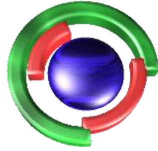
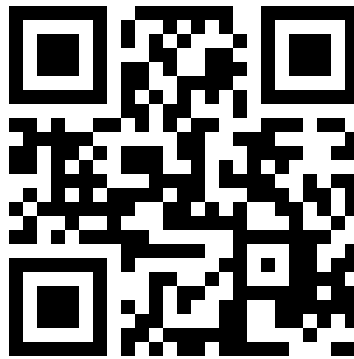
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs



By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

The Definitive Guide to the ARM® Cortex-M3

Second Edition

Joseph Yiu

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

Newnes is an imprint of Elsevier

https://hemanthrajhemu.github.io

Contents

Foreword by Paul Kimelman xvii
Foreword by Richard York xx
Foreword by Wayne Lyons xxi
Preface xxiii
Acknowledgments xxiii
Conventions xxiv
Terms and Abbreviations xxv

CHAPTER 1 Introduction 1

1.1 What Is the ARM Cortex-M3 Processor? 1
1.2 Background of ARM and ARM Architecture 2
1.2.1 A Brief History 2
1.2.2 Architecture Versions 3
1.2.3 Processor Naming 5
1.3 Instruction Set Development 7
1.4 The Thumb-2 Technology and Instruction Set Architecture 8
1.5 Cortex-M3 Processor Applications 9
1.6 Organization of This Book 10
1.7 Further Reading 10

CHAPTER 2 Overview of the Cortex-M3 11

2.1 Fundamentals 11
2.2 Registers 12
2.2.1 R0–R12: General-Purpose Registers 12
2.2.2 R13: Stack Pointers 12
2.2.3 R14: The Link Register 13
2.2.4 R15: The Program Counter 13
2.2.5 Special Registers 14
2.3 Operation Modes 14
2.4 The Built-In Nested Vectored Interrupt Controller 15
2.4.1 Nested Interrupt Support 15
2.4.2 Vectored Interrupt Support 16
2.4.3 Dynamic Priority Changes Support 16
2.4.4 Reduction of Interrupt Latency 16
2.4.5 Interrupt Masking 16
2.5 The Memory Map 16
2.6 The Bus Interface 17
2.7 The MPU 18

https://hemanthrajhemu.github.io

2.8 The Instruction Set 18
2.9 Interrupts and Exceptions 19
2.9.1 Low Power and High Energy Efficiency 20
2.10 Debugging Support 21
2.11 Characteristics Summary 22
2.11.1 High Performance 22
2.11.2 Advanced Interrupt-Handling Features 22
2.11.3 Low Power Consumption 23
2.11.4 System Features 23
2.11.5 Debug Supports 23

CHAPTER 3 **Cortex-M3 Basics 25**
3.1 Registers 25
3.1.1 General Purpose Registers R0 through R7 25
3.1.2 General Purpose Registers R8 through R12 25
3.1.3 Stack Pointer R13 26
3.1.4 Link Register R14 28
3.1.5 Program Counter R15 28
3.2 Special Registers 29
3.2.1 Program Status Registers 29
3.2.2 PRIMASK, FAULTMASK, and BASEPRI Registers 30
3.2.3 The Control Register 31
3.3 Operation Mode 32
3.4 Exceptions and Interrupts 35
3.5 Vector Tables 36
3.6 Stack Memory Operations 36
3.6.1 Basic Operations of the Stack 37
3.6.2 Cortex-M3 Stack Implementation 37
3.6.3 The Two-Stack Model in the Cortex-M3 39
3.7 Reset Sequence 40

CHAPTER 4 **Instruction Sets 43**
4.1 Assembly Basics 43
4.1.1 Assembler Language: Basic Syntax 43
4.1.2 Assembler Language: Use of Suffixes 44
4.1.3 Assembler Language: Unified Assembler Language 45
4.2 Instruction List 46
4.2.1 Unsupported Instructions 51
4.3 Instruction Descriptions 52
4.3.1 Assembler Language: Moving Data 53
4.3.2 LDR and ADR Pseudo-Instructions 56
4.3.3 Assembler Language: Processing Data 57

https://hemanthrajhemu.github.io

CHAPTER

Introduction

1

IN THIS CHAPTER

What Is the ARM Cortex-M3 Processor? 1
Background of ARM and ARM Architecture 2
Instruction Set Development 7
The Thumb-2 Technology and Instruction Set Architecture 8
Cortex-M3 Processor Applications 9
Organization of This Book 10
Further Reading 10

1.1 WHAT IS THE ARM CORTEX-M3 PROCESSOR?

The microcontroller market is vast, with more than 20 billion devices per year estimated to be shipped in 2010. A bewildering array of vendors, devices, and architectures is 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 communication channels and advanced peripherals are 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

Copyright © 2010, Elsevier Inc. All rights reserved.
DOI: 10.1016/B978-0-12-382091-4.00007-4

https://hemanthrajhemu.github.io

- *Enhanced determinism*: guaranteeing that critical tasks and interrupts are serviced as quickly as possible and 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 bits
- *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. Because 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 (MPU). 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.

1.2 BACKGROUND OF ARM AND ARM ARCHITECTURE

1.2.1 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.

https://hemanthrajhemu.github.io

THE CORTEX-M3 PROCESSOR VERSUS 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, readers are advised to check the particular chip manufacturer's documentation.

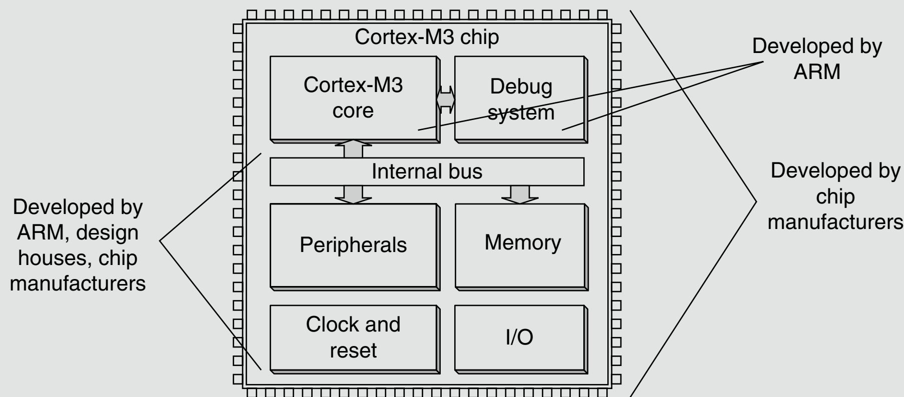


FIGURE 1.1

The Cortex-M3 Processor versus the Cortex-M3-Based MCU.

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 IPs. 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.

1.2.2 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).

https://hemanthrajhemu.github.io

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

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

- *A Profile (ARMv7-A)*: Application processors which are designed to handle complex applications such as high-end embedded operating systems (OSs) (e.g., Symbian, Linux, and Windows Embedded). These processors 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.

