00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page i

Advanced Embedded Systems Concepts using the Renesas RX63N Microcontroller

BY JAMES M. CONRAD

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page ii

Micriμm Press
1290 Weston Road, Suite 306
Weston, FL 33326
USA

www.micrium.com

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where Micriμm Press is aware of a trademark claim, the product name appears in initial capital letters, in all capital letters, or in accordance with the vendor's capitalization preference. Readers should contact the appropriate companies for more complete information on trademarks and trademark registrations. All trademarks and registered trademarks in this book are the property of their respective holders.

Copyright © 2014 by James M. Conrad except where noted otherwise. Published by Micriμm Press. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher; with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

The programs and code examples in this book are presented for instructional value. The programs and examples have been carefully tested, but are not guaranteed to any particular purpose. The publisher and content contributors do not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors or omissions. The publisher and content contributors assume no liability for damages resulting from the use of the information in this book or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

Library of Congress subject headings:

1. Embedded computer systems
2. Real-time data processing
3. Computer software—Development

For bulk orders, please contact Micriμm Press at: +1 954 217 2036

ISBN: 978-1-935772-95-8

Please report errors or forward any comments and suggestions to jmconrad@uncc.edu.

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page iii

Preface

This book is the result of a long relationship the author has enjoyed with Renesas Electronics America, Inc. (and one of its predecessors, Mitsubishi Electronics). I originally worked with this company because of their commitment to providing a low-cost evaluation board and free development software that students could purchase and use in classes and senior design projects. Over the years the boards have remained as affordable (and popular) as ever, and the software development tools available have added more functionality while still available for free to our students.

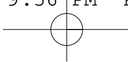
I have been teaching embedded systems courses for over fourteen years (and working in the field even longer). I had not been able to find a book suitable for using in an undergraduate course that would lend itself to the theoretical and applied nature of embedded systems design. Renesas had been asking us to create a book for several years, and the introduction of the new RX62N microcontroller offered a wonderful opportunity to work with this powerful device and integrate it into my classes. An update to the original RX62N book was made to take advantage of additional features of the RX63N processor. A book covering the advanced features of the RX63N was requested by popular demand - and this book is the result.

This book also has a radical feature not seen in many books currently on the market (if any). It is freely available for download and is also provided with the Renesas RX63N evaluation board. It is also available for purchase in hardcopy form for a modest price.

This book can be used on its own for an Advanced Microprocessors/ Microcontrollers/ Embedded Systems class or it can be used as a supplement in many different types of classes.

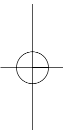
This book would not have been possible had it not been for the assistance of numerous people. Several students and educators contributed to and extensively tested some of the chapters, including: Joseph Collins (1), Aditya Bahulekar (1), Jason Wright (2, 3), Sultana Alimi (2, 3, 4), Aswin Ramakrishnan (4), Sravankumar Kambam (4, 5), Shweta Gupte (5), Swapneel Chitale (6), Shashank Hebbale (7), Pratik Jadhav (7), Jeremy Sabo (8), Ruban Veeraragavan (8), Bhanu Patibandala (9), Vishwas Subramanian (9), Sameer Sondur (10), Sunil Gurram (10), Gopinath Shanmuga Sundaram (11), Akshatha Udayashankar (11), and Vamsi Alla (12). Stephanie Conrad heavily edited versions of the chapters. Thanks go to the publisher, Linda Foegen, and especially June Harris, Rob Dautel and Todd DeBoer of Renesas for their help in getting this book produced and published (and for their patience!). Many, many thanks go to the reviewers who offered valuable suggestions to make this book better, especially David Brown, Mitch Ferguson, Barry Williams, Jean LaBrosse, John Donovan, Anthony Harris, Anthony Canino, Nicholas Gillotte, John A. Onuska, Rick Pray, Mark Radley, David Thomas, and students from my UNC Charlotte Embedded Systems course.

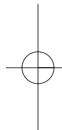
00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page iv

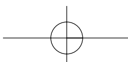


I would like to personally thank my parents, the Conrads, and my in-laws, the Warrens, for their continued assistance and guidance through the years while I worked on books. Also, I would especially like to thank my children, Jay, Mary Beth, and Caroline, and my wife Stephanie for their understanding when I needed to spend more time on the book than I spent with them.

James M. Conrad, March 2014







00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page v

Foreword

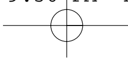
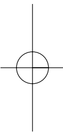
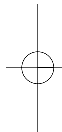
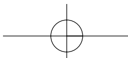
For more than a decade the microcontroller world has been dominated by the quest for ultra-low power, high performance devices—two goals that are typically mutually exclusive. The Renesas RX MCU quickly achieved market leadership by achieving both of these goals with a highly innovative architecture. The RX Family enables embedded designs that previously would have required some uncomfortable tradeoffs.

However there are no simple solutions to complex problems, and mastering all of the RX63N's advanced features is not a task to be undertaken lightly. Fortunately in this book Dr. Conrad, has crafted a guidebook for embedded developers that moves smoothly from concepts to coding in a manner that is neither too high level to be useful nor too detailed to be clear. It explains advanced software engineering techniques and shows how to implement them in RX63N-based applications, moving from a clear explanation of problems to techniques for solving them to line-by-line explanations of example code.

Modern embedded applications increasingly require hardware/software co-design, though few engineers are equally conversant with both of these disciplines. In this book the author takes a holistic approach to design, both explaining and demonstrating just how software needs to interact with RX63N hardware. Striking a balance between breadth and depth it should prove equally useful and illuminating for both hardware and software engineers.

Whether you are a university student honing your design skills, a design engineer looking for leading edge approaches to time-critical processes, or a manager attempting to further your risk management techniques, you will find Jim's approach to embedded systems to be stimulating and compelling.

Peter Carbone
Renesas
March, 2014

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page vii

Contents

Preface iii

Foreword v

CHAPTER 1

Renesas Assembly Language 1

1.1 Introduction 1
- **1.1.1** Introduction to Assembly Language 1
- **1.1.2** What We Will Learn 2

1.2 Basic Concepts of Data Storage and Use 2
- **1.2.1** RX63N Register Set 3
- **1.2.2** Data Types 4

1.3 Basic Concepts of Renesas Assembly Language 5
- **1.3.1** Addressing Modes 5
  - *Immediate* 6
  - *Register Direct* 6
  - *Register Indirect* 7
  - *Register Relative* 7
- **1.3.2** RX63N Instruction Set 8
  - *Data Transfer* 8
  - *Arithmetic and Logic* 9
  - *Floating-Point Operations* 11
  - *Control Transfer* 11
  - *Specialty Instructions* 12
- **1.3.3** Important Addresses in Memory 13
- **1.3.4** Basic Rules and Process for Writing Source Code 15
- **1.3.5** Inline Assembly 18

1.4 Basic Examples 19
- **1.4.1** Set Up Ports and Turn on LEDs 19
  - *C Function* 19
  - *Assembly Function* 20

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page viii

1.5 Recap 20
1.6 References 21
1.7 Exercises 21

CHAPTER 2
Function Calls and Stacks 23

2.1 Learning Objectives 23
2.2 Basic Concepts 23
2.2.1 Introduction to Function Calls and Stacks 23
2.2.2 Rules for Passing Arguments and Variable Declaration 23
2.2.3 Concept of Type Conversion in Function Calls 26
2.2.4 Stack Usage, Allocating, and Deallocating Stack Frames 28
2.3 Basic Example 29
2.4 Advanced Concepts 32
2.4.1 Memory Mapping 32
2.5 Advanced Example 34
2.6 Recap 40
2.7 References 40
2.8 Exercises 41

CHAPTER 3
Floating Point Unit and Operations 43

3.1 Learning Objectives 43
3.2 Basic Concepts 43
3.2.1 Floating Point Basics 43
Floating Point Representation 43
IEEE 754 Floating Point Standard 44
The IEEE 754 Floating Point Standard 45

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page ix

Single Precision (32 bits) 45
Double Precision (64 bits) 47
3.2.2 Pipelining Basics 49
Instruction Fetch 49
Instruction Decode 50
Execution 50
Memory Access 50
Write-Back 50
3.2.3 Floating Point in RX63N 50
Floating Point Unit 51
Floating Point Registers 51
Floating Point Instructions 54
FADD 54
FSUB 54
FCMP 55
Floating Point Exceptions 55
Overflow 55
Underflow 56
Inexact 56
Divide-By-Zero 56
Invalid Operation 56
3.3 Basic Examples 56
3.3.1 Operations Explaining Floating Point Exceptions 56
3.4 Advanced Concepts of RX63N Floating Point Unit 60
3.4.1 FPSW in Detail 60
Rounding-Modes 61
Cause Flags 61
Exception Flags 63
Exception Handling Enable Bits 63
Denormalized Number Bit 63
Floating Point Error Summary Flag 63
Reserved Bits 63

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page x

3.4.2 Floating Point Exception Handling Procedure 63
Acceptance of the Exception 63
Hardware Pre-Processing 64
Processing of User-Written Program Code 64
Hardware Post-Processing 66
3.5 Advanced Examples 66
3.5.1 Fixed-Point and Floating-Point Operation Time Calculation 66
*Int*69
Float 69
Double 69
3.5.2 Matrix Multiplication Time Calculation 70
*Int*74
Float 74
Double 74
3.6 Recap 74
3.7 References 75
3.8 Exercises 75

CHAPTER 4
Advanced Operating System Usage 77

4.1 Learning Objectives 77
4.2 Basic Concepts of Operating Systems 77
4.2.1 Multitasking, Re-entrant Functions 77
Multitasking 77
4.2.2 Semaphores 78
4.2.3 Task Communication, Synchronization, and Memory Management 79
4.3 Basic Examples 79
4.3.1 Example 1 79
4.3.2 Example 2 80

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xi

4.4 Advanced Concepts of Operating System Usage 80
4.4.1 Getting Started 80
4.4.2 Task Management 82
Task States 82
Task Creation 82
Task Priorities 83
Deleting a Task 83
Task Scheduling 84
4.4.3 Queue Management 84
Queue Reads 84
Queue Writes 85
Creating a Queue 85
Compound Types 86
4.4.4 Interrupt Management 86
Binary Semaphores for Synchronization 87
4.5 Complex Examples 88
4.5.1 Task Management 88
4.5.2 Queue Management 89
4.5.3 Interrupt Management 90
4.6 References 91
4.7 Exercises 92

CHAPTER 5

Digital Signal Processing 93

5.1 Learning Objectives 93
5.2 Basic Concepts of Digital Signal Processing 93
5.3 Basic Concepts of RX DSP Library 93
5.3.1 DSP Library Kernels 94
5.3.2 Data Types Supported and Data Structure 95
Data Types 95
Data Structures 95
Complex Data 95

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xii

Vector and Matrices 96
Kernel Handles 96
Floating Point Exceptions 97
5.3.3 Function Naming Convention and Arguments 97
Function Arguments 98
5.4 Basic Examples 98
5.4.1 Finite Impulse Response (FIR) Filter 98
Description 99
FIR Data Structure 99
Declaration 100
Main Function 101
Sample_dsp_fir() 102
5.4.2 Matrix Multiplication 103
Function Call Format 103
Description 105
Explanation 106
5.4.3 Fast Fourier Transform (FFT) 107
5.5 Recap 109
5.6 References 109
5.7 Exercise 110

CHAPTER 6

Direct Memory Access Controller 111

6.1 Learning Objectives 111
6.2 Basic Concepts 111
Important DMAC Registers 114
Modes of Operation 119
Normal Transfer Mode 119
Repeat Transfer Mode 120
Block Transfer Mode 122

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xiii

6.3 Basic Examples 123
Internal Data Transfer in Normal Transfer Mode 123
Internal Data Transfer in Repeat Transfer Mode 125
Internal Data Transfer in Block Transfer Mode 126
6.4 Advanced Concepts 126
Cluster Transfer Mode 128
6.5 Advanced Examples 131
External Data Transfer in Normal Transfer Mode 131
External Data Transfer in Repeat Transfer Mode 131
External Data Transfer in Block Transfer Mode 132
External Data Transfer in Cluster Transfer Mode 133
6.6 Examples with Interrupts 135
6.7 Recap 137
6.8 References 138
6.8 Exercises 138

CHAPTER 7

Flash and EEPROM Programming 139

7.1 Learning Objectives 139
7.2 Basic Concepts 139
7.2.1 Flash Memory Overview 139
7.2.2 Operating Modes Associated with Flash Memory 141
7.2.3 Block Configuration of the ROM 142
7.2.4 Block Configuration of the E2 DataFlash 144
7.2.5 FCU 145
FCU Modes 145
FCU Commands 147
7.2.6 FCU Register Descriptions 148
Flash Write Erase Protection Register (FWEPROR) 148
Flash Mode Register (FMODR) 149

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xiv

Flash Access Status Register (FASTAT) 149
Flash Ready Interrupt Enable Register (FRDYIE) 150
E2 DataFlash Read Enable Register 0 (DFLRE0) 151
E2 DataFlash Read Enable Register 1 (DFLRE1) 152
E2 DataFlash P/E Enable Register 0 (DFLWE0) 153
E2 DataFlash P/E Enable Register 1 (DFLWE1) 154
FCU RAM Enable Register (FCURAME) 155
Flash Status Register 0 (FSTATR0) 155
Flash Status Register 1 (FSTATR1) 157
Flash P/E Mode Entry Register (FENTRYR) 158
Flash Protection Register (FPROTR) 159
Flash Reset Register (FRESETR) 160
FCU Command Register (FCMDR) 161
FCU Processing Switching Register (FCPSR) 161
E2 DataFlash Blank Check Control Register (DFLBCCNT) 162
Flash P/E Status Register (FPESTAT) 162
E2 DataFlash Blank Check Status Register (DFLBCSTAT) 163
Peripheral Clock Notification Register (PCKAR) 163
7.2.7 Mode Transitions 164
7.2.8 Programming and Erasure Procedures 165
7.2.9 Programming 166
7.2.10 Erasure 166
7.2.11 Simple Flash API for RX63N 167
Features 167
7.3 Basic Examples 168
7.4 Advanced Concepts 170
7.4.1 Virtual EEPROM for RX63N 170
Records 172
Data Management 172
VEE in Action 172
Assigning VEE Records to VEE Sectors 173
Allocating VEE Blocks 174

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xv

Data Structure that Holds VEE Project Data Configuration 175
VEE Record 176
7.4.2 Protection 176
Software Protection 176
Command-Locked State 177
7.4.3 User Boot Mode 178
Boot Mode System Configuration 178
State Transitions in Boot Mode 179
Automatic Adjustment of the Bit Rate 180
ID Code Protection (Boot Mode) 181
Control Code 181
ID Code 182
Program Example for ID Code Setting 182
7.5 Complex Examples 183
7.6 Recap 185
7.7 References 185
7.8 Exercises 185

CHAPTER 8

Universal Serial Bus (USB) Connectivity 187

8.1 Learning Objectives 187
8.2 Basic Concepts of USB Connectivity 187
8.2.1 USB Interface Specifications 187
8.2.1.1 Cabling and Connectors 187
8.2.1.2 Electrical Specifications 189
8.2.2 Host and Devices 189
USB Host 190
USB Device 191
8.2.3 Transfers, Transactions, and Frames 191

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xvi

8.2.4 Class Drivers 192

8.2.4.1 Communication Device Class 192

8.2.4.2 Human Interface Device Class 194

8.2.4.3 Mass Storage Device Class 194

8.3 Basic Examples 197

8.3.1 Example 1: Detecting a Device 197

Format 197

Argument 197

Return Value 197

Description 197

8.3.2 Example 2: Ending Connection to a USB Device 197

Format 197

Argument 198

Return Value 198

Description 198

8.3.3 Example 3: Receiving 198

Format 198

Argument 198

Return Value 199

Description 199

8.4 Human Interface Device Driver Class 200

8.4.1 Overview 200

8.4.2 Reports 201

8.4.3 Architecture 201

8.4.4 More Examples of HID Driver Functions 201

Format 203

Argument 203

Return Value 203

Description 203

8.5 Recap 207

8.6 References 207

8.7 Exercises 207

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xvii

CHAPTER 9

RX63N Ethernet Controller **209**

9.1 Learning Objectives 209
9.2 Basic Concepts of Ethernet and Internet Protocol 209
9.2.1 Ethernet Network Topologies 209
9.2.2 Internet Protocol 212
9.2.3 Example 212
9.3 Ethernet Controller 213
9.4 Ethernet Direct Memory Access Controller 222
9.5 Renesas Ethernet Driver API 227
Format 227
Parameters 227
Return Values 227
Format 227
Parameters 227
Return Values 227
Format 228
Parameters 228
Return Values 228
Format 228
Parameters 228
Return Values 229
9.5.1 Example 1: Transmitting Ethernet Frames 229
9.5.2 Example 2: Receiving Ethernet Frames 230
9.6 Recap 231
9.7 References 231
9.8 Exercises 232

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xviii

CHAPTER 10

CAN Bus 233

10.1 Learning Objectives 233

10.2 Theory of Can Protocol 233

10.2.2 CAN Bus Details 234

10.2.3 Different CAN Bus Standards 235

10.2.4 Types of Frames and Their Architectures 236

Data and Remote Frame 236

Error Frame 237

Overload Frame 238

10.2.5 Bus Arbitration 238

10.2.6 Message Broadcasting 239

10.2.7 Data Transfer Synchronization 240

10.2.8 Error Detection and Fault Confinement 240

10.2.9 Different CAN Bus Standards 242

10.3 Basic Concepts 242

10.3.1 Registers 244

Control Register (CTLR) 245

Bit Configuration Register (BCR) 245

Mask Register k (MKRk) (k = 0 to 7) 246

Mask Invalid Register (MKIVLR) 246

Mailbox Register j (MBj) (j = 0 to 31) 247

Mailbox Interrupt Enable Register (MIER) 248

Message Control Register j (MCTLj) (j = 0 to 31) 248

Status Register (STR) 249

10.3.2 Reception and Transmission 249

Reception 250

Transmission 252

10.3.3 Example 1: Initialization of CAN Bus 255

10.3.4 Example 2: Reception and Transmission 258

Reception 258

Explanation 258

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xix

Transmission 259
Explanation 259
10.3.5 Main Function 260
10.4 Advanced Concepts 262
10.4.1 CAN Reset Mode 262
10.4.2 CAN Halt Mode 263
10.4.3 CAN Sleep Mode 264
10.4.4 CAN Operation Mode 264
10.4.5 CAN Communication Speed Setting 265
CAN Clock Setting 265
Bit Timing Setting 266
Bit Rate 266
10.4.6 Acceptance Filtering and Masking Functions 266
10.4.7 CAN Interrupts 268
10.5 Complex Examples 269
Using Interrupts 269
10.6 Can Bus Application Programming Interface 270
10.7 Recap 273
10.8 References 274
10.9 Exercises 274

CHAPTER 11

Watchdog Timer and Brownout 275

11.1 Learning Objectives 275
11.2 Basic Concepts of Watchdog Timers 275
11.3 Watchdog Timer in RX63N 275
11.3.1 Register Description 276
11.4 Advanced Concepts of the Watchdog Timer 281
11.4.1 Register Start Mode 281
11.4.2 Auto-Start Mode 282
Control over Writing to the WDTCR and WDTRCR Registers 283

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xx

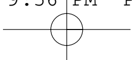
11.5 Independent Watchdog Timer (IWDT) 283
11.5.1 Register Description 285
11.6 Examples 290
11.7 Basic Concepts of Brownout Condition 291
11.7.1 How Brownout Occurs 291
11.7.2 Automatically Detecting a Brownout Condition 291
11.8 Recap 293
11.9 References 293
11.10 Exercises 293

CHAPTER 12

Processor Settings and Running in Low Power Modes 295

12.1 Learning Objectives 295
12.2 RX63N Startup Process 295
12.3 Basic Concepts of Low Power Consumption 300
12.3.1 Introduction to the Concept of Low Power Consumption 300
12.3.2 Various Processor Settings to Achieve Low Power Consumption 301
12.3.3 Overview: Various Low Power Consumption Modes 301
12.3.4 Processor Settings: Various Registers Used 301
12.3.5 Functions: Description of Operation in Different Functions 302
12.4 Basic Examples 311
12.4.1 Example 1: Setting the Multi Clock Function 311
12.4.2 Example 2: Setting the Module Clock Function 311
12.4.3 Example 3: Setting the SDCLK Output Control Function 312
12.5 Advanced Concepts of Low Power Consumption 313
12.5.1 Sleep Mode 313
12.5.2 All-Module Clock Stop Mode 313
12.5.3 Software Standby Mode 315
12.5.4 Deep Software Standby Mode 315

00.ES_Conrad_RX63N_Advanced_FM.qxd:Conrad 3/4/14 9:56 PM Page xxi



12.6 Advanced Concept Examples 316

12.6.1 Example 1: Sleep Mode 316

12.6.2 Example 2: Software Standby Mode 318

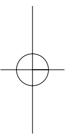
12.6.3 Example 3: Deep Software Standby Mode 320

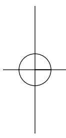
12.7 Recap 323

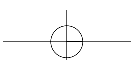
12.8 References 323

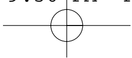
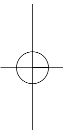
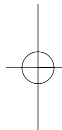
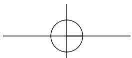
12.9 Exercises 323

Index 325







01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 1

Renesas Assembly Language

1.1 INTRODUCTION

1.1.1 Introduction to Assembly Language

The RX63N microcontroller executes a single instruction for each clock cycle [1, p.17]. These instructions are 1 to 8 bytes long and are stored in program memory in the native language: machine code. The processor fetches each of these instructions from program memory and executes them to perform the tasks that we require of it.

When HEW or E^{2}Studio is used to develop C code for the Renesas RX63N, the compiler converts each C instruction into the machine language equivalent. For some instructions, the conversion ratio is 1:1 (that is, one C instruction equates to a single machine language instruction); however, some high-level C instructions are converted into a larger set of machine instructions. This conversion saves the user time and allows coding to be more intuitive. In some situations, however, the compiler does not adequately optimize the conversion of C code to machine code; therefore, writing our own machine code equivalent can solve this optimization problem. For example, consider a task that runs once and isn't time sensitive. Writing low-level machine code can be time consuming and unnecessary. But now consider a task that will be executed often and must be completed in a shorter amount of time than the compiler optimizations allow. A block of custom machine code could be very useful in this case. In addition to possibly speeding up processes, using machine code also allows the user to make more decisions about how the program is executed, such as choosing which registers will be used.

Writing machine language in 1's and 0's is cumbersome, and debugging such a program would be prohibitively difficult. Luckily, there's a programming language in which each instruction corresponds to a single machine language instruction: assembly language. Since this is a 1:1 conversion, programming in assembly is essentially the same as programming in machine language.

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 2

1.1.2 What We Will Learn

Every processor and microcontroller has a specific instruction set defined in the datasheet and/or software manuals that describes the behavior of each assembly instruction. In this chapter, we look at those instructions and learn how to use them in HEW. Assembly language is a subject that merits several college-level lectures in itself, so the following chapter focuses on specifics related to the RX63N only. Students should supplement their reading with additional books about assembly language for a more comprehensive understanding.

1.2 BASIC CONCEPTS OF DATA STORAGE AND USE

The fundamental functions of a computer are as follows:

- Data processing
- Data storage
- Data movement
- Data control

This chapter discusses how the microcontroller manipulates and moves data around using assembly language instructions.

Viewing the memory features of the RX63N before proceeding to memory allocation in the MCU is important. According to the Renesas RX63N Hardware Manual, the following are the memory specifications:

Memory	ROM	Capacity: ROMless, 256 Kbytes, 384 Kbytes, 512 Kbytes, 768 Kbytes, 1 Mbyte, 1.5 Mbytes, 2 Mbytes
		100 MHz, no-wait access
		On-board programming: Four types
		Off-board programming (parallel programmer mode) (for products with 100 pins or more)
	RAM	Capacity: 64 Kbytes, 128 Kbytes, 192 Kbytes, 256 Kbytes
		100 MHz, no-wait access
	E2 DataFlash	Capacity: 32 Kbytes
		Programming/erasing: 100,000 times

Figure 1.1 Memory features of the RX63N [1], page 51.

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 3

1.2.1 RX63N Register Set

The RX63N has sixteen general-purpose registers, nine control registers, and one accumulator for Digital Signal Processing (DSP) instructions. All register definitions can be found in the RX63N User's Manual: Software [1]. This chapter discusses how the data is arranged in the registers.

General-purpose register

b31 b0

R0 (SP)*1
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15

Control register

b31 b0

ISP (Interrupt stack pointer)
USP (User stack pointer)
INTB (Interrupt table register)
PC (Progam counter)
PSW (Process status word)
BPC (Backup PC)
BPSW (Backup PSW)
FINTV (Fast interrupt vector register)
FPSW (Floating point status word)

DSP instruction register

b63 b0

ACC (Accumulator)

Note 1. The stack pointer (SP) can be the interrupt stack pointer (ISP) or the user stack pointer (USP), according to the value of the U bit in the PSW.

Figure 1.2 Register set of the CPU [2], page 117.

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 4

1.2.2 Data Types

The RX63N supports integer, floating point, bitwise, and string data types.

1. **Integer:**
   The integer (int) is the most common data type. Integers are commonly used not only for arithmetic operations, but also as flags and counters. Figure 1.3 describes the different data lengths of the integer data type that the RX Family can handle.

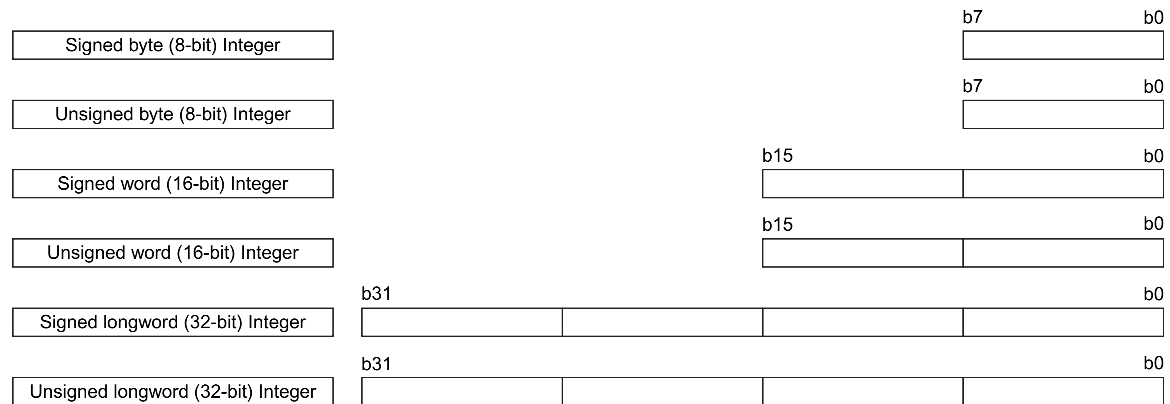


Figure 1.3 Integer length representations [1], page 31.

2. **Floating-Point:**
   IEEE defines four types of precision for floating point operations: single precision, double precision, single-extended precision, and double-extended precision. The RX Family only supports single precision (32 bits) floating point computations. The RX operations used with floating point operands include FADD, FCMP, FDIV, FMUL, FSUB, FTOI, ITOF, and ROUND. The applications of these operations can be found in the Renesas User's Manual: Software [1]. Floating point numbers and operations are described in more detail in Chapter 4.

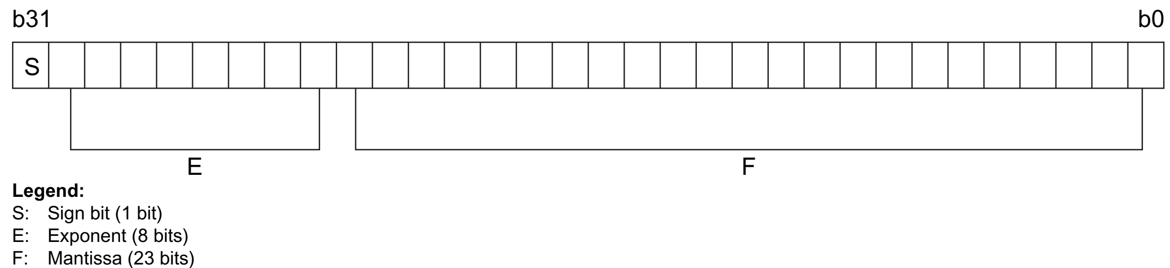


Figure 1.4 Single precision floating point representation.

3. **Bitwise:**

For a bitwise operation to take place, a number taken in its binary form is manipulated bit by bit rather than the entire bit stream at once. For example, binary operations would include taking a logical AND, OR, or XOR of two binary numbers.

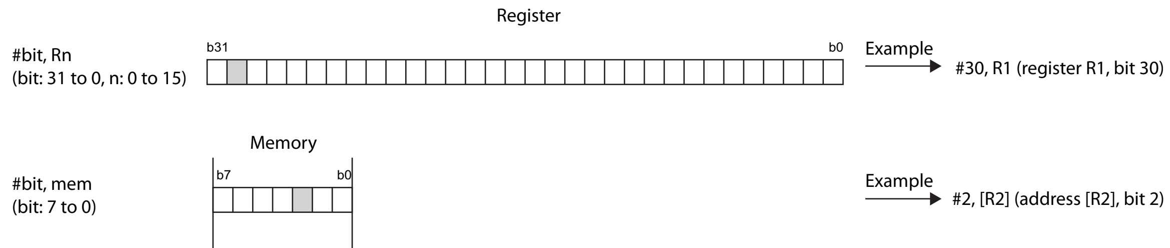


Figure 1.5 Bitwise Operation [1], page 32.

4. **Strings:**

The strings data type consists of a sequence of characters often used to represent text, spaces, and symbols such as punctuation. Because of this structure, a high number of bits for strings are needed. Strings may be consecutive byte (8 bit), word (16 bit), or longword (32 bit) units. The RX Family provides several string manipulation instructions.

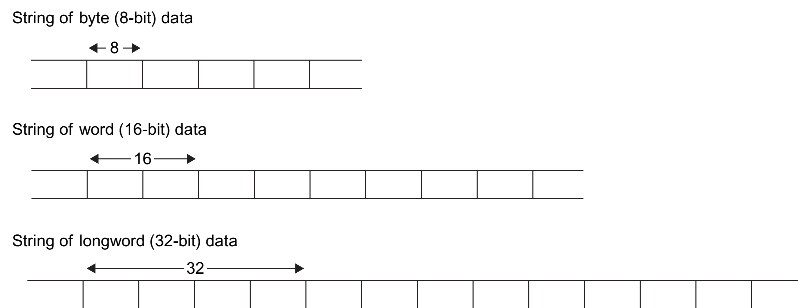


Figure 1.6 String data sizes [1], page 32.

1.3 BASIC CONCEPTS OF RENESAS ASSEMBLY LANGUAGE

In this section, we introduce assembly language and the basic concepts required to use it effectively.

1.3.1 Addressing Modes

The RX63N has a total of 10 addressing modes that define how the syntax of assembly instructions are read and interpreted by the compiler. These modes can all be found in the

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 6

Renesas RX63N Group User's Manual: Software [1]. The manual also includes the specifics on each addressing mode. In this section, we learn how to use four of the most common addressing modes.

Immediate

The immediate addressing mode includes the instruction, immediately available data in an operand, and a destination location, and has the general form:

```
INSTRUCTION   OPERAND, DESTINATION
```

Let's say we would like to store a value to a register, and this value is known to the programmer. For example, the value could be the start of a loop counter, 0x255, and we would like to store this value in the register R5. The value is called immediate, since it will be stored immediately into R5 and does not require retrieving the value from another memory location. Following is an example of the format for this instruction:

```
MOV.B   #255H, R5
```

In this case, MOV.B is the instruction, #225H is the operand, and R5 is the destination.

This same format can be used with other instructions, such as:

- ADD
- AND
- BCLR
- BNOT
- BSET
- BTST
- CMP
- MOV
- MUL
- OR
- SUB

Register Direct

The register direct addressing mode moves data from one memory location to another. We can again use an example in which we are trying to load a value into register R5. This time, let's assume that the value is located in register R1. If we don't know what the value of R1 is, then we can't use an immediate addressing mode. Instead, we can say:

```
MOV.B   R1, R5
```

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 7

which is of the general form:

```
INSTRUCTION   SOURCE, DESTINATION
```

While this is most often used with a MOV instruction, it is also the addressing mode for JMP and JSR instructions in which the value of R*n* (the source) is transferred to the program counter (PC). Also note that the source can be a memory location specified in hexadecimal format, not just a register.

Register Indirect

The register indirect addressing mode is similar to register direct, except that the operand contains the address of the data that will be stored in the destination. For example, if we know that the memory location of 08C00Dh is stored in R1, then at any time we can move the data from 08C00Dh to a new location:

```
MOV   [R1], R5
```

The value located at 08C00Dh is now stored in R5. Note that the form for this addressing mode is as follows:

```
INSTRUCTION   [SOURCE ADDRESS], DESTINATION
```

Register Relative

Register Relative addressing is similar to register indirect addressing. This addressing mode uses a familiar format:

```
INSTRUCTION   offset[SOURCE ADDRESS], DESTINATION
```

The source here, however, is a relative address. A relative address is a memory location that is specified by its proximity relation to another address. This addressing mode uses the following format:

```
offset[SOURCE ADDRESS]
```

The source address for offset can be a positive or negative number. The address can be any valid register. For example, if the source for data is memory address 0x000005, and the memory address 0x000000 is already stored in register R1 from an earlier instruction (`"MOV   #000000H, R1"`), then we can read from memory location 0x000005 using:

```
1. MOV   5[R1], R5
```

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 8

After executing this instruction, R5 now holds the data copied from memory location 0x000005.

1.3.2 RX63N Instruction Set

The Renesas RX63N has 90 instructions available to the programmer, consisting of 73 basic instructions, eight floating point instructions, and nine digital signal processing instructions. All programs, even if written in a higher level language like C, are composed of a specific combination of these 90 assembly language instructions. A detailed description of each instruction can be found be found in the Renesas RX63N Group User's Manual: Software [1]. This chapter explores some basic instructions. These basic instructions are similar in construct to others not described here.

The five essential instruction types are:

1. Data Transfer
2. Arithmetic and Logic
3. Floating-Point Operations
4. Control Transfer
5. Specialty and Other

Data Transfer

Following are some common data transfer instructions:

- MOV: Transfer data
- MOVU: Transfer unsigned data
- POP: Restore data from the stack
- PUSH: Save data on the stack
- POPM: Restore multiple registers from the stack
- PUSHM: Saving multiple registers
- STZ: Transfer with condition
- STNZ: Transfer with condition

The most common instruction in most programs is MOV and MOVU (unsigned). These instructions are used to store immediate data, and to copy data from one memory location to another. It is used with a length modifier of .B, .W, or .L, as discussed in Section 1.2.4. When the programmer wants to store a known value, the immediate addressing can be used as follows:

```
MOV   #255H, R5
```

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 9

Following is an example of moving data from one register to another using the register direct addressing mode:

```
MOV    R1, R2
```

In the following example data is moved from one memory location into a register (R5) using Register Indirect addressing mode. Note that in this example that the memory address where the data is read is 0x000001H.

```
MOV    #000001H, R1
MOV    [R1], R5
```

Arithmetic and Logic

Following are common arithmetic instructions:

- ADD: Addition without carry
- ADC: Addition with carry
- SUB: Subtract without borrow
- SBB: Subtract with borrow
- MUL: Multiplication
- DIV: Signed division
- DIVU: Unsigned division
- ABS: Absolute value
- INC: Increment
- DEC: Decrement
- CMP: Comparison

Simple arithmetic instructions share a common addressing mode. The following examples demonstrate arithmetic instructions.

Add R5 to R1 and store result in R1:

```
ADD    R5, R1
```

Subtract R2 from R3 and place the result in R3:

```
SUB    R2, R3
```

Multiply R6 from R4 and place the result in R4:

```
MUL    R6, R4
```

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 10

Divide R1 from R2 and place result in R1:

```
DIV    R2, R1
```

Compare the values of R3 and R4. Note that the result will be stored by a change in a flag state in the PSW register:

```
CMP    R3, R4
```

The following are some common logic and bit manipulation instructions:

- AND: Logical AND
- NOT: Logical complementation
- OR: Logical OR
- XOR: Logical exclusive OR
- SHLL: Logical and arithmetic shift to the left
- SHLR: Logical shift to the right
- NEG: 2's compliment

Examples:

Perform the bitwise AND operation on R1 and R2 and store the result in R2 (overwriting). If R1 contains 0x00FF00FF andR2 contains 0x0000FFFF, the result of 0x000000FF would be stored in R2.

```
AND    R1, R2
```

Perform the same operation, but now store the result in R3, leaving R1 and R2 unchanged:

```
AND    R1, R2, R3
```

Perform an exclusive OR operation on R1 and R2 and place the result in R2:

```
XOR    R1, R2
```

Shift R4 to the right by 3 positions and store the result in R4 (overwriting):

```
MOV    #0003H, R1    ;store the value "3" in R1
SHLR   R1, R4        ;shift R4 by value in R1 (3)
```

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 11

Perform the same operation, but store result in R5, leaving R4 unchanged:

```
MOV   #0003H, R1   ;store the value "3" in R1
SHLR  R1, R4, R5
```

Take the 2's Compliment of the value of R5, storing the result in R5:

```
NEG   R5
```

Take the 2's Compliment of R5, but store the result in R6, leaving R5 unchanged:

```
NEG   R5, R6
```

Floating-Point Operations

Floating-point operation instructions are available for facilitating a program with more precise arithmetic operations. Note, however, that it is sometimes better to perform these functions with a combination of simple arithmetic and logic instructions, or simply an approximate with integers, in order to save processor time. This is especially true with floating-point division. The following are the RX63N's floating-point instructions:

- FADD: Floating-point addition
- FCMP: Floating-point comparison
- FDIV: Floating-point division
- FMUL: Floating-point multiplication
- FSUB: Floating-point subtraction
- FTOI: Floating-point to integer conversion
- ITOF: Integer to floating-point conversion
- ROUND: Conversion from floating-point to integer

The usage of floating-point operation instructions is similar to arithmetic instructions, using the following familiar format:

```
INSTRUCTION  SOURCE, SOURCE/DESTINATION
```

Control Transfer

The following are some common Control Transfer instructions:

- BRA: Unconditional relative branch
- BCnd: Relative conditional branch (many types of Cnd)

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 12

- BSR: Relative subroutine branch
- JMP: Unconditional jump
- JSR: Jump to a subroutine
- RTS: Return from a subroutine
- RTSD: Return from a subroutine and release stack frame

Examples:

Branch to a destination address specified by PC = PC + R1:

```
BRA    R1
```

Branch to a destination address specified by PC = R1:

```
JMP    R1
```

Return from a subroutine:

```
RTS
```

Specialty Instructions

No Operation: No Operation (NOP) is a unique instruction available for assembly language. It wastes one clock cycle without doing any useful work, but can be very useful for timing. For example, one use is looping a 50 MHz clock continuously every 0.00004 seconds (or at a rate of 25 kHz). This means that the loop needs to execute in exactly 2000 clock cycles. If the current loop takes only 1997 clock cycles, then add three wasted cycles with three NOP instructions at the end. This cycle would look like:

```
1. NOP
2. NOP
3. NOP
```

Another case where NOP can be useful is when waiting for another event to happen before continuing. For example, waiting for a signal to arrive from a peripheral, or waiting for an operation to finish exiting the processor's pipeline (some instructions take more than one clock cycle to execute).

String Manipulations: The following is a list of some common string manipulation instructions:

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 13

- SSTR: Store a string
- SCMPU: Compare a string
- SMOVB: Transfer a string backwards
- SMOVF: Transfer a string forwards

DSP: The Renesas RX63N has specialty DSP instructions, which are covered in Chapter 11. These instructions are:

- MACHI: Multiply-Accumulate the high-order word
- MACLO: Multiply-Accumulate the low-order word
- MULHI: Multiply the high-order word
- MULLO: Multiply the low-order word
- MVFACHI: Move the high-order longword from accumulator
- MVFACMI: Move the middle-order longword from accumulator
- MVTACHI: Move the high-order longword to accumulator
- MVTACLO: Move the low-order longword to accumulator
- RACW: Round the accumulator word

1.3.3 Important Addresses in Memory

High-level languages such as C do not often require the programmer to know specific memory addresses, since the compiler and header files can take care of that automatically. For example, a local variable in memory is automatically assigned an address in RAM by the compiler, while malloc() and calloc() are functions used to dynamically manipulate heap space during runtime. In assembly, however, the programmer has the opportunity to choose which registers and memory addresses are used. In this section, we cover the basic memory addresses for RAM/Flash, special function registers, and general purpose registers. The RX63N memory map is summarized on the following page.

Only a portion of the 4-GByte memory address range is useable for general purpose storage. Areas that are designated "reserved," "read only," or "write only" are not available for programmer use (except "write only," which is ultimately used to store machine code generated by the assembler). Variables must be stored in either RAM or on-chip ROM (also known as flash), depending on the type of variable. In general, constants are stored in ROM while data that is likely to change, including all local variables, are stored in RAM. Global variables can be stored in either RAM or ROM, depending on the application. Heap memory, allocated using malloc() and calloc() in C, is stored in RAM.

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 14

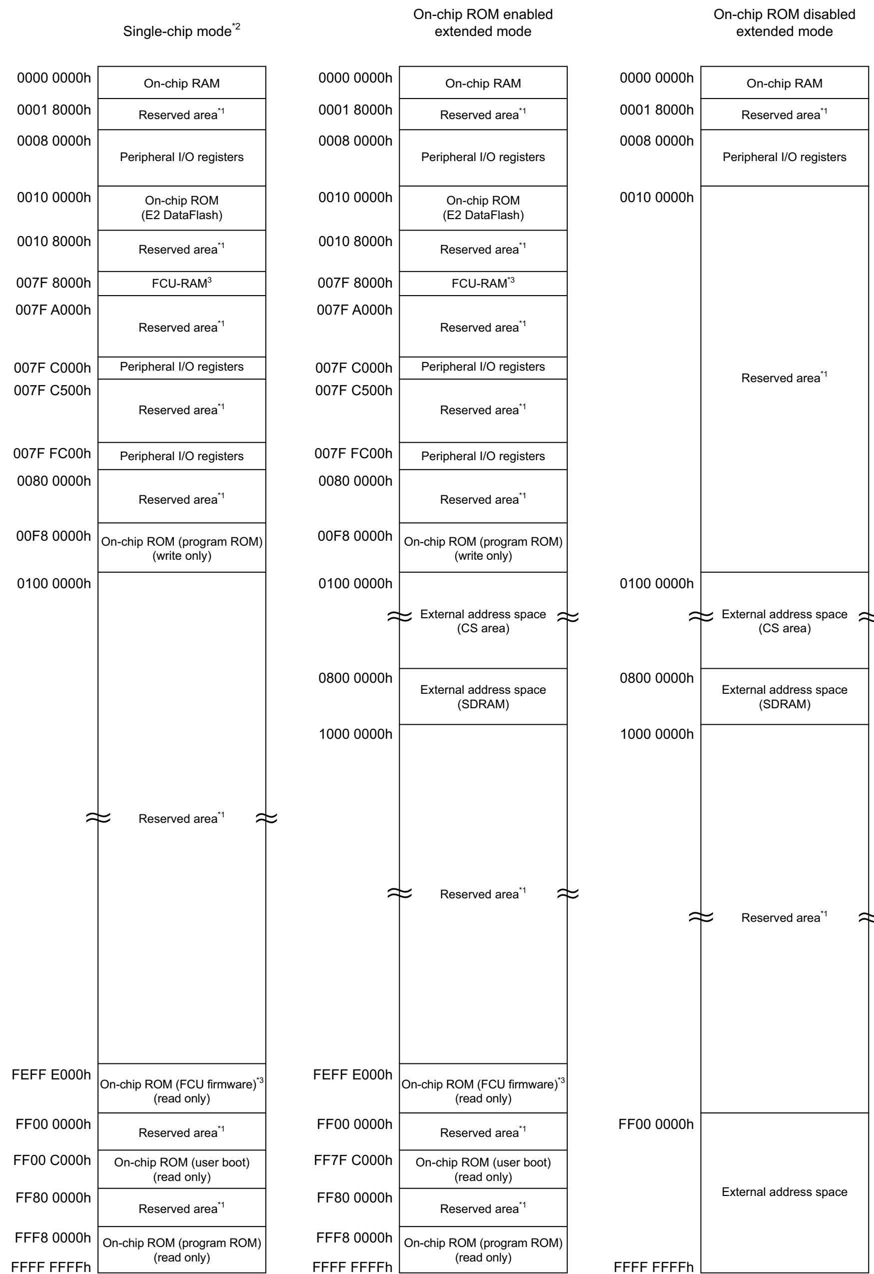


Figure 1.7 RX63N Memory Map [2], page 151.

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 15

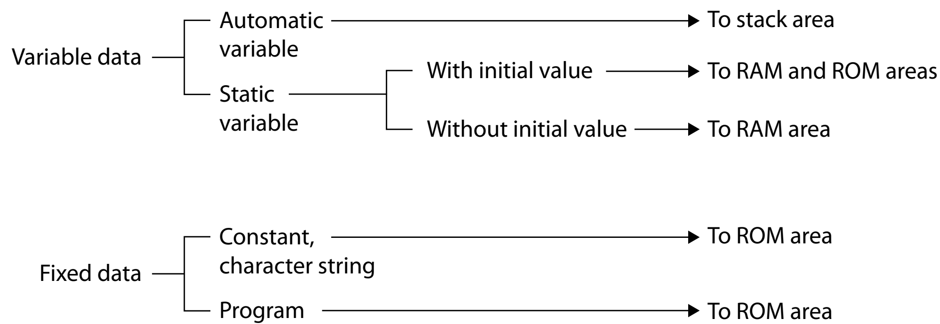


Figure 1.8 Variable Types and Their Storage Destination.

1.3.4 Basic Rules and Process for Writing Source Code

To write effective assembly language, it is necessary to understand the mnemonics and notation used by the assembler. This notation includes the names of addresses in memory, reference points like the stack pointer frame base FB, size specifiers like .W, and register names like R1. The following figure lists relevant RX Family notations.

Size modifiers are used often with both data transfer instructions and branch instructions. If the programmer wants to specify a single byte, the .B modifier is used, as shown in the following example:

```
MOV.B   R1, R2
```

If a 16-bit branch is desired, then the .W modifier is used as shown in the following example:

```
BRA.W   R1
```

A 32-bit modifier of .L can be used to signify a longword, as shown in the following example:

```
MOV.L   R1, R2
```

Writing an entire source program from scratch is not necessary, as setting up the processor, memory, and peripherals is an extensive task. Instead, inserting assembly subroutines into existing .src files is the easiest and most effective way to use assembly with the RX63N, assuming that the system has already been set up. As an example of how to set up and use assembly, follow the step-by-step process to create a function that

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 16

Table 1.1 RX Family Notations [1], page 3.

CLASSIFICATION	NOTATION	MEANING
Symbols	IMM	Immediate value
	SIMM	Immediate value for sign extension according to the processing size
	UIMM	Immediate value for zero extension according to the processing size
	src	Source of an instruction operand
	dest	Destination of an instruction operand
	dsp	Displacement of relative addressing
	pcdsp	Displacement of relative addressing of the program counter
	[ ]	Represents indirect addressing
	Rn	General-purpose register. R0 to R15 are specifiable unless stated otherwise.
	Rs	General-purpose register as a source. R0 to R15 are specifiable unless stated otherwise.
	Rs2	Used in the description for the ADD, AND, CMP, MUL, OR, PUSHM, SUB, and TST instructions. In these instructions, since two general-purpose registers can be specified for an operand, the first general-purpose register specified as a source is described as Rs and the second general-purpose register specified as a source is described as Rs2.
	Rd	General-purpose register as a destination. R0 to R15 are specifiable unless stated otherwise.
	Rd2	Used in the description for the POPM and RTSD instructions. In these instructions, since two general-purpose registers can be specified for an operand, the first general-purpose register specified as a destination is described as Rd and the second general-purpose register specified as a destination is described as Rd2.
	Rb	General-purpose register specified as a base register. R0 to R15 are specifiable unless stated otherwise.
	Ri	General-purpose register as an index register. R0 to R15 are specifiable unless stated otherwise.
	Rx	Represents a control register. The PC, ISP, USP, INTB, PSW, BPC, BPSW, FINTV, and FPSW are selectable, although the PC is only selectable as the src operand of MVFC and PUSHC instructions.
	flag	Represents a bit (U or I) or flag (O, S, Z, or C) in the PSW.
Values	000b	Binary number
	0000h	Hexadecimal number

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 17

Table 1.1 RX Family Notations [1], page 3.—*Continued*

CLASSIFICATION	NOTATION	MEANING
Bit length	#IMM:8 etc.	Represents the effective bit length for the operand symbol.
	:1	Indicates an effective length of 1 bit.
	:2	Indicates an effective length of 2 bits.
	:3	Indicates an effective length of 3 bits.
	:4	Indicates an effective length of 4 bits.
	:5	Indicates an effective length of 5 bits.
	:8	Indicates an effective length of 8 bits.
	:16	Indicates an effective length of 16 bits.
	:24	Indicates an effective length of 24 bits.
	:32	Indicates an effective length of 32 bits.
Size specifiers	MOV.W etc.	Indicates the size that an instruction handles.
	.B	Byte (8 bits) is specified.
	.W	Word (16 bits) is specified.
	.L	Longword (32 bits) is specified.
Branch distance specifiers	BRA.A etc.	Indicates the length of the valid bits to represent the distance to the branch relative destination.
	.S	3-bit PC forward relative is specified. The range of valid values is 3 to 10.
	.B	8-bit PC relative is specified. The range of valid values is −128 to 127.
	.W	16-bit PC relative is specified. The range of valid values is −32768 to 32767.
	.A	24-bit PC relative is specified. The range of valid values is −8388608 to 8388607.
	.L	32-bit PC relative is specified. The range of valid values is −2147483648 to 2147483647.
Size extension specifiers added to memory operands	dsp:16[Rs].UB etc.	Indicates the size of a memory operand and the type of extension. If the specifier is omitted, the memory operand is handled as longword.
	.B	Byte (8 bits) is specified. The extension is sign extension.
	.UB	Byte (8 bits) is specified. The extension is zero extension.
	.W	Word (16 bits) is specified. The extension is sign extension.
	.UW	Word (16 bits) is specified. The extension is zero extension.
	.L	Longword (32 bits) is specified.

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 18

returns the square of its input. In C, this function and the calling function would look like the following:

```
 1. int squared(int r) {
 2.    return (r*r);
 3. }
 4.
 5. int main(void) {
 6.    int squared_var, squaring_var;
  .
  .
10.    squared_var = squared(squaring_var);
  .
  .
  .
```

To create the assembly language function, in the assembly file (file extension .src), declare the function as global. Note that the previous example should be placed at the top of the source file, along with the other declarations. Next, declare the section of memory in which to work. The value in register R1 contains `squaring_var`, which will be `r` in the function `squared`, according to the rules of the Renesas C++ compiler [3]. R1 has the value of `int r` so multiplying it by itself will result in r^2, as desired. Notice that the result is now stored in R1, where is can be retrieved by the calling function. All that is left to do is close the function and return using `RTS`. If this is the end of the source file, then add `.END`.

```
1. .GLB _squared
2. .SECTION P,CODE
3. _squared:
4. MUL.W  R1,R1
5. RTS
6. .END
```

1.3.5 Inline Assembly

Many compilers support the integration of assembly instructions into C code. This implementation is called inline assembly. The most common syntax is of the following form, in which `asm()` is a built-in function ("assembly instruction"); or for more than one instruc-

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 19

tion, encapsulate each instruction in quotation marks and append `\n\t` to each instruction except the last, as in the following example:

```
1. asm("MOV.W   #000aH,R1\n\t"
2.     "MOV.W       #0010H,R2\n\t"
3.     "MOV.W       R2,R3\n\t"
4.     "MOV.W       -2[R12],R4 \n\t"
5.     "JSR         $function1");
```

Alternatively, some compilers use the form:

```
1. #pragma asm
2. ;assembly instructions are inserted here
3. #pragma endasm
```

The RX toolchain that accompanies the RX63N Evaluation Board supports inline assembly. The "#pragma inline_asm" macro performs inline expansion of an assembly language function. This macro has the advantage that it provides a C-type look to the code. The compiler even includes special options and functions that allow users to control the assembly instructions that are generated by the compiler, such as options to prevent DIV, DIVU, and DIVX instructions, which can be a program bottleneck. The GNURX toolchain, which is compatible with HEW, has the `asm()` function built in, but it's functionality is very limited. In general, writing programs in C will provide highly efficient machine code.

If inline assembly must be used, there are numerous third party compilers and toolchains that support the Renesas RX Family of microcontrollers and fully support the `asm()` function or `#pragma asm` declaration.

1.4 BASIC EXAMPLES

1.4.1 Set Up Ports and Turn on LEDs

For this example, assume that the function `LEDfunc()` is called from `main()`.

C Function

```
1. void LEDfunc(void) {
2.     PORTD.PDR.BYTE = 0x20;   //set PORTD to output
3.     PORTE.PDR.BYTE = 0x0f;   //set PORTE Pins 0-3 as output
```

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 20


```
4.    PORTD.PODR.BYTE = 0x00;    //turn on LEDs tied to PORTD
5.    PORTE.PODR.BYTE = 0x00;    //turn on LEDs tied to PORTE
6. }
```

Assembly Function

```
1. .GLB    _LEDfunc
2. _LEDfunc:
3. MOV.L     #08C00DH,R5     ;store address of PORTD Data Direction
                             Register
4. MOV.B     #FFH,[R5]       ;set PORTD DDR to output
5. MOV.B     #0FH,01H[R5]    ;set PORTE DDR Pins 0 to 3 to output
6. MOV.B     #00H,20H[R5]    ;turn on LEDs tied to PORTD
7. MOV.B     #00H,21H[R5]    ;turn on LEDs tied to PORTE
8. RTS
```

Explanation:

```
;Declare the LEDfunc() function as global
.GLB _LEDfunc
;Begin the function
_LEDfunc:
MOV.L   #08C00DH,R5
;Move the memory address 0x8c00d into register R5
MOV.B   #FFH,[R5]
;Move the value 0xff into the memory location that R5 points to
MOV.B   #0FH,01H[R5]
;Move the value 0x0f into the memory address located at *R5 + 0x01
MOV.B   #00H,20H[R5]
;Move the value 0x00 into the memory address located at *R5 + 0x20
MOV.B   #00H,21H[R5]
;Move the value 0x00 into the memory address located at *R5 + 0x21
RTS
;Return from the subroutine
```

1.5 RECAP

Assembly language is a low-level, 1:1 equivalent of machine code that can be used to improve program speed and efficiency. Usually a compiler converts C code into assembly, but

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 21

in the case that our compiler is not sufficiently optimized, we can program our own .src files. The RX63N has 10 addressing modes which must be followed for proper assembly, and numerous other modifiers and syntaxes which are highlighted in Section 1.2.4. The examples provided, along with the RX Family software and hardware manuals, are a guide to getting started with this powerful tool.

1.6 REFERENCES

[1] Renesas Electronics, Inc. (April, 2013). *RX63N Group, RX631 Group User's Manual: Software,* Rev 1.20.

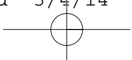
[2] Renesas Electronics, Inc. (February, 2013). *RX63N Group, RX631 Group User's Manual: Hardware,* Rev 1.60.

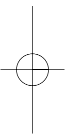
[3] Renesas Electronics Inc. (2011). *RX Family C/C++ Compiler, Assembler Optimizing Linkage Editor, User's Manual,* Rev. 1.0.

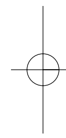
1.7 EXERCISES

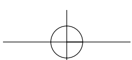
1. How does MOV.L differ from MOV.B and MOV.W?
2. Given the following information, write code to transfer the 16-bit contents of memory at location A to location B:
   - Memory A is located in memory at an offset of 20h from memory location C
   - Memory location C is located at an offset of—1h from memory location 083000h
   - Memory location B is located at an offset of 12 from the stack pointer
   - Register R1 is available, and can store 16 bits
   - Register R5 is available, and can store 24 bits
3. Multiply R1 by R2, and store the result in R3.
4. Multiply the data in memory locations 0x00000a and 0x00000b, and store the result in 0x00000c. Use relative register addressing mode.
5. What is the difference between a JMP instruction and a BRA instruction?
6. How can we use inline assembly with the RX63N?

01.ES_Conrad_RX63N_Advanced_CH01.qxd:RX63N Advanced 3/4/14 12:04 PM Page 22









Function Calls and Stacks

2.1 LEARNING OBJECTIVES

This chapter discusses how the microcontroller manipulates and moves data in memory using the concepts of function calls, stacks, and registers. In this chapter the reader will learn:

- The concept of a function calling interface
- Rules concerning the registers—how to use registers in a function call, parameter passing, and return data types
- Memory mapping of the RX63N microcontroller
- The concept of the stack and stack pointer

2.2 BASIC CONCEPTS

2.2.1 Introduction to Function Calls and Stacks

Functions, stacks, and registers work together to send and obtain data at the right time and without error from the correct memory address. In general, the registers store data processed by the CPU. Blocks of memory are sectioned off in a "stack" of data and memory addresses, and are available for use by functions and their associated variables. A stack is a data structure used to hold and move data to and from registers at the programmer's discretion. In this process, data is transferred by the CPU from a register to the stack. The data is held in the stack until the register reads the data again to return values. When a function is called in the program, the data associated with that function is stored to the stack memory for later use.

2.2.2 Rules for Passing Arguments and Variable Declaration

Function calls are necessary, especially in complex algorithms. Since a function may have one or more declared variables (pieces of data) that can be used for various purposes; these

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:03 PM Page 24

values must be applied to different parts in the program when the function is called to implement an equation or an instruction. When a function is called, it can pass through *arguments* which represent the variable(s) that are needed by the function in order to perform arithmetic or an instruction. This structure is referred to as "passing by value." As seen in the following example, the parameters for this instruction are declared within the function definition:

```
 1. #include <stdio.h>
 2. void main() {
 3.    int num1, num2, sum;
 4.    num1 = 10;
 5.    num2 = 16;
 6.    sum = add(num1,num2);
 7. }
 8.
 9. int add(int a, int b) {
10.    ..................
11.    ...................
12. }
```

In the example, `add` and `main` are functions. The programmer called the `add` function in the `main` function to obtain the sum of two defined integers `num1` and `num2`. The arguments in the main function are `num1` and `num2` and the parameters of `add` are `a` and `b`. Here the values of `num1` and `num2` are passed instead of their address values. The purpose of this line of code is to be able to modify the arguments without changing the value inside of the original variables in `main`. Including arguments within a function is not always necessary.

Each time a function is activated (run), space is needed to store data; this is called an activation record that stores the following:

- arguments—data passed to a function (if a large number of arguments are passed)
- local variables
- return value
- other bookkeeping information

Calling a function B from function A involves:

1. Possibly placing arguments in a mutually-agreed location (registers and/or the stack)
2. Transferring control from function A to function B
3. Allocating space for B's local data
4. Executing the function B
5. Possibly placing return value in a mutually-agreed location

6. Deallocating space for B's
7. Returning control to the function A

This list describes the most basic concept of arguments within function calls. Arguments and parameters must be stored in memory when passed and/or declared. Two methods are available for passing an argument to a function (two memory locations): through a register or on the stack. The microcontroller uses four registers (R1 to R4) to pass arguments from the calling to the called function. These arguments are stored first, starting with the smallest numbered register. When the registers are full, they are pushed to the stack. Refer to Figure 2.1 for the rules on register usage when a function is called and when it returns. Remember the terminology of functions: arguments are values passed; once passed, they become parameters of the called function.

REGISTER	REGISTER VALUE DOES NOT CHANGE DURING FUNCTION CALL	FUNCTION ENTRY	FUNCTION EXIT
R0	Guaranteed	Stack pointer	Stack pointer
R1	Not guaranteed	Parameter 1	Return value 1
R2	Not guaranteed	Parameter 2	Return value 2
R3	Not guaranteed	Parameter 3	Return value 3
R4	Not guaranteed	Parameter 4	Return value 4
R5	Not guaranteed	—	(Undefined)
R6	Guaranteed	—	(Value at function entry is held)
R7	Guaranteed	—	(Value at function entry is held)
R8	Guaranteed	—	(Value at function entry is held)
R9	Guaranteed	—	(Value at function entry is held)
R10	Guaranteed	—	(Value at function entry is held)
R11	Guaranteed	—	(Value at function entry is held)
R12	Guaranteed	—	(Value at function entry is held)
R13	Guaranteed	—	(Value at function entry is held)
R14	Not guaranteed	—	(Undefined)
R15	Not guaranteed	Pointer to return value of structure	(Undefined)

Figure 2.1 Rules to use registers [1], page 231.

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 26

The following example illustrates a detailed version of the previous example. The arguments passed to the function add will be stored in registers R1, R2, and R3.

```
 1. #include <stdio.h>
 2. // The function add is prototyped in the beginning of the code
 3. // before the main function, allowing the compiler to "see" the
 4. // functions that will be executed in the program.
 5.
 6. int add(int a, int b);
 7. void main() {
 8.    int num1, num2, sum;    //These variables are local to main and
 9.                            //will be stored in main's stack.
10.    num1 = 10;
11.    num2 = 16;
12.    sum = add(num1, num2);
13.
14. }
15.
16. int add(int a, int b) {
17.    int c;
18.
19.    c = a + b;
20. return c;
21. }
22. //Here, a = num1 and b = num2.
```

A function call is similar to an Interrupt Service Routine (ISR) in that a new routine initiates somewhere during a current routine. The key difference is that interrupts cannot provide a return value, whereas function calls can provide a return value. The other key difference is the function is initiated in sequence as coded within main where as an ISR is initiated asynchronously.

2.2.3 Concept of Type Conversion in Function Calls

A frequently used conversion technique is *type casting.* When a data type variable needs to be converted to another data type, the programmer can use type casting. Changing data types results in a size change of the variable and, in turn, what data it can hold. Change is appropriate when the programmer wishes to leave the variable as its original type, but use it as another data type in an equation or require a different data size for other reasons, such as truncation.

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 27


```
...............
int a;       \\ a is a signed 16 bit integer
int b = 1;   \\ we have declared and initialized b
char c = 2;  \\ c is a signed 8 bit integer

a = ((int)c*10) + b;
...............
```

In the previous example, `c` has been converted from `char` to `int`. Since the `a` variable is an integer, `c` must also be an integer. Casting a variable is the safest and most efficient way of converting variable data types. For example, if `c` was a number much greater than 16 bits as an `int`, then the result of `a` would have been truncated because the data size of a `char` is smaller than that of an `int`.

Data types of return values can be easily converted in the same fashion. At the function call, the programmer includes syntax similar to the syntax used for converting a single variable. The following example demonstrates a converted function call.

```
 1. #include <stdio.h>
 2.
 3. float compute(float x, float y);
 4. void main() {
 5.    int a, b;
 6.    float c;
 7.    a = 70;
 8.    b = 20;
 9.    c = compute(a, b);
10.    printf ("%3.1f\n", c);
11.
12.    c = (int)compute(a, b);  //The return value of compute is
casted.
13.    printf ("%3.1f\n", c);
14. }
15.
16. float compute(float x, float y) {
17.    float z;
18.    z = x / y;
19.    return z;
20.}
21.
```

In the function `compute` (line 16), the division of `x` and `y` result in a floating point number `z`. The return value `z` is then converted to an integer when compute is called in the `main`

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 28

function. The solution should be 2.5, but the returned value prints as 3 at the second call, because it has been type casted as an integer. Keep in mind that converting data types also means converting sizes of values.

2.2.4 Stack Usage, Allocating, and Deallocating Stack Frames

Before the start of any program, a block of memory is sectioned off for the program's local and global variables, arguments of called functions, return values, and addresses of the stored data in memory. This memory consists of the data, heap, and stack.

Stacks are located in the RAM available in the MCU. The RX63N has a RAM space of up to 256 KB and an address space of up to 4 Gbytes. The following is a representation of the memory stack from the hardware manual. The top stack address represents address 0 in hex and the bottom represents 4 Gbytes in hex. Much more on memory is covered in other sections of this book, but for now we are focused on how data is located in memory.

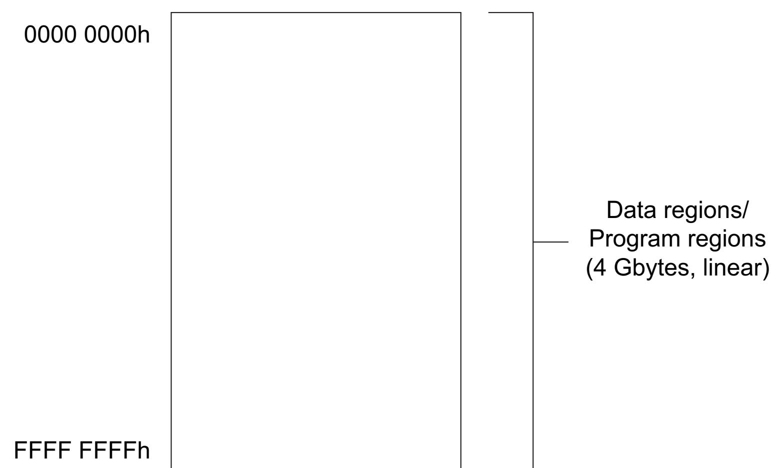


Figure 2.2 RX63N Address Space.

A stack can be thought of as a block of memory, as mentioned earlier. Each block has data stored in a particular order, depending on when variables are declared, initialized, and passed through, called functions. The term "SP" in the following figure stands for stack pointer. The SP is a register containing the smallest address located on the top of the stack. The stack pointer moves as data is allocated and deallocated to and from the stack; that is, the value of SP decreases when data is allocated ("pushed") onto the stack and the value of SP increases when data is deallocated ("popped") from the stack.

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 29

The size of the stack is limited, and care must be taken to ensure it is large enough for holding all of the pushed data and PCs needed during execution of programs. Since the smallest address value is at the very top, any attempt to push data onto the stack beyond the top of the stack causes a *stack overflow.*

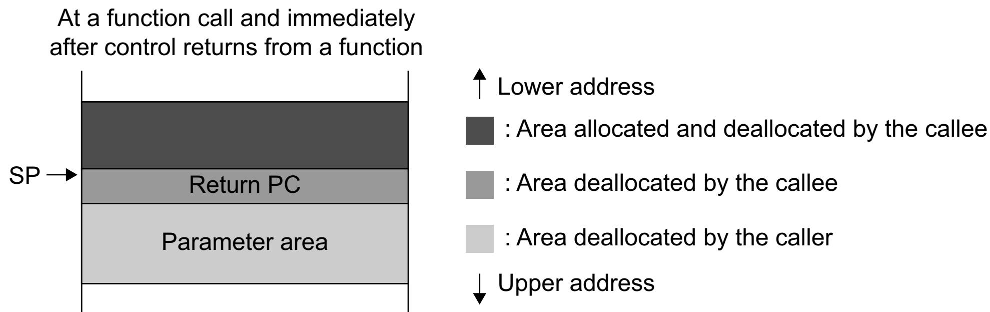


Figure 2.3 Allocation and Deallocation of a Stack Frame [1], page 230.

