reset handler, a default NMI handler, a default hard fault handler, and a default interrupt handler. These exception handlers might need to be customized or modified, depending on your application. For example, if a peripheral interrupt is required for your application, you need to change the vector table so that the Interrupt Service Routine (ISR) you created will be executed when the interrupt is triggered.

The default exception handlers are in the form of assembly code inside Startup.s. However, the exception handlers can be implemented in C or in a different assembly program file. In these



Memory Map

Current Mapped

```
000:  0x00000000 - 0x00000A3F  exec read write
001:  0x00000A40 - 0x0000FFFF  exec read
002:  0x20000000 - 0x20001FFF  read write
003:  0x40004000 - 0x40007FFF  read write
004:  0x40024000 - 0x40024FFF  read write
005:  0x400FE000 - 0x400FFFFF  read write
```

Map Range - Example: 0x40000000, 0x4000FFFF

0x4000C000, 0x4000DFFF

☑ Read
☑ Write
☐ Execute

Figure 20.26 Adding UART Memory to Simulator Memory Setup

cases, the IMPORT command in the assembler will be required to indicate that the interrupt handler address label is defined in another file. The next section contains an example that illustrates how this command is used as well as illustrating simple exception handling in C.

Stopwatch Example with Interrupts

This example includes the use of exceptions such as SYSTICK and the interrupt (UART0). The stopwatch to be developed has three states, as illustrated in Figure 20.27.

Based on the previous example, the stopwatch is controlled by the PC using the UART interface. To simplify the example code, we fix the operating speed at 50 MHz.

The timing measurement is carried out by the SYSTICK, which interrupts the processor at 100 Hz. The SYSTICK is running from the core clock frequency at 50 MHz. Every time the SYSTICK exception handler is executed, if the stopwatch is running, it increments the counter variable *TickCounter*.

Since display of text via UART is relatively slow, the control of the stopwatch is handled inside the exception handler, and the display of the text and stopwatch value is carried out in the main (Thread level). A simple software state machine is used to control the start, stop, and clear of the stopwatch. The state machine is controlled via the UART handler, which is triggered every time a character is received.

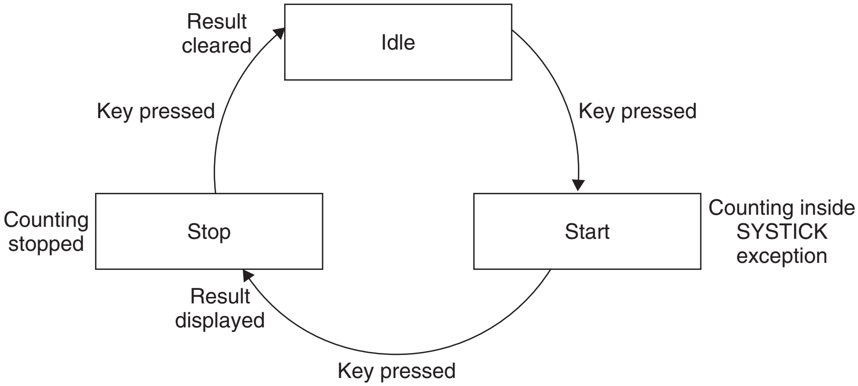


Figure 20.27 State Machine Design for Stopwatch

Using the same procedure we used for the “Hello World” example, let’s start a new project called *stopwatch*. Instead of having hello.c, a C program file called stopwatch.c is added:

```
========== stopwatch.c ===========
#include “stdio.h”
#define CR     0x0D        // Carriage return
#define LF     0x0A        // Linefeed

void Uart0Init(void);
void SysTickInit(void);
void SetClockFreq(void);
void DisplayTime(void);
void PrintValue(int value);
int  sendchar(int ch);
int  getkey(void);
void Uart0Handler(void);
void SysTickHandler(void);

// Register addresses
#define SYSCTRL_RCC   ((volatile unsigned long *)(0x400FE060))
#define SYSCTRL_RIS   ((volatile unsigned long *)(0x400FE050))
#define SYSCTRL_RCGC1 ((volatile unsigned long *)(0x400FE104))
#define SYSCTRL_RCGC2 ((volatile unsigned long *)(0x400FE108))
#define GPIOPA_AFSEL  ((volatile unsigned long *)(0x40004420))
#define UART0_DATA    ((volatile unsigned long *)(0x4000C000))
#define UART0_FLAG    ((volatile unsigned long *)(0x4000C018))
#define UART0_IBRD    ((volatile unsigned long *)(0x4000C024))
#define UART0_FBRD    ((volatile unsigned long *)(0x4000C028))
#define UART0_LCRH    ((volatile unsigned long *)(0x4000C02C))
#define UART0_CTRL    ((volatile unsigned long *)(0x4000C030))
#define UART0_IM      ((volatile unsigned long *)(0x4000C038))
```

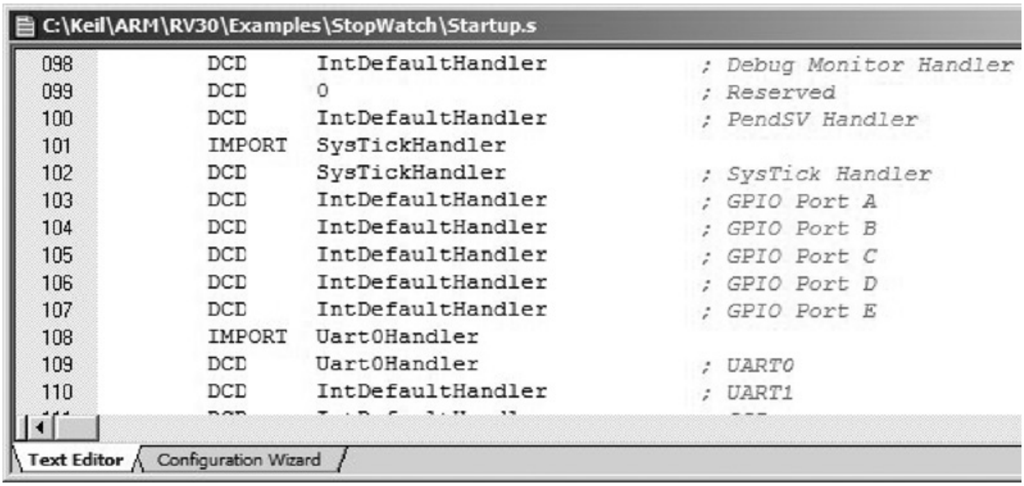
```
#define UART0_RIS    ((volatile unsigned long *)(0x4000C03C))
#define UART0_ICR    ((volatile unsigned long *)(0x4000C044))

#define NVIC_IRQ_EN0 ((volatile unsigned long *)(0xE000E100))

// Global variables
volatile int            CurrState;    // State machine
volatile unsigned long  TickCounter;  // Stop watch value
volatile int            KeyReceived;  // Indicate user pressed a key
volatile int            userinput ;   // Key pressed by user

#define IDLE_STATE 0                  // Definition of state machine
#define RUN_STATE  1
#define STOP_STATE 2

int main (void)
{
int    CurrStateLocal; // local variable

// Initialize global variable
CurrState = 0;
KeyReceived = 0;

// Initialization  of hardware
SetClockFreq(); // Setup clock setting (50MHz)
Uart0Init();    // Initialize Uart0
SysTickInit();  // Initialize Systick

printf (“Stop Watch\n”);

while (1) {
  CurrStateLocal = CurrState;    // Make a local copy because the
  // value could change by UART handler at any time.
  switch (CurrStateLocal) {
    case (IDLE_STATE):
      printf ("\nPress any key to start\n");
      break;
    case (RUN_STATE):
      printf ("\nPress any key to stop\n");
      break;
    case (STOP_STATE):
      printf ("\nPress any key to clear\n");
      break;
    default:
      CurrState = IDLE_STATE;
      break;
    } // end of switch
    while (KeyReceived == 0) {
      if (CurrState==RUN_STATE){
        DisplayTime();
        }
      }; // Wait for user input
      if (CurrStateLocal==STOP_STATE) {
         TickCounter=0;
         DisplayTime(); // Display to indicate result is cleared
         }
       else if (CurrStateLocal==RUN_STATE) {
         DisplayTime(); // Display result
         }
    if (KeyReceived!=0) KeyReceived=0;

  }; // end of while loop
} // end of main

void SetClockFreq(void)
{
// Set BYPASS, clear USRSYSDIV and SYSDIV
*SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF83FFFFF) | 0x800 ;
// Clr OSCSRC, PWRDN and OEN
*SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFCFCF);
// Change SYSDIV, set USRSYSDIV     and Crystal value
*SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF87FFC3F) | 0x01C002C0;
// Wait until PLLLRIS is set
while ((*SYSCTRL_RIS & 0x40)==0); // wait until PLLLRIS is set
// Clear bypass
*SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFF7FF) ;
return;
}
// UART0 initialization
void Uart0Init(void)
{

*SYSCTRL_RCGC1 = *SYSCTRL_RCGC1 | 0x0003; // Enable UART0 & UART1
                  clock
*SYSCTRL_RCGC2 = *SYSCTRL_RCGC2 | 0x0001; // Enable PORTA clock

*UART0_CTRL = 0;     // Disable UART
*UART0_IBRD = 27;    // Program baud rate for 50MHz clock
*UART0_FBRD = 9;
*UART0_LCRH = 0x60;  // 8 bit, no parity
*UART0_CTRL = 0x301; // Enable TX and RX, and UART enable
*UART0_IM   = 0x10;  // Enable UART interrupt for receive data
*GPIOPA_AFSEL = *GPIOPA_AFSEL | 0x3; // Use GPIO pins as UART0
*NVIC_IRQ_EN0 = (0x1<<5); // Enable UART interrupt at NVIC

return;
}
```

```
// SYSTICK initialization
void SysTickInit(void)
{

#define NVIC_STCSR   ((volatile unsigned long *)(0xE000E010))
#define NVIC_RELOAD  ((volatile unsigned long *)(0xE000E014))
#define NVIC_CURRVAL ((volatile unsigned long *)(0xE000E018))
#define NVIC_CALVAL  ((volatile unsigned long *)(0xE000E01C))

*NVIC_STCSR   = 0;   // Disable SYSTICK
*NVIC_RELOAD  = 499999; // Reload value for 100Hz with 50MHz clock
*NVIC_CURRVAL = 0;   // Clear current value
*NVIC_STCSR   = 0x7; // Enable SYSTICK with interrupt, core clock
return;
}
// SYSTICK exception handler
void SysTickHandler(void)
{
if (CurrState==RUN_STATE) {
  TickCounter++;
  }
return;
}
// UART0 RX interrupt handler
void Uart0Handler(void)
{
userinput = getkey();
// Indicate a key has been received
KeyReceived++;
// De-assert UART interrupt
*UART0_ICR  = 0x10;
// Switch state
switch (CurrState) {
  case (IDLE_STATE):
    CurrState = RUN_STATE;
    break;
  case (RUN_STATE):
    CurrState = STOP_STATE;
    break;
  case (STOP_STATE):
    CurrState = IDLE_STATE;
    break;
  default:
    CurrState = IDLE_STATE;
    break;
  } // end of switch
return;
}
// Display the time value
void DisplayTime(void)
{
unsigned long  TickCounterCopy;
unsigned long  TmpValue;

sendchar(CR);
TickCounterCopy = TickCounter; // Make a local copy because the
// value could change by SYSTICK handler at any time.
TmpValue        = TickCounterCopy / 6000; // Minutes
PrintValue(TmpValue);
TickCounterCopy = TickCounterCopy - (TmpValue * 6000);
TmpValue        = TickCounterCopy / 100;   // Seconds
sendchar(':');
PrintValue(TmpValue);
TmpValue        = TickCounterCopy - (TmpValue * 100);
sendchar(':');
PrintValue(TmpValue); // mini-seconds
sendchar(' ');
sendchar(' ');
return;
}
// Display decimal value
void PrintValue(int value)
{
printf ("%d", value);
return;
}

// Output a character to UART0 (used by printf function to output data)
int sendchar (int ch)  {
  if (ch == '\n')  {
    //while ((*UART0_FLAG & 0x8)); // Wait if it is busy
    while ((*UART0_FLAG & 0x20)); // Wait if TXFIFO is full
    *UART0_DATA = CR;  // output extra CR to get correct
  }                               // display on hyperterminal
  //while ((*UART0_FLAG & 0x8)); // Wait if it is busy
  while ((*UART0_FLAG & 0x20)); // Wait if TXFIFO is full
  return (*UART0_DATA = ch);   // output data
}
// Get user input
int getkey (void)  { // Read character from Serial Port
  while (*UART0_FLAG & 0x10); // Wait if RX FIFO empty
  return (*UART0_DATA);
}
// Retarget text output
```

```
int fputc(int ch, FILE *f) {
  return (sendchar(ch));
}
========== end of file ===========
```