[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 any processor-based system, you may or may not be able to get this response due to choice of OS, interrupt latency, or memory latency, as well as if the CPU is running a higher priority interrupt.

https://hemanthrajhemu.github.io

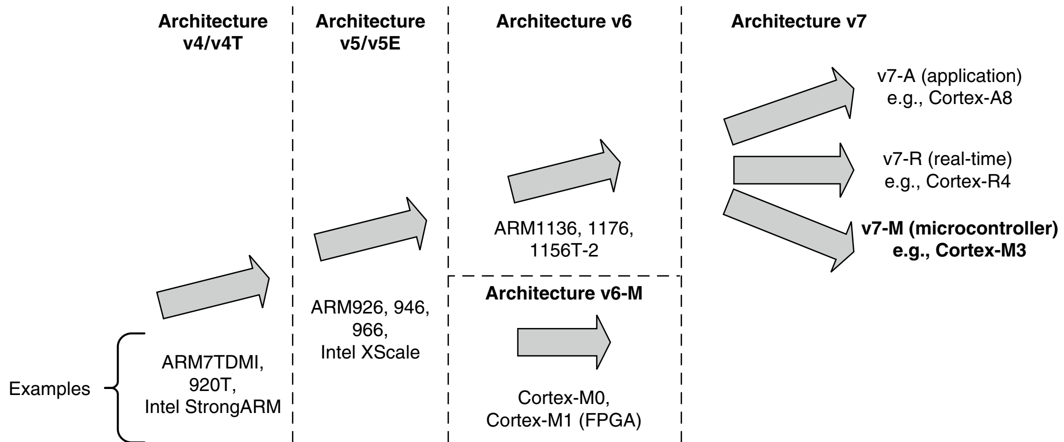


FIGURE 1.2

The Evolution of ARM Processor Architecture.

This book focuses on the Cortex-M3 processor, but it is only one of the Cortex product families that use 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), which is based on the ARMv7-R profile (see Figure 1.2).

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.

1.2.3 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

https://hemanthrajhemu.github.io

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 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.

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 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.

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-M0	ARMv6-M		NVIC
Cortex-M1	ARMv6-M	FPGA TCM interface	NVIC
Cortex-M3	ARMv7-M	MPU (optional)	NVIC

[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.

https://hemanthrajhemu.github.io

Table 1.1 ARM Processor Names *Continued*

Processor Name	Architecture Version	Memory Management Features	Other Features
Cortex-R4	ARMv7-R	MPU	DSP
Cortex-R4F	ARMv7-R	MPU	DSP + Floating point
Cortex-A8	ARMv7-A	MMU + TrustZone	DSP, Jazelle, NEON + floating point
Cortex-A9	ARMv7-A	MMU + TrustZone + multiprocessor	DSP, Jazelle, NEON + floating point

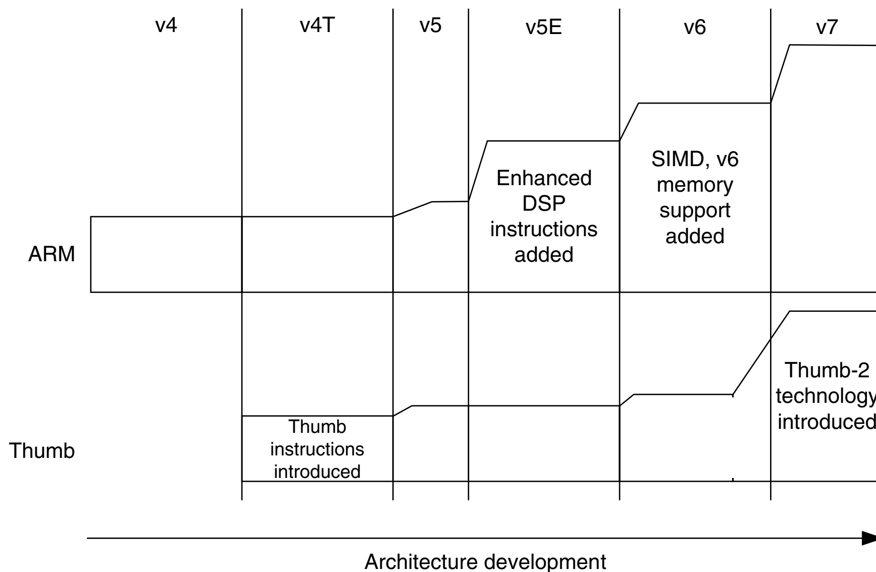


FIGURE 1.3

Instruction Set Enhancement.

1.3 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 (see Figure 1.3).

Historically (since ARM7TDMI), two different instruction sets are supported on the ARM processor: the ARM instructions that are 32 bits and Thumb instructions that are 16 bits. During program execution, the processor can be dynamically switched between the ARM state and the Thumb state to use either

https://hemanthrajhemu.github.io

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.

As the architecture version has been updated, extra instructions have been added to both ARM instructions and Thumb instructions. Appendix B 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 the Cortex-M3 instruction set, the complete details are specified in the *ARM v7-M Architecture Application Level Reference Manual* [Ref. 2]. Appendix A of this book also covers information regarding instruction sets required for software development.

1.4 THE THUMB-2 TECHNOLOGY AND INSTRUCTION SET ARCHITECTURE

The Thumb-2[3] technology extended the Thumb Instruction Set Architecture (ISA) into a highly efficient and powerful instruction set that delivers significant benefits in terms of ease of use, code size, and performance (see Figure 1.4). The extended instruction set in Thumb-2 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.

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

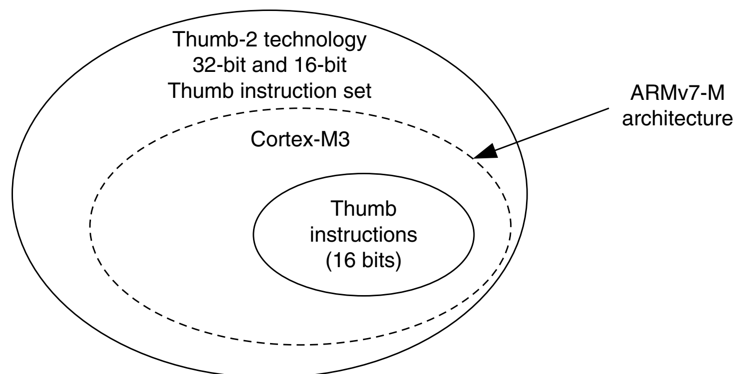


FIGURE 1.4

The Relationship between the Thumb Instruction Set in Thumb-2 Technology and the Traditional Thumb.

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

https://hemanthrajhemu.github.io

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.

With support for both 16-bit and 32-bit instructions in the Thumb-2 instruction 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.