A program counter (PC) is a processor register that holds the address of the next executable instruction. The PC is pushed on the stack when a subroutine is called. That way the system knows where to "return" to regular processing when the Return from Subroutine instruction is executed and the PC is popped off of the stack.

Sometimes a called subroutine needs additional memory space for variable storage; an example could be a counting variable used for `"for"` loops (`for (i = 1; i < 10; i++)`). These temporary, or automatic variables, use space in a lower address from the PC. In Figure 2.3, this would be the stack space identified in dark grey. Once the subroutine is called, the called subroutine further decrements the SP to account for temporary variables. For example, if `int i;` is defined in the subroutine, then the SP would be decremented by 2 bytes to account for this 2-byte variable.

2.3 BASIC EXAMPLE

Consider the following simple C code:

```
int main2() {
   int a, b, c;
   a = 10;
   b = 16;
```

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 30


```
   c = compute(a,b);
}
int compute(int x, int y) {
   int z;
   z = x + y;
   return(z);
}
```

The equivalent assembly language listing (with the inline C code) is shown as follows:

```
;int main2 () {
 1             .glb        _main2
 2 _main2:                              ; function: main2
 3             .STACK      _main = 16
;int a, b, c;
 4             SUB         #0CH,R0
;a = 10;
 5             MOV.L       #0000000AH,R14
 6             MOV.L       R14,[R0]
;b = 16;
 7             MOV.L       #00000010H,R14
 8             MOV.L       R14,04H[R0]
;c = compute (a,b);
 9             MOV.L       [R0],R1
10             MOV.L       04H[R0],R2
11             BSR         _compute
12             MOV.L       R1,R14
13             MOV.L       R14,08H[R0]
;return(c);
14             MOV.L       08H[R0],R14
15             MOV.L       R14,R1
;}
16             ADD         #0CH,R0
17             RTS
;int compute(int x, int y) {
18             .glb        _compute
19 _compute:                            ; function: compute
20             .STACK      _compute = 16
;int z;
21             SUB         #0CH,R0
```

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 31


```
22          MOV.L       R2,08H[R0]
23          MOV.L       R1,04H[R0]
;z = x + y;
24          MOV.L       04H[R0],R14
25          MOV.L       08H[R0],R1
26          ADD         R14,R1,R2
27          MOV.L       R2,[R0]
;return(z)
28          MOV.L       [R0],R14
29          MOV.L       R14,R1
;}
30          ADD         #0CH,R0
31          RTS
```

In the equivalent assembly code, line 1 sets up the `main` function as global, line 2 labels the main function, and line 3 defines the stack element size. Line 4 subtracts 0x0C from R0. Since R0 is the stack pointer, we have allocated 12 bytes, or 3 longwords, to the stack. Line 5 puts the value of a (0x0A) into register R14, which is then put on the stack with line 6. Lines 7 and 8 perform a similar task with storing the value of variable `b`. Note that `b` is stored 4 bytes ahead of a (using register indirect addressing: 04H[R0]), since each variable is a longword. Next, the program must prepare for a function call, so the values of `a` and `b` are stored in registers R1 and R2 in lines 9 and 10. The BSR command branches to the `compute` subroutine, and the program jumps to line 18.

Lines 18, 19, and 20 define and declare the function similarly to lines 1, 2, and 3 in the main subroutine. Again, we allocate space on the stack (another 12 bytes) with the SUB command in line 21. The values of `a` and `b`, which were stored in R1 and R2 for the purpose of transferring data to the `compute` function, are then stored at 04H[R0] and 08H[R0], respectively. Lines 24 and 25 then move the data from the stack to registers R14 and R1 to perform the addition instruction. Note that the move instructions can sometimes be redundant, but ensure that data is exactly in the right location at the right time. Line 26 performs the addition of R14 and R1, and stores the result in R2. The result in R2 is then stored on the stack at [R0]. Line 28 transfers this data to a general purpose register, and line 29 then moves this data to a register that is suitable to passing from this subroutine back to `main`. The ADD instruction at line 30 serves the purpose of deallocating 12 bytes from the stack in preparation for the return to `main`, and line 31 actually performs the return.

After returning from `compute`, the program counter points to line 12, which moves the data returned from the function (at R1) to a general purpose register (R14). That data is then stored on the stack at 08H[R0] in line 13. The `main` function then prepares to return the value of `c` by moving the data at 08H[R0] to R14 (line 14), then from R14 to R1, a register designed for passing arguments (line 15). Finally, the stack space allocated for main is

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 32

deallocated by subtracting 0x0C from the stack pointer R0 in line 16, and the `main` function returns with line 17.

2.4 ADVANCED CONCEPTS

2.4.1 Memory Mapping

A *memory map* shows the data allocation of addresses to ROM, RAM, or flash memory in a system. The following example shows where the variables and executable program are stored in memory. The compiler determines where the data is stored.

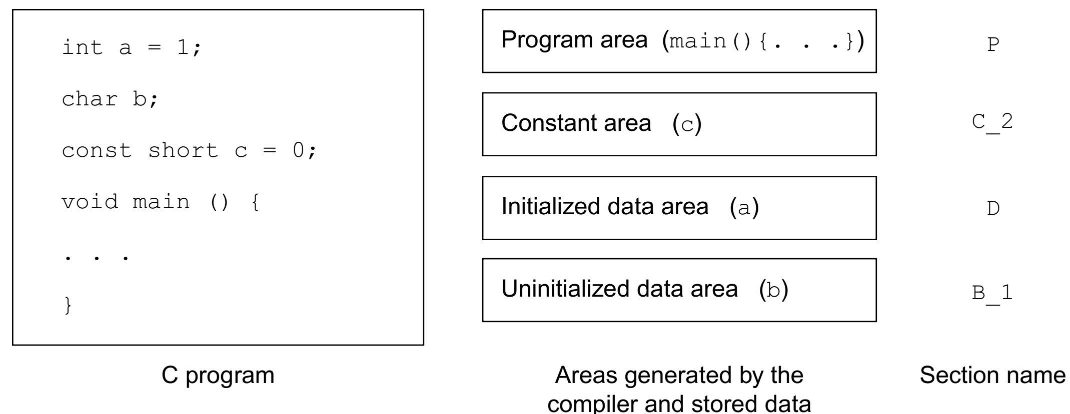


Figure 2.4 Mapping data into sections by type [1], page 222.

The following figure shows a memory map in various operating modes. The accessible areas differ according to the operating mode and the states of control bits. As mentioned earlier, this microcontroller has a 4 Gbyte address space ranging from 0000 0000h to FFFF FFFFh. The reserved areas shown in all three operating modes are not accessible by users/programmers. In general, the figure in the previous example depicts a high level view of what is happening between the executed code and the stack.

The simplest way to map to memory is to use a pointer. A *pointer* variable holds the address of the data, rather than the data itself. To store the address of a variable into a pointer, use the reference operator "&" in front of the variable. A pointer is declared by using the indirection operator "*" and specifying the data type (and size) which the address will hold. The following example shows where two variables and their pointers are moved around in a stack.

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 33

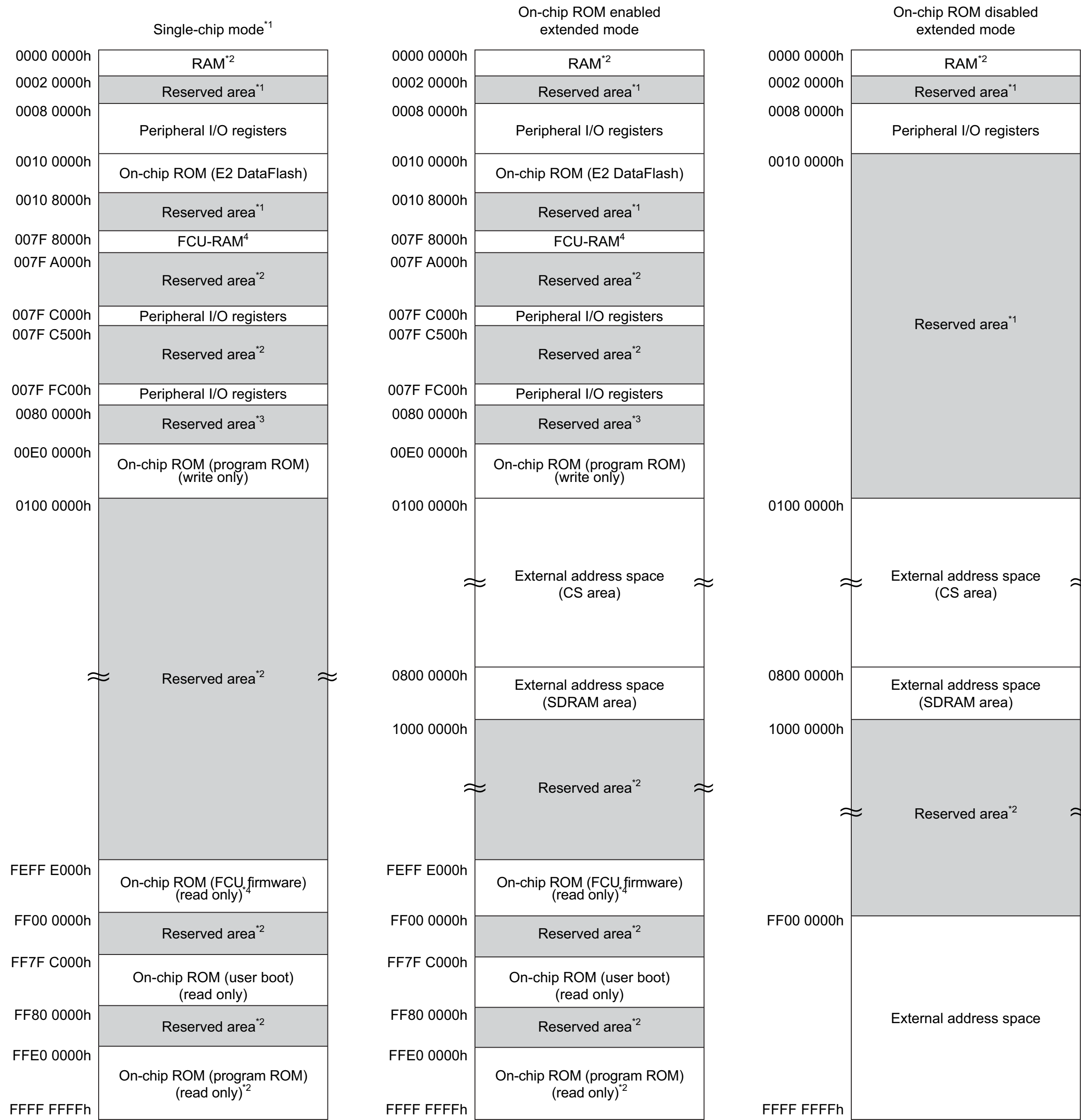


Figure 2.5 Memory map in each operating mode [2], page 151.—*Continued*

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 34

ROM (bytes)			RAM (bytes)	
CAPACITY	ADDRESS (FOR READING ONLY)	ADDRESS (FOR PROGRAMMING ONLY)	CAPACITY	ADDRESS
2 M	FFE0 0000h to FFFF FFFFh	00E0 0000h to 00FF FFFFh	256 K	0000 0000h to 0003 FFFFh
			192 K	0000 0000h to 0002 FFFFh
			128 K	0000 0000h to 0001 FFFFh
1.5 M	FFE8 0000h to FFFF FFFFh	00E8 0000h to 00FF FFFFh	256 K	0000 0000h to 0003 FFFFh
			192 K	0000 0000h to 0002 FFFFh
			128 K	0000 0000h to 0001 FFFFh
1 M	FFF0 0000h to FFFF FFFFh	00F0 0000h to 00FF FFFFh	256 K	0000 0000h to 0003 FFFFh
			192 K	0000 0000h to 0002 FFFFh
			128 K	0000 0000h to 0001 FFFFh
768 K	FFF4 0000h to FFFF FFFFh	00F4 0000h to 00FF FFFFh		
512 K	FFF8 0000h to FFFF FFFFh	00F8 0000h to 00FF FFFFh		
384 K	FFFA 0000h to FFFF FFFFh	00FA 0000h to 00FF FFFFh		
256 K	FFFC 0000h to FFFF FFFFh	00FC 0000h to 00FF FFFFh		
512 K	FFF8 0000h to FFFF FFFFh	00F8 0000h to 00FF FFFFh	64 K	0000 0000h to 0000 FFFFh
384 K	FFFA 0000h to FFFF FFFFh	00FA 0000h to 00FF FFFFh		
256 K	FFFC 0000h to FFFF FFFFh	00FC 0000h to 00FF FFFFh		

Note 1. The address space in boot mode and user boot mode/USB boot mode is the same as the address space in single-chip mode.
Note 2. The capacity of ROM/RAM differs depending on the products.
Note 3. Reserved areas should not be accessed.
Note 4. For details on the FCU, see section 46 of [2], Flash Memory.

Figure 2.5 *Continued.*

2.5 ADVANCED EXAMPLE

Consider the following simple C code:

```
const int globalD = 6;
int compute(int x, int y);
int squared(int r);
void main() {
   int a, b, c;
   a = 10;
   b = 16;
   c = compute(a,b);
}
```

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 35


```
int compute(int x, int y) {
   int z;
   z = squared(x);
   z = z + squared(y) + globalD;
   return(z);
}
int squared(int r) {
   return (r*r);
}
```

The equivalent assembly language listing (with the inline C code) is shown as follows:

```
;Optimization level = 0
const int globalD = 6;
;void main() {
 1.           .glb         _main
 2. _main:                              ; function: main
 3.           .STACK       _main=16
;int a, b, c
 4.           SUB          #0CH,R0
;a = 10;
 5.           MOV.L        #0000000AH,R14
 6.           MOV.L        R14,[R0]
;b = 16;
 7.           MOV.L        #00000010H,R14
 8.           MOV.L        R14,04H[R0]
;c = compute(a,b)
 9.           MOV.L        [R0],R1
10.           MOV.L        04H[R0],R2
11.           BSR          _compute
12.           MOV.L        R1,R14
13.           MOV.L        R14,08H[R0]
;}
14.           ADD          #0CH,R0
15.           RTS
;int compute(int x, int y) {
16.           .glb         _compute
17. _compute:                            ; function: compute
18.           .STACK       _compute=16
;int z;
19.           SUB          #0CH,R0
```

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 36


```
20.            MOV.L       R2,08H[R0]
21.            MOV.L       R1,04H[R0]
;z = squared(x);
22.            MOV.L       04H[R0],R1
23.            BSR         _squared
24.            MOV.L       R1,R14
25.            MOV.L       R14,[R0]
;z = z + squared(y) + globalD;
26.            MOV.L       08H[R0],R1
27.            BSR         _squared
28.            MOV.L       [R0],R14
29.            ADD         R14,R1,R5
30.            MOV.L       #_globalD,R14
31.            MOV.L       [R14],R2
32.            ADD         R5,R2,R14
33.            MOV.L       R14,[R0]
;return(z);
34.            MOV.L       [R0],R14
35.            MOV.L       R14,R1
;}
36.            ADD         #0CH,R0
37.            RTS
;int squared(int r) {
38.            .glb        _squared
39. _squared:                              ; function: squared
40.            .STACK      _squared=8
41.            SUB         #04H,R0
42.            MOV.L       R1,[R0]
;return (r*r);
43.            MOV.L       [R0],R14
44.            MOV.L       [R0],R2
45.            MUL         R14,R2,R5
46.            MOV.L       R5,R1
;}
47.            ADD         #04H,R0
48.            RTS
```

Lines 1 to 11: The main subroutine is first declared as global in line 1, and labeled in line 2. The stack is then declared in line 3. To allocate space on the stack, we subtract the necessary amount of bytes from the stack

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 37

pointer, R0. This happens in line 4, where 12 bytes are allocated. The immediate value 0x0A (the value of variable `a`) is then moved into register 14 in line 5, and then R14 is moved onto the stack at location [R0] in line 6. A similar sequence happens with lines 7 and 8, where the value of variable `b` is ultimately stored on the stack at location 04H[R0]. The subroutine then prepares for a function call by storing the data from variables `a` and `b` in registers R1 and R2 in lines 9 and 10. Note that these registers are designed for passing arguments to and from functions. Line 11 calls the "compute" function.

Lines 16 to 23: The `compute` subroutine declarations with lines 16, 17, and 18 are similar to lines 1, 2, and 3 for the `main` subroutine. Another 0x0C bytes are allocated to the stack in line 19, and the contents of the passed arguments from R1 and R2 (variables `x` and `y`) are stored in this new allocated space at locations 04H[R0] and 08H[R0], respectively. Then, the program prepares for another branch by passing the argument stored at 04H[R0] (the variable `x`) into R1 in line 22. Line 23 branches to the `squared` function.

Lines 38 to 48: The `squared` subroutine again starts off with declarations in lines 38, 39, and 40. Next, line 41 allocates 4 bytes to the stack, and line 42 stores the data from R1 (the `x` variable passed from the `compute` function) on the stack at location [R0]. The next two lines replicate this data in registers R14 and R2, and line 45 performs a multiplication of these two registers, storing the result in R5. The function then prepares to return this value back to `compute` by storing it in R1 (line 46). Next, 4 bytes are deallocated from the stack by adding 0x04 to the stack pointer, R0, in line 47. Finally, the function returns to `compute` with line 48.

Lines 24 to 27: The value that was returned from `squared` is moved out of R1 into a general purpose register, R14, in line 24. This data is then moved onto the `compute` stack at location [R0] in line 25. This stores the variable `z`. Line 26 prepares another function call by storing the value located at 08H[R0] in R1. This is the value of variable `y`, which will be passed to `squared` by branching at line 27.

Lines 38 to 48: The `squared` subroutine performs the same as described earlier. The only difference is that this time the value of y is passed into this function, and the returned result will be y^2.

Lines 28 to 37: The value of the `z` variable, stored on the stack at [R0], is now moved into R14 with line 28. Line 29 adds R14 to R1, and stores

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 38

this result in R5. Thus, R5 contains: $z + y^2$. To get the desired final result of $z = z + y^2 + \text{globalD}$, we still need to add the value of globalD to the data in R5. The location of variable globalD is stored in R14 with line 30. Note that we are assuming globalD is declared earlier in the program, at the header. The contents at the memory location pointed to by R14 is then stored directly in R2 with line 31. The contents of R5 and R2 are then summed with line 32, and the result stored in R14. This desired value is then stored on the stack at [R0] with line 33. Although it is redundant, [R0] is then moved into R14, which is them moved into R1, where it will be ready to be returned to another subroutine. This redundancy is not always necessary, but without time constraints, we can make sure that our value is not lost or overwritten by precisely moving it from the stack to a general purpose register, and from a general purpose register into a register designed to pass arguments. Line 36 deallocates the 12 bytes that were allocated to the stack in line 19. Line 37 returns to the `main` subroutine.

Lines 12 to 15: The argument from `compute` is stored in a general purpose register, R14, with line 12. Then, line 13 stores this result on the `main` stack at location 08H[R0]. Since the data is ultimately not used, the subroutine deallocates its stack with line 14 and returns with line 15. If the data were used by another function, then it would have been stored in R1, R2, R3, or R4 before returning.

As an example of the value of optimization, compare the previous assembly language code with the following code:

```
;Optimization level = 1
;const int globalD = 6;
;void main() {
 1.            .glb        _main
_main:                            ; function: main
 2.            .STACK      _main=16
;int a, b, c
 3.            SUB         #0CH,R0
;a = 10;
 4.            MOV.L       #0000000AH,R1
;b = 16;
 5.            MOV.L       #00000010H,R2
;c = compute(a,b)
```

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 39


```
 6.           BSR         _compute
 7.           MOV.L       R1,R5
 8.           MOV.L       R5,R1
;}
 9.           RTSD        #0CH

;int compute(int x, int y) {
10.           .glb        _compute
_compute:                         ; function: compute
11.           .STACK      _compute=12
;int z;
12.           PUSH.L      R6
13.           SUB         #04H,R0
;z = squared(x);
14.           MOV.L       R2,R6
15.           BSR         _squared
16.           MOV.L       R1,R4
17.           MOV.L       R4,[R0]
;z = z + squared(y) + globalD;
18.           MOV.L       R6,R1
19.           BSR         _squared
20.           MOV.L       [R0],R4
21.           ADD         R4,R1,R5
22.           MOV.L       #_globalD,R14
23.           MOV.L       [R14],R2
24.           ADD         R5,R2,R4
;return(z)
25.           MOV.L       R4,R1
;}
26.           RTSD        #08H,R6-R6

;int squared(int r) {
27.           .glb        _squared
_squared:                         ; function: squared
28.           .STACK      _squared=4
    ;return (r*r);
29.           MUL         R1,R1,R5
30.           MOV.L       R5,R1
;}
31.           RTS
```

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 40

In the optimized code, the compiler has removed extra data movement instructions. For example, in the `squared` function, the compiler generated instructions that do not even store the passed value `r` onto the stack. This optimization reduced the number of assembly language instructions in `squared` from eight to three. The optimized version has only 24 executable instructions versus 39 in the non-optimized version.

2.6 RECAP

This chapter discussed the fundamental concepts of data control in memory. Significant errors in embedded systems are often caused by inaccurate data type assignment and register addressing. Microcontrollers, especially the RX63N, rely exclusively on accurate register assignment to allow control and communications of data across peripherals on the board. For this reason, it is crucial to understand where data is stored in registers.

Data types handled by the RX63N include integer, floating point, bitwise, and strings. They can be converted to other data types using *type casting.* A function may have one to several variables used to execute some algorithm. When a function is called, it passes through *arguments,* which are copies of these variables to use.

The lifetime of a variable is tracked in memory by a block of memory called the *stack.* The stack represents memory where temporary variables are held. It is also the location which holds the return addresses of the program instruction that called a functional to executed. The RX63N has a 4 Gbyte memory address, allowing for large program spaces and large data sizes for complex algorithms.

Assembly language can be user created and modified. However, a programmer must be cautious not to overwrite memory spaces with unwanted data. Only a few assembly instructions are used frequently in the RX63N compiler. If a programmer wants to replace data in a specific memory address or obtain the address of data, a *pointer* could be used; therefore, modifying assembly code is not always necessary. For more information specific to the RX63N microcontroller, visit section 8.2 of the *RX Family C/C++ Compiler, Assembler, Optimizing Linkage Editor* document [5].

2.7 REFERENCES

[1] Renesas Electronics Inc. (2011). *RX Family C/C++ Compiler, Assembler Optimizing Linkage Editor, User's Manual,* Rev. 1.0.

[2] Renesas Electronics, Inc. (February, 2013). *RX63N Group, RX631 Group User's Manual: Hardware,* Rev. 1.60.

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 41

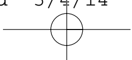
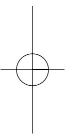
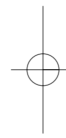
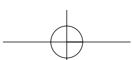
2.8 EXERCISES

1. For our processor and compiler, does the stack grow toward larger or smaller addresses?
2. What is the total address space for a RX63N microcontroller?
3. What causes a stack overflow?
4. What moves on the stack as data and is allocated/deallocated?
5. Why would you want to use a pointer in your function?
6. What is the easiest way to pass variables of the following type to another function:
   a. char
   b. long
   c. 4 integers
7. What are the stack contents after the following program executes? Assume the stack before this executes is empty, with 2 bytes allocated.

```
1. jsr   $newfunc1
2. jsr   $newfunc2
3. $newfunc1
4. rts
5. $newfunc2
6. Rts
7.
```

8. How can we efficiently pass an integer from one subroutine to another?
9. How would we pass a string from one subroutine to another?

02.ES_Conrad_RX63N_Advanced_CH02.qxd:RX63N Advanced 3/4/14 12:04 PM Page 42

Floating Point Unit and Operations

3.1 LEARNING OBJECTIVES

All microprocessors can store and operate on integers. However, often we wish to operate on real numbers. We can represent these values with fixed-point or floating-point approaches. Fixed-point math is nearly as fast as integer math. Its main drawback is that the designer must understand the sizes of the smallest and largest data values in order to determine the appropriate fixed-point representation. Floating-point math automatically adjusts the scaling to account for large and small values.

Generally floating-point math is emulated in software, but some microprocessors have special hardware called a Floating Point Unit (FPU) for executing these operations natively, greatly increasing the speed. Though the FPU varies with the microcontroller, or with different families of microcontrollers, we will be discussing the FPU in the RX63N microcontroller and comparing it with the software emulation of a FPU on the QSK62P board.

In this chapter the reader will learn:

- The Instruction Set Architecture (ISA) of the RX63N Floating Point Unit
- Differences between a fixed point and a floating point operation
- Handling floating point exceptions

3.2 BASIC CONCEPTS

3.2.1 Floating Point Basics

Floating Point Representation

Before taking a look at the FPU in the RX63N, let's look at the basics of floating point representation.

Consider a floating point number, say 241.32478_{10}. It denotes the following.

10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}
2	4	1	.	3	2	4	7	8

Figure 3.1 Decimal floating point representation.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 44

The number in the box is the significand, 10 is the base and the power to which 10 is raised is the exponent.

We can represent the above number in exponential form as 2.4132478×10^2.

Consider a binary number, say 1011.0110_2. The number can be represented as:

2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	1	1	.	0	1	1	0

Figure 3.2 Binary floating point representation.

We can represent the above binary number in exponential form as 1.0110110×2^3.

From the above notations, we can generalize the floating point expression as:

$$(-1)^{\text{sign}} \times \text{Significand} \times \text{Base}^{\text{Exponent}}$$

Though the decimal floating point representation is simple, it cannot be used by microcontrollers to work on floating point numbers. Since they work on binary digits, the IEEE 754 Floating Point Standard was introduced to represent decimal numbers as binary numbers.

IEEE 754 Floating Point Standard

In order to standardize floating point arithmetic, the IEEE 754 floating point standard was first introduced in 1985. Though the current version of IEEE 754 is IEEE 754–2008, it includes the original information in IEEE 754–1985 as well.

The binary floating point numbers represented by IEEE 754 are in sign magnitude form where the most significant bit (MSB) is the sign bit followed by the biased exponent and significand without the MSB. The field representation of IEEE 754 is as follows.

Figure 3.3 IEEE 754 floating point representation.

The IEEE 754 defines four formats for representing floating point values.

- Single precision (32 bits)
- Double precision (64 bits)

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 45

- Single extended precision (≥ 43 bits, not commonly used)
- Double extended precision (≥ 79 bits, not commonly used)

The IEEE 754 Floating Point Standard

Though the number of bits varies in each precision, the format for representing the floating point number is the same. The format for representing a floating point number according to IEEE 754 is as follows.

$$(-1)^{\text{sign}} \times (1 + \text{Significand}) \times 2^{\text{Exponent-Bias}}$$

Since we will not be using single extended precision and double extended precision, we will look at single precision and double precision in detail.

Single Precision (32 bits)

Single precision consists of 4 bytes (or) 32 bits, where the 32 bits are divided into the following fields.

Figure 3.4 IEEE 754 single precision floating point representation.

In order to obtain the single precision representation, a few general steps are to be followed.

- Convert the decimal into a binary number.
- Express the binary number in normalized exponential notation. Since we are representing binary numbers, the power will be raised to 2, which is the base.
- Since we need to represent the exponent as an 8 bit binary number, we add 127 to the exponent in the normalized notation if the number we need to represent is positive.
- If the number we need to represent is negative, we subtract the exponent in the normalized notation from 127.
- Having done that, we add the significand bits by extending it to 23 bits.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 46

EXAMPLE 1

A decimal number can be represented as a single precision number as follows. Consider a number, say 10.0_{10}

$$10.0_{10} = 1010 \times 2^{0} \text{ or } 1.010 \times 2^{3}$$

In the above number 1.010×2^3, the significand is 010 as the number before the decimal point will always be 1. Since the significand here does not have a specific pattern of repetition, the 23 bit significand field will be the significand appended with zeroes at the end. From the above number:

Sign Bit: 0 (Since the number is positive).

Exponent: 127 + 3 = 130. For simplification, let's split 130 as 128 + 2. As 128 is nothing but 8-bit or 10000000_2

$$128_{10} + 2_{10} = 10000000_2 + 00010_2 = 10000010_2$$

Significand: 0100 0000 0000 0000 0000 000 (010 appended with zeroes)

From the single precision field representation, we can represent this number as:

Figure 3.5 Example 1; single precision conversion.

EXAMPLE 2

Consider a negative floating point number, say -0.1_{10}.

$$-0.1_{10} = 0.000110011 \times 2^{0} \text{ or } 1.10011 \times 2^{-4}$$

In the binary number 1.1**0011,** the 0011 pattern is repeated. Since the number has a pattern which gets repeated, the significand is 1 followed by a pattern of 0011.

Therefore, from the above number:

Sign bit: 1 (Since the number is negative)

Exponent: 127 − 4 = 123

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 47

Since 123 is less than 128, we cannot adopt the technique used in Example 3.1.

We know $123_{10} = 0111\ 1011_2$

Significand: 1001 1001 1001 1001 1001 100

From the single precision field representation, we can represent this number as,

Figure 3.6 Example 2; single precision conversion.

Double Precision (64 bits)

Double precision consists of 8 bytes (or) 64 bits, where the 64 bits are divided into the following fields.

Figure 3.7 IEEE 754 double precision floating point representation.

Obtaining the double precision numbers, follow the same steps we adopted for single precision numbers but with different bits.

In order to obtain the double precision representation, the following steps are adopted:

- Convert the decimal into a binary number.
- Express the binary number in normalized exponential notation. Since we are representing binary numbers, the power will be raised to 2, which is the base.
- Since we need to represent the exponent as an 11 bit binary number, we add 1023 to the exponent in the normalized notation if the number we need to represent is positive.
- If the number we need to represent is negative, we subtract the exponent in the normalized notation from 1023.
- Having done that, we add the significand bits which is 52 bits wide.

Note that the Renesas RX63N does not have 64-bit floating point registers, so double precision operations are handled by library calls. Note that single precision operations are sufficient for the precision of many applications. Some compilers have option switches to convert double precision variables and operations into single precision variables and operations.

EXAMPLE 3

Consider Example 3.1. To represent the number 10_{10} to a double precision number, we do the following.

$$10 = 1010 \times 2^0 \text{ or } 1.010 \times 2^3$$

In the above number 1.010×2^3, the significand is 010 as the number before the decimal point will always be 1. Since the significand here does not have a specific pattern of repetition, the 52 bit significand field will be the significand appended with zeroes at the end.

From the above number,

Sign Bit: 0 (Since the number is positive).

Exponent: $1023 + 3 = 1026_{10} = 10000000010_2$
Significand: 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 (010 appended with zeroes)
From the double precision field representation, we can represent this number as:

Figure 3.8 Example 3; double precision conversion.

EXAMPLE 4

Consider Example 3.2. To represent the number -0.1_{10} to a double precision number, we do the following:

$$-0.1 = 0.00011 \times 2^0 \text{ or } 1.10011 \times 2^{-4}$$

Therefore, from the above number,

Sign Bit: 1 (Since the number is negative)

Exponent: $1023 - 4 = 1019$
$1019_{10} = 01111111011_2$

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 49

Significand: 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010

From the double precision field representation, we can represent this number as:

Figure 3.9 Example 4; double precision conversion.

3.2.2 Pipelining Basics

In order to understand the floating point operations in microcontrollers and the difference between fixed point and floating point operations, it is necessary to take a look at the some of the basics of pipelining.

Pipelining is the technique adopted for increasing the instruction throughout and reducing the number of clock cycles required to execute an instruction by microprocessors. It is a method of realizing temporal parallelism, an assembly line processing technique. The instruction is broken down into a sequence of sub-tasks, each of which can be executed concurrently with other stages in the pipeline. The basic instruction cycle consists of five stages:

- Instruction Fetch
- Instruction Decode
- Execute
- Memory Access
- Write Back

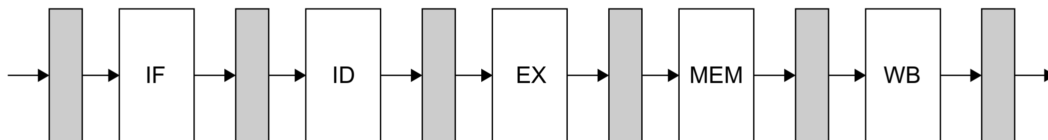


Figure 3.10 A basic pipeline.

Instruction Fetch

- The Instruction Fetch (IF) cycle fetches the current instruction from the memory into the program counter (PC).
- It updates the PC to the next sequential instruction.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 50

Instruction Decode

- The Instruction Decode cycle (ID) decodes the instruction and reads the registers corresponding to the specified register source.
- It performs an equality test of registers for a possible branch.
- The decoding is done in parallel with reading registers.

Execution

The Execution (EX) cycle does the following:

- Memory Reference—Calculates effective address.
- Register—Register Operation—ALU performs the operation specified by the ALU opcode on the values read from the register file.
- Register-Immediate Operation—Operation specified by the ALU on the value read from the register and the sign-extended immediate.

Memory Access

The Memory Access (MEM) cycle does the following operations:

- Load—A read operation performed by the memory using the calculated effective address.
- Store—A write operation performed by the memory from the register read from the register file using the effective address.

Write-Back

- The Write-Back (WB) cycle writes the result into the register file, whether it comes from the memory system (load) or from the ALU (ALU instruction).

3.2.3 Floating Point in RX63N

In order to handle floating point operations, RX63N has the following:

- A Floating Point Unit
- Floating Point Instructions
- Floating Point Registers
- Floating Point Exceptions

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 51

Floating Point Unit

The Floating Point Unit (FPU) in RX63N operates on single precision floating point operands (32-bit) and the data-types and floating point exceptions conform to the IEEE 754 standard. This implementation facilitates the FPU to operate directly on the CPU registers rather than the dedicated register sets, thereby avoiding extra load/store operations and improving the performance. Figure 3.11 shows the floating point unit architecture in RX63N.

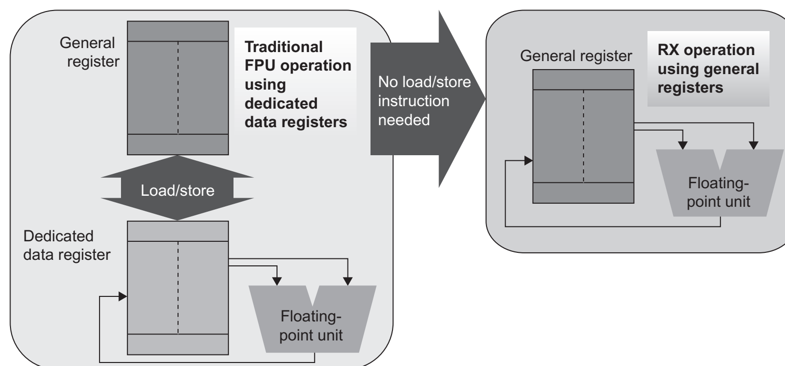


Figure 3.11 Architecture used by RX63N with dedicated registers for floating point operations.

Floating Point Registers

The RX63N uses a single floating point register, Floating-Point Status Word (FPSW) to indicate the results of the floating-point operations. It is a 32-bit register consisting of flags, bits representing modes for rounding off floating point numbers, exception enable/disable bits and reserved bits. Figure 3.12 shows the FPSW bits and Table 3.1 shows the bit definitions for the FPSW.

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	FS	FX	FU	FZ	FO	FV	—	—	—	—	—	—	—	—	—	—
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	EX	EU	EZ	EO	EV	—	DN	CE	CX	CU	CZ	CO	CV	RM[1:0]	
Value after reset:	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

Figure 3.12 Floating-point status word [2], Page 24.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 52

TABLE 3.1 Floating-Point Status Word Bit Definition [2], Page 24.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b1, b0	RM[1:0]	Floating-point rounding-mode setting bits	b1 b0	R/W
			0 0: Round to the nearest value	
			0 1: Round towards 0	
			1 0: Round towards $+\infty$	
			1 1: Round towards $-\infty$	
b2	CV	Invalid operation cause flag	0: No invalid operation has been encountered.	R/(W)[*1]
			1: Invalid operation has been encountered.	
b3	CO	Overflow cause flag	0: No overflow has occurred.	R/(W)[*1]
			1: Overflow has occurred.	
b4	CZ	Division-by-zero cause flag	0: No division-by-zero has occurred.	R/(W)[*1]
			1: Division-by-zero has occurred.	
b5	CU	Underflow cause flag	0: No underflow has occurred.	R/(W)[*1]
			1: Underflow has occurred.	
b6	CX	Inexact cause flag	0: No inexact exception has been generated.	R/(W)[*1]
			1: Inexact exception has been generated.	
b7	CE	Unimplemented processing cause flag	0: No unimplemented processing has been encountered.	R/(W)[*1]
			1: Unimplemented processing has been encountered.	
b8	DN	0 flush bit of denormalized number	0: A denormalized number is handled as a denormalized number.	R/W
			1: A denormalized number is handled as 0.[*2]	R/W
b9	—	Reserved	When writing, write 0 to this bit. The value read is always 0.	R/W
b10	EV	Invalid operation exception enable bit	0: Invalid operation exception is masked.	R/W
			1: Invalid operation exception is enabled.	
b11	EO	Overflow exception enable bit	0: Overflow exception is masked.	R/W
			1: Overflow exception is enabled.	

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 53

TABLE 3.1 Floating-Point Status Word Bit Definition [2], Page 24.—*Continued*

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b12	EZ	Division-by-zero exception enable bit	0: Division-by-zero exception is masked. 1: Division-by-zero exception is enabled.	R/W
b13	EU	Underflow exception enable bit	0: Underflow exception is masked. 1: Underflow exception is enabled.	R/W
b14	EX	Inexact exception enable bit	0: Inexact exception is masked. 1: Inexact exception is enabled.	R/W
b25 to b15	—	Reserved	When writing, write 0 to these bits. The value read is always 0.	R/W
b26	FV*3	Invalid operation flag	0: No invalid operation has been encountered. 1: Invalid operation has been encountered.*8	
b27	FO*4	Overflow flag	0: No overflow has occurred. 1: Overflow has occurred.*8	R/W
b28	FZ*5	Division-by-zero flag	0: No division-by-zero has occurred. 1: Division-by-zero has occurred.*8	R/W
b29	FU*6	Underflow flag	0: No underflow has occurred. 1: Underflow has occurred.*8	R/W
b30	FX*7	Inexact flag	0: No inexact exception has been generated. 1: Inexact exception has been generated.*8	R/W
b31	FS	Floating-point error summary flag	This bit reflects the logical OR of the FU, FZ, FO, and FV flags.	R

Notes:
1. When 0 is written to the bit, the bit is set to 0; the bit remains the previous value when 1 is written.
2. Positive denormalized numbers are treated as +0, negative denormalized numbers as –0.
3. When the EV bit is set to 0, the FV flag is enabled.
4. When the EO bit is set to 0, the FO flag is enabled.
5. When the EZ bit is set to 0, the FZ flag is enabled.
6. When the EU bit is set to 0, the FU flag is enabled.
7. When the EX bit is set to 0, the FX flag is enabled.
8. Once the bit has been set to 1, this value is retained until it is cleared to 0 by software.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 54

Floating Point Instructions

The RX63N uses eight floating point instructions to perform floating point operations. All floating point instructions work on single precision floating point operands conforming to IEEE754 standard. The floating point instructions are as follows:

FADD

Function: Adds two floating point operands and stores the result in the register file.
Operations performed: Immediate-Register and Register-Register.
Number of cycles taken: Four.

FADD Operation:

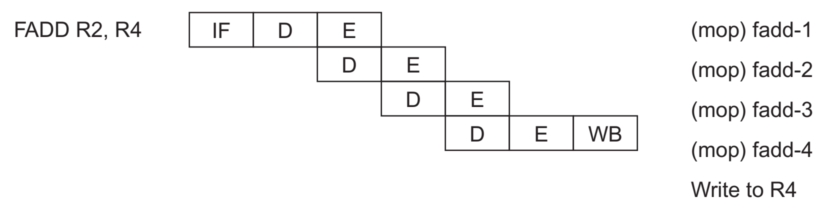


Figure 3.13 FADD Operation (Register-Register, Immediate-Register) [1], Page 138.

FSUB

Function: Subtracts the second floating point operand from the first floating point operand and stores the result in the register file.
Operation performed: Immediate-Register, Register-Register.
Number of cycles taken: Four.

FSUB Operation:

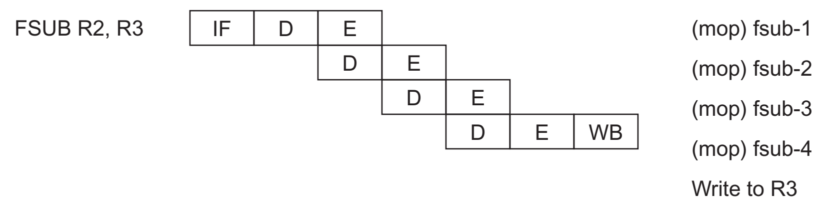


Figure 3.14 FSUB Operation (Register-Register, Immediate-Register) [1], Page 138.

FCMP

Function: Compares the floating point operands and updates the status register based on the results. The difference between FSUB and FCMP is that, FCMP does not store the result anywhere.

Operation performed: Immediate-Register, Register-Register.
Number of cycles taken: Four.

FSUB Operation:

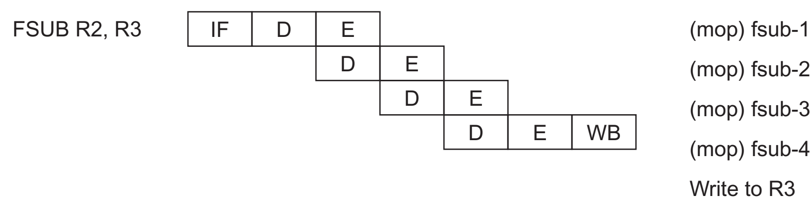


Figure 3.15 FSUB Operation (Register-Register, Immediate-Register) [1], Page 138.

Floating Point Exceptions

The floating point exceptions are basically exceptions which arise due to inappropriate floating point operations. According to IEEE 754 standard, there are five floating-point exceptions as follows:

- Overflow
- Underflow
- Inexact
- Division-by-zero
- Invalid Operation

Overflow

This exception occurs when the result of a floating-point operation is too large. Under such circumstances, the result cannot be represented as a single precision floating point number (32 bit) or a double precision floating point number (64 bit).

- The limits for signed single precision floating point numbers are 3.4E − 38 to 3.4E + 38.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 56

- The limits for unsigned single precision floating point numbers are 0 to 6.8E38.
- The limits for signed double precision floating point numbers are 1.7E − 308 to 1.7E + 308.
- The limits for unsigned double precision floating point numbers are 0 to 3.4E + 308.

If the result exceeds these limits, an overflow exception is raised.

Underflow

This exception occurs when the result is too small to be represented as a normalized floating-point number.

Inexact

This exception occurs when the result is not a single precision floating point number. Rounding off the result and using it as a single precision floating point number may cause this exception.

Divide-By-Zero

This exception occurs when a floating point number is divided by zero.

Invalid Operation

This exception occurs when the result is ill-defined. For example, an operation leading to a result which is $\pm\infty$ (Infinty) or NaN (Not a number) could cause this exception.

For example, $1.0 / 0.0 = +\infty$, $0.0 / 0.0 =$ NaN.

Floating point operations could result in a signaling NaN (SNan) or a quiet Nan (QNaN). Each of these types of NaN and a description of when they generate exceptions are described in the software user's manual [2].

3.3 BASIC EXAMPLES

3.3.1 Operations Explaining Floating Point Exceptions

Before working on the RX63N's FPU, it is important to take a look at the operations causing Floating Point Exceptions, so that they can be handled efficiently. The Floating-point error summary flag (FS) notifies the programmer that an error (overflow, underflow, invalid operation, inexact, or division-by-zero) has occurred.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 57

EXAMPLE 1

Overflow Exception

Consider the following example:

```
1. float a, c;
2. a = 999999969999999923.235;
3. c = a * 999999999999969999999923.235;
```

The above operation causes an Overflow exception. To observe this exception, the contents of the FPSW can be viewed using the Renesas HEW Integrated Development Environment (IDE) tool.

Open up the HEW IDE tool, and in the debugging tool, click on the (Registers icon) in the CPU Toolbox.

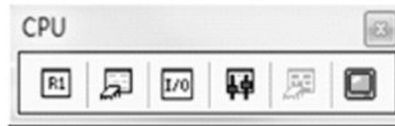


Figure 3.16 CPU toolbox.

For the above example, the FPSW was C8000100, where the Radix is Hexadecimal.

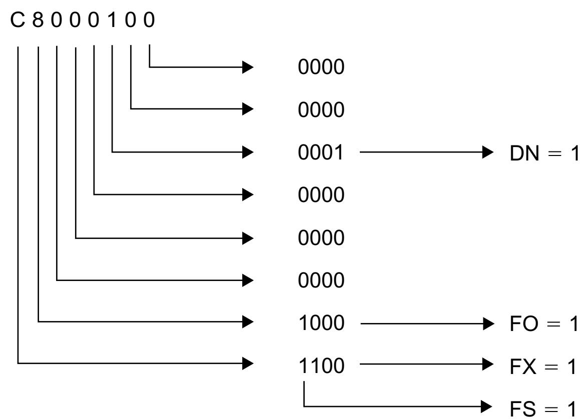


Figure 3.17 Example 1; FPSW.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 58

From the above example:

- DN = 1, where a denormalized number is handled as 0.
- FO = 1, where an overflow exception has occurred.
- FX = 1, where an inexact exception has been generated.
- FS = 1, where the logical OR of the exception flags, except inexact exception, is 1; meaning an overflow/underflow/divide-by-zero/invalid operation exception flag has been set.

EXAMPLE 2

Underflow Exception

Consider the following example:

```
1. float a, c;
2. a = 0.0000000000000000000000000000000005623;
3. c = a * 0.00000000000000000000000001235431435234;
```

The above operation causes an Underflow Exception. The FPSW has the following value.

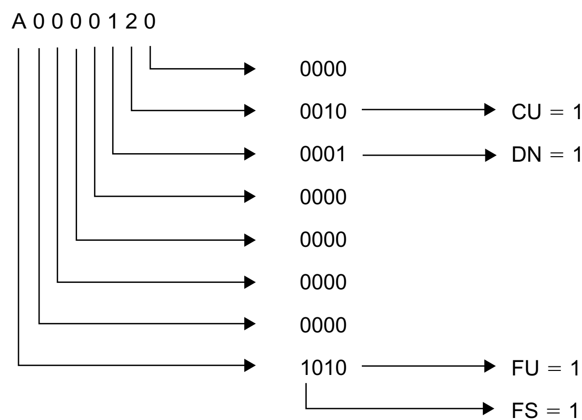


Figure 3.18 Example 2; FPSW.

From the above example:

- CU = 1, where an underflow operation has occurred.
- DN = 1, where a denormalized number is handled as 0.

- FU = 1, where an underflow exception has occurred and the underflow flag has been set.
- FS = 1, where the logical OR of the exception flags, except inexact exception, is 1; meaning an overflow/underflow/divide-by-zero/invalid operation exception flag has been set.

EXAMPLE 3

Divide-by-Zero Exception

Consider the following example:

```
1. float a, c;
2. a = 1.2;
3. c = a / 0;
```

The above code returns a divide-by-zero exception and the FPSW has the following value.

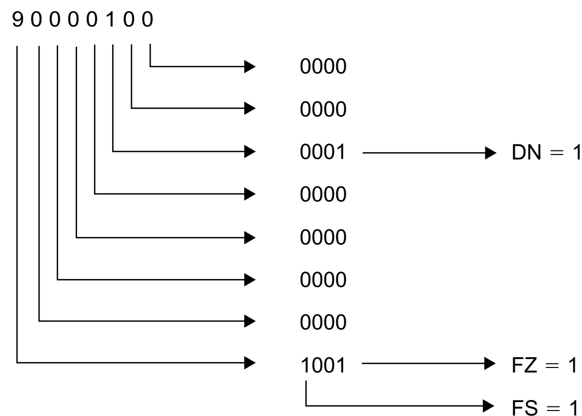


Figure 3.19 Example 3; FPSW.

From the above example:

- DN = 1, where a denormalized number is handled as 0.
- FZ = 1, where Division-by-zero flag has been set.
- FS = 1, where the Floating-point error summary flag has been set.

EXAMPLE 4

Invalid Operation

Consider the following example:

```
1. float c;
2. c = 0.0 / 0.0;
```

The above code returns an Invalid Operation and the FPSW has the following value.

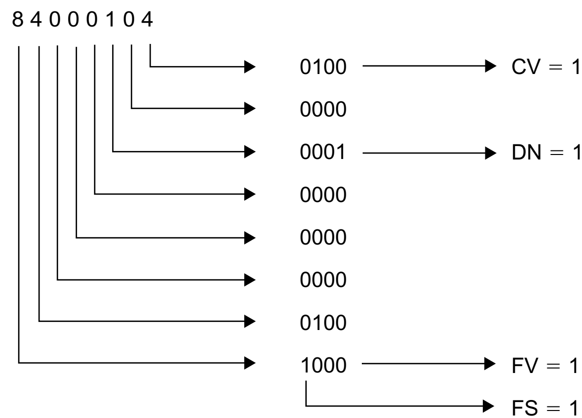


Figure 3.20 Example 4; FPSW.

From the above example:

- CV = 1, where an invalid operation has occurred.
- DN = 1, where a denormalized number is handled as 0.
- FV = 1, where an invalid operation exception has occurred and the invalid operation flag has been set.
- FS = 1, where the the floating-point error summary flag has been set.

3.4 ADVANCED CONCEPTS OF RX63N FLOATING POINT UNIT

3.4.1 FPSW in Detail

In order to perform complex floating point operations and handle floating point exceptions, a detailed knowledge about the FPSW is important. The bits in the FPSW can be categorized as follows:

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 61

Rounding-Modes

By setting the FPSW.RM[1:0] bits, the floating point value or the result can be rounded to 0, $+\infty$, $-\infty$ or the nearest value. The RM[1:0] bits represent the rounding-mode. By default, the values are rounded towards the nearest absolute value. Consider Table 3.2, for RM[1:0] bit definitions.

TABLE 3.2 RM[1:0] Bit Definitions in FPSW [2], Page 27.

BIT	SYMBOL	BIT NAME	DESCRIPTION
b1, b0	RM[1:0]	Floating-point rounding-mode setting bits	b1 b0
			0 0: Round to the nearest value
			0 1: Round towards 0
			1 0: Round towards $+\infty$
			1 1: Round towards $-\infty$

- Round to the nearest value: With this rounding-mode, the inexact value or the inexact result is rounded to the nearest absolute value. If two absolute values are equally close, the result is the one with the even alternative.
- Round towards 0: With this rounding mode, the inexact value or the inexact result is rounded to the smallest available absolute value in the direction of 0.
- Round towards $+\infty$: The inexact value is rounded to the nearest available value in the direction of $+\infty$.
- Round towards $-\infty$: The inexact value is rounded to the nearest available value in the direction of $-\infty$.

Cause Flags

In order to handle the floating point exceptions and floating point exceptions that are generated upon detection of unimplemented processing, the cause flags are used. When a cause flag is set to 1, it means a corresponding exception has occurred. The bit that has been set to 1 is cleared to 0 when an FPU instruction is executed.

The Cause Flags in the FPSW are:

- Invalid Operation Exception Cause Flag (CV)
- Overflow Exception Cause Flag (CO)

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 62

- Underflow Exception Cause Flag (CU)
- Divide-By-Zero Exception Cause Flag (CZ)
- Inexact Exception Cause Flag (CX)
- Unimplemented Processing Cause Flag (CE)

Exception Flags

The Exception Flags are used to indicate the occurrence of exceptions. If the exception flag is set to 1, it means the corresponding error has occurred. The exception flags available in the FPSW are:

- Invalid Operation Flag (FV)
- Overflow Flag (FO)
- Underflow Flag (FU)
- Division-by-zero flag (FZ)
- Inexact Flag (FX)

Exception Handling Enable Bits

When any of the five floating point exceptions occur, the bit decides whether the CPU will start handling the exception. When the bit is set to 0, the exception handling is masked. When the bit is set to 1, exception handling is enabled.

The Exception Handling bits in the FPSW are:

- Invalid Operation Exception Enable bit (EV)
- Overflow Exception Enable bit (EO)
- Underflow Exception Enable bit (EU)
- Divide-by-zero Exception Enable bit (EZ)
- Inexact Exception Enable bit (EX)

Denormalized Number Bit

A denormalized number is a number of the form $1 \times \alpha^{-n}$, where α is the base of the number (2 for decimal numbers and 10 for binary numbers) and n is the exponent.

When the denormalized number bit (DN) is set to 1, a denormalized number is handled as 0. When the DN bit is set to 0, the denormalized number is handled as a denormalized number.

From the basic examples, it can be seen that the DN bit is set to 1 when an Overflow, Underflow, Invalid Operation, or a Divide-by-zero exception occurs. This denotes that the result of the operation cannot be handled as a denormalized number.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 63

Floating Point Error Summary Flag

The Floating Point Error Summary flag (FS) bit represents the logical OR of the following Exception Enable bits:

- Invalid Operation Exception
- Overflow Exception
- Underflow Exception
- Divide-by-Zero Exception

Since the occurrence of inexact operation is common and is taken care of by the RM[1:0] bits under certain circumstances, the FS bit does not reflect the occurrence of an inexact operation.

Reserved Bits

The reserved bits are restricted for the programmer, as they are used by the microcontroller itself. The reserved bits have a value of 0.

3.4.2 Floating Point Exception Handling Procedure

Though the floating-point exceptions appear similar to other exceptions, the handling routine for the floating-point exceptions vary slightly from that of other instructions. Floating-point exceptions are of the lowest priority and handling the exceptions can be classified as follows:

- Acceptance of the exception
- Hardware pre-processing
- Processing of the user-written code
- Hardware post-processing

Acceptance of the Exception

When a floating-point exception occurs, the CPU suspends the execution of the current instruction and stores the PC value of the instruction that is generated by the exception on the Stack. It is for this reason; the floating-point instruction is of instruction canceling type.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 64

Hardware Pre-Processing

The following steps take place during hardware pre-processing:

- Preserving the PSW
  The value of the PSW is stored in the stack.
  PSW → Stack
- Updating the PSW
  The following bits are updated:
  - I → 0
  - U → 0
  - PM → 0
- Preserving the PC
  The value of the PC is stored in the stack.
  PC → Stack
- Updating the PC
  The PC is updated with branch-destination address fetched from the vector address FFFFFFE4h in the fixed vector table. The processing is then shifted to the exception handling routine.

Processing of User-Written Program Code

The following steps take place while processing user-written code.

- Preserving general purpose registers
  The contents of the general purpose registers are stored in the stack.
  General purpose registers → Stack
- User code execution
  Corresponding user code is executed. In general, the code is inserted in the Excep_FloatingPoint() interrupt/function.
- General Purpose registers restoration
  The contents of the general purpose registers are stored in the stack.
  Stack → General purpose registers
- RTE Execution
  After restoring the contents of the general-purpose registers, the Return from Exception (RTE) is executed. This marks the end of exception handling. Following this step will be hardware post-processing.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 65

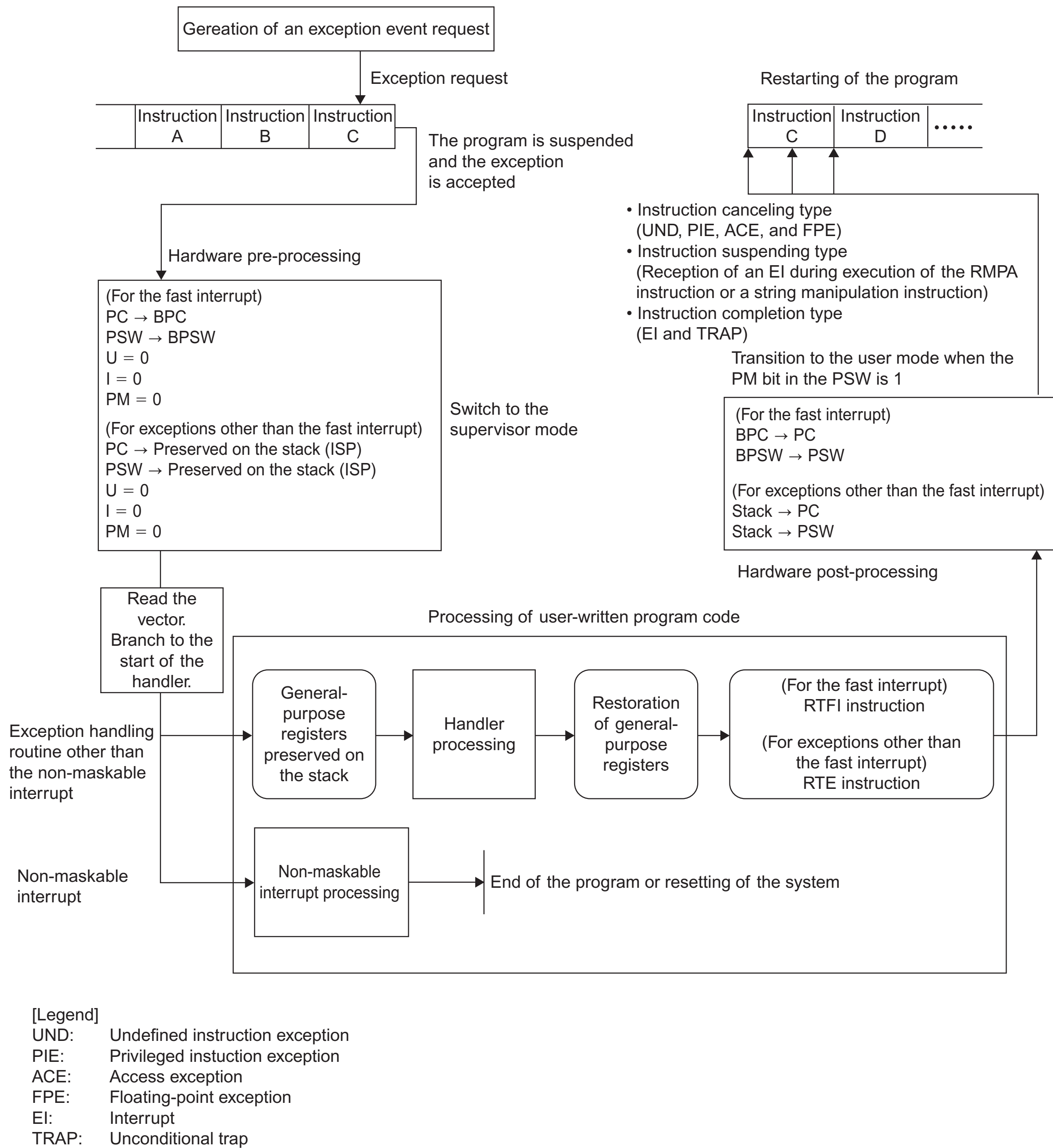


Figure 3.21 Exception handling procedure [1], Page 328.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 66

Hardware Post-Processing

The following steps take place during hardware post-processing:

- Restoring PC
  The contents of the PC are restored from the Stack.
  Stack → PC.
- Restoring PSW
  The contents of the PC are restored from the Stack.
  Stack → PC.

Following the exception handling routine, the program is restarted. Figure 3.21 explains the exception handling procedure.

3.5 ADVANCED EXAMPLES

3.5.1 Fixed-Point and Floating-Point Operation Time Calculation

This example calculates the time taken for a single fixed-point operation and the time taken for a single floating point operation based on switch presses. For example, if Switch 1 is pressed, an Integer operation is carried out and the time taken for the operation is displayed.

In order to calculate the operation time, we need a timer. Let's say we are using timer 0 (TMR0), with a clock source of PCLK/2 (to be more precise). In order to calculate the time taken, we initialize the timer and reset the Timer Counter (TCNT). The following would be the function to initialize the timer.

```
 1. void  Init_Timer() {
 2.    MSTP(TMR0) = 0;
 3.    TMR0.TCR.BYTE = 0x40;
 4.    TMR1.TCR.BYTE = 0x00;
 5.    TMR01.TCCR = 0x180E;
 6.    TMR0.TCSR.BYTE = 0xE0;
 7.    TMR1.TCSR.BYTE = 0xE0;
 8.    TMR01.TCORA = 48000000 / 8192;
 9.    TMR01.TCNT = 0;
10.    IR (TMR0, CMIA0) = 0;
11.    IPR(TMR0, CMIA0) = 4;
12.    IEN(TMR0, CMIA0) = 1;
13. }
```

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 67

In Line 2 we are powering up the TMR unit using the appropriate Module Stop Control Register. In lines 3 through 7 various timer control registers are set to cascade TMR 0 and TMR 1, count at PCLK ÷ 8192. In line 8 Timer Constant Register A (TCORA) is set. If PCLK = 48 MHz and we are using PCLK ÷ 8192 to increment the timer then one count = 8,192 ÷ 48 = 170.667 us. One second = 48,000,000 ÷ 8,192 = 5,859.375 counts. In line 9 the Timer Counter register is cleared. In lines 10 through 12 the interrupt is reset, the interrupt priority is set and the interrupt is enabled.

We will also need a function to reset the timer, which will deactivate the timer unit and reset the timer counter.

```
1. void Reset_Timer(void) {
2.    MSTP(TMR0) = 1;   //Deactivate TMR0
3.    TMR0.TCNT = 0x00;   //Reset TCNT
4. }
```

The following code would be the main function:

```
 1. void main(void) {
 2.
 3.    char str [30];
 4.    int i;
 5.    long int int1, int2, int3;
 6.    float float1, float2, float3;
 7.    double double1, double2, double3;
 8.    int1 = 2; int2 = 3;
 9.    float1 = 2.0; float2 = 3.0;
10.    double1 = 2.0; double2 = 3.0;
11.    lcd_initialize();
12.    lcd_clear();
13.    lcd_display(LCD_LINE1, "RENESAS");
14.    lcd_display(LCD_LINE2, "YRDKRX63N");
15.    R_SWITCHES_Init();
16.
17.    while (1) {
18.
19.       if(SW1 == 0) {   //If SW1 pressed
20.          Init_Timer();
21.          for(i = 0; i < 10; i++)
22.             int3 = (int1 * int2);
23.          sprintf((char*)str, "C:%d", TMR01.TCNT);
24.          Reset_Timer();
25.          lcd_display(LCD_LINE3, str);
```

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 68


```
26.         }
27.
28.         else if(SW2 == 0) {
29.            Init_Timer();
30.            for(i = 0; i < 10; i++)
31.               float3 = (float1 * float2);
32.            sprintf((char*)str, "C:%d", TMR01.TCNT);
33.            Reset_Timer();
34.            lcd_display(LCD_LINE3, str);
35.         }
36.
37.         else if(SW3 == 0) {
38.            Init_Timer();
39.            for(i = 0; i < 10; i++)
40.               double3 = (double1 * double2);
41.            sprintf((char*)str, "C:%d", TMR01.TCNT);
42.            Reset_Timer();
43.            lcd_display(LCD_LINE3, str);
44.         }
45.
46.         else {
47.            sprintf((char *)str, "Press any Switch");
48.            lcd_display(LCD_LINE1, str);
49.         }
50      }
51. }
```

In the previous code, consider the snippet from Line 19 to Line 26:

```
19. if(SW1 == 0) { //If SW1 pressed
20.    Init_Timer();
21.    for(i = 0; i < 10; i++)
22.       int3 = (int1 * int2);
23.    sprintf((char*)str, "C:%d", TMR01.TCNT);
24.    Reset_Timer();
25.    lcd_display(LCD_LINE3, str);
26. }
```

Line 20 initiates the timer and starts counting. Line 21 and line 22 perform integer multiplication for ten iterations. Since the time taken for iteration would be less, we consider ten iterations and divide the result by 10. Similar to the above code snippet, the multiplication operation is performed for float and double, and the value of the counter is found.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 69

From the above code, it was found that the operations performed take the following time:

Int

Timer Count = 80

$$\text{Time Taken} = \frac{\text{Timer Count}}{\left(10 \times \frac{\text{PCLK}}{2}\right)}$$

Since we use $\frac{\text{PCLK}}{2}$ clock and we perform ten iterations, we divide the timer count by 10 and $\frac{\text{PCLK}}{2}$ (PCLK = 16.5 MHz here).

Therefore,

$$\text{Time Taken} = \frac{80}{\left(10 \times \frac{16.5}{2}\right)} = 0.9696\ \mu\text{s}$$

Float

Timer Count = 90

Therefore,

$$\text{Time Taken} = \frac{90}{\left(10 \times \frac{16.5}{2}\right)} = 1.0909\ \mu\text{s}$$

Double

Timer Count = 90

Therefore,

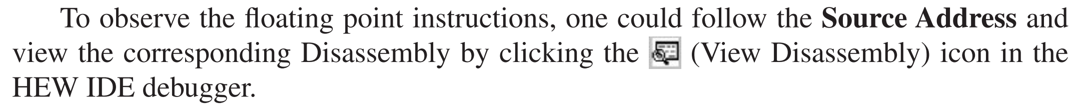
$$\text{Time Taken} = \frac{90}{\left(10 \times \frac{16.5}{2}\right)} = 1.0909\ \mu\text{s}$$

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 70

From the above example, the time taken for a floating point operation in the RX63N processor was found to be 1.0909 µs.

To observe the floating point instructions, one could follow the **Source Address** and view the corresponding Disassembly by clicking the (View Disassembly) icon in the HEW IDE debugger.

For example, consider the following figure.



266	FFFFCFFA			f(i – 0; i < 10; i++)
227	FFFFD006			float3 = (float1 * float2);

Figure 3.22 Code snippet in view source tab.

The above code generates the following assembly code after building/compilation:

FFFFD006	ED0E0C		MOV.L	30H[R0],R14
FFFFD009	AB0D		MOV.L	34H[R0],R5
FFFFD00B	FC8FF5E		FMUL	R5,R14
FFFFD00E	E70E0E		MOV.L	R14,38H[R0]
FFFFD011	ED0E08		MOV.L	20H[R0],R14
FFFFD014	621E		ADD	#1H,R14
FFFFD016	E70E08		MOV.L	R14,20H[R0]
FFFFD019	ED0E08		MOV.L	20H[R0],R14
FFFFD01C	61AE		CMP	#0AH,R14
FFFFD01E	29E8		BLT.B	0FFFFD006H

Figure 3.23 Assembly code in view disassembly tab.

3.5.2 Matrix Multiplication Time Calculation

Similar to the above example, let's examine the time taken to calculate a simple 3×3 matrix.