The UART initialization has changed slightly to enable interrupts when a character is received via the UART interface. To enable the UART interrupt request, the interrupt has to be enabled at the UART interrupt mask register as well at the NVIC. For the SYSTICK, only the exception control at the SYSTICK Control and Status register needs to be programmed.

In addition, a number of extra functions are added, including the UART and SYSTICK handlers, display functions, and SYSTICK initialization. Depending on the design of the peripheral, an exception/interrupt handler might need to clear the exception/interrupt request. In this case, the UART handler clears the UART interrupt request using the Interrupt Clear Register (UART0_ICR). The startup code Startup.s is also modified to set up the exception handlers (see Figure 20.28).

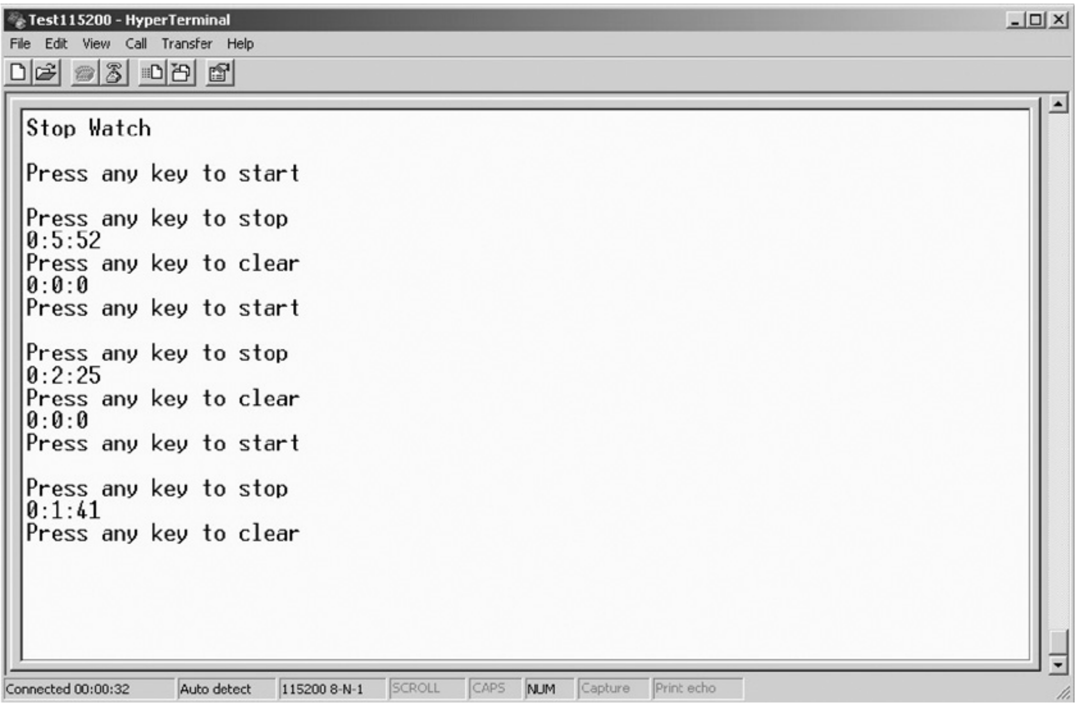


```
        DCD     IntDefaultHandler           ; Debug Monitor Handler
        DCD     0                           ; Reserved
        DCD     IntDefaultHandler           ; PendSV Handler
        IMPORT  SysTickHandler
        DCD     SysTickHandler              ; SysTick Handler
        DCD     IntDefaultHandler           ; GPIO Port A
        DCD     IntDefaultHandler           ; GPIO Port B
        DCD     IntDefaultHandler           ; GPIO Port C
        DCD     IntDefaultHandler           ; GPIO Port D
        DCD     IntDefaultHandler           ; GPIO Port E
        IMPORT  Uart0Handler
        DCD     Uart0Handler                ; UART0
        DCD     IntDefaultHandler           ; UART1
```

Figure 20.28 Adding SysTickHandler and Uart0Handler to the Vector Table by IMPORT and DCD Commands

Since the handlers are in the C program file, the IMPORT command is needed so that the assembler knows that the address label is from a different file.

After the program is compiled and downloaded to the evaluation board, it can then be tested by connecting to a PC running HyperTerminal. Figure 20.29 shows the result.



```
Stop Watch

Press any key to start

Press any key to stop
0:5:52
Press any key to clear
0:0:0
Press any key to start

Press any key to stop
0:2:25
Press any key to clear
0:0:0
Press any key to start

Press any key to stop
0:1:41
Press any key to clear
```

Figure 20.29 Output of the Stopwatch Example on HyperTerminal Console

Note: If the test board is a Luminary Micro evaluation board and the Virtual COM Port is used for the UART communication, this example might not work correctly (keystrokes in HyperTerminal cannot be sent to the board) due to a problem with the Virtual COM port device driver. In this case, you might need to test the program on a different PC with only the device driver and without the RealView MDK installation.

This page intentionally left blank

APPENDIX A

Cortex-M3 Instructions Summary

This material is reproduced from the *Cortex-M3 Technical Reference Manual* with permission from ARM Limited. Instructions marked with a plus sign (+) indicate that the flag (APSR) gets updated.

Supported 16-Bit Thumb Instructions

Table A.1 Supported 16-bit Instructions

Assembler	Operation
`ADC <Rd>, <Rm>`+	Add register value and C flag to register value: `Rd = Rd + Rm + C`
`ADD <Rd>, <Rn>, #<immed_3>`+	Add immediate 3-bit value to register: `Rd = Rn + immed_3`
`ADD <Rd>, #<immed_8>`+	Add immediate 8-bit value to register: `Rd = Rd + immed_8`
`ADD <Rd>, <Rn>, <Rm>`+	Add low register value to low register value: `Rd = Rn + Rm`
`ADD <Rd>, <Rm>`	Add high register value to low or high register value
`ADD <Rd>, PC, #<immed_8>*4`	Add 4× (immediate 8-bit value) + (word aligned PC value) to register: `Rd = PC + 4*immed_8`
`ADD <Rd>, SP, #<immed_8>*4`	Add 4× (immediate 8-bit value) + (word aligned SP value) to register: `Rd = SP + 4*immed_8`
`ADD SP, #<immed_7>*4`	Add 4× (immediate 7-bit value) to SP: `SP = SP + 4*immed_7`

(Continued)

Table A.1 (Continued)

Assembler	Operation
AND <Rd>, <Rm>+	Bitwise AND register: Rd = Rd AND Rm
ASR <Rd>, <Rm>, #<immed_5>$^{+}$	Arithmetic shift right by immediate number: Rd = Rm >> immed_5
ASR <Rd>, <Rs>$^{+}$	Arithmetic shift right by number in register: Rd = Rm >> Rs

B<cond> <target_address_8>	Branch conditional:
	if <cond> then
	PC = (PC+4)+(SignExtend(target_address_8)*2)

B <target_address_11>	Branch unconditional: PC = (PC+4)+(SignExtend(target_address_11)*2)
BIC <Rd>,<Rm>+	Bit clear: Rd = Rd AND (NOT Rm)
BKPT <immed_8>	Software breakpoint
BL <target_address_11>	Branch with link
BLX <Rm>	Branch with link and exchange (Rm[bit0] must be 1)
BX <Rm>	Branch with exchange (Rm[bit0] must be 1)
CBNZ <Rn>,<label>	Compare and branch if nonzero (forward branch only)
CBZ <Rn>,<label>	Compare and branch if zero (forward branch only)
CMN <Rn>, <Rm>+	Compare negation of register value with another register value: compute (Rn - (-Rm)) and update flags
CMP <Rn>, #<immed_8>+	Compare register with immediate 8-bit value
CMP <Rn>, <Rm>+	Compare registers
CMP <Rn>, <Rm>+	Compare high registers to low or high register
CPSIE <i or f> CPSID <i or f>	Change processor state CPSIE enable interrupt by clearing PRIMASK(i) or FAULTMASK(f) CPSID disable interrupt by setting PRIMASK(i) or FAULTMASK(f)
CPY <Rd>, <Rm>	Copy a high or low register value to another high or low register
EOR <Rd>, <Rm>+	Bitwise exclusive OR register values
IT<x> <cond> IT<x><y> <cond> IT<x><y><z> <cond>	IF-THEN conditional block; conditionally execute following two to four instructions based on condition *<cond>*
LDMIA <Rn>!, <registers>	Load Multiple Increment After; load multiple words from memory starting from address specified by Rn