1.5 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 MPU, making it ideal for highly integrated and cost-sensitive automotive applications.
- *Data communications*: The processor's low power and high efficiency, coupled with instructions in Thumb-2 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 on 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.

https://hemanthrajhemu.github.io

1.6 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 through 6, Cortex-M3 Basics
- Chapters 7 through 9, Exceptions and Interrupts
- Chapters 10 and 11, Cortex-M3 Programming
- Chapters 12 through 14, Cortex-M3 Hardware Features
- Chapters 15 and 16, Debug Supports in Cortex-M3
- Chapters 17 through 21, Application Development with Cortex-M3
- Appendices

1.7 FURTHER READING

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.
- *Cortex-M3 User Guides* are available from MCU vendors. In some cases, this user guide is available as a part of a complete microcontroller product manual. This document contains a programmer's model for the ARM Cortex-M3 processor, and instruction set details, and is customized by each MCU vendors to match their microcontroller implementations.
- 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 because it provides a summary on the Cortex-M3 processor.

https://hemanthrajhemu.github.io

CHAPTER

Overview of the Cortex-M3

2

IN THIS CHAPTER

Fundamentals 11
Registers 12
Operation Modes 14
The Built-In Nested Vectored Interrupt Controller 15
The Memory Map 16
The Bus Interface 17
The MPU 18
The Instruction Set 18
Interrupts and Exceptions 19
Debugging Support 21
Characteristics Summary 22

2.1 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 (see Figure 2.1). The processor has a Harvard architecture, which means that 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 performance of the processor 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 Memory Protection Unit (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 watchpoints.

Copyright © 2010, Elsevier Inc. All rights reserved.
DOI: 10.1016/B978-0-12-382091-4.00008-6

https://hemanthrajhemu.github.io

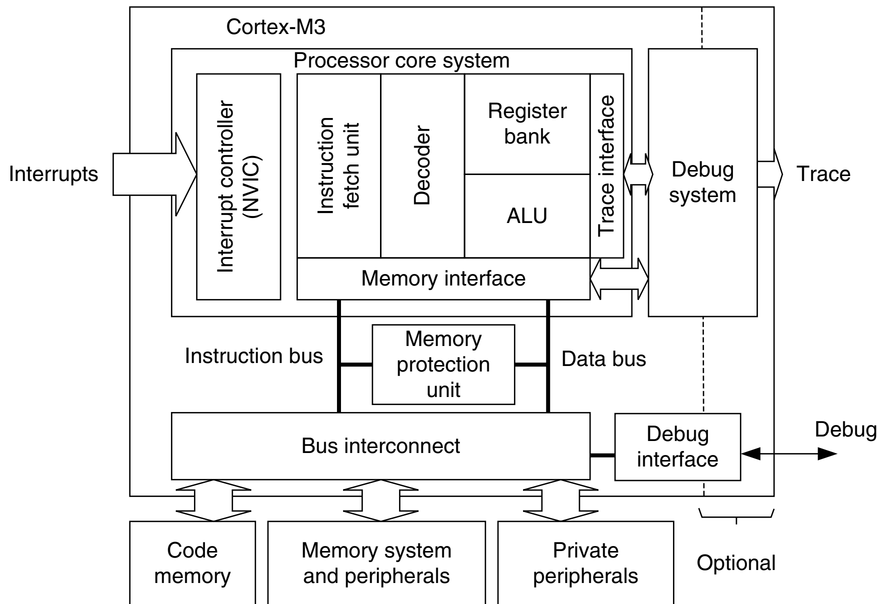


FIGURE 2.1

A Simplified View of the Cortex-M3.

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

2.2 REGISTERS

The Cortex-M3 processor has registers R0 through R15 (see Figure 2.2). R13 (the stack pointer) is banked, with only one copy of the R13 visible at a time.

2.2.1 R0–R12: General-Purpose Registers

R0–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–R7).

2.2.2 R13: Stack Pointers

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

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

The lowest 2 bits of the stack pointers are always 0, which means they are always word aligned.

https://hemanthrajhemu.github.io

Name		Functions (and banked registers)	
R0		General-purpose register	Low registers
R1		General-purpose register	
R2		General-purpose register	
R3		General-purpose register	
R4		General-purpose register	
R5		General-purpose register	
R6		General-purpose register	
R7		General-purpose register	
R8		General-purpose register	High registers
R9		General-purpose register	
R10		General-purpose register	
R11		General-purpose register	
R12		General-purpose register	
R13 (MSP)	R13 (PSP)	Main Stack Pointer (MSP), Process Stack Pointer (PSP)	
R14		Link Register (LR)	
R15		Program Counter (PC)	

FIGURE 2.2

Registers in the Cortex-M3.

2.2.3 R14: The Link Register

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

2.2.4 R15: The Program Counter

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

2.2.5 Special Registers

The Cortex-M3 processor also has a number of special registers (see Figure 2.3). They are as follows:

- Program Status registers (PSRs)
- Interrupt Mask registers (PRIMASK, FAULTMASK, and BASEPRI)
- Control register (CONTROL)

These registers have special functions and can be accessed only by special instructions. They cannot be used for normal data processing (see Table 2.1).

https://hemanthrajhemu.github.io

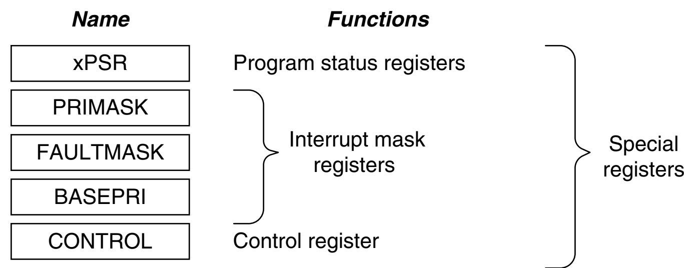


FIGURE 2.3

Special Registers in the Cortex-M3.

Table 2.1 Special Registers and Their Functions

Register	Function
xPSR	Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault
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

For more information on these registers, see Chapter 3.

2.3 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 (see Figure 2.4). 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.

When the processor is running a main program (thread mode), it can be either in 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 (see Figure 2.5). 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.

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,

https://hemanthrajhemu.github.io

	Privileged	*User*
When running an exception handler	Handler mode	
When not running an exception handler (e.g., main program)	Thread mode	Thread mode

FIGURE 2.4

Operation Modes and Privilege Levels in Cortex-M3.

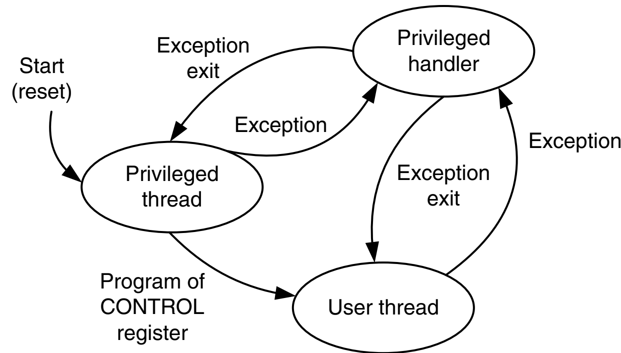


FIGURE 2.5

Allowed Operation Mode Transitions.

it can be used in conjunction with privilege levels to protect critical memory locations, such as programs and data for OSs.

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.

2.4 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 as follows:

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

2.4.1 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

https://hemanthrajhemu.github.io

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.

2.4.2 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.

2.4.3 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 ISR is completed, so their priority can be changed without risk of accidental reentry.

2.4.4 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, and handling of late arrival interrupts. Details of these optimization features are covered in Chapter 9.

2.4.5 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.