In order to choose between int, float, and double, let's use the following code for user-friendliness:

```
1. void Init_Display() {
2.
3.    char str [30];
4.    sprintf((char *)str, "Matrix Mult");
5.    lcd_display(LCD_LINE1, str);
6.    sprintf((char *)str, "SW1: Int");
7.    lcd_display(LCD_LINE2,str);
```

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 71


```
 8.    sprintf((char *)str, "SW2: Float");
 9.    lcd_display(LCD_LINE3,str);
10.    sprintf((char *)str, "SW3: Double");
11.    lcd_display(LCD_LINE4,str);
12. }
```

The previous code above asks the user to choose the operation to perform. Pressing SW1 performs matrix multiplication with integer variables, whereas pressing SW2 and SW3 performs matrix multiplication using single precision and double precision floating point variables. The multiply operations are defined below:

```
 1. void int_Multiply(int matA[3][3], int matB[3][3], int matC[3][3]) {
 2.    int i, j, k;
 3.    for(i = 0; i < 3; i++)
 4.       for(j = 0; j < 3; j++)
 5.          for(k = 0; k < 3; k++)
 6.             matC [i][j] += matA [i][k] * matB [k][j];
 7. }
 8.
 9. void float_Multiply(float matA[3][3], float matB[3][3], float
    matC[3][3]) {
10.    int i, j, k;
11.    for(i = 0; i < 3; i++)
12.       for(j = 0; j < 3; j++)
13.          for(k = 0; k < 3; k++)
14.             matC [i][j] += matA [i][k] * matB [k][j];
15. }
16.
17. void dbl_Multiply(double matA[3][3], double matB[3][3], double
    matC[3][3]) {
18.    int i, j, k;
19.    for(i = 0; i < 3; i++)
21.       for(j = 0; j < 3; j++)
22.          for(k = 0; k < 3; k++)
23.             matC [i][j] += matA [i][k] * matB [k][j];
24.}
```

In the previous code, lines 1 through 7 perform matrix multiplication using integers, lines 9 through 15 perform matrix multiplication using single precision floating-point numbers, and lines 17 through 24 perform matrix multiplication using double precision floating-point numbers.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 72

We use the Init_Timer() and Reset_Timer() functions discussed in the previous example to activate and deactivate the timer unit.

The main function would be as follows:

```
 1. void main(void) {
 2.
 3.    char str [30];
 4.    int i, j;
 5.    int Timer_Count;
 6.    float time;
 7.    int int_matA [3][3],int_matB [3][3];
 8.    int int_matC [3][3];
 9.    float float_matA [3][3],float_matB [3][3];
10.    float float_matC [3][3];
11.    double dbl_matA [3][3], dbl_matB [3][3];
12.    double dbl_matC [3][3];
13.    for(i = 0; i < 3; i++)
14.       for(j = 0; j < 3; j++)
15.          int_matA [i][j] = j * 2;
16.    for(i = 0; i < 3; i++)
17.       for(j = 0; j < 3; j++)
18.          int_matB [i][j] = (9 - j) * 2;
19.    for(i = 0; i < 3; i++)
20.       for(j = 0; j < 3; j++)
21.          float_matA [i][j] = j * 1.1;
22.    for(i = 0; i < 3; i++)
23.       for(j = 0; j < 3; j++)
24.          float_matB [i][j] = (9 - j) * 1.1;
25.    for(i = 0; i < 3; i++)
26.       for(j = 0; j < 3; j++)
27.          dbl_matA [i][j] = j * 1.1;
28.    for(i = 0; i < 3; i++)
29.       for(j = 0; j < 3; j++)
30.          dbl_matB [i][j] = (9 - j) * 1.1;
31.    lcd_initialize();
32.    lcd_clear();
33.    R_SWITCHES_Init();
34.    Init_Display();
35.
36.    while (1) {
```

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 73


```
37.
38.         if(SW1 == 0) {
39.            lcd_clear();
40.            Init_Timer();
41.            int_Multiply(int_matA, int_matB, int_matC);
42.            Timer_Count = TMR01.TCNT;
43.            Reset_Timer();
44.            sprintf((char *)str, "Count:%d", Timer_Count);
45.            lcd_display(LCD_LINE3, str);
46.            Timer_Count = 0;
47.         }
48.
49.         if(SW2 == 0) {
50.            lcd_clear();
51.            Init_Timer();
52.            float_Multiply(float_matA, float_matB, float_matC);
53.            Timer_Count = TMR01.TCNT;
54.            Reset_Timer();
55.            sprintf((char *)str, "Count:%d", Timer_Count);
56.            lcd_display(LCD_LINE3,str);
57.            Timer_Count = 0;
58.         }
59.
60.         if(SW3 == 0) {
61.            lcd_clear();
62.            Init_Timer();
63.            dbl_Multiply(dbl_matA, dbl_matB, dbl_matC);
64.            Timer_Count = TMR01.TCNT;
65.            Reset_Timer();
66.            sprintf((char *)str, "Count:%d", Timer_Count);
67.            lcd_display(LCD_LINE3, str);
68.            Timer_Count = 0;
69.         }
70.      }
71. }
```

In the above code lines 13 through 30 initialize the matrices in corresponding data-type. When switch 1 is pressed, the code enters line 38. The integer matrix multiplication is performed at line 41 and the corresponding count is measured as the previous example.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 74

From the previous example, the following results were observed:

Int

Timer Count = 112

$$\text{Time Taken} = \frac{111}{\left(10 \times \frac{16.5}{2}\right)} = 1.3454\ \mu\text{s}$$

Float

Timer Count = 197

$$\text{Time Taken} = \frac{197}{\left(10 \times \frac{16.5}{2}\right)} = 2.3878\ \mu\text{s}$$

Double

Timer Count = 198

$$\text{Time Taken} = \frac{198}{\left(10 \times \frac{16.5}{2}\right)} = 2.4\ \mu\text{s}$$

From the example, time taken to perform matrix multiplication of two 3 × 3 matrices was found. Also, a slight difference in time taken to perform single precision and double precision floating point matrix multiplication operation was observed.

3.6 RECAP

In this chapter, the Floating Point Unit (FPU) used in the Renesas RX63N was analyzed in detail, where the floating point instructions, floating point exceptions and floating point registers were discussed.

By applying this knowledge about the FPU, the RX63N board can be used in signal processing, image processing applications, and applications that require precise round-off

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 75

(for example, in applications involving currency, floating-point rounding off errors should be considered in order to value currency as $$.cc where “$$” denotes the dollar value and “cc” denotes the cent value) with an understanding of the time-memory tradeoff between a microcontroller with an FPU and one without an FPU.

3.7 REFERENCES

[1] Renesas Electronics, Inc., *RX63N Group, RX631 Group User’s Manual: Hardware, Rev 1.60,* February 2013.

[2] Renesas Electronics, Inc., *RX Family User’s Manual: Software, Rev 1.20,* April 2013.

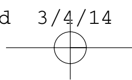
3.8 EXERCISES

1. Write a C program to identify different Floating Point Exceptions.
2. Consider the following operations:

   Addition: 2.0 + 3.0
   Multiplication: 2.0 * 3.0
   Division: 2.0/3.0

   Do different operations involving floating-point (addition, multiplication, and division) take the same time to complete? Justify your answer. Use single precision floating point numbers for the operations. Use Switch 1 (SW1), Switch 2 (SW2), and Switch 3 (SW3) to perform Addition, Multiplication, and Division respectively and display the time taken as the result.
3. Write a C program on the RX63N to calculate the time taken to perform the operations stated in Exercise 2. Use single precision floating point numbers for the operations.
4. Implement Exercise 3 using double precision floating point numbers. Does it take the same time as single precision floating point numbers? Justify your answer.
5. Identify the floating point instructions involved in single precision and double precision operations in Exercises 3 and 4 from the disassembly. Do they look alike? Explain your answer.

03.ES_Conrad_RX63N_Advanced_CH03.qxd:RX63N Advanced 3/4/14 12:03 PM Page 76

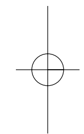


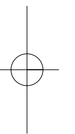
6. Write a C program on the RX63N to calculate the time taken to perform the inverse of the given matrix A.

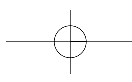
$$A = \begin{matrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 0 & -2 \end{matrix}$$

7. Multiply the inverted A matrix with B matrix given below. Identify the floating point exceptions involved.

$$B = \begin{matrix} 0.001 & 2.400 & 0.120 \\ 3.143 & 2.743 & 4.901 \\ 0.917 & -0.008 & -2.001 \end{matrix}$$







Advanced Operating System Usage

4.1 LEARNING OBJECTIVES

In this chapter the reader will learn:

- Multitasking, re-entrant functions
- Semaphores
- Message passing and memory management

4.2 BASIC CONCEPTS OF OPERATING SYSTEMS

4.2.1 Multitasking, Re-entrant Functions

The capacity and performance of today's embedded systems have been increasing to match the expanding requirements assigned to them. For example, smartphones perform tasks once reserved only for general purpose computers. Most tasks that embedded systems are expected to perform need to be done at the same time. For example, an embedded computer that controls an autonomous robot is expected to read the sensor data, as well as send control data to the servos at the same time. How is this achieved?

Multitasking

Multitasking has long been a common feature of general purpose computing systems. As the speed and memory of embedded processors has increased, it has become common to find multitasking in today's embedded systems.

When an operating system can run multiple tasks concurrently then it is said to be multitasking. Advanced operating systems like Micriμm's MicroC/OS-III or FreeRTOS provide the user the ability to multitask. The ability of an operating system to multitask provides the designer with numerous of benefits. Complex tasks can be broken down into a

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 78

large number of small tasks which can be run in parallel. When multitasking is handled by the operating system the designer is free of implementing various timing and synchronization logic, and can concentrate more on the core application logic.

When tasks execute simultaneously, new types of problems arise. One major problem is shared data. One way of addressing this problem is using reentrant functions. A reentrant function has three major characteristics:

1. A reentrant function may not use variables in a non-atomic way, unless they are stored on the stack of the calling task, or are the private variables of that task.
2. A reentrant function may not call other functions that are not reentrant.
3. A reentrant function may not use the hardware in a non-atomic way.

FreeRTOS [1, 2] ships with many reentrant functions and libraries (it even includes third party versions of common functions such as printf/sprint that are very light and reentrant in nature). When the user includes libraries or functions that are not reentrant in nature, care must be taken such that protection is provided using mutexes and semaphores.

4.2.2 Semaphores

The FreeRTOS kernel uses semaphores to solve the problem of shared memory. Two types of semaphores are available: binary and counting. Binary semaphores take two values, 0 and 1; whereas counting semaphores can take a value between 0 to 2^N-1, where N is the number of bits used to address the semaphore.

Semaphore API functions permit the block time to be specified. This block time indicates the maximum number of 'ticks' that a task should wait before entering the blocked state when the semaphore is not immediately available. If more than one task waits on the same semaphore, then the task with the highest priority is the task that is unblocked the next time the semaphore becomes available.

Think of a binary semaphore as a queue that can only hold one item. Therefore the queue can only be empty or full (hence binary). Tasks and interrupts using the queue do not care what the queue holds. They only want to know if the queue is empty or full. This mechanism can be exploited to synchronize. For example, it can be used for a task with an interrupt.

Several API's are available to manage semaphores in the FreeRTOS's kernel; they have the following general format:

```
vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore )
```

This macro creates a semaphore by using the existing queue mechanism. The queue length is 1, as this is a binary semaphore. The data size is 0 since no data is stored.

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 79

4.2.3 Task Communication, Synchronization, and Memory Management

When tasks are concurrently running in an operating system, synchronization and communication between them is very important. The FreeRTOS uses the concept of queues to achieve synchronization.

Queues are the primary form of inter-task communication. They can be used to send messages between tasks, and between interrupts and tasks. In most cases they are used as thread safe FIFO (First In First Out) buffers, with new data being sent to the back of the queue, although data can also be sent to the front of the queue.

The FreeRTOS queue usage model manages to combine simplicity with flexibility, which are attributes that are normally mutually exclusive.

The RTOS kernel allocates RAM each time a task, queue, mutex, software timer, or semaphore is created. The standard C library malloc() and free() functions can sometimes be used for this purpose, but do have the following short comings:

- They are not always available on embedded systems.
- They take up valuable code space.
- They are not thread safe.
- They are not deterministic (the amount of time taken to execute the function will differ from call to call).

Hence an alternative memory allocation implementation is required.

An embedded/real time system can have very different RAM and timing requirements. A single RAM allocation algorithm is only appropriate for a subset of applications.

To avoid problems, FreeRTOS keeps the memory allocation API in its portable layer. The portable layer is outside of the source files that implement the core RTOS functionality, thus allowing an application that is specific and appropriate for the real time system being developed. When the RTOS kernel requires RAM, instead of calling malloc() it calls pvPortMalloc(). When RAM is being freed, instead of calling free(), the RTOS kernel calls vPortFree().

4.3 BASIC EXAMPLES

4.3.1 Example 1

Task structure of a task in FreeRTOS is as follows:

```
1. void vATaskFunction( void *pvParameters ) {
2.     for( ;; ) {
          //Task application code here
3.     }
4. }
```

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 80

4.3.2 Example 2

```
1. xSemaphoreHandle xSemaphoreCreateCounting(
2.     unsigned portBASE_TYPE uxMaxCount,
3.     unsigned portBASE_TYPE uxInitialCount
4. );
```

The previous example shows a macro that creates a counting semaphore by using the existing queue mechanism.

4.4 ADVANCED CONCEPTS OF OPERATING SYSTEM USAGE

4.4.1 Getting Started

The RX600 Series of microcontrollers is ideally suited for several embedded system designs. The increasing complexity of these designs provides a need for intelligently managing the resources available. An infinite while loop might seem feasible for an application that is small, such as communicating between an accelerometer that is connected to the board by way of the UART. But when a complex application is considered, such as implementing a webserver on the RX63N, having an operating system to provide abstraction is very useful.

When an application has real time requirements, the need for a real time kernel becomes apparent. Real time requirements can be divided into two types:

- **Soft real time requirements:** These requirements have a deadline, but missing this deadline does not result in catastrophic results. For example, dropping a few frames during a web conference would decrease the quality of the rendered video but would not crash the application.
- **Hard real time requirements:** These requirements have a deadline and missing this deadline would cause the application to fail. For example the hardware in the cruise control of a missile should have accurate timing to avoid catastrophic results.

Developing embedded software without the use of real time kernels is possible, and in fact seems easier and fast. But having a real time kernel provides a layer of abstraction for the programmer by handling various intricate details such as task handling, and maintaining memory consistency and coherence. As with all systems where abstraction has been provided, this kernel adds an additional overhead to the system. Hence the user should carefully evaluate whether having a real time kernel would increase the performance of the application.

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 81

FreeRTOS is a real time kernel that can be built on top of the RX63N applications to meet real time requirements. The applications can be organized with multiple independent threads of execution. The user can assign a higher priority to threads that implement hard real time requirements, and lower priorities to threads that implement soft real time requirements.

The FreeRTOS kernel has been ported to the RX600 Series of microcontrollers. The features of the port are as follows:

- Preemptive or Cooperative operation
- Flexible task priority assignment
- Binary semaphores
- Counting semaphores
- Recursive semaphores
- Mutexes
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace hook macros
- Optional commercial licensing and support

FreeRTOS uses the software interrupt available on the RX63N. This kernel is not usable by the application. It also requires the exclusive use of a timer peripheral, which is capable of generating a fast periodic interrupt. The timer to use can be defined in a user defined callback function. By default the sample applications use the compare match timer zero CMT0 to generate this interrupt.

```
void vApplicationSetupTimerInterrupt( void );
```

The above is a prototype for the callback function the kernel calls to setup the timer interrupt.

The memory footprint of FreeRTOS is very small. It usually consumes only about 6.5 K of RAM and a few hundred bytes of extra RAM. Some tasks also use RAM as the task stack. FreeRTOS is an excellent kernel for users to experiment with. One major reason is that it is open source. Because it is open source, any modifications made to the kernel should remain open source, although the components of the application that the user has developed can be closed source and can remain intellectual property.

The example zip files showing how to execute a sample application on the RX600 Series can be found at: http://www.freertos.org/Documentation/code/

The application can be loaded onto the RX63N board using the Renesas compiler and the HEW application.

4.4.2 Task Management

Tasks in the FreeRTOS kernel are usually implemented as C functions. These tasks should return void and must take a void pointer parameter. The prototype is usually as follows:

```
void task( void  *pvparameters );
```

The task can be considered a small program. Once execution of the task begins, it executes forever. The following is an example of how a typical task structure would look:

```
 1. void task(void *pvparameters) {
 2.    // The variables of a task can be declared as any
 3.    // normal variables each time the task is called the
 4.    // function will have its own copy of variables
 5.    int a,b;
 6.    while(1) {
 7.       //C ode to specify the functionality of the task goes here.
 8.       // The tasks are usually implemented as an infinite loop
 9.    }
10.    // If the task manages to come out of the loop it should be
11.    // deleted. The null parameter passed indicates that the
12.    //particular task which has to be deleted has called this
       function
13.    vTaskDelete( NULL);
14. }
```

Task States

A task can be in several states. In the case of processers that have a single core in order to execute tasks, the task being executed by the system is said to be in active state, while the other tasks are said to be in dormant state. A task that is waiting to change from a dormant state to an active state can be considered a ready task. When a task changes state to active, it starts execution from the next instruction it was about to execute before it switched states.

Task Creation

Tasks are Created in FreeRTOS using the xTaskCreate() API. This is the most complex API in FreeRTOS and it is used extensively. The function prototype is as follows:

```
1. portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
2.    const signed char* const pcName,
```

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 83


```
3.    unsigned short usStackDepth,
4.    void *pvParameters,
5.    unsigned portBASE_TYPE uxPriority,
6.    xTaskHandle *pxCreatedTask
7. );
```

Every task is created in a ready state. The parameters within the function prototype provide information about the task being created. For example the pvTaskCode is a pointer to the code that defines the functionality of the task. A detailed explanation of all the parameters can be found in the FreeRTOS reference manual.

Task Priorities

The uxPriority parameter of the xTaskCreate() API assigns an initial priority to the task being created. The priority can be changed after the FreeRTOS scheduler has been started using the vTaskPrioritySet() API. The maximum number of priorities available is user defined and can be set in the FreeRTOSconfig. h. Any number of tasks can be assigned the same priority. Lower priority values denote a lower priority, with 0 being the lowest priority.

The periodic interrupt called the Tick Interrupt is used by the scheduler to assign time slices to the tasks. The scheduler makes sure that the task with the highest priority gets the time slice. The function of API prototype is to change the task priorities is as follows:

```
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE
uxNewPriority);
```

pxTask = The handle of the task whose priority is being modified.
uxNewPriority = The priority to which the subject task is set.
The priority of the task can be queried by using the uxTaskPriorityGet() API.

Deleting a Task

A task can be deleted using the vTaskDelete() API. An interesting point to note here is that a task can delete itself using this API. Deleted tasks no longer exist and cannot enter the running state. When a task is deleted, it is the responsibility of the task to free memory. The memory allocated to the task by the kernel is freed automatically by the kernel; however, any resources allocated to the task by the task implementation code must be explicitly freed by the task.

The function prototype for the API to delete tasks is as follows.

```
1. void vTaskDelete( xTaskHandle pxTaskToDelete )
```

pxTaskToDelete = The handle of the task to be deleted.

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 84

Task Scheduling

FreeRTOS provides prioritized pre-emptive scheduling as well as co-operative scheduling. The features of fixed prioritized preemptive scheduling are as follows:

- Each task is assigned a priority.
- Each task can exist in one of the several states.
- Only one task can exist in the running state.
- The scheduler always selects the task in the highest priority ready state to enter the running state.

In this type of scheduling the kernel cannot change the priority of the tasks once they are assigned. The tasks can change once the priorities are assigned to them.

4.4.3 Queue Management

Applications using the FreeRTOS kernel can be developed as independent tasks, which are essentially small programs that run forever. For the application to be useful, these tasks must communicate with each other. The main method of communication between these tasks is the queue.

A Queue is a data storage mechanism that can hold a finite number of fixed size elements. The length of the queue defines the amount of data that can be stored. Usually queues are FIFO in nature. The queues are entities in their own right and are not owned by anyone. Multiple tasks can write to and read from the queue.

Queue Reads

Let us consider two tasks, A and B, which need to communicate with each other. This happens in FreeRTOS by the means of queues. Task A attempts to get data, which it requires for its operation, by reading the data from the queue. The task specifies a block time; i.e., the maximum amount of time it will wait before it gets data from the queue (in case the data is not available in the queue). The task waiting for data from the queue is put in the blocked state by the scheduler and goes back to ready state when the data is available in the queue.

Multiple tasks may attempt to read data from the queue; however, the scheduler unblocks only one task when data becomes available. The task that has the highest priority is unblocked. In scenarios where multiple tasks with the same priority are waiting on the queue, the task that has been waiting for the longest amount of time is unblocked.

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 85

If the data is not available by the end of the block time, the task is automatically set to ready state by the scheduler.

Queue Writes

Writes to the queue are similar to the reads form the queue. Any task that attempts to write to the queue specifies its own block time in case the queue is not empty. Multiple tasks may attempt to write to the queue. The scheduler unblocks the task that has the highest priority when there are multiple tasks with the same priority. The task that has been waiting for the longest of time is unblocked first.

Creating a Queue

Queues in FreeRTOS must be created before they can be used. Queues can be created using the xQueueCreate() API command. This command creates queues with an xQueueHandle queue type and return value. RAM is allocated by FreeRTOS when a QUEUE is created from the FreeRTOS heap. This RAM is used to store the queue data structures as well as items in the queue. If no space is allocated, then the API will return null.

```
1. xQueueHandle xQueueCreate ( unsigned portBASE_TYPE uxQueueLength,
2.    unsigned portBASE_TYPE uxItemSize
3. );
```

Where uxQueueLength = Maximumm number of items the queue can hold at ny time

uxitemSize = The size in bytes of each data item that can be stored in the queue

FreeRTOS provides API's to move the data into the queue. The command xQueueSendToBack() is used to send data to the back (tail) of a queue, and the command xQueueSendToFront()to send data to the front of a queue.

Other API's that are useful when dealing with queues are as follows:

- xTicksToWait(): The maximum amount of time to wait in the blocked state can be defined using this API.
- xQueueRecieve(): This API is used to receive an item from the queue and the item received from the queue is deleted from the queue.
- xQueuePeek(): This API is used to remove an item from the queue without the item being removed from the queue.
- uxQueueMessagesWaiting(): This API is used to query the number of items that are currently in the queue.

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 86

Compound Types

When working with queues, it is common for a task to receive data from multiple sources on a single queue. For example, an embedded system may have a task that keeps track of the health of the embedded system. This task has to receive data from various other tasks that control various peripherals on the embedded system. Assuming all the tasks use a single queue to communicate with each other in these cases, the receiver tasks must know which queue items came from which task in order to determine the health of that particular peripheral. In order for the receiver to know which task it is receiving, the data structure of the queue contains the value of the data as well as the origin of the data.

The following example shows how a queue holds structures of type xData. The structure in this example contains data values as well as a code indicating where the data originated.

```
1. typedef struct {
2.    int data;
3.    int origincode;
4. } xData;
```

In the previous example, the variable data signifies the data the peripheral intends to transfer, whereas the origin code specifies which peripheral is transferring the data.

When a huge amount of data needs to be transferred using queues, the pointers to the data are transferred instead of the actual data. Care must be taken in this case so that that the RAM the pointers point to is valid and consistent.

4.4.4 Interrupt Management

Interrupts form a crucial part of any non-trivial embedded system. Usually several peripherals are connected to the embedded system in order to transfer data to the main code; in other words, the code that is running the primary application of the embedded system by way of interrupts.

How much code should be placed in the ISR and how much should be placed in the main code are two issues that should be considered while dealing with interrupts. FreeRTOS provides API's that can be used to implement the strategy chosen in a simple and efficient manner.

The RX63N architecture and the FreeRTOS port allows the user to write ISR's entirely in C, even when the ISR wants a context switch. The exact syntax of the ISR depends on the compiler being used. Another major issue is synchronization between the tasks and ISR's.

Binary Semaphores for Synchronization

Binary semaphores can be used to unblock a task each time an interrupt related to that task occurs. Thus, effective synchronization between the task and the interrupt is maintained. The majority of the code handling the interrupt can be handled by the task, while only a small portion of the code, which is fast, can be placed inside the ISR. One more advantage of using this approach is that if the interrupt belongs to a critical operation, the priority of the task that handles the interrupt code can be increased so that it is effectively serviced. The context switch can be included inside the ISR so that the ISR returns directly to the handler task once it completes execution.

Before a semaphore is used it must be created. The following API is used to create the binary semaphore:

void vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore)
xSemaphore = The semaphore actually being created.

Other important API's while dealing with semaphores are the following:

xSemaphoreTake() : This semaphore can be used to take an available semaphore.
xTicksToWait(); This semaphore specifies the amount of time the semaphore should be put in blocked state while waiting for the semaphore.
xSemaphoreGiveFromISR(): This API can be used to give back a semaphore.

One issue with using binary semaphores is that binary semaphores can latch, only one interrupt event at a time. Any events that occurred before the latched event is processed are lost. This problem can be avoided by using a counting semaphore instead of a binary semaphore. When the event handler uses a semaphore each time an event occurs, the value of the semaphore is incremented and the handler task takes a semaphore each time it processes the event. This addition decrements the semaphore and each counting semaphore has to be created with an initial value.

As shown by the following example, counting semaphores should be created with an initial value. The API can be used to create counting semaphores.

```
1. xSemaphoreHandle xSemaphoreCreateCounting(
2.    unsigned portBASE_TYPE uxMaxCount,
3.    unsigned powerbase_TYPE uxIinitialCount
4. );
```

uxMaxCount—The maximum value the semaphore will count to.
uxInitialCount—The initial count value of the semaphore after it has been created.

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 88

4.5 COMPLEX EXAMPLES

4.5.1 Task Management

The following example illustrates how to define and create tasks in FreeRTOS.

```
1. void task1(void *pvParameters) {
2.    for(; ;)   //infinite loop {
3.       const char *name = "This is task1\n";
4.       int i;
5.       vprintString(name)
6.       for(i = 0;i < main_delay_loop_count; i ++)
7.          {}   //a simple delay implementation
8.    }
9. }
```

The previous example shows the implementation of a task in FreeRTOS. The following example shows an example of two tasks.

```
10. void task2(void *pvParameters) {
11.    for(; ;) {   //infinite loop.
12.       const char *name = "This is task2\n";
13.       int i;
14.       vprintString(name)
15.       for(i = 0; i < main_delay_loop_count;i ++)
16.       {}   //a simple delay implementation
17.    }
18. }
```

A simple implementation of the second task is shown in the previous example. These tasks can be created from the main code of the program as follows:

```
19. int main(void) {
20.    xTaskCreate(task1,          //pointer to the task
21.       "Task1 being created",   //text info about the task for
                                     debugging
22.       120,                     //stack depth in words
23.       NULL,                    //there are no task parameters
24.       1,                       //The priority of the task is 1
25.       NULL                     //task handle is not being used.
26.       );
27.    xTaskCreate(task2, "Task2 being created", 120, NULL, 1, NULL);
```

```
28.    vTaskStartScheduler();   //start the task scheduler.
29.    for( ; ;);
30. }
```

4.5.2 Queue Management

The following example demonstrates how to create a queue and then send and receive data from the queue:

```
 1. static void sender( void *para) {
 2.    int val;
 3.    portBASE_TYPE xStatus;
 4.    val = (int) para;
 5.    while(1) {
 6.       xStatus = xQueueSendToBack(xQueue,
          //queue to which data is sent
 7.          &val,   //Address of the data that is being sent.
10.          0);   //A block time of zero is being specified as the queue
                   //does not contain more than one element.
11.       if(xStatus ! = pdPASS) {   //This means that the send operation
              //could not be completed as the queue is full.
12.           vPrintString ("send operation failed\n");
13.       }
14.       taskYIELD();   //allow other tasks to execute.
15.    }
16. }
17. static void receiver ( void *para) {
18.    int recval;
19.    portBASE_TYPE xStatus;
20.    const portTickType xTicksToWait = 150;
       //block time for the task
21.    while(1) {
22.       if(uxQueueMessagesWaiting( xQueue)!=0) {
          //check-is the queue empty?
23.          vPrintString ("Queue not empty\n");
24.       }
25.       xStatus = xQueueRecieve( xQueue,
          //The queue to send the data
26.          &recval,   // address to store the received value
27.          xTicksToWait );   //block time
28.       if(xStatus == pdPASS) {
```

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 90


```
29.          vPrintStringAndNumber ("Received data is", recvalue);
30.       }
31.       else {
32.          vPrintString ("Reception error");
33.       }
34.    }
35. }
36. xQueueHandle xQueue;
37. int main(void) {
38.    xQueue = xQueueCreate(10, sizeof(int));   //create queue w/max
       of 10 vals
39.    if(xQueue!= NULL) {
40.       //create an instance of sender task that writes a value 30
41.       //continuously to the queue.
42.       xTaskCreate(sender, "sender task1",120, (void*) 30,1,NULL);
43.       //create an instance of sender task that writes a value 60
44.       //continuously to the queue.
45.       xTaskCreate(sender, "sender task2",120, (void*) 60,1,NULL);
46.       //create a receiver task.
47.       xTaskCreate(Receiver, "receiver task",120,NULL,2,NULL);
48.       vTaskStartScheduler();
49.    }
50.    while(1);
51. }
```

4.5.3 Interrupt Management

The following example illustrates how to create a binary semaphore and use it to handle an interrupt using FreeRTOS. For simplicity a simple periodic task is used to generate a software generated interrupt.

```
 1. static void ptask( void *para) {
 2.    while(1) {
 3.       //manually simulating the interrupt periodically.
 4.       vTaskDelay( 500/portTICK_RATE_MS);
 5.       mainTRIGGER_INTERRUPT();
 6.    }
 7. }
 8. static void handler (void *para) {
 9.    //create the semaphore that’s been already defined.
```

04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 91


```
10.     xSemaphoreTake( xBinarySemaphore,0);
11.     while(1) {
12.        //use the semaphore to wait for the event
13.        xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
14.        vPrintString ("Handling the interrupt");   //process the
                                                       event
15.    }
16. }
```

The interrupt ISR is written using the #pragma method which is a directive to the compiler instructing it to build interrupthandler() as an interrupt handler and install it in the relevant position of the interrupt vector table.

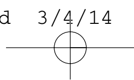
```
14. #pragma interrupt (interrupthandler(vect = _VECT(_CMT_CMT1 ),
    enable))
15. static void interrupthandler(void) {
16.    port_BASE_TYPE  xHigherPriorityTaskWoken = pdFALSE;
17.    //give the semaphore back to unblock the task.
18.    xSemaphoreGiveFromISR(xBinarySemaphore,
       &xHigherPriorityTaskWoken);
19.    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
20. }
21. int main(void) {
22.    vSemaphoreCreateBinary( xBinarySemaphore);
23.    if(xBinarySemaphore! = NULL) {
24.       //enable the peripheral interrupt used by
          mainTRIGGER_INTERRUPT();
25.       rvSetupSoftwareInterrupt();
26.       xTaskCreate(handler, "handler task", 120,NULL,3,NULL);
27.       xTaskCreate(ptask, "periodic task", 120,NULL,1,NULL);
28.       vTaskStartScheduler();
29.    }
30.    while(1);
31. }
```

4.6 REFERENCES

[1] Barry, R. (2009). *Using the FreeRTOS real time kernel: a practical guide.* Holborn, London: Real Time Engineers Limited.

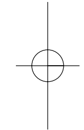
[2] http://www.freertos.org

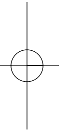
04.ES_Conrad_RX63N_Advanced_CH04.qxd:RX63N Advanced 3/4/14 8:19 PM Page 92

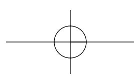


4.7 EXERCISES

1. Describe the differences between co-operative scheduling and pre-emptive scheduling.
2. What kind of embedded applications usually require an operating system? Give an example.
3. Give one method that FreeRTOS provides by means of which tasks can communicate with each other.
4. Give one way of addressing the problem of shared memory.
5. Give two disadvantages of using the C library functions malloc() and free().
6. Define soft real time requirements
7. Give an example of a hard real time requirement.
8. Explain how the priority of a task can be modified in FreeRTOS.
9. Explain the concept of queues in FreeRTOS.
10. What is the main issue that needs to be addressed while dealing with interrupts in embedded systems?
11. Give one method of synchronization that can be used to synchronize interrupts.







Digital Signal Processing

5.1 LEARNING OBJECTIVES

In this chapter the reader will learn:

- how signal conditioning and signal processing play an important role in embedded applications. The RX DSP library has been introduced to reduce the development time for the users, as the user need not build the filter from scratch but instead would use the library functions.
- the type of functions available, data types supported, and naming convention of the functions.
- initializing the DSP type hardware features that are available and calling the RX DSP library functions.

5.2 BASIC CONCEPTS OF DIGITAL SIGNAL PROCESSING

Signal conditioning and signal processing play an important role in most embedded applications where analog values from sensors, microphones, etc. are used. In these embedded applications, digital signal processing is used to measure, filter, and/or compress continuous analog signals. The first step is usually to convert the signal from an analog to a digital form by sampling and then digitizing it using an analog-to-digital converter (ADC). After the conversion is completed, a stream of digital data is available. DSP algorithms are used to filter and compress the digital values. When these values are required by an application, the digital values are converted back to analog signals using a digital-to-analog converter (DAC).

5.3 BASIC CONCEPTS OF RX DSP LIBRARY

The RX DSP library [1] has used the word *kernel* to refer to the common DSP functions such as FIR filter and Fast Fourier Transform. This naming convention has been used to avoid confusion between the common DSP functions and the functions used within them. For example, the FIR kernel is one of the common DSP functions and it would, in turn, have a number of functions within it.

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 94

The RX DSP Library version 3.0 (CCRX) for High-performance Embedded Workshop Application Note [2] has additional information on building projects and library use. A similar Application Note is available for the E2 Studio toolset.

5.3.1 DSP Library Kernels

The RX DSP library consists of a total of 35 kernels. These kernels include hundreds of functions. The kernels are categorized into five types, which are: Statistical kernel, Filter kernel, Transform kernel, Complex number kernel, and Matrix kernel. The functions of each kernel include:

- **Statistical kernels**
  - Mean
  - Max/Min value
  - Max/Min value with index
  - Mean and Max Absolute Value
  - Variance
  - Histogram
  - Mean Absolute Deviation
  - Median
- **Filter kernels**
  - Generic FIR
  - IIR Biquad
  - Leaky LMS Adaptive
  - Lattice FIR
  - Lattice IIR
  - Single-pole IIR
  - Generic IIR
- **Transform kernels**
  - Forward Complex DFT
  - Inverse Complex DFT
  - Forward Real DFT
  - Inverse Complex-conjugate-symmetric DFT
  - Forward Complex FFT
  - Inverse Complex FFT
  - Forward Real FFT
  - Inverse Complex-conjugate-symmetric FFT
- **Complex number kernels**
  - Complex Magnitude
  - Complex Magnitude squared
  - Complex Phase

- Complex Add
- Complex Subtract
- Complex Multiply
- Complex Conjugate

- **Matrix kernels**
  - Matrix Add
  - Matrix Subtract
  - Matrix Multiply
  - Matrix Transpose
  - Matrix Scale

There are eight variants of the DSP libraries based on the combinations of the following thee execution attributes:

1. With or without a Floating Point Unit
2. Little or Big Endian
3. With or without error handling

5.3.2 Data Types Supported and Data Structure

Data Types

The RX DSP library supports three data types:

1. 16-bit fixed point,
2. 32-bit fixed point, and
3. 32-bit floating point.

The input data, output data, and coefficients for all types of kernels in the DSP library are assumed to have signed numerical format, even if it is known in advance that the kernel is limited to non-negative values.

Data Structures

The RX DSP library defines the following three types of data structures: complex numbers, vector and matrices, and algorithm kernel handles.

Complex Data

Complex data, as the name suggests, contains a real part and an imaginary part. The data structure of complex numbers pairs two data values to form a single complex data element. Structures can be defined for 16-bit fixed point (cplxi16_t), 32-bit fixed point (cplxi32_t)

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 96

and 32-bit floating point (cplxf32_t). An example of a 16-bit fixed point complex data structure is as follows:

```
1. typedef struct {
2.    int16_t re;
3.    int16_t im;
4. } cplxi16_t;
```

When using the DSP library, if a 16-bit fixed point data structure is needed, we can use the above structure and define it as follows:

```
cplxi16_t inputData;
```

Vector and Matrices

Vector and matrix data structures contain vector or matrix dimensions, and a pointer to the actual array.

A vector data structure pairs data and a pointer to an actual array of data. The user is responsible for allocating buffer memory for the vector data. A vector structure is defined as:

```
1. typedef struct {
2.    uint32_t n;
3.    void *data;
4. } vector_t;
```

Matrix data structures include two data values (row and column size), and a pointer to the actual array of data. Similar to vectors, the user is also responsible for allocating buffer memory for the matrix data. A matrix structure is defined as:

```
1. typedef struct {
2.    uint16_t nRows
3.    uint16_t nCols;
4.    void *data;
5. } matrix_t;
```

Kernel Handles

All the information that is needed by the kernel from the user for its operation is aggregated in a data structure called a handle. When the user wants to use a kernel, they should set up all the elements of the handle specific to that kernel. Once completed, the user can call the kernel by passing the pointer to the handle, inputting the start address, outputting the start address, and other information required that may vary for each ker-

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 97

nel. Users must not change the handle parameters anywhere in the program, without re-initializing the Kernel. Examples for setting up a handle for FIR filter are discussed later in this chapter.

Floating Point Exceptions

Whenever floating point data is to be used, it is advisable to check for floating point errors. The library does not use an implicit floating point error checking mechanism. The user is required to provide a floating point exception handling mechanism using the IEEE Floating Point Standard.

5.3.3 Function Naming Convention and Arguments

All the functions used in the RX DSP library have a fixed naming convention as illustrated in the following:

R_DSP_<kernel>[_variant][_function]_<intype><outtype>(arguments)

Or

R_DSP_<kernel>[_variant][_function]_<intype>(arguments)

In the example template, the components of the function name have the following meanings:

R_DSP_	a fixed prefix that identifies the function as a component of the DSP library.
<kernel>	the name of the DSP kernel; e.g., “FIR” for an FIR filter kernel.
[_variant]	an optional variation of the kernel. For example, Complex Magnitude kernel has a “_Fast” variant, indicating a faster implementation.
[_function]	an option that indicates that the function performs a setup or management task, rather than the DSP algorithm processing. For example, the FIR filter has an “_Init” function that initializes the filter state.
<intype> & <outtype>	indicate the input and output data formats, respectively.

Input and output types can be:

- i16—16-bit fixed point
- ci16—Complex 16-bit fixed point

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 98

- i32—32-bit fixed point
- ci32—Complex 32-bit fixed point
- f32—32-bit floating point
- cf32—Complex 32-bit floating point

Function Arguments

Once the handle elements are set up then the user only needs to pass the handle along with other parameters to the specific kernel. In this section the user can see the arguments that are passed with the functions. Function arguments can be fixed point or floating point. Appropriate conversion should be provided to make sure that fixed point arguments receive fixed point variables.

<handle>:	a pointer to kernel handle data structure containing kernel-specific state, coefficient, parameters and options.
<input1> . . . <inputN>:	one or more input arguments passed as pointers for most data types, except scalar data. Scalar data values may be passed directly.
<output1> . . . <outputN>:	one or more output pointers.
<additional options>:	any kernel parameters or options that are not included in the kernel handle data structure.

Functions which return the size that needs to be allocated during runtime, such as when using the command "malloc", sometimes may also return a negative value, to indicate an error condition. The "malloc" command expects a "size_t" parameter, which is unsigned data type. Since our platform is 32 bit, therefore, therefore this will return a 32 bit value. As a general rule, it is better to keep all size-related variables as 32 bit unsigned integers for the RX63N.

5.4 BASIC EXAMPLES

The examples provided in these sections give a basic idea on how to use DSP library functions. The examples include functions for using the Finite Impulse Response (FIR) filter and Fast Fourier Transform (FFT).

5.4.1 Finite Impulse Response (FIR) Filter

The block Finite Impulse Response (FIR) filter kernel operates on a user selectable number of input samples and produces the same number of output samples each time it is invoked.

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 99

The function format of the FIR filter which should be used for the function call is:

```
int32_t R_DSP_FIR_<intype><outtype> (const r_dsp_firfilter_t * handle,
const vector_t * input, vector_t * output)
```

Description

The block FIR filter kernels implements a finite impulse response filter on each input sample. The following equation shows the general structure of a *T*-tap FIR filter where *h* represents the coefficients, *x* represents the input data, and *y* represents the output data.

$$y(n) = \sum_{i=0}^{T-1} h(i) * x(n - i)$$

Each output sample is the result of performing a FIR filter of *n* taps. The FIR filter kernel supports a number of input and output data types as shown in the previous example (for example, 16-bit fixed point, 32-bit fixed point, 32-bit floating point). This is shown graphically in Figure 5.1.

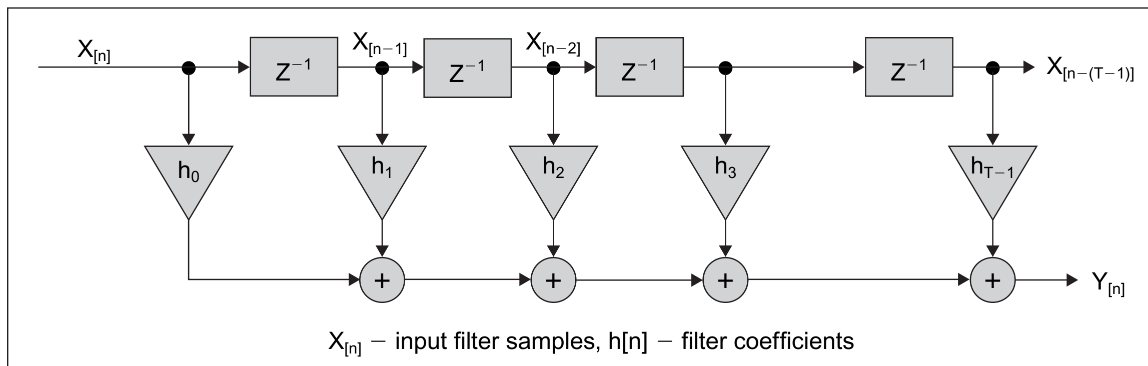


Figure 5.1 FIR filter [1], page 47.

FIR Data Structure

All variants of the FIR kernel use a handle to the filter of type r_dsp_firfilter_t. This handle is passed as part of the call to the filter. The data structure for the handle type is as follows:

```
1. typedef struct {
2.   uint32_t taps;
3.   void * coefs;
4.   void * state;
```

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 100


```
5.   int32_t scale;
6.   uint32_t options;
7. } r_dsp_firfilter_t;
```

Each member of the data structure is as follows:

- taps = Number of filter taps.
- coefs = Pointer to the coefficient vector (must be the same data type as the input vector). The content of this array is maintained by the user.
- state = Pointer to the internal state of the filter, including the delay line and any other implementation-dependent state. The memory for the internal state is allocated by the user and the content of the internal state is maintained by the kernel.
- scale = Scaling factor for the output data. Results are right-shifted by scale prior to writing the output to memory. The scale must be equal to the number of fraction bits of coefficient.
- options = A bit-mapped parameter controlling options. See "Rounding and Saturation Support" in software overview section ([1], page 17), for the definition of available modes. Currently, only rounding mode and saturation are supported for this.

The following example shows the initialization and run-time usage for the FIR filter with real 16-bit fixed-point input and output data. A similar mechanism is used for the other data formats (32-bit real fixed point, 32-bit real floating point, 16-bit complex fixed point, 32-bit complex fixed point, and 32-bit complex floating point).

Declaration

```
1. #define NUM_TAPS 10
2. #define NUM_SAMPLES 10
3. #define FRACTION_BITS 15
4. #define CONVERSION_CONST ((1 << FRACTION_BITS)-1)
5. static cplxi16_t myCoeffs[NUM_TAPS] = {
   {(int16_t)(0.0029024*CONVERSION_CONST),(int16_t)(0.0210426*CONVERSION_CONST)},
   {(int16_t)(0.0100975*CONVERSION_CONST),(int16_t)(0.0460262*CONVERSION_CONST)},
   {(int16_t)(0.0098667*CONVERSION_CONST),(int16_t)(0.0547532*CONVERSION_CONST)},
   {(int16_t)(0.0010075*CONVERSION_CONST),(int16_t)(0.0490032*CONVERSION_CONST)},
   {(int16_t)(0.0149086*CONVERSION_CONST),(int16_t)(0.0336059*CONVERSION_CONST)},
   {(int16_t)(0.0336059*CONVERSION_CONST),(int16_t)(0.0149086*CONVERSION_CONST)},
   {(int16_t)(0.0490032*CONVERSION_CONST),(int16_t)(0.0010075*CONVERSION_CONST)},
   {(int16_t)(0.0547532*CONVERSION_CONST),(int16_t)(0.0098667*CONVERSION_CONST)},
   {(int16_t)(0.0460262*CONVERSION_CONST),(int16_t)(0.0100975*CONVERSION_CONST)},
   {(int16_t)(0.0210426*CONVERSION_CONST),(int16_t)(0.0029024*CONVERSION_CONST)}
```

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 101


```
   };
6. static cplxi16_t inputData[NUM_TAPS - 1 + NUM_SAMPLES*2] = {
   {0, 0},//x(-9), start of delayline
   {0, 0},//x(-8)
   {0, 0},//x(-7)
   {0, 0},//x(-6)
   {0, 0},//x(-5)
   {0, 0},//x(-4)
   {0, 0},//x(-3)
   {0, 0},//x(-2)
   {0, 0},//x(-1)
   //start of 1st block input
   {(int16_t)( 1.0000*CONVERSION_CONST),(int16_t)( 0.0000*CONVERSION_CONST)},//x(0)
   {(int16_t)( 0.0530*CONVERSION_CONST),(int16_t)( 0.2662*CONVERSION_CONST)},//x(1)
   {(int16_t)( 0.7877*CONVERSION_CONST),(int16_t)(-0.3263*CONVERSION_CONST)},//x(2)
   {(int16_t)( 0.4080*CONVERSION_CONST),(int16_t)( 0.6106*CONVERSION_CONST)},//x(3)
   {(int16_t)( 0.3210*CONVERSION_CONST),(int16_t)(-0.3210*CONVERSION_CONST)},//x(4)
   {(int16_t)( 0.8155*CONVERSION_CONST),(int16_t)( 0.5449*CONVERSION_CONST)},//x(5)
   {(int16_t)(-0.0300*CONVERSION_CONST),(int16_t)( 0.0725*CONVERSION_CONST)},//x(6)
   {(int16_t)( 0.9202*CONVERSION_CONST),(int16_t)( 0.1830*CONVERSION_CONST)},//x(7)
   {(int16_t)( 0.0000*CONVERSION_CONST),(int16_t)( 0.5878*CONVERSION_CONST)},//x(8)
   {(int16_t)( 0.6072*CONVERSION_CONST),(int16_t)(-0.1208*CONVERSION_CONST)},//x(9)
   //start of 2nd block input
   {(int16_t)( 0.3536*CONVERSION_CONST),(int16_t)( 0.8536*CONVERSION_CONST)},//x(10)
   {(int16_t)( 0.0977*CONVERSION_CONST),(int16_t)(-0.0653*CONVERSION_CONST)},//x(11)
   {(int16_t)( 0.6984*CONVERSION_CONST),(int16_t)( 0.6984*CONVERSION_CONST)},//x(12)
   {(int16_t)(-0.2326*CONVERSION_CONST),(int16_t)( 0.3481*CONVERSION_CONST)},//x(13)
   {(int16_t)( 0.7025*CONVERSION_CONST),(int16_t)( 0.2910*CONVERSION_CONST)},//x(14)
   {(int16_t)(-0.1622*CONVERSION_CONST),(int16_t)( 0.8155*CONVERSION_CONST)},//x(15)
   {(int16_t)( 0.3090*CONVERSION_CONST),(int16_t)( 0.0000*CONVERSION_CONST)},//x(16)
   {(int16_t)( 0.1949*CONVERSION_CONST),(int16_t)( 0.9800*CONVERSION_CONST)},//x(17)
   {(int16_t)(-0.2157*CONVERSION_CONST),(int16_t)( 0.0893*CONVERSION_CONST)},//x(18)
   {(int16_t)( 0.4847*CONVERSION_CONST),(int16_t)( 0.7255*CONVERSION_CONST)},//x(19)
   };
7. static cplxi16_t outputData[NUM_SAMPLES*2];
8. static int32_t myScale = FRACTION_BITS;
```

Main Function

```
 9. void main(void) {
10.    status = sample_dsp_fir();
```

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 102


```
11.    if(status < 0) return;
12.    while(1);
13. }
```

Sample_dsp_fir()

```
14. int32_t sample_dsp_fir (void) {
15.    r_dsp_firfilter_t myFilterHandle;
16.    vector_t myInput;
17.    vector_t myOutput;
18.    int32_t myFIRFlags = R_DSP_STATUS_OK;
19.    myFilterHandle.taps = NUM_TAPS;
20.    myFilterHandle.scale = myScale;
21.    myFilterHandle.options =(R_DSP_SATURATE | R_DSP_ROUNDING_
       NEAREST);
22.    myFilterHandle.state = (void *)&inputData[0];
23.    myFilterHandle.coefs = (void *)myCoeffs;
24.    myFIRFlags = R_DSP_FIR_Init_ci16ci16(&myFilterHandle);
25.    if(myFIRFlags != R_DSP_STATUS_OK)
26.       return myFIRFlags;
27.    myInput.n = NUM_SAMPLES;
28.    myInput.data = (void *)&inputData[NUM_TAPS - 1];
29.    myOutput.data = (void *)outputData;
30.    myFIRFlags = R_DSP_FIR_0063i16ci16(&myFilterHandle, &myInput,
          &myOutput);
31.    if(myFIRFlags != R_DSP_STATUS_OK)
32.       return myFIRFlags;
33.    myFilterHandle.state = (void *)&inputData[NUM_SAMPLES];
34.    myInput.data= (void *)&inputData[NUM_TAPS - 1 + NUM_SAMPLES];
35.    myOutput.data = (void *)&outputData[NUM_SAMPLES];
36.    myFIRFlags = R_DSP_FIR_ci16ci16(&myFilterHandle, &myInput,
          &myOutput);
37.    if(myFIRFlags != R_DSP_STATUS_OK)
38.       return myFIRFlags;
39.    return myFIRFlags;
40. }
```

The explanations for the FIR code example are as follows:

Lines 1 to 4 define the macro for number of taps, sample, fraction bits, and conversion constant. Line 5 stores the coefficient in time-reversed order. Line 6 stores the input in time-sequential order with delay followed by first block input and second block input. Line 7

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 103

declares an output array of size 2*number of samples (20), which could store two blocks of output. Line 8 sets the scale number of fraction bits of the coefficient. Line 9 is the start of the main function. Line 10 calls the *sample_dsp_fir* function storing the return value of the function in a variable status. Line 11 determines if the value of status variable is less than zero and if the condition is true returns the control.

In sample_dsp_fir function, line 15 declares a member to the structure r_dsp_firfilter_t. Line 16 through 17 declare an input vector and output vector. Line 18 declares a variable myFIRFlags which is used to store status of the process. Line 19 through 23 assign values to the members of r_dsp_firfilter_t (refer to FIR data structure section in earlier part of this chapter). Line 27 through 29 sets up the input and outputs block and waits for the first block of inputs. Line 30 processes the first block of input by calling the function R_DSP_FIR_ci16ci16 passing the address myfilterHandle, input and output. Line 34 through 35 sets up the input and outputs block and waits for the first block of inputs. Line 36 processes the second block of input by calling the function R_DSP_FIR_ci16ci16 passing the address myfilterHandle, input and output. At this point myoutput.n holds the number of output samples generated by library, where the data are written to the array pointed to myOutput.data.

5.4.2 Matrix Multiplication

The RX DSP library provides built-in functions for matrix addition, subtraction, multiplication, transpose and scale which the user can simply implement by calling these functions. Using these functions directly instead of developing code for them saves time for the user and at the same time implements the functionalities with fewer instructions, thereby improving performance.

The example provided in this section is for matrix multiplication. By going through this section, the reader should be able to use the other matrix kernel functions effectively. This function performs the multiplication of two matrices, and generates a matrix product. The first input matrix is the multiplicand; the second is the multiplier. The *j*th element in the *i*th row of the matrix product is the dot product of the *i*th row of the multiplicand, and the *j*th column of the multiplier. In general, a matrix multiply is not commutative; specifically the multiplicand and multiplier cannot be exchanged. The number of columns of the multiplicand should be identical to the number of rows of the multiplier. The output matrix should have the same number of rows as the multiplicand, and the same number of columns as the multiplier.

Function Call Format

```
int32_t R_DSP_MatrixMul_<intype><outtype>
(const matrix_t* inputA, const matrix_t* inputB, matrix_t* output,
scale_t shift, uint16_t options)
```

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 104

where

`<intype>`	input type of matrix elements
`<outtype>`	output type of matrix elements
`matrix_t* inputA`	Pointer to first input matrix
`matrix_t* inputB`	Pointer to the second input matrix
`matrix_t* output`	Pointer to the output matrix
`scale_t shift`	Scaling factor for the output data (i.e. right shift answer)
`uint16_t options`	A bit-mapped parameter controlling option, associated with rounding mode and saturation.

Parameters used in matrix multiplication functions:

inputA	Pointer to the multiplicand *matrix*. Neither the matrix structure nor the actual data pointed to in the structure is altered by the function.
inputA→nRows	Number of rows of the matrix. This value is read by the function.
inputA→nCols	Number of columns of the matrix. This value is read by the function.
inputA→data	Pointer to the first element of the matrix.
inputB	Pointer to the multiplier matrix. Neither the matrix structure nor the actual data pointed to in the structure is altered by the function.
inputB→nRows	Number of rows of the matrix. This value is read by the function.
inputB→nCols	Number of columns of the matrix. This value is read by the function.
inputB→data	Pointer to the first element of the matrix.
output	Pointer to the output *matrix*. Both the matrix structure and the actual output data are altered by the function.
output→nRows	Number of rows of the matrix. This value will be updated by the function.
output→nCols	Number of columns of the matrix. This value will be updated by the function.
output→data	Points to the first element of the output matrix. The whole matrix will be overwritten by the function.
shift	This scaling factor depends on the particular implementation, but usually means right-shifting the output number a specified number of bits. A zero here means the output is not scaled.
options	The controlling options determine rounding mode: truncated (default) or nearest as well as saturate (default) or no-saturate. A "NULL" in this field selects the defaults.

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 105

Description

To explain how the matrix multiplication kernel function works we have used two input matrixes A and B; the product of the input matrixes is stored in matrix C. The working operation of the function is as follows:

$$C = \begin{bmatrix} c0,0 & c0,1 & \dots & c0, M-1 \\ c1,0 & c1,1 & \dots & c1, M-1 \\ \dots & \dots & \dots & \dots \\ cN-1,0 & cN-1,1 & \dots & cN-1, M-1 \end{bmatrix} = A * B$$

$$= \begin{bmatrix} a0,0 & a0,1 & \dots & a0, M-1 \\ a1,0 & a1,1 & \dots & a1, M-1 \\ \dots & \dots & \dots & \dots \\ aN-1,0 & aN-1,1 & \dots & aN-1, M-1 \end{bmatrix} * \begin{bmatrix} b0,0 & b0,1 & \dots & b0, K-1 \\ b1,0 & b1,1 & \dots & b1, K-1 \\ \dots & \dots & \dots & \dots \\ bM-1,0 & bM-1,1 & \dots & bM-1, K-1 \end{bmatrix}$$

$$= \begin{bmatrix} \sum_0^{M-1} a0, j * bj, 0 & \sum_0^{M-1} a0, j * bj, 1 & \dots & \sum_0^{M-1} a0, j * bj, K-1 \\ \sum_0^{M-1} a1, j * bj, 0 & \sum_0^{M-1} a1, j * bj, 1 & \dots & \sum_0^{M-1} a1, j * bj, K-1 \\ \dots & \dots & \dots & \dots \\ \sum_0^{M-1} aN-1, j * bj, 0 & \sum_0^{M-1} aN-1, j * bj, 1 & \dots & \sum_0^{M-1} aN-1, j * bj, K-1 \end{bmatrix}$$

Figure 5.2 Matrix Multiply [1], page 138.

where N is A's number of rows. M is A's number of columns, and B's number of rows. K is B's number of columns. The matrix product will have N rows and K columns.

The code example below explains setting up the handle and the parameters required to call the R_DSP_MatrixMul() function. In this example, we simply fill a matrix with some numbers as a demonstration; you, of course, would need to fill the matrices with your valid data!

```
 1. #define NUM_ROWS_A      4
 2. #define NUM_COLUMNS_A   4
 3. #define NUM_ROWS_B      4
 4. #define NUM_COLUMNS_B   4
 5. int32_t dataLeft[NUM_ROWS_A * NUM_COLUMNS_A];
 6. int32_t dataRight[NUM_ROWS_B * NUM_COLUMNS_B];
 7. int32_t dataOut[NUM_ROWS_A * NUM_COLUMNS_B];
 8. matrix_t channLeft;
 9. matrix_t channRight;
10. matrix_t channOut;
```

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 106


```
11. scale_t shift;              //scaling factor
12. r_dsp_status_t my_status;   //holds status of multiply
13. int i;                      //used as a counter
14.
15. //fill the matrices
16. for (i = 0; i < (NUM_ROWS_A * NUM_COLUMNS_A); i++) {
17.    chann Left[i] = 2;
18.    channLeft[i] = 3;
19. }
20.
21. // set up the function parameters
22. shift.i32 = 0;   //do not scale the result of the multiply
23. channLeft.data = (void *) dataLeft;
24. channRight.data = (void *) dataRight;
25. channLeft.nRows = NUM_ROWS_A;
26. channLeft.nCols = NUM_COLUMNS_A;
27. channRight.nRows = NUM_ROWS_B;
28. channRight.nCols = NUM_COLUMNS_B;
29. channOut.data = (void *) dataOut;
30. R_DSP_MatrixMul_i32i32 (&channLeft, &channRight, &channOut,shift,
    NULL);
31. //if an error occurred, you should have a function to handle
32. if (my_status != R_DSP_STATUS_OK) error();
```

Explanation

Lines 1 to 4 are macro definitions for number of columns and rows. Lines 5 through 7 declare two input and one output arrays of size, number of rows * number of columns. Line 8 through 10 declare three variables channLeft(input), channRight(input), and channOut(output) with data type as a matrix which is a structure. Line 11 creates the scaling variable that will later be used in the multiply function call. Line 12 allocates the variable for the return status of the multiply operation. Lines 16 through 19 simply fill the matrices with example data.

Line 22 assigns 0 as the scaling factor (no right-shift of the resultant multiply). Line 23 assigns the data variable of channLeft structure to one of the input arrays (dataLeft). Line 24 assigns the data variable of channRight structure to another input array (dataRight). Lines 25 and 26 assign the nRows and nCols variable of channLeft structure to number of rows and columns of input A respectively. Lines 27 and 28 assign the nRows and nCols variable of channRight structure to number of rows and columns of input *B* respectively. Line 29 assigns the data variable of channOut to the output array.

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 107

Line 30 calls the R_DSP_MatrixMul_i32i32 () function, passing the pointer to two input matrixes and one output matrix. In the function parameters i32i32 denotes that input and output are both 32-bit fixed point. If the input has to be changed to 16-bit fixed then i16i32 should be used. Line 32 is needed to see if the multiply operation was successful, otherwise your own error handling function should gracefully recover from this error.

The DSP Library does not treat matrices like we conventionally treat them in C. Hence matrices can be stored or accessed considering them as an array of size rows * columns.

5.4.3 Fast Fourier Transform (FFT)

The RX DSP library provides a built in Fourier Transform Kernel which helps in estimating and calculating the spectrum of the signal. This section describes how to use this API and its functions using handles as described earlier. DSP library offers API for Discrete Fourier Transform (DFT) as well as Fast Fourier Transform (FFT). In this section we primarily discuss how to use the FFT kernel with a real time input signal given to the board.

The Fourier Transform converts a time domain signal to frequency domain. The Inverse Fourier Transform does the opposite; it converts the frequency domain signal back to time domain. The only difference between DFT and FFT is that FFT utilizes a much faster algorithm, and hence is more efficient in terms of speed than DFT. FFT is most efficient when the number of samples taken is a power of 2. The RX DSP library facilitates any number of samples between 16 and 8192. The order of the FFT refers to the base-2 of the logarithm of points in the transform.

Important header files needs to be included before we can use the transform kernel.

```
#include "r_dsp_transform.h"
#include "r_dsp_types.h"
```

R_dsp_transform.h provides, the kernel definitions for FFT, and *r_dsp_types.h* provides the structure definitions for all data structures used in the library.

There is some preliminary setup involved before we can use FFT kernel. Since the FFT algorithm requires twiddle factors and bit reverse LUT's, we need to allocate memory for these. It is advisable to allocate this memory dynamically. The number of twiddle factors required is equal to the number of points in the transform. First we declare static variables for holding the sizes of the twiddle array and the LUT's.

```
static size_t ntwb;
static size_t nbrb;
static size_t nwkb;
```

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 108

The handle for the kernel can be initialized with N being the points in the transform and using default options initially, which can be later overwritten.

```
//default options set, handle h defined.
static r_dsp_fft_t h = {N, 0};
```

Vector data needs to be defined to hold the time domain data and frequency domain samples.

```
vector_t vtime = {N, (void *)buf_time};
vector_t vfreq = {N/2, (void *)buf_freq};
```

Time domain data can be filled up with values taken real time from an ADC.

```
vtime.data = (void *)adc_volts;
```

Notice that our output signal contains only N/2 elements, because the Fourier transform of a real valued signal is a complex conjugate symmetric array. Hence only N/2 elements are needed to completly define the Fourier transform as the remaining elements are redundant.

For allocating memory to twiddles, we first call a function to determine the size of the array needed. This function is called in runtime, hence memory is allocated dynamically.

```
R_DSP_FFT_BufSize_i32ci32(&h, &ntwb, &nbrb, &nwkb);
h.twiddles = ntwb ? malloc(ntwb) : NULL; //
h.bitrev = nbrb ? malloc(nbrb) : NULL;
//we are not using any windowing. If windowing is used, then
//window size needs to be determined separately
h.window = NULL;
```

We can use options to scale the data at each stage, if the data is expected to be too large. This is optional but not necessary.

```
h.options  = R_DSP_FFT_OPT_SCALE;
```

After allocating the memory, we need to initialize the handle.

```
status = R_DSP_FFT_Init_i32ci32(&h);
```

status is the return type of the handle. It is used for debugging purpose and to check if the function call is successful. If it is successful, it will return R_DSP_STATUS_OK. Hence we can use this as an error checking condition as follows:

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 109


```
if (status != R_DSP_STATUS_OK){    // check if everything is okay
   error();
}
```

After the initialization is done, we just need to call the initialized handle using the function call as follows:

```
// take the actual transform, data stored in vector vfreq
status=R_DSP_FFT_i32ci32(&h, &vtime, &vfreq);
```

This function calls, puts the transform coefficients in vector *vfreq,* which points to an array *buf_freq,* which we initialized in the starting. This is called out-of-place normal order calculation. It is referred to as out of place because a different vector is being used for storing the output. If the same input vector, is used for storing the output result as well (in which case the original vector will be overwritten) then, it is called in-place computation.

The Fourier coefficients stored in the output array may consist of complex values, especially if the input is a real valued signal, as mentioned above. In that case, the output consists of interleaved real and complex data. For example, if one element of the array is a 32 bit unsigned integer then the most significant 16 bits consist of the real part and least significant 16 bits consist of complex data.

It is necessary to take care of the data format being used. The Library implicitly treats all data as signed integers, and hence care needs to be taken while dealing with unsigned integers.

5.5 RECAP

The RX_DSP library provides 36 kernel functions. The RX_DSP library uses kernel in place of function to avoid the confusion between the major functions and the function used inside the major functions. For example, the library uses the FIR kernel and all the functions inside the FIR kernel as functions. All the function calls define the input type, output type for all kernels and arguments of function which differ to each kernel. The handle is used to set up the required values for a kernel and the pointer to the handle is passed along with the function. The examples provide the explanation of FIR, matrix multiplication, and Fast Fourier Transform kernels.

5.6 REFERENCES

[1] Renesas Electronics, Inc. (August 2013). *RX DSP Library API version 3.0 User's Manual: Software,* Rev 1.0.

[2] Renesas Electronics, Inc. (August 2013), Application Note: RX DSP Library version 3.0 (CCRX) for High-performance Embedded Workshop, Rev. 1.0

05.ES_Conrad_RX63N_Advanced_CH05.qxd:RX63N Advanced 3/4/14 8:43 PM Page 110

5.7 EXERCISE

1. What is the difference between a kernel and a function in the RX DSP library?
2. Provide the function call instruction used for FIR filter with input type as 16-bit fixed point and output type as 32-bit fixed point. Also explain the function arguments used in the function call.
3. All variants of the FIR kernel use a handle to the filter of type r_dsp_firfilter_t. Provide the data structure of the handle and an explanation of the structure member.
4. The RX DSP library consists of how many functions?
5. Is the following an example of a valid 16-bit floating-point complex data structure? Why or why not?

```
typedef struct{
   int16_t re;
   int16_t im;
} cplxf16_t;
```

Chapter 6

Direct Memory Access Controller

6.1 LEARNING OBJECTIVES

Direct Memory Access (DMA) is a feature of microcontrollers and microprocessors that allows hardware subsystems to access system memory directly without depending on the central processor. For example, large blocks of data can be directly transferred from the RAM subsystem to a video memory subsystem.

In this chapter we will learn:

- Fundamentals of DMA.
- How it helps the efficiency and execution throughput of a microcontroller system.
- How to implement the DMAC hardware for transfers.

6.2 BASIC CONCEPTS

A DMA Controller (DMAC) is a device that controls the system bus and directly transfers information from one part of the system to another. This transfer task is often necessary because large blocks of data must be rapidly moved, sometimes at speeds that are faster than is practical if the central processor is involved. The DMAC is a module used to transfer data without the Central Processing Unit (CPU) processing each data byte directly. When a DMA transfer request is generated, the DMAC transfers data stored at the transfer source address to the transfer destination address.

There is a difference in performance when there is a processor involved between huge memory transfers and when there is only a DMAC involved. When a processor is involved in large memory transfers, the overall data transfer is slower since the data transfer rate is largely dependent on the processor. This is because with every byte or word of data transferred, the processor must fetch the instruction, calculate the from/to addresses, fetch the data, and write the data. When DMAC is used, data transfers are independent of the processor, are interrupt-based, use dedicated hardware for addressing, and are thus faster. DMA based data transfers have considerably less overhead, making them very efficient. When DMA is used, the processor is free to perform other important tasks while the data transfer takes place independently (Figure 6.1).