Table A.1 (Continued)

Assembler	Operation
LDR <Rd>,[<Rn>,#<immed_5>*4]	Load memory word from base register address + 5-bit immediate offset
LDR <Rd>,[<Rn>,<Rm>]	Load memory word from base register address + register offset
LDR <Rd>,[PC,#<immed_8>*4]	Load memory word from PC address + 8-bit immediate offset
LDR <Rd>,[SP,#<immed_8>*4]	Load memory word from SP address + 8-bit immediate offset
LDRB <Rd>,[<Rn>,#<immed_5>]	Load memory byte[7:0] from base register address + 5-bit immediate offset
LDRB <Rd>,[<Rn>,<Rm>]	Load memory byte[7:0] from base register address + register offset
LDRH <Rd>,[<Rn>,#<immed_5>*2]	Load memory half word[15:0] from base register address + 5-bit immediate offset
LDRH <Rd>,[<Rn>,<Rm>]	Load memory half word[15:0] from base register address + register offset
LDRSB <Rd>,[<Rn>,<Rm>]	Load signed memory byte[7:0] from base register address + register offset
LDRSH <Rd>,[<Rn>,<Rm>]	Load signed memory half word[7:0] from base register address + register offset
LSL <Rd>, <Rm> #<immed_5>$^{+}$	Logical shift left by immediate number: `Rd = Rd << immed_5`
LSL <Rd>, <Rs>+	Logical shift left by number in register: `Rd = Rd << Rs`
LSR <Rd>, <Rm> #<immed_5>$^{+}$	Logical shift right by immediate number: `Rd = Rd >> immed_5`
LSR <Rd>, <Rs>$^{+}$	Logical shift right by number in register: `Rd = Rd >> Rs`
MOV <Rd>, #<immed_8>$^{+}$	Move immediate 8-bit value to register: `Rd = immed_8`
MOV <Rd>, <Rn>$^{+}$	Move low register value to low register
MOV <Rd>, <Rm>	Move high or low register value to high or low register
MUL <Rd>, <Rm>+	Multiply register value: `Rd = Rd * Rm`
MVN <Rd>, <Rm>+	Move complement of register value to register: `Rd = NOT(Rm)`
NEG <Rd>,<Rm>+	Negative register value and store in register: `Rd = 0-Rm`

(Continued)

Table A.1 (Continued)

Assembler	Operation
NOP	No operation
ORR <Rd>, <Rm>$^+$	Bitwise OR register Rd = Rd OR Rm
POP <registers>	Pop registers from stack
POP <registers, PC>	Pop registers and PC from stack
PUSH <registers>	Push registers into stack
PUSH <registers, LR>	Push registers and LR into stack
REV <Rd>, <Rn>	Reverse bytes in word and copy to register: Rd = {Rn[7:0],Rn[15:8],Rn[23:16],Rn[31:24]}
REV16 <Rd>, <Rn>	Reverse bytes in two half words and copy to register: Rd = {Rn[23:16],Rn[31:24],Rn[7:0],Rn[15:8]}
REVSH <Rd>, <Rn>	Reverse bytes in low half word[15:0], sign-extend, and copy to register: Rd = SignExtend({Rn[7:0],Rn[15:8]})
ROR <Rd>, <Rs>$^+$	Rotate right by amount in register
SBC <Rd>, <Rm>$^+$	Subtract register value and borrow (~C) from register value: Rd = Rd - Rm - NOT(C)
SEV	Send event
STMIA <Rn>!, <registers>	Store multiple registers word to sequential memory locations
STR <Rd>,[<Rn>,#<immed_5>*4]	Store register word to register address + 5-bit immediate offset
STR <Rd>,[<Rn>,<Rm>]	Store register word to base register address + register offset
STR <Rd>,[PC,#<immed_8>*4]	Store register word to PC address + 8-bit immediate offset
STR <Rd>,[SP,#<immed_8>*4]	Store register word to SP address + 8-bit immediate offset
STRB <Rd>,[<Rn>,#<immed_5>]	Store register byte[7:0] to register address + 5-bit immediate offset
STRB <Rd>,[<Rn>,<Rm>]	Store register byte[7:0] tp register address + register offset
STRH <Rd>,[<Rn>,#<immed_5>*2]	Store register half word[15:0] to register address + 5-bit immediate offset
STRH <Rd>,[<Rn>,<Rm>]	Store register half word[15:0] to register address + register offset
SUB <Rd>, <Rn>, #<immed_3>$^+$	Subtract immediate 3-bit value from register: Rd = Rn - immed_3
SUB <Rd>, #<immed_8>$^+$	Subtract immediate 8-bit value from register value: Rd = Rd - immed_8
SUB <Rd>, <Rn>, <Rm>$^+$	Subtract register values: Rd = Rn - Rm

Table A.1 (Continued)

Assembler	Operation
`SUB SP, #<immed_7>*4`	Subtract 4(immediate 7-bit value) from SP: `SP = SP - immed_7*4`
`SVC <immed_8>`	Operating system service call with 8-bit immediate call code
`SXTB <Rd>, <Rm>`	Extract byte[7:0] from register, move to register, and sign extend to 32-bit
`SXTH <Rd>, <Rm>`	Extract half word[15:0] from register, move to register, and sign extend to 32-bit
`TST <Rn>, <Rm>`+	Test register value for set bits by ANDing it with another register value: `Rn AND Rm`
`UXTB <Rd>, <Rm>`	Extract byte[7:0] from register, move to register, and zero extend to 32-bit
`UXTH <Rd>, <Rm>`	Extract half word[15:0] from register, move to register, and zero extend to 32-bit
`WFE`	Wait for event
`WFI`	Wait for interrupt

Supported 32-Bit Thumb-2 Instructions

Instructions with *{S}* update flags (APSR) only if the *S* suffix is used. Instructions marked with a plus sign (+) indicate that the flag (APSR) gets updated.

Note: To support immediate data of commonly required value ranges, many Thumb-2 instructions use an immediate data-encoding scheme, labeled *modify_constant* in Table A.2. The details of this encoding scheme are documented in the *ARM Architecture Application Level Reference Manual,* Section A5.2, "Immediate Constants. "

Table A.2 Supported 32-bit Instructions

Assembler	Operation
`ADC{S}.W <Rd>,<Rn>, #<modify_constant(immed_12)>`	Add register value, immediate value, and C flag to register value: `Rd = Rd + modify_constant(immed_12) + C`
`ADC{S}.W <Rd>,<Rn>,<Rm>{,<shift>}`	Add register value, shifted register value, and C bit: `Rd = Rn + (Rm<<shift) + C`
`ADD{S}.W <Rd>,<Rn>, #<modify_constant(immed_12)>`	Add register value and immediate value to register value: `Rd = Rd + modify_constant(immed_12)`

(Continued)

Table A.2 (Continued)

Assembler	Operation
ADD{S}.W <Rd>,<Rn>,<Rm>{,<shift>}	Add register value, shifted register value: Rd = Rn + (Rm<<shift)
ADDW.W <Rd>, <Rn>, #<immed_12>	Add register value and immediate 12-bit value
AND{S}.W <Rd>,<Rn>, #<modify_constant(immed_12)>	Bitwise AND register value with immediate value
AND{S}.W <Rd>,<Rn>,<Rm>{,<shift>}	Bitwise AND register value with shifted register value
ASR{S}.W <Rd>,<Rn>,<Rm>	Arithmetic shift right by number in register
B.W	Unconditional branch
B{cond}.W <label>	Conditional branch
BFC.W <Rd>, #<lsb>, #<width>	Clear bit field
BFI.W <Rd>, <Rn>, #<lsb>, #<width>	Insert bit field from one register value into another
BIC{S}.W <Rd>,<Rn>, #<modify_constant(immed_12)>	Bitwise AND register value with complement of immediate value
BIC{S}.W <Rd>,<Rn>,<Rm>{,<shift>}	Bitwise AND register value with complement of shifted register value
BL <label>	Branch with link
CLZ.W <Rd>, <Rn>	Count leading zeros in register value
CLREX.W	Clear exclusive access monitor status
CMN.W <Rn>, #<modify_constant(immed_12)> +	Compare register value with two's complement of immediate value
CMN.W <Rn>, <Rm>{, <shift>}+	Compare register value with two's complement of register value
CMP.W <Rn>, #<modify_constant(immed_12)> +	Compare register value with immediate value
CMP.W <Rn>, <Rm>{, <shift>}+	Compare register value with register value
DMB	Data memory barrier
DSB	Data synchronization barrier
EOR{S}.W <Rd>,<Rn>, #<modify_constant(immed_12)>	Exclusive OR register value with immediate value
EOR{S}.W <Rd>,<Rn>,<Rm>{,<shift>}	Exclusive OR register value with shifted register value
ISB	Instruction synchronization barrier
LDM{IA|DB}.W <Rn>{!}, <registers>	Load multiple memory register, increment after or decrement before
LDR.W <Rxf>, [<Rn>,#<offset_12>]	Read memory word from base register address + immediate 12-bit value offset

Table A.2 (Continued)

Assembler	Operation
`LDR.W <Rxf>, [<Rn>], #+/-<offset_8>`	Read memory word from base register address with immediate 8-bit value offset, post-indexed
`LDR.W <Rxf>, [<Rn>, #+/-<offset_8>]!`	Read memory word from base register address with immediate 8-bit value offset, pre-indexed
`LDR.W <Rxf>, [<Rn>,<Rm> {,LSL #<shift>}]`	Read memory word from base register address with shifted register value offset (shift range from 0 to 3)
`LDR.W <Rxf>, [PC, #+/-<offset_12>]`	Read memory word from PC with immediate 12-bit value offset
`LDR.W PC, [<Rn>,#<offset_12>]`	Read branch target from base register address + immediate 12-bit value offset and branch
`LDR.W PC, [<Rn>], #+/-<offset_8>`	Read branch target from base register address with immediate 8-bit value offset, post-indexed, and branch
`LDR.W PC, [<Rn>, #+/-<offset_8>]!`	Read branch target from base register address with immediate 8-bit value offset, pre-indexed, and branch
`LDR.W PC, [<Rn>,<Rm> {,LSL #<shift>}]`	Read branch target from base register address with shifted register value offset (shift range from 0 to 3), and branch
`LDR.W PC, [PC, #+/-<offset_12>]`	Read branch target from PC with immediate 12-bit value offset and branch
`LDRB.W <Rxf>, [<Rn>,#<offset_12>]`	Read memory byte from base register address + immediate 12-bit value offset
`LDRB.W <Rxf>, [<Rn>], #+/-<offset_8>`	Read memory byte from base register address with immediate 8-bit value offset, post-indexed
`LDRB.W <Rxf>, [<Rn>, #+/-<offset_8>]!`	Read memory byte from base register address with immediate 8-bit value offset, pre-indexed
`LDRB.W <Rxf>, [<Rn>,<Rm> {,LSL #<shift>}]`	Read memory byte from base register address with shifted register value offset (shift range from 0 to 3)
`LDRB.W <Rxf>, [PC, #+/-<offset_12>]`	Read memory byte from PC with immediate 12-bit value offset
`LDRD.W <Rxf1>,<Rxf2>, [<Rn>,#+/-<offset_8>*4] {!}`	Read double word from memory from base register address +/− immediate offset, pre-indexed
`LDRD.W <Rxf1>,<Rxf2>, [<Rn>],#+/-<offset_8>*4`	Read double word from memory from base register address +/− immediate offset, post-indexed
`LDREX.W <Rxf>, [<Rn> {,#<offset_8>*4}]`	Exclusive load word from base register address with immediate offset
`LDREXB.W <Rxf>,[<Rn>]`	Exclusive load byte from register address
`LDREXH.W <Rxf>,[<Rn>]`	Exclusive load half word from register address