2.5 THE MEMORY MAP

The Cortex-M3 has a predefined memory map. This allows the built-in peripherals, such as the interrupt controller and the 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 ranges as shown in Figure 2.6.

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 Random Access Memory (RAM) region.

https://hemanthrajhemu.github.io

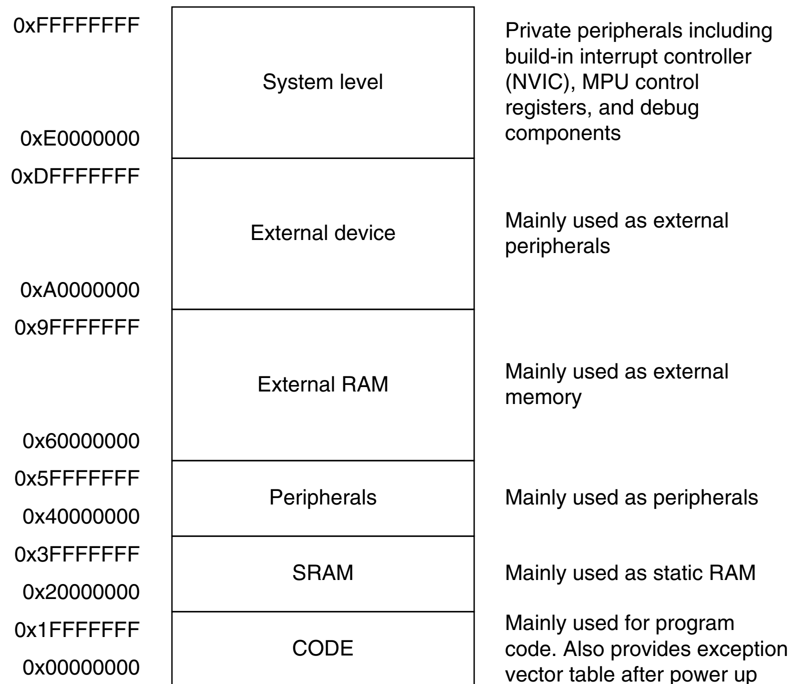


FIGURE 2.6

The Cortex-M3 Memory Map.

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

2.6 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 as follows:

- 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 other 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 Static Random Access Memory (SRAM), peripherals, external RAM, external devices, and part of the system-level memory regions.

https://hemanthrajhemu.github.io

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

2.7 THE MPU

The Cortex-M3 has an optional 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 OS can set up the MPU to protect data use by the OS kernel and other privileged processes 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.

2.8 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 central processing unit (CPU) had two operation states: a 32-bit ARM state and a 16-bit Thumb state. In the ARM state, the instructions are 32 bits and can execute all supported instructions with very high performance. In the Thumb state, the instructions are 16 bits, 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, see Figure 2.7) to switch between the states, and ARM and Thumb codes might need to be compiled separately in different files. This increases the complexity of software development and reduces maximum efficiency of the CPU core.

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 the 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, such as:

- 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

https://hemanthrajhemu.github.io

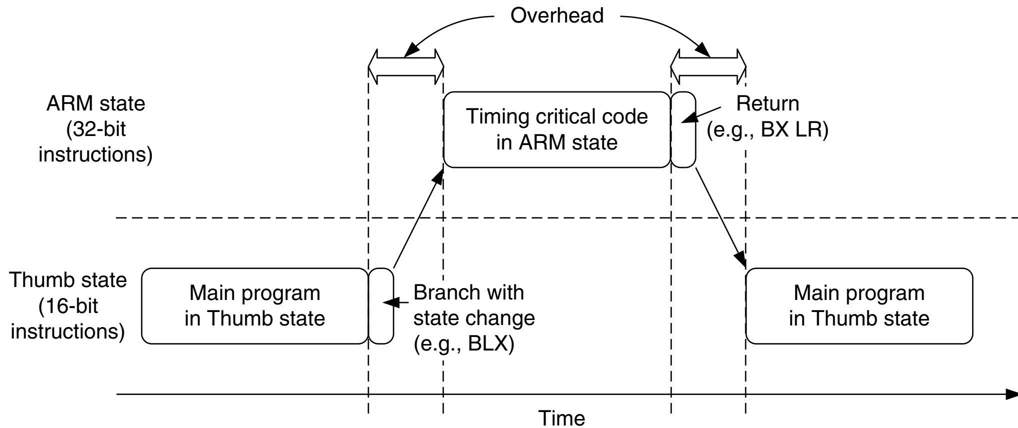


FIGURE 2.7

Switching between ARM Code and Thumb Code in Traditional ARM Processors Such as the ARM7.

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

- *UFBX, BFI, and BFC*: Bit field extract, insert, and clear instructions
- *UDIV and SDIV*: Unsigned and signed divide instructions
- *WFE, WFI, and SEV*: Wait-For-Event, Wait-For-Interrupts, and Send-Event; these allow the processor to enter sleep mode and to handle task synchronization on multiprocessor systems
- *MSR and MRS*: Move to special register from general-purpose register and move special register to general-purpose register; 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 use 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 Single Instruction–Multiple Data (SIMD) is not implemented on the Cortex-M3. In addition, a few Thumb instructions are not supported, such as Branch with Link and Exchange (BLX) with immediate (used to switch processor state from Thumb to ARM), a couple of change process state (CPS) instructions, and the SETEND (Set Endian) instructions, which were introduced in architecture v6. For a complete list of supported instructions, refer to Appendix A.

2.9 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

https://hemanthrajhemu.github.io

exception handling. It has a number of system exceptions plus a number of external Interrupt Request (IRQs) (external interrupt inputs). There is no fast interrupt (FIQ) (fast interrupt in ARM7/ARM9/ARM10/ARM11) 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.

2.9.1 Low Power and High Energy Efficiency

The Cortex-M3 processor is designed with various features to allow designers to develop low power and high energy efficient products. First, it has sleep mode and deep sleep mode supports, which can work with various system-design methodologies to reduce power consumption during idle period.

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	NMI (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	Program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	Supervisor 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 service
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-ISR is executed unconditionally.

https://hemanthrajhemu.github.io

Second, its low gate count and design techniques reduce circuit activities in the processor to allow active power to be reduced. In addition, since Cortex-M3 has high code density, it has lowered the program size requirement. At the same time, it allows processing tasks to be completed in a short time, so that the processor can return to sleep modes as soon as possible to cut down energy use. As a result, the energy efficiency of Cortex-M3 is better than many 8-bit or 16-bit microcontrollers.

Starting from Cortex-M3 revision 2, a new feature called Wakeup Interrupt Controller (WIC) is available. This feature allows the whole processor core to be powered down, while processor states are retained and the processor can be returned to active state almost immediately when an interrupt takes place. This makes the Cortex-M3 even more suitable for many ultra-low power applications that previously could only be implemented with 8-bit or 16-bit microcontrollers.

2.10 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 Joint Test Action Group (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. Through 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 Serial-Wire JTAG Debug Port (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 Personal Computer [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 processor 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.

https://hemanthrajhemu.github.io

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 them. 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.

2.11 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.