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 112

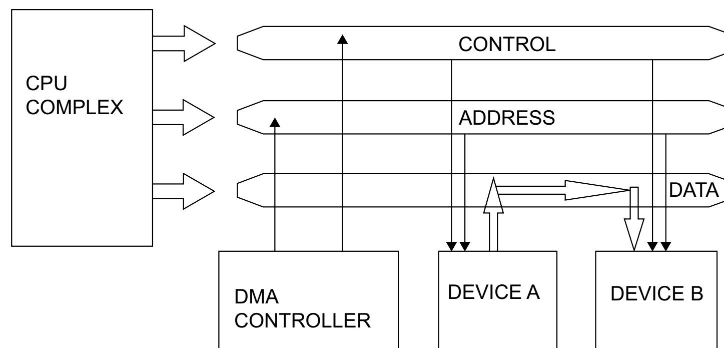


Figure 6.1 General DMAC block diagram.

Displaying pictures on a video screen is one example. Displaying pictures on a typical monitor requires a complete refresh scan (one frame) of the screen 60 times a second. If a screen is 1280 by 800 pixels and each pixel requires 3 bytes to define its color, then over 184 million bytes need to be moved every second. For a computer to perform efficiently, it needs additional circuitry to read and write these bytes. Quite a few methods are available to accomplish this, but one solution is to store these bytes in a special display memory with built-in scanning circuitry and an arbitration scheme between the memory accessed by the CPU and the memory accessed by scanning the circuitry. A bus arbitration scheme is used to manage the multiple devices that try to master the bus simultaneously. This solution is used today by graphics RAM. This solution is commonly called a frame buffer.

Due to demanding repetitive timing constraints, DMA is not recommended for directly scanning the regular memory for video displays. This method is, however, one of the most preferred methods for quickly transferring information. For example, DMA is often used when a picture or part of a picture needs to be moved quickly between memory and the frame buffer. It is also widely used when the buffer contents for a hard disk or a flash drive need to be quickly transferred to new locations.

Another example of using DMA is configuring the analog to digital converter (ADC) to continuously sample data and transferring the data directly into RAM.

In general, the DMAC is used as follows:

1. The DMAC is instructed to make a transfer either by the CPU or a peripheral.
2. The DMA requests the controller to gain control of the bus from the CPU, other processors, or controllers which might currently be using the bus.
3. The devices relinquish control of the bus and put their lines into a tri-state condition.
4. The DMAC takes over the bus, generating its own control signals and address for the bus.
5. The DMAC executes the information transfer.
6. The DMAC relinquishes control of the bus, often informing the CPU it has completed the transfer via an interrupt.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 113

Table 6.1 contains the general specifications of the DMAC. The RX63N/RX631 Group uses a four-channel direct memory access controller (DMAC). Of interest is the identification of three modes of operation: Normal Transfer Mode, Repeat Transfer Mode, and Block Transfer Mode.

TABLE 6.1 Specifications of DMAC [1], page 518.

ITEM		DESCRIPTION
Number of channels		4 (DMACm (m = 0 to 3))
Transfer space		512 Mbytes (0000 0000h to 0FFF FFFFh and F000 0000h to FFFF FFFFh excluding reserved areas)
Maximum transfer volume		1 Mbyte (Maximum number of transfers in block transfer mode: 1,024 data × 1,024 blocks)
DMA request source		■ Activation source selectable for each channel Software trigger Interrupt requests from peripheral modules or trigger input to external interrupt input pins
Channel priority		Channel 0 > Channel 1 > Channel 2 > Channel 3 (Channel 0: Highest)
Transfer data	Single data	Bit length: 8, 16, 32 bits
	Block size	Number of data: 1 to 1,024
Transfer mode	Normal transfer mode	■ One data transfer by one DMA transfer request ■ Free running mode (setting in which total number of data transfers is not specified) settable
	Repeat transfer mode	■ One data transfer by one DMA transfer request ■ Program returns to the transfer start address on completion of the repeat size of data transfer specified for the transfer source or destination ■ Maximum settable repeat size: 1,024
	Block transfer mode	■ One block data transfer by one DMA transfer request ■ Maximum settable block size: 1,024 data
Selective functions	Extended repeat area function	■ Function in which data can be transferred by repeating the address values in the specified range with the upper bit values in the transfer address register fixed ■ Area of 2 bytes to 128 Mbytes separately settable as extended repeat area for transfer source and destination
Interrupt request	Transfer end interrupt	Generated on completion of transferring data volume specified by the transfer counter
	Transfer escape end interrupt	Generated when the repeat size of data transfer is completed or the extended repeat area overflows
Power consumption reduction function		Module-stop state can be set

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 114

Important DMAC Registers

The block diagram of the Renesas RX63N DMAC is shown in Figure 6.2. The DMAC receives its control register settings from the Internal Main Bus 2 and uses Internal Peripheral Bus 1 to control the peripheral devices for a DMA transfer. The CPU must set several control registers in order to start the transfers. The four DMACs are labeled DMAC0 through DMAC3. The important registers are listed as follows:

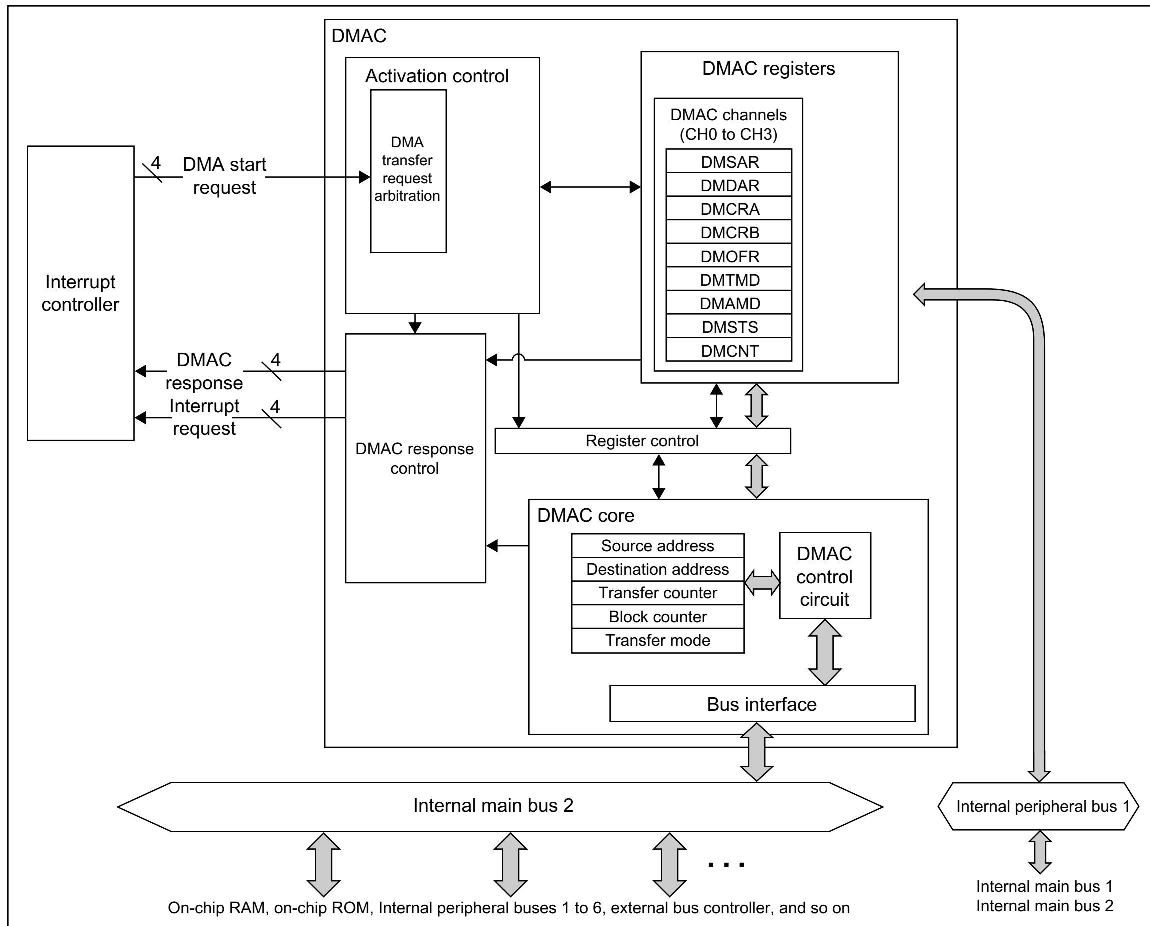


Figure 6.2 Block Diagram of DMAC [1], page 519.

DMA Source Address Register (DMSAR): The 32-bit DMA Source Address Register (DMSAR) specifies the transfer source start address. A user sets DMSAR while DMAC activation is disabled (the DMST bit in DMAST = 0) or DMA transfer is disabled (the

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 115

DTE bit in DMCNT = 0). Setting bits 31 to 29 is invalid; a value of bit 28 is extended to bits 31 to 29. Reading DMSAR returns the extended value. An example of setting an address would be:

```
DMAC0.DMSAR = 0x15000;
```

DMA Destination Address Register (DMDAR): The 32-bit DMA Destination Address Register (DMDAR) specifies the transfer destination start address. The user sets DMDAR while DMAC activation is disabled (the DMST bit in DMAST = 0) or DMA transfer is disabled (the DTE bit in DMCNT = 0). Setting bits 31 to 29 is invalid; a value of bit 28 is extended to bits 31 to 29. Reading DMDAR returns the extended value. An example of setting an address would be:

```
DMAC0.DMDAR = 0x17000;
```

DMA Transfer Count Register (DMCRA): The 32-bit DMA Transfer Count Register (DMCRA) specifies the number of transfer operations. It has a high 16-bit word (DMCRAH) and a low16-bit word (DMCRAL) that are used differently, depending on the mode of operation. Reference Figure 6.3 for the bit positions. An example of setting a transfer count would be:

```
DMAC0.DMCRA = 0x000a;
```

- Normal Transfer Mode (MD[1:0] Bits in DMACm.DMTMD = 00b)
  The DMCRAL functions as a 16-bit transfer counter. The number of transfer operations is one when the setting is 0001h and 65535 when it is FFFFh. The value is reduced in increments of one each time data is transferred. When the setting is 0000h, no specific number of transfer operations is set. The data transfer is performed with the transfer counter stopped (free running mode). DMCRAH is not used in Normal Transfer Mode. Write 0000h to DMCRAH.
- Repeat Transfer Mode (MD[1:0] Bits in DMACm.DMTMD = 01b)
  The DMCRAH specifies the repeat size and DMCRAL functions as a 10-bit transfer counter. The number of transfer operations is one when the setting is 001h, 1023 when it is 3FFh, and 1024 when it is 000h. In Repeat Transfer Mode, a value in the range of 000h to 3FFh (1 to 1024) can be set for DMCRAH and DMCRAL. Setting bits 15 to 10 in DMCRAL is invalid. Write 0 to these bits. The value in DMCRAL is decremented by one each time data is transferred until it reaches 000h, at which point the value in DMCRAH is loaded into the DMCRAL.
- Block Transfer Mode (MD[1:0] Bits in DMACm.DMTMD = 10b)
  The DMCRAH specifies the block size and DMCRAL functions as a 10-bit block size counter. The block size is one when the setting is 001h, 1023 when it is 3FFh,

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 116

and 1024 when it is 000h. In Block Transfer Mode, a value in the range of 000h to 3FFh can be set for the DMCRAH and the DMCRAL. Setting bits 15 to 10 in the DMCRAL is invalid. Write 0 to these bits. The value in the DMCRAL is decremented by one each time data is transferred until it reaches 000h, at which point the value in the DMCRAH is loaded into the DMCRAL.

DMA Block Transfer Count Register (DMCRB): The 16-bit DMA Block Transfer Count Register (DMCRB) specifies the number of block transfer operations or repeat transfer operations in Block and Repeat Transfer Mode, respectively. Only the lowest 10 bits are used. The number of transfer operations is one when the setting is 001h, 1023 when it is 3FFh, and 1024 when it is 000h. In Repeat Transfer Mode, the value is decremented by one when the final data of one repeat size is transferred. In Block Transfer Mode, the value is decremented by one when the final data of one block size is transferred. In Normal Transfer Mode, the DMCRB is not used—the setting is invalid. An example of setting a block transfer count would be:

```
DMAC0.DMCRB = 0x09;
```

DMA Transfer Mode Register (DMTMD): The 16-bit DMA Transfer Mode Register (DMTMD) specifies the DMA request source, the transfer data size, the repeat area, and the mode of operation. These fields are shown in Figure 6.3. For the DTS[1:0] select bits, either set the source or destination as the repeat area in Repeat or Block Transfer Mode. In Normal Transfer Mode, setting these bits is invalid. An example of setting the transfer mode to Normal would be:

```
DMAC0.DMTMD.BIT.MD = 0;
```

DMA Interrupt Setting Register (DMINT): The 8-bit DMA Interrupt Setting Register (DMINT) enables or disables five different types of interrupts associated with DMA transfers. These fields are shown in Figure 6.4. An example of setting the register to enable Transfer End Interrupts would be:

```
DMAC0.DMINT.BYTE = 0x10;
```

DMA Transfer Enable Register (DMCNT): The 8-bit DMA Transfer Enable Register (DMCNT) enables or disables DMA transfers. The register uses a single bit, bit 0 (DTE), to enable transfer when set to 1 or disable transfer when set to 0. When the DMST bit in the DMAST is set to 1 (DMAC activation is enabled) and this bit is set to 1 (DMA transfer is enabled), the DMA transfer can be started for the corresponding channel. The DTE bit can be read. A 0 is read from the bit when the user writes 0 to the bit, the specified total volume

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 117

DMA Transfer Mode Register (DMTMD)

Address(es): DMAC0.DMTMD 0008 2010h, DMAC1.DMTMD 0008 2050h
DMAC2.DMTMD 0008 2090h, DMAC3.DMTMD 0008 20D0h

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	MD[1:0]		DTS[1:0]		—	—	SZ[1:0]		—	—	—	—	—	—	DCTG[1:0]	
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b1, b0	DCTG[1:0]	DMA Request Source Select	b1 b0	R/W
			0 0: Software	
			0 1: Interrupts*1 from peripheral modules or external interrupt input pins	
			1 0: Setting prohibited	
			1 1: Setting prohibited	
b7 to b2	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b9, b8	SZ[1:0]	Transfer Data Size Select	b9 b8	
			0 0: 8 bits	
			0 1: 16 bits	
			1 0: 32 bits	
			1 1: Setting prohibited	
b11, b10	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b13, b12	DTS[1:0]	Repeat Area Select	b13 b12	R/W
			0 0: The destination is specified as the repeat area or block area.	
			0 1: The source is specified as the repeat area or block area.	
			1 0: The repeat area or block area is not specified.	
			1 1: Setting prohibited	
b15, b14	MD[1:0]	Transfer Mode Select	b15 b14	R/W
			0 0: Normal transfer	
			0 1: Repeat transfer	
			1 0: Block transfer	
			1 1: Setting prohibited	

Note 1. DMAC activation source is selected using the DMRSRm registers of the ICU. For details on DMAC activation sources, see Table 15.3, Interrupt Vector Table in section 15, Interrupt controller (ICUb).

Figure 6.3 Transfer Mode Register [1], page 523.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 118

DMA Interrupt Setting Register (DMINT)

Address(es): DMAC0.DMINT 0008 2013h, DMAC1.DMINT 0008 2053h
DMAC2.DMINT 0008 2093h, DMAC3.DMINT 0008 20D3h

	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	DTIE	ESIE	RPTIE	SARIE	DARIE
Value after reset:	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	DARIE	Destination Address Extended Repeat Area Overflow Interrupt Enable	0: Disables an interrupt request for an extended repeat area overflow on the destination address 1: Enables an interrupt request for an extended repeat area overflow on the destination address	R/W
b1	SARIE	Source Address Extended Repeat Area Overflow Interrupt Enable	0: Disables an interrupt request for an extended repeat area overflow on the source address 1: Enables an interrupt request for an extended repeat area overflow on the source address	R/W
b2	RPTIE	Repeat Size End Interrupt Enable	0: Disables the repeat size end interrupt request. 1: Enables the repeat size end interrupt request.	R/W
b3	ESIE	Transfer Escape End Interrupt Enable	0: Disables the transfer escape end interrupt request. 1: Enables the transfer escape end interrupt request.	R/W
b4	DTIE	Transfer End Interrupt Enable	0: Disables the transfer end interrupt request. 1: Enables the transfer end interrupt request.	R/W
b7 to b5	—	Reserved	These bits are read as 0. The write value should be 0.	R/W

Figure 6.4 DMA Interrupt Setting Register [1], page 523.

of data transfer is completed, a DMA transfer is stopped by the repeat size end interrupt, or a DMA transfer is stopped by the extended repeat area overflow interrupt. An example of setting the register to enable Transfer End Interrupts would be:

```
DMAC0.DMCNT.BYTE = 0x1;
```

Additional DMAC registers are listed in the RX63N hardware manual [1]. Some of the miscellaneous DMAC registers are:

- DMA Address Mode Register (DMAMD)
- DMA Offset Register (DMOFR)

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 119

- DMA Transfer Enable Register (DMCNT)
- DMA Software Start Register (DMREQ)
- DMA Status Register (DMSTS)
- DMA Activation Source Flag Control Register (DMCSL)
- DMA Module Activation Register (DMAST)

Modes of Operation

The RX63N microcontroller has a 4-channel Direct Memory Access Controller (DMAC) designed especially for internal bus transfer. The DMAC transfers data in three common modes of operation: Normal Transfer Mode, Repeat Transfer Mode, and Block Transfer Mode.

Normal Transfer Mode

In Normal Transfer Mode, one unit of data (i.e. byte, word) is transferred by one transfer request. Using the DMCRAL of DMACm channel, a maximum of 65535 number of transfer operations is possible. For free running mode, a number of transfer operations bits are set to 0000h. In this situation, the data transfer count is stopped and transfers are made.

In Normal Transfer mode, setting the DMCRB of the DMACm is invalid. Excluding the free running mode, a transfer end interrupt request can be generated after completion of the specified number of transfer operations. Table 6.2 summarizes the register up-

TABLE 6.2 Register Update Operation in Normal Transfer Mode [1], page 534.

REGISTER	FUNCTION	UPDATE OPERATION AFTER COMPLETION OF A TRANSFER BY ONE TRANSFER REQUEST
DMACm.DMSAR	Transfer source address	Increment/decrement/fixed/offset addition*[1]
DMACm.DMDAR	Transfer destination address	Increment/decrement/fixed/offset addition*[1]
DMACm.DMCRAL	Transfer count	Decremented by one/not updated (in free running mode)
DMACm.DMCRAH	—	Not updated (Not used in normal transfer mode)
DMACm.DMCRB	—	Not updated (Not used in normal transfer mode)
Note 1. Offset addition can be specified only for DMAC0.		

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 120

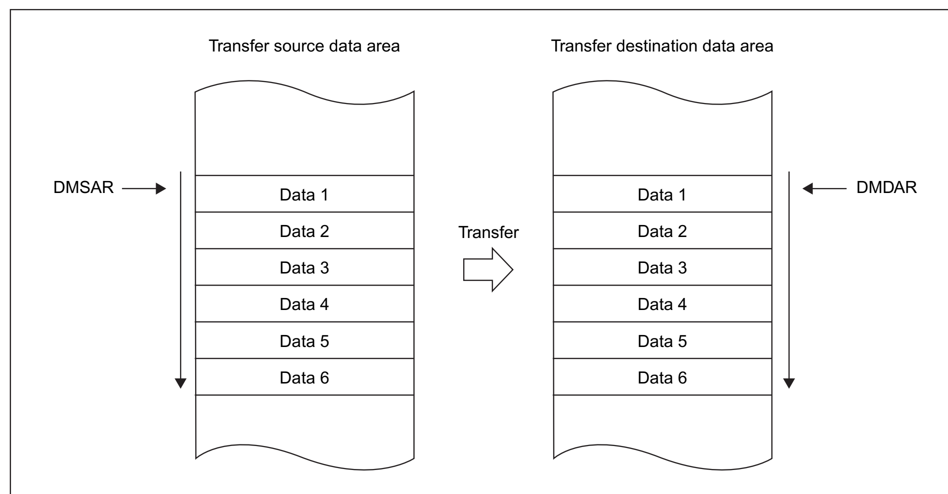


Figure 6.5 Operation in Normal Transfer Mode [1], page 534.

date operation in Normal Transfer Mode. Figure 6.5 shows the flow of data in Normal Transfer Mode.

Repeat Transfer Mode

In Repeat Transfer Mode, one unit of data (i.e. byte, word) is transferred using one transfer request. The total repeat transfer size has a maximum of 1 Kbyte of data that can be set using the DMCRA of the DMACm. The number of repeat transfer operations can be set to a maximum of 1 Kcount using the DMCRB of the the DMACm. Hence, a maximum of 1 Mbyte of data (1 Kbyte data $\times$ 1 Kcount of repeat transfer operations) can be set as a total data transfer size.

Either the transfer source or the transfer destination can be specified as a repeat area. When transfer of the repeat size data is completed, the address of the specified repeat area (the DMSAR of the DMACm or the DMDAR of the DMACm) returns to the transfer start address. When data of the specified repeat size has all been transferred in Repeat Transfer Mode, the DMA transfer can be halted and the repeat size end interrupt can be requested. The DTE bit in the DMCNT of the DMACm should be updated with 1 so that transfer can be resumed. A transfer end interrupt request can be generated once a specified number of repeat transfer operations are completed. Table 6.3 summarizes the register update operation in Repeat Transfer Mode. Figure 6.6 shows the flow of data in Repeat Transfer Mode.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 121

TABLE 6.3 Register Update Operation in Repeat Transfer Mode [1], page 535.

REGISTER	FUNCTION	UPDATE OPERATION AFTER COMPLETION OF A TRANSFER BY ONE TRANSFER REQUEST	
		WHEN DMACm.DMCRAL IS NOT 1	WHEN DMACm.DMCRAL IS 1 (TRANSFER OF THE LAST DATA IN REPEAT SIZE)
DMACm.DMSAR	Transfer source address	Increment/decrement/ fixed/offset addition*[1]	■ DMACm.DMTMD.DTS[1:0] = 00b Increment/decrement/fixed/offset addition*[1] ■ DMACm.DMTMD.DTS[1:0] = 01b Initial value of DMACm.DMSAR ■ DMACm.DMTMD.DTS[1:0] = 10b Increment/decrement/fixed/offset addition*[1]
DMACm.DMDAR	Transfer destination address	Increment/decrement/ fixed/offset addition*[1]	■ DMACm.DMTMD.DTS[1:0] = 00b Initial value of DMACm.DMDAR ■ DMACm.DMTMD.DTS[1:0] = 01b Increment/decrement/fixed/offset addition*[1] ■ DMACm.DMTMD.DTS[1:0] = 10b Increment/decrement/fixed/offset addition*[1]
DMACm.DMCRAH	Repeat size	Not updated	Not updated
DMACm.DMCRAL	Transfer count	Decremented by one	DMACm.DMCRAH
DMACm.DMCRB	Count of repeat transfer operations	Not updated	Decremented by one

Note 1. Offset addition can be specified only for DMAC0.

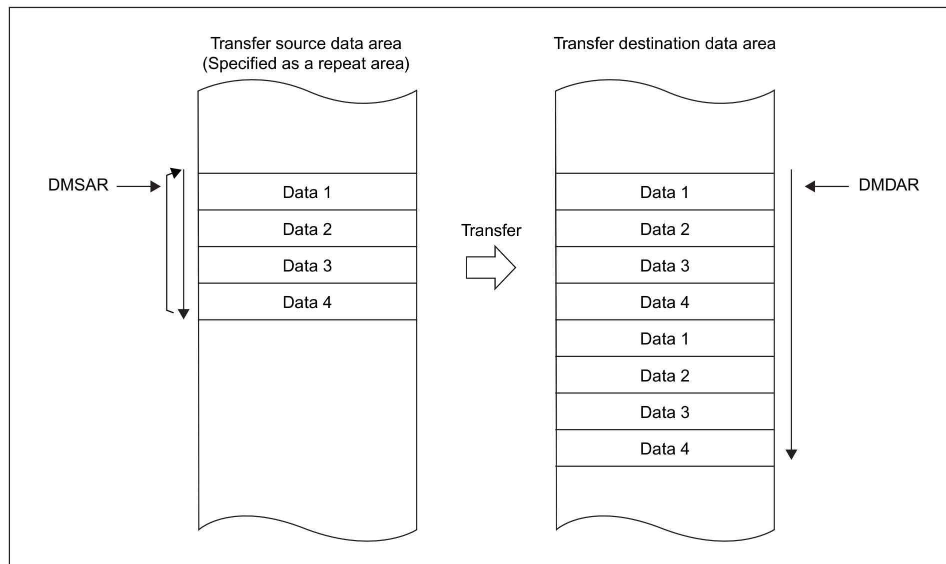


Figure 6.6 Operation in Repeat Transfer Mode [1], page 535.

Block Transfer Mode

A single block of data is transferred by one transfer request, in Block Transfer Mode. Using the DMCRA of the DMACm, a maximum of 1 Kbyte of data can be set as a total block transfer size. Using the DMCRB of the DMACm, a maximum of 1 Kcount can be set as the number of block transfer operations. Hence, a maximum of 1 Mbyte of data (1 Kbyte data × 1 Kcount (no space between K and count_of block transfer operations) can be set as a total data transfer size.

Either the transfer source or transfer destination can be specified as a block area. The address of the specified block area (the DMSAR or the DMDAR of the DMACm) returns to the transfer start address when a transfer of a single of block data has been completed. A DMA transfer can be stopped and the repeat size end interrupt can be requested when a single block of data has all been transferred in Block Transfer Mode. A DMA transfer can be resumed by writing 1 to the DTE bit in the DMCNT of the DMACm in the repeat size end interrupt handling. A transfer end interrupt request can be generated after completion of the specified number of block transfer operations.

Table 6.4 summarizes the register update operation in Block Transfer Mode, and Figure 6.7 shows the flow of data in Block Transfer Mode.

TABLE 6.4 Register Update Operation in Block Transfer Mode [1], page 537.

REGISTER	FUNCTION	UPDATE OPERATION AFTER COMPLETION OF SINGLE-BLOCK TRANSFER BY ONE TRANSFER REQUEST
DMACm.DMSAR	Transfer source address	■ DMACm.DMTMD.DTS[1:0] = 00b Increment/decrement/fixed/offset addition*[1] ■ DMACm.DMTMD.DTS[1:0] = 01b Initial value of DMACm.DMSAR ■ DMACm.DMTMD.DTS[1:0] = 10b Increment/decrement/fixed/offset addition*[1]
DMACm.DMDAR	Transfer destination address	■ DMACm.DMTMD.DTS[1:0] = 00b Initial value of DMACm.DMDAR ■ DMACm.DMTMD.DTS[1:0] = 01b Increment/decrement/fixed/offset addition*[1] ■ DMACm.DMTMD.DTS[1:0] = 10b Increment/decrement/fixed/offset addition*[1]
DMACm.DMCRAH	Block size	Not updated
DMACm.DMCRAL	Transfer count	DMACm.DMCRAH
DMACm.DMCRB	Count of block transfer operations	Decremented by one

Note 1. Offset addition can be specified only for DMAC0.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 123

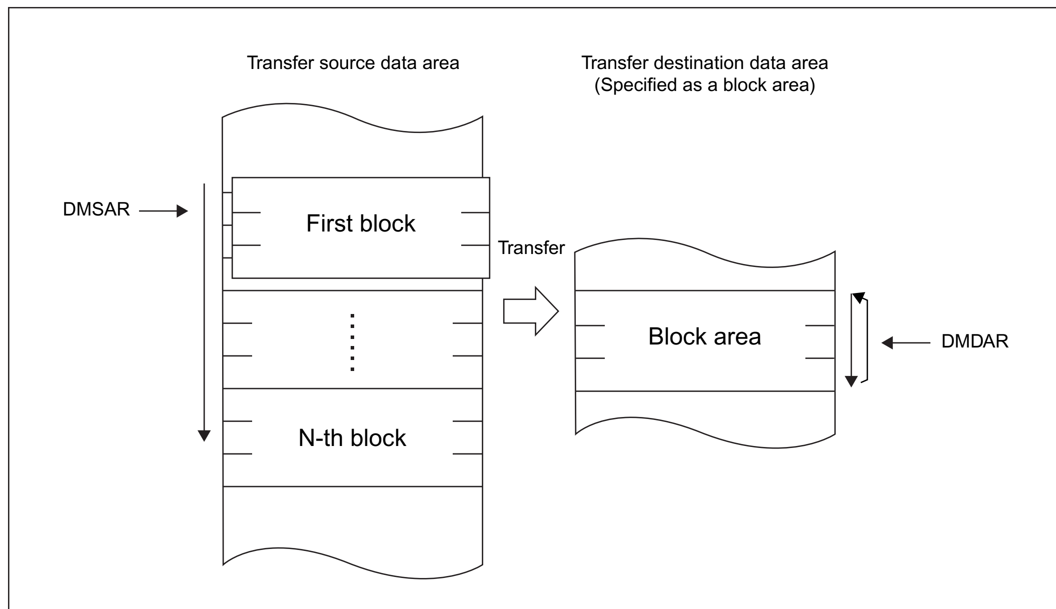


Figure 6.7 Operation in Block Transfer Mode [1], page 537.

6.3 BASIC EXAMPLES

The DMAC registers should be set in a consistent procedure. Renesas has suggested the procedure shown in Figure 6.8 for setting these registers. The remainder of this section shows examples of initiating an internal DMA transfer in each of the modes of operation.

Internal Data Transfer in Normal Transfer Mode

The following code listing is used for the implementation of a simple initialization program for an internal DMA data transfer in Normal Transfer Mode. Explanations of each line are included.

```
 1. void DMA_Normal_Transfer_init() {
 2.     DMAC0.DMCNT.BYTE = 0x0;
 3.     DMAC0.DMTMD.BIT.MD = 0;
 4.     DMAC0.DMSAR = 0x15000;
 5.     DMAC0.DMDAR = 0x17000;
 6.     DMAC0.DMCRA = 0x00a;
 7.     DMAC0.DMINT.BYTE = 0x10;
 8.     DMAC0.DMCNT.BYTE = 0x1;
 9.     DMAC.DMAST.BIT.DMST = 0x1;
10. }
```

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 124

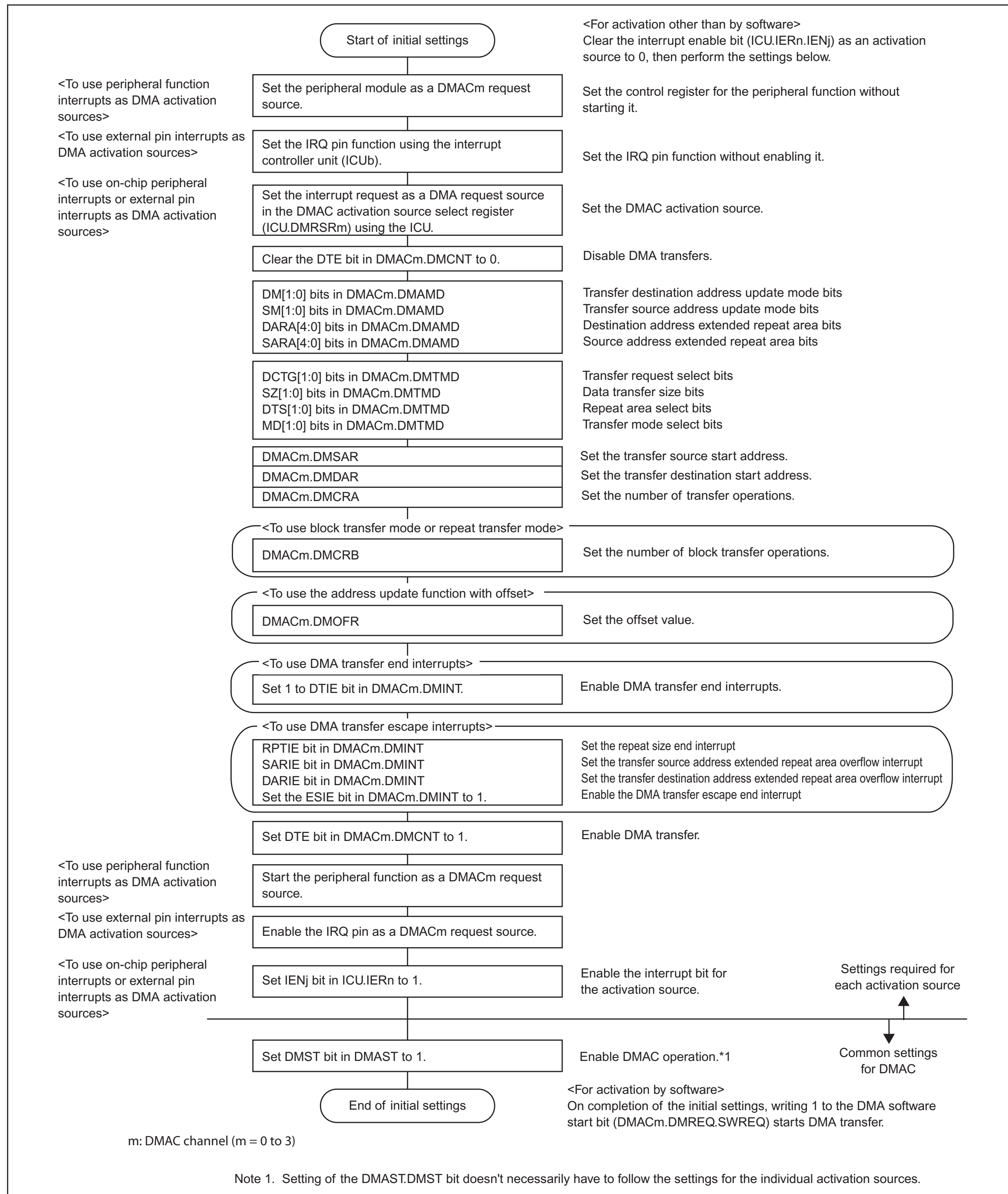


Figure 6.8 Register Setting Procedure [1], page 547.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 125

Explanation of the Code: In line 1 the DMAC Normal Transfer Mode Initialization Function initializes the code. Line 2, to disable the DMA transfer, the DTE bit in DMCNT register is set to 0. Line 3 selects the Normal Data Transfer Mode 'MD' in the DMTMD register, which are set to 00b. Line 4, the DMSAR register is set to source address 0x15000 in the RAM addressing area. Line 5, the DMDAR register is set to destination address 0x17000 in the RAM addressing area. To allow 10 transfers, DMCRA sets the transfer count to 10 (0x00a) in line 6. At line 7 the DTIE bit in the DMINT register is set to 1 to enable the transfer end interrupt request, and in line 8, to enable DMA transfer the DTE bit in the DMCNT register is set to 1. To enable DMAC operation, the DMST bit is set in line 9.

Internal Data Transfer in Repeat Transfer Mode

The following code listing is used for the implementation of a simple initialization program for an internal DMA data transfer in Repeat Transfer Mode. Explanations of each line are included.

```
 1. void DMA_Repeat_Transfer_init() {
 2. DMAC0.DMCNT.BYTE = 0x0;
 3. DMAC0.DMTMD.BIT.MD = 1;
 4. DMAC0.DMSAR = 0x15000;
 5. DMAC0.DMDAR = 0x17000;
 6. DMAC0.DMCRA = 0x000a;
 7. DMAC0.DMCRB = 0x09;
 8. DMAC0.DMINT.BYTE = 0x10;
 9. DMAC0.DMCNT.BYTE = 0x1;
10. DMAC.DMAST.BIT.DMST = 0x1;
11. }
```

Explanation of the Code: Line 1, the DMA Repeat Transfer Mode Initialization Function is initialized. In line 2 the DTE bit in the DMCNT register is set to 0 to disable the DMA transfer. Line 3, the 'MD' bits in DMTMD register are set to 01b to select the DMA Repeat Transfer Mode. Line 4, the DMSAR register is set to source address 0x15000 in the RAM addressing area. At line 5 the DMDAR register is set to the destination address 0x17000 in the RAM addressing area. Line 6, the DMCRA sets the repeat size and transfer count to 10 (0x00a), allowing 10 transfers. At line 7 the DMCRB is set to 0x09, which is the total repeat transfer operations performed. Line 8, the DTIE bit in the DMINT register is set to 1 to enable transfer end interrupt request. At line 9 the DTE bit in DMCNT register is set to 1 to enable the DMA transfer. To enable DMAC operation, the DMST bit is set in line 10.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 126

Internal Data Transfer in Block Transfer Mode

The following code listing is used for the implementation of a simple initialization program for an internal DMA data transfer in Block Transfer Mode. Explanations of each line are included.

```
 1. void DMA_Block_Transfer_init() {
 2.     DMAC0.DMCNT.BYTE = 0x0;
 3.     DMAC0.DMTMD.BIT.MD = 2;
 4.     DMAC0.DMSAR = 0x15000;
 5.     DMAC0.DMDAR = 0x17000;
 6.     DMAC0.DMCRA = 0x000a;
 7.     DMAC0.DMCRB = 0x09;
 8.     DMAC0.DMINT.BYTE = 0x10;
 9.     DMAC0.DMCNT.BYTE = 0x1;
10.     DMAC.DMAST.BIT.DMST = 0x1;
11. }
```

Explanation of the Code: At line 1 is the DMA Block Transfer Mode Initialization function. Line 2 shows the DTE bit in the DMCNT register is set to 0 to disable the DMA transfer. Line 3, the 'MD' bits in the DMTMD register are set to 10b to select the DMA Block Transfer Mode. Line 4, the DMSAR register is set to source address 0x15000 in the RAM addressing area. Line 5, the DMDAR register is set to destination address 0x17000 in the RAM addressing area. Line 6, the DMCRA sets the block size and transfer count to 10 (0x00a), allowing 10 transfers. At line 7 the DMCRB is set to 0x09, which is the total block transfer operations performed. Line 8, the DTIE bit in the DMINT register is set to 1 to enable the transfer end interrupt request. At line 9 the DTE bit in the DMCNT register is set to 1 to enable the DMA transfer. To enable DMAC operation, the DMST bit is set in line 10.

6.4 ADVANCED CONCEPTS

As discussed earlier, the RX63N microcontroller has a 4-channel Direct Memory Access Controller (DMAC) designed especially for internal bus transfer. The RX63N also has a 2-channel External Direct Memory Access Controller (EXDMAC) used exclusively for external bus transfers. Though the basic functionality of both the DMAC and the EXDMAC is the same, there are some specification-related differences. The EXDMAC deals with memory access requested by peripheral modules to the RX63N. Also, external interrupt requests are sent to RX63N requesting the use of EXDMAC.

The operation of the DMAC and the EXDMAC in Normal Transfer Mode, Repeat Transfer Mode, and Block Transfer Mode is similar. The EXDMAC has a special additional mode of data transfer named Cluster Transfer Mode. Figure 6.9 shows the block diagram of the External DMAC for DMA transfers from external peripherals and memory.

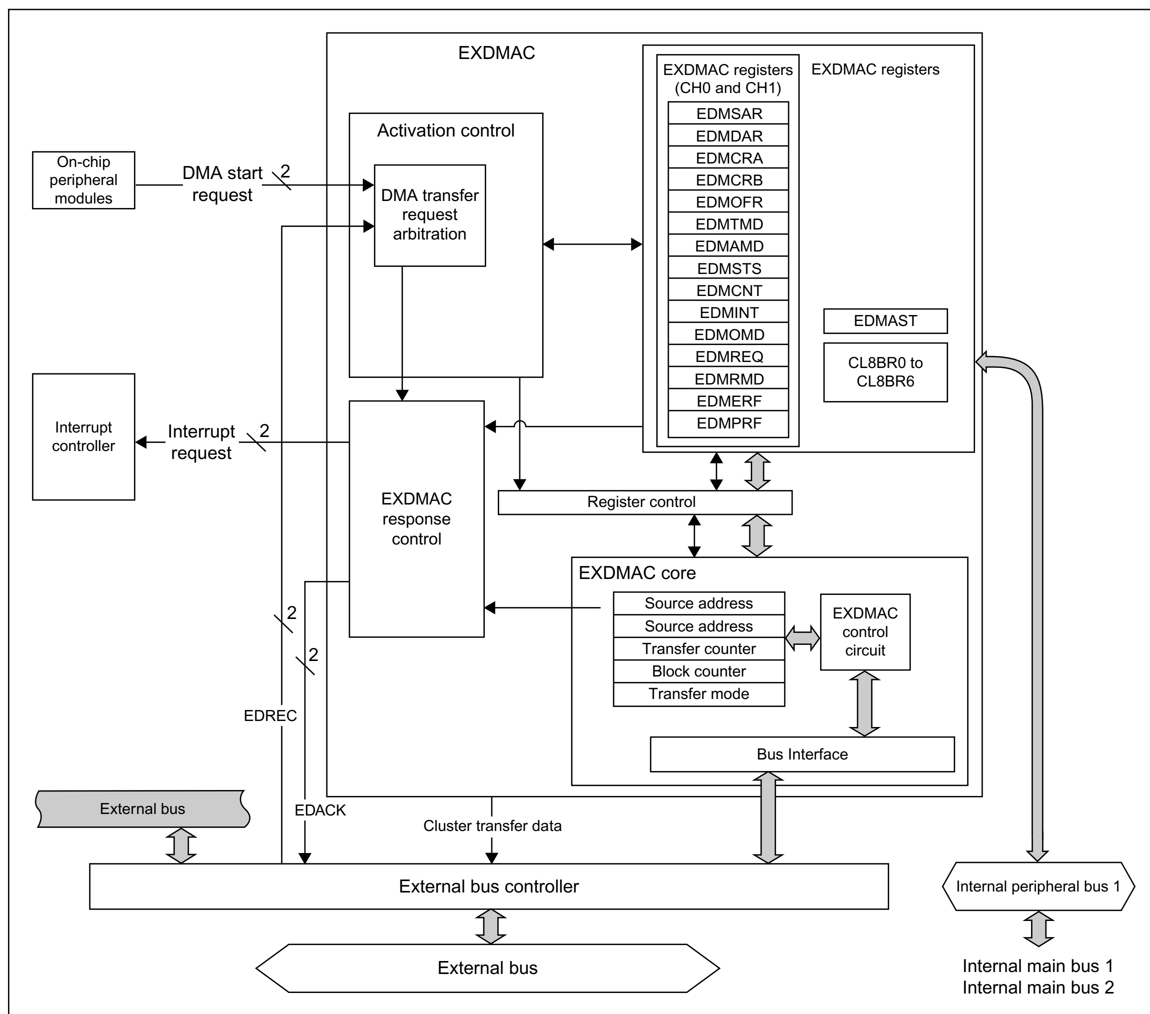


Figure 6.9 Block Diagram of EXDMAC [1], page 556.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 128

The EXDMAC registers are similar in operation to DMAC Registers. The EXDMAC requires some additional special function registers which are described in more detail in the hardware manual [1]:

- Cluster Buffer Register *y* (CLSBRy) (*y* = 0 to 7)
- EXDMA Peripheral Request Flag Register (EDMPRF)
- EXDMA External Request Flag Register (EDMERF)
- EXDMA External Request Sense Mode Register (EDMRMD)

Cluster Transfer Mode

A single cluster of data is transferred by one transfer request, in cluster transfer mode. Using the EDMCRA of the EXDMACn, a maximum of 8 Kbytes of data can be set as a total cluster transfer size. Using the EDMCRB of the EXDMACn, a maximum of 1 Kcount can be set as the number of cluster transfer operation. Hence, a maximum of 8 Kbytes of data × 1Kcount = 8 Mbytes can be set as a total data transfer size.

The cluster transfer mode can be selected from among cluster transfer dual address mode, cluster transfer read address mode, and cluster transfer write address mode.

- **Cluster transfer dual address mode**
  (EXDMACn.EDMTMD.MD[1:0] = 11b, EXDMACn.EDMAMD.AMS = 0)
  A single cluster data is transferred by one transfer request from the transfer source address to the cluster buffers. From the cluster buffers to the transfer destination address, a single cluster data is then transferred.

- **Cluster transfer read address mode**
  (EXDMACn.EDMTMD.MD[1:0] = 11b, EXDMACn.EDMAMD.AMS = 1, EXDMACn.EDMAMD.DIR = 0)
  By one transfer request from the transfer source address to the cluster buffers, a single cluster data is transferred.

- **Cluster transfer write address mode**
  (EXDMACn.EDMTMD.MD[1:0] = 11b, EXDMACn.EDMAMD.AMS = 1, EXDMACn.EDMAMD.DIR = 1)
  By one transfer request from the cluster buffers to the transfer destination address, a single cluster data is transferred.

In the Cluster Transfer Mode, on completion of the transfer of each cluster of data, the external DMA transfer stops and a repeat-size completed interrupt request can be

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 129

generated. By writing 1 to the EXDMACn.EDMCNT.DTE bit during processing of the repeat-size-completed interrupt, the external DMA transfer can be restarted. A repeat-size-completed interrupt request can also be generated once transfer of clusters for a specified number of times is completed. Table 6.5 shows the register update operation in Cluster Transfer Mode, and Figure 6.10 shows the general data flow of the operations.

TABLE 6.5 Register Update Operation in Cluster Transfer Mode [1], page 582.

REGISTER	FUNCTION	UPDATE OPERATION AFTER COMPLETION OF SINGLE-CLUSTER TRANSFER BY ONE TRANSFER REQUEST
EXDMACn.EDMSAR	Transfer source address	■ EXDMACn.EDMTMD.DTS[1:0] = 00b Increment/decrement/fixed/offset addition*[1] ■ EXDMACn.EDMTMD.DTS[1:0] = 01b Initial value of EXDMACn.EDMSAR ■ EXDMACn.EDMTMD.DTS[1:0] = 10b Increment/decrement/fixed/offset addition*[1]
EXDMACn.EDMDAR	Transfer destination address	■ EXDMACn.EDMTMD.DTS[1:0] = 00b Initial value of EXDMACn.EDMDAR ■ EXDMACn.EDMTMD.DTS[1:0] = 01b Increment/decrement/fixed/offset addition*[1] ■ EXDMACn.EDMTMD.DTS[1:0] = 10b Increment/decrement/fixed/offset addition*[1]
EXDMACn.EDMCRAH	Cluster size	Not updated
EXDMACn.EDMCRAL	Transfer count	EXDMACn.EDMCRAH
EXDMACn.EDMCRB	Cluster count	Decremented by one

Note 1. Offset addition can be specified only for EXDMAC0.
In read address mode, the transfer destination address EXDMACn.EDMADAR is fixed (invalid).
In write address mode, the transfer destination address EXDMACn.EDMASAR is fixed (invalid).

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 130

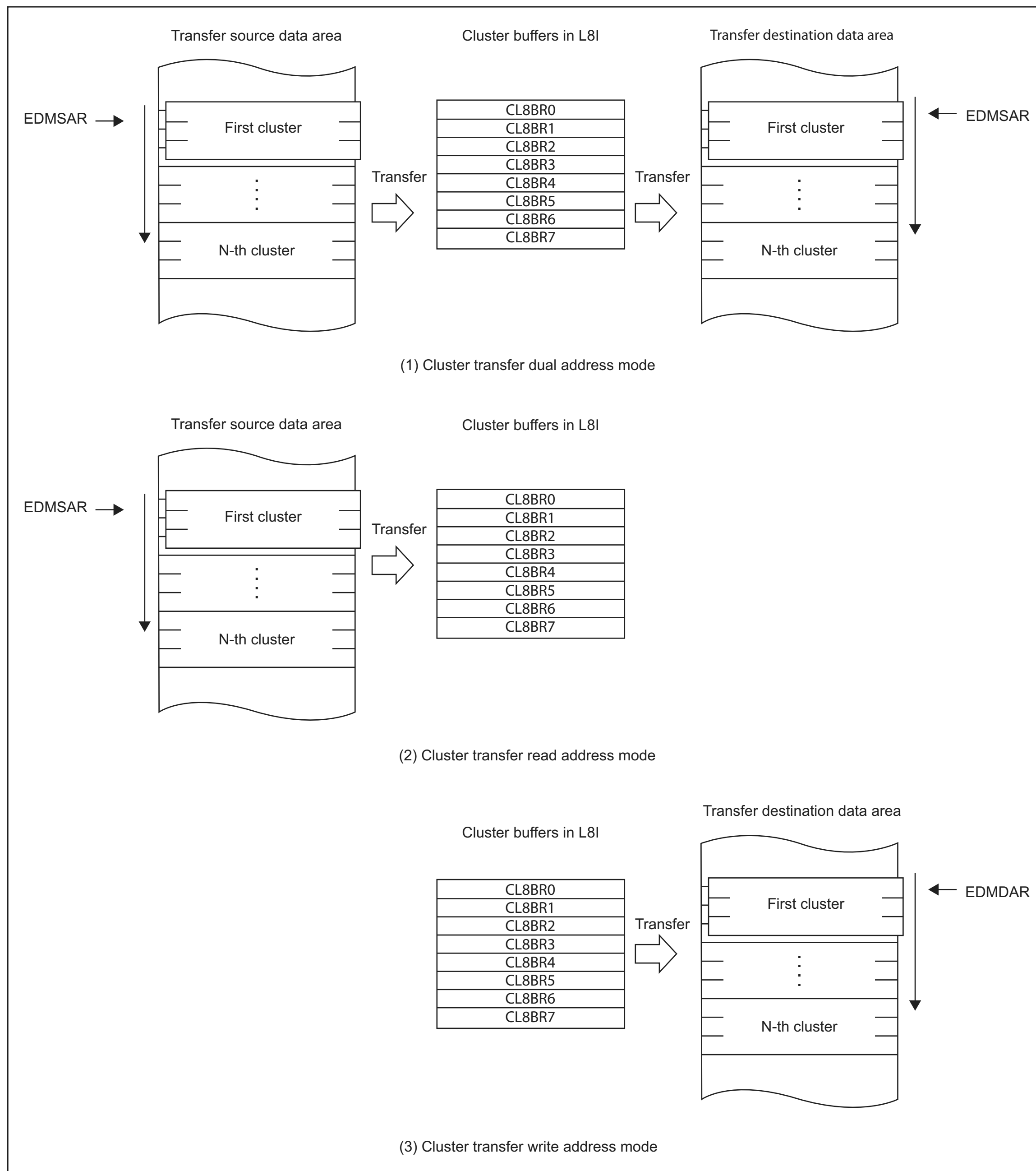


Figure: 6.10 Operation in Cluster Transfer Mode [1], page 583.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 131

6.5 ADVANCED EXAMPLES

External Data Transfer in Normal Transfer Mode

The following code listing is used for the implementation of a simple initialization program for an External DMA data transfer in Normal Transfer Mode. Explanations of each line are included.

```
 1. void EXDMA_Normal_Transfer_init() {
 2.    EXDMAC0.EDMCNT.BYTE = 0x0;
 3.    EXDMAC0.EDMTMD.BIT.MD = 0;
 4.    EXDMAC0.EDMSAR = 0x15000;
 5.    EXDMAC0.EDMDAR = 0x17000;
 6.    EXDMAC0.EDMCRA = 0x00a;
 7.    EXDMAC0.EDMINT.BYTE = 0x10;
 8.    EXDMAC0.EDMCNT.BYTE = 0x1;
 9.    EDMAC.DMAST.BIT.DMST = 0x1;
10. }
```

Explanation of the Code: Line 1 shows the EXDMAC Normal Transfer Mode Initialization Function. Line 2, the DTE bit in the EDMCNT register is set to 0 to disable the EXDMA transfer. Line 3, the 'MD' bits in the EDMTMD register are set to 00b to select the Normal Data Transfer Mode. At line 4, the EDMSAR register is set to source address 0x15000 in the RAM addressing area. Line 5, the EDMDAR register is set to the destination address 0x17000 in the RAM addressing area. Line 6, the EDMCRA sets the transfer count to 10 (0x00a), allowing 10 transfers. At line 7, the DTIE bit in the EDMINT register is set to 1 to enable the transfer end interrupt request. At line 8, the DTE bit in the EDMCNT register is set to 1 to enable the EXDMA transfer. To enable EDMAC operation, the DMST bit is set in line 9.

External Data Transfer in Repeat Transfer Mode

The following code listing is used for the implementation of a simple initialization program for an External DMA data transfer in Repeat Transfer Mode. Explanations of each line are included.

```
 1. void EXDMA_Repeat_Transfer_init() {
 2.    EXDMAC0.EDMCNT.BYTE = 0x0;
 3.    EXDMAC0.EDMTMD.BIT.MD = 1;
 4.    EXDMAC0.EDMSAR = 0x15000;
 5.    EXDMAC0.EDMDAR = 0x17000;
```

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 132


```
 6.     EXDMAC0.EDMCRA = 0x000a;
 7.     EXDMAC0.EDMCRB = 0x09;
 8.     EXDMAC0.EDMINT.BYTE = 0x10;
 9.     EXDMAC0.EDMCNT.BYTE = 0x1;
10.     EDMAC.DMAST.BIT.DMST = 0x1;
11. }
```

Explanation of the Code: Line 1 shows the EXDMA Repeat Transfer Mode Initialization Function. Line 2 shows the DTE bit in the EDMCNT register is set to 0 to disable the EXDMA transfer. Line 3 shows the 'MD' bits in the EDMTMD register are set to 01b to select the EXDMA Repeat Transfer Mode. At line 4 the EDMSAR register is set to source address 0x15000 in the RAM addressing area. At line 5 the EDMDAR register is set to destination address 0x17000 in the RAM addressing area. Line 6, the EDMCRA sets the repeat size and transfer count to 10 (0x00a), allowing 10 transfers. Line 7, the EDMCRB is set to 0x09 which is the total repeat transfer operations performed. At line 8, the DTIE bit in EDMINT register is set to 1 to enable the transfer end interrupt request. Line 9 shows the DTE bit in the EDMCNT register is set to 1 to enable the EXDMA transfer. To enable EDMAC operation, the DMST bit is set in line 10.

External Data Transfer in Block Transfer Mode

The following code listing is used for the implementation of a simple initialization program for an External DMA data transfer in Block Transfer Mode. Explanations of each line are included.

```
 1. void EXDMA_Block_Transfer_init() {
 2.     EXDMAC0.EDMTMD.BIT.MD = 2;
 3.     EXDMAC0.EDMINT.BYTE = 0x10;
 4.     EXDMAC0.EDMCNT.BYTE = 0x0;
 5.     EXDMAC0.EDMSAR = 0x15000;
 6.     EXDMAC0.EDMDAR = 0x17000;
 7.     EXDMAC0.EDMCRA = 0x000a;
 8.     EXDMAC0.EDMCRB = 0x09;
 9.     EXDMAC0.EDMCNT.BYTE = 0x1;
10.     EDMAC.DMAST.BIT.DMST = 0x1;
11. }
```

Explanation of the Code: Line 1 shows the EXDMA Block Transfer Mode Initialization function. Line 2, the DTE bit in the EDMCNT register is set to 0 to disable the EXDMA transfer. At line 3, the 'MD' bits in the EDMTMD register are set to 10b to select the EXDMA Block Transfer Mode. Line 4, the EDMSAR register is set to source address

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 133

0x15000 in the RAM addressing area. Line 5, the EDMDAR register is set to destination address 0x17000 in the RAM addressing area. At line 6 the EDMCRA sets the block size and transfer count to 10 (0x00a), allowing 10 transfers. Line 7, the EDMCRB is set to 0x09 which is the total block transfer operations performed. Line 8 shows the DTIE bit in the EDMINT register is set to 1 to enable the transfer end interrupt request. Line 9, the DTE bit in EDMCNT register is set to 1 to enable the EXDMA transfer. To enable EDMAC operation, the DMST bit is set in line 10.

External Data Transfer in Cluster Transfer Mode

The following code listings are used for the implementation of simple initialization programs for an External DMA data transfer in Cluster Transfer Mode. Explanations of each line are included.

Code for Cluster Transfer Dual Address Mode:

```
 1. void EXDMA_Cluster_Transfer1_init() {
 2. EXDMAC0.EDMCNT.BYTE = 0x0;
 3. EXDMAC0.EDMAMD.BIT.AMS = 0;
 4. EXDMAC0.EDMTMD.BIT.MD = 3;
 5. EXDMAC0.EDMSAR = 0x15000;
 6. EXDMAC0.EDMDAR = 0x17000;
 7. EXDMAC0.EDMCRA = 0x000a;
 8. EXDMAC0.EDMCRB = 0x09;
 9. EXDMAC0.EDMINT.BYTE = 0x10;
10. EXDMAC0.EDMCNT.BYTE = 0x1;
11. EDMAC.DMAST.BIT.DMST = 0x1;
12. }
```

Explanation of the Code: Line 1 shows the EXDMA Cluster Transfer Mode: Dual Address Mode Initialization function. Line 2, the DTE bit in the EDMCNT register is set to 0 to disable the EXDMA transfer. At line 3, Bit 17: AMS in EDMAMD is set to 0, which is the selection of Dual Address Mode. Line 4, the 'MD' bits in the EDMTMD register are set to 11b (3) to select the EXDMA Cluster Transfer Mode. Line 5, the EDMSAR register is set to source address 0x15000 in the RAM addressing area. At line 6 the EDMDAR register is set to destination address 0x17000 in the RAM addressing area. Line 7, the EDMCRA sets the cluster size and transfer count to 10 (0x00a), allowing 10 transfers. Line 8, the EDMCRB is set to 0x09, which is the total cluster transfer operations performed. Line 9, the DTIE bit in the EDMINT register is set to 1 to enable transfer end interrupt request. At line 10 the DTE bit in the EDMCNT register is set to 1 to enable the EXDMA transfer. To enable EDMAC operation, the DMST bit is set in line 11.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 134

Code for Cluster Transfer Read Address Mode:

```
 1. void EXDMA_Cluster_Transfer2_init() {
 2.    EXDMAC0.EDMCNT.BYTE = 0x0;
 3.    EXDMAC0.EDMAMD.BIT.AMS = 1;
 4.    EXDMAC0.EDMAMD.BIT.DIR = 0;
 5.    EXDMAC0.EDMTMD.BIT.MD = 3;
 6.    EXDMAC0.EDMSAR = 0x15000;
 7.    EXDMAC0.EDMDAR = 0x17000;
 8.    EXDMAC0.EDMCRA = 0x000a;
 9.    EXDMAC0.EDMCRB = 0x09;
10.    EXDMAC0.EDMINT.BYTE = 0x10;
11.    EXDMAC0.EDMCNT.BYTE = 0x1;
12.    EDMAC.DMAST.BIT.DMST = 0x1;
13. }
```

Explanation of the Code: Line 1 shows the EXDMA Cluster Transfer Mode: Read Address Mode Initialization function. Line 2, the DTE bit in the EDMCNT register is set to 0 to disable the EXDMA transfer. At line 3, the Bit 17: AMS in the EDMAMD is set to 1, which is the selection of Single Address Mode. Line 4, the DIR bit in the EDMAMD is set to 0 to select the EDMSAR as the transfer source register to in turn select the Read Address Mode. At line 5 the 'MD' bits in the EDMTMD register are set to 11b (3) to select the EXDMA Cluster Transfer Mode. Line 6, the EDMSAR register is set to source address 0x15000 in the RAM addressing area. Line 7, the EDMDAR register is set to destination address 0x17000 in the RAM addressing area. Line 8, the EDMCRA sets the cluster size and transfer count to 10 (0x00a), allowing 10 transfers. Line 9, the EDMCRB is set to 0x09, which is the total cluster transfer operations performed. Line 10, the DTIE bit in the EDMINT register is set to 1 to enable the transfer end interrupt request. At line 11 the DTE bit in the EDMCNT register is set to 1 to enable the EXDMA transfer. To enable EDMAC operation, the DMST bit is set in line 12.

Code listing for Cluster Transfer Write Address Mode:

```
 1. void EXDMA_Cluster_Transfer3_init() {
 2.    EXDMAC0.EDMCNT.BYTE = 0x0;
 3.    EXDMAC0.EDMAMD.BIT.AMS = 1;
 4.    EXDMAC0.EDMAMD.BIT.DIR = 1;
 5.    EXDMAC0.EDMTMD.BIT.MD = 3;
 6.    EXDMAC0.EDMSAR = 0x15000;
 7.    EXDMAC0.EDMDAR = 0x17000;
 8.    EXDMAC0.EDMCRA = 0x000a;
```

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 135


```
 9.     EXDMAC0.EDMCRB = 0x09;
10.     EXDMAC0.EDMINT.BYTE = 0x10;
11.     EXDMAC0.EDMCNT.BYTE = 0x1;
12.     EDMAC.DMAST.BIT.DMST = 0x1;
13. }
```

Explanation of the Code: Line 1 shows EXDMA Cluster Transfer Mode: Write Address Mode Initialization function. Line 2, the DTE bit in the EDMCNT register is set to 0 to disable the EXDMA transfer. Line 3, Bit 17: AMS in the EDMAMD is set to 1, which is the selection of the Single Address Mode. Line 4, the DIR bit in the EDMAMD is set to 1 to select the EDMSAR as the transfer source register to, in turn, select the Write Address Mode. At line 5 the 'MD' bits in the EDMTMD register are set to 11b (3) to select the EXDMA Cluster Transfer Mode. At line 6 the EDMSAR register is set to source address 0x15000 in the RAM addressing area. Line 7, the EDMDAR register is set to destination address 0x17000 in the RAM addressing area. Line 8, the EDMCRA sets the cluster size and transfer count to 10 (0x00a), allowing 10 transfers. Line 9, the EDMCRB is set to 0x09, which is the total cluster transfer operations performed. At line 10 the DTIE bit in the EDMINT register is set to 1 to enable the transfer end interrupt request. Line 11, the DTE bit in the EDMCNT register is set to 1 to enable the EXDMA transfer. To enable EDMAC operation, the DMST bit is set in line 12.

6.6 EXAMPLES WITH INTERRUPTS

The following example is available in the Renesas RX63N HEW sample code directories for the DMAC. The `Init_DMAC` function configures DMAC channel 0 for a single block software-triggered transfer. The transfer source address is set to fixed and defined to be that of the variable `gDMA_DataSource`. In fact, the source address holds the character 'X'. The transfer destination address is set to be incremented after each transfer and defined to be the address of buffer `gDMA_DataBuff`. Each location of this 1024 byte buffer will, at the end of this DMA operation, be filled with the character 'X'. A transfer interrupt request to the CPU is set to occur after all data has transferred. The function `Init_DMAC` would need to be called from your own code.

Many of the DMAC registers are used in similar ways to the earlier examples, like DMCNT, DMTMD, DMSAR, DMDAR, DMCRA, DMCRB, and DMINT. Register DMAMD is set so that a one-to-many transfer is made. DMREQ (Software Start Register) is set to 1 to generate a DMA transfer request. The DMA Activation Source Flag Control Register (DMCSL) is set so that at the end of transfer, the interrupt flag of the activation source issues an interrupt to the CPU. See the Hardware manual [1] for more detail on the use of these registers.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 136


```
 1. #include <stdint.h>        //standard integer type definitions
 2. #include "iodefine.h"       //Defines RX63N port registers
 3. #include "rskrx63ndef.h"    //Defines macros of RX63N user LEDs and switches
 4. #include "vect.h"           //Defines interrupt prototypes used in this file
 5.
 6. uint8_t gDMA_DataBuff[1024];      //DMAC Destination buffer
 7. uint8_t gDMA_DataSource = 'X';    //DMAC transfer source variable
 8. void Init_DMAC(void);
 9.
10. void Init_DMAC(void) {
11.    DMAC0.DMCNT.BIT.DTE = 0;    //Disable DMA transfers
12.
13.    //Transfer dest address is incremented, source address is fixed
14.    DMAC0.DMAMD.WORD = 0x0080;
15.
16.    //Configure Block transfer, the source is specified as the block area
17.    //area 8-bit transfer data size, Software request source
18.    DMAC0.DMTMD.WORD = 0x9000;
19.
20.    DMAC0.DMSAR = &gDMA_DataSource;         //Global variable source
21.    DMAC0.DMDAR = gDMA_DataBuff;            //Global variable dest
22.    DMAC0.DMCRA = sizeof(gDMA_DataBuff);    //Number of moves
23.    DMAC0.DMCRB = 0x1;                      //Block transfer ops = 1
24.
25.    //Set activation source's intrpt flag to issue an interrupt to CPU
26.    DMAC0.DMCSL.BIT.DISEL = 0x1;
27.    DMAC0.DMINT.BIT.DTIE = 0x1;     //Enable DMA transfer end interrupts
28.    IPR(DMAC,DMAC0I) = 0x5;         //Set DMAC0 interrupt priority lvl to 5
29.    IEN(DMAC,DMAC0I) = 0x1;         //Enable DMAC0 interrupts
30.    IR(DMAC,DMAC0I) = 0x0;          //Clear DMAC0 interrupt flag
31.    DMAC0.DMCNT.BIT.DTE = 0x1;      //Enable DMA transfers
32.    DMAC.DMAST.BIT.DMST = 0x1;      //Enable DMAC operation
33.    DMAC0.DMREQ.BIT.SWREQ = 0x1;    //Enable and trigger the DMAC transfer
34. }
35.
36. //Interrupt function handler called on the completion of the block
37. //block transfer. The status of the DMAC is read and if
38. //the transfer was completed successfully, LED1 is turned on
39. void Excep_DMACA_DMAC0(void) {
40.    if(DMAC0.DMSTS.BIT.DTIF) {      //Read the status for channel 0
41.       LED1 = LED_ON;               //Turn on LED1 to indicate end of xfer
```

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 137


```
42.        DMAC0.DMSTS.BIT.DTIF = 0;    //Clear the transfer end interrupt flag
43.    }
44. }
```

Lines 28 to 30 set up the interrupt levels and activate interrupts for notifying the CPU when the DMA transfer is complete. When complete, the interrupt service routine (lines 39 to 44) will run and simply turn on an LED. The DMA Status Register (DMSTS), bit 4 (DTIF), holds a value which is 1 when a transfer end interrupt has been generated. The bit needs to be clears, which is done in line 42.

6.7 RECAP

In this chapter, we introduced the DMAC and the EXDMAC modules in the Renesas RX63N. Direct Memory Access (DMA) is used to transfer data without the supervision of the processor. Several uses and advantages of using the DMA have been identified. A better understanding of the DMAC is provided with respect to the DMAC block diagram and the DMAC functionality list. The main data transfer modes for the DMAC (Normal, Repeat, and Block Transfer Modes) have been explained with the help of diagrams and illustrative code examples. We have discussed the need for having an EXDMAC in the RX63N for DMA transfer in an external data bus. The EXDMAC functionality list and several special function registers have also been introduced. The EXDMAC's four main data transfer modes (namely Normal, Repeat, Block, and Cluster Transfer Modes) have been explained with the help of diagrams and code examples.