(Continued)

Table A.2 (Continued)

Assembler	Operation
`LDRH.W <Rxf>, [<Rn>,#<offset_12>]`	Read memory half word from base register address + immediate 12-bit value offset
`LDRH.W <Rxf>, [<Rn>], #+/-<offset_8>`	Read memory half word from base register address with immediate 8-bit value offset, post-indexed
`LDRH.W <Rxf>, [<Rn>, #+/-<offset_8>]!`	Read memory half word from base register address with immediate 8-bit value offset, pre-indexed
`LDRH.W <Rxf>, [<Rn>,<Rm> {,LSL #<shift>}]`	Read memory half word from base register address with shifted register value offset (shift range from 0 to 3)
`LDRH.W <Rxf>, [PC, #+/-<offset_12>]`	Read memory half word from PC with immediate 12-bit value offset
`LDRSB.W <Rxf>, [<Rn>,#<offset_12>]`	Read memory byte from base register address + immediate 12-bit value offset, sign extend, and copy to register
`LDRSB.W <Rxf>, [<Rn>], #+/-<offset_8>`	Read memory byte from base register address with immediate 8-bit value offset, sign extend, and copy to register, post-indexed
`LDRSB.W <Rxf>, [<Rn>, #+/-<offset_8>]!`	Read memory byte from base register address with immediate 8-bit value offset, sign extend, and copy to register, pre-indexed
`LDRSB.W <Rxf>, [<Rn>,<Rm> {,LSL #<shift>}]`	Read memory byte from base register address with shifted register value offset (shift range from 0 to 3), sign extend, and copy to register
`LDRSB.W <Rxf>, [PC, #+/-<offset_12>]`	Read memory byte from PC with immediate 12-bit value offset, sign extend, and copy to register
`LDRSH.W <Rxf>, [<Rn>,#<offset_12>]`	Read memory half word from base register address + immediate 12-bit value offset, sign extend, and copy to register
`LDRSH.W <Rxf>, [<Rn>], #+/-<offset_8>`	Read memory half word from base register address with immediate 8-bit value offset, sign extend, and copy to register, post-indexed
`LDRSH.W <Rxf>, [<Rn>, #+/-<offset_8>]!`	Read memory half word from base register address with immediate 8-bit value offset, sign extend, and copy to register, pre-indexed
`LDRSH.W <Rxf>, [<Rn>,<Rm> {,LSL #<shift>}]`	Read memory half word from base register address with shifted register value offset (shift range from 0 to 3), sign extend, and copy to register
`LDRSH.W <Rxf>, [PC, #+/-<offset_12>]`	Read memory half word from PC with immediate 12-bit value offset, sign extend, and copy to register

Table A.2 (Continued)

Assembler	Operation
`LDRT.W   <Rxf>, [<Rn>,#<offset_8>]`	Word store with translation; in privileged mode, write to base register address with immediate offset and with user access level
`LDRBT.W  <Rxf>, [<Rn>,#<offset_8>]`	Byte store with translation; in privileged mode, write to base register address with immediate offset and with user access level
`LDRHT.W  <Rxf>, [<Rn>,#<offset_8>]`	Half-word store with translation; in privileged mode, write to base register address with immediate offset and with user access level
`LSL{S}.W  <Rd>, <Rn>, <Rm>`	Logical shift left register value by number in register
`LSR{S}.W  <Rd>, <Rn>, <Rm>`	Logical shift right register value by number in register
`MLA.W   <Rd>, <Rn>, <Rm>, <Racc>`	Multiple accumulate: Multiply two signed or unsigned register values and add the low 32 bits to a register value: `Rd = (Rn*Rm) + Racc`
`MLS.W   <Rd>, <Rn>, <Rm>, <Racc>`	Multiply and subtract; multiply two signed or unsigned register values and subtract the low 32 bits from a register value: `Rd = Racc - (Rn*Rm)`

`MOV{S}.W  <Rd>,`
`#<modify_constant(immed_12)>`
	Move immediate value to register:
	`Rd = modify_constant(immed_12)`

`MOV{S}.W  <Rd>, <Rm>{, <shift>}`	Move shifted register value to register
`MOVT.W  <Rd>, #<immed_16>`	Move immediate 16-bit value to top half word[31:16] of register; lower half not affected
`MOVW.W  <Rd>, #<immed_16>`	Move immediate 16-bit value to bottom half word[15:0] of register and clear upper half word
`MRS  <Rd>, <sreg>`	Read special registers and copy to register
`MSR  <sreg>,<Rd>`	Write register value to special register
`MUL.W  <Rd>, <Rn>, <Rm>`	Multiply two signed or unsigned values: `Rd = Rm * Rn`
`NOP.W`	No operation

`ORN{S}.W <Rd>,<Rn>,`
`#<modify_constant(immed_12)>`
	Bitwise OR NOT register value with immediate value

`ORN{S}.W <Rd>,<Rn>,<Rm>{,<shift>}`	Bitwise OR NOT register value with shifted register value

`ORR{S}.W <Rd>,<Rn>,`
`#<modify_constant(immed_12)>`
	Bitwise OR register value with immediate value

`ORR{S}.W <Rd>,<Rn>,<Rm>{,<shift>}`	Bitwise OR register value with shifted register value

(Continued)

Table A.2 (Continued)

Assembler	Operation
`POP.W   <registers>`	Pop registers from stack
`POP.W   <registers, PC>`	Pop registers and PC from stack
`PUSH.W  <registers>`	Push registers into stack
`PUSH.W  <registers, LR>`	Push registers and LR into stack
`RBIT.W  <Rd>, <Rm>`	Reverse bit order
`REV.W   <Rd>, <Rm>`	Reverse bytes in word
`REV16.W <Rd>, <Rn>`	Reverse bytes in each half word
`REVSH.W <Rd>, <Rn>`	Reverse bytes in bottom half word and sign extend
`ROR{S}.W <Rd>, <Rn>, <Rm>`	Rotate right by number in register

`RSB{S}.W <Rd>,<Rn>,`
`#<modify_constant(immed_12)>`
	Reverse subtract register value from immediate value

`RSB{S}.W <Rd>,<Rn>,<Rm>{,<shift>}`	Reverse subtract register value from shifted register value
`RRX{S}.W <Rd>, <Rm>`	Rotate right extended by 1 bit

`SBC{S}.W <Rd>,<Rn>,`
`#<modify_constant(immed_12)>`
	Subtract immediate value and C bit from register value

`SBC{S}.W <Rd>,<Rn>,<Rm>{,<shift>}`	Subtract shifted register value and C bit from register value

`SBFX.W  <Rd>, <Rn>,`
`#<lsb>, #<width>`
	Copy bit field from register and sign extend to 32 bits

`SDIV.W  <Rd>, <Rn>, <Rm>`	Signed divide: `Rd = Rn/Rm`
`SEV`	Send Event

`SMLAL.W  <RdLo>, <RdHi>,`
`<Rn>, <Rm>`
	Multiply signed words and add signed extended value to two-register value:
	`{RdHi,RdLo} = (Rn*Rm)+{RdHi,RdLo}`

`SMULL.W  <RdLo>, <RdHi>,`
`<Rn>, <Rm>`
	Multiply two signed register values:
	`{RdHi,RdLo} = (Rn*Rm)`

`SSAT.W  <Rd>, #<imm>,`
`<Rn>{,<shift>}`
	Signed saturate shifted register value to bit position in immediate value; update Q flag if saturation takes place

`STM{IA|DB}.W <Rn>{!}, <registers>`	Write multiple register words to consecutive memory locations; increment after or decrement before
`STR.W  <Rxf>, [<Rn>, #<offset_12>]`	Write word to base register address + immediate 12-bit value offset

`STR.W  <Rxf>, [<Rn>],`
` #+/-<offset_8>`
	Write word to base register address with immediate 8-bit value offset, post-indexed

Table A.2 (Continued)

Assembler	Operation
`STR.W <Rxf>, [<Rn>, #+/-<offset_8>]!`	Write word to base register address with immediate 8-bit value offset, pre-indexed
`STR.W <Rxf>, [<Rn>,<Rm> {,LSL #<shift>}]`	Write word to base register address with shifted register value offset (shift range from 0 to 3)
`STRB.W <Rxf>, [<Rn>, #<offset_12>]`	Write byte to base register address + immediate 12-bit value offset
`STRB.W <Rxf>, [<Rn>], #+/-<offset_8>`	Write byte to base register address with immediate 8-bit value offset, post-indexed
`STRB.W <Rxf>, [<Rn>, #+/-<offset_8>]!`	Write byte to base register address with immediate 8-bit value offset, pre-indexed
`STRB.W <Rxf>, [<Rn>,<Rm> {,LSL #<shift>}]`	Write byte to base register address with shifted register value offset (shift range from 0 to 3)
`STRD.W <Rxf1>,<Rxf2>, [<Rn>,#+/-<offset_8>*4] {!}`	Write double word to memory from base register address +/− immediate offset, pre-indexed
`STRD.W <Rxf1>,<Rxf2>, [<Rn>],#+/-<offset_8>*4`	Write double word to memory from base register address +/− immediate offset, post-indexed
`STREX.W <Rxf>, [<Rn> {,#<offset_8>*4}]`	Exclusive store word from base register address with immediate offset
`STREXB.W <Rxf>,[<Rn>]`	Exclusive store byte from register address
`STREXH.W <Rxf>,[<Rn>]`	Exclusive store half word from register address
`STRH.W <Rxf>, [<Rn>, #<offset_12>]`	Write half word to base register address + immediate 12-bit value offset
`STRH.W <Rxf>, [<Rn>], #+/-<offset_8>`	Write half word to base register address with immediate 8-bit value offset, post-indexed
`STRH.W <Rxf>, [<Rn>, #+/-<offset_8>]!`	Write half word to base register address with immediate 8-bit value offset, pre-indexed
`STRH.W <Rxf>, [<Rn>,<Rm> {,LSL #<shift>}]`	Write half word to base register address with shifted register value offset (shift range from 0 to 3)
`STRT.W <Rxf>, [<Rn>,#<offset_8>]`	Word store with translation; in privileged mode, write to base register address with immediate offset and with user access level
`STRBT.W <Rxf>, [<Rn>,#<offset_8>]`	Byte store with translation; in privileged mode, write to base register address with immediate offset and with user access level
`STRHT.W <Rxf>, [<Rn>,#<offset_8>]`	Half word store with translation; in privileged mode, write to base register address with immediate offset and with user access level