2.11.1 High Performance

The Cortex-M3 processor delivers high performance in microcontroller products:

- Many instructions, including multiply, are single cycle. Therefore, 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 bits) and the Thumb state (16 bits), 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 bits. 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.

2.11.2 Advanced Interrupt-Handling Features

The interrupt features on the Cortex-M3 processor are easy to use, very flexible, and provide high interrupt processing throughput:

- The built-in NVIC supports up to 240 external interrupt inputs. The vectored interrupt feature considerably 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.

https://hemanthrajhemu.github.io

- The Cortex-M3 processor automatically pushes registers R0–R3, R12, Link register (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 an 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.

2.11.3 Low Power Consumption

The Cortex-M3 processor is suitable for various low-power applications:

- 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 WFI or 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 low power or standard semiconductor process technology.

2.11.4 System Features

The Cortex-M3 processor provides various system features making it suitable for a large number of applications:

- 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.

2.11.5 Debug Supports

The Cortex-M3 processor includes comprehensive debug features to help software developers design their products:

- 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

https://hemanthrajhemu.github.io

- 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, make 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

https://hemanthrajhemu.github.io

CHAPTER

Cortex-M3 Basics

3

IN THIS CHAPTER

Registers 25
Special Registers 29
Operation Mode 32
Exceptions and Interrupts 35
Vector Tables 36
Stack Memory Operations 36
Reset Sequence 40

3.1 REGISTERS

As we've seen, the Cortex™-M3 processor has registers R0 through R15 and a number of special registers. R0 through R12 are general purpose, but some of the 16-bit Thumb® instructions can only access R0 through 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.

3.1.1 General Purpose Registers R0 through R7

The R0 through 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 bits; the reset value is unpredictable.

3.1.2 General Purpose Registers R8 through R12

The R8 through 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 bits; the reset value is unpredictable (see Figure 3.1).

Copyright © 2010, Elsevier Inc. All rights reserved.
DOI: 10.1016/B978-0-12-382091-4.00009-8

https://hemanthrajhemu.github.io

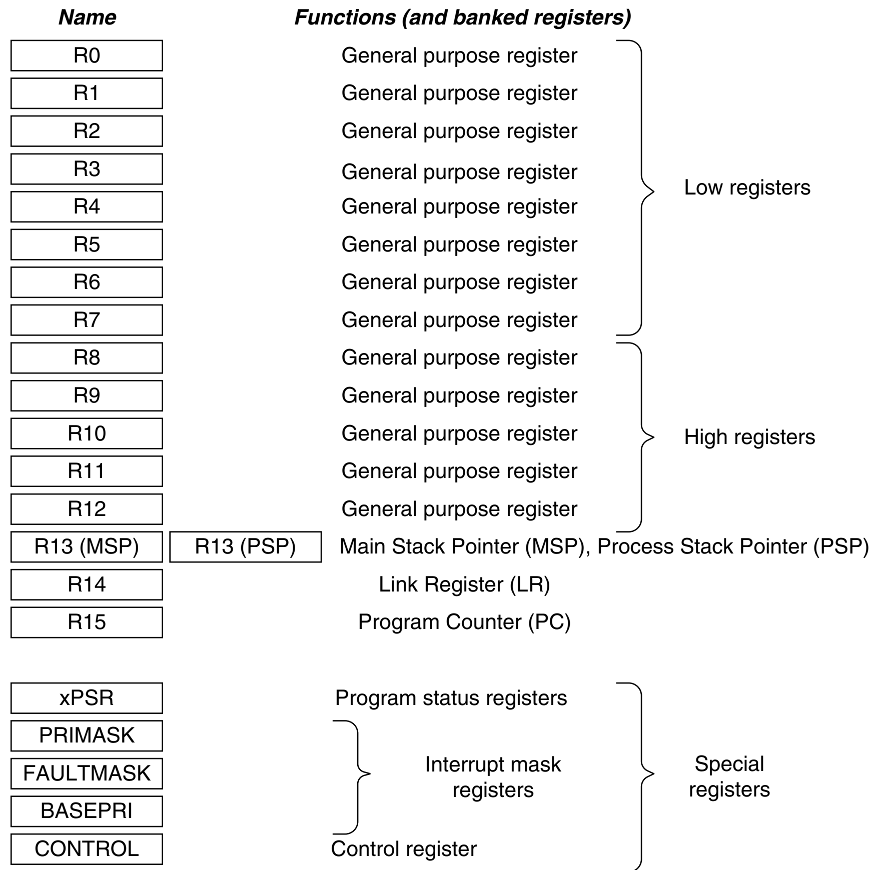


FIGURE 3.1

Registers in the Cortex-M3.

3.1.3 Stack Pointer R13

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

- *Main Stack Pointer (MSP) or SP_main in ARM documentation*: This is the default SP; it is used by the operating system (OS) kernel, exception handlers, and all application codes that require privileged access.
- *Process Stack Pointer (PSP) or SP_process in ARM documentation*: This is used by the base-level application code (when not running an exception handler).

https://hemanthrajhemu.github.io

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.

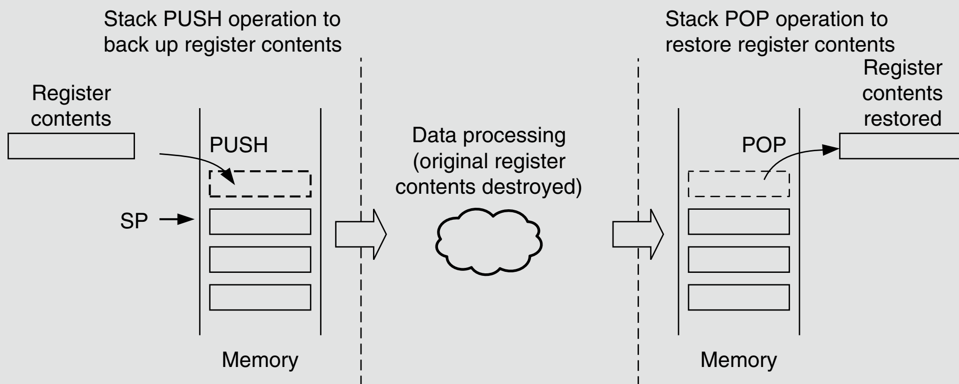


FIGURE 3.2

Basic Concept of Stack Memory.

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

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

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 SP 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
```

https://hemanthrajhemu.github.io

Instead of using *R13,* you can use *SP* (for SP) 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 SP 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 in system with embedded OS running.

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

3.1.4 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 (PC) when a subroutine or function is called—for example, when you're using the branch and link (BL) 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 PC 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 technology, this least significant bit (LSB) is writable and readable.

3.1.5 Program Counter R15

R15 is the PC. You can access it in assembler code by either R15 or PC. Because of 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, normally by 4. For example:

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

In other instructions like literal load (reading of a memory location related to current PC value), the effective value of PC might not be instruction address plus 4 due to alignment in address calculation. But the PC value is still at least 2 bytes ahead of the instruction address during execution.

Writing to the PC will cause a branch (but LRs do not get updated). Because an instruction address must be half word aligned, the LSB (bit 0) of the PC 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.

https://hemanthrajhemu.github.io

3.2 SPECIAL REGISTERS

The special registers in the Cortex-M3 processor include the following (see Figures 3.3 and 3.4):

- 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
```

3.2.1 Program Status Registers

The PSRs are subdivided into three status registers:

- Application Program Status register (APSR)
- Interrupt Program Status register (IPSR)
- Execution Program Status register (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.