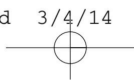
6.8 REFERENCES

[1] Renesas Electronics, Inc. (February, 2013). *RX63N Group, RX631 Group User's Manual: Hardware*, Rev. 1.60.

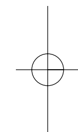
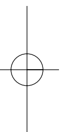
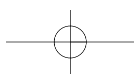
6.8 EXERCISES

1. What are the main EXDMAC registers to be updated during a DMA transfer?
2. What are the primary DMAC Activation Sources?
3. What are the main DMAC Registers to be updated during a DMA transfer?
4. List the main transfer modes for the DMAC and the EXDMAC.
5. What are the primary EXDMAC activation sources?
6. List the similarities between the DMAC and the EXDMAC.

06.ES_Conrad_RX63N_Advanced_CH06.qxd:RX63N Advanced 3/4/14 9:51 PM Page 138



7. What are the three sub-modes of Cluster Transfer Mode in the EXDMAC?
8. List three advantages of using a DMAC transfer.
9. List major differences between the DMAC and the EXDMAC.
10. List three registers with functions unique to the EXMDAC.
11. Write code to initialize Cluster Transfer Mode in the EXDMAC in Dual Address Mode.
12. Write code to initialize Cluster Transfer Mode in the EXDMAC in Read Address Mode.
13. Write code to initialize Cluster Transfer Mode in the EXDMAC in Write Address Mode.
14. Write code to set the following DMAC and EXDMAC Registers for different functionality:
    a. Select 'Normal Transfer Mode' for EXDMAC channel 0.
    b. Disable the DMA transfer on DMAC channel 1.
    c. Enable the DMA transfer on DMAC channel 2.
    d. Enable the Transfer End Interrupt Request for EXDMAC channel 1.
    e. Select the Cluster Transfer Mode: Read Address Mode on EXDMAC channel 0.

Flash and EEPROM Programming

7.1 LEARNING OBJECTIVES

In this chapter, the reader will learn:

- a basic understanding of flash memory and EEPROM.
- about the block configuration of ROM and E2 DataFlash.
- about flash memory register descriptions.
- how to set up the Flash Control Unit (FCU) for flash and EEPROM programming.
- how to use the different operating modes of the FCU and the RX63N.
- how to write to the flash and EEPROM.
- about protection features of the RX63N.

7.2 BASIC CONCEPTS

Electrically Erasable Programmable Read-Only Memory (EEPROM) is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed.

Flash memory refers to a particular type of EEPROM. The difference between flash memory and regular EEPROM is that EEPROM erases its content 1 byte at a time. This makes it slow to update. Flash memory can erase its data in entire blocks, making it a preferable technology for applications that require frequent updating of large amounts of data, as in the case of a memory stick for a digital electronic device.

Two ROMs are provided on the RX63N—Zero wait flash (for program storage) of up to 2 MB, and E2 DataFlash (for data storage) of 32 KB.

7.2.1 Flash Memory Overview

Table 7.1 lists the specifications of the ROM and E2 DataFlash memory and Figure 7.1 is a block diagram of the ROM, E2 DataFlash memory, and related modules.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 140

TABLE 7.1 Specifications of ROM/E2 DataFlash [1], page 1729.

<table>
<tr><th>ITEM</th><th>ROM</th><th>E2 DATAFLASH</th></tr>
<tr><td>Memory space</td><td>User area: 2 Mbytes max.
User boot area: 16 Kbytes</td><td>Data area: 32 Kbytes</td></tr>
<tr><td>Read cycle</td><td>A read operation takes one cycle of ICLK.</td><td>A read operation takes six cycles of FCLK in words or bytes.</td></tr>
<tr><td>Programming/ erasing method</td><td colspan="2">■ The chip incorporates a dedicated sequencer (FCU) for programming of the ROM/E2 DataFlash.
■ Programming and erasing the ROM/E2 DataFlash are handled by issuing commands to the FCU.</td></tr>
<tr><td>Value after erasure</td><td>FFh</td><td>Undefined</td></tr>
<tr><td>BGO (background operation)</td><td colspan="2">The CPU is able to execute program code from the ROM while the E2 DataFlash memory is being programmed or erased.</td></tr>
<tr><td>Suspension and resumption</td><td colspan="2">■ The CPU is able to execute program code from the ROM during suspension of programming or erasure.
■ Programming and erasure of the ROM/E2 DataFlash can be restarted (resumed) after suspension.</td></tr>
<tr><td>Units of programming and erasure</td><td>■ Units of programming for the user area or user boot area: 128 bytes
■ Units of erasure for the user area: In block units
■ Units of erasure for the user boot area: 16 Kbytes</td><td>■ Unit of programming for the data area: 2 bytes
■ Unit of erasure for the data area: 32 bytes</td></tr>
<tr><td>On-board programming (four types)</td><td colspan="2">Programming in boot mode:
■ The asynchronous serial interface (SCI1) is used.
■ The transfer rate is adjusted automatically.
■ The user boot area can also be programmed.

Programming in USB boot mode:
■ USB0 is used.
■ Dedicated hardware is not required, so direct connection to a PC is possible.

Programming in the user boot mode:
■ Able to create original boot programs of the user’s making.

Programming by a routine for ROM/E2 DataFlash programming within the user program:
■ This allows ROM/E2 DataFlash programming without resetting the system.</td></tr>
<tr><td>Off-board programming (for products with 100 pins or more)</td><td>A Flash programmer can be used to program the user area and user boot area.</td><td>A Flash programmer cannot be used to program the data area.</td></tr>
<tr><td>Protection—Software controlled protection</td><td>The registers and lock bits can be set to prevent unintentional programming.</td><td>■ The registers can be used to prevent unintentional programming or reading.
■ Protection with the registers is performed on a 2-Kbyte basis.</td></tr>
<tr><td>Protection—FCU command—lock</td><td colspan="2">When abnormal operations are detected during programming/erasure, this function disables any further programming/erasure.</td></tr>
</table>

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 141

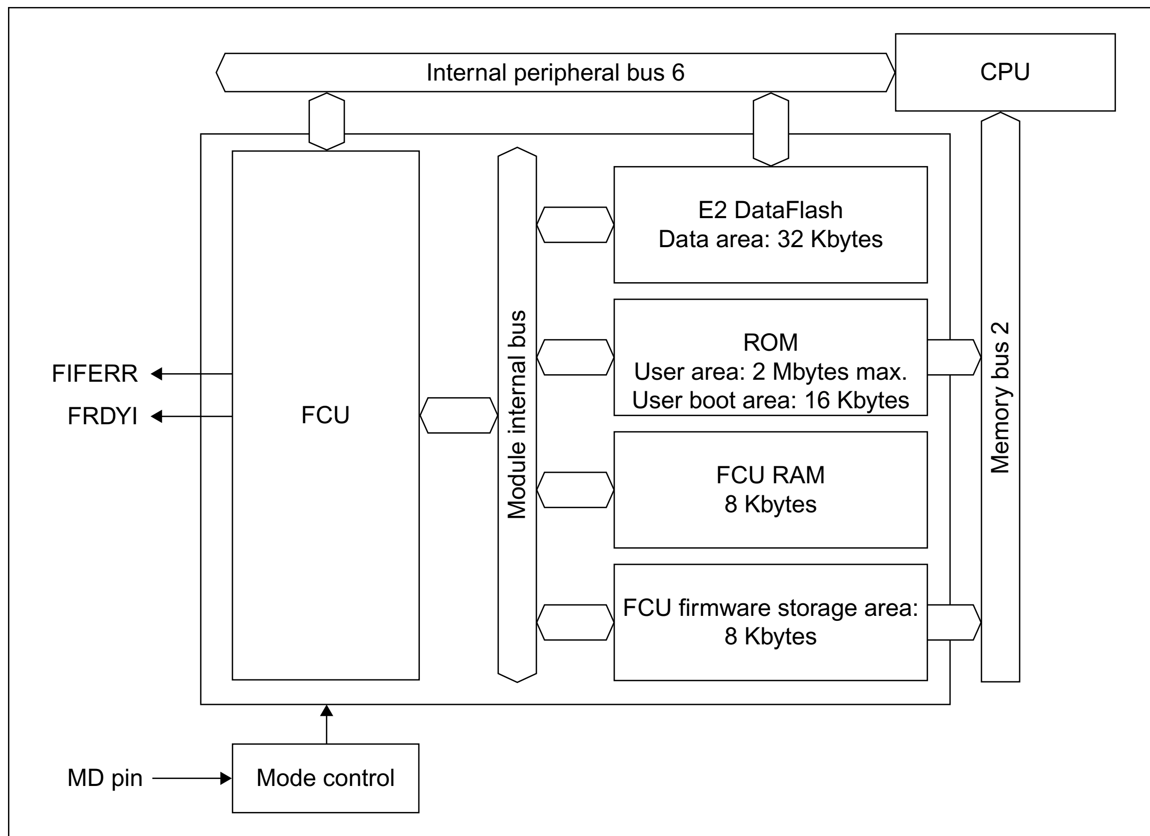


Figure 7.1 Block Diagram of ROM/E2 DataFlash [1], page 1730.

Figure 7.1 shows the block diagram of the ROM and the E2 DataFlash. The commands issued to the ROM and the E2 DataFlash are handled by the FCU, which is discussed in a later section. A memory bus acts as an interface between the ROM, the E2 DataFlash, and the CPU. An internal bus communicates with the ROM, the E2 DataFlash, and the FCU. The FCU has its own RAM and storage area.

7.2.2 Operating Modes Associated with Flash Memory

The ROM and the E2 DataFlash can be read, programmed, and erased on the board in boot mode, USB boot mode, user boot mode, single-chip mode (with on-chip ROM enabled), or on-chip ROM enabled extended mode. The area where programming and erasure are permitted, the area from which booting up proceeds, and areas erased at the time of booting up differ with the mode. The differences between modes are indicated in Table 7.2.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 142

TABLE 7.2 Difference Between Modes.

ITEM	BOOT MODE	USB BOOT MODE	USER BOOT MODE	SINGLE CHIP MODE
Environment for program-ming and erasure	On-board programming	On-board programming	On-board programming	On-board programming
Programming and erasable area	User area/user boot area/data area	User area/data area	User area/data area	User area/data area
Division into erasable blocks	Possible	Possible	Possible	Possible
Boot program at reset	Boot program	USB boot program	User boot program	User program

Important observations made from the table are:

- Programming and erasure of the user boot area are only possible in boot mode.
- In boot mode, a host is able to program, erase, or read the user area, user boot area, or data area via a Serial Communications Interface (SCI).
- In boot mode, on-chip RAM is employed for the boot program. Therefore, the data on the on-chip RAM is not retained.
- Booting-up in USB boot mode and user boot mode is from the user boot area. The user boot area of the product as shipped holds the USB boot program, which is capable of reading from or writing to the user area and data area.
- Furthermore, rewriting of the user boot area in boot mode can enable reading from or writing to the user area and data area via any desired interface.

7.2.3 Block Configuration of the ROM

The user area (area 0 to area 3) is divided into blocks with different sizes, and erasure proceeds in block units. The configuration of the blocks of the user area is shown in the Figure 7.2.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 143

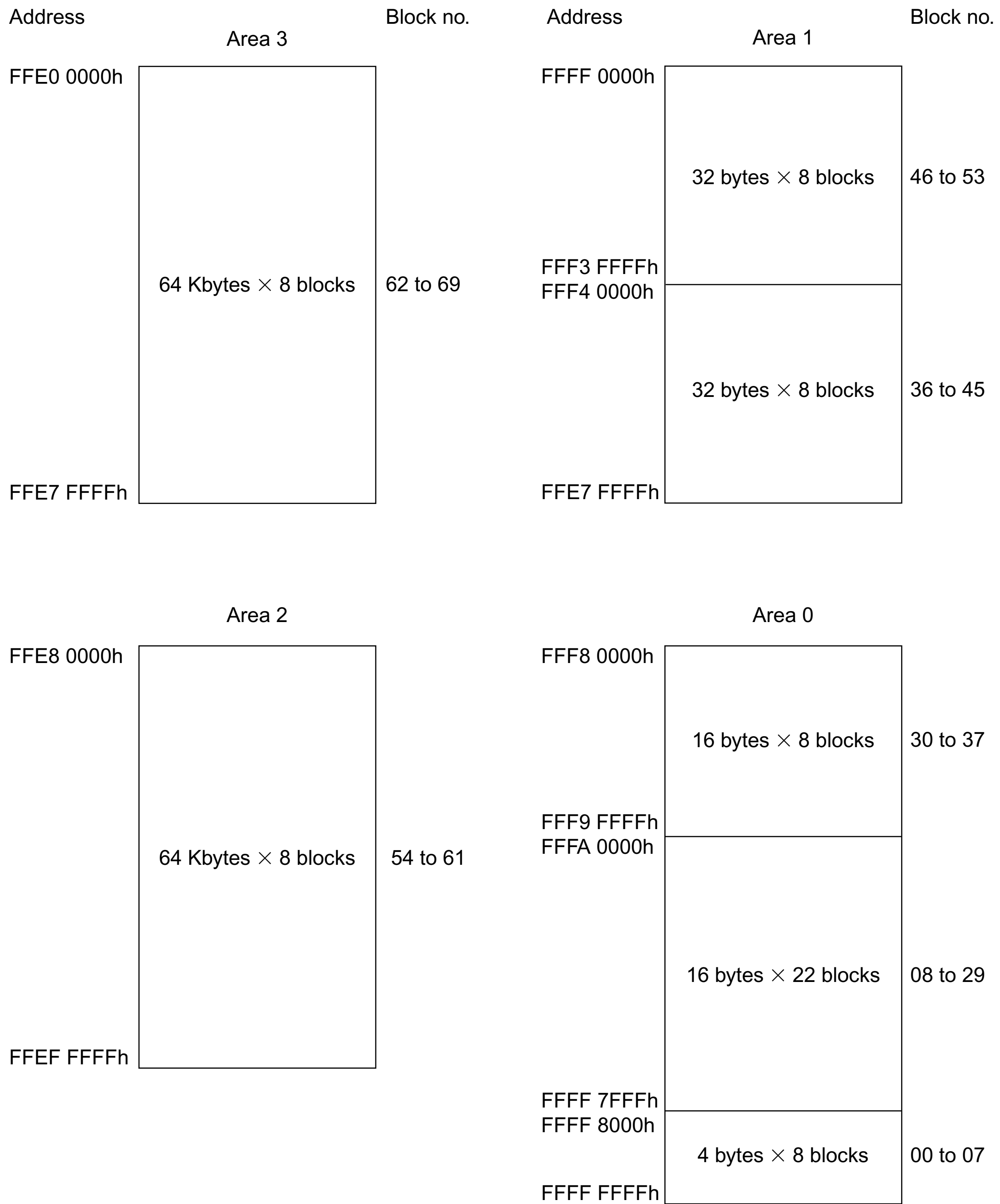


Figure 7.2 Block Configuration of the User Area [1], page 1732.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 144

7.2.4 Block Configuration of the E2 DataFlash

The data area is divided into 1024 blocks, and erasure is executed in block units. The relations between the blocks and addresses of the data area, and the corresponding bits to support permission of reading and of programming and erasure are listed in Table 7.3. The address of block N (address where block N starts) is calculated from the following formula:

$$\text{Address of block } N = (N \times 32) + \text{address where the data area starts (0010 0000h).}$$

TABLE 7.3 Block Configuration of the Data Area [1], page 1734.

BLOCK NO	START ADDRESS	READING AND PROGRAMMING/ ERASURE ENABLE BIT	BLOCK NO	START ADDRESS	READING AND PROGRAMMING/ ERASURE ENABLE BIT
0000–	0010 0000h-	DFLRE0.DBRE00	0512–	0010 4000h-	DFLRE1.DBRE08
0063	0010 07E0h	DFLWE0.DBWE00	0575	0010 47E0h	DFLWE1.DBWE08
0064–	0010 0800h-	DFLRE0.DBRE01	0576–	0010 4800h-	DFLRE1.DBRE09
0127	0010 0FE0h	DFLWE0.DBWE01	0639	0010 4FE0h	DFLWE1.DBWE09
0128–	0010 1000h-	DFLRE0.DBRE02	0640–	0010 5000h-	DFLRE1.DBRE10
0191	0010 17E0h	DFLWE0.DBWE002	0703	0010 57E0h	DFLWE1.DBWE10
0192–	0010 1800h-	DFLRE0.DBRE03	0704–	0010 5800h-	DFLRE1.DBRE11
0255	0010 1FE0h	DFLWE0.DBWE03	0767	0010 5FE0h	DFLWE1.DBWE11
0256–	0010 2000h-	DFLRE0.DBRE04	0768–	0010 6000h-	DFLRE1.DBRE12
0319	0010 27E0h	DFLWE0.DBWE04	0831	0010 67E0h	DFLWE1.DBWE12
0320–	0010 2800h-	DFLRE0.DBRE05	0832–	0010 6800h-	DFLRE1.DBRE13
0383	0010 2FE0h	DFLWE0.DBWE05	0895	0010 6FE0h	DFLWE1.DBWE13
0384–	0010 3000h-	DFLRE0.DBRE06	0896–	0010 7000h-	DFLRE1.DBRE14
0447	0010 37E0h	DFLWE0.DBWE06	0959	0010 77E0h	DFLWE1.DBWE14
0448–	0010 3800h-	DFLRE0.DBRE07	0960–	0010 7800h-	DFLRE1.DBRE15
0511	0010 3FE0h	DFLWE0.DBWE07	1023	0010 7FE0h	DFLWE1.DBWE15

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 145

7.2.5 FCU

The ROM and the E2 DataFlash operations are performed by issuing commands to the FCU, a dedicated sequencer. The mode transitions of the FCU and the system of commands are described as follows. The descriptions apply in common to boot mode, USB boot mode, user boot mode, single-chip mode (with on-chip ROM enabled), and on-chip ROM enabled expansion mode.

FCU Modes

The FCU has eight modes. Transitions between modes are caused by modifying FENTRYR or issuing FCU commands. Since the E2 DataFlash P/E mode is included in ROM read mode, high-speed reading from the ROM is possible in E2 DataFlash P/E mode.

1. **ROM Read Modes:** The ROM read modes are for high-speed reading of the ROM. Reading from an address can be accomplished in one cycle of ICLK.
2. **ROM/E2 DataFlash Read Mode:** This mode is for reading the ROM and E2 DataFlash memory. The FCU does not accept FCU commands. The FCU enters this mode when the FENTRYR.FENTRYn bits (n = 0 to 3) are set to 0 with the FENTRYR.FENTRYD bit set to 0.
3. **ROM P/E Modes:** The ROM P/E modes are for programming and erasure of the ROM. High-speed reading of the ROM is not possible in these modes. Reading from an address within the range for reading causes a ROM-access violation and the FASTAT.CMDLK bit is set to 1. There are three ROM P/E modes:
   a. **ROM P/E Normal Mode:** The transition to ROM P/E normal mode is the first transition in the process of programming or erasing the ROM. The FCU enters this mode when the FENTRYR.FENTRYD bit is set to 0, with any of the FENTRYR.FENTRYn bits (n = 0 to 3) set to 1 in ROM read mode, or when the normal mode transition command is received in ROM P/E modes. Reading from an address within the range for programming and erasure while any of the FENTRYR.FENTRYn bits (n = 0 to 3) are set to 1 causes a ROM-access violation, and the FASTAT.CMDLK bit is set to 1 (command-locked state).
   b. **ROM Status Read Mode:** In the ROM status read mode, the state of the ROM can be read. The FCU enters this mode when a status read mode transition command is received, or when a command other than the normal mode transition or lock-bit read mode transition command is received in ROM P/E modes.ROM status read mode encompasses the states where the FSTATR0. FRDY bit is 0 and the FASTAT.CMDLK bit is set to1 (command-locked state) after an error has occurred. Reading from an address within the range for

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 146

programming and erasure while any of the FENTRYR.FENTRYn bits (n = 0 to 3) are 1 allows the value of FSTATR0 to be read.

c. **ROM Lock-Bit Read Mode:** In the ROM lock-bit read mode, reading the ROM allows the lock bits to be read. The FCU enters this mode when a lock-bit read mode transition command is received in ROM P/E modes. Reading from an address within the range for programming and erasure while any of the FENTRYR.FENTRYn bits (n = 0 to 3) are 1 allows the value of the lock bit of the block including the accessed address to be read from all the read bits.

4. **E2 DataFlash P/E Modes:** These modes are for programming and erasure of the E2 DataFlash memory. Although high-speed reading from the ROM is possible, reading from the E2 DataFlash is not allowed. Although FCU commands for the E2 flash memory are accepted in this mode, FCU commands for the ROM are not. The FCU enters this mode when the FENTRYR.FENTRYn bits (n = 0 to 3) are all set to 0 and the FENTRYR.FENTRYD bit is set to 1. There are three E2 DataFlash P/E modes:

   a. **E2 DataFlash P/E Normal Mode:** The transition to E2 DataFlash P/E normal mode is the first transition in the process of programming or erasing the E2 DataFlash. The FCU enters this mode when the FENTRYR.FENTRYD bit is set to 1 and the FENTRYR.FENTRYn bits (n = 0 to 3) are all set to 0 in ROM/E2 DataFlash read mode, or when the normal mode transition command is received in E2 DataFlash P/E modes.

   b. **E2 DataFlash Status Read Mode:** The E2 DataFlash status read mode is for reading information on the state of the E2 DataFlash. The FCU enters this mode when a status read mode transition command is received, or when a command other than the normal mode transition and lock-bit read mode transition command is received in E2 DataFlash P/E modes. E2 DataFlash status read mode encompasses the states where the FRDY bit in FSTATR0 is 0 and the FASTAT.CMDLK bit is set to 1 (command-locked state) after an error has occurred. Reading from an address within the E2 DataFlash area will actually read the value of the FSTATR0 register. High-speed reading of the ROM is possible.

   c. **E2 DataFlash Lock-Bit Read Mode:** Since the E2 DataFlash memory does not have lock bits, the lock bits are not read even if a transition to this mode is made. If the E2 DataFlash memory area is read after a transition to this mode, an E2 DataFlash access violation is not generated, but the values read are undefined. High-speed reading of the ROM is possible. The FCU enters this mode when a lock-bit read mode transition command is received in E2 DataFlash P/E modes.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 147

FCU Commands

FCU commands consist of commands for mode transitions of the FCU and for programming and erasure. Table 7.4 lists the FCU commands for use with the ROM and the E2 DataFlash.

TABLE 7.4 FCU Commands [1], page 1762.

COMMAND	ROM	E2 DATAFLASH
P/E normal mode transition	Shifts to normal mode	Shifts to normal mode
Status-read mode transition	Shifts to status-read mode	Shifts to status-read mode
Lock-bit read mode transition	Shifts to lock-bit read mode	Shifts to lock-bit read mode
Peripheral clock notification	Sets the FCLK	
Programming	ROM programming	E2 DataFlash programming
Block Erase	ROM erasure	E2 DataFlash erasure
P/E suspend	Suspends programming/erasure	
P/E resume	Resumes programming/erasure	
Status register clear	Clears the ILGERR and FCU command-lock bit	
Lock-bit read 2	Reads the lock bit of a specified block	—
Lock-bit programming	Programs the lock bit of a specified block	—
Blank checking	—	Checks whether the E2 flash memory is blank

The lock-bit read 2 commands is for the ROM also used as the blank check command for the E2 DataFlash memory. That is, when a lock-bit read 2 command is issued for the E2 DataFlash, blank checking is executed for the E2 DataFlash memory. Commands for the FCU are issued by writing an FCU command to addresses within the range for ROM programming and erasure or an address in the E2 DataFlash.

7.2.6 FCU Register Descriptions

Some registers are common to the ROM and the E2 DataFlash, while others are dedicated to one or the other.

Flash Write Erase Protection Register (FWEPROR)

Programming and erasure of the ROM or the E2 DataFlash memory, programming and erasure of lock bits, reading of lock bits, and blank checking by software are prohibited when protected. FWEPROR is initialized by a reset due to the signal on the RES# pin, by transitions to software standby and deep software standby, and by the power supply voltage falling below the threshold for detection.

TABLE 7.5 FWEPROR Register, Address(es): 0008 C296h [1], page 1735.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b1, b0	FLW[1:0]	Flash Programming/ Erasure	b1:b0	R/W
			0 0: Disables programming and erasure of the ROM/E2 DataFlash, programming and erasure of lock bits, reading of lock bits, and blank checking	
			0 1: Enables programming and erasure of the ROM/E2 DataFlash, programming and erasure of lock bits, reading of lock bits, and blank checking	
			1 0: Disables programming and erasure of the ROM/E2 DataFlash, programming and erasure of lock bits, reading of lock bits, and blank checking	
			1 1: Disables programming and erasure of the ROM/E2 DataFlash, programming and erasure of lock bits, reading of lock bits, and blank checking	
b7 to b2	—	Reserved	These bits are read as 0. The write value should be 0.	R/W

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 149

Flash Mode Register (FMODR)

TABLE 7.6 FMODR Register Address(es): 007F C402h [1], page 1736.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b3 to b0	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b4	FRDMD	FCU Read Method Select	This bit selects internal processing by the FCU when a 0x71 command is issued.	R/W
b7 to b5	—	Reserved	These bits are read as 0. The write value should be 0.	R/W

Regarding the FMODR, when a 0x71 command is issued in relation to the FCU, this bit selects internal processing by the FCU. Internal processing by the FCU differs according to the address where the 0x71 command was issued. This register is common to the ROM and the E2 DataFlash. When the on-chip ROM is disabled, the data read from this register is 00h and writing is disabled.

Flash Access Status Register (FASTAT)

TABLE 7.7 FASTAT Register, Address(es): 007f C410h [1], page 1737.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	DFLWPE	E2 DataFlash Programming/ Erasure Protection Violation Flag	0: Programming/erasure protection violation	R/W
			1: No programming/erasure protection violation	
b1	DFLRPE	E2 DataFlash Read Protection Violation Flag	0: Read protection violation	R/W
			1: No read protection violation	
b2	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b3	DFLAE	E2 DataFlash Access Violation Flag	0: No E2 DataFlash access violation	R/W
			1: E2 DataFlash access violation	
b4	CMDLK	FCU Command-Lock Flag	0: FCU accepts the command	R
			1: FCU does not accept the command	
b6, b5	—	Reserved	These bits are read as 0. The write value should be 0.	
b7	ROMAE	ROM Access Violation Flag	0: No ROM access violation	R/W
			1: ROM access violation	

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 150

This register is common to the ROM and the E2 DataFlash. When the on-chip ROM is disabled, the data read from this register is 00h and writing is disabled. All the bits are described as below:

DFLWPE Bit (E2 DataFlash Programming/Erasure Protection Violation): This bit is used to indicate whether or not the programming/erasure protection set by DFLWEy ($y = 0, 1$) is violated. When the DFLWPE bit is set to 1, the FSTATR0.ILGLERR bit is set to 1, and the CMDLK bit becomes 1 (command-locked state).

DFLRPE Bit (E2 DataFlash Read Protection Violation Flag): This bit is used to indicate whether or not the reading protection set by DFLREy (y = 0, 1) is violated. When the DFLRPE bit is set to 1, the FSTATR0.ILGLERR bit is set to 1, and the CMDLK bit becomes 1 (command-locked state).

DFLAE Bit (E2 DataFlash Access Violation Flag): This bit indicates whether an E2 DataFlash access violation occurred. When the DFLAE bit is set to 1, the ILGLERR bit in FSTATR0 is set to 1, and the CMDLK bit becomes 1 (command-locked state).

CMDLK Bit (FCU Command-Lock Flag): This bit is used to indicate whether the FCU can receive commands. When any bit of the FASTAT register is set to 1, the CMDLK bit is set to 1, and the FCU receives no commands. To enable the FCU to receive commands, a status register clear command must be issued to the FCU after setting FASTAT to 10h.

ROMAE Bit (ROM Access Violation Flag): This bit indicates whether a ROM access violation occurred. When the ROMAE bit is set to 1, the FSTATR0.ILGLERR bit is set to 1, and the CMDLK bit becomes 1 (command-locked state).

Flash Ready Interrupt Enable Register (FRDYIE)

TABLE 7.8 FRDYIE Register, Address(es): 007F C412h [1], page 1741.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	FRDYIE	Flash Ready Interrupt Enable	0: FRDYI interrupt requests disabled	R/W
			1: FRDYI interrupt requests enabled	
b7 to b1	—	Reserved	These bits are read as 0. The write value should be 0.	R/W

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 151

This register is common to the ROM and the E2 DataFlash. When the on-chip ROM is disabled, the data read from this register is 00h and writing is disabled.

FRDYIE Bit (Flash Ready Interrupt Enable): This bit is to enable or disable a flash ready interrupt request when programming/erasure is completed. If the FRDYIE bit is set to 1, a flash ready interrupt request (FRDYI) is generated when execution of the FCU command has completed (FSTATR0.FRDY bit changes from 0 to 1).

E2 DataFlash Read Enable Register 0 (DFLRE0)

TABLE 7.9 DFLRE0 Register Address(es): 007F C440h [1], page 1742.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	DBRE00	0000–0063 Block Read Enable	0: Read disabled 1: Read enabled	R/W
b1	DBRE01	0064–0127 Block Read Enable		R/W
b2	DBRE02	0128–0191 Block Read Enable		R/W
b3	DBRE03	0192–0255 Block Read Enable		R/W
b4	DBRE04	0256–0319 Block Read Enable		R/W
b5	DBRE05	0320–0383 Block Read Enable		R/W
b6	DBRE06	0384–0447 Block Read Enable		R/W
b7	DBRE07	0448–0511 Block Read Enable		R/W
b15 to b8	KEY[7:0]	Key Code	These bits control permission and prohibition of writing to the DFLRE0 register.	R/W*1
			To modify the DFLRE0 register, write 2Dh to the 8 higher-order bits and the desired value to the 8 lower-order bits as a 16-bit unit.	

Note 1. Write data is not retained.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 152

DFLRE0 is a register to enable or disable the 0000 to 0511 blocks of the data area to be read. Reading is enabled or disabled in 2-Kbyte units (64 blocks). This register is dedicated to the E2 DataFlash. When the on-chip ROM is disabled, the data read from this register is 0000h and writing is disabled.

E2 DataFlash Read Enable Register 1 (DFLRE1)

TABLE 7.10 DFLRE1 Register, Address(es): 007F C442h [1], page 1743.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	DBRE00	0512–575 Block Read Enable	0: Read disabled	R/W
			1: Read enabled	
b1	DBRE01	0576–0639 Block Read Enable		R/W
b2	DBRE02	0640–0703 Block Read Enable		R/W
b3	DBRE03	0704–0767 Block Read Enable		R/W
b4	DBRE04	0768–0831 Block Read Enable		R/W
b5	DBRE05	0832–0895 Block Read Enable		R/W
b6	DBRE06	0896–0959 Block Read Enable		R/W
b7	DBRE07	0960–1023 Block Read Enable		R/W
b15 to b8	KEY[7:0]	Key Code	These bits control permission and prohibition of writing to the DFLRE0 register. To modify the DFLRE0 register, write 2Dh to the 8 higher-order bits and the desired value to the 8 lower-order bits as a 16-bit unit.	R/W

DFLRE1 is a register to enable or disable the 0512 to 1023 blocks of the data area to be read. Reading is enabled or disabled in 2 Kbyte units (64 blocks). This register is dedicated to the E2 DataFlash. When the on-chip ROM is disabled, the data read from this register is 0000h and writing is disabled.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 153

E2 DataFlash P/E Enable Register 0 (DFLWE0)

TABLE 7.11 DFLWE0 Register Address(es): 007F C450h [1], page 1744.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	DBWE00	0000–0063 Block Programming/ Erasure Enable	0: Read disabled 1: Read enabled	R/W
b1	DBWE01	0064–0127 Block Programming/ Erasure Enable		R/W
b2	DBWE02	0128–0191 Block Programming/ Erasure Enable		R/W
b3	DBWE03	0192–0255 Block Programming/ Erasure Enable		R/W
b4	DBWE04	0256–0319 Block Programming/ Erasure Enable		R/W
b5	DBWE05	0320–0383 Block Programming/ Erasure Enable		R/W
b6	DBWE06	0384–0447 Block Programming/ Erasure Enable		R/W
b7	DBWE07	0448–0511 Block Programming/ Erasure Enable		R/W
b15 to b8	KEY[7:0]	Key Code	These bits control permission and prohibition of writing to the DFLWE0 register. To modify the DFLWE0 register, write 1Eh to the 8 higher-order bits and the desired value to the 8 lower-order bits as a 16-bit unit.	R/W

DFLWE0 is a register to enable or disable the 0000 to 0511 blocks of the data area to be programmed or erased. Programming or erasing is enabled or disabled in 2-Kbyte units (64 blocks). This register is dedicated to the E2 DataFlash. When the on-chip ROM is disabled, the data read from this register is 0000h and writing is disabled.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 154

E2 DataFlash P/E Enable Register 1 (DFLWE1)

TABLE 7.12 DFLWE1 Register, Address(es): 007F C452h [1], page 1744.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	DBWE00	0512–575 Block Programming/ Erasure Enable	0: Read disabled 1: Read enabled	R/W
b1	DBWE01	0576–0639 Block Programming/ Erasure Enable		R/W
b2	DBWE02	0640–0703 Block Programming/ Erasure Enable		R/W
b3	DBWE03	0704–0767 Block Programming/ Erasure Enable		R/W
b4	DBWE04	0768–0831 Block Programming/ Erasure Enable		R/W
b5	DBWE05	0832–0895 Block Programming/ Erasure Enable		R/W
b6	DBWE06	0896–0959 Block Programming/ Erasure Enable		R/W
b7	DBWE07	0960–1023 Block Programming/ Erasure Enable		R/W
b15 to b8	KEY[7:0]	Key Code	These bits control permission and prohibition of writing to the DFLWE0 register. To modify the DFLWE0 register, write E1h to the 8 higher-order bits and the desired value to the 8 lower-order bits as a 16-bit unit.	R/W

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 155

DFLWE1 is a register to enable or disable the 0512 to 1023 blocks of the data area to be programmed or erased. Programming or erasing is enabled or disabled in 2-Kbyte units (64 blocks).This register is dedicated to the E2 DataFlash. When the on-chip ROM is disabled, the data read from this register is 0000h and writing is disabled.

FCU RAM Enable Register (FCURAME)

TABLE 7.13 FCURAME Register, Address(es): 007F C454h [1], page 1746.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	FCRME	FCU RAM Enable	0: Access to the FCU RAM disabled	R/W
			1: Access to the FCU RAM enabled	
b7 to b1	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b15 to b8	KEY[7:0]	Key Code	These bits control permission and prohibition of writing to the FCURAME register.	R/W
			To modify the FCURAME register, write C4h to the 8 higher-order bits and the desired value to the 8 lower-order bits as a 16-bit unit.	

This register is common to the ROM and the E2 DataFlash. When the on-chip ROM is disabled, the data read from this register is 0000h and writing is disabled.

FCRME Bit (FCU RAM Enable): This bit is used to enable or disable access to the FCU RAM. When writing to the FCU RAM, set FENTRYR to 0000h and stop the FCU. Furthermore, data in the FCU RAM cannot be read regardless of whether access to the FCU RAM is enabled or disabled. Values read when reading is attempted are undefined.

Flash Status Register 0 (FSTATR0)

FSTATR0 is also reset by setting the FRESETR.FRESET bit to 1. When the on-chip ROM is disabled, the data read from this register is 00h and writing is disabled. This register is common to the ROM and the E2 DataFlash.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 156

TABLE 7.14 FSTATR0 Register, Address(es): 007F FFb0h [1], page 1747.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	PRGSPD	Programming Suspend Status Flag	0: Other than the status described below	R
			1: During programming suspend processing or programming suspended	
b1	ERSSPD	Erasure Suspend Status Flag	0: Other than the status described below	R
			1: When erasure suspend processing or erasure suspended	
b2	—	Reserved	This bit is read as 0 and cannot be modified.	R
b3	SUSRDY	Suspend Ready Flag	0: P/E suspend commands cannot be received	R
			1: P/E suspend commands can be received	
b4	PRGERR	Programming Error Flag	0: Programming terminates normally	R
			1: An error occurs during programming	
b5	ERSERR	Erasure Error	0: Erasure terminates normally	R
			1: An error occurs during erasure	
b6	ILGLERR	Illegal Command Error Flag	0: FCU detects no illegal command or illegal ROM/E2 DataFlash access	R
			1: FCU detects an illegal command or illegal ROM/E2 DataFlash access	
b7	FRDY	Flash Ready Flag	0: During programming/erasure, During suspending programming/erasure, During the lock-bit read 2 command processing, During the peripheral clock notification command processing, During the blank check processing of E2 DataFlash	R
			1: Processing described above is not performed	

PRGSPD Bit (Programming Suspend Status Flag): This bit is used to indicate that the FCU enters the programming suspend processing state or programming suspended state.

ERSSPD Bit (Erasure Suspend Status Flag): This bit is used to indicate that the FCU enters the erasure suspend processing state or erasure suspended state.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 157

SUSRDY Bit (Suspend Ready Flag): This bit is used to indicate whether the FCU can receive a P/E suspend command.

PRGERR Bit (Programming Error Flag): This bit is used to indicate the result of the ROM/E2 DataFlash programming process by the FCU. When the PRGERR bit is set to 1, the FASTAT.CMDLK bit becomes 1 (command-locked state).

ERSERR Bit (Erasure Error Flag): This bit is used to indicate the result of the ROM/E2 DataFlash erasure process by the FCU. When the ERSERR bit is set to 1, the FASTAT.CMDLK bit becomes 1 (command-locked state).

ILGLERR Bit (Illegal Command Error Flag): This bit is used to indicate that the FCU detects any illegal command or ROM/E2 DataFlash access. When the ILGLERR bit is set to 1, the FASTAT.CMDLK bit becomes 1 (command-locked state).

Flash Status Register 1 (FSTATR1)

TABLE 7.15 FSTATR1 Register, Address(es): 007F FFb1h [1], page 1749.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b3 to b0	—	Reserved	This bit is read as 0 and cannot be modified.	R
b4	FLOCKST	Lock Bit Status	0: Protected	R
			1: Not Protected	
b6,b5	—	Reserved	This bit is read as 0 and cannot be modified.	R
B7	FCUERR	FCU error Flag	0: No error occurs in the FCU processing	R
			1: An error occurs in the FCU processing	

FSTATR1 is also reset by setting the FRESETR.FRESET bit to 1. When the on-chip ROM is disabled, the data read from this register is 00h and writing is disabled. This register is common to the ROM and the E2 DataFlash.

FLOCKST Bit (Lock-Bit Status): This bit is to reflect the read data of a lock bit when using the lock-bit read 2 commands. When the FSTATR0.FRDY bit is set to 1 after a lock-bit read 2 command is issued, the value of the lock-bit status is stored in the FLOCKST bit. The value of the FLOCKST bit is retained until the completion of the next lock-bit read 2 command.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 158

FCUERR Bit (FCU Error Flag): This bit is used to indicate that an error occurs in the FCU internal processing. When the FCUERR bit is set to 1, set the FRESETR.FRESET bit to 1 to initialize the FCU. Additionally, recopy the FCU firmware from the FCU firmware area to the FCU RAM area.

Flash P/E Mode Entry Register (FENTRYR)

TABLE 7.16 FENTRYR Register, Address(es): 007F FFB2h [1], page 1750.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	FENTRY0	ROM P/E Mode Entry 0	0: Area 0 is in ROM read mode	R/W
			1: Area 0 is in ROM P/E mode	
b1	FENTRY1	ROM P/E Mode Entry 1	0: Area 1 is in ROM read mode	R/W
			1: Area 1 is in ROM P/E mode	
b2	FENTRY2	ROM P/E Mode Entry 2	0: Area 2 is in ROM read mode	R/W
			1: Area 2 is in ROM P/E mode	
b3	FENTRY3	ROM P/E Mode Entry 3	0: Area 3 is in ROM read mode	R/W
			1: Area 3 is in ROM P/E mode	
b6 to b4	—	Reserved		R/W
b7	FENTRYD	E2 DataFlash P/E mode entry	0: E2 DataFlash is in read mode	R/W
			1: E2 DataFlash is in P/E mode	
b15 to b8	FEKEY[7:0]	Key Code	These bits control permission and prohibition of writing to the FENTRYR register. To modify the FENTRYR register, write AAh to the 8 higher-order bits and the desired value to the 8 lower-order bits as a 16-bit unit.	R/W

To place the ROM/E2 DataFlash in ROM P/E mode so that the FCU can accept commands, either the FENTRYD or FENTRYn bits (n = 0 to 3) must be set to 1. Note that if a value is set other than AA01h, AA02h, AA04h, AA08h, and AA80h in FENTRYR, the FSTATR0.ILGLERR bit is set to 1 and the FSATAT.CMDLK bit is set to 1 (command-locked state). The ROM exists from area 0 up to area 3 at the maximum, and the FENTRY0 to FENTRY3 bits correspond to the respective areas. The FENTRYn bits

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 159

(n = 0 to 3) for areas that are not present cannot be set to 1. FENTRYR is also reset when the FRESETR.FRESET bit is set to 1. When on-chip ROM is disabled, the data read from FENTRYR is 0000h and writing is disabled. This register is common to the ROM and the E2 DataFlash.

FENTRYn Bit (ROM P/E Mode Entry n: n = 0 to 3): This bit is used to place area n (n = 0 to 3) in P/E mode.

Write-enable conditions (when all of the following conditions are met):

- On-chip ROM is enabled.
- The FSTATR0.FRDY bit is set to 1.
- AAh is written to the FEKEY[7:0] bits in word access.
- The write-enable conditions are met, FENTRYR is set to 0000h, and 1 is written to the FENTRYn (n = 0 to 3).
- Data is written in byte access.
- Data is written in word access when the FEKEY[7:0] bits are other than AAh.
- When the write-enable conditions are met, 0 is written to the FENTRYn (n = 0 to 3) bit.
- When the write-enable conditions are met and FENTRYR is other than 0000h, data is written to FENTRYR.

Flash Protection Register (FPROTR)

TABLE 7.17 FPROTR Register, Address(es): 007F FFB4h [1], page 1752.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	FPROTCN	Lock-Bit Protection Cancel	0: Protection with a lock bit enabled	R/W
			1: Protection with a lock bit disabled	
b7 to b1	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b15 to b8	FPKEY[7:0]	Key Code	These bits control permission and prohibition of writing to the FPROTR register.	R/W
			To modify the FPROTR register, write 55h to the 8 higher-order bits and the desired value to the 8 lower-order bits as a 16-bit unit.	

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 160

FPROTR is also reset when the FRESETR.FRESET bit is set to 1. When the on-chip ROM is disabled, the data read from FPROTR is 0000h and writing is disabled. This register is dedicated to the ROM.

FPROTCN Bit (Lock-Bit Protection Cancel): 55h is written to the FPKEY[7:0] bits and 1 is written to the FPROTCN bit in word access when the value of FENTRYR is other than 0000h.

- Data is written in byte access the FPKEY[7:0] bits are 55h.
- Data is written in word access when the FPKEY[7:0] bits are other than 55h.
- 55h is written to the FPKEY[7:0] bits and 0 is written to the FPROTCN bit in word access.
- The value of FENTRYR is 0000h.

Flash Reset Register (FRESETR)

TABLE 7.18 FRESETR Register, Address(es): 007F FFB6h [1], page 1753.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	FRESET	Flash Reset	0: FCU is not reset	R/W
			1: FCU is reset	
b7 to b1	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b15 to b8	FPKEY[7:0]	Key Code	These bits control permission and prohibition of writing to the FRESETR register.	R/W
			To modify the FRESETR register, write CCh to the 8 higher-order bits and the desired value to the 8 lower-order bits as a 16-bit unit.	

When the on-chip ROM is disabled, the data read from FRESETR is 0000h and writing is disabled. This register is common to the ROM and the E2 DataFlash.

FRESET Bit (Flash Reset): When the FRESET bit is set to 1, programming/erasure operations for the ROM/E2 DataFlash are forcibly terminated and the FCU is initialized. High voltage is applied to the ROM/E2 DataFlash during programming/erasure. To ensure the time required for dropping the voltage applied to the memory, keep the FRESET bit set to 1 for tFCUR when initializing the FCU. While the FRESET bit is kept as 1, the user ap-

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 161

plication is prohibited from reading the ROM/E2 DataFlash. Additionally, when the FRESET bit is set to 1, the FCU commands cannot be used because FENTRYR is initialized.

FCU Command Register (FCMDR)

TABLE 7.19 FCMDR Register Address(es): 007F FFBAh [1], page 1754.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b7 to b0	PCMDR[7:0]	Precommand	Store the command immediately before the last command received by the FCU.	R
b15 to b8	CMDR[7:0]	Command	Store the last command received by the FCU.	R

FCMDR is also initialized when the FRESETR.FRESET bit is set to 1. When the on-chip ROM is disabled, the data read from FCMDR is 0000h and writing is disabled. This register is common to the ROM and the E2 DataFlash.

FCU Processing Switching Register (FCPSR)

TABLE 7.20 FCPSR Register, Address(es): 007F FFC8h [1], page 1755.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	ESUSPMD	Erasure Suspend mode	0: Suspension priority mode	R/W
			1: Erasure priority mode	
b15 to b1	—	Reserved	These bits are read as 0. The write value should be 0.	R/W

FCPSR is also reset when the FRESETR.FRESET bit is set to 1. When the on-chip ROM is disabled, the data read from FCPSR is 0000h and writing is disabled. This register is common to the ROM and the E2 DataFlash.

ESUSPMD Bit (Erasure Suspend Mode): This bit is to select the erasure suspend mode for when a P/E suspend command is issued while the FCU executes the erasure processing for the ROM/E2 DataFlash.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 162

E2 DataFlash Blank Check Control Register (DFLBCCNT)

TABLE 7.21 DFLBCCNT Register, Address(es): 007F FFCAh [1], page 1755.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b10 to b0	BCADR[10:0]	Blank Check Address Setting	Set the address of the area to be checked	R/W
b14 to b11	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b15 to b8	BCSIZE	Blank Check Size Setting	0: The size of the area to be blank checked is 2 bytes	R/W
			1: The size of the area to be blank checked is 2 Kbytes	

DFLBCCNT is also reset when the FRESETR.FRESET bit is set to 1. When the on-chip ROM is disabled, the data read from DFLBCCNT is 0000h and writing is disabled. This register is dedicated to the E2 DataFlash.

BCADR[10:0] Bits (Blank Check Address Setting): These bits are used to set the address of the area to be checked when the size of the area to be checked by a blank check command is 2 bytes (the BCSIZE bit is 0). Set the BCADR[0] bit to 0.When the BCSIZE bit is 0, the start address of the area to be checked is obtained by adding the DFLBCCNT setting value to the block start address (in 2-Kbyte units) specified at issuance of a blank check command. When the BCSIZE bit is 1, the setting of the BCADR[10:0] bits will be ignored.

Flash P/E Status Register (FPESTAT)

TABLE 7.22 FPESTAT Register, Address(es): 007F FFCCh [1], page 1756.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b7 to b0	PEERRST[7:0]	P/E Error Status	00h: No error	R
			01h: Programming error against areas protected by a lock bit	
			02h: Programming error due to sources other than the lock-bit protection	
			11h: Erasure error against areas protected by a lock bit	
			12h: Erasure error due to sources other than the lock-bit protection	
			(Values other than above are reserved)	
b15 to b8	—	Reserved	These bits are read as 0 and cannot be modified.	R

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 163

FPESTAT is also reset when the FRESETR.FRESET bit is set to 1. When on-chip ROM is disabled, the data read from FPESTAT is 0000h and writing is disabled. This register is dedicated to the ROM.

PEERRST[7:0] Bits (P/E Error Status): These bits are used to indicate the reason of an error that occurs during the programming/erasure processing for the ROM. The value of the PEERRST[7:0] bits is valid only when the FSTATR0.FRDY bit is set to 1 while the FSTATR0.ERSERR bit or FSTATR0.PRGERR bit is 1. The value of the reason of the past error is retained in the PEERRST[7:0] bits when the ERSERR bit and the PRGERR bit is 0.

E2 DataFlash Blank Check Status Register (DFLBCSTAT)

TABLE 7.23 DFLBCSTAT Register, Address(es): 007F FFCEh [1], page 1756.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	BCST	Blank Check Status	0: The area to be blank-checked is erased (blank)	R
			1: 0 or 1 is written in the area to be blank-checked	
b15 to b1	—	Reserved	These bits are read as 0.The write value should be 0.	R/W

DFLBCSTAT is also reset when the FRESETR.FRESET bit is set to 1. When on-chip ROM is disabled, the data read from DFLBCSTAT is 0000h and writing is disabled. This register is dedicated to the E2 DataFlash.

Peripheral Clock Notification Register (PCKAR)

TABLE 7.24 PCKAR Register Address(es): 007F FFE8h [1], page 1757.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b7 to b0	PCKA[7:0]	Peripheral Clock Notification	These bits are used to set the flash interface clock (FCLK) at the programming/erasure for the ROM/E2 DataFlash.	R/W
b15 to b8	—	Reserved	These bits are read as 0. The write value should be zero.	R/W

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 164

PCKA[7:0] Bits (Peripheral Clock Notification): These bits are used to set the FCLK at the programming/erasure for the ROM/E2 DataFlash.

- Set the FCLK frequency in the PCKA[7:0] bits and issue a peripheral clock notification command before programming/erasure. Do not change the frequency during programming/erasure for the ROM/E2 DataFlash.
- Write a setting to the PCKA[7:0] bits as a binary value that selects the operating frequency in MHz units.
- For example, when the operating frequency of the FCLK is 35.9 MHz, the setting value is calculated as follows:
  - Round 35.9 off to a whole number.
  - Convert 36 to binary and set the upper bits and lower bits of the PCKA[7:0] bits to 00h and 24h (0010 0100b).

7.2.7 Mode Transitions

- Switching to ROM Read Mode or ROM/E2 DataFlash Read Mode: High-speed reading of the ROM requires clearing of the FENTRYR.FENTRYn (n = 0 to 3) bits to 0000b, which places the FCU in ROM read mode. Writing of 02h as a byte to FWEPROR is also required to disable programming and erasure. Before switching the FCU from ROM P/E mode to read mode, ensure that all processing of FCU commands has been completed and that the FCU has not detected an error. For a transition to ROM/E2 DataFlash read mode, the FENTRYR.FENRTYn (n = 0 to 3) bits and the FENTRYD bit must be set to 0.
- Switching to P/E Mode: A transition to ROM P/E mode is required before executing an FCU command for programming or erasure of the ROM. Setting any of the FENTRYR.FENTRYn (n = 0 to 3) bits to 1 causes a transition to ROM P/E mode for programming and erasure of the corresponding address range.
- A transition to E2 DataFlash P/E mode is required before executing an FCU command for programming or erasure of the E2 DataFlash. For a transition to E2 DataFlash P/E mode, set the FENTRYR.FENTRYD bit to 1. Before actually proceeding to program or erase the ROM, enable programming and erasure by writing 01h as a byte to the FWEPROR (Flash Write Erase Protection Register).
- Switching to P/E Normal Mode: Two methods are available for the transition to P/E normal mode: setting FENTRYR while the FCU is in ROM/E2 DataFlash read mode or issuing the normal mode transition command while the FCU is in P/E mode. The normal mode transition command is issued by writing FFh to a ROM programming/erasure address or to an E2 DataFlash address.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 165

- Switching to Status Read Mode: Issuing a status read mode transition command or an FCU command other than a normal mode transition or lock-bit read mode transition command places the FCU in status read mode. The status read mode transition command is issued to place the FCU in ROM status read mode, and the value of FSTATR0 is obtained by reading from a ROM programming/erasure address or an E2 DataFlash address and is then checked.
- Switching to ROM Lock-Bit Read Mode: Clearing the FMODR.FRDMD bit (memory area reading method) issues a lock-bit read mode transition (lock-bit read 1) command. After the transition to lock-bit read mode, the lock-bit value is obtained by reading from a ROM programming/erasure address. All bits of a value thus read have the value of the lock bit of the block that contains the accessed address. Since there are no lock bits for the E2 DataFlash, undefined data are read from the E2 DataFlash area after a transition of the lock-bit read mode is made.

7.2.8 Programming and Erasure Procedures

The flow of procedures for programming or erasing the ROM or E2 DataFlash is as given as follows:

1. Firmware Transfer to the FCU RAM: FCU commands can only be used if the FCU RAM holds the firmware for the FCU. The FCU RAM does not hold the FCU firmware immediately after the chip has been booted up, so the firmware must be copied from the FCU firmware area to the FCU RAM. Furthermore, when the FSTATR1.FCUERR bit is set to 1, the FCU must be reset and the firmware recopied because the firmware stored in the FCU RAM may have been corrupted.
2. Jump to the on-chip RAM: Since fetching instructions from the ROM is not possible while the ROM is being programmed or erased, instructions have to be fetched from an area other than the ROM. Copy the required program code to the on-chip RAM and then make a jump to the address where the code starts in the on-chip RAM.
3. P/E mode transition: The FCU is placed in P/E mode by setting the FENTRYR.FENTRYn ($n = 0$ to 3) bits and FWEPROR register.
4. Error check.
5. Issue a peripheral clock notification command: FCLK is used in programming and erasing the ROM or E2 DataFlash, so the frequency of this clock has to be set in the PCKAR. Frequencies in the range from 1 to 100 MHz are selectable. If a frequency within this range has not been set, the FCU will detect the error leading the FASTAT.CMDLK bit being set to 1 (command-locked state).
6. Check the execution result of the FCU command.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 166

7.2.9 Programming

The programming command is used to write data to the ROM or the E2 DataFlash.

ROM Programming: In the first and second cycles for the programming command, respectively, the values E8h and 40h are written to the address range for programming and erasure of the ROM. In the third cycle, write the actual data to be programmed, as a word unit, to the start address of the target area for programming. For this start address, always use an address that is aligned on a 128-byte boundary. In the fourth to the 66th cycles, write the data for programming in 63 word-unit rounds to the address range for programming and erasure of the ROM. Once the value D0h has been written to the address range for programming and erasure of the ROM in the 67th cycle, the FCU begins the actual process of programming the ROM. The FSTATR0.FRDY bit can be used to check whether or not the programming has been completed. Addresses that can be used in the first to 67th cycles differ according to the setting of the FENTRYR.FENTRYn bits (n = 0 to 3). Ensure that the addresses suit the setting of FENTRYR.FENTRYn bits. If issuing of the command is attempted for an address in the area for which the P/E mode is disabled by the FENTRYR register, the FCU will detect the error leading the FASTAT.CMDLK bit being set to 1 (command-locked state). In cases where the target range (in the third to 66th cycles) includes addresses that do not require programming, use FFFFh as the data for programming to those addresses. Furthermore, when a lock is programmed so that protection by the lock bit becomes effective, the FPROTR.FPROTCN bit must be set to 1.

E2 DataFlash Programming: Write E8h to an address within the E2 DataFlash area in the first cycle of the programming command, and 01h in the second cycle. In the third cycle, write the first word of data for programming to the address where the target area for programming starts. This address must be on a 2-byte boundary. After writing words to addresses in the E2 DataFlash area one time, write byte D0h to an address within the E2DataFlash area in the fourth cycle; the FCU will then start actual programming of the E2 DataFlash. Read the FRDY bit in FSTATR0 to confirm the completion of the E2 DataFlash programming. When programming, for locking to prohibit programming and erasure according to the setting of the DFLWEy register (y = 0, 1), the relevant bit of the DFLWEy register (y = 0, 1) must be set to 1.

7.2.10 Erasure

To erase the ROM/E2 DataFlash, use the block erasure command. In the first cycle of a block-erasure command, 20h is written to an address for programming or erasure of the ROM or an address in the E2 DataFlash. In the second cycle, when D0h is written to any

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 167

address within the target block for erasure, the FCU starts processing to erase the ROM or E2 DataFlash. The FSTATR0.FRDY bit can be checked to confirm the completion of erasure. When the CPU reads ROM that has been erased, the value read is FFFF FFFFh. In the case of the E2 DataFlash, values read are undefined. In the case of the ROM, the FPROTR.FPROTCN bit must be set to 1, if protection by the lock bit is in effect, for a block to be erased. Note that the E2 DataFlash has a programming and erasure protection function that is controlled by DFLWEy (y = 0, 1).When erasure for locking to prohibit programming and erasure according to the setting of the DFLWEy register (y = 0, 1) proceeds, the relevant bit of the DFLWEy register (y = 0, 1) must be set to 1.

7.2.11 Simple Flash API for RX63N

The Simple Flash API is provided by Renesas to make the process of programming and erasing on-chip flash areas easier. Both the ROM and E2 DataFlash areas are supported. The API in its simplest form can be used to perform blocking erase and program operations. The term 'blocking' means that when a program or erase function is called, the function does not return until the operation has finished. When a flash operation is on-going, that flash area cannot be accessed by the user. If an attempt to access the flash area is made, the flash control unit will transition into an error state. For this reason 'blocking' operations are preferred by some users to prevent the possibility of a flash error. But there are other cases where blocking operations are not desired.

If the user is writing data to the E2 DataFlash, for example, the ROM can still be read. In this case many users would like for the E2 DataFlash write or erase to occur in the background (non-blocking) while their application continues to run in ROM. RX600 MCUs support this feature and it is available in the Simple Flash API. The user can also perform non-blocking ROM operations as well, but application code will need to be located outside of the ROM. For more information, please refer the hardware manual provided by Renesas for RX63N.

Features

Below is a list of the features supported by the Simple Flash API.

- Blocking, erasing, and programming of User ROM
- Non-blocking, background operation, erasing, and programming of user ROM
- Blocking, erasing, programming, and blank checking of E2 DataFlash
- Non-blocking, background operation, erasing, programming, and blank checking of E2 DataFlash
- Callback functions for when flash operation has finished (only with non-blocking)
- ROM to ROM transfers

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 168

- E2 DataFlash to E2 DataFlash transfers
- Lock-bit protection
- Lock-bit set/read

7.3 BASIC EXAMPLES

EXAMPLE 1

This function initializes the FCU peripheral block.

```
 1. static uint8_t flash_init(void){
 2.    uint32_t * src, * dst;
 3.    uint16_t i;
 4.    FLASH.FAEINT.BIT.ROMAEIE = 0;
 5.    FLASH.FAEINT.BIT.CMDLKIE = 0;
 6.    FLASH.FAEINT.BIT.DFLAEIE = 0;
 7.    FLASH.FAEINT.BIT.DFLRPEIE = 0;
 8.    FLASH.FAEINT.BIT.DFLWPEIE = 0;
 9.    IPR(FCU, FIFERR) = 0;
10.    IEN(FCU, FIFERR) = 0;
11.    #if defined(DATA_FLASH_BGO) || defined(ROM_BGO)
12.       IPR(FCU, FRDYI) = FLASH_READY_IPL;
13.       IEN(FCU, FRDYI) = 1;
14.    #else
15.       IPR(FCU, FRDYI) = 0;
16.       IEN(FCU, FRDYI) = 0;
17.    #endif
18.    if(FLASH.FENTRYR.WORD != 0x0000){
19.       FLASH.FENTRYR.WORD = 0xAA00;
20.       while(0x0000 != FLASH.FENTRYR.WORD){
21.       }
22.    }
23.    FLASH.FCURAME.WORD = 0xC401;
24.    src = (uint32_t *)FCU_PRG_TOP;
25.    dst = (uint32_t *)FCU_RAM_TOP;
26.    for( i=0; i<(FCU_RAM_SIZE/4); i++){
27.       *dst = *src;
28.       src++;
29.       dst++;
30.    }
```

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 169


```
31.     g_fcu_transfer_complete = 1;
32.     return FLASH_SUCCESS;
33. }
```

Line 4 to 8 disables the FCU interrupts in the FCU block. Line 9 and 10 disables the flash interface error interrupt (FIFERR). If the background operations for the flash are enabled, then enable the flash-ready interrupt (FRDYI) (line 12 and 13) or else disable the flash-ready interrupt (line 15 and 16). Then transfer the firmware to the FCU RAM. To use the FCU commands, the FCU firmware must be stored in the FCU RAM. Before writing to the FCU RAM, the FENTRY must be cleared to stop the FCU (line 18). Disable the FCU from accepting commands. Clear both the FENTRY0 (ROM) and FENTRYD (E2 DataFlash) bits to 0 (line 19). Read the FENTRYR to ensure it has been set to 0. Note that the top byte of the FENTRYR register is not retained and is read as 0x00 (line 20). Then Enable the FCU RAM (line 23). Then copy the FCU firmware to the FCU RAM. Source: FEFFE000h to FF000000h (FCU firmware area). Destination: 007F8000h to 007FA000h (FCU RAM area). Iterate for loop to copy the FCU firmware (line 26) and copy data from the source to the destination pointer (line 27). If the FCU firmware transfer is complete, set the flag to 1 (line 31).

EXAMPLE 2

The following commands are used to allow read and program permissions to the E2 DataFlash area. This function does not have to execute from in the RAM. It must be in the RAM, though, if ROM_BGO is enabled and this function is called during a ROM P/E operation. The arguments passed are as follows:

1. read_en_mask -
   - Bitmasked value. Bits 0–3 represent each Data Blocks 0–3 (respectively).
   - ‘0’=no Read access.
   - ‘1’=Allows Read by CPU
2. write_en_mask -
   - Bitmasked value. Bits 0–3 represent each Data Blocks 0–3 (respectively).
   - ‘0’=no Erase/Write access.
   - ‘1’=Allows Erase/Write by FCU

```
1. void R_FlashDataAreaAccess(uint16_t read_en_mask, uint16_t
   write_en_mask){
2.     //This line Sets Read access for the E2 DataFlash blocks DB0-DB7
3.     FLASH.DFLRE0.WORD = 0x2D00 | (read_en_mask & 0x00FF);
4.     //This line Sets Read access for the E2 DataFlash blocks DB8-DB15
```

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 170


```
 5.    FLASH.DFLRE1.WORD = 0xD200 | ((read_en_mask >> 8) & 0x00FF);
 6.    //This line Sets Erase/Program access for the E2 DataFlash
         blocks DB0-DB7
 7.    FLASH.DFLWE0.WORD = 0x1E00 |  (write_en_mask & 0x00FF);
 8.    //This line Sets Erase/Program access for the E2 DataFlash
         blocks DB8-DB15
 9.    FLASH.DFLWE1.WORD = 0xE100 |  ((write_en_mask >> 8) & 0x00FF);
10. }
```

EXAMPLE 3

This function copies Flash API code from the ROM to the RAM. This function needs to be called before any program/erase functions. This function need not be used when we are dealing with only flash programming.

```
 1. void R_FlashCodeCopy(void){
 2.    uint8_t * p_ram_section;
 3.    uint8_t * p_rom_section;
 4.    uint32_t  bytes_copied;
 5.    p_ram_section = (uint8_t *) sectop("RPFRAM");
 6.    p_rom_section = (uint8_t *) sectop("PFRAM");
 7.    for (bytes_copied=0; bytes_copied < secsize("PFRAM");
       bytes_copied++){
 8.       p_ram_section[bytes_copied]= p_rom_section[bytes_copied];
 9.    }
10. }
```

Lines 2 and 3 name pointers to the beginning of the RAM and the ROM sections respectively. Line 5 and 6 initializes the RAM and ROM section pointers respectively. Lines 7 to 9 copy code from the ROM to the RAM, 1 byte at a time.

7.4 ADVANCED CONCEPTS

7.4.1 Virtual EEPROM for RX63N

Many users wish to use the E2 DataFlash on their MCUs as they would an EEPROM. The problem with this is that the E2 DataFlash on the RX63N does not have 1-byte write or 1-byte erase capabilities. Even if their RX MCU did offer this granularity there would also

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 171

be the issue of wear leveling. To help solve these issues the Virtual EEPROM project (VEE for short) was created.

The VEE project offers these features:

- Wear leveling is used to increase E2 DataFlash longevity
- Easy to use API interface to safely read and write
- Uses the background operation feature of MCUs so E2 DataFlash operations do not block the user application
- Automatically recovers from resets or power downs that occur during programs and erases
- Adapts to flashes with different program sizes, erase sizes, block sizes, and number of E2 DataFlash blocks
- Highly configurable to adjust to unique needs of each user

The VEE project is a software layer that sits on top of the Renesas provided Flash API, shown in Figure 7.3. The Virtual EEPROM requires that the user's MCU have hardware support for background operations (BGO) on the E2 DataFlash and that the Flash API has software support. Background operation means that flash operations do not block. In a blocking system (one that does not have support for BGO) when a flash operation is started, control is not given back to the user application until the operation has completed. With a system that supports BGO, control is given back to the user application immediately after the operation has been successfully started. The user will be alerted by a callback function, or can poll to see if the operation has completed. Since the VEE project uses the BGO capabilities of the MCU this means less time is taken away from the user application.

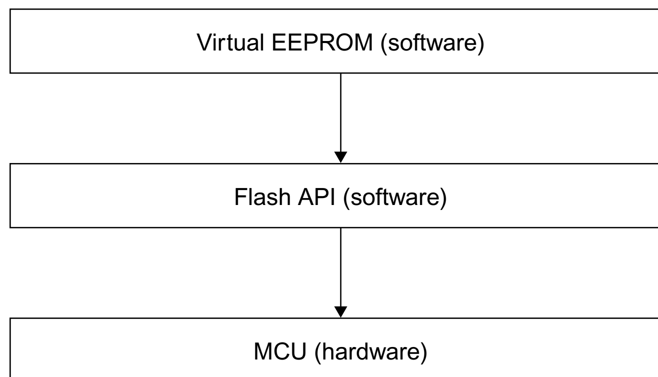


Figure 7.3 Project Layers [2], page 2.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 172

Records

When data is written to the VEE it is done using VEE Records. Each record has some information that must be filled in by the user as well as a pointer to the data to be stored. Users can store as much data as they wish with each record. When the user writes a record to the VEE it stores the data and the record information together. Each record is identified by a unique ID. If the user writes a record with the same ID as a record that was previously written then it will be written as a new record and the older record will no longer be valid. The reason this is done is for wear leveling purposes. The user does have the option of configuring the VEE code to ignore duplicate writes.

Data Management

With the VEE project the MCU's E2 DataFlash area is split up into VEE Sectors. These do not correspond to real sectors on the MCU. The VEE project requires at least one VEE Sector. Each VEE Sector is made up of at least two VEE Blocks. Each VEE Block can be made up of one or more flash blocks on the MCU. One VEE Block has the latest stored data for that VEE Sector at any given time. The VEE Blocks ping-pong data back and forth as one becomes full. When a write occurs and there is no room left in the current VEE Block, a defrag occurs in which the latest data is transferred to the next VEE Block in the VEE Sector. Once the data has been transferred the old VEE Block can be erased so that it will be ready when the new VEE Block becomes full. This moving of data is used for wear leveling purposes.