(Continued)

Table A.2 (Continued)

Assembler	Operation
`SUB{S}.W <Rd>,<Rn>, #<modify_constant(immed_12)>`	Subtract immediate value from register: `Rd = Rd - modify_constant(immed_12)`
`SUB{S}.W <Rd>,<Rn>,<Rm>{,<shift>}`	Subtract shifted register value from register: `Rd = Rn + (Rm<<shift)`
`SUBW.W <Rd>, <Rn>, #<immed_12>`	Subtract immediate 12-bit value from register: `Rd = Rd - immed_12`
`SXTB.W <Rd>, <Rm>, {, <rotation>}`	Sign extend byte to 32 bits; Rd=sign_extend(byte(rotate_right(Rm))), rotate can be 0–3 bytes
`SXTH.W <Rd>, <Rm>, {, <rotation>}`	Sign extend halfword to 32 bits; Rd=sign_extend(hword(rotate_right(Rm))), rotate can be 0–3 bytes.
`TBB.W [<Rn>, <Rm>]`	Table branch byte
`TBH.W [<Rn>, <Rm>, LSL #1]`	Table branch half word
`TEQ.W <Rn>, #<modify_constant(immed_12)>` +	Test equivalent between register and immediate value
`TEQ.W <Rn>,<Rm> {,<shift>}`+	Test equivalent between register and shifted register
`TST.W <Rn>, #<modify_constant(immed_12)>` +	Test register value for set bits by ANDing it with immediate value
`TST.W <Rn>,<Rm> {,<shift>}`+	Test register value for set bits by ANDing it with shifted register
`UBFX.W <Rd>, <Rn>, #<lsb>, #<width>`	Copy bit field from register and zero extend to 32 bits
`UDIV.W <Rd>, <Rn>, <Rm>`	Unsigned divide: `Rd = Rn/Rm`
`UMLAL.W <RdLo>, <RdHi> , <Rn>, <Rm>`	Multiply unsigned words and add unsigned extended value to two-register value: `{RdHi,RdLo} = (Rn*Rm)+{RdHi,RdLo}`
`UMULL.W <RdLo>, <RdHi> , <Rn>, <Rm>`	Multiply two unsigned register values: `{RdHi,RdLo} = (Rn*Rm)`
`USAT.W <Rd>, #<imm>, <Rn>{,<shift>}`	Unsigned saturate shifted register value to bit position in immediate value
`UXTB.W <Rd>, <Rm>, {, <rotation>}`	Unsign extend byte to 32 bits; Rd=unsign_extend(byte(rotate_right(Rm))), rotate can be 0–3 bytes.

Table A.2 (Continued)

Assembler	Operation
`UXTH.W <Rd>, <Rm>, {, <rotation>}`	Unsign extend half word to 32 bits; Rd=unsign_extend(hword(rotate_right(Rm))), rotate can be 0–3 bytes.
`WFE.W`	Wait for event
`WFI.W`	Wait for interrupt

This page intentionally left blank

APPENDIX B

16-Bit Thumb Instructions and Architecture Versions

Most of the 16-bit Thumb instructions are available in architecture v4T (ARM7TDMI). However, a number of them are added in architecture v5, v6, and v7. Table B.1 lists these instructions.

Table B.1 Change of 16-bit Instruction Support in Various Recent ARM Architecture Versions

Instruction	v4T	v5	v6	Cortex-M3 (v7-M)
BKPT	N	Y	Y	Y
BLX	N	Y	Y	BLX <reg> only
CBZ, CBNZ	N	N	N	Y
CPS	N	N	Y	CPSIE <i/f>, CPSID <i/f>
CPY	N	N	Y	Y
NOP	N	N	N	Y
IT	N	N	N	Y
REV (various forms)	N	N	Y	REV, REV16, REVSH
SEV	N	N	N	Y
SETEND	N	N	Y	N
SWI	Y	Y	Y	Changed to SVC
SXTB, SXTH	N	N	Y	Y
UXTB, UXTH	N	N	Y	Y
WFE, WFI	N	N	N	Y

This page intentionally left blank

APPENDIX C

Cortex-M3 Exceptions Quick Reference

Exception Types and Enables

Table C.1 Quick Summary of Cortex-M3 Exception Types and Their Priority Configurations

Exception Type	Name	Priority (Level Address)	Enable
1	Reset	−3	Always
2	NMI	−2	Always
3	Hard fault	−1	Always
4	MemManage	Programmable (0xE000ED18)	NVIC SHCSR (0xE000ED24) bit[16]
5	BusFault	Programmable (0xE000ED19)	NVIC SHCSR (0xE000ED24) bit[17]
6	Usage fault	Programmable (0xE000ED1A)	NVIC SHCSR (0xE000ED24) bit[18]
7–10	–	–	–
11	SVC	Programmable (0xE000ED1F)	Always
12	Debug monitor	Programmable (0xE000ED20)	NVIC DEMCR (0xE000EDFC) bit[16]
13	–	–	–
14	PendSV	Programmable (0xE000ED22)	Always
15	SysTick	Programmable (0xE000ED23)	SYSTICK CTRLSTAT (0xE000E010) bit[1]
16–255	IRQ	Programmable (0xE000E400)	NVIC SETEN (0xE000E100)

Stack Contents After Exception Stacking

Table C.2 Exception Stack Frame

Address	Data	Push Order
Old SP (N) ->	(Previously pushed data)	–
(N–4)	PSR	2
(N–8)	PC	1
(Nv12)	LR	8
(N–16)	R12	7
(N–20)	R3	6
(N–24)	R2	5
(N–28)	R1	4
New SP (N–32) ->	R0	3
Note: If double word stack alignment feature is used and the SP was not double word aligned when the exception occured, the stack frame top might begin at ((OLD_SP-4) AND OXFFFFFFF8), and the rest of the tabe moves one word down.		

APPENDIX D

NVIC Registers Quick Reference

Table D.1 Interrupt Controller Type Register (0xE000E004)

Bits	Name	Type	Reset Value	Description
4:0	INTLINESNUM	R	–	Number of interrupt inputs in step of 32: 0 = 1 to 32 1 = 33 to 64 ...

Table D.2 SYSTICK Control and Status Register (0xE000E010)

Bits	Name	Type	Reset Value	Descriptions
16	COUNTFLAG	R	0	Read as 1 if counter reach 0 since last time this register is read. Clear to 0 automatically when read or when current counter value is cleared.
2	CLKSOURCE	R/W	0	0 = external reference clock (STCLK) 1 = use core clock
1	TICKINT	R/W	0	1 = Enable SYSTICK interrupt generation when SYSTICK timer reaches 0 0 = Do not generate interrupt
0	ENABLE	R/W	0	SYSTICK timer enable

Table D.3 SYSTICK Reload Value Register (0xE000E014)

Bits	Name	Type	Reset Value	Descriptions
23:0	RELOAD	R/W	0	Reload value when timer reaches 0

Table D.4 SYSTICK Current Value Register (0xE000E018)

Bits	Name	Type	Reset Value	Descriptions
23:0	CURRENT	R/Wc	0	Read to return current value of the timer Write to clear counter to 0; clearing of current value should also clear COUNTFLAG in SYSTICK Control and Status register

Table D.5 SYSTICK Calibration Value Register (0xE000E01C)

Bits	Name	Type	Reset Value	Description
31	NOREF	R	–	1 = No external reference clock (STCLK not available) 0 = External reference clock available
30	SKEW	R	–	1 = calibration value is not exactly 10 ms 0 = calibration value is accurate
23:0	TENMS	R/W	0	Calibration value for 10 ms. SoC designer should provide this value via Cortex-M3 input signals. If this value is read as 0, it means calibration value is not available.

Table D.6 External Interrupt SETEN Registers (0xE000E100–0xE000E11C)

Address	Name	Type	Reset Value	Description
0xE000E100	SETENA0	R/W	0	Enable for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 ... bit[31] for interrupt #31
0xE000E104	SETENA1	R/W	0	Enable for external interrupt #32–63
...	–	–	–	–

Table D.7 External Interrupt CLREN Registers (0xE000E180–0xE000E19C)

Address	Name	Type	Reset Value	Description
0xE000E180	CLRENA0	R/W	0	Clear Enable for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 ... bit[31] for interrupt #31

(Continued)

Table D.7 (Continued)

Address	Name	Type	Reset Value	Description
0xE000E184	CLRENA1	R/W	0	Clear Enable for external interrupt #32–63
…	–	–	–	–

Table D.8 External Interrupt SETPEND Registers (0xE000E200–0xE000E21C)

Address	Name	Type	Reset Value	Description
0xE000E200	SETPEND0	R/W	0	Pending for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 … bit[31] for interrupt #31
0xE000E204	SETPEND1	R/W	0	Pending for external interrupt #32–63
…	–	–	–	–

Table D.9 External Interrupt CLRPEND Registers (0xE000E280–0xE000E29C)

Address	Name	Type	Reset Value	Description
0xE000E280	CLRPEND0	R/W	0	Clear Pending for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 … bit[31] for interrupt #31
0xE000E284	CLRPEND1	R/W	0	Clear Pending for external interrupt #32–63
…	–	–	–	–

Table D.10 External Interrupt ACTIVE Registers (0xE000E300–0xE000E31C)

Address	Name	Type	Reset Value	Description
0xE000E300	ACTIVE0	R	0	Active status for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 … bit[31] for interrupt #31
0xE000E304	ACTIVE1	R	0	Active status for external interrupt #32–63
…	–	–	–	–