You can read the PSRs 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
```

	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.

https://hemanthrajhemu.github.io

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

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
ARM (general)	N	Z	C	V	Q	IT	J	Reserved	GE[3:0]	IT	E	A	I	F	T	M[4:0]
ARM7 TDMI	N	Z	C	V	Reserved								I	F	T	M[4:0]

FIGURE 3.5

Current Program Status Registers in Traditional ARM Processors.

In ARM assembler, when accessing xPSR (all three PSRs 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.

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 ARM7. 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.

3.2.2 PRIMASK, FAULTMASK, and BASEPRI Registers

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

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.

https://hemanthrajhemu.github.io

Table 3.2 Cortex-M3 Interrupt Mask Registers

Register Name	Description
PRIMASK	A 1-bit register, when this is set, it allows nonmaskable interrupt (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 8 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).

To access the PRIMASK, FAULTMASK, and BASEPRI registers, a number of functions are available in the device driver libraries provided by the microcontroller vendors. For example, the following:

```
x = __get_BASEPRI(); // Read BASEPRI register
x = __get_PRIMARK(); // Read PRIMASK register
x = __get_FAULTMASK(); // Read FAULTMASK register
__set_BASEPRI(x); // Set new value for BASEPRI
__set_PRIMASK(x); // Set new value for PRIMASK
__set_FAULTMASK(x); // Set new value for FAULTMASK
__disable_irq(); // Clear PRIMASK, enable IRQ
__enable_irq(); // Set PRIMASK, disable IRQ
```

Details of these core register access functions are covered in Appendix G. A detailed introduction of Cortex Microcontroller Software Interface Standard (CMSIS) can be found in Chapter 10.

In assembly language, 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.

3.2.3 The Control Register

The control register is used to define the privilege level and the SP selection. This register has 2 bits, as shown in Table 3.3.

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.

https://hemanthrajhemu.github.io

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 0 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.

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 CONTROL[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 in C, the following CMSIS functions are available in CMSIS compliant device driver libraries:

```
x = __get_CONTROL(); // Read the current value of CONTROL
__set_CONTROL(x); // Set the CONTROL value to x
```

To access the control register in assembly, the MRS and MSR instructions are used:

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

3.3 OPERATION MODE

The Cortex-M3 processor supports two modes and two privilege levels (see Figure 3.6).

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 (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

https://hemanthrajhemu.github.io

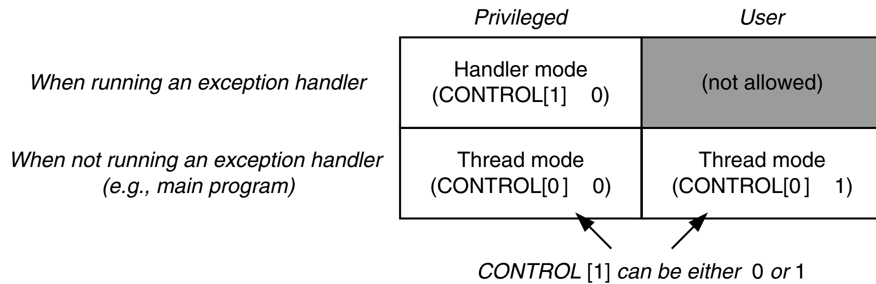


FIGURE 3.6

Operation Modes and Privilege Levels in Cortex-M3.

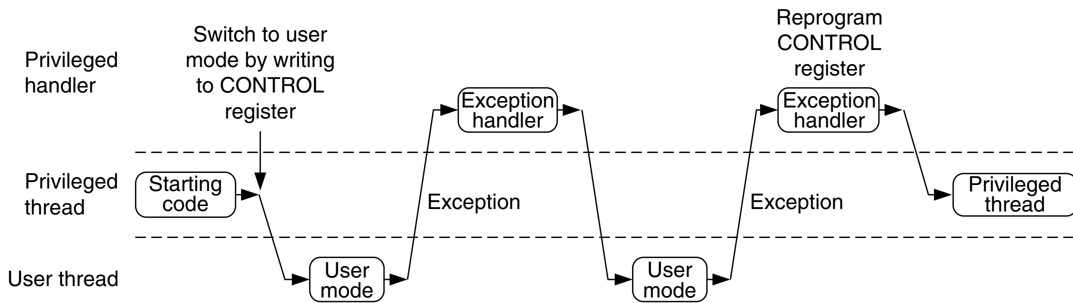


FIGURE 3.7

Switching of Operation Mode by Programming the Control Register or by Exceptions.

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. (See Figures 3.7).

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 Nested Vectored Interrupt Controller (NVIC). In addition, if the Memory Protection Unit (MPU) is present, it is possible to block user programs from accessing memory regions used by privileged processes.