Having different VEE Sectors allows the user to separate data. One reason for doing this would be to separate frequently written data from infrequently written data. An example is if a user had one large block of data that was written once a day and a small block of data that was written every minute. The small block will fill up the current block quickly and will force a defrag frequently which means the large block of data that has not been changed will need to be transferred even though the data is the same as before. The large block can also cause defragmentation to happen more often since it can take up a significant portion of the VEE Block's available space.

VEE in Action

VEE Records are stored in VEE Blocks. At least 2 VEE Blocks are needed to make a VEE Sector. At any point in time within the VEE project there is a maximum of one valid record per unique ID. As records with the same ID are written, the newest record becomes the valid record and the previously written records are ignored.

When a VEE write is issued to a VEE Block that does not have enough room for the record, a defrag is needed. A defrag will move all valid records to another VEE Block. This is shown in Figure 7.4 where the write of VEE Record 1 will not fit into VEE Block 0. This forces a defrag where all valid records will be moved to VEE Block 1.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 173

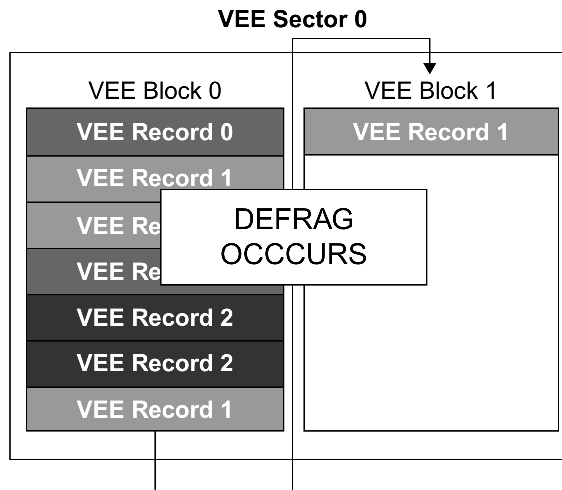


Figure 7.4 Filling up Block 0 [2], page 3.

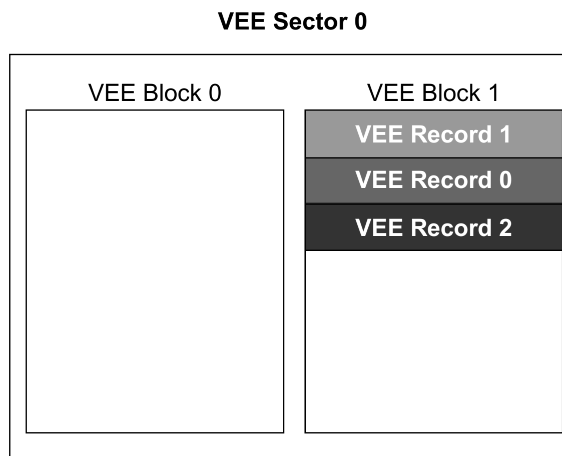


Figure 7.5 Defrag moves data to Block 1 [2], page 3.

Assigning VEE Records to VEE Sectors

VEE Records are assigned to VEE Sectors at compile time via the g_vee_RecordLocations[] array. There is one entry in this array per unique ID in the VEE project. The value for each entry in the array is which VEE Sector the record will be located in. Following is an example configuration where there are 2 VEE Sectors and we want the first four records to be located in VEE Sector 0 and the last four records to be located in VEE Sector 1.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 174


```
 1. const uint8_t g_vee_RecordLocations[VEE_MAX_RECORD_ID] = {
 2.     0,/* Record 0 will be in sector 0 */
 3.     0,/* Record 1 will be in sector 0 */
 4.     0,/* Record 2 will be in sector 0 */
 5.     0,/* Record 3 will be in sector 0 */
 6.     1,/* Record 4 will be in sector 1 */
 7.     1,/* Record 5 will be in sector 1 */
 8.     1,/* Record 6 will be in sector 1 */
 9.     1,/* Record 7 will be in sector 1 */
10. };
```

Allocating VEE Blocks

After defining how many VEE Sectors will be used, the user must decide where the VEE Blocks inside of the sectors will be allocated. This is done using two separate arrays. The names presented below are the defaults used. The first array is named g_vee_sect#_block_addresses[], where the '#' is replaced with the sector number. Each entry of the array defines the starting address of a VEE Block in this particular sector. There will be one of these arrays defined for each VEE Sector.

The second array is named g_vee_sect#_df_blocks[][2], where the '#' is once again replaced with the sector number. This is a 2D array so each entry is another array. The internal array is an array that holds the first and last E2 DataFlash blocks on the MCU that make up this VEE Block. There will be one of these arrays defined for each VEE Sector. An example is shown of a system with the following setup:

- Two VEE Sectors
- Two VEE Blocks per VEE Sector
- Four MCU E2 DataFlash blocks per VEE Block
- VEE Sector 0 will be allocated lower in memory than VEE Sector 1

```
1. const uint32_t g_vee_sect0_block_addresses[] = {   //Sector 0
2.     0x100000,   //Start address of VEE Block 0
3.     0x102000   //Start address of VEE Block 1
4. };
5. const uint16_t g_vee_sect0_df_blocks[][2] = {
6.     {BLOCK_DB0, BLOCK_DB3},   //Start & end DF blocks making up
                                   VEE Block 0
7.     {BLOCK_DB4, BLOCK_DB7}   //Start & end DF blocks making up VEE
                                  Block 1
8. };
9. const uint32_t g_vee_sect1_block_addresses[] ={   //Sector 1
```

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 175


```
10.     0x104000,   //Start address of VEE Block 0
11.     0x106000   //Start address of VEE Block 1
12. };
13.     const uint16_t g_vee_sect1_df_blocks[][2] = {
14.     {BLOCK_DB8, BLOCK_DB11},   //Start & end DF blocks making up
                                    VEE Block 0
15.     {BLOCK_DB12, BLOCK_DB15}   //Start & end DF blocks making up
                                    VEE Block 1
16. };
```

Data Structure that Holds VEE Project Data Configuration

The g_vee_Sectors array is the data structure that is used in the VEE project code for obtaining information about the current systems VEE data configuration. Each entry defines a VEE Sector and holds the following information:

- The ID of the sector
- How many VEE Blocks make up this sector
- The size (in bytes) of this sector
- The starting MCU addresses for each VEE Block in this sector
- The number of MCU E2 DataFlash blocks per VEE Block
- The start and end MCU E2 DataFlash blocks for each VEE Block

An example is shown of a VEE project with two different sized VEE Sectors.

```
 1. const vee_sector_t g_vee_Sectors[ VEE_NUM_SECTORS ] =
 2.     {   //Sector 0
 3.        0,   //ID is 0
 4.        2,   //There are 2 VEE Blocks in this sector
 5.        8192,   //Size of each VEE Block
 6.        //Starting addresses for each VEE Block
 7.        (const uint32_t *)g_vee_sect0_block_addresses,
 8.        //Number of E2 DataFlash blocks per VEE Block
 9.        //(End Block # - Start Block # + 1)
10.        4,
11.        //Start & end DF blocks making up VEE Blocks
12.        g_vee_sect0_df_blocks
13.     },
14.     {   //Sector 1
15.        1,   //ID is 1
16.        2,   //There are 2 VEE Blocks in this sector
```

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 176


```
17.         6144,   //Size of each VEE Block
18.         //Starting addresses for each VEE Block
19.         (const uint32_t *)g_vee_sect1_block_addresses,
20.         //Number of E2 DataFlash blocks per VEE Block
21.         //(End Block # - Start Block # + 1)
22.         3,
23.         //Start & end DF blocks making up VEE Blocks
24.         g_vee_sect1_df_blocks
25.    };
```

VEE Record

When reading and writing to the VEE using the provided API functions, the user sends in data using a VEE Record data structure. This structure is shown next.

```
 1. //VEE Record Structure
 2. typedef struct{
 3.    //Unique record identifier, cannot be 0xFF!
 4.    vee_var_data_t ID;
 5.    //Number of bytes of data for this record
 6.    vee_var_data_t size;
 7.    //Valid or error checking field
 8.    vee_var_data_t check;
 9.    //Which VEE Block this record is located in, user does not set
         this
10.    vee_var_data_t block;
11.    //Pointer to record data
12.    uint8_t far * pData;
13. } vee_record_t;
```

7.4.2 Protection

Protection against programming/erasure for the ROM/E2 DataFlash includes software protection and the command-locked state.

Software Protection

With the software protection, the ROM/E2 DataFlash programming/erasure is prohibited by the settings of the control registers or user area lock bit. When the software protection is

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 177

violated and a ROM/E2 DataFlash programming/erasure-related command is issued, the FCU detects an error and the FASTAT.CMDLK bit is set to 1 (command-locked state).

1. Protection through FWEPROR: If the FWEPROR.FLWE[1:0] bits are not set to 01b, programming cannot be performed in any of the modes.
2. Protection through FENTRYR: When the FENTRYR.FENTRY3, FENTRY2, FENTRY1, FENTRY0 bits are 0, the ROM/E2 DataFlash read mode is selected. Because the FCU command cannot be received in the ROM/E2 DataFlash read mode, ROM/E2 DataFlash programming/erasure is prohibited. When an FCU command is issued in the ROM/E2 DataFlash read mode, the FCU detects an illegal command error and the FASTAT.CMDLK bit is set to 1 (command-locked state).
3. Protection through the Lock Bit: Each block in the user area includes a lock bit. When the FPROTR.FPROTCN bit is 0, blocks whose lock bit is set to 0 are prohibited from being programmed/erased. To program or erase blocks whose lock bit is set to 0, set the FPROTCN bit to 1. When the lock-bit protection is violated and a ROM programming/erasure-related command is issued, the FCU detects a programming/erasure error and the FASTAT.CMDLK bit is set to 1 (command-locked state).
4. Protection through DFLWEy: When the DBWEj (j = 00 to 15) bit in DFLWEy (y = 0, 1) is 0, programming and erasure of block DBj in the data area is disabled. If an attempt is made to program or erase block DBj while the DBWEj bit is 0, the FCU detects a programming/erasure protection error and the FASTAT.CMDLK bit is set to 1 (command-locked state).
5. Protection through DFLREy: When the DBREj bit (j = 00 to 15) in DFLREy (y = 0, 1) is 0, reading of block DBj in the data area is disabled. If an attempt is made to read block DBj while the DBREj bits are 0, the FCU detects a read protection error and the FASTAT.CMDLK bit is set to 1 (command-locked state).

Command-Locked State

With the command-locked state, the FCU detects malfunctions caused by command issuance errors and prohibited access occurrences, and a command is prohibited from being received. When any bit from among the status bits (the ILGLERR, ERSERR, and PRGERR bits in FSTATR0, the FSTATR1.FCUERR bit, and the ROMAE, DFLAE, DFLRPE, and DFLWPE bits in FASTAT), the FCU will be in the command-locked state (FASTAT.CMDLK bit is set to 1), so programming and erasure of the ROM/E2 DataFlash are prohibited. To clear the command-locked state, a status register clear command must be issued with FASTAT set to 10h. While the interrupt enable bit in FAEINT is 1, if the corresponding bit in FASTAT is set to 1, a flash interface error (FIFERR) interrupt occurs.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 178

7.4.3 User Boot Mode

There are many methods of booting from the flash. We discuss only the User boot mode since it is the one which is most commonly used.

If the low level is on the MD pin and the high level is on the PC7 pin at the time of release from the reset state, the chip starts in user boot mode. The reset vector at this time points to the address FF7F FFFCh of the user boot area. In user boot mode, it is possible to perform programming using a given interface; user area or data area can be programmed or erased by issuing the FCU command. Note that programming to the user boot area should be performed in boot mode.

Boot Mode System Configuration

In boot mode, the host sends control commands and data for programming, and the user area, data area, and user boot area are programmed or erased accordingly. An on-chip SCI handles transfer between the host and RX63N/RX631 in asynchronous mode. Tools for transmission of control commands and the data for programming must be prepared in the host. When the RX63N/RX631 is activated in boot mode, the program on the boot area is executed. This program automatically adjusts the bit rate of the SCI and controls programming/erasure by receiving control commands from the host.

Figure 7.6 shows the system configuration for operations in boot mode. Table 7.25 shows the input and output pins associated with the ROM/E2 DataFlash.

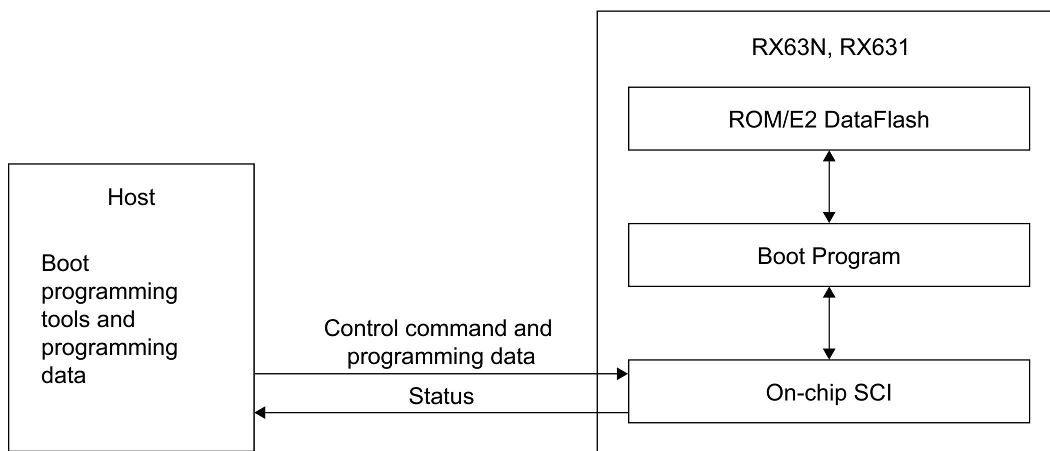


Figure 7.6 Systems Configuration for Operations in Boot Mode [1], page 1787.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 179

TABLE 7.25 Input and Output Pins Associated with the ROM/E2 DataFlash [1], page 1792.

PIN NAME	I/O	MODE TO BE USED	USE
MD	Input	Boot mode User boot mode USB boot mode	Selection of operation mode
PC7	Input		Selection of boot mode (SCI boot), user boot mode, or USB boot mode
PF2/RXD1 (177/176-pin package) P30/RXD1 (145/144/100/64/48-pin packages)	Input	Boot mode	For host communication (to receive data through SCI)
PF0/TXD1 (177/176-pin package) P26/TXD1 (145/144/100/64/48-pin packages)	Output		For host communication (to transmit data through SCI)
USB0_DP.USB0_DM	I/O	USB boot mode	Data Input/output of USB
P14/USB0_DPUPE	I/O		Control of pull-up for USB
P16/USB0_VBUS	Input		Detection of connection and disconnection of USB cables
P35	Input		Selection of USB bus-power mode or self power mode

State Transitions in Boot Mode

1. Matching the Bit Rates: When the RX63N/RX631 is activated in boot mode, the bit rate of the SCI is automatically adjusted to match that of the host. On completion of this automatic bit rate adjustment, the RX63N/RX631 transmits the value 00h to the host. On subsequent correct reception of the value 55h sent from the host, the RX63N/RX631 enters the wait for a command for inquiry or selection.
2. Waiting for a Command for Inquiry or Selection: This state is for inquiries on the area size, the area configuration, the addresses where the areas start, the state of support, and selection of the device, clock mode, and bit rate. The RX63N/RX631 receives a programming/erasure state transition command issued by the host and then enters the state to determine whether ID code protection is enabled or disabled.
3. Judging ID Code Protection: This state is for determining whether ID code protection is enabled or disabled. The control code and ID code written in the ROM are used to determine whether ID code protection is enabled or disabled. When enabled, the state of waiting for the ID code is entered. When disabled, the user area

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 180

and data area are completely erased, and the wait for programming and erasure commands is entered.

4. Waiting for an ID Code: This state is for waiting for the control code and ID code to be sent from the host. The control code and ID code sent by the host are compared with the code stored in the ROM, and the state of waiting for programming and erasure commands is entered if the two match. If they do not match, the next transition is back to the state of waiting for an ID code. However, if the ID codes fail to match three times in a row and also the state of protection is authentication method 1, the ROM is completely erased, and the state of waiting for an ID code is entered again. Turn the power off and start all over.
5. Waiting for a Command for Programming or Erasure: In this state, programming and erasure proceed in accordance with commands from the host. In response to the reception of a command, the RX63N/RX631 enters the wait for the data to use in programming, the wait for specification of the erasure block to be erased, or the state of executing the processing of commands, such as read and check. When the RX63N/RX631 receives a programming selection command, it enters the state of waiting for the data to use in programming. After the host has issued the programming selection command, the process continues with the address where programming is to start and then the data for programming. Setting of FFFF FFFFh as the address where programming is to start indicates the completion of programming, and the next transition is from the wait for the data to use in programming to the wait for programming and erasure commands.

When the RX63N/RX631 receives an erasure selection command, it enters the state of waiting for specification of the erasure block to be erased. After the host has issued the erasure selection command, the process continues with the number of the erasure block to be erased. Setting of FFh as the number of the erasure block indicates the completion of erasure, and the next transition is from the wait for specification of the erasure block to the wait for programming and erasure commands. Since the user area, user boot area, and data area are all completely erased during the interval between booting up in boot mode and the transition to the wait for programming and erasure commands, execution of erasure is not necessary unless data newly programmed in boot mode is to be erased without a further reset.

Other than the programming and erasure commands, commands for execution in this state include those for checksum of the user area and user boot area, blank checking, reading from memory, and acquiring status information.

Automatic Adjustment of the Bit Rate

When the RX63N/RX631 is booted up in boot mode, asynchronous transfer by the SCI is used to measure the periods at low level of consecutive bytes with value 00h that are

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 181

sent from the host. While the period at low level is being measured, set the host's SCI transfer format to 8-bit data, one stop bit, no parity, and a transfer rate of 9,600 bps or 19,200 bps. The RX63N/RX631 calculates the host's SCI bit rate from the measured periods at low level, adjusts its own bit rate accordingly, and then sends the value 00h to the host.

If reception of the value 00h by the host is successful, the host responds by sending the value 55h to the RX63N/RX631. If successful reception of 00h by the host is not possible, reboot the RX63N/RX631 in boot mode, and then repeat the process of automatically adjusting the bit rate. If reception of the value 55h by the RX63N/RX631 is successful, it responds by sending E6h to the host, and if successful reception of 55h by the RX63N/RX631 is not possible, it responds by sending FFh to the host.

ID Code Protection (Boot Mode)

This function is used to prohibit reading/programming/erasure from the host, such as the PC.

After automatic adjustment of the bit rate when booting up in boot mode, the ID code transmitted from the host and the control and ID codes written to the ROM are used to determine disabling or enabling of ID code protection. When ID code protection is enabled, the code sent from the host is compared with the control code and ID code in the ROM to determine whether they match, and reading/programming/erasure will be enabled only when the two match.

The control code and ID code in the ROM consists of four 32-bit words. Figure 7.7 shows the configuration of the control code and ID code. The ID code should be set in 32-bit units.

	31 — 24	23 — 16	15 — 8	7 — 0
FFFF FFA0h	Control code	ID code 1	ID code 2	ID code 3
FFFF FFA4h	ID code 4	ID code 5	ID code 6	ID code 7
FFFF FFA8h	ID code 8	ID code 9	ID code 10	ID code 11
FFFF FFACh	ID code 12	ID code 13	ID code 14	ID code 15

Figure 7.7 Configuration of Control Code and ID Code in ROM [1], page 1792.

Control Code

The control code determines whether ID code protection is enabled or disabled and the method of authentication to use with the host. Table 7.26 lists how the control code determines the method of authentication.

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 182

TABLE 7.26 Specifications for ID Code Protection [1], page 1792.

CONTROL CODE	ID CODE	STATE OF PROTECTION	OPERATIONS AT THE TIME OF SCI CONNECTION
45h	As desired	Protection enabled (authentication method 1)	Matching ID code: The command wait is entered.
			Non-matching ID code: The ID code protection wait is entered again.
			However, if a non-matching ID code is received three times in a row, all blocks are erased.
52h	Sequences other than 50h, 72h, 6Fh, 74h, 65h, 63h, 74h, FFh, . . ., FFh	Protection enabled (authentication method 2)	Matching Id code: The command wait is entered.
			Non-matching ID code: The ID code protection wait is entered again.
	50h, 72h, 6Fh, 74h, 65h, 63h, 74h, FFh, . . ., FFh	Protection enabled (authentication method 3)	Always judged to be a non-matching ID code.
Other than above	—	Protection disabled	All blocks are erased.

ID Code

The ID code can be set to any desired value. However, if the control code is 52h and the ID code is 50h, 72h, 6Fh, 74h, 65h, 63h, 74h, FFh, . . ., FFh (from the ID code 1 field), there is no determination of matching and the ID code is always considered to be non-matching. Accordingly, reading, programming, and erasure from the host are prohibited.

Program Example for ID Code Setting

The following assembler directives set up a control code of 45h and an ID code of 01h, 02h, 03h, 04h, 05h, 06h, 07h, 08h, 0Ah, 0Bh, 0Ch, 0Dh, 0Eh, 0Fh (from the ID code 1 field).

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 183


```
SECTION ID_CODE,CODE
.ORG 0FFFFFFA0h
.LWORD 45010203h
.LWORD 04050607h
.LWORD 08090A0Bh
.LWORD 0C0D0E0Fh
```

7.5 COMPLEX EXAMPLES

EXAMPLE 1

The following example creates a function to notify the clock supplied to the Flash unit. The input to this function would be the Flash address (flash_addr) we will be erasing or writing to.

```
 1. static uint8_t notify_peripheral_clock(FCU_BYTE_PTR flash_addr){
 2.    int32_t wait_cnt;
 3.    FLASH.PCKAR.WORD = (FLASH_CLOCK_HZ/1000000);
 4.    *flash_addr = 0xE9;
 5.    *flash_addr = 0x03;
 6.    *(FCU_WORD_PTR)flash_addr = 0x0F0F;
 7.    *(FCU_WORD_PTR)flash_addr = 0x0F0F;
 8.    *(FCU_WORD_PTR)flash_addr = 0x0F0F;
 9.    *flash_addr = 0xD0;
10.    wait_cnt = WAIT_MAX_NOTIFY_FCU_CLOCK;
11.    while(FLASH.FSTATR0.BIT.FRDY == 0){
12.       wait_cnt—;
13.       if(wait_cnt == 0){
14.          flash_reset();
15.          return FLASH_FAILURE;
16.       }
17.    }
18.    if(FLASH.FSTATR0.BIT.ILGLERR == 1){
19.       return FLASH_FAILURE;
20.    }
21.    return FLASH_SUCCESS;
22. }
```

Line 2 declares a wait counter variable. Line 3 notifies the PCLK and sets frequency of PCLK in MHz. Line 4 to 8 executes the peripheral clock notification command. Line 9 sets the timeout wait duration. Line 11 checks for the FRDY bit to be zero. If it is then it

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 184

decrements the wait counter variable (line 12). If the wait counter reaches zero, it means that the timeout duration has elapsed and it assumes operation failure, hence it resets the FCU (lines 13 through 15).

EXAMPLE 2

This example shows how to read the program lock bit which is set to ensure safety of the Flash. The input to this function would be the block from which the lock bit is read.

```
 1. uint8_t R_FlashReadLockBit(uint8_t block){
 2.    FCU_BYTE_PTR pAddr;
 3.    uint8_t result = FLASH_SUCCESS;
 4.    uint8_t lock_bit;
 5.    if( flash_grab_state(FLASH_LOCK_BIT) != FLASH_SUCCESS ){
 6.       return FLASH_BUSY;
 7.    }
 8.    pAddr = (FCU_BYTE_PTR)(g_flash_BlockAddresses[ block ]);
 9.    g_current_mode = ROM_PE_MODE;
10.    if( enter_pe_mode((uint32_t)pAddr) != FLASH_SUCCESS){
11.       exit_pe_mode();
12.       flash_release_state();
13.       return FLASH_FAILURE;
14.    }
15.    *pAddr = 0x71;
16.    if(FLASH.FSTATR0.BIT.ILGLERR == 1) {
17.       result = FLASH_FAILURE;
18.    }
19.    else{
20.       lock_bit = *pAddr;
21.       if(lock_bit != 0x00){
22.          result = FLASH_LOCK_BIT_NOT_SET;
23.       }
24.       else{
25.          result = FLASH_LOCK_BIT_SET;
26.       }
27.    }
28.    exit_pe_mode();
29.    flash_release_state();
30.    return result;
31. }
```

07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 185

In this function, Line 2 declares an address pointer. Line 3 declares an operation result container variable. Line 4 holds the outcome of lock-bit read. If the attempt to grab state fails, then it exits the function stating that the flash is busy (line 5 to 7). Line 8 is the Flash command address to get the value in the block. Then FCU is set to ROM PE mode and exit PE mode if the operation is successful, or else it exits the function by making the flag FLASH_FAILURE as 1 (lines 9 to 14).

Now switch to ROM lock-bit read mode by the setting values to the appropriate bits of the register (line 15). Now read the lock-bit value. If the lock bit is set, then the FLASH_LOCK_BIT_SET is set to 1, or else the FLASH_LOCK_BIT_NOT_SET is set to 1 (lines 21 to 26).

7.6 RECAP

In this chapter, Flash and EEPROM programming for Renesas RX63N was analyzed in detail, where the different modes of operations and FCU registers were discussed. The chapter begins by explaining the difference between flash and EEPROM programming. The different modes of operation, namely programming/erasure mode, read only mode, and lock-bit mode were analyzed in detail. The FCU registers were explained at the bit level. The EEPROM project for erasing data byte by byte was explained in brief. A few examples were provided to demonstrate the FCU registers used for flash programming.

7.7 REFERENCES

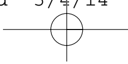
[1] Renesas Electronics, Inc. (February, 2013). *RX63N Group, RX631 Group User's Manual: Hardware,* Rev. 1.60.

[2] Renesas Electronics America, Inc. (February, 2013). *RX600 & RX200 Series Virtual EEPROM for RX, Renesas 32-Bit Microcomputer, Application Note,* Rev.1.70.

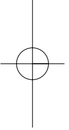
7.8 EXERCISES

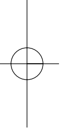
1. What is the major difference between Flash memory and EEPROM?
2. What are the two types of flash memory used in RX63N and what are their sizes?
3. What are the different types of operating modes associated with flash memory?
4. Say, for example, the data area for the DataFlash starts at 0010 0000h. Calculate the address of block 127 in the block configuration of the flash.
5. What are the different ways the flash can be protected?

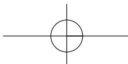
07.ES_Conrad_RX63N_Advanced_CH07.qxd:RX63N Advanced 3/4/14 12:01 PM Page 186



6. What are the applications of flash memory?
7. Write a C function to clear the status of the E2 DataFlash operation.
8. What are the different states that are possible in a VEE operation related to Virtual EEPROM?
9. What is the difference between a blocking and non-blocking operation?
10. Why should there be hardware support for background operations for a Virtual EEPROM project?







Universal Serial Bus (USB) Connectivity

8.1 LEARNING OBJECTIVES

This chapter discusses the Universal Serial Bus (USB) basics, implementation of the USB specification, and some advanced USB features. The first section covers the basics of USB 2.0. In the second section we will see how to communicate with the USB device. The third section explores the advanced concepts of USB connectivity. The last section describes the Human Interface Device Driver class in detail.

8.2 BASIC CONCEPTS OF USB CONNECTIVITY

8.2.1 USB Interface Specifications

The USB is an industry standard for serial communication. The USB specification contains details about the electrical signaling of the various pins on the connector and its physical dimensions, protocol layer etc. USB has many advantages when compared to other interfaces, such as a common connector, interoperability between many devices, simplified connectivity to enable 'Plug and Play,' and higher data transfer rates at lower costs.

8.2.1.1 Cabling and Connectors

USB uses a four-wire cable interface. Two of the wires, labeled as "D+" and "D−", are used in differential mode for transmitting and receiving data, and the other two wires are for power and ground. Historically, there were two different connectors designed (Type A and B), one on each end of a USB cable.

Original USB Connectors: The Type A Connector is used for upstream communication and connects to the host. The Type B Connector is used for downstream communication and connects to the device. Physically, the power pins (VBus and Gnd) are longer than the data pins so that power is applied to the device first before the physical data connection is

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 188

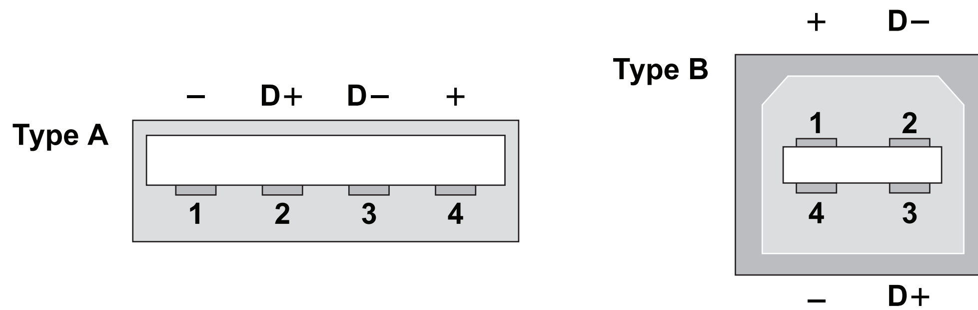


Figure 8.1 USB Connector Types A and B.

TABLE 8.1 USB Wire Usage

WIRE	NAME	USAGE	COLOR
1	Ground	Power	Black
2	D+	Data	Green
3	D–	Data	White
4	VBus	Power	Red

made. Also, when unplugging the device the data pins disconnect before the power supply pins.

USB Mini Connectors: Mini connectors are a substitute to the standard B type connector. As the name suggests these connectors are smaller and hence used on smaller portable and hand-held devices. The mini-B connectors have an additional fifth pin, named ID, but it is not connected. Mini connectors were replaced by even smaller micro connectors typically used in small handheld devices.

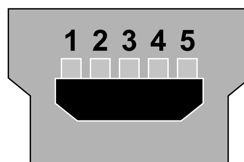


Figure 8.2 USB Mini Connector.

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 189

TABLE 8.2 USB Mini Wire Usage.

WIRE	NAME	USAGE	COLOR
1	Ground	Power	Black
2	ID	No Connection	
3	D+	Data	Green
4	D−	Data	White
5	VBus	Power	Red

USB On-The-Go (OTG) Micro Connectors: One of the biggest concerns with USB is that it is host controlled; that is, if the USB host is shut off, the USB stops working. Also, USB does not support peer-to-peer communication. A USB device (like a camera) cannot communicate directly with another USB device (like a printer); they need to communicate via a common host (camera to PC to printer). To resolve this problem a new standard, *USB OTG,* was created which can act as both host and peripheral. The connectors on the USB OTG devices are the Micro-USB type. With the ID pin, these connectors determine the type of device, whether a Micro-A or a Micro-B plug is being used. When a Micro-A plug is connected the resistance on the ID pin to ground is low ($<10\ \Omega$); in this case ID=FALSE. When a Micro-B plug is used, the resistance on the ID pin to ground is greater ($>100\ \Omega$); in this case ID=TRUE [1].

8.2.1.2 *Electrical Specifications*

There are specific electrical specifications of USB interfaces and cabling. Most prominent is that for USB 2.0 a device may draw a maximum of 500 mA from a single port [1]. Some other electrical characteristics include:

- Either the D+ or the D− line will be pulled high. If the D+ is high, the device is Full or High Speed. If D− is high, the device is Low speed.
- High Speed negotiation protocol occurs during the Bus Reset Phase.
- After detecting the reset signal, a high speed device will signal the host with a 480 Mbps chirp [1].

8.2.2 Host and Devices

USB can connect a large number of devices using a tiered star topology as shown in Figure 8.3. The key elements in USB topology are the host, hubs, and devices. The hubs

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 190

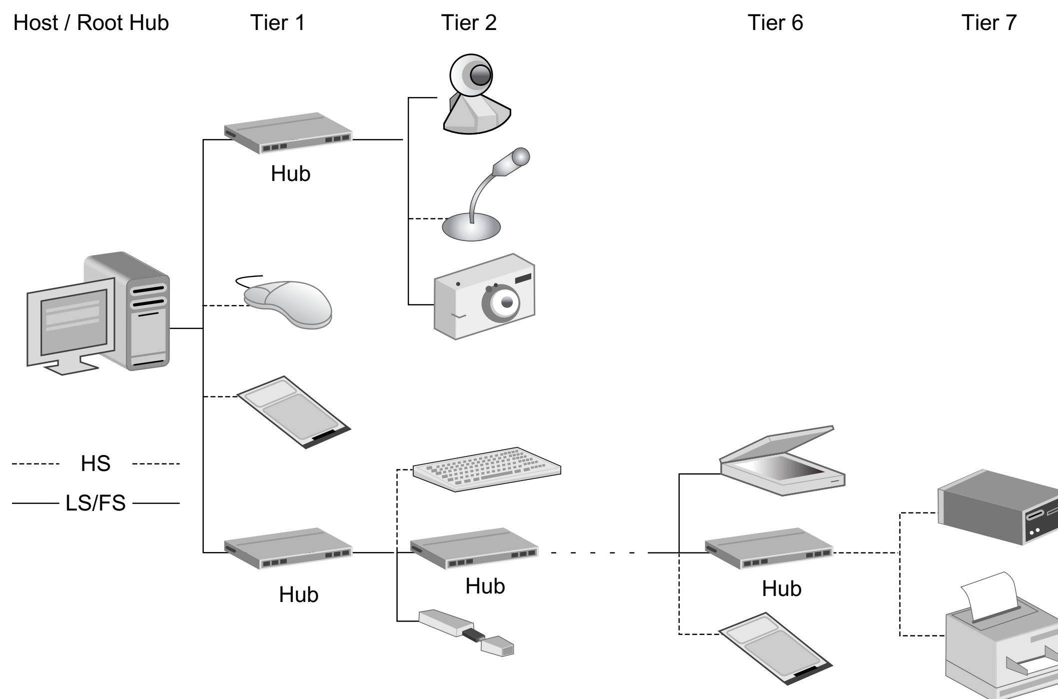


Figure 8.3 USB topology [1], page 36.

are bridges that expand the logical and physical fan-out of the network. A hub has a single upstream connection (to a host or a hub closer to the host) and many downstream connections (to devices). The host schedules and initiates data transfers. Up to 127 devices can be connected to any one USB host at any given time. The host and hubs power the devices. We can have a maximum of seven levels of tiers, with each tier formed from a hub.

USB Host

The USB host communicates to devices using a *USB host controller.* The USB host controllers have their own specifications. With USB 1.1, there were two Host Controller Interface Specifications:

UHCI (Universal Host Controller Interface), developed by Intel, placed more burden on software (Microsoft) for cheaper hardware.
OHCI (Open Host Controller Interface), developed by Compaq, Microsoft, and National Semiconductor, put more burden on hardware (Intel) for simpler software.

For USB 2.0 a new HCI Specification was needed to describe the USB's register level details. So EHCI (Enhanced Host Controller Interface) was introduced in this specification.

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 191

The USB host is responsible for detecting and enabling devices, managing bus access, performing error checking, providing power to devices, and initiating communications on the bus. The host's communication with devices consists mainly of queries.

USB Device

The functionality of a USB device is identified by unique class codes. The host uses these class codes to identify the connected device and load the appropriate driver. This makes the host independent of the device that is connected to it. This also makes it easier for the host to adapt to newer devices and support them, irrespective of the manufacturer. Examples of USB devices are keyboards, webcams, speakers, and mice.

8.2.3 Transfers, Transactions, and Frames

A transfer is the largest unit of communication in USB and consists of one or more transactions that can carry data to or from an endpoint. A transaction is made up of a sequence of three packets: Token packet (Header), Data packet (Payload), and Status/Handshake packet (Figure 8.4).

- *Token Packet:* There are three types of token packets:
  - In—Tells USB device that the host wants to read information.
  - Out—Informs USB device that the host wishes to send information.
  - Setup—Used to begin control transfers.
- *Data Packet:* The maximum payload size for low-speed devices is 8 bytes, the maximum payload size for high-speed devices is 1024 bytes, and data must be sent in multiples of bytes.
- *Handshaking Packet:* Tells if the data and token are successfully received. Also reports if the endpoint is stalled or not accepting data.

The Universal Serial Bus specification defines four transfer/endpoint types:

- *Control Transfers:* Control Transfers are used by the host to send standard requests during and after enumeration [1]. The host learns about the device's capabilities through Standard requests.
- *Bulk Transfers:* Bulk transfers are targeted for devices that exchange bulk blocks of data, to such an extent that it takes all of the available bus bandwidth. Error detection and retransmission mechanisms are implemented in hardware to guarantee data integrity and reliability. Timing is not guaranteed in bulk transfers. Mass storage devices are the best example of devices that use bulk transfers.
- *Interrupt Transfers:* Interrupt transfers are for devices with latency constraints [1]. Devices using interrupt transfer are provided with a polling interval which determines

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 192

when the scheduled data is transferred over the bus. Interrupt transfers are typically used to notify events.

- *Isochronous Transfers:* Isochronous transfers are used by audio and video devices that require data delivery to happen at a constant rate with a certain level of error-tolerance. Retransmission is not supported in isochronous transfers.

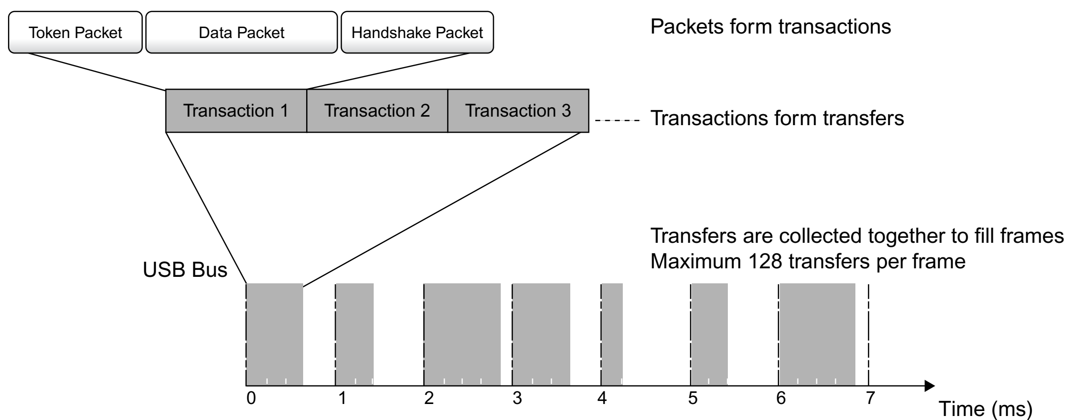


Figure 8.4 USB Frame Breakdown [1], page 53.

8.2.4 Class Drivers

8.2.4.1 Communication Device Class

The Communications Device Class (CDC) defines a framework to encapsulate existing communication service standards using a USB link. Various telecommunication and networking devices are included in the CDC. These include such devices as analog modems and ISDN terminal adapters. Examples of networking devices include Ethernet adapters and hubs. Communication devices handle call management, data transmission, and general device management. Seven major groups of devices are categorized by the CDC based on the model of communication they use. A single group may include numerous subclasses. Aside from the CDC base class, each group of devices has its own specification document. The seven groups are as follows:

- Public Switched Telephone Network devices.
- Integrated Services Digital Network devices, such as terminal adaptors and telephones.

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 193

- Ethernet Control Model devices.
- Asynchronous Transfer Mode devices.
- Wireless Mobile Communications devices.
- Ethernet Emulation Model devices.
- Network Control Model devices.

A Communications Class Interface (CCI) is a CDC device interface that manages the device and, optionally, the calls. The device management handles general device control, device configuration, and the notification of events to the host. The call management handles the formation and termination of calls. Because the communication model specifications supported by a CDC device are defined by its CCI, all CDC devices must have a CCI. Any defined USB class interface, such as Audio or vendor-specific interfaces, can be paired with the CCI. All vendor-specific interfaces are represented with Data Class Interfaces (DCIs).

A DCI is a CDC device interface that manages data transmission. A specific format is not required for transmitted and/or received data. The data used by DCIs can follow a proprietary format or simply be raw data from a communication line. DCIs are always subordinate to a CCI.

At least one CCI and zero or more DCIs are required for any given CDC device. One CCI and any subordinate DCI together provide a feature to the host often called a function. Several functions can exist in a CDC composite device, and the device would therefore be composed of several sets of CCI and DCI(s) as shown in Figure 8.5.

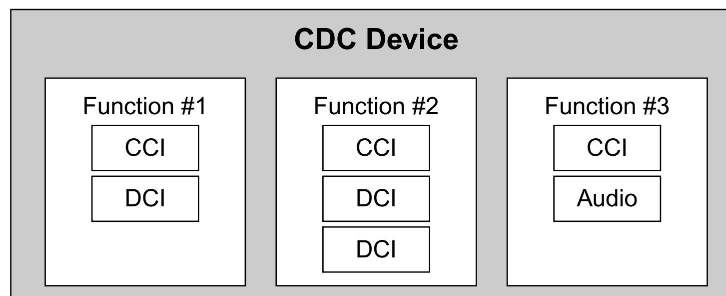


Figure 8.5 CDC Function Examples [1], page 165.

Table 8.3 provides a list of all the different types of endpoints distinguished by their data flow direction, interface, and application. CDC devices use combinations of these endpoints. An interrupt endpoint is often used by communication devices to notify the host of any events.

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 194

TABLE 8.3 CDC Endpoints [1], page 166.

ENDPOINT	DIRECTION	INTERFACE	USE FOR
Control IN	Device-to-host	CCI	Standard requests for enumeration, class-specific requests, device management and, (optionally) call management.
Control OUT	Host-to-device	CCI	Standard requests for enumeration, class-specific requests, device management and, (optionally) call management.
Interrupt or bulk IN	Device-to-host	CCI	Events notification, such as ring detect, serial line status, network status.
Bulk or isochronous IN	Device-to-host	DCI	Raw or formatted data communication.
Bulk or isochronous OUT	Host-to-device	DCI	Raw or formatted data communication.

The seven major models of communication encompass several subclasses that describe the way the device should use the CCI to handle the device and call management. All of the possible subclasses and their corresponding communication models are shown in Table 8.4.

8.2.4.2 Human Interface Device Class

This driver class will be discussed in detail in Section 8.4.

8.2.4.3 Mass Storage Device Class

The Mass Storage Class (MSC) provides the means to transfer information such as executable programs, source code, documents, images, and configuration data to and from a USB device. The USB device is usually a flash drive or an SD card, and a host recognizes it as an external storage medium.

A file system is required to define how the files are arranged within the storage media. However, a conforming device does not need any particular file system in order to be

TABLE 8.4 CDC Subclasses [1], page 167.

SUBCLASS	COMMUNICATION MODEL	EXAMPLE OF DEVICES USING THIS SUBCLASS
Direct Line Control Model	PSTN	Modem devices directly controlled by the USB host
Abstract Control Model	PSTN	Serial emulation devices, modem devices controlled through a serial command set
Telephone Control Model	PSTN	Voice telephony devices
Multi-Channel Control Model	ISDN	Basic rate terminal adaptors, primary rate terminal adaptors, telephones
CAPI Control Model	ISDN	Basic rate terminal adaptors, primary rate terminal adaptors, telephones
Ethernet Networking Control Model	ECM	DOC-SIS cable modems, ADSL modems that support PPPoE emulation, Wi-Fi adaptors (IEEE 802.11-family), IEEE 802.3 adaptors
ATM Networking Control	ATM	ADSL modems
Wireless Handset Control Model	WMC	Mobile terminal equipment connecting to wireless devices
Device Management	WMC	Mobile terminal equipment connecting to wireless devices
Mobile Direct Line Model	WMC	Mobile terminal equipment connecting to wireless devices
OBEX	WMC	Mobile terminal equipment connecting to wireless devices
Ethernet Emulation Model	EEM	Devices using Ethernet frames as the next layer of transport.
		Not intended for routing and Internet connectivity devices
Network Control Model	NCM	IEEE 802.3 adaptors carrying high-speed data bandwidth on network

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 196

accessible with the USB mass storage class specification. Instead, reading and writing sectors of data is done through a simple interface called the Small Computer System Interface (SCSI) transparent command set.

MSC transport protocols supported by the USB mass storage host class are Bulk-Only Transport and Control/Bulk/Interrupt Transport. Mass storage commands follow a protocol which utilizes a Command Block Wrapper (CBW) and a Command Status Wrapper (CSW). The protocol is shown in Figure 8.6.

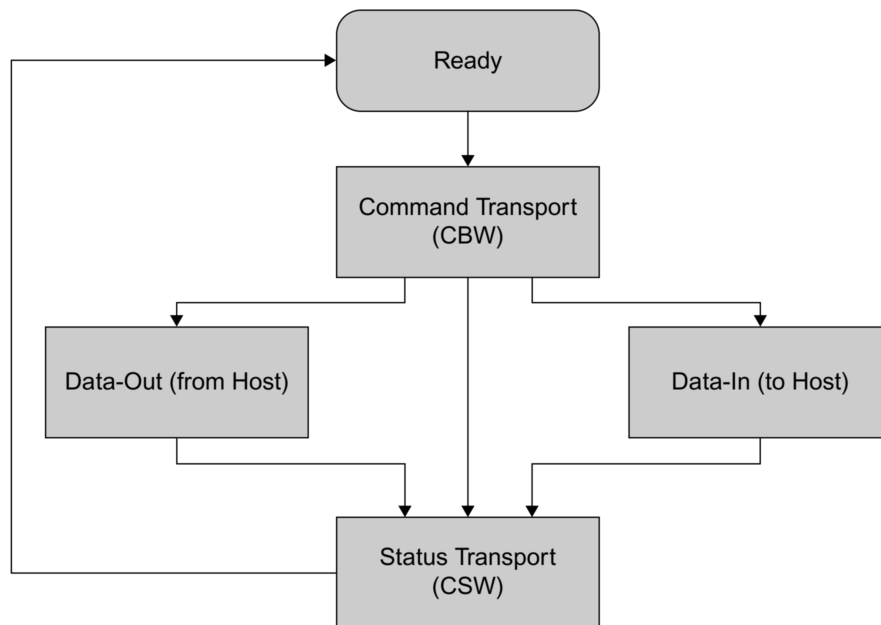


Figure 8.6 MSC Protocol [1], page 217.

8.3 BASIC EXAMPLES

The following examples where taken from Rev. 2.10 of Renesas' application notes on their USB Host Human Interface Device Class Driver [2]. For the most up to date information on Renesas' drivers, go to their website and download the supporting documentation for their drivers.

8.3.1 Example 1: Detecting a Device

Format:

int16_t open(int8_t *name, uint16_t mode, uint16_t flg)

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 197

Argument:

*name	Class Code
mode	Open mode, set to 0 (not used)
flg	Open flag, set to 0 (not used)

Return Value:

—File number

Description:

This function will enumerate the connected USB device. A hardware pipe based on the USB information received will be set up, and a connection with the USB device will be established. If enumeration and HW pipe allocation are normal, this function returns a file number ranging from 0x10 to 0x1f. If Enumeration and HW pipe allocation fail, (−1) is returned. After the file number is received by the caller, the USB device class communications using read() can be performed.

NOTES:

- Call this function from the user application program.
- Because the file number is required for USB device class communications using read(), the open function must be called before performing the communication.
- The second argument (mode) and third argument (flag) cannot be used with API: please set both to 0.

EXAMPLE

```
1. int16_t usb_smp_fn;
2. void usb_apl_open() {
3.    usb_smp_fn = open((int8_t *)USB_CLASS_HHID, 0, 0);
4.    if(usb_smp_fn != -1) {
5.       //USB Transfer
6.    }
7. }
```

8.3.2 Example 2: Ending Connection to a USB Device

Format

int16_t close(int16_t fileno)

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 198

Argument

fileno File number

Return Value

0: Successful
−1: Failure

Description

This function ends the connection with the USB device given by the file number. If the operation ends successfully, (0) is returned; if the operation fails, (−1) is returned.

NOTES:

- Call this function from the user application program.

EXAMPLE

```
1. int16_t usb_smp_fn;
2. void usb_apl_close() {
3.    USB_ER_t err;
4.    err = close(usb_smp_fn);
5.    if(err == USB_OK) {
6.       usb_smp_fn = -1;
7.    }
8. }
```

8.3.3 Example 3: Receiving

Format

int32_t read(int16_t fileno, uint8_t *buf, int32_t count)

Argument

fileno File number
*buf Pointer to data buffer
count Data transfer size

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 199

Return Value

.. Error Code. Always (−1)

Description

This function executes a data receive request for the USB device class specified by the file number. Data is read from the FIFO buffer in the specified data transfer size (3rd argument) and then stored in the data buffer (2nd argument). When the receive process is complete, the call-back function set in control (USB_CTL_RD_NOTIFY_SET) is called. The actual read size can be obtained from control (USB_CTL_RD_LENGTH_GET) after the receive process is complete.

NOTES:

- Call this function from the user application program.
- This function only executes a data receive request and does not block any processes. Therefore the return value is always −1.
- Use control (USB_CTL_RD_NOTIFY_SET) to register the call-back function for a notification of the data transfer being complete and then call the API.

EXAMPLE

```
 1. int16_t usb_spvendor_bulk_fn;
 2. void* data_len;
 3. int32_t size;
 4. uint8_t *buf;
 5. void* state;
 6.
 7. //Processing at the time of the completion of reception
 8. void usb_smp_Read_Notify (USB_UTR_t *ptr, uint16_t data1,
    uint16_t data2) {
 9.    USB_ER_t err;
10.    //Receiving data length check
11.    err = control(usb_spvendor_bulk_fn, USB_CTL_RD_LENGTH_GET,
       &data_len);
12.    if(err != USB_CTL_ERR_PROCESS_COMPLETE) {
13.       //Error Processing
14.    }
15. }
16.
```

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 200


```
17. void usb_apl_read() {
18.    USB_ER_t err;
19.    //Set data receive complete notification call-back
20.    control(usb_smp_fn, USB_CTL_RD_NOTIFY_SET,
       (void*)&usb_smp_Read_Notify);
21.    //receiving data request
22.    read(usb_smp_fn, (uint8_t *)buf, (int32_t)size);
23.    //receiving request status check
24.    err = control(usb_spvendor_bulk_fn, USB_CTL_GET_RD_STATE,
       (void*)&state);
25.    if(err != USB_CTL_ERR_PROCESS_COMPLETE) {
26.       //Error Processing
27.    }
28. }
```

8.4 HUMAN INTERFACE DEVICE DRIVER CLASS

8.4.1 Overview

USB supports Human Interface Devices (mouse, keyboard, joystick); Human Interface Devices (HID) are supported by the HID Driver Class. Table 8.5 provides a list of all the different types of endpoints distinguished by their data flow direction, interface, and application. HID devices use combinations of these endpoints. An interrupt endpoint is often used by communication devices to notify the host of any events.

TABLE 8.5 HID Class Endpoints Usage [1], page 186.

ENDPOINT	DIRECTION	USAGE
Control IN	Device-to-host	Standard requests for enumeration, class-specific requests, and data communication (input, Feature reports sent to the host with GET_REPORT requests)
Control OUT	Host-to-device	Standard requests for enumeration, class-specific requests, and data communication (output, Feature reports received from the host with SET_REPORT requests)
Interrupt IN	Device-to-host	Data communication (input and Feature reports)
Interrupt OUT	Host-to-device	Data Communication (output and Feature reports)

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:57 PM Page 201

8.4.2 Reports

The exchange of data between a host and an HID is done via reports. A report gives formatted information about controls and other physical entities of the HID. A user can manipulate these controls and operate parts of the device. These controls could be a switch, a button on a mouse, or a knob, for example. Other entities are used to notify the user about the state of a device such as LEDs on a keyboard notifying the user about the caps lock or key pad being on.

Analysis of a report descriptor via a parser provides the host information about the use and format of report data. Data provided by each control in a device is described by a report descriptor which is composed of items. Items are specific pieces of information about the device. They consist of a 1-byte prefix and variable-length data. Items are broken into three categories [1]:

- Main item defines or groups certain types of data fields.
- Global item describes data characteristics of a control.
- Local item describes data characteristics of a control.

Different functions, also called tags, define each item type and can be seen as sub-items. Each of these sub-items belongs to one of the three principal item types. A brief overview of the item functions in each item type is given in Table 8.6.

8.4.3 Architecture

Devices are enumerated by the host operating system (OS) using the control endpoints. After enumeration, interrupt endpoints are used by the host to start transmission/reception to/from the device.

A device using the HID class must interact with an OS layer (Figure 8.7) specific to this class. OS services required for the internal functioning of the HID class are provided by this HID OS layer. This layer is independent of any specific OS.

When the HID class is initializing, the report provided by the application is validated by a report parser. The initialization will fail if any error is detected by the parser.

8.4.4 More Examples of HID Driver Functions

As with the previous code examples, the following examples where taken from Rev. 2.10 of Renesas' application notes on their USB Host Human Interface Device Class Driver [2].

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:58 PM Page 202

TABLE 8.6 Function Description for each item [1], pages 187–189.

ITEM TYPE	ITEM FUNCTION	DESCRIPTION
Main	Input	Describes information about the data provided by one or more physical controls.
	Output	Describes data sent to the device.
	Feature	Describes device configuration information sent to or received from the device which influences the overall behavior of the device or one of its components.
	Collection	Group related items (Input, Output or Feature).
	End of Collection	Closes a collection.
Global	Usage Page	Identifies a function available within the device.
	Logical Minimum	Defines the lower limit of the reported values in logical units.
	Logical Maximum	Defines the upper limit of the reported values in logical units.
	Physical Minimum	Defines the lower limit of the reported values in physical units, that is the Logical Minimum expressed in physical units.
	Physical Maximum	Defines the upper limit of the reported values in physical units, that is the Logical Maximum expressed in physical units.
	Unit Exponent	Indicates the unit exponent in base 10. The exponent ranges from -8 to $+7$.
	Unit	Indicates the unit of the reported values. For instance, length, mass, temperature units, etc.
	Report Size	Indicates the size of the report fields in bits.
	Report ID	Indicates the prefix added to a particular report.
	Report Count	Indicates the number of data fields for an item.
Local	Usage	Represents an index to designate a specific Usage within a Usage Page. It indicates the vendor's suggested use for a specific control or group of controls. A usage supplies information to an application developer about what a control is actually measuring.
	Usage Minimum	Defines the starting usage associated with an array or bitmap.
	Usage Maximum	Defines the ending usage associated with an array or bitmap.
	Designator Index	Determines the body part used for a control. Index points to a designator in the Physical descriptor.
	Designator Minimum	Defines the index of the starting designator associated with an array or bitmap.
	Designator Maximum	Defines the index of the ending designator associated with an array or bitmap.
	String Index	String index for a String descriptor. It allows a string to be associated with a particular item or control.
	String Minimum	Specifies the first string index when assigning a group of sequential strings to controls in an array or bitmap.
	String Maximum	Specifies the last string index when assigning a group of sequential strings to controls in an array or bitmap.
	Delimiter	Defines the beginning or end of a set of local items.

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:58 PM Page 203

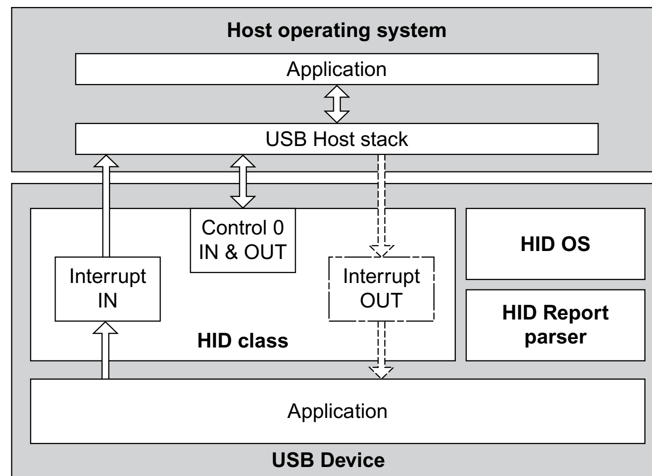


Figure 8.7 USB Stack [1], page 192.

Format

int16_t control(int16_t fileno, USB_CTRLCODE_t code, void *data)

Argument

fileno	File number
*buf	Pointer to data buffer
*data	Pointer to data (Use of this argument is different for different control codes. See Table 8.7)

Return Value

0 : Successful
−1 : Failure

Description

This function's processing depends on the control code.
If an unsupported code is specified, the function sends (−1) as the return value.

NOTES:

- Call this function from the user application program.
- If the user is using the ANSI method and specifies "USB_CTL_HID_CLASS_REQUEST" as the control code (2nd argument), the user can issue the following class requests (Table 8.8). Assign the definition of the Class Request to the "bRequestCode" member.

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:58 PM Page 204

TABLE 8.7 Supported Control Codes [2], page 25.

CONTROL CODE	DESCRIPTION
USB_CTL_USBIP_NUM	Get the USB module number.
	USB module number is set to the 3rd argument.
USB_CTL_RD_NOTIFY_SET	Register the function called back when a data receive request is completed.
	Set the call-back function in the 3rd argument.
USB_CTL_RD_LENGTH_GET	Get the data length read from the FIFO buffer when the data is read.
	The data length is set to the 3rd argument.
USB_CTL_GET_RD_STATE	Get the state when data is read.
	The state is set to the 3rd argument.
USB_CTL_H_RD_TRANSFER_END	Forcibly end the data transfer in the pipe relevant to the 1st argument (File number).
USB_CTLH_CHG_DEVICE_STATE	Change state of connected USB device.
	Set the state value to the 3rd argument.
USB_CTL_H_GET_DEVICE_INFO	Get state of connected USB device.
	The state is set to the 3rd argument.
USB_CTL_HID_CLASS_REQUEST	Issue a class request for HID, the request is given by the 3rd argument.

TABLE 8.8 Class Requests [2], page 26.

CLASS REQUEST	DEFINITION VALUE
Get_Descriptor(HID)	USB_HID_GET_HID_DESCRIPTOR
Get_Descriptor(Report)	USB_HID_GET_REPORT_DESCRIPTOR
Get_Descriptor(Physical)	USB_HID_GET_PHYSICAL_DESCRIPTOR
Set_Report	USB_HID_SET_REPORT
Get_Report	USB_HID_GET_REPORT
Set_Idle	USB_HID_SET_IDLE
Get_Idle	USB_HID_GET_IDLE
Set_Protocol	USB_HIS_SET_PROTOCOL
Get_Protocol	USB_HID_GET_PROTOCOL

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:58 PM Page 205

EXAMPLES

This code will retrieve the USB module number.

```
1. int16_t usb_smp_fn;
2. void getHIDModuleNumber(USB_UTR_t *ptr) {
3.    int16_t num;
4.    :
5.    //Confirmation USBIP Number
6.    control(usb_smp_fn, USB_CTL_USBIP_NUM, (void*) &num);
7.    :
8. }
```

The following function ends a data transfer.

```
1. int16_t usb_smp_fn;
2. void endHIDTransfer(USB_UTR_t *ptr){
3.   USB_CTL_PARAMETER_t smp_parameter;
4.    :
5.    smp_parameter.transfer_end.status = USB_DATA_STOP;
6.    //Forcibly ends data reception
7.    control(usb_smp_fn, USB_CTL_H_RD_TRANSFER_END,
      (void*)&smp_parameter);
8.    :
9. }
```

This example changes the state of a USB HID.

```
1. int16_t usb_smp_fn;
2. void setHIDState(USB_UTR_t *ptr) {
3.    USB_CTL_PARAMETER_t smp_parameter;
4.    :
5.    //Callback function
6.    smp_parameter.device_state.complete = ptr->complete;
7.    smp_parameter.device_state.msginfo = USB_DO_STALL;
8.    //Changing USB device information
9.    control(usb_smp_fn, USB_CTL_H_CHG_DEVICE_STATE,
      (void*)&smp_parameter);
10.   :
11. }
```

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:58 PM Page 206

This function will return the state of a USB device.

```
 1. int16_t usb_smp_fn;
 2. uint16_t *smp_tbl;
 3. void getHIDInfo(USB_UTR_t *ptr) {
 4.    USB_CTL_PARAMETER_t smp_parameter;
 5.    :
 6.    smp_parameter.device_information.tbl = smp_tbl;
 7.    /* Getting USB device information */
 8.    control(usb_smp_fn, USB_CTL_H_GET_DEVICE_INFO,
       (void*)&smp_parameter);
 9.    :
10. }
```

The following example performs a class request.

```
 1. uint16_t devaddr;
 2. void HIDClassRequest(USB_UTR_t *ptr){
 3.    USB_HHID_CLASS_REQUEST_PARM_t class_req;
 4.
 5.    //Class Request
 6.    class_req.bRequestCode = USB_HID_GET_HID_DESCRIPTOR;
 7.
 8.    //Device address of HID device
 9.    class_req.devadr = ptr->tranadr;
10.    class_req.ip = ptr->ip;
11.    class_req.ipp = ptr->ipp;
12.    //Pointer to the buffer that the class requests
13.    class_req.tranadr = p_data;
14.    class_req.complete = ptr->complete;
15.
16.    //HID Class Request
17.    control(usb_smp_fn, USB_CTL_CLASS_REQUEST, (void*)&class_req);
18. }
```

For information about using the control codes relevant to Read(), please see Section 8.3.3; Example 3: Receiving.

08.ES_Conrad_RX63N_Advanced_CH08.qxd:RX63N Advanced 3/4/14 9:58 PM Page 207

8.5 RECAP

In this chapter we covered the Universal Serial bus peripheral on the Renesas RX63N. We started with a brief overview of the USB concepts. Then we covered how to enable the USB and how to transfer data between two USB devices. Additionally, this chapter described the various class drivers of the USB. In the advanced concepts section we covered explained briefly about the implementation of the various driver classes. Our final example showed how to set up and use the Human Interface device driver class.

Renesas has several examples on their websites of USB applications in addition to reference [3].

8.6 REFERENCES

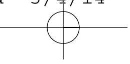
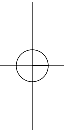
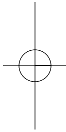
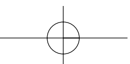
[1] The Micriμm USB Team (2012). *μc/USB Device™ Universal Serial Bus Device Stack,* Micriμm Press, Inc.

[2] Renesas Electronics, Inc. (April, 2013). *Application Note: Renesas USB MCU and USB ASSP USB Host Human Interface Device Class Driver (HHID),* Rev. 2.10.

[3] Renesas Electronics, Inc. (February, 2013). *RX63N Group, RX631 Group User's Manual: Hardware,* Rev. 1.60.

8.7 EXERCISES

1. What is a USB and who maintains its protocols?
2. What are the different connector types used for a USB? Briefly describe each of them.
3. What type of network configuration does the USB use and how many devices can be connected at one time?
4. What are the different USB transfer types? Briefly explain each of them.
5. Describe the Communication Device Class Functions and explain the purpose of the interfaces used for them.
6. Describe the phases used by the Mass Storage Device Class.
7. How does a Hardware Human Interface Device exchange data? Explain in detail.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:51 AM Page 209

RX63N Ethernet Controller

9.1 LEARNING OBJECTIVES

The term Ethernet can be understood as a family of networking technologies for Local Area Networks (LANs). This technology has been standardized as IEEE 802.3. Ethernet networks can also connect to the internet through a router, which provides access to Wide Area Networks. The availability of inexpensive Ethernet chips that handle the details of Ethernet transmission and reception has made the use of Ethernet technology feasible for embedded devices. The RX63N comes with an 802.3x compliant Ethernet MAC capable of 10/100 Mbps as well as an Ethernet DMA controller [1].

In this chapter the reader will learn how to:

- Set up an Ethernet connection
- Transmit data over an Ethernet connection
- Receive data over an Ethernet connection

9.2 BASIC CONCEPTS OF ETHERNET AND INTERNET PROTOCOL

Ethernet LANs consist of nodes and interconnected media. Network nodes can be either Data Terminal Equipment (DTEs) or Data Communication Equipment (DCEs), which forward and transmit frames in the network. DTEs can be PCs, workstations or servers. DCEs can be devices such as repeaters, network switches, network interface cards or modems.

9.2.1 Ethernet Network Topologies

Ethernet LANs can be configured in a variety of ways, but the three basic configurations are point-to-point connections, coaxial bus topology, and star-connected topology. The most basic configuration is the point-to-point connection in which only two nodes and a network link are involved. The connection may be DTE-to-DTE, DTE-to-DCE, or

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:51 AM Page 210

DCE-to-DCE. The length of the cable or network link depends on the type of cable and the method of transmission, which will be discussed a little later.

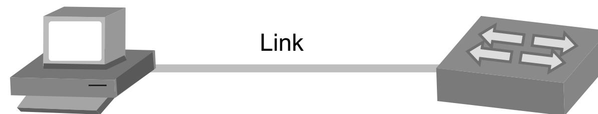


Figure 9.1 Point-to-Point Connection [2], page 18.

In the Co-axial Bus topology, individual nodes are connected to buses, and these buses are interconnected through repeaters. Earlier Ethernet networks used this method of interconnection. The size of the network is restricted because of constraints on the segment lengths and the number of connections.

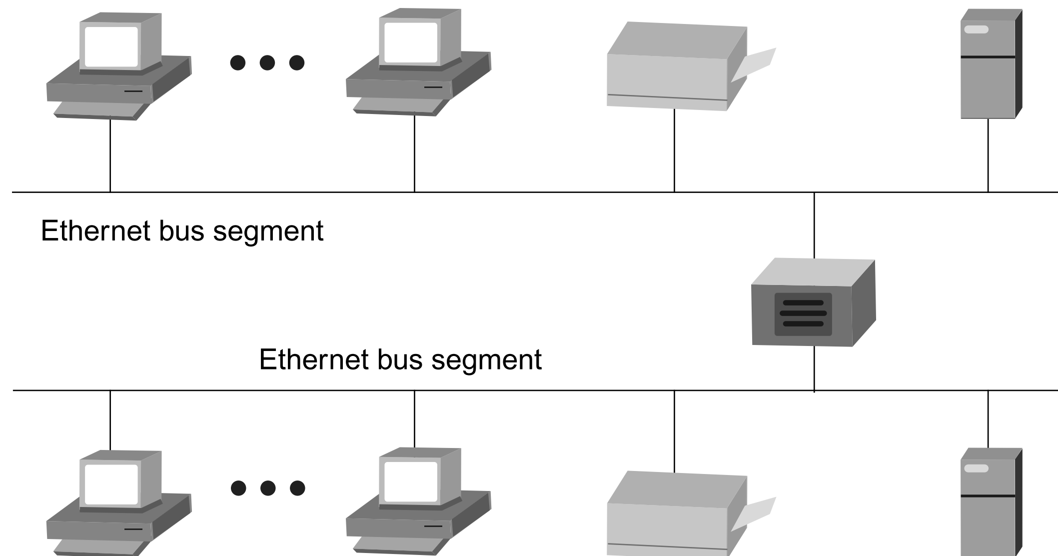


Figure 9.2 Co-axial Bus topology [2], page 18.