Table D.11 External Interrupt Priority-Level Register (0xE000E400–0xE000E4EF; Listed as Byte Addresses)

Address	Name	Type	Reset Value	Description
0xE000E400	PRI_0	R/W	0	Priority level external interrupt #0
0xE000E401	PRI_1	R/W	0	Priority level external interrupt #1
...	–	–	–	–
0xE000E41F	PRI_31	R/W	0	Priority level external interrupt #31
...	–	–	–	–

Table D.12 CPU ID Base Register (Address 0xE000ED00)

Bits	Name	Type	Reset Value	Description
31:24	IMPLEMENTER	R	0x41	Implementer code; ARM is 0x41
23:20	VARIANT	R	0x0 / 0x1	Implementation defined variant number
19:16	Constant	R	0xF	Constant
15:4	PARTNO	R	0xC23	Part number
3:0	REVISION	R	0x0 / 0x1	Revision code

Table D.13 Interrupt Control and State Register (0xE000ED04)

Bits	Name	Type	Reset Value	Description
31	NMIPENDSET	R/W	0	NMI pended
28	PENDSVSET	R/W	0	Write 1 to pend system call; Read value indicates pending status
27	PENDSVCLR	W	0	Write 1 to clear PendSV pending status
26	PENDSTSET	R/W	0	Write 1 to pend Systick exception; Read value indicates pending status
25	PENDSTCLR	W	0	Write 1 to clear Systick pending status
23	ISRPREEMPT	R	0	Indicate that a pending interrupt is going to be active in next step (for debug)
22	ISRPENDING	R	0	External interrupt pending (excluding system exceptions such as NMI for fault)
21:12	VECTPENDING	R	0	Pending ISR number
11	RETTOBASE	R	0	Set to 1 when the processor is running an exception handler and will return to thread level if interrupt return and no other exceptions pending
9:0	VECTACTIVE	R	0	Current running interrupt service routine

Table D.14 Vector Table Offset Register (Address 0xE000ED08)

Bits	Name	Type	Reset Value	Description
29	TBLBASE	R/W	0	Table base in Code (0) or RAM (1)
28:7	TBLOFF	R/W	0	Table offset value from Code region or RAM region

Table D.15 Application Interrupt and Reset Control Register (Address 0xE000ED0C)

Bits	Name	Type	Reset Value	Description
31:16	VECTKEY	R/W	–	Access key; 0x05FA must be written to this field to write to this register, otherwise the write will be ignored. The read-back value is 0xFA05.
15	ENDIANESS	R	–	Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian. This can only change after a reset.
10:8	PRIGROUP	R/W	0	Priority Group.
2	SYSRESETREQ	W	–	Request chip control logic to generate a reset.
1	VECTCLRACTIVE	W	–	Clear all active state information for exceptions. Typically used in debugging or OS to allow the system to recover from a system error. (Reset is safer.)
0	VECTRESET	W	–	Reset the Cortex-M3 (except debug logic), but this will not reset circuits outside the processor.

Table D.16 System Control Register (0xE000ED10)

Bits	Name	Type	Reset Value	Description
4	SEVONPEND	R/W	0	Send Event on Pending. Wake up from WFE if a new interrupt is pended, regardless of whether the interrupt has priority higher than current level.
3	Reserved	–	–	–
2	SLEEPDEEP	R/W	0	Enable SLEEPDEEP output signal when entering sleep mode.
1	SLEEPONEXIT	R/W	0	Enable SleeponExit feature.
0	Reserved	–	–	–

Table D.17 Configuration Control Register (0xE000ED14)

Bits	Name	Type	Reset Value	Description
9	STKALIGN	R/W	0	Force exception-stacking start in double-word aligned address.[1]
8	BFHFNMIGN	R/W	0	Ignore data bus fault during hard fault and NMI handlers.
7:5	Reserved	–	–	Reserved.
4	DIV_0_TRP	R/W	0	Trap on divide by 0.
3	UNALIGN_TRP	R/W	0	Trap on unaligned accesses.
2	Reserved	–	–	Reserved.
1	USERSETMPEND	R/W	0	If set to 1, allow user code to write to Software Trigger Interrupt register.
0	NONBASETHRDENA	R/W	0	Nonbase thread enable. If set to 1, allows exception handler to return to thread state at any level by controlling return value.

[1] Only available from revision 1 of the Cortex-M3. Revision 0 does not have this feature.

Table D.18 System Exceptions Priority-Level Register (0xE000ED18–0xE000ED23; Listed as Byte Addresses)

Address	Name	Type	Reset Value	Description
0xE000ED18	PRI_4	R/W	0	Priority level for memory management fault
0xE000ED19	PRI_5	R/W	0	Priority level for bus fault
0xE000ED1A	PRI_6	R/W	0	Priority level for usage fault
0xE000ED1B	–	–	–	–
0xE000ED1C	–	–	–	–
0xE000ED1D	–	–	–	–
0xE000ED1E	–	–	–	–
0xE000ED1F	PRI_11	R/W	0	Priority level for SVC
0xE000ED20	PRI_12	R/W	0	Priority level for debug monitor
0xE000ED21	–	–	–	–
0xE000ED22	PRI_14	R/W	0	Priority level for PendSV
0xE000ED23	PRI_15	R/W	0	Priority level for SYSTICK

Table D.19 System Handler Control and State Register (0xE000ED24)

Bits	Name	Type	Reset Value	Description
18	USGFAULTENA	R/W	0	Usage fault handler enable
17	BUSFAULTENA	R/W	0	Bus fault handler enable

Table D.19 (Continued)

Bits	Name	Type	Reset Value	Description
16	MEMFAULTENA	R/W	0	Memory management fault enable
15	SVCALLPENDED	R/W	0	SVC pended; SVCall is started but was replaced by a higher priority exception
14	BUSFAULTPENDED	R/W	0	Bus Fault pended; bus fault handler is started but was replaced by a higher-priority exception
13	MEMFAULTPENDED	R/W	0	Memory management fault pended; memory management fault started but was replaced by a higher-priority exception
12	USGFAULTPENDED	R/W	0	Usage fault pended; usage fault started but was replaced by a higher-priority exception
11	SYSTICKACT	R/W	0	Read as 1 if SYSTICK exception is active
10	PENDSVACT	R/W	0	Read as 1 if PendSV exception is active
8	MONITORACT	R/W	0	Read as 1 if debug monitor exception is active
7	SVCALLACT	R/W	0	Read as 1 if SVCall exception is active
3	USGFAULTACT	R/W	0	Read as 1 if usage fault exception is active
1	BUSFAULTACT	R/W	0	Read as 1 if bus fault exception is active
0	MEMFAULTACT	R/W	0	Read as 1 if memory management fault is active
Note: Bit 12 (USGFAULTPENDED) is not available on revision 0 of Cortex-M3.				

Table D.20 Memory Management Fault Status Register (0xE000ED28; Byte Size**)**

Bits	Name	Type	Reset Value	Description
7	MMARVALID	–	0	Indicate MMAR is valid
6:5	–	–	–	–
4	MSTKERR	R/Wc	0	Stacking error
3	MUNSTKERR	R/Wc	0	Unstacking error
2	–	–	–	–
1	DACCVIOL	R/Wc	0	Data access violation
0	IACCVIOL	R/Wc	0	Instruction access violation

Table D.21 Bus Fault Status Register (0xE000ED29; Byte Size**)**

Bits	Name	Type	Reset Value	Description
7	BFARVALID	–	0	Indicate BFAR is valid
6:5	–	–	–	–

(Continued)

Table D.21 (Continued)

Bits	Name	Type	Reset Value	Description
4	STKERR	R/Wc	0	Stacking error
3	UNSTKERR	R/Wc	0	Unstacking error
2	IMPREISERR	R/Wc	0	Imprecise data access violation
1	PRECISERR	R/Wc	0	Precise data access violation
0	IBUSERR	R/Wc	0	Instruction access violation

Table D.22 Usage Fault Status Register (0xE000ED2A; Half Word Size)

Bits	Name	Type	Reset Value	Description
9	DIVBYZERO	R/Wc	0	Indicate divide by zero will take place (can only be set if DIV_0_TRP is set)
8	UNALIGNED	R/Wc	0	Indicate unaligned access will take place (can only be set if UNALIGN_TRP is set)
7:4	–	–	–	–
3	NOCP	R/Wc	0	Attempt to execute a coprocessor instruction
2	INVPC	R/Wc	0	Attempt to do exception with bad value in EXC_RETURN number
1	INVSTATE	R/Wc	0	Attempt to switch to invalid state (e.g., ARM)
0	UNDEFINSTR	R/Wc	0	Attempt to execute an undefined instruction

Table D.23 Hard Fault Status Register (0xE000ED2C)

Bits	Name	Type	Reset Value	Description
31	DEBUGEVT	R/Wc	0	Indicate hard fault is triggered by debug event
30	FORCED	R/Wc	0	Indicate hard fault is taken because of bus fault/memory management fault/ usage fault
29:2	–	–	–	–
1	VECTBL	R/Wc	0	Indicate hard fault is caused by failed vector fetch
0	–	–	–	–

Table D.24 Debug Fault Status Register (0xE000ED30)

Bits	Name	Type	Reset Value	Description
4	EXTERNAL	R/Wc	0	EDBGRQ signal asserted
3	VCATCH	R/Wc	0	Vector fetch occurred

Table D.24 (Continued)

Bits	Name	Type	Reset Value	Description
2	DWTTRAP	R/Wc	0	DWT match occurred
1	BKPT	R/Wc	0	BKPT instruction executed
0	HALTED	R/Wc	0	Halt requested in NVIC

Table D.25 Memory Manage Address Register MMAR (0xE000ED34)

Bits	Name	Type	Reset Value	Description
31:0	MMAR	R	–	Address that caused memory manage fault

Table D.26 Bus FaultManage Address Register BFAR (0xE000ED38)

Bits	Name	Type	Reset Value	Description
31:0	BFAR	R	–	Address that caused bus fault

Table D.27 Auxiliary Fault Status Register (0xE000ED3C)

Bits	Name	Type	Reset Value	Description
31:0	Vendor controlled	R/Wc	0	Vendor controlled (optional)

Table D.28 MPU Type Register (0xE000ED90)

Bits	Name	Type	Reset Value	Description
23:16	IREGION	R	0	Number Instruction region; because ARM v7-M architecture uses a unified MPU, this is always 0
15:8	DREGION	R	0 or 8	Number of regions supported by this MPU
0	SEPARATE	R	0	This is always 0 because the MPU is always unified