In simple applications, there is no need to separate the privileged and user access levels. In these cases, there is no need to use user access level and no need to program the control register.

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 SPs is automatic upon entering or leaving the exception handlers (see section 3.6.3). This topic is discussed in more detail in Chapter 8.

The mode and access level of the processor are defined by the control register. When the control register bit 0 is 0, the processor mode changes when an exception takes place (see Figures 3.8 and 3.9).

https://hemanthrajhemu.github.io

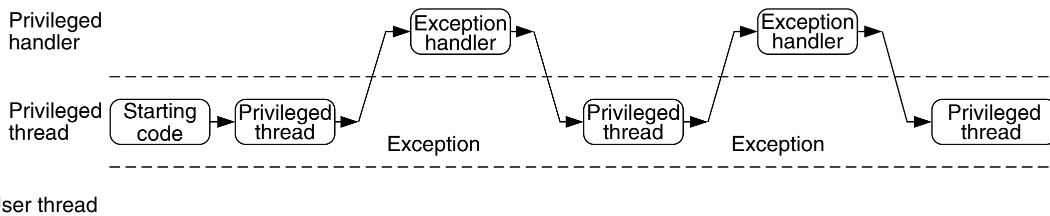


FIGURE 3.8

Simple Applications Do Not Require User Access Level in Thread Mode.

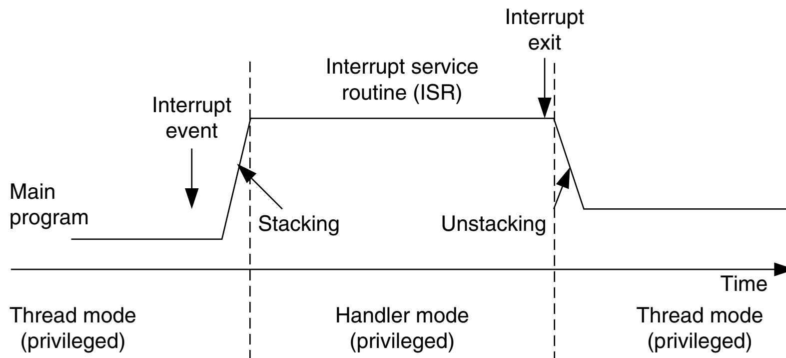


FIGURE 3.9

Switching Processor Mode at Interrupt.

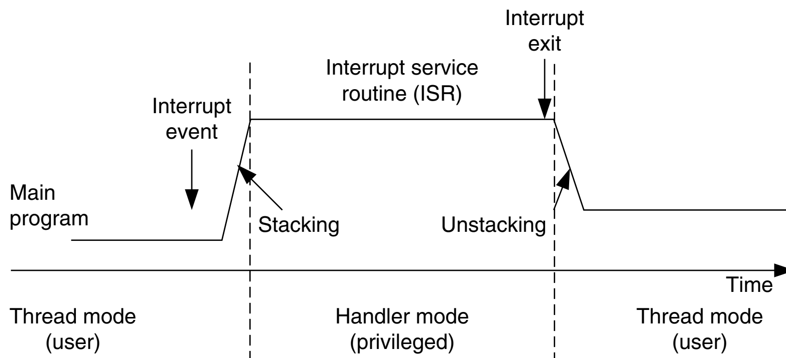


FIGURE 3.10

Switching Processor Mode and Privilege Level at Interrupt.

When control register bit 0 is 1 (thread running user application), both processor mode and access level change when an exception takes place (see Figure 3.10).

Control register bit 0 is programmable only in the privileged level (see Figure 2.5). For a user-level program to switch to privileged state, it has to raise an interrupt (for example, supervisor call [SVC]) and write to CONTROL[0] within the handler.

https://hemanthrajhemu.github.io

3.4 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 a nonmaskable interrupt (NMI) input signal. The actual use of NMI depends on the design of the microcontroller or system-on-chip (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 exception 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.

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

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	Bus fault	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	Supervisor call via SVC instruction
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

https://hemanthrajhemu.github.io

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

3.5 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 inside the system memory, 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 (see Table 3.5). After reset, this relocation control register is reset to 0; therefore, the vector table is located in address 0x0 after reset.

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 \times 4 = $ 0x00000008. The address 0x00000000 is used to store the starting value for the MSP.

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

3.6 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.)

https://hemanthrajhemu.github.io

3.6.1 Basic Operations of the Stack

In general, stack operations are memory write or read operations, with the address specified by an SP. Data in registers is saved into stack memory by a PUSH operation and can be restored to registers later by a POP operation. The SP 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 3.11). When PUSH/POP instructions are used, the SP is incremented/decremented automatically.

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 3.12).

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 3.13).

3.6.2 Cortex-M3 Stack Implementation

The Cortex-M3 uses a full-descending stack operation model. The SP points to the last data pushed to the stack memory, and the SP decrements before a new PUSH operation. See Figure 3.14 for an example showing execution of the instruction PUSH {R0}.

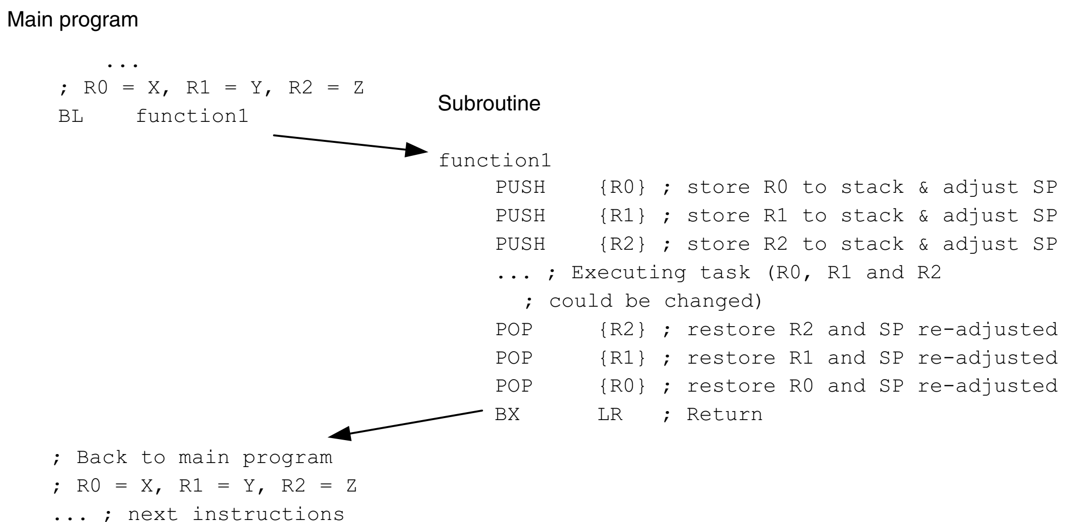


FIGURE 3.11

Stack Operation Basics: One Register in Each Stack Operation.

https://hemanthrajhemu.github.io