Ethernet supports a bus topology inherently. In this topology each node shares the same physical medium. Before any transmission, a node listens first and transmits if it decides that that there is no traffic on the medium. However, two nodes may decide to transmit at the same time and this will lead to a collisions. This also known as Carrier Sense Multiple Access with Collision Detection (CSMA/CD). A bus topology is half-duplex.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:51 AM Page 211

Due to cost reductions in cables and switches, nodes can be connected to each other through star topology where the node connections are peer to peer and full duplex. In full duplex mode, there is no need for carrier sense (listen first then transmit) and there are no collisions. This is because the physical medium is only between two nodes and transmit and receive are on separate wires. This topology not only allows full 10/100Mbps TX and RX communication but also increases the reliability of the network since a single link failure is only affects the node involved and not the entire bus.

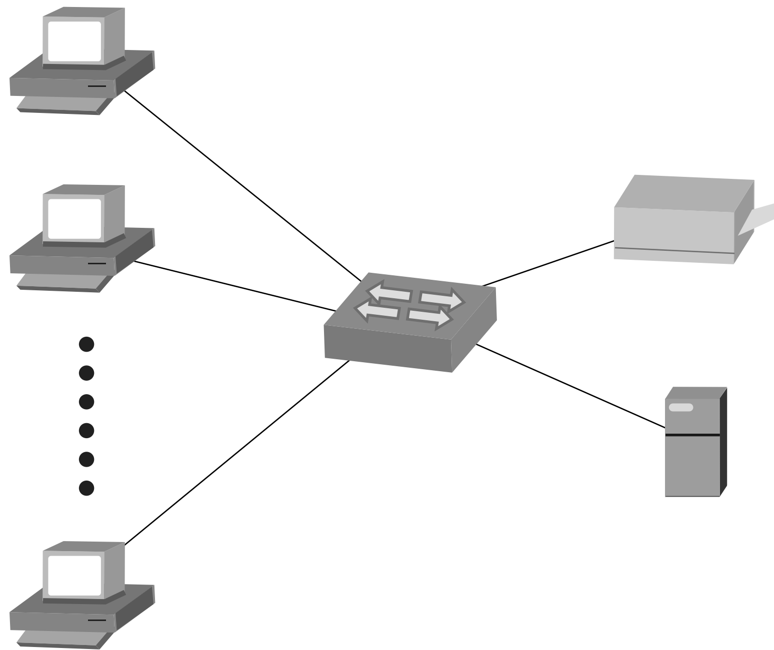


Figure 9.3 Star topology [2], page 18.

Newer networks are no longer connected using the bus topology configuration; instead they use a star configuration in which the central element is a network switch. In fact, this star topology is the most common topology and the one you are most likely to have encountered. This is the topology common with PCs connected via Ethernet cables to a router (or, likely, to just a RJ45 socket in the wall that eventually connects to switches and routers.

Ethernet is the physical-layer part of the entire seven-layer OSI model of communication. This is similar to the way we use the UART or CAN bus. This physical layer is most often used by the Internet protocol suite.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:51 AM Page 212

9.2.2 Internet Protocol

The Internet protocol suite is a set of well defined communications protocols commonly used for Internet communications. Two of the most common (and important) of these protocols is TCP/IP (Transmission Control Protocol and Internet Protocol). TCP/IP identifies how data is formatted and addressed at the communications source, then how it is routed and received at the communications destination. Although there are several “layers” associated with these protocols, often users interface with only the highest layer, the application layer. Examples of applications include File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), and the telnet virtual terminal communications protocol.

Renesas has provided sample code to demonstrate the RX63N board’s Ethernet capabilities [3]. This code includes the popular open source uIP Embedded TCP/IP Stack [4]. The code package can be downloaded from the Renesas site. A project with a demonstration can be built and downloaded to the RX63N board. Some versions of this project can be built with the free demo version of HEW as well as E2 Studio. uIP can support one TCP application at a time.

To communicate using this Renesas Ethernet API, the hardware address of the board (MAC address) must be known. This six-byte array is typically stored in E2Flash at location 0x00107FF2 to 0x00107FF7. The Renesas range of MAC addresses has the first four bytes of 0x00, 0x30, 0x55, and 0x80. The provided API functions can check this correct address range.

You can attach an Ethernet cable between the RX63N board and a router and run an API function to obtain an IP address. This function sends out a DHCP request to the DCHP server (in the router) in order to obtain the IP address. Once an IP address is obtained DHCP messages are sent at periodic intervals to extend the lease period for the IP address. This IP address of the RX63N board could be printed on the screen.

Once the board has an IP address, you can connect to the board using any number of TCP/IP protocols. These particular protocols are described in more detail in the uIP Manual. For example, a PC connected to the same router could issue a telnet command:

```
telnet <RX63N_IP_ADDRESS> 1000
```

You would need to write an application to establish a connection to the TCP port on the RX63N board. Whenever a TCP packet is received by uIP, it calls this application. This application is listening at port 1000.

9.2.3 Example

In the following example, the Ethernet API functions are called to set up the Ethernet Controllers and initialize uIP. We check to ensure an Ethernet cable is attached and connected to a router. Then we initialize the DHCP and telnet protocol, then wait for Ethernet packets and work with them.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:51 AM Page 213


```
 1. extern const struct uip_eth_addr my_mac;
 2. int main(void) {
 3.    //Set Ethernet address.
 4.    uip_setethaddr(my_mac);
 5.    uip_init();
 6.
 7.    //If links is down, wait until initialization is complete
 8.    //in case the Ethernet cable is not plugged in.
 9.    while (R_Ether_CheckLink_ZC(ch) != R_ETHER_OK);
10.
11.    //Initialize the MAC.
12.    R_Ether_Open_ZC(ch, (uint8_t*)&my_mac.addr[0],
       (void **)&uip_buf);
13.
14.    //R_Ether_WaitLink must be called at least once after
15.    //R_Ether_Open to complete link and Ethernet initializations.
16.    while (R_Ether_WaitLink_ZC(ch) != R_ETHER_OK);
17.
18.    //Initialize DHCP
19.    dhcpc_init(&my_mac.addr[0], 6);
20.    //We start to listen for connections on TCP port 1000.
21.    uip_listen(HTONS(1000));
21.
22.    while (1) {   //do forever
22.       uip_len = R_Ether_Read(ch, (void *)uip_buf);
23.       if (uip_len > 0) {
24.          //if something was received through telnet, then process
25.          .
26.          .
27.          .
28. }
```

There are many supporting files needed and this is only a small snippet of code from the provided project. Further descriptions and examples of the Ethernet API are show in Section 9.5.

9.3 ETHERNET CONTROLLER

This section includes detail of what is involved in the hardware associated with Ethernet. An Ethernet controller chip is the hardware that controls the transmission and reception of data in conformance to the IEEE 802.3 standards. The RX63N supports both the Media

Independent Interface (16 pins, 25 MHz) and Reduced Media Independent Interface (6 pins, 50 MHz) physical layer interfaces. Connecting a physical-layer Large-Scale Integrated device (LSI) (PHY-LSI) by complying with this standard enables the Ethernet Controller (ETHERC) to perform the transmission and reception of Ethernet/IEEE802.3 frames. The controller has one Media Access Control (MAC) layer interface port and is connected to the Ethernet Direct Memory Access Controller (EDMAC) inside this LSI, which carries out high-speed data transfer to and from the memory.

The Ethernet transmitter assembles the transmit data into a frame and outputs it to the physical layer interface (MII/RMII) when there is a transmit request from the transmit EDMAC. The data transmitted by way of the MII/RMII is transmitted to the lines by the PHY-LSI.

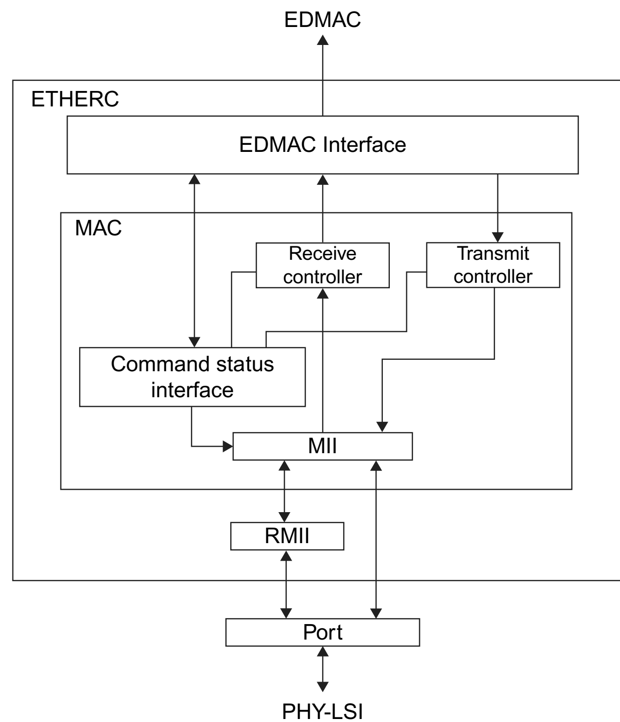


Figure 9.4 Configuration of the Ethernet Controller [1], page 1111.

When the transmit enable (ECMR.TE) bit is set, the transmitter enters an idle state. When a transmit request is issued by the transmit EDMAC, the ETHERC sends the preamble to the MII/RMII after a carrier has been detected and the transmission has been delayed with an equivalent frame interval time. If full-duplex transfer is selected, which does not require carrier detection, the preamble is sent as soon as a transmit request is issued by the transmit EDMAC. The transmitter then sends the SFD, data, and CRC sequentially.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:51 AM Page 215

At the end of a transmission, the transmit EDMAC generates a Transmission Complete (TC) interrupt. If a collision or the carrier-not-detected state occurs during data transmission, an interrupt is reported. The transmitter enters the idle state after waiting for the frame interval time and continues transmission if there is more transmit data.

Transmission is re-tried only when data of 512 bits or less is transmitted. If a collision is detected during transmission of data greater than 512 bits, only the collision indicator message is transmitted.

The ETHERC receiver separates the frame sent by the MII/RMII into preamble, SFD (start frame delimiter), data, and CRC, and the fields from DA (destination address) to the CRC data are transferred to the receiving EDMAC. Figure 9.6 shows the state transitions of the ETHERC receiver.

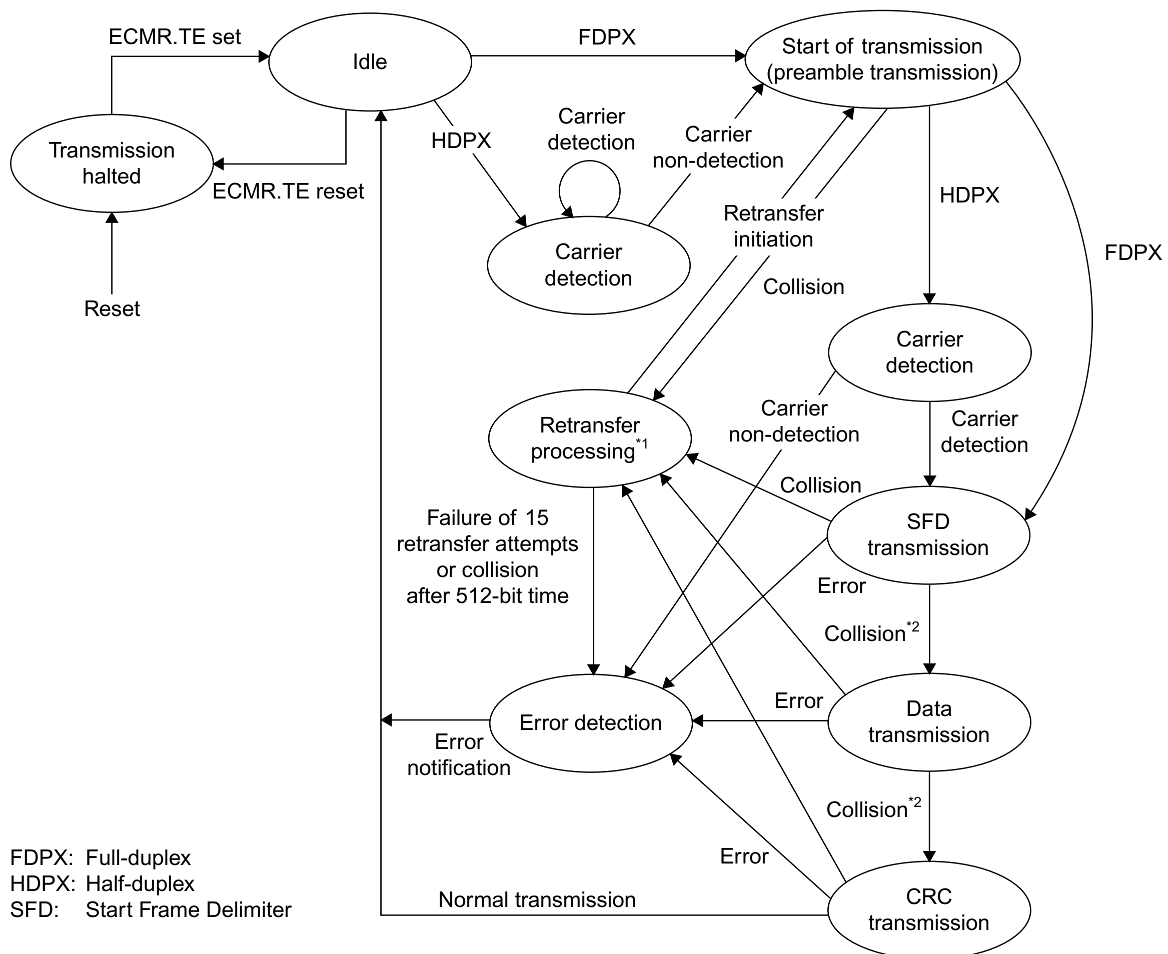


Figure 9.5 Transmission of Ethernet Frames [1], page 1131.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:51 AM Page 216

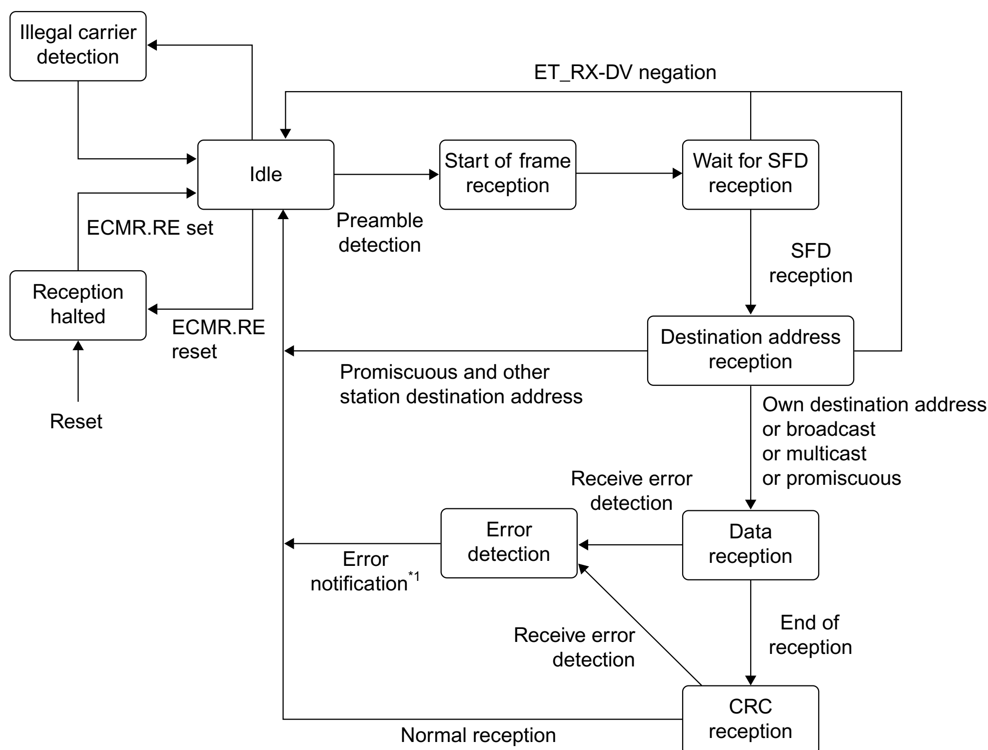


Figure 9.6 Reception of Ethernet Frames [1], page 1132.

When the receive enable (ECMR.RE) bit is set, the receiver enters the receive idle state. Upon detecting an SFD after a receive packet preamble, the receiver starts the receive process. It discards a frame with an invalid pattern. In normal mode, the receiver starts data reception if the destination address of the frame matches the RX63N address, or if the broadcast or multicast frame type is specified.

In promiscuous mode, the receiver starts reception for any type of frame. After receiving data from the MII/RMII, the receiver carries out a CRC check in the frame data field. The result is indicated as a status bit in the descriptor, after the frame data has been written to memory. The receiver reports an error status in the case of an abnormality. After the reception of one frame, the receiver prepares for the reception of the next frame if the receive enable bit is set (ECMR.RE = 1) in the ETHERC mode.

ETHERC Mode Register (ECMR): The Ethernet Mode Control Register specifies the operating mode of the controller. The settings in the ECMR should be made in the

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 217

ETHERC Mode Register (ECMR)

Address(es): 000C 0100h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	—	—	—	—	—	—	—	—	—	—	—	TPC	ZPF	PFR	RXF	TXF
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	PRCEF	—	—	MPDE	—	—	RE	TE	—	ILB	RTM	DM	PRM
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	PRM	Promiscuous Mode	0: ETHERC performs normal operation 1: ETHERC performs promiscuous mode operation	R/W
b1	DM	Duplex Mode	0: Half-duplex transfer is specified. 1: Full-duplex transfer is specified.	R/W
b2	RTM	Transmission/ Reception Rate	0: 10 Mbps 1: 100 Mbps	R/W
b3	ILB	Internal Loop Back Mode	0: Normal data transmission/reception is performed. 1: Data loopback is performed inside the MAC in the ETHERC when DM = 1.	R/W
b4	—	Reserved	This bit is always read as 0. The write value should always be 0.	R/W
b5	TE	Transmission Enable	0: Transmitting function is disabled. 1: Transmitting function is enabled.	R/W
b6	RE	Reception Enable	0: Receiving function is disabled. 1: Receiving function is enabled.	R/W
b8, b7	—	Reserved	These bits are always read as 0. The write value should always be 0.	R/W
b9	MPDE	Magic Packet™ Detection Enable	0: Magic Packet™ detection is not enabled. 1: Magic Packet™ detection is enabled.	R/W

Figure 9.7 Ethernet Mode Control Register [1], page 1113.—*Continued*

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b11, b10	—	Reserved	These bits are always read as 0. The write value should always be 0.	R/W
b12	PRCEF	CRC Error Frame Reception Enable	0: A frame with a CRC error is received as a frame with an error.	R/W
			1: A frame with a CRC error is received as a frame without an error.	
b15 to b13	—	Reserved	These bits are always read as 0. The write value should always be 0.	R/W
b16	TXF	Operating Mode for Transmitting Port Flow Control	0: PAUSE frame detection is disabled. (Automatic PAUSE frame is not transmitted)	R/W
			1: Flow control for the transmitting port is enabled. (Automatic PAUSE frame is transmitted as required)	
b17	RXF	Operating Mode for Receiving Port Flow Control	0: PAUSE frame detection is disabled	R/W
			1: Flow control for the receiving port is enabled	
b18	PFR	PAUSE Frame Receive Mode	0: PAUSE frame is not transferred to EDMAC	R/W
			1: PAUSE frame is transferred to EDMAC	
b19	ZPF	PAUSE Frame Usage with TIME = 0 Enable	0: Control of a PAUSE frame whose TIME parameter value is 0 is disabled	R/W
			1: Control of a PAUSE frame whose TIME parameter value is 0 is enabled	
b20	TPC	PAUSE Frame Transmission	0: PAUSE frames are transmitted even within PAUSE periods.	R/W
			1: PAUSE frames are not transmitted within PAUSE periods.	
b31 to b21	—	Reserved	These bits are always read as 0. The write value should always be 0.	R/W

Figure 9.7 Ethernet Mode Control Register [1], page 1113.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 219

initialization process, after a reset in most cases. The operating mode setting must not be changed while the transmitting and receiving functions are enabled. To switch the operating mode, return the ETHERC and the EDMAC to their initial states by means of the software reset bit (SWR) in the EDMAC mode register (EDMR) of the EDMAC before making settings again.

PRM Bit (Promiscuous Mode): Setting the PRM bit enables all Ethernet frames to be received. "All Ethernet frames" means all receivable frames, irrespective of any differences or the enabled/disabled status of destination address, broadcast address, multicast bit, etc.

RTM Bit (Transmission/Reception Rate): This bit specifies the transmission and reception bit rate when RMII is selected.

TE Bit (Transmission Enable): If a switch is made from transmitting function enabled (TE = 1) to disabled (TE = 0) while a frame is being sent, the transmitting function will be enabled until transmission of the corresponding frame is completed.

RE Bit (Reception Enable): If a switch is made from receiving function enabled (RE = 1) to disabled (RE = 0) while a frame is being received, the receiving function will be enabled until reception of the corresponding frame is completed.

MPDE Bit (Magic Packet™ Detection Enable): This bit enables or disables Magic Packet™ detection by hardware to allow activation from the Ethernet.

ZPF Bit (PAUSE Frame Usage with TIME = 0 Enable): When the ZPF bit is set to 0, the next frame is not transmitted until the time specified by the Timer value has elapsed. On receiving a PAUSE frame with a Timer value of 0, the PAUSE frame is discarded.

When the ZPF bit is cleared to 0 and the data size in the receive FIFO becomes smaller than the setting of the flow control, then start the FIFO threshold setting register (FCFTR) of the EDMAC. This operation needs to be completed before the Timer value elapses or an automatic PAUSE frame with a Timer value of 0 is transmitted. On receiving a PAUSE frame with a Timer value of 0, the transmission wait state is canceled.

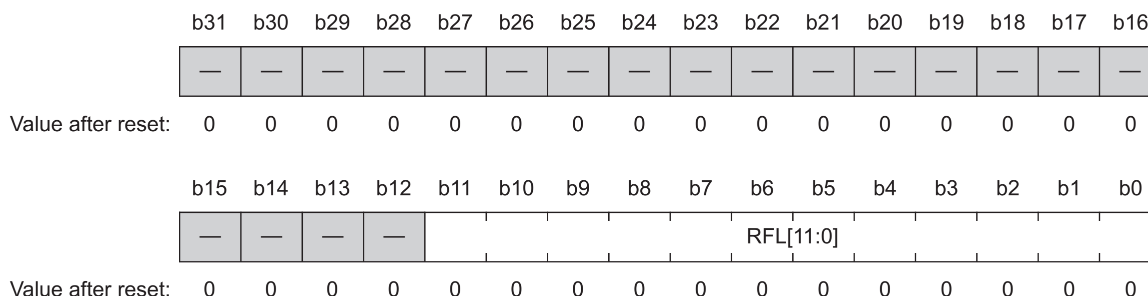
Receive Frame Length Register (RFLR): The maximum frame length which can be received by the RX63N can be set in the Receive Frame length Register. The RFLR should only be set when receive function is disabled.

RFL[11:0] Bits (Receive Frame Length 11 to 0): These bits denote the value for the maximum frame length in bytes. If the received data were to exceed this value it would result in a frame length error and the part of data that exceeds the specified length would be discarded.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 220

Receive Frame Length Register (RFLR)

Address(es): 000C 0108h



BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b11 to b0	RFL[11:0]	Receive Frame Length 11 to 0	000h to 5EEh: 1,518 bytes	R/W
			5EFh: 1,519 bytes	
			5F0h: 1,520 bytes	
			: :	
			7FFh: 2,047 bytes	
			800h to FFFh: 2,048 bytes	
b31 to b12	—	Reserved	These bits are always read as 0. The write value should always be 0.	R/W

Figure 9.8 Receive Length Frame Register [1], page 1115.

Ethernet Controller Status Register (ECSR): The ECSR register represents the status in the Ethernet controller. Each state is notified to the CPU by interrupts, and the EESR.ESI bit in the EESR register (ETHERC/EDMAC status register) of the EDMAC indicates which interrupt has been generated. These interrupt sources can be enabled or disabled in the ETHERC Interrupt Permission Register (ECSIPR).

ICD Bit (Illegal Carrier Detection): This bit is set to 1 if the PHY-LSI detects an illegal carrier on the line. In such a case the PHY-LSI sends a signal to the RX63N which results in the setting of this bit.

LCHNG Bit (Link Signal Change): This bit indicates that the ET_LINKSTA signal input from the PHY-LSI has changed from high to low or low to high. The current Link state is indicated by the LNKSTA pin status bit (LMON) in the PHY status register (PSR).

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 221

PSRTO Bit (PAUSE Frame Retransmit Retry Over): This bit indicates whether the retransmit count for retransmitting a PAUSE frame when flow control is enabled has exceeded the retransmit upper limit set in the automatic PAUSE frame retransmit count register (TPAUSER).

ETHERC Status Register (ECSR)

Address(es): 000C 0110h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	—	—	—	—	—	—	BFR	PSRTO	—	LCHNG	MPD	ICD
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	ICD	Illegal Carrier Detection	0: PHY-LSI has not detected an illegal carrier on the line 1: PHY-LSI has detected an illegal carrier on the line	R/W
b1	MPD	Magic Packet™ Detection	0: Magic Packet™ has not been detected 1: Magic Packet™ has been detected	R/W
b2	LCHNG	Link Signal Change	0: Change in the LINKSTA signal has not been detected 1: Change in the LINKSTA signal has been detected (high to low or low to high)	R/W
b3	—	Reserved	This bit is always read as 0. The write value should always be 0.	R/W
b4	PSRTO	PAUSE Frame Retransmit Retry Over	0: PAUSE frame retransmit count has not exceeded the upper limit 1: PAUSE frame retransmit count has exceeded the upper limit	R/W
b5	BFR	Continuous Broadcast Frame Reception	0: Continuous reception of broadcast frames has not been detected. 1: Continuous reception of broadcast frames has been detected.	R/W
b31 to b6	—	Reserved	These bits are always read as 0. The write value should always be 0.	R/W

Figure 9.9 Ethernet Controller Status Register [1], page 1116.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 222

9.4 ETHERNET DIRECT MEMORY ACCESS CONTROLLER

The RX63N Group has an on-chip direct memory access controller (EDMAC) directly connected to the Ethernet controller (ETHERC).

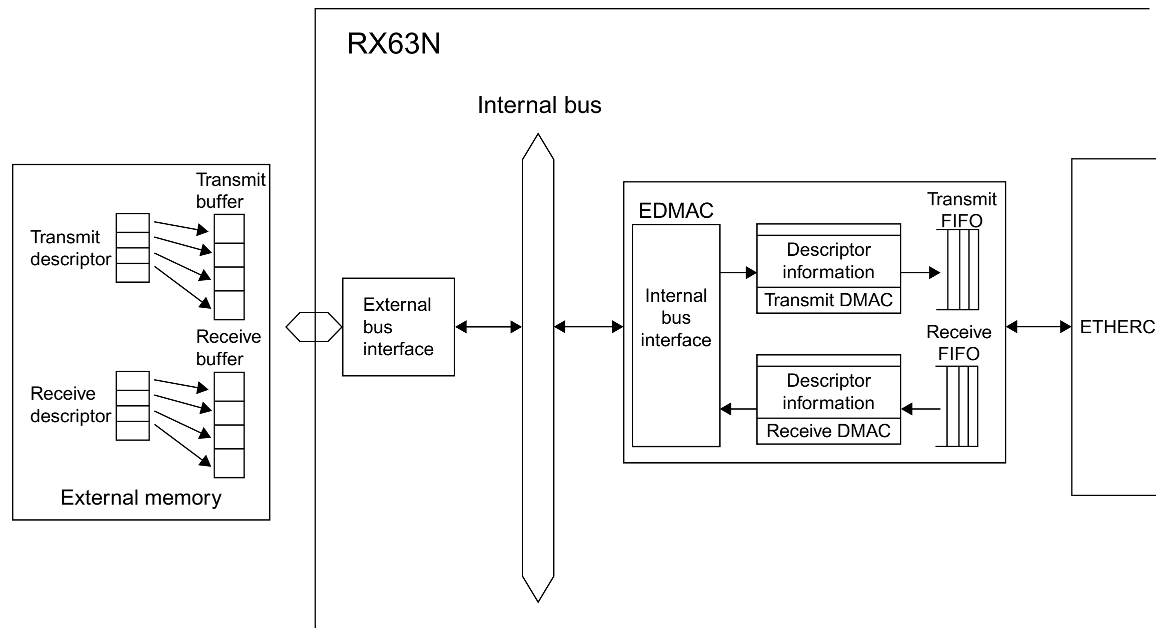


Figure 9.10 Configuration of EDMAC Buffers and Descriptors [1], page 1143.

This reduces the load on the CPU, thus enabling efficient data transmission and reception. The EDMAC controls most of the buffer management by using descriptors. The EDMAC reads the descriptors, which holds the control information. The descriptors corresponding to each buffer hold the buffer pointers and other information. The EDMAC reads transmit data from the transmit buffer and writes receive data to the receive buffer according to the control information. By arranging such multiple descriptors continuously (i.e., making a descriptor list), data can be transmitted or received sequentially.

A transmit descriptor list and a receive descriptor list should be created in memory space by the communication program prior to transmission and reception. The start addresses of these lists should be set in the transmit descriptor list start address register and the receive descriptor list start address register. The start addresses of the descriptor lists should be placed on the address boundaries in accordance with the descriptor length specified in the EDMAC mode register (EDMR). Here, the start address of the transmit buffer can be placed on a longword, word, or byte boundary.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 223

EDMAC Mode Register (EDMR): The operating mode of the EDMAC can be set in the EDMR register. Setting this register during transmission or reception might result in wrong values. Therefore, this register should be set after reset at initialization. While the transmission or reception function is enabled, it is prohibited from modifying the operating mode.

EDMAC Mode Register (EDMR)

Address(es): 000C 0000h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	—	—	—	—	—	DE	DL[1:0]		—	—	—	SWR
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	SWR	Software Reset	[Writing]	R/W
			0: Disabled	
			1: Internal hardware is reset*[1]	
			[Reading]	
			Read as 0.	
b3 to b1	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b5, b4	DL[1:0]	Transmit/Receive Descriptor Length	b5 b4	R/W
			0 0: 16 bytes	
			0 1: 32 bytes	
			1 0: 64 bytes	
			1 1: 16 bytes	
b6	DE	Big Endian Mode/ Little Endian Mode*[2]	0: Big endian mode (longword access)	R/W
			1: Little endian mode (longword access)	
b31 to b7	—	Reserved	These bits are read as 0. The write value should be 0.	R/W

Notes: 1. Registers other than TDLAR, RMFCR, TFUCR, and RFOCR are reset.
2. This setting is effective for received data and data for transmission. It does not apply to descriptors or registers (support is only for big endian).

Figure 9.11 EDMAC Mode Register [1], page 1144.

EDMAC Transmit Request Register (EDTRR): This register issues transmit directives to the EDMAC. After having transmitted one frame, the EDMAC reads the next descriptor. If the transmit descriptor active bit in this descriptor is set (active), the EDMAC continues transmission. Otherwise, the EDMAC clears the TR bit and stops the transmit DMAC operation.

EDMAC Transmit Request Register (EDTRR)

Address(es): 000C 0008h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	TR
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	TR	Transmit Request	0: Transmission-halted state Writing 0 does not stop transmission. Termination of transmission is controlled by the active bit of the transmit descriptor.	R/W
			1: Transmission start The EDMAC starts reading the target descriptor and sends a frame whose transmission active bit is set to 1.	
b31 to b1	—	Reserved	These bits are read as 0. The write value should be 0.	R/W

Figure 9.12 EDMAC Transmit Request Register [1], page 1145.

EDMAC Receive Request Register (EDRRR): This register is used to issue receive directives to the EDMAC. Setting the Receive Request bit of this register enables the receiving function, and the EDMAC reads the receive descriptor. After data has been received for the receive buffer size, the EDMAC reads the next receive descriptor and

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 225

becomes ready for frame reception. If the receive descriptor active bit of that receive descriptor is set to 0 (inactive), the EDMAC clears the Receive Request bit and stops the receive DMAC operation.

EDMAC Receive Request Register (EDRRR)

Address(es): 000C 0010h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	RR
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	RR	Receive Request	0: Receiving function is disabled*[1] 1: Receive descriptor is read, and the EDMAC becomes ready to receive	R/W
b31 to b1	—	Reserved	These bits are read as 0. The write value should be 0.	R/W

Note 1. If the receiving function is disabled during frame reception, write-back is not performed successfully to the receive descriptor. Following pointers to read a receive descriptor become abnormal and the EDMAC cannot operate successfully. In this case, to make EDMAC reception enabled again, execute a software reset by the EDMR.SWR bit.
To disable the EDMAC receiving function without executing a software reset, specify the ECMR.RE bit in the ETHERC. Next, after the EDMAC has completed the reception and write-back to the receive descriptor has been confirmed, disable the receiving function using EDRRR.

Figure 9.13 EDMAC Receive Request Register [1], page 1146.

Transmit Descriptor List Start Address Register (TDLAR): TDLAR specifies the start address of the transmit descriptor list. TDLAR must not be modified during transmission. The transmission halted state results in the EDTRR.TR bit being set to zero. In this case TDLAR can be updated to continue transmission. Depending on the descriptor length specified in EDMR.DL[1:0], the lower bits of the register are set to zero.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 226

Transmit Descriptor List Start Address Register (TDLAR)

Address(es): 000C 0018h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	TDLA[31:0]															
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	TDLA[31:0]															
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b31 to b0	TDLA[31:0]	Transmit Descriptor Start Address	16-byte boundary: TDLA[3:0] = 0000b	R/W
			32-byte boundary: TDLA[4:0] = 00000b	
			64-byte boundary: TDLA[5:0] = 000000b	

Figure 9.14 Transmit Descriptor List Start Address Register [1], page 1147.

Receive Descriptor List Start Address Register (RDLAR): The register specifies the start address of the receive descriptor list. The RDLAR must not be modified during reception. Depending on the Descriptor length specified in the EDMR.DL[1:0], the lower bits of the register are set to zero.

Receive Descriptor List Start Address Register (RDLAR)

Address(es): 000C 0020h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	RDLA[31:0]															
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	RDLA[31:0]															
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b31 to b0	RDLA[31:0]	Receive Descriptor Start Address	16-byte boundary: RDLA[3:0] = 0000b	R/W
			32-byte boundary: RDLA[4:0] = 00000b	
			64-byte boundary: RDLA[5:0] = 000000b	

Figure 9.15 Receive Descriptor List Start Address Register [1], page 1148.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 227

9.5 RENESAS ETHERNET DRIVER API

The Renesas Driver provides application functions for using the Ethernet module on the board. The zero copy Ethernet driver uses circular buffers to transmit and receive Ethernet Frames using the Ethernet Direct Memory Access Controller (EDMAC) frame reception and transmission complete interrupts. These functions have an additional suffix _ ZC. Functions also do the same job without using circular buffers. Let us see the functions available.

R_Ether_Open_ZC :

Format

```
int32_t R_Ether_Open_ZC(uint32_t ch, uint8_t mac_addr[], void **buf);
```

Parameters

ch—Specifies the Ethernet Controller channel number.
mac_addr—Specifies the MAC address of EtherC.
buf—Points to the buffer pointer used by the stack.

Return Values

```
R_ETHER_OK(0) , R_ETHER_ERROR(-1)
```

This function initializes the Ethernet controller and the direct memory access controller subsystems. The Ethernet Direct Memory Access Controller descriptors and buffers are set up for initial use. The MAC address is used to initialize the MAC address registers in the Ethernet Controller. The pointer to buffer pointer is initialized with the first available transmit buffer. This provides a data buffer to the stack for transmitting data. By default, the physical device is configured to auto-negotiate mode. If there is only one Ethernet channel, then the channel number is set to zero. If there are two Ethernet channels, then 0 and 1 are used as channel numbers.

R_Ether_Close_ZC :

Format

```
int32_t R_Ether_Close_ZC(uint32_t ch);
```

Parameters

ch—Specifies the Ethernet Controller channel number.

Return Values

```
R_ETHER_OK(0) , R_ETHER_ERROR(-1)
```

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 228

This function disables transmit and receive functionality of the Ethernet Controller. If there is only one Ethernet channel, then the channel number is set to zero. If there are two Ethernet channels, then 0 and 1 are used as channel numbers.

R_Ether_Read_ZC:

Format

```
int32_t R_Ether_Read_ZC(uint32_t ch, void **buf);
```

Parameters

ch—Specifies the EtherC channel number.
buf—Points to the buffer pointer used by the stack.

Return Values

Returns the number of bytes received. A zero value indicates no data is received.

This function receives data into the application receive buffer. The driver's buffer pointer of the read data is returned in the parameter buffer. Returning the pointer allows the operation to be performed with zero-copy.

The return value shows the number of received bytes. If no data is available at the time of the call, a zero value is returned. The direct memory access hardware operates independent of the function and reads data off the Ethernet link into a buffer pointed to by the receive descriptor. It updates the status of the receive descriptor as new data is processed.

The buffer pointed to by the Ethernet Direct Memory Access Controller-Receive Descriptor is statically allocated by the driver. If there is only one Ethernet channel, then the channel number is set to zero. If there are two Ethernet channels, then 0 and 1 are used for the channel numbers.

R_Ether_Write_ZC:

Format

```
int32_t R_Ether_Write_ZC(uint32_t ch, void **buf, uint32_t len);
```

Parameters

ch—Specifies the EtherC channel number.
buf—Points to the buffer pointer used by the stack.
len—Ethernet frame length.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 229

Return Values

R_ETHER_OK (0)
R_ETHER_ERROR (−1)

The function transmits data from the application transmit buffer. It moves transmit data to a buffer pointed to by the transmit E-DMAC descriptor. It updates the status of the transmit descriptor as new data is processed. Data written is transmitted by the Controller. If there is only one Ethernet channel, then the channel number is set to zero. If there are two Ethernet channels, then 0 and 1 are used for the channel numbers.

9.5.1 Example 1: Transmitting Ethernet Frames

Let us see an example of transmitting data using the Renesas Ethernet Driver functions. The structure s_frame defines an Ethernet frame which contains information which is to be used, such as the Destination MAC address protocol. The program transmits ten frames on channel zero. Initial settings are done by R_Ether_Open(). If this function does not return an error, then the ten frames are transmitted in the for loop using R_Ether_Write(). Finally when the transmission completes R_Ether_Close() disables the controller.

```
 1. #include "iodefine.h"
 2. #include "r_ether.h"
 3. #pragma section ETH_BUFF
 4. typedef struct{
 5.    uint8_t frame[BUFSIZE];
 6.    int32_t len;
 7.    uint8_t wk[12];
 8. } USER_BUFFER;
 9. USER_BUFFER recv[10];
10. #pragma section
11. extern void main(void);
12. static uint8_t s_frame[] = {
13.    0xff,0xff,0xff,0xff,0xff,0xff,   //Destination MAC address
14.    0x00,0x01,0x02,0x03,0x04,0x05,   //Source MAC address
                                          (00:01:02:03:04:05)
15.    0x08,0x06,                       //Type (ARP)
16.    0x00,0x01,                       //+—H/W type = Ethernet
17.    0x08,0x00,                       //+—Protocol type = IP
18.    0x06,0x04,                       //+—HW/protocol address length
19.    0x00,0x01,                       //+—OPCODE = request
```

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 230


```
20.    0x00,0x01,0x02,0x03,0x04,0x05,   //+—Source MAC address
                                          (00:01:02:03:04:05)
21.    0xc0,0xa8,0x00,0x03,             //+—Source IP address
                                            (192.168.0.3)
22.    0x00,0x00,0x00,0x00,0x00,0x00,   //+—Inquiry MAC address
23.    0xc0,0xa8,0x00,0x05,             //+—Inquiry IP
                                          address(192.168.0.5)
24. };
25. static uint8_t mac_addr[6] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05};
26. void SampleEthernetTransmission(void);
27. void SampleEthernetReception(void);
28. void main(void) {
29.    SampleEthernetTransmission();
30.    while(1);
31. }
32. void SampleEthernetTransmission(void) {
33.    int32_t i;
34.    int32_t ret;
35.    //==== Ethernet initial setting ====
36.    ret = R_Ether_Open(0, mac_addr);
37.    if( ret == R_ETHER_OK ) {
38.       //==== 10-frame transmission start ====
39.       for( i=0; i<10; i++ ) {
40.          //——Transmission ——
41.          ret = R_Ether_Write( 0, s_frame, sizeof(s_frame) );
42.          if( ret != R_ETHER_OK ) {
43.             break;
44.          }
45.       }
46.    }
47.    //Check transmission completion
48.    while(EDMAC.EDTRR.BIT.TR != 0);
49.    //==== Ethernet transmission and reception stop ====
50.    R_Ether_Close(0);
51. }
```

9.5.2 Example 2: Receiving Ethernet Frames

Similar to the function for transmitting Ethernet frames, the SampleEthernetReception() function receives the ten Ethernet frames. The Ethernet Modules must be initialized by

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 231

R_Ether_Open(). If successfully initialized, the for loop receives ten frames using R_Ether_Read(), and finally R_Ether_Close is used to disable the Ethernet Module.

```
 1. void SampleEthernetReception(void) {
 2.    int32_t i;
 3.    int32_t ret;
 4.    //==== Ethernet initial setting ====
 5.    ret = R_Ether_Open(0, mac_addr);
 6.    if( ret == R_ETHER_OK ) {
 7.       //==== Start reception of 10 frames ====
 8.       for( i=0; i<10; i++ ) {
 9.          //-- Reception --
10.          recv[i].len = R_Ether_Read(0, recv[i].frame);
11.          if( recv[i].len == 0 ) {
12.             i--;
12.          }
13.       }
14.    }
15.    //==== Ethernet transmission/reception halted ====
16.    R_Ether_Close(0);
17. }
```

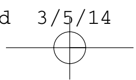
9.6 RECAP

Ethernet is a standard for networking technologies defined as IEEE 802.3. Ethernet networks can be configured in a point to point, bus, or star topology. The star topology, which requires a network switch in the center of the network, is the prevalent method. The RX63N comes with an 802.3x compliant Ethernet MAC capable of 10/100 Mbps as well as an Ethernet DMA controller. The EDMAC controls most of the buffer management by using descriptors. A transmit descriptor list and a receive descriptor list should be created in memory space by the communication program prior to transmission and reception. The Renesas Driver for the Ethernet Peripheral provides application functions for using the Ethernet module on the board. These functions are used either by an application written by the user or by another layer of abstraction called a stack.

9.7 REFERENCES

[1] Renesas Electronics, Inc. (February, 2013). *RX63N Group, RX631 Group User's Manual: Hardware,* Rev 1.60.

09.ES_Conrad_RX63N_Advanced_CH09.qxd:RX63N Advanced 3/5/14 7:52 AM Page 232



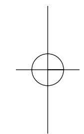
[2] http://docwiki.cisco.com/wiki/Ethernet_Technologies#Ethernet_Network_Elements

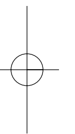
[3] Renesas Electronics, Inc. (June, 2012). *Application note: RX63N Group, Zero-Copy Ethernet Driver Demonstration,* Rev 1.0.

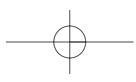
[4] Adam Dunkels, The uIP Embedded TCP/IP Stack, The uIP 1.0 Reference Manual, Swedish Institute of Computer Science, June 2006.

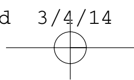
9.8 EXERCISES

1. How do you identify the MAC address of your RX63N board?
2. Write the code to read the MAC address of your RX63N board and assign a default address if the original is not appropriate.
3. How do you identify the IP address of your RX63N board? What steps are followed to assign an IP address to the board?
4. How to initialize the Ethernet transmission/reception on RX63N?
5. Using the code from the examples, write the additional code needed to transmit a 100 Kbyte file to the receiving device at IP address (192.168.0.10).









CAN Bus

10.1 LEARNING OBJECTIVES

Controller Area Network (CAN) is an asynchronous serial communication protocol which follows ISO 11898 standards and is widely accepted in automobiles due to its real time performance, reliability, and compatibility with a wide range of devices. CAN is a two wire differential bus with data rates of up to 1 Mbps. Its robust, low cost, and versatile technology make CAN a good choice in other applications where inter-processor communication or elimination of excessive wiring is needed. Some of the areas it is widely used are industrial machinery, avionics, medical equipment, and home automation.

In this chapter the reader will learn:

- What is a CAN bus.
- Why use a CAN bus.
- Transmitting and receiving using a CAN bus.

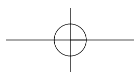
10.2 THEORY OF CAN PROTOCOL

Controller Area Network is a serial communication protocol which is mainly used for reducing wired interconnections in a vehicle. Some of the benefits in implementing the CAN protocol in automobiles are:

- Reduced wired interconnections
- Low cost implementation
- Speed, reliability, and error resistance
- Worldwide acceptance

The main characteristics of the CAN protocol are:

- Multi master hierarchy
- Priority based bus access



- Baud rate up to 1 Mbps/sec
- Error detection and fault confinement

An embedded systems designer needs to carefully analyze and assess technologies when building a system. The CAN bus meets many important characteristics of a simple yet dependable communication protocol for many embedded system applications:

- Real Time Control: The CAN bus can be used in hard real-time systems since it has low latency and a built-in priority structure.
- Reliability/Robustness: The bus is robust since it uses differential signaling, which amounts to built-in noise immunity. It also supports error recovery since the protocol implements retransmission if error checking detects a transmission was not successful.
- Flexibility of messaging: Many different devices can be attached to the bus, and these devices can use peer to peer or multicast messaging.
- Simplicity: It is easy to add new nodes to the CAN bus and is easy to program if you use an API.
- Economy: The bus wiring cost is low because it consists of only two wires that can be several meters long. The interface hardware cost is low and is supported by many vendors. Since the hardware handles the low level protocol, no software is needed for lower level data communication.
- Scalability: It is easy to expand a network—simply attach a node using two wire connections. Nodes can even be added while the bus is active.
- Availability: There are many microcontrollers and interface parts that support CAN, and many APIs and debugging tools readily available.

10.2.2 CAN Bus Details

A CAN bus is a half-duplex, two wire differential bus. The two lines, CAN_L and CAN_H, form the communication bus for the nodes to transmit data or information. The logic levels used on the bus are dominant and recessive levels, where dominant level is referred when TTL = 0V and recessive level is referred when TTL = 5V. The dominant level always overrides the recessive level and this concept is used to implement the bus arbitration.

The voltage levels on the CAN bus varies from 1.5 volts to 3.5 volts. The logic levels are calculated as the voltage difference between the two lines.

$$V_{diff} = V_{CAN_H} - V_{CAN_L} \qquad \textbf{Formula 10.1}$$

If the difference voltage (V_{diff}) is 2 volts, it is considered as a dominant level and if it is 0 volts, it is considered as a recessive level.

In the CAN protocol, nodes communicate data or information through messages termed as frames. A frame is transmitted on to the bus only when the bus is in an idle state. There are four different types of frames which are used for communication over the CAN bus.

- Data Frame—Used to send data
- Remote Frame—Used to request data
- Error Frame—Used to report an error condition
- Overload Frame—Used to request a delay between two data or remote frames

The first two frames, Data and Remote, are generated by the user (i.e., when they use an API). The other two are typically generated by an underlying system and are not user generated.

The frames transmitted from one node will be received by all the other nodes on the network using message broadcasting. The user sets up what frames are to be received and sent. Message filtering, which is then provided automatically by the CAN controller hardware, decides whether the received frame is relevant to that node or not. If any error occurs due to reception or transmission, an error frame will be transmitted on the bus to let the network know of the error. Since the error frame starts with six dominant bits, it will have the highest priority when the bus is idle. As soon as the error is detected, the CAN protocol implements the fault confinement techniques to overcome the error. The fault confinement feature in the CAN protocol differentiates between a temporary error and a permanent failure of a node. If the error is due to permanent failure of the node, it automatically detaches the defective node from the bus without causing any problems to the network.

10.2.3 Different CAN Bus Standards

There are several CAN physical layer and other standards:

- **ISO 11898–1:** CAN Data Link Layer and Physical signaling.
- **ISO 11898–2:** CAN High-Speed Medium Access Unit: ISO 11898–2 uses a two-wire balanced signaling scheme. It is the most used physical layer in car power-train applications and industrial control networks.
- **ISO 11898–3:** CAN Low-Speed, Fault-Tolerant, Medium-Dependent Interface.
- **ISO 11898–4:** CAN Time-Triggered Communication: ISO 11898–4 standard defines the time-triggered communication on CAN (TTCAN). It is based on the CAN data link layer protocol providing a system clock for the scheduling of messages.
- **ISO 11898–5:** CAN High-Speed Medium Access Unit with Low-Power Mode.
- **ISO 11898–6:** CAN High-Speed Medium access unit with selective wake-up functionality.
- **ISO 11992–1:** CAN fault-tolerant for truck/trailer communication.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 236

- **ISO 11783–2:** 250 kb/s, Agricultural Standard: ISO 11783–2 uses four unshielded twisted wires; two for CAN and two for terminating bias circuit (TBC) power and ground. This bus is used on agricultural tractors. This bus is intended to provide inter-connectivity with any implementation adhering to the standard.

10.2.4 Types of Frames and Their Architectures

As mentioned earlier, a CAN provides four different types of message frames for communication. The architecture of each frame is discussed in this section.

Data and Remote Frame

Data and Remote frames are user requested (e.g., by an API).The architecture of the data and the remote frame are exactly the same. A data frame has higher priority than a remote frame. Each data and remote frame starts with a Start of Frame (SOF) field and ends with an End of Frame (EOF) field. Figure 10.1 shows the architecture of data and remote frames.

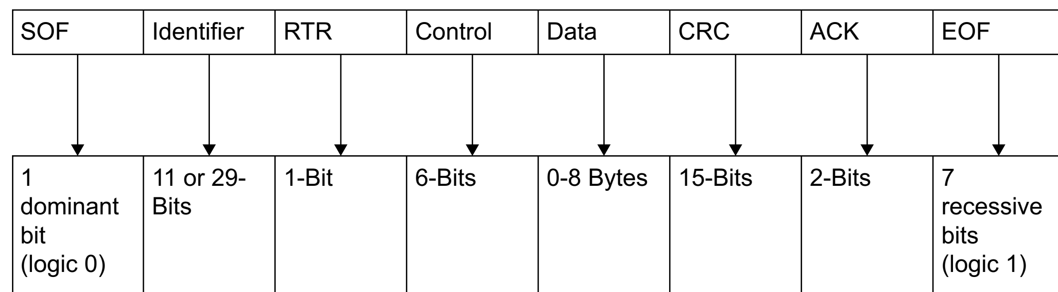


Figure 10.1 Architecture of data and remote frame.

The following are the fields in data and remote frame:

- SOF field (1 bit)—Indicates the beginning of the frame. A single dominant bit represents a start of a frame. It is also used for data transfer synchronization.
- Arbitration Field—This contains two sub fields, Message Identifier and RTR field.
    - Message Identifier (11/29 bits)—This field contains a message ID for each frame which is either 11 (Standard ID) or 29 (Extended ID) bits. No two message frames in the CAN network should have the same message ID. A message ID which has a low decimal value is considered as a high priority message.
    - Remote Transmission Request (RTR) (1 bit)—The RTR field distinguishes a data frame from a remote frame.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 237

- Control Field (6 bits)—This contains two sub fields, the IDE and the DLC field, as well as a reserved bit.
  - Identifier Extension (IDE) bit (1 bit)—This bit indicates the format of the message ID in the frame, either a standard 11-bit format or extended 29-bit format.
  - Reserved (1 bit).
  - Data Length Code (DLC) field (4 bits)—This field is used to set the amount of data being transferred from one node to another node. In a remote frame, these bits represent the amount of data it is requesting.
- Data Field—This field contains the actual data and it is not applicable for a remote frame.
- CRC field (16 bits)—The CRC field consists of the CRC Sequence and a CRC Delimiter bit.
  - CRC Sequence field (15 bits)—This 15-bit field contains the frame check sequence without the stuffing bits.
  - CRC Delimiter bit (1 bit)—This bit is used to provide processing time for the CRC Sequence field.
- ACK field (2 bits)—The ACK field consists of a 1-bit Acknowledgement Slot field and an Acknowledgment Delimiter bit (which is always recessive).
- EOF field (7 bits)—Indicates the end of the frame. A 7-bit continuous recessive bit represents the end of frame.

A node uses a data frame to transmit data to any other node on the network. The RTR field determines whether the message frame should act as a data frame or a remote frame. When the RTR bit is set to dominant level, then the message frame will act as a data frame. A maximum of eight bytes of data can be transferred using a single data frame. Each data frame will be assigned a unique message ID during which the node decides whether the data is relevant or not.

A remote frame is used to request a data frame from any node on the network. When the RTR bit is set to a recessive level, then the message frame will act as a remote frame. While requesting data from a node, the length of the data field in the control field (DLC bits) of the remote frame should be the same as the requesting data frame, otherwise a bus collision occurs. As soon as the remote frame is accepted by a node, a data frame will be transmitted on to the bus with the requested data. When two or more nodes on the network request the same message at the same time, a bus collision occurs.

Error Frame

An error frame is transmitted onto the bus whenever a transmission or reception error occurs due to a faulty node or bus problems. An error frame consists of three fields:

- Error flag (6 bits)—It is six dominant bits which indicates the transmitting or receiving error on the bus.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 238

- Error Delimiter (8 bits)—It is represented as a sequence of eight recessive bits. After transmitting the error flag each node transmits a single recessive bit and waits for the bus level to change to recessive. Only after the bus level is recessive, the remaining seven recessive bits will be sent onto the bus.
- Interframe Space (3 bits)—It is represented as the minimum space between any type (data, remote, error, overload) of frame and a following data or remote frame. It contains three recessive bits.

The error delimiter and the interframe space are used to synchronize the nodes to the error frame transmitted onto the bus.

Overload Frame

The overload frame takes the same form as the error frame but the overload frame is used to request a delay between the transmission of the next data or remote frame. It consists of two fields, an Overload flag and an Overload delimiter.

- Overload flag (6 bits)—It contains six dominant bits which indicates the transmitting or receiving error on the bus.
- Overload Delimiter (8 bits)—It is represented as a sequence of eight recessive bits. After transmitting the error flag each node transmits a single recessive bit and waits for the bus level to change to recessive. Only after the bus level is recessive will the remaining seven recessive bits be sent onto the bus.
- Interframe Space (3 bits)—It is represented as minimum space between frames of any type (data, remote, error, overload) and a following data or remote frame. It transmits three consecutive recessive bits on to the bus. When the Interframe space is being transmitted, no node on the network is allowed to transmit any of the frames except the overload frame.

10.2.5 Bus Arbitration

In a single bus communication protocol, when two or more nodes request access to the bus then a bus arbitration technique must be implemented. Usually bus access will be given to a node with a high priority. The bus arbitration technique also avoids data collisions as long as no two nodes transmit the same ID.

The CAN protocol provides a non-destructive bus arbitration mechanism. It assigns a recessive level to the bus only if all the nodes on the bus output a recessive level and it assigns a dominant level if any one of the nodes on the network output a dominant level. When a dominant bit and a recessive bit request access for the bus, the dominant bit is given the access as it is considered as the high priority. The bus arbitration on a CAN net-

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 239

work follows an AND gate logic as shown in Table 10.1. Note that there is no loss of time during arbitration—the dominant address just over runs the higher address.

TABLE 10.1 Bus Arbitration on CAN Bus.

NODE 1	NODE 2	BUS LOGIC LEVEL
Dominant	Dominant	Dominant
Dominant	Recessive	Dominant
Recessive	Dominant	Dominant
Recessive	Recessive	Recessive

When transmitting a frame onto the bus, bus access to a node will be given based upon the message ID of the frame. As the dominant level is considered high priority, the message ID with more dominant bits is considered a high priority message. When the bus is idle, the bus access will be assigned to the node which transmits a message with a higher priority.

10.2.6 Message Broadcasting

The CAN protocol is based on a message broadcasting mechanism in which the frames transmitted from one node are received by every node on the network. The receiving nodes will only react to the data that is relevant to them. Messages in the CAN are not acknowledged due to an unnecessary increase of traffic. However, the receiving node checks for the frame consistency and acknowledges the consistency. If the acknowledge is not received from any or all the nodes of the network, the transmitting node posts an error message to the bus. If any of the nodes are unable to decode the transmitted message due to internal malfunction or any other problem, the entire bus will be notified of the error and the node re-transmits the frame. If there is an internal malfunction in a node, that particular node reports an error for each frame it receives. Due to this, most bandwidth of the network will be allocated to error frames as they have higher priority (starts with six consecutive dominant bits). To overcome this problem the CAN protocol supports a bus off state in a node, in which the node will be detached from the bus if it reports an error for more than a predefined value. The bus off state of a node is implemented to avoid the breakdown of the network due to a single node.

While broadcasting data frames on the bus, each node on the bus receives every data frame transmitted on to the bus. As the CAN protocol does not support IDs for the nodes and the receiver does not know the information of the transmitter of the frame, each data

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 240

frame goes through an acceptance filtering process at the receiving node, which is dependent on the message ID (standard or extended) of the frame.

The process of data requesting in the CAN protocol is carried out by the remote frame. The RTR bit in a frame decides whether the frame is a remote frame or data frame. When the RTR bit is set to recessive level the frame will act as a remote frame. When a node is requesting a data frame from another node, the message identifier section (ID bits) and the data length section (DLC bits) in the remote frame should be of the same value as that in the data frame that is requested, otherwise an error will be reported on the bus.

10.2.7 Data Transfer Synchronization

Each node in the CAN network will have different oscillators running at slightly different frequencies (0.5 percent crystal accuracy), so to make all the nodes work synchronously while transferring data the CAN protocol uses the falling edge of the SOF bit (transition from recessive to dominant bus level).

The bit coding used in the CAN bus is a Non-Return-to-Zero principle in which the bit level remains constant during the entire bit time, which creates a node synchronization problem during the transmission of larger bit blocks of same polarity. To overcome this problem the CAN protocol uses a bit stuffing mechanism.

Bit Stuffing: The CAN protocol allows only five consecutive bits of the same polarity between the SOF bit and the Data field. If more than five consecutive bits of the same polarity are transmitted on to the bus it will be considered as an error condition. So to transmit data with more than five consecutive bits of the same polarity the CAN protocol inserts a complementary bit of opposite polarity at the transmitter end, and at the receiver end the filtering should be performed to get rid of the stuffed bit. Bit stuffing is applied only in data and remote frames and it is not applicable after CRC field.

10.2.8 Error Detection and Fault Confinement

The CAN protocol implements a series of error detection mechanisms which contributes to the high level of reliability and error resistance. It also implements fault confinement mechanisms for proper function of the network. The error detection mechanisms implemented by the hardware are:

- Bit monitoring—Transmitter compares each bit that is transmitted onto the bus with the data it is transmitting and reports an error if there is a change in the data transmitted.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 241

- Checksum check—Every data and remote frame has a 15-bit CRC field which carries the checksum of the frame and is used to detect errors at the receiver.
- Bit stuffing—The CAN protocol allows only five consecutive bits of the same polarity. Bit stuffing is implemented during transmission of more than five consecutive bits of the same polarity. If more than five consecutive bits of the same polarity are transmitted, the bus takes it as an error frame as the first six bits of the error frame are dominant bits.
- Frame check—Each transmitting and receiving node checks for the consistency of the frames.
- Acknowledge check—Each receiving node transmits a frame consistency acknowledge to the transmitting node.

Whenever an error occurs on the bus, each node on the network receives the error frame and the transmitting node serves the error by re-transmitting the frame.

The CAN protocol implements fault confinement techniques to ensure that the communication on the network never fails. Consider a situation in which a node has an internal malfunction due to electrical disturbances and transmits an error frame for every frame it receives. For serving these kind of errors, the CAN protocol is supplied with two counters, a transmit error counter and a receive error counter. The corresponding counter is incremented each time a failure in the transmission/reception occurs. The counter is decremented whenever there is a successful transmission/reception. The counter value does not decrement when the value is zero. Based upon the values of the two counters the CAN nodes will have three states: Error active, Error passive, and Bus off state.

- Error active state—Every node after reset starts with this state in which it transmits an error flag (six consecutive dominant bits) whenever it receives an error frame.
- Error passive state—A node enters into this state when a receive error counter or transmit error counter value is equal to or greater than 127. When the node is in error passive state, it transmits an error flag with six consecutive recessive bits.
- Bus off state—A node enters into this state when a transmit error counter value increases more than 255.

To confirm the CAN error state, either poll the CAN status register, or use the CAN error interrupt. If Error Passive is reached, notify user with, for example, LED. If Bus Off is reached, stop the application from communicating, and wait until the Error Active state is reached. When (or if) this is detected, reinitialize CAN from scratch and restart the CAN application. As long as you do not go to the Bus Off state, you can and should communicate. This is one of the benefits of using CAN—it is robust with error recover

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 242

mechanisms. The simplest way to check for Bus Off is to poll once every main program loop.

Should a node go into the Bus Off state, use the CAN error interrupt or poll with the Check Error function once every cycle in the main routine check the node state. If the node has reached Bus Off state a certain number of times within a certain time period, you may want to send a warning message to the user (i.e., light an LED). If the Bus Off state is reached, stop communication and continue polling to see when the peripheral has returned to the normal Error Active state. When the node has recovered, it is important to reinitialize the CAN peripheral and the application to make sure the slots are in a known state.

10.2.9 Different CAN Bus Standards

There are several CAN physical layer and other standards:

- **ISO 11898–1:** CAN Data Link Layer and Physical signaling.
- **ISO 11898–2:** CAN High-Speed Medium Access Unit: ISO 11898–2 uses a two-wire balanced signaling scheme. It is the most used physical layer in car power-train applications and industrial control networks.
- **ISO 11898–3:** CAN Low-Speed, Fault-Tolerant, Medium-Dependent Interface.
- **ISO 11898–4:** CAN Time-Triggered Communication: ISO 11898–4 standard defines the time-triggered communication on CAN (TTCAN). It is based on the CAN data link layer protocol providing a system clock for the scheduling of messages.
- **ISO 11898–5:** CAN High-Speed Medium Access Unit with Low-Power Mode.
- **ISO 11898–6:** CAN High-Speed Medium access unit with selective wake-up functionality.
- **ISO 11992–1:** CAN fault-tolerant for truck/trailer communication.
- **ISO 11783–2:** 250 kb/s, Agricultural Standard: ISO 11783–2 uses four unshielded twisted wires; two for CAN and two for terminating bias circuit (TBC) power and ground. This bus is used on agricultural tractors. This bus is intended to provide inter-connectivity with any implementation adhering to the standard.

10.3 BASIC CONCEPTS

The RX63N/RX631 Group implements three channels of the CAN (Controller Area Network) module that complies with the ***ISO11898–1*** Specifications. The CAN module

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 243

transmits and receives both formats of messages, namely the standard identifier (11 bits) (identifier is hereafter referred to as ID) and extended ID (29 bits).

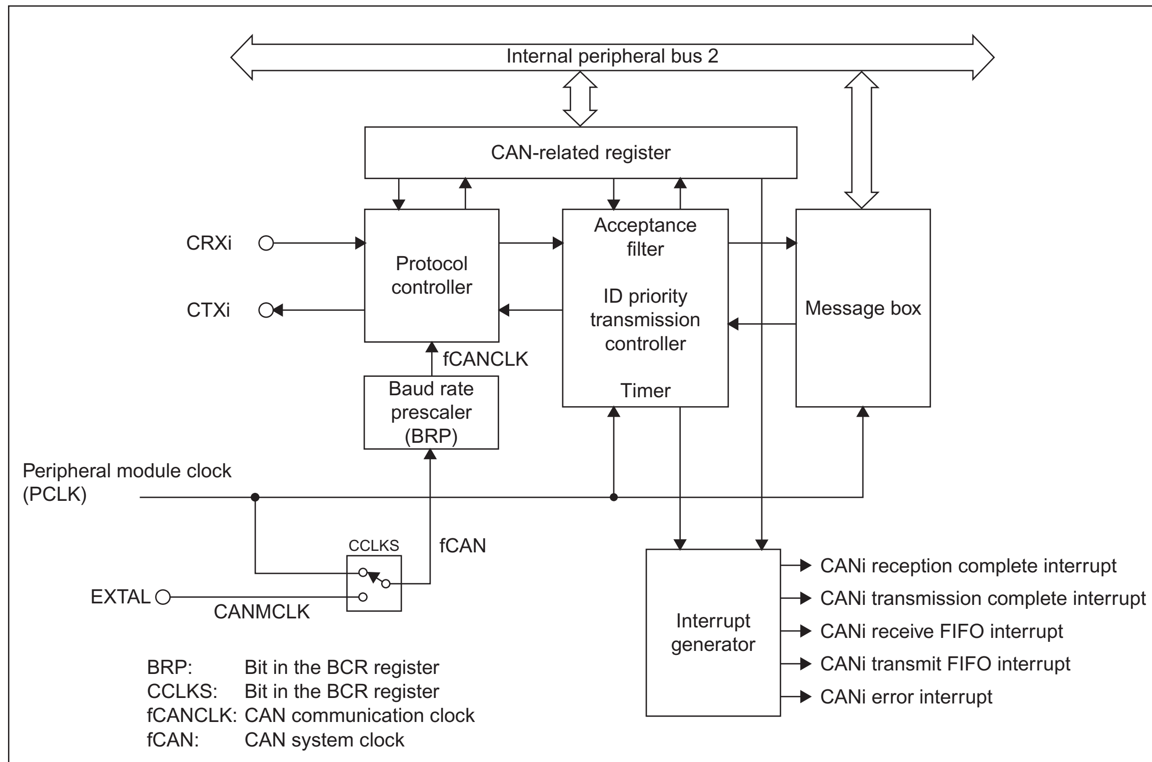


Figure 10.2 Block Diagram of CAN Module ($i = 0$ to 2) [1], page 1477.

- *CRXi and CTXi (i = 0 to 2):* CAN input and output pins.
- *Protocol controller:* Handles CAN protocol processing such as bus arbitration, bit timing at transmission and reception, stuffing, and error handling.
- *Message box:* Consists of thirty-two mailboxes which can be configured as either transmit or receive mailboxes. A unique individual ID, a data length code, a data field (8 bytes), and a time stamp is provided for each mailbox.
- *Acceptance filter:* Performs filtering of received messages. MKR0 to MKR7 are used for the filtering process.
- *Timer:* Used for the time stamp function. The timer value when a message is stored into the mailbox is written as the time stamp value.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 244

- *Interrupt generator:* It generates the following interrupts according to the condition met:

  CANi reception complete interrupt
  CANi transmission complete interrupt
  CANi receive FIFO interrupt
  CANi transmit FIFO interrupt
  CANi error interrupt

TABLE 10.2 Pin Configuration [1], page 1478.

PIN NAME	I/O	FUNCTION
CRX0	Input	Pin for receiving data
CTX0	Output	Pin for transmitting data
CRX1	Input	Pin for receiving data
CTX1	Output	Pin for transmitting data
CRX2	Input	Pin for receiving data
CTX2	Output	Pin for transmitting data

10.3.1 Registers

There are thirty-two mailboxes per channel, along with setup registers in the RX63N processor for configuring the CAN bus. These mailboxes can be operated in either "Normal mode" or "First in First out (FIFO) mode" by setting up the corresponding registers.

- Normal mailbox mode: In which all thirty-two mailboxes can be configured to either transmission or reception mailboxes. Beginners should use this mode until they become more familiar with CAN bus use.
- FIFO mailbox mode: In which twenty-four mailboxes can be configured to either transmission or reception mailboxes. In the remaining eight mailboxes, the first four mailboxes can be configured as FIFO transmission and the other four mailboxes are configured as FIFO reception mailboxes.

All data to be transmitted are first stored in the transmission mailbox and all data received is stored in the reception mailbox. All thirty-two mailboxes can be used for either trans-

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 245

mission or reception. A status register is available in the RX63N which records the status of all the events that occur in a particular node.

Control Register (CTLR)

Control Register (CTLR)

Address(es): CAN0.CTLR 0009 0840h, CAN1.CTLR 0009 1840h, CAN2.CTLR 0009 2840h

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	RBOC	BOM[1:0]		SLPM	CANM[1:0]		TSPS[1:0]		TSRC	TPM	MLM	IDFM[1:0]		MBM
Value after reset:	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0

Figure 10.3 CAN control register [1], page 1481.

For information on various fields and register functions refer to [1], page 1481.

Bit Configuration Register (BCR)

Bit Configuration Register (BCR)

Address(es): CAN0.BCR 0009 0844h, CAN1.BCR 0009 1844h, CAN2.BCR 0009 2844h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	TSEG1[3:0]				—	—	BRP[9:0]									
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	SJW[1:0]		—	TSEG2[2:0]			—	—	—	—	—	—	—	CCLKS
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 10.4 Bit Configuration Register (BCR) [1], page 1485.

For information on various fields and register functions refer to [1], page 1485.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 246

Mask Register k (MKRk) (k = 0 to 7)

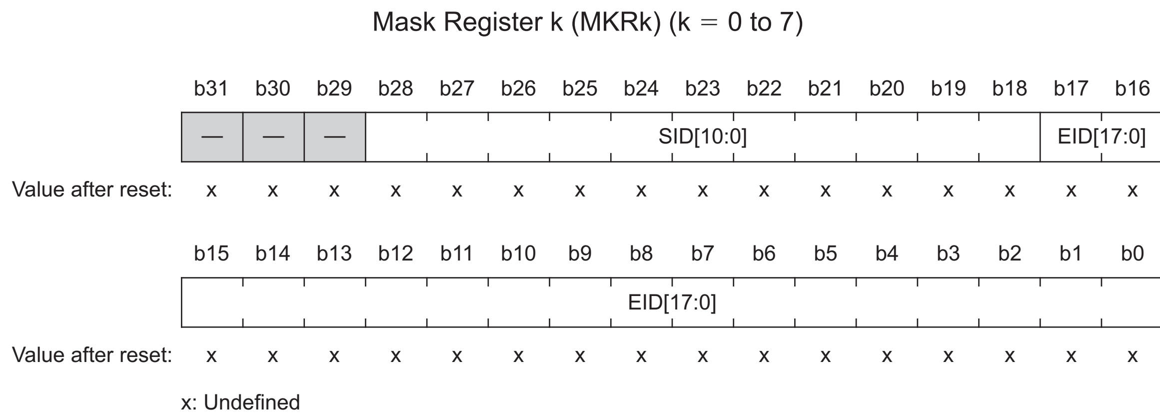


Figure 10.5 Mask Register [1], page 1487.

For information on various fields and register functions refer to [1], page 1487.

Mask Invalid Register (MKIVLR)

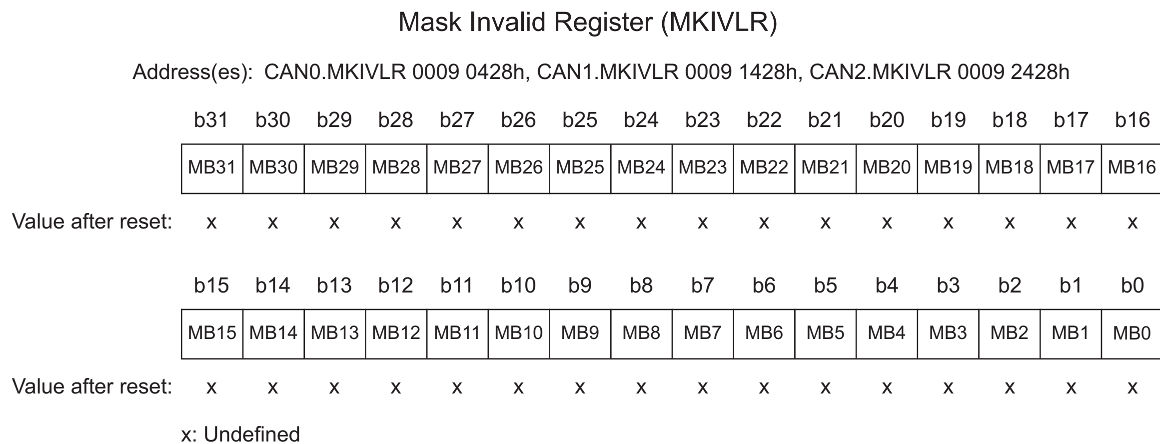


Figure 10.6 Mask invalid register [1], page 1489.

Each bit in MKIVLR corresponds to a mailbox. Bit (i) in MKIVLR corresponds to mailbox (i) (MBi). When a bit is set to 1, the relevant acceptance mask register becomes invalid for the corresponding mailbox. When a mask invalid bit is set to 1, a message is received

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 247

by the corresponding mailbox only if the receive message ID matches the mailbox ID exactly. The MKIVLR should be written to in the CAN reset or CAN halt mode.

Mailbox Register j (MBj) (j = 0 to 31)

Figure 10.7 shows the CAN data frame configuration. The value after reset of the CANi mailbox is undefined. MBj should be written to only when the related MCTLj (j = 0 to 31) is 00h and the corresponding mailbox is not processing an abort request.

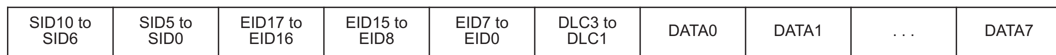


Figure 10.7 CAN data frame configuration [1], page 1490.

The previous value of each mailbox is retained unless a new message is received.

Address(es): CAN0.MB0 to CAN0.MB63 0009 0200h to 0009 03FFh, CAN1.MB0 to CAN1.MB63 0009 1200h to 0009 13FFh, CAN2.MB0 to CAN2.MB63 0009 2200h to 0009 23FFh

Figure 10.8 SID and EID bit setting of MBj [1], page 1491.

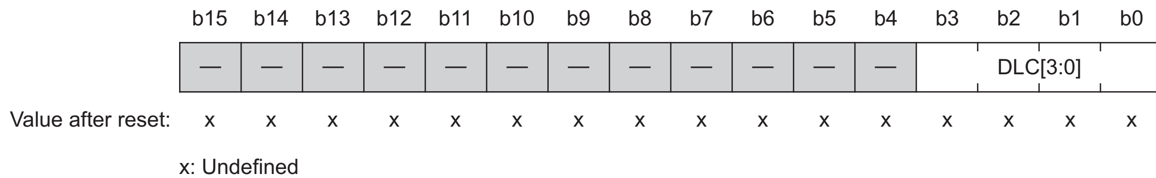


Figure 10.9 DLC bit setting of MBj [1], page 1490.

For information on various fields and register functions refer to [1], page 1490.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 248

Mailbox Interrupt Enable Register (MIER)

Mailbox Interrupt Enable Register (MIER)

Address(es): CAN0.MIER 0009 042Ch, CAN1.MIER 0009 142Ch, CAN2.MIER 0009 242Ch

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	MB31	MB30	MB29	MB28	MB27	MB26	MB25	MB24	MB23	MB22	MB21	MB20	MB19	MB18	MB17	MB16
Value after reset:	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	MB15	MB14	MB13	MB12	MB11	MB10	MB9	MB8	MB7	MB6	MB5	MB4	MB3	MB2	MB1	MB0
Value after reset:	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

x: Undefined

Figure 10.10 Mailbox interrupt enable register (MIER) [1], page 1491.

Each bit corresponds to interrupt enabling or disabling of that mailbox (MB[0]-MB[31]).

Message Control Register j (MCTLj) (j = 0 to 31)

Message Control Register j (MCTLj) (j = 0 to 31)

Address(es): CAN0.MCTL0 to CAN0.MCTL31 0009 0820h to 0009 083Fh,
CAN1.MCTL0 to CAN1.MCTL31 0009 1820h to 0009 183Fh,
CAN2.MCTL0 to CAN2.MCTL31 0009 2820h to 0009 283Fh

- Transmit mode (when the TRMREQ bit is 1 and the RECREQ bit is 0)

	b7	b6	b5	b4	b3	b2	b1	b0
	TRMREQ	RECREQ	—	ONESHOT	—	TRMABT	TRMACTIVE	SENTDATA
Value after reset:	0	0	0	0	0	0	0	0

- Receive mode (when the TRMREQ bit is 0 and the RECREQ bit is 1)

	b7	b6	b5	b4	b3	b2	b1	b0
	TRMREQ	RECREQ	—	ONESHOT	—	MSGLOST	INVALDATA	NEWDATA
Value after reset:	0	0	0	0	0	0	0	0

Figure 10.11 MCTL bit settings for transmit/receive mode [1], page 1495.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 249

For information on various fields and register functions refer to [1], page 1495.

Status Register (STR)

Status Register (STR)

Address(es): CAN0.STR 0009 0842h, CAN1.STR 0009 1842h, CAN2.STR 0009 2842h

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	RECST	TRMST	BOST	EPST	SLPST	HLTST	RSTST	EST	TABST	FMLST	NMLST	TFST	RFST	SDST	NDST
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 10.12 Status register [1], page 1504.

For information on various fields and register functions refer to [1], page 1504.

10.3.2 Reception and Transmission

When a mailbox is configured as a receive mailbox or a one-shot receive mailbox, note the following:

- Before a mailbox is configured as a receive mailbox or a one-shot receive mailbox, set MCTLj to 00h.
- A received message is stored into the first mailbox that matches the condition according to the result of receive-mode setting and acceptance filtering. Upon deciding the mailbox to store the received message, the mailbox with the smaller number has higher priority.
- In CAN operation mode, when the CAN module transmits a message whose ID matches with the ID/mask set of a mailbox configured to receive messages, the CAN module never receives the transmitted data. In self-test mode, however, the CAN module will receive its transmitted data. In this case, the CAN module returns ACK.

When configuring a mailbox as a transmit mailbox or a one-shot transmit mailbox, note the following:

- Before a mailbox is configured as a transmit mailbox or a one-shot transmit mailbox, ensure that MCTLj is 00h and that there is no pending abort process.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 250

Table 10.3 lists how to make the CAN communication mode settings.

TABLE 10.3 Settings for CAN Receive and Transmit Modes [1], page 1528.

MCTLJ. TRMREQ	MCTLJ. RECREQ	MCTLJ. ONESHOT	COMMUNICATION MODE OF MAILBOX
0	0	0	Mailbox disabled or transmission being aborted.
0	0	1	Can be configured only when transmission or reception from a mailbox programmed in one-shot mode is aborted.
0	1	0	Configured as a receive mailbox for a data frame or a remote frame.
0	1	1	Configured as a one-shot receive mailbox for a data frame or a remote frame.
1	0	0	Configured as a transmit mailbox for a data frame or a remote frame.
1	0	1	Configured as a one-shot transmit mailbox for a data frame or a remote frame.
1	1	0	Do not set.
1	1	1	Do not set.
j = 0 to 31			

Reception

This example shows the operation of overwriting the first message when the CAN module receives two consecutive CAN messages which match the receiving conditions of MCTLj (j = 0 to 31). Figure 10.13 shows an operation example of data frame reception in overwrite mode.

1. When an SOF is detected on the CAN bus, the RECST bit in STR is set to 1 (reception in progress) if the CAN module has no message ready to start transmission.
2. The acceptance filter processing starts at the beginning of the CRC field to select the receive mailbox.
3. After a message has been received, the NEWDATA bit in MCTLj for the receive mailbox is set to 1 (a new message is being stored or has been stored to the mailbox). The INVALDATA bit in MCTLj is set to 1 (a message is being updated) at the same time, and then the INVALDATA bit is set to 0 (message valid) again after the complete message is transferred to the mailbox.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 251

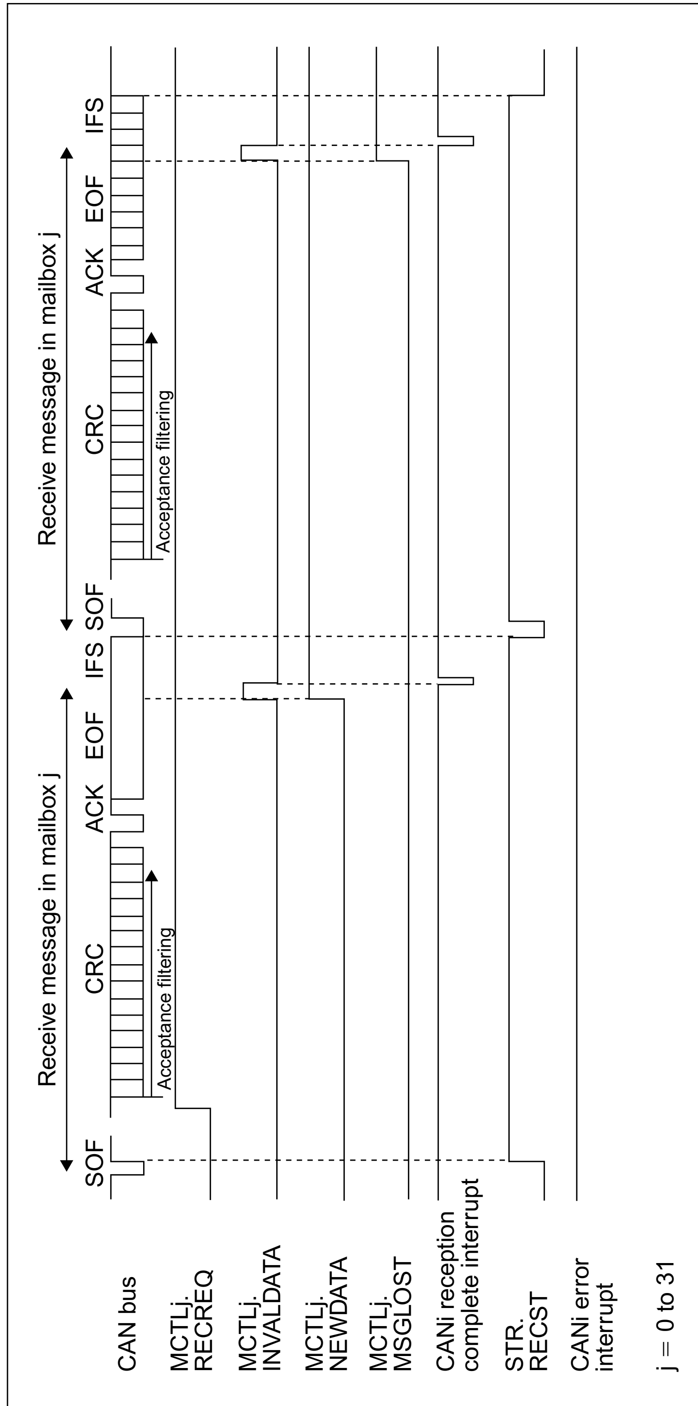


Figure 10.13 Data frame reception in overwrite mode [1], page 1529.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 252

4. When the interrupt enable bit in MIER for the receive mailbox is 1 (interrupt enabled), the CANi reception complete interrupt request is generated. This interrupt (CANi reception complete interrupt) is generated when the INVALDATA bit is set to 0.
5. After reading the message from the mailbox, the NEWDATA bit needs to be set to 0 by a program.
6. In overwrite mode, if the next CAN message has been received into a mailbox whose NEWDATA bit is still set to 1, the MSGLOST bit in MCTLj is set to 1 (message has been overwritten). The new received message is transferred into the mailbox. The CANi reception complete interrupt request is generated the same as in step 4.

Figure 10.14 shows the operation example of data frame reception in overrun mode. This example shows the operation of overrunning the second message when the CAN module receives two consecutive CAN messages which match the receiving conditions of MCTLj (j = 0 to 31).

Steps 1 to 5 are the same as in overwrite mode.

7. In overrun mode, if the next CAN message has been received before the NEWDATA bit in MCTLj is set to 0, the MSGLOST bit in MCTLj is set to 1 (message has been overrun). The new received message is discarded and a CANi error interrupt request is generated if the corresponding interrupt enable bit in EIER is set to 1 (interrupt enabled).

Transmission

Figure 10.15 shows an operation example of data frame transmission.

1. When a TRMREQ bit in MCTLj (j = 0 to 31) is set to 1 (transmit mailbox) in the bus-idle state, the mailbox scan processing starts to decide the highest-priority mailbox for transmission. Once the transmit mailbox is decided, the TRMACTIVE bit in MCTLj is set to 1 (from the acceptance of a transmission request to the completion of transmission, or error/arbitration-lost), the TRMST bit in STR is set to 1 (transmission in progress), and the CAN module starts transmission.
2. If other TRMREQ bits are set, the transmission scan processing starts with the CRC delimiter for the next transmission.
3. If transmission is completed without losing arbitration, the SENTDATA bit in MCTLj is set to 1 (transmission completed) and the TRMACTIVE bit is set to 0 (transmission is pending or transmission is not requested). If the interrupt enable bit in MIER is 1 (interrupt enabled), the CANi transmission complete interrupt request is generated.
4. When requesting the next transmission from the same mailbox, set bits SENTDATA and TRMREQ to 0, then set the TRMREQ bit to 1 after checking that bits SENTDATA and TRMREQ have been set to 0.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 253

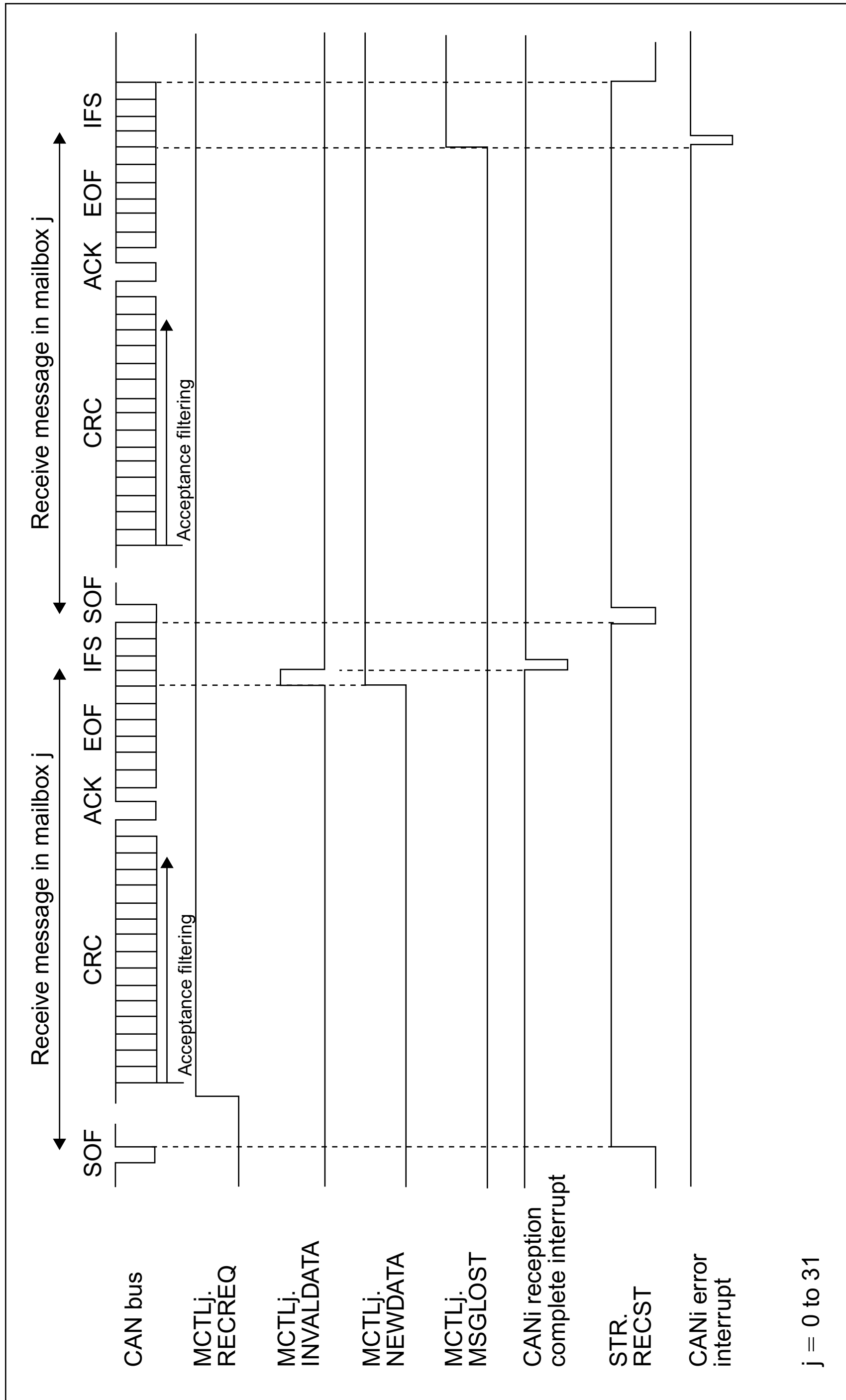


Figure 10.14 Data frame reception in overrun mode [1], page 1530.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 254

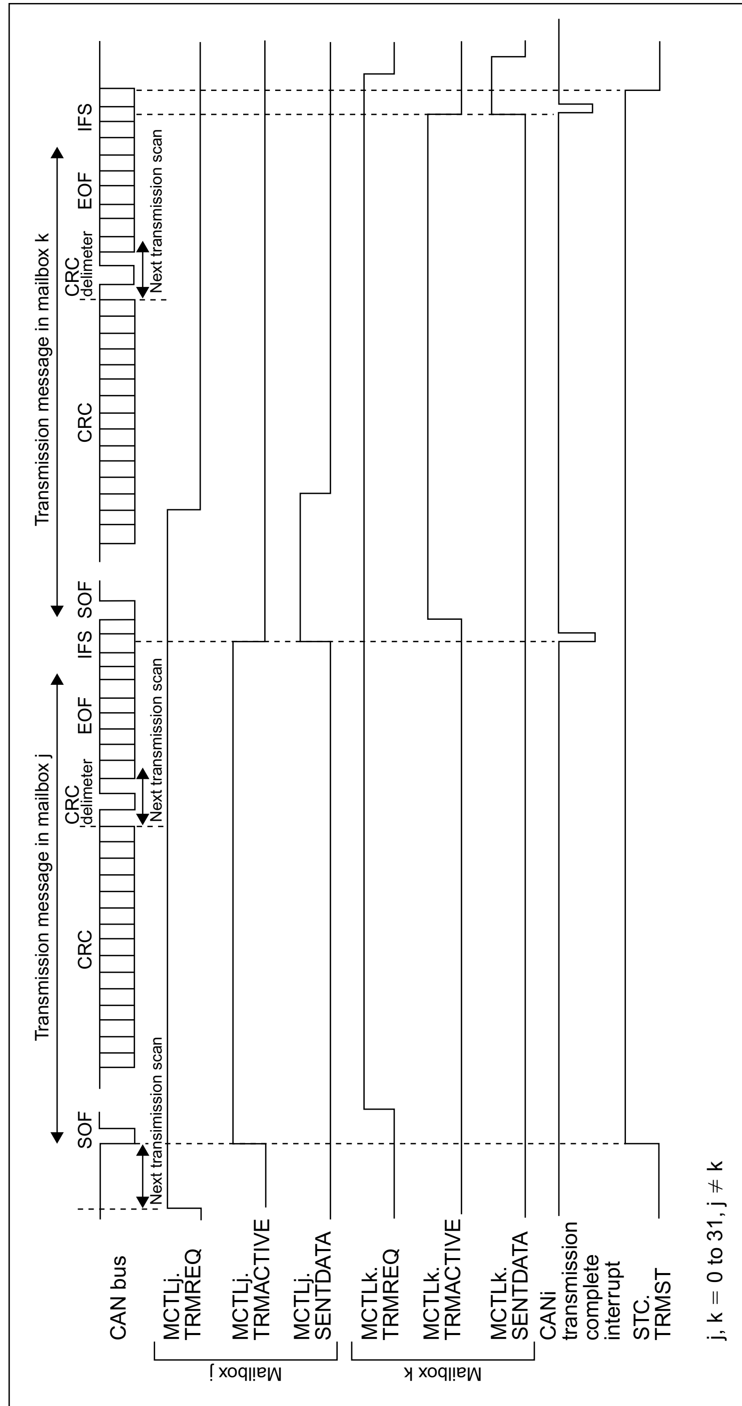


Figure 10.15 Data frame transmission [1], page 1531.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 255

10.3.3 Example 1: Initialization of CAN Bus

The following code shows how to initialize the CAN communications:

```
 1. int CANInit() {
 2.    int i,j;
 3.    SYSTEM.PRCR.WORD = 0xA502;
 4.    SYSTEM.MSTPCRB.BIT.MSTPB0 = 0;
 5.    SYSTEM.PRCR.WORD = 0xA500;
 6.    CAN0.CTLR.BIT.SLPM = 0;
 7.    CAN0.CTLR.BIT.CANM = 1;
 8.    CAN0.CTLR.BIT.BOM = 0;
 9.    CAN0.CTLR.BIT.MBM = 0;
10.    CAN0.CTLR.BIT.IDFM = 0;
11.    CAN0.CTLR.BIT.MLM = 1;
12.    CAN0.CTLR.BIT.TPM = 0;
13.    CAN0.CTLR.BIT.TSPS = 3;
14.
15.    //Setting up Baud Rate
16.    CAN0.BCR.BIT.BRP = 19;    //Baud Rate to 100kbps
17.                              //fCANCLK = 48M/20 = 2.4M
18.    CAN0.BCR.BIT.TSEG1 = 14;    //Tq = TSEG1 + TSEG2 + SJW =
                                     fCANCLK/Baud Rate
19.    CAN0.BCR.BIT.TSEG2 = 7;    //TSEG2 < TSEG1
20.    CAN0.BCR.BIT.SJW = 1;
21.    CAN0.MKIVLR.LONG = 0xFFFFFFFF;
22.    CAN0.CTLR.BIT.CANM = 2
23.    //Configuring Mailboxes in CAN halt mode
24.    for (i = 0; i < 32; i++) {
25.       CAN0.MB[i].ID.LONG = 0x00;
26.       CAN0.MB[i].DLC = 0x00;
27.       for (j = 0; j < 8; j++)
28.          CAN0.MB[i].DATA[j] = 0x00;
29.       for (j = 0; j < 2; j++)
30.          CAN0.MB[i].TS = 0x00;
31.    }
32.    CAN0.CTLR.BIT.CANM =0;
33.    CAN0.CTLR.BIT.TSRC = 1;
34.    if (CAN0.STR.BIT.EST)
35.       return 0;
36.    if (CAN0.EIFR.BYTE)
37.       return 0;
```

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 256


```
38.    CAN0.EIFR.BYTE = 0x00;
39.    if (CAN0.ECSR.BYTE)
40.       return 0;
41.    CAN0.ECSR.BYTE = 0x00;
42.    //Setting Up PORTS
43.    SYSTEM.PRCR.WORD = 0xA50B;
44.    MPC.PWPR.BYTE = 0x00;
45.    MPC.PWPR.BYTE = 0x40;
46.    PORT3.PMR.BIT.B3 = 0;
47.    PORT3.PDR.BIT.B3 = 0;
48.    PORT3.PMR.BIT.B2 = 0;
49.    PORT3.PDR.BIT.B2 = 0;
50.    MPC.P32PFS.BYTE = 0x10;
51.    MPC.P33PFS.BYTE = 0x10;
52.    PORT3.PMR.BIT.B3 = 1;
53.    PORT3.PMR.BIT.B2 = 1;
54.    PORT3.PDR.BIT.B3 = 0;
55.    PORT3.PDR.BIT.B2 = 1;
56.    MPC.PWPR.BYTE = 0x80;
57.    //End of Setting Up Ports
58.    CAN0.CTLR.BIT.CANM = 2;
59.    CAN0.MIER.LONG = 0x00;
60.    //Set the Rx Mailbox.
61.    CAN0.MCTL[rxmbx].BYTE = 0;
62.    CAN0.MB[rxmbx].ID.BIT.SID = 0x001;
63.    CAN0.MB[rxmbx].ID.BIT.RTR = 0;
64.    CAN0.MB[rxmbx].ID.BIT.IDE = 0;
65.    CAN0.MCTL[rxmbx].BYTE |= 0x40;
66.    CAN0.MKR[1].BIT.SID = 0x7FF;
67.    CAN0.MKIVLR.LONG & = ~(0x0010);
68.    CAN0.CTLR.BIT.CANM = 0;
69.    //Set the TX Mailbox*/
70.    txframe.id = 1;
71.    txframe.dlc = 8;
72.    for(i = 0; i < 8; i++)
73.       txframe.data[i] = i;
74.    CAN0.MCTL[txmbx].BYTE = 0;
75.    CAN0.MB[txmbx].ID.BIT.SID = 1;
76.    CAN0.MB[txmbx].ID.BIT.IDE = 0;
77.    CAN0.MB[txmbx].DLC = 0x8;
78.    CAN0.MB[txmbx].ID.BIT.RTR = 0;
79.    for (i = 0; i < 8; i++)
```

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 257


```
80.        CAN0.MB[txmbx].DATA[i] = txframe.data[i];
81.     CAN0.MCTL[mbox_nr].BIT.TX.TRMREQ = 1;
82.     return 1;
83. }
```

Line 3 shows how the user disables the write protect option in the protect register. Line 4 enables the CAN by setting the MSTPB0 to 0 in the Module Stop Control Register B (MSTPRB). Line 6 to line 13 is used to set and reset the bits of the Control Register (CTLR); line 6 Exits the CAN sleep mode by setting SLPM bit to 0, line 7 selects the CAN reset mode, line 8 selects the Reset mode. Line 8 does the function of selecting Normal mode in the Bus Off Recovery mode, line 9 selects the normal mailbox mode. Line 10 selects the Standard ID for CAN; line 11 sets the MLM bit to 1 to select Overrun mode. In line 12 the TPM bit is set to 0, the ID priority transmit mode is selected and the transmission priority complies with the CAN bus arbitration rule. Line 13 selects the prescaler for the timestamp, and updates every eight times.

Lines 16 through 20 are used to set the Bit Configuration Register (BCR). Line 16 sets the Baud Rate to 100 Kbps; line 18 and line 19 select the bit timing, the ranges are TSEG1 = 4 Tq (time quantum) to 16 Tq, TSEG2 = 2 Tq to 8 Tq and SJW = 1 Tq to 4 Tq (For a more detailed explanation refer 10.4.5). Line 20 selects the Resynchronization Jump Width Control bit to 1 to select 4 Tq of bit timing range. Line 21 invalidates the mask by setting all the bits of the Mask Invalid Register to 1, line 22 selects the CAN halt mode by setting the CTLR.CANM[1:0] bits to 2.

Line 24 runs a for loop to select all the thirty-two mailboxes IDs and Data Length to 0 bytes and line 30 sets the DATA bit of each mailbox to 0. Line 32 selects the CAN Operate mode. Line 33 resets the time stamp counter (TSRC) in the Control Register. Line 34 checks for any CAN errors and returns 0 if the Error Status Flag (EST) is set. Line 36 checks for all the eight sources of the interrupts of the EIFR; if this is true then it detects an error and returns 0. Line 38 sets all the bits of the EIFR register to 0, line 39 checks if any bit is set in the ECSR register and returns 0 indicating it failed. Line 41 sets the PRCR register to write protect off mode.

Lines 44 and 45 set the Write Protect Register, lines 46 through 49 set the CRX0 as the output and CTX0 as the input for the transmission and the reception. Lines 50 and 51 set the Pin Function Control Registers for CTX0 and CRX0 in the registers in the multifunction pin controller (MPC). Lines 52 through 55 are used to select the input and output for the CAN bus (for the CRX0 and CTX0). Line 58 sets the CAN halt mode for the function. Line 59 is used to disable the interrupts on the mailboxes since we are using polling.

Line 61 clears the message mailbox control register; line 62 sets the STD ID number for the mailboxes. The RTR bit in MBj selects a data frame or a remote frame, line 63 selects the data frame. Line 64 selects the standard ID. Line 65 configures the Message Mailbox Control Register for the reception. Line 67 is used to set the Mask Invalidate Register to such that the acceptance mask register is valid only for the mailboxes 16 to 23. Line 68 sets the CAN in the Operate CAN mode.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 258

Lines 70 and 71 set the TX ID and the number of data bits in the Transmission frame. Lines 72 and 73 are used to initialize the data that has to be sent. Line 74 resets the Message Mailbox Control for CAN0. Line 75 sets the Standard ID to 1 to compare the received messages with the corresponding mailbox ID. Line 76 disables the Extended ID messages' transmission; line 77 selects the number of data bytes that needs to be transmitted (8). Line 78 disables the Remote frame transmission. Line 80 initializes the mailbox's data with the frame transmission data. Line 81 enables transmission.

10.3.4 Example 2: Reception and Transmission

Reception

```
 1. int rxpoll(int mailboxno) {
 2.    int polling = 0x80;
 3.    while ((CAN0.MCTL[mailboxno].BIT.RX.INVALDATA) && polling)
 4.       polling—;
 5.    if (polling == 0)
 6.       //Still updating mailbox. Come back later.
 7.       return 0;
 8.    else {
 9.       if (CAN0.MCTL[mailboxno].BIT.RX.NEWDATA == 1)
10.       return 1;
11.    }
12. }
13. void rxread(int mailboxno ,int sid) {
14.    int i;
15.    rxframe.id = CAN0.MB[mailboxno].ID.BIT.SID;
16.    rxframe.dlc = CAN0.MB[mailboxno].DLC;
17.    for(i = 0; i < rxframe.dlc; i++)
18.       rxframe.data[i] = CAN0.MB[mailboxno].DATA[i];
19.    if (CAN0.MCTL[mailboxno].BIT.RX.MSGLOST)
20.       CAN0.MCTL[mailboxno].BIT.RX.MSGLOST = 0;
21.    CAN0.MCTL[mailboxno].BIT.RX.NEWDATA = 0;
22. }
```

Explanation

In the function `rxpoll`, line 3 checks if the message has been received by checking the invalid bit in message control register j (MCTLj) for a particular mailbox. The INVALDATA bit in MCTLj is set to 1 (message is being updated) at the same time, and then the INVALDATA bit

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 259

is set to 0 (message valid) again after the complete message is transferred to the mailbox. After the reception, line 9 checks the availability of data in the mailbox and returns 1, indicating that the message has been received.

In the function `rxread`, line 13 defines a function to read received data for a particular mailbox number and a standard ID. Line 15 copies the standard id number of reception and line 16 copies the number of bytes (indicating by the DLC) of data into a variable. Line 18 copies the data for processing from the DATA field in a particular mailbox. Line 19 checks if the message has been lost by checking the MSGLOST bit in the MCTLj register. Line 20 resets the MSGLOST bit for notification and line 21 resets the NEWDATA bit to 0 to enable detection of later receptions.

Transmission

```
 1. void CANTX(CANtxrx *txf,int txbx) {
 2.    int i;
 3.    CAN0.MCTL[mbox_nr].BIT.TX.TRMREQ = 0;
 4.    CAN0.MCTL[mbox_nr].BIT.TX.SENTDATA = 0;
 5.    for(i = 0; i < 8; i++)
 6.       CAN0.MB[txbx].DATA[i] = txf -> data[i];
 7.    CAN0.MCTL[mbox_nr].BIT.TX.TRMREQ = 1;
 8. }
 9. int check_sent(int mbxnr) {
10.    if (CAN0.MCTL[mbxnr].BIT.TX.SENTDATA == 0)
11.       return 1;
12.    else {
13.       CAN0.MCTL[mbox_nr].BIT.TX.TRMREQ = 0;
14.       CAN0.MCTL[mbox_nr].BIT.TX.SENTDATA = 0;
15.    return 0;
16.    }
17. }
```

Explanation

In the function `CANTX`, line 3 disables transmission for a particular mailbox and line 4 resets the SENTDATA bit in the MCTLj register to 0. Line 5 runs a loop eight times to initialize the data into the registers for transmission. Line 6 sets the data for transmission into DATA registers of the mailboxes. Line 7 enables transmission.

In the function `check_sent`, line 10 checks for the outcome of completion of the transmission by checking the SENTDATA bit in the MCTLj register and returns 1 if the transmission is not complete. Line 13 and line 14 sets SENTDATA and TRMREQ to 0 for the availability of the next transmission.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 260

10.3.5 Main Function

The following structure imitates the frame received from the CAN bus. It is used to store Data, ID, and DLC.

```
1. typedef struct {
2.    uint32_t id;
3.    uint8_t dlc;
4.    uint8_t data[8];
5. } CANtxrx;
```

We will use this structure in the main program:

```
 1. #include <stdint.h>
 2. #include <stdbool.h>
 3. #include <stdio.h>
 4. #include <machine.h>
 5. #include "platform.h"
 6. typedef struct {
 7.    uint32_t id;
 8.    uint8_t dlc;
 9.    uint8_t data[8];
10. } CANtxrx;
11. int rxint = 0, txint = 0;
12. //Receive Mailbox number:4   Transmit Mailbox number:1
13. #define txmbx 1
14. #define rxmbx 4
15. //function Prototypes
16. void rxread(int mailboxno ,int sid);
17. int caninit();
18. void CANTX(CANtxrx *txf,int txbx);
19. void checkbusy(int txbx);
20. int check_sent(int mbxnr);
21. int rxpoll(int mailboxno);
22. void rxread(int mailboxno ,int sid);
23. CANtxrx rxframe,txframe;
24.
25. void main() {
26.    int i;
27.    char l0[] = "  ", l1[] = "  ", l2[] = "  ", l3[] = "  ",
       l4[] = "  ", l5[] = "  ",
```

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 261


```
28.     l6[] = "  ", l7[] = "  ", super[] = "  ";
29.     lcd_initialize();
30.     lcd_clear();
31.     interrupts();
32.     i = caninit();
33.     if(i == 0)
34.        while(1);
35.     CANTX( &txframe,txmbx);
36.     for(i = 0x100; i > 0; i—)
37.        nop();
38.     //LCDDISPLAY("Transmission Initiated.");
39.     //while(check_sent(txmbx));
40.     while(1) {
41.        if(txint == 1) {
42.           txint = 0;
43.           sprintf(l0, "%2X", txframe.data[0]);
44.           sprintf(l1, "%2X", txframe.data[1]);
45.           sprintf(l2, "%2X", txframe.data[2]);
46.           sprintf(l3, "%2X", txframe.data[3]);
47.           sprintf(l4, "%2X", txframe.data[4]);
48.           sprintf(l5, "%2X", txframe.data[5]);
49.           sprintf(l6, "%2X", txframe.data[6]);
50.           sprintf(l7, "%2X", txframe.data[7]);
51.           sprintf(super, "%2s%2s%2s%2s", l0, l1, l2, l3);
52.           lcd_display(LCD_LINE2, "TxData");
53.           lcd_display(LCD_LINE3, super);
54.           sprintf(super, "%2s%2s%2s%2s", l4, l5, l6, l7);
55.           lcd_display(LCD_LINE4, super);
56.        }
57.        if(rxint == 1) {
58.           rxread(rxmbx ,1);
59.           if(rxframe.id == 1) {   //lcd
60.              sprintf(l0, "%2X", rxframe.data[0]);
61.              sprintf(l1, "%2X", rxframe.data[1]);
62.              sprintf(l2, "%2X", rxframe.data[2]);
63.              sprintf(l3, "%2X", rxframe.data[3]);
64.              sprintf(l4, "%2X", rxframe.data[4]);
65.              sprintf(l5, "%2X", rxframe.data[5]);
66.              sprintf(l6, "%2X", rxframe.data[6]);
67.              sprintf(l7, "%2X", rxframe.data[7]);
68.              sprintf(super, "%2s%2s%2s%2s", l0, l1, l2, l3);
```

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 262


```
69.                lcd_display(LCD_LINE5, "RxData");
70.                lcd_display(LCD_LINE6, super);
71.                sprintf(super, "%2s%2s%2s%2s", l4, l5, l6, l7);
72.                lcd_display(LCD_LINE7, super);
73.                txframe.id = rxframe.id;
74.                txframe.dlc = rxframe.dlc;
75.                for(i = 0; i < rxframe.dlc; i++)
76.                   txframe.data[i] = rxframe.data[i] + 2;
77.                CANTX( &txframe,txmbx);
78.             }
79.             rxint = 0;
80.          }
81.       }
82. }
```

Lines 13 and 14 define the transmit mailbox (1) and receive mailbox (4). Line 23 has two structure elements for transmission and reception. Lines 27 and 28 have the LCD manipulation declarations. Line 35 sends the contents of the CAN tx buffer through the CAN bus. When transmission is over the txint variable becomes 1 and the string manipulation procedure begins for LCD from line 41 to 56. When the reception is done, the rxint variable becomes 1 and the string manipulation begins for the LCD from 59 to 72. From line 73 to 77 the code does the work of adding 2 to the received data and transmitting it again.

10.4 ADVANCED CONCEPTS

The CAN module has the following four operating modes:

- CAN reset mode
- CAN halt mode
- CAN operation mode
- CAN sleep mode

Figure 10.16 shows the transition between CAN operating modes.

10.4.1 CAN Reset Mode

CAN reset mode is provided for the CAN communication configuration. When the CTLR.CANM[1:0] bits are set to 01b or 11b, the CAN module enters CAN reset mode.

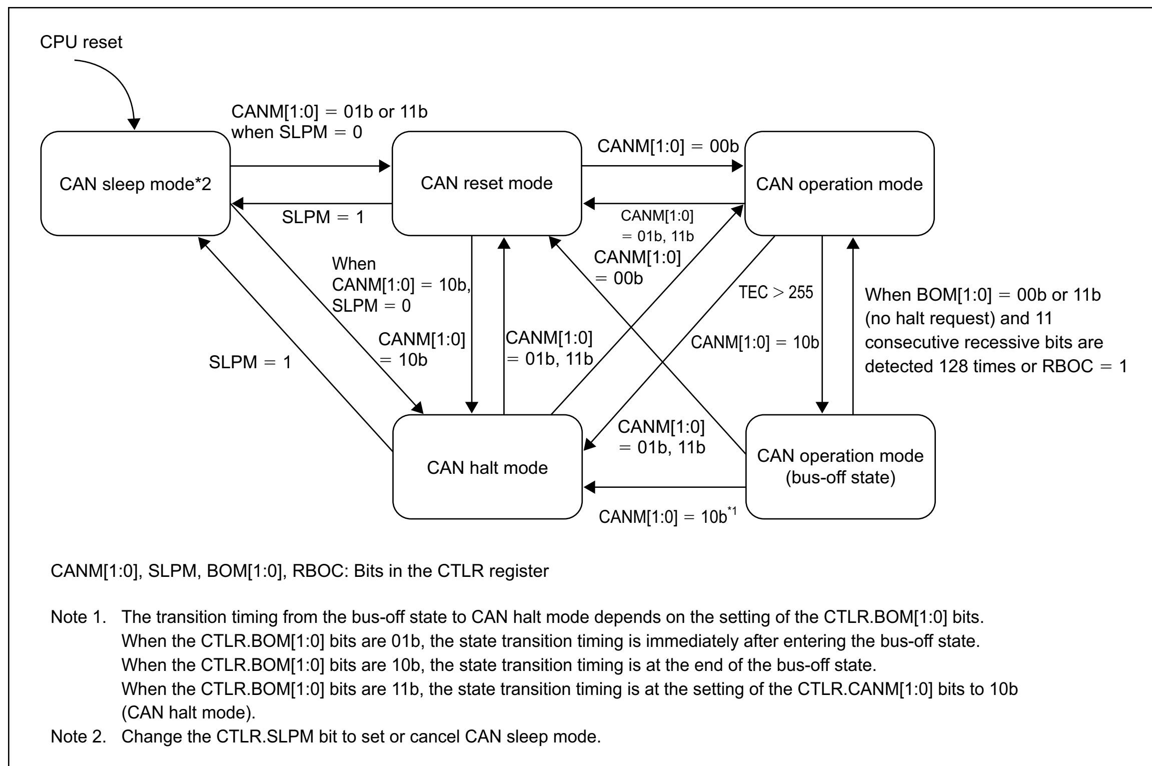


Figure 10.16 Transition between CAN operating modes [1], page 1518.

Then, the STR.RSTST bit is set to 1. Do not change the CTLR.CANM[1:0] bits until the RSTST bit is set to 1. Set the BCR before exiting the CAN reset mode to any other modes.

10.4.2 CAN Halt Mode

CAN halt mode is used for mailbox configuration and test mode setting. When the CTLR.CANM[1:0] bits are set to 10b, CAN halt mode is selected. Then the STR.HLTST bit is set to 1. Do not change the CTLR.CANM[1:0] bits until the HLTST bit is set to 1. All registers except for bits RSTST, HLTST, and SLPST in STR remain unchanged when the CAN enters CAN halt mode.

Do not change the CTLR (except for bits CANM[1:0] and SLPM) and EIER in the CAN halt mode. The BCR can be changed in the CAN halt mode only when listen-only mode is selected for automatic baud rate detection.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 264

10.4.3 CAN Sleep Mode

CAN sleep mode is used for reducing current consumption by stopping the clock supply to the CAN module. After a reset from an MCU pin or a software reset, the CAN module starts from the CAN sleep mode.

When the SLPM bit in CTLR is set to 1, the CAN module enters CAN sleep mode. Then, the SLPST bit in STR is set to 1. Do not change the value of the SLPM bit until the SLPST bit is set to 1. The other registers remain unchanged when the CAN module enters CAN sleep mode.

Write to the SLPM bit in CAN reset mode and CAN halt mode. Do not change any registers (except for the SLPM bit) during the CAN sleep mode. Read operation is still allowed. When the SLPM bit is set to 0, the CAN module is released from the CAN sleep mode. When the CAN module exits the CAN sleep mode, the other registers remain unchanged.

10.4.4 CAN Operation Mode

The CAN operation mode is used for CAN communication. When the CANM[1:0] bits in CTLR are set to 00b, the CAN module enters the CAN operation mode. Then bits RSTST and HLTST in STR are set to 0. Do not change the value of the CANM[1:0] bits until bits RSTST and HLTST are set to 0. If eleven consecutive recessive bits are detected after entering the CAN operation mode, the CAN module is in the following states:

- The CAN module becomes an active node on the network, thus enabling transmission and reception of CAN messages.
- Error monitoring of the CAN bus, such as receive and transmit error counters, is performed.

During the CAN operation mode, the CAN module may be in one of the following three sub-modes, depending on the status of the CAN bus.

- Idle mode: Transmission or reception is not being performed.
- Receive mode: A CAN message sent by another node is being received.
- Transmit mode: A CAN message is being transmitted. The CAN module receives a message transmitted by the local node simultaneously when self-test mode 0 (TSTM[1:0] bits in TCR = 10b) or self-test mode 1 (TSTM[1:0] bits = 11b) is selected.

Figure 10.17 shows the sub-modes of CAN operation mode.

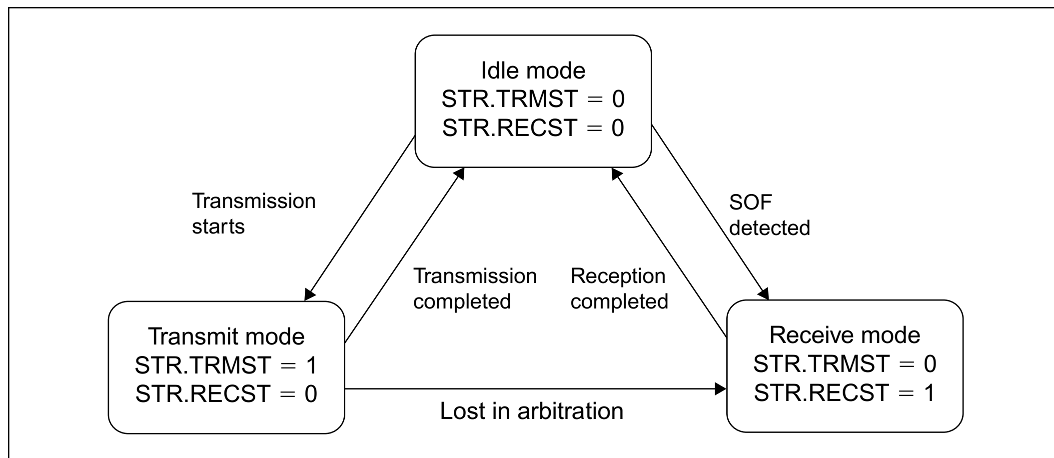


Figure 10.17 Sub-modes of CAN operation mode [1], page 1521.

10.4.5 CAN Communication Speed Setting

The following description explains about the CAN communication speed setting.

CAN Clock Setting

The CAN module has a CAN clock selector. The CAN clock can be set by the CCLKS bit and the BRP[9:0] bits in BCR. Figure 10.18 shows a block diagram of the CAN clock generator.

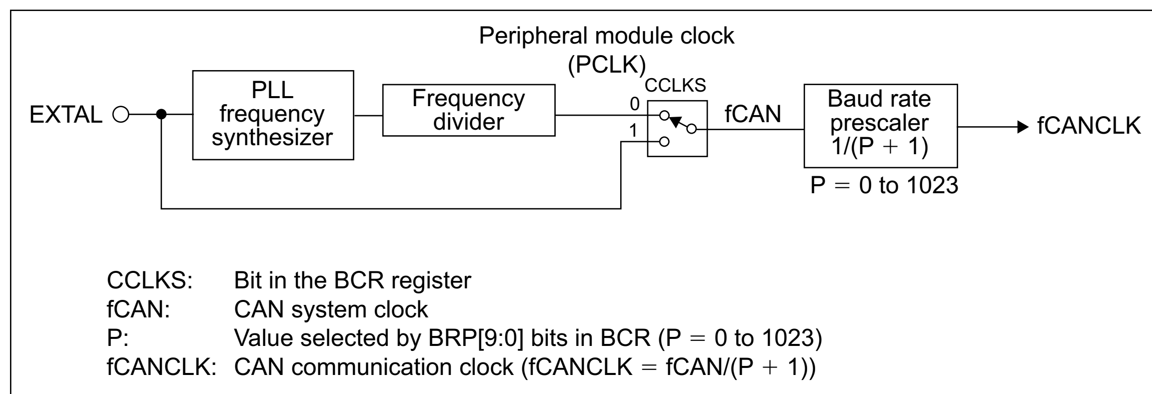


Figure 10.18 CAN clock generator [1], page 1523.

Bit Timing Setting

The bit time consists of the following three segments. Figure 10.19 shows the bit timing.

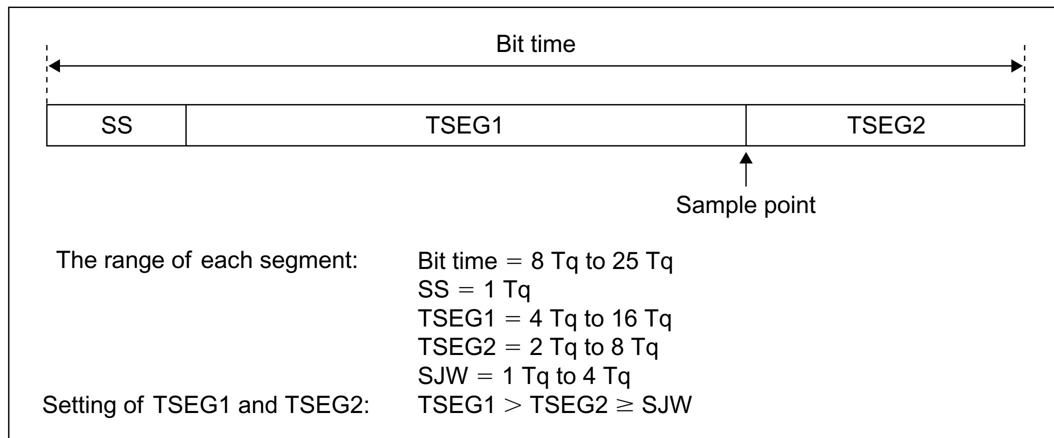


Figure 10.19 Bit timing [1], page 1523.

Bit Rate

The bit rate depends on the division value of fCAN (CAN clock), the division value of the baud rate prescaler, and the number of Tq for 1 bit of time.

$$\text{Bit rate [bps]} = \frac{fCAN}{\text{Baud rate prescaler division value}^{*1} \text{ X number of Tq of 1 bit time}} = \frac{fCANCLK}{\text{Number of Tq of 1 bit time}}$$

Note 1: Baud rate prescaler division value = P + 1 (P: 0 to 1023).
P: Setting of the BRP[9:0] bits in BCR.

10.4.6 Acceptance Filtering and Masking Functions

The acceptance filtering function and the masking function allows the user to select and receive messages with a specified range of multiple IDs for mailboxes. Registers MKR0 to MKR7 can perform masking of the standard ID and the extended ID of 29 bits.

- MKR0 corresponds to mailboxes [0] to [3]
- MKR1 corresponds to mailboxes [4] to [7]

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 267

TABLE 10.4 Bit Rate Examples [1], page 1521.

fCAN	50 MHz		48 MHz		40 MHz		32 MHz	
BIT RATE	NUMBER OF Tq	P + 1	NUMBER OF Tq	P + 1	NUMBER OF Tq	P + 1	NUMBER OF Tq	P + 1
1 Mbps	10Tq	5	8Tq	6	10Tq	4	8Tq	4
	25Tq	2	12Tq	4	20Tq	2	16Tq	2
			16Tq	3				
500 kbps	10Tq	10	8Tq	12	10Tq	8	8Tq	8
	25Tq	4	12Tq	8	20Tq	4	16Tq	4
			16Tq	6				
250 kbps	10Tq	20	8Tq	24	10Tq	16	8Tq	16
	25Tq	8	12Tq	16	20Tq	8	16Tq	8
			16Tq	12				
125 kbps	10Tq	40	8Tq	48	10Tq	32	8Tq	32
	25Tq	16	12Tq	32	20Tq	16	16Tq	16
			16Tq	24				
83.3 kbps	10Tq	60	8Tq	72	8Tq	60	8Tq	48
	25Tq	24	12Tq	48	10Tq	48	16Tq	24
			16Tq	36	16Tq	30		
					20Tq	24		
33.3 kbps	10Tq	150	8Tq	180	8Tq	150	8Tq	120
	25Tq	60	12Tq	120	10Tq	120	10Tq	96
			16Tq	90	20Tq	60	16Tq	60
							20Tq	48

- MKR2 corresponds to mailboxes [8] to [11]
- MKR3 corresponds to mailboxes [12] to [15]
- MKR4 corresponds to mailboxes [16] to [19]
- MKR5 corresponds to mailboxes [20] to [23]
- MKR6 corresponds to mailboxes [24] to [27] in normal mailbox mode and the receive FIFO mailboxes [28] to [31] in FIFO mailbox mode.
- MKR7 corresponds to mailboxes [28] to [31] in normal mailbox mode and the receive FIFO mailboxes [28] to [31] in FIFO mailbox mode.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 268

The MKIVLR disables acceptance filtering individually for each mailbox. The IDE bit in MBj is valid when the IDFM[1:0] bits in CTLR are 10b (mixed ID mode). The RTR bit in MBj selects a data frame or a remote frame. In FIFO mailbox mode, normal mailboxes (mailboxes [0] to [23]) use the single corresponding register among MKR0 to MKR5 for acceptance filtering. Receive FIFO mailboxes (mailboxes [28] to [31]) use two registers MKR6 and MKR7 for acceptance filtering. Also, the receive FIFO uses two registers, FIDCR0 and FIDCR1, for ID comparison. Bits EID[17:0], SID[10:0], RTR, and IDE in MB28 to MB31 for the receive FIFO are disabled.

Since acceptance filtering depends on the result of two logic AND operations, two ranges of IDs can be received into the receive FIFO. MKIVLR is disabled for the receive FIFO. If both the standard ID and extended ID are set in the IDE bits in FIDCR0 and FIDCR1 individually, both ID formats are received. If both the data frame and remote frame are set in the RTR bits in FIDCR0 and FIDCR1 individually, both data and remote frames are received. When combination with two ranges of IDs is not necessary, set the same mask value and the same ID into both the FIFO ID and the mask register.

10.4.7 CAN Interrupts

The CAN module provides the following CAN interrupts for each channel. Table 10.5 lists CAN interrupts.

- CANi reception complete interrupt (mailboxes 0 to 31) [RXMi]
- CANi transmission complete interrupt (mailboxes 0 to 31) [TXMi]
- CANi receive FIFO interrupt [RXFi]
- CANi transmit FIFO interrupt [TXFi]
- CANi error interrupt [ERSi]

There are eight types of interrupt sources for the CANi error interrupts. These sources can be determined by checking EIFR.

- Bus error
- Error-warning
- Error-passive
- Bus-off entry
- Bus-off recovery
- Receive overrun
- Overload frame transmission
- Bus lock

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 269

TABLE 10.5 CAN Interrupts [1], page 1532.

MODULE	INTERRUPT SYMBOL	INTERRUPT SOURCE	SOURCE FLAG
CANi	ERSi	Bus lock detected	EIFR.BLIF
		Overload frame transmission detected	EIFR.OLIF
		Overrun detected	EIFR.ORIF
		Bus-off recovery detected	EIFR.BORIF
		Bus-off entry detected	EIFR.BOEIF
		Error-passive detected	EIFR.EPIF
		Error-warning detected	EIFR.EWIF
		Bus error detected	EIFR.BEIF
	RXFi	Receive FIFO message received (MIER[29] = 0)	RFCR.RFUST[2:0]
		Receive FIFO warning (MIER[29] = 1)	
	TXFi	Transmit FIFO message transmission completed (MIER[25] = 0)	TFCR.TFUST[2:0]
		FIFO last message transmission completed (MIER[25] = 1)	
	RXMi	Mailbox [0] to [31] message received	MCTL0.NEWDATA to MCTL31.NEWDATA
	TXMi	Mailbox [0] to [31] message transmission completed	MCTL0.SENTDATA to MCTL31.SENTDATA

10.5 COMPLEX EXAMPLES

Using Interrupts

```
1. #pragma interrupt tx_ISR(vect = VECT_CAN0_TXM0, enable)
2. void tx_ISR(void) {
3.    txint = 1;
4.    if(check_sent(txmbx) == 0);
5.    txint = 1;
6. }
7. #pragma interrupt rx_ISR(vect = VECT_CAN0_RXM0, enable)
8. void rx_ISR(void) {
```

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 270


```
 9. if(rxpoll(4) == 1))
10.     rxint = 1;
11. }
12. void interrupts() {
13.     IEN(CAN0, TXM0) = 1;
14.     IPR(CAN0, TXM0) = 2;
15.     //Configure CAN Rx interrupt.
16.     IEN(CAN0, RXM0) = 1;
17.     IPR(CAN0, RXM0) = 2;
18.     CAN0.MIER.LONG = 0x00000000;
19.     ICU.IPR[18].BIT.IPR = 2;
20. }
```

10.6 CAN BUS APPLICATION PROGRAMMING INTERFACE

Now that you have seen the methods to use the CAN bus registers on the RX63N, you can appreciate the work Renesas has done to provide an application programming interface (API) for the same functionality. The software and documentation can be downloaded from Renesas. This section includes only a small part of what is provided in the Application Note *RX600 Series CAN Application Programming Interface, Rev. 2.03* [2].

The software download includes three files:

- `r_can_api.h`contains the definitions of the APIs, #defines, and data structures. This file should be used as-is and should not be changed.
- `r_can_api.c`contains the actual CAN code. This file should be used as-is and should not be changed.
- `config_r_can_api.h`contains a convenient location to change configurable definitions for the CAN APIs, like determining if you will use polling or interrupts, setting baud rates, and identifying CAN channel pins.

The CAN API includes functions in four major groups:

- **Initialization, port and peripheral control**—initialize the CAN peripheral registers and configure the CAN and transceiver ports.
- **Send**—set up a mailbox to transmit and to check that it was sent successfully.
- **Receive**—set up a mailbox to receive and to retrieve a message.
- **Error check**—check the CAN bus status of the node.

This code is an example of using the APIs to create a system where sends and receives use interrupts to set status bits and the main program sends/consumes data.

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 271

The CAN Bus initialization (lines 1 to 21) configure the mailboxes and baud rate. Note that the specific baud rate values were set in the `config_r_can_api.h` file and used in `R_CAN_Create`.

```
 1. uint32_t initCAN(void) {
 2.    uint32_t init_status = R_CAN_OK;
 3.    init_status = R_CAN_Create(can_channel);
 4.    if (init_status != R_CAN_OK) {
 5.       lcd_display(LCD_LINE8, "Error");
 6.       return init_status;
 7.    }
 8.    R_CAN_PortSet(can_channel, ENABLE);   //Normal Run Mode
 9.    //Enter Mailboxes into Halt mode
10.    init_status |= R_CAN_Control(can_channel, HALT_CANMODE);
11.    init_status |= R_CAN_RxSet(can_channel, CANBOX_RX, rx_id,
       DATA_FRAME);
12.    R_CAN_RxSetMask(can_channel, CANBOX_RX, 0x7FF);
13.    tx_dataframe.id = tx_id;
14.    tx_dataframe.dlc = 4;
15.    for(int i = 0; i < tx_dataframe.dlc; i++) {
16.       tx_dataframe.data[i] = 0x00;
17.    }
18.    init_status |= R_CAN_Control(can_channel, OPERATE_CANMODE);
19.    rx_dataframe.id = rx_id;
20.    return init_status;
21. }
22.
```

The three interrupt service routines (lines 23 to 45) are used to set global status variables instead of relying on CAN polling API functions like `R_CAN_Txcheck`. The third interrupt service routine is configured to capture spurious CAN errors.

```
23. #pragma interrupt CAN0_TXM0_ISR(vect = VECT_CAN0_TXM0, enable)
24. void CAN0_TXM0_ISR(void) {
25.    uint32_t api_status = R_CAN_OK;
26.    api_status = R_CAN_TxCheck(can_channel, CANBOX_TX);
27.    if (R_CAN_OK == api_status) {
28.       tx_sentdata_flag = 1;
29.    }
30. }
31.
```

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 272


```
32. #pragma interrupt CAN0_RXM0_ISR(vect = VECT_CAN0_RXM0, enable)
33. void CAN0_RXM0_ISR(void) {
34.    uint32_t api_status = R_CAN_OK;
35.    api_status = R_CAN_RxPoll(can_channel, CANBOX_RX);
36.    if (R_CAN_OK == api_status) rx_newdata_flag = 1;
37.
38. }
39.
40. //The APIs set this interrupt up regardless of your settings.
41. //It must be included or the errors will cause unwanted
      interrupts
42. #pragma interrupt CAN_ERS_ISR(vect = VECT_ICU_GROUPE0, enable)
43. void CAN_ERS_ISR(void) {
44.    nop();
45. }
```

The CAN Bus send/receive example below does nothing more than receive four bytes of character data, increment each character by the value 1, and send these changed characters. Lines 46 through 59 set up our main program. The infinite while loop (from lines 67 to 94) runs two major tasks: display successfully transmitted data (lines 68 to 77); and receive, then transmit data (lines 78 through 93). Note that the actual reception of data is performed in line 81, and the actual transmission of data is performed in line 92.

```
46. #include <machine.h>
47. #include "platform.h"
48. #include "config_r_can_rapi.h"
49. #include "r_can_api.h"
50.
51. uint32_t initCAN(void);
52.
53. can_frame_t tx_dataframe;
54. can_frame_t rx_dataframe;
55. uint32_t can_channel = 0;
56. uint32_t tx_sentdata_flag = 0;
57. uint32_t rx_newdata_flag = 0;
58. uint32_t tx_id = 0x001;
59. uint32_t rx_id = 0x001;
60.
61. void main(void) {
62.    uint32_t can_status = R_CAN_OK;
63.    uint8_t disp_buf[13] = {0};
```

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 273


```
64.    lcd_initialize();
65.    lcd_clear();
66.    can_status = initCAN();
67.    while(1) {
68.       if(tx_sentdata_flag == 1) {
69.          tx_sentdata_flag = 0;
70.          lcd_display(LCD_LINE2, "Tx OK   ");
71.          sprintf((char *)disp_buf, "%02X%02X%02X%02X",
72.             tx_dataframe.data[0],
73.             tx_dataframe.data[1],
74.             tx_dataframe.data[2],
75.             tx_dataframe.data[3] );
76.          lcd_display(LCD_LINE3, disp_buf);
77.       }
78.       if(rx_newdata_flag == 1) {
79.          rx_newdata_flag = 0;
80.          lcd_display(LCD_LINE4, "Rx OK. Read:");
81.          can_status = R_CAN_RxRead(can_channel,
             CANBOX_RX,&rx_dataframe);
82.          sprintf((char *)disp_buf, "%02X%02X%02X%02X",
83.             rx_dataframe.data[0],
84.             rx_dataframe.data[1],
85.             rx_dataframe.data[2],
86.             rx_dataframe.data[3] );
87.          lcd_display(LCD_LINE5, disp_buf);
88.          tx_dataframe.data[0] = rx_dataframe.data[0] + 1;
89.          tx_dataframe.data[1] = rx_dataframe.data[1] + 1;
90.          tx_dataframe.data[2] = rx_dataframe.data[2] + 1;
91.          tx_dataframe.data[3] = rx_dataframe.data[3] + 1;
92.          R_CAN_TxSet(can_channel, CANBOX_TX, &tx_dataframe,
             DATA_FRAME);
93.       }
94.    }
95. }
```

10.7 RECAP

This chapter reviewed the general theory and implementation of the CAN bus. We started by covering the theory behind the CAN protocol, its message formats and standards along with its application fields and benefits. Then we covered the CAN module in the RX63N

10.ES_Conrad_RX63N_Advanced_CH10.qxd:RX63