Table D.29 MPU Control Register (0xE000ED94)

Bits	Name	Type	Reset Value	Description
2	PRIVDEFENA	R/W	0	Privileged default memory map enable
1	HFNMIENA	R/W	0	If set to 1, enable MPU during hard fault handler and NMI handler; otherwise, the MPU is not enabled for the hard fault handler and NMI
0	ENABLE	R/W	0	Enable the MPU if set to 1

Table D.30 MPU Region Number Register (0xE000ED98)

Bits	Name	Type	Reset Value	Description
7:0	REGION	R/W	–	Select a region to be programmed

Table D.31 MPU Region Base Address Register (0xE000ED9C)

Bits	Name	Type	Reset Value	Description
31:N	ADDR	R/W	–	Base address of the region; N is dependent on the region size.
4	VALID	R/W	–	If this is 1, the REGION defined in bit[3:0] will be used in this programming step; otherwise, the region selected by MPU Region Number register is used
3:0	REGION	R/W	–	This field overrides MPU Region Number register if VALID is 1; otherwise, this is ignored

Table D.32 MPU Region Base Attribute and Size Register (0xE000EDA0)

Bits	Name	Type	Reset Value	Description
31:29	Reserved	–	–	–
28	XN	R/W	–	Instruction access Disable (1=Disable)
27	Reserved	–	–	–
26:24	AP	R/W	–	Data Access Permission field
23:22	Reserved	–	–	–
21:19	TEX	R/W	–	Type Extension field
18	S	R/W	–	Shareable
17	C	R/W	–	Cacheable
16	B	R/W	–	Bufferable
15:8	SRD	R/W	–	Subregion disable
7:6	Reserved	–	–	–
5:1	REGION SIZE	R/W	–	MPU Protection Region size
0	SZENABLE	R/W	–	Region enable

Table D.33 MPU Alias Registers (0xE000EDA4–0xE000EDB8)

Address	Name	Description
0xE000EDA4	Alias of D9C	MPU Alias 1 Region Base Address register
0xE000EDA8	Alias of DA0	MPU Alias 1 Region Attribute and Size register

Table D.33 (Continued)

Address	Name	Description
0xE000EDAC	Alias of D9C	MPU Alias 2 Region Base Address register
0xE000EDB0	Alias of DA0	MPU Alias 2 Region Attribute and Size register
0xE000EDB4	Alias of D9C	MPU Alias 3 Region Base Address register
0xE000EDB8	Alias of DA0	MPU Alias 3 Region Attribute and Size register

Table D.34 Debug Halting Control and Status Register (0xE000EDF0)

Bits	Name	Type	Reset Value	Description
31:16	KEY	W	–	Debug key; value of 0xA05F must be written to this field to write to this register, otherwise the write will be ignored
25	S_RESET_ST	R	–	Core has been reset or is being reset; this bit is clear on read
24	S_RETIRE_ST	R	–	Instruction is completed since last read; this bit is clear on read
19	S_LOCKUP	R	–	When this bit is 1, the core is in locked state
18	S_SLEEP	R	–	When this bit is 1, the core is in sleep mode
17	S_HALT	R	–	When this bit is 1, the core is halted
16	S_REGRDY	R	–	Register read/write operation is completed
15:6	Reserved	–	–	Reserved
5	C_SNAPSTALL	R/W	–	Use to break a stalled memory access
4	Reserved	–	–	Reserved
3	C_MASKINTS	R/W	–	Mask interrupts while stepping; can only be modified when the processor is halted
2	C_STEP	R/W	–	Single step the processor; valid only if C_DEBUGEN is set
1	C_HALT	R/W	–	Halt the processor core; valid only if C_DEBUGEN is set
0	C_DEBUGEN	R/W	–	Enable halt mode debug

Table D.35 Debug Core Register Selector Register (0xE000EDF4)

Bits	Name	Type	Reset Value	Description
16	REGWnR	W	–	Direction of data transfer: Write = 1, Read = 0
15:5	Reserved	–	–	–

(Continued)

Table D.35 (Continued)

Bits	Name	Type	Reset Value	Description
4:0	REGSEL	W	–	Register to be accessed: 00000 = R0 00001 = R1 ... 01111 = R15 10000 = xPSR/Flags 10001 = MSP (Main Stack Pointer) 10010 = PSP (Process Stack Pointer) 10100 = Special registers : [31:24] Control [23:16] FAULTMASK [15:8] BASEPRI [7:0] PRIMASK Others values are reserved

Table D.36 Debug Core Register Data Register (0xE000EDF8)

Bits	Name	Type	Reset Value	Description
31:0	Data	R/W	–	Data register to hold register read result or to write data into selected register

Table D.37 Debug Exception and Monitor Control Register (0xE000EDFC)

Bits	Name	Type	Reset Value	Description
24	TRCENA	R/W	0	Trace system enable; to use DWT, ETM, ITM, and TPIU, this bit must be set to 1
23:20	Reserved	–	–	Reserved
19	MON_REQ	R/W	0	Indication that the debug monitor is caused by a manual pending request rather than hardware debug events
18	MON_STEP	R/W	0	Single step the processor; valid only if MON_EN is set
17	MON_PEND	R/W	0	Pend the monitor exception request; the core will enter a monitor exception when priority allowed

Table D.37 (Continued)

Bits	Name	Type	Reset Value	Description
16	MON_EN	R/W	0	Enable the debug monitor exception
15:11	Reserved	–	–	Reserved
10	VC_HARDERR	R/W	0	Debug trap on hard faults
9	VC_INTERR	R/W	0	Debug trap on interrupt/exception service errors
8	VC_BUSERR	R/W	0	Debug trap on bus faults
7	VC_STATERR	R/W	0	Debug trap on usage fault state errors
6	VC_CHKERR	R/W	0	Debug trap on usage fault-enabled checking errors (e.g., unaligned, divide by zero)
5	VC_NOCPERR	R/W	0	Debug trap on usage fault no coprocessor errors
4	VC_MMERR	R/W	0	Debug trap on memory management fault
3:1	Reserved	–	–	Reserved
0	VC_CORERESET	R/W	0	Debug trap on core reset

Table D.38 Software Trigger Interrupt Register (0xE000EF00)

Bits	Name	Type	Reset Value	Description
8:0	INTID	W	–	Writing the interrupt number set the pending bit of the interrupt.

Table D.39 NVIC Peripheral ID Registers (0xE000EFD0–0xE000EFFC)

Address	Name	Type	Reset Value	Description
0xE000EFD0	PERIPHID4	R	0x04	Peripheral ID register
0xE000EFD4	PERIPHID5	R	0x00	Peripheral ID register
0xE000EFD8	PERIPHID6	R	0x00	Peripheral ID register
0xE000EFDC	PERIPHID7	R	0x00	Peripheral ID register
0xE000EFE0	PERIPHID0	R	0x00	Peripheral ID register
0xE000EFE4	PERIPHID1	R	0xB0	Peripheral ID register
0xE000EFE8	PERIPHID2	R	0x0B/0x1B	Peripheral ID register
0xE000EFEC	PERIPHID3	R	0x00	Peripheral ID register
0xE000EFF0	PCELLID0	R	0x0D	Component ID register
0xE000EFF4	PCELLID1	R	0xE0	Component ID register
0xE000EFF8	PCELLID2	R	0x05	Component ID register
0xE000EFFC	PCELLID0	R	0xB1	Component ID register
Note: PERIPHID2 value is 0x0B for Cortex-M3 revision 0, 0x1B for revision 1.				

This page intentionally left blank

APPENDIX E

Cortex-M3 Troubleshooting Guide

Overview

One of the challenges of using the Cortex-M3 is to locate problems when the program goes wrong. The Cortex-M3 processor provides a number of fault status registers to assist in troubleshooting (see Table E.1).

Table E.1 Fault Status Registers on Cortex-M3

Address	Register	Full Name	Size
0xE000ED28	MMSR	MemManage Fault Status register	Byte
0xE000ED29	BFSR	Bus Fault Status register	Byte
0xE000ED2A	UFSR	Usage Fault Status register	Half word
0xE000ED2C	HFSR	Hard Fault Status register	Word
0xE000ED30	DFSR	Debug Fault Status register	Word
0xE000ED3C	AFSR	Auxiliary Fault Status register	Word

The MMSR, BFSR, and UFSR registers can be accessed in one go using a word transfer instruction. In this situation the combined fault status register is called the Configurable Fault Status Register (CFSR).

Another important piece of information is the stacked Program Counter (PC). This is located in memory address [SP + 0x24]. Since there are two stack pointers in the Cortex-M3, the fault handler might need to determine which stack pointer was used before obtaining the stacked PC.

In addition, for bus faults and memory management faults, you might also able to determine the address that caused the fault. This is done by accessing the MemManage

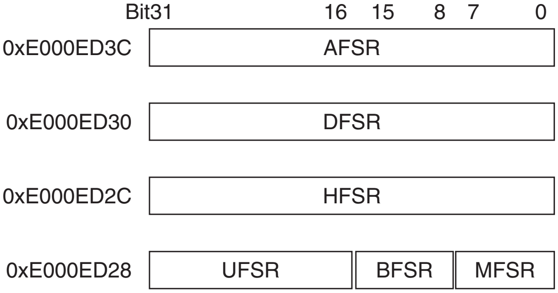


Figure E.1 Accessing Fault Status Registers

(Memory Management) Fault Address Register (MMAR) and the Bus Fault Address Register (BFAR). The contents of these two registers are only valid when the MMAVALID bit (in MMSR) or BFARVALID bit (in BFSR) is set. The MMAR and BFAR are physically the same register, so only one of them can be valid at a time (see Table E.2).

Table E.2 Fault Address Registers on Cortex-M3

Address	Register	Full Name	Size
0xE000ED34	MMAR	MemManage Fault Address register	Word
0xE000ED38	BFAR	Bus Fault Address register	Word

Finally, the Link Register (LR) value when entering the fault handler might also provide hints about the cause of the fault. In the case of faults caused by invalid EXC_RETURN value, the value of LR when the fault handler is entered shows the previous LR value when the fault occured. Fault handler can report the faulty LR value, and software programmers can then use this information to check why the LR ends up with an illegal return value.

Developing Fault Handlers

In most cases, fault handlers for development and for real running systems differ from one another. For software development, the fault handler should focus on reporting the type of error, whereas the fault handler for running systems will likely focus on system recovery actions. Here we cover only the fault reporting because system recovery actions highly depend on design type and requirements.

In complex software, instead of outputting the results inside the fault handler, the contents of these registers can be copied to a memory block and then you can use PendSV to report the fault details later. This avoids potential faults in display or outputting routines causing lockup. For simple applications this might not matter, and the fault details can be output directly within the fault handler routine.