```
Main program
        ...
    ; R0 = X, R1 = Y, R2 = Z            Subroutine
    BL    function 1  ------>
                                        function 1
                                            PUSH    {R0-R2} ; Store R0, R1, R2 to stack
                                            ... ; Executing task (R0, R1 and R2
                                              ; could be changed)
                                            POP     {R0-R2} ; restore R0, R1, R2
                               <------      BX      LR   ; Return

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

FIGURE 3.12

Stack Operation Basics: Multiple Register Stack Operation.

```
Main program
        ...
    ; R0 = X, R1 = Y, R2 = Z            Subroutine
    BL    function 1  ------>
                                        function 1
                                            PUSH    {R0-R2, LR} ; Save registers
                                                        ; including link register
                                            ... ; Executing task (R0, R1 and R2
                                                ; could be changed)
                               <------      POP     {R0-R2, PC} ; Restore registers and
                                                        ; return

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

FIGURE 3.13

Stack Operation Basics: Combining Stack POP and RETURN.

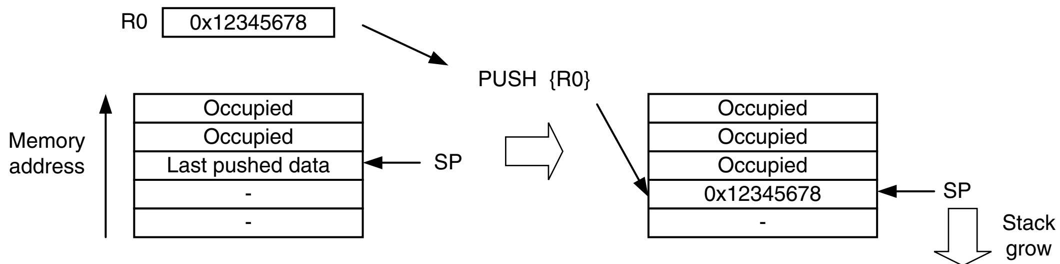


FIGURE 3.14

Cortex-M3 Stack PUSH Implementation.

https://hemanthrajhemu.github.io

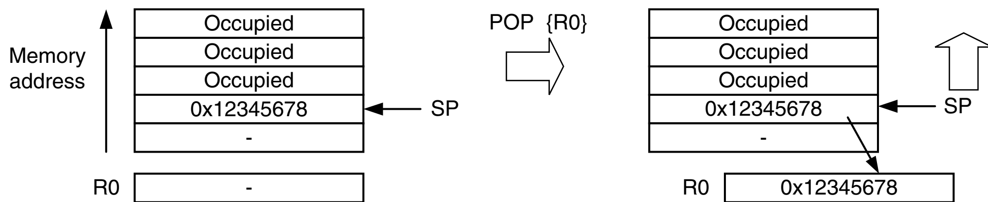


FIGURE 3.15

Cortex-M3 Stack POP Implementation.

For POP operations, the data is read from the memory location pointer by SP, and then, the SP is incremented. The contents in the memory location are unchanged but will be overwritten when the next PUSH operation takes place (see Figure 3.15).

Because 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 1 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 SP will also be adjusted.

3.6.3 The Two-Stack Model in the Cortex-M3

As mentioned before, the Cortex-M3 has two SPs: the MSPS and the 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 (see Figure 3.16). 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[1] is 1, the PSP is used in thread mode (see Figure 3.17). In this arrangement, the main program and the exception handler can have separate stack memory regions. This can prevent

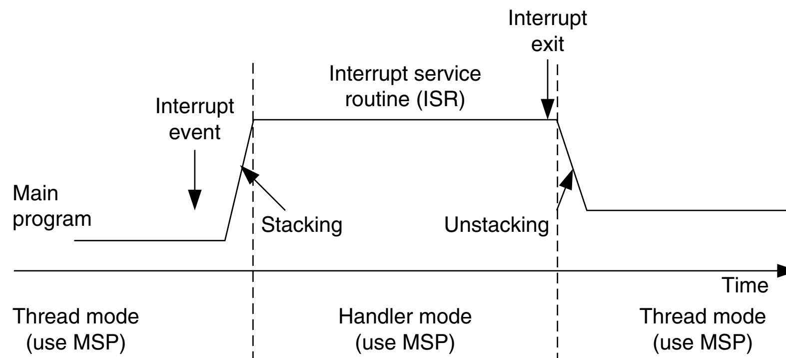


FIGURE 3.16

CONTROL[1] 0: Both Thread Level and Handler Use Main Stack.

https://hemanthrajhemu.github.io

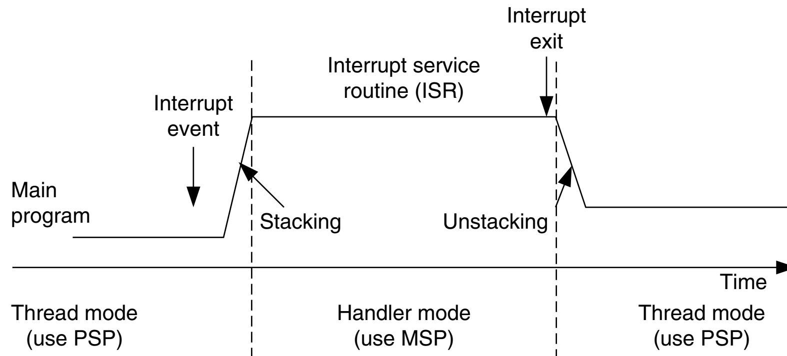


FIGURE 3.17

CONTROL[1]=1: Thread Level Uses Process Stack and Handler Uses Main Stack.

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).

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 values:

```
x = __get_MSP(); // Read the value of MSP
__set_MSP(x); // Set the value of MSP
x = __get_PSP(); // Read the value of PSP
__set_PSP(x); // Set the value of PSP
```

In general, it is not recommended to change current selected SP values in a C function, as the stack memory could be used for storing local variables. To access the SPs in assembly, you can use 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 SVC). In addition, the OS can change the PSP pointer value—for example, during context switching in multitasking systems.

3.7 RESET SEQUENCE

After the processor exits reset, it will read two words from memory (see Figure 3.18):

- Address 0x00000000: Starting value of R13 (the SP)
- Address 0x00000004: Reset vector (the starting address of program execution; LSB should be set to 1 to indicate Thumb state)

https://hemanthrajhemu.github.io

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

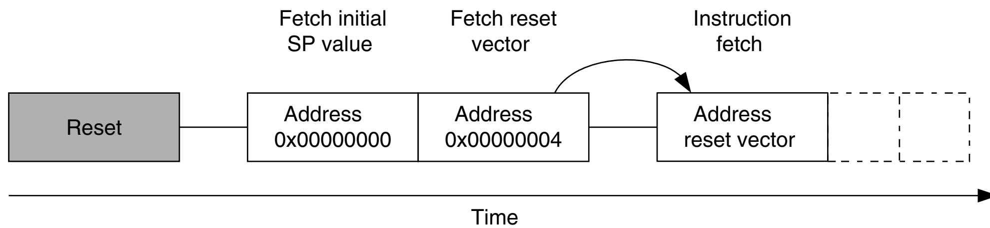


FIGURE 3.18

Reset Sequence.

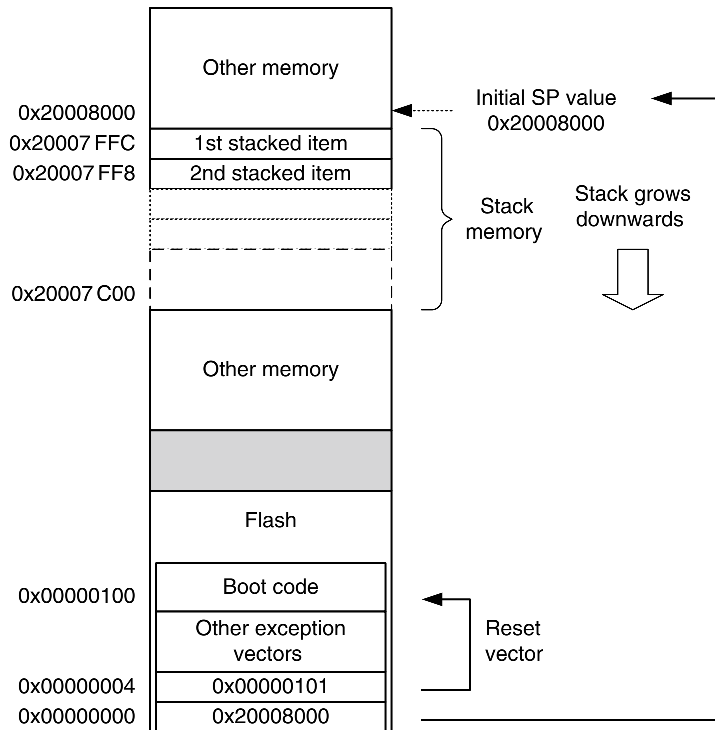


FIGURE 3.19

Initial Stack Pointer Value and Initial Program Counter Value Example.

https://hemanthrajhemu.github.io

not branch instructions. The first vector in the vector table (exception type 1) is the reset vector, which is the second piece of data fetched by the processor after reset.

Because the stack operation in the Cortex-M3 is a full descending stack (SP decrement before store), the initial SP 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 KB), 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 (see Figure 3.19). 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 SP 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 SP 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 Chapters 10 and 20 for ARM tools and in Chapter 19 for the GNU tool chain.

https://hemanthrajhemu.github.io