Report Fault Status Registers

The most basic step of a fault handler is to report the fault status register values. These include:

- UFSR
- BFSR
- MMSR
- HFSR
- DFSR
- AFSR (optional)

Report Stacked PC

The step for getting the stacked PC is similar to the SVC example in this book.

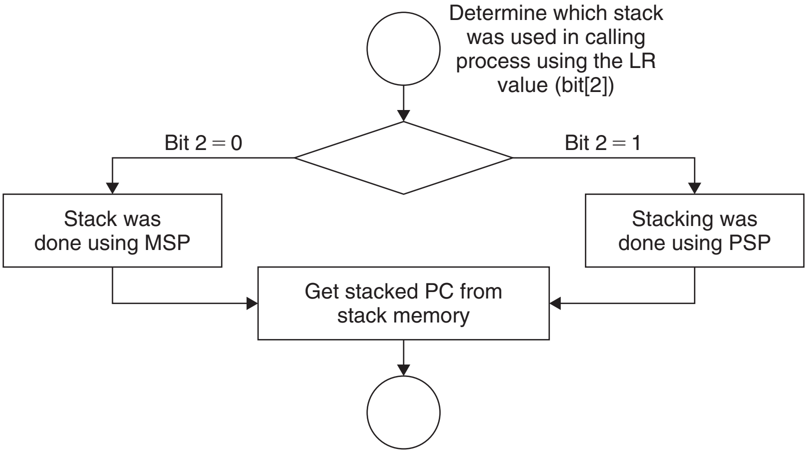

Determine which stack was used in calling process using the LR value (bit[2])
Bit 2 = 0
Bit 2 = 1
Stack was done using MSP
Stacking was done using PSP
Get stacked PC from stack memory

Figure E.2 Getting the Value of a Stacked PC from Stack Memory

This process can be carried out in assembly language as:

```
TST    LR, #0x4     ; Test EXC_RETURN number in LR bit 2
ITTEE  EQ           ; if zero (equal) then
MRSEQ  R0, MSP      ; Main Stack was used, put MSP in R0
LDREQ  R0,[R0,#24] ; Get stacked PC from stack.
MRSNE  R0, PSP     ; else, Process Stack was used, put PSP in R0
LDRNE  R0,[R0,#24] ; Get stacked PC from stack.
```

To help with debugging, we should also create a disassembled code list file so that we can locate the problem easily.

Read Fault Address Register

The fault address register can be erased after the MMARVALID or BFARVALID is cleared. To correctly access the fault address register, the following procedure should be used:

1. Read BFAR/MMAR.
2. Read BFARVALID/MMARVALID. If it is zero, the BFAR/MMAR read should be discarded.
3. Clear BFARVALID/MMARVALID.

The reason for this procedure instead of reading valid bits first is to prevent a fault handler being preempted by another higher-priority fault handler after the valid bit is read, which could lead to the following erroneous fault-reporting sequence:

1. Read BFARVALID/MMARVALID.
2. Valid bit is set, going to read BFAR/MMAR.
3. Higher-priority exception preempts existing fault handler, which generates another fault, causing another fault handler to be executed.
4. The higher-priority fault handler clears the BFARVALID/MMARVALID bit, causing the BFAR/MMAR to be erased.
5. After returning to the original fault handler, the BFAR/MMAR is read, but now the content is invalid and leads to incorrect reporting of the fault address.

Therefore it is important to read the BFARVALID/MMARVALID after reading the Fault Address register to ensure that the address register content is valid.

Clear Fault Status Bits

After the fault reporting is done, the fault status bit in the FSR should be cleared so that next time the fault handler is executed, the previous faults will not confuse the fault handler. In addition, if the fault address valid bit is not clear, the fault address register will not get an update for the next fault.

Others

It is often necessary to save the contents of LR in the beginning of a fault handler. However, if the fault is caused by a stack error, pushing the LR to stack might just make things worst. As we know, R0–R3 and R12 should already been saved, so we could copy LR to one of these registers before doing any function calls.

Understanding the Cause of the Fault

After obtaining the information we need, we can establish the cause of the problem. Tables E.3–E.7 list some of the common reasons that faults occur.

Table E.3 MemManage Fault Status Register

Bit	Possible Causes
MSTKERR	Error occurred during stacking (starting of exception): 1) Stack pointer is corrupted. 2) Stack size goes too large, reaching a region not defined by the MPU or disallowed in the MPU configuration.
MUNSTKERR	Error occurred during unstacking (ending of exception). If there was no error stacking but the error occurred during unstacking, it might be: 1. Stack pointer was corrupted during exception. 2. MPU configuration changed by exception handler.
DACCVIOL	Violation to memory access protection, which is defined by MPU setup. For example, user application trying to access privileged-only region.
IACCVIOL	1. Violation to memory access protection, which is defined by MPU setup. For example, user application trying to access privileged-only region. Stacked PC might be able to locate the code that has caused the problem. 2. Branch to nonexecutable regions. 3. Invalid exception return code. 4. Invalid entry in exception vector table. For example, loading of an executable image for traditional ARM core into the memory, or exception occurred before vector table is set. 5. Stacked PC corrupted during exception handling.

Table E.4 Bus Fault Status Register

Bit	Possible Causes
STKERR	Error occurred during stacking (starting of exception): 1. Stack pointer is corrupted. 2. Stack size goes too large, reaching an undefined memory region. 3. PSP is used but not initialized.
UNSTKERR	Error occurred during unstacking (ending of exception). If there was no error stacking but error occurred during unstacking, it might be that the stack pointer was corrupted during exception.
IMPREISERR	Bus error during data access; could be caused by a device not having been initialized, access of privileged-only device in user mode, or the transfer size is incorrect for the specific device.

(Continued)

Table E.4 (Continued)

Bit	Possible Causes
PRECISERR	Bus error during data access. The fault address may be indicated by BFAR. A bus error could be caused by a device not having been initialized, access of privileged-only device in user mode, or the transfer size is incorrect for the specific device.
IBUSERR	1. Violation to memory access protection, which is defined by MPU setup. For example, user application trying to branch to privileged-only region. 2. Branch to nonexecutable regions. 3. Invalid exception return code. 4. Invalid entry in exception vector table. For example, loading an executable image for traditional ARM core into the memory, or exception occurred before vector table is set. 5. Stacked PC corrupted during exception handling.

Table E.5 Usage Fault Status Register

Bit	Possible Causes
DIVBYZERO	Divide by zero takes place and DIV_0_TRP is set. The code causing the fault can be located using stacked PC.
UNALIGNED	Unaligned access attempted with UNALIGN_TRP is set. The code causing the fault can be located using stacked PC.
NOCP	Attempt to execute a coprocessor instruction. The code causing the fault can be located using stacked PC.
INVPC	1. Invalid value in EXC_RETURN number during exception return. For example: • Return to thread with EXC_RETURN = 0xFFFFFFF1 • Return to handler with EXC_RETURN = 0xFFFFFFF9 To investigate the problem, the current LR value provides the value of LR at the failing exception return. 2. Invalid exception active status. For example: • Exception return with exception active bit for the current exception already cleared. Possibly caused by use of VECTCLRACTIVE or clearing of exception active status in NVIC SHCSR. • Exception return to thread with one (or more) exception active bit still active. 3. Stack corruption causing the stacked IPSR to be incorrect. For an INVPC fault, the Stacked PC shows the point where the faulting exception interrupted the main/preempted program. To investigate the cause of the problem, it is best to use the exception trace feature in ITM. 4. ICI/IT bit invalid for current instruction. This can happen when a multiple-load/store instruction gets interrupted and, during the interrupt handler, the stacked PC is modified. When the interrupt return takes place, the nonzero ICI bit is applied to an instruction that does not use ICI bits. The same problem can also happen due to corruption of stacked PSR.

Table E.5 (Continued)

Bit	Possible Causes
INVSTATE	1. Loading branch target address to PC with LSB equals zero. Stacked PC should show the branch target. 2. LSB of vector address in vector table is zero. Stacked PC should show the starting of exception handler. 3. Stacked PSR corrupted during exception handling, so after the exception the core tries to return to the interrupted code in ARM state.
UNDEFINSTR	1. Use of instructions not supported in the Cortex-M3. 2. Bad/corrupted memory contents. 3. Loading of ARM object code during link stage. Check compile steps. 4. Instruction align problem. For example, if the GNU tool chain is used, omitting of .align after .ascii might cause the next instruction to be unaligned (to start in an odd memory address instead of half word addresses).

Table E.6 Hard Fault Status Register

Bit	Possible Causes
DEBUGEVF	Fault is caused by debug event: 1. Breakpoint/watchpoint events. 2. If the hard fault handler is executing, it might be caused by execution of BKPT without enable monitor handler (MON_EN=0) and halt debug not enabled (C_DEBUGEN=0). By default some C compilers might include semihosting code that use BKPT.
FORCED	1. Trying to run SVC/BKPT within SVC/monitor or another handler with same or higher priority. 2. A fault occurred, but its corresponding handler is disabled or cannot be started because another exception with same or higher priority is running or because exception mask is set.
VECTBL	Vector fetch failed. Could be caused by: 1. Bus fault at vector fetch 2. Incorrect vector table offset setup

Table E.7 Debug Fault Status Register

Bit	Possible Causes
EXTERNAL	EDBGRQ signal has been asserted.
VCATCH	Vector catch event has occurred.
DWTTRAP	DWT watchpoint event has occurred.
BKPT	1. Breakpoint instruction is executed. 2. FPB unit generated a breakpoint event.

(Continued)

Table E.7 (Continued)

Bit	Possible Causes
	In some cases BKPT instructions are inserted by C startup code as part of the semihosting debugging setup. This should be removed for a real application code. Refer to your compiler document for details.
HALTED	Halt request in NVIC.

Other Possible Problems

A number of other common problems are in Table E.8.

Table E.8 Other Possible Problems

Situation	Possible Causes
No program execution	Vector table could be set up incorrectly: • Located in incorrect memory location. • LSB of vectors (including hard fault handler) is not set to 1. • Use of branch instruction (as in vector table in traditional ARM processor) in the vector table. Generate a disassembly code listing to check whether the vector table is set up correctly.
Program crashes after a few instructions	Possibly caused by incorrect endian setting or incorrect stack pointer setup (check vector table) or use of C object library for traditional ARM processor (ARM code instead of Thumb code). The offending C object library code could be part of the C startup routine. Check compiler and linker options to ensure that Thumb or Thumb-2 library files are used.

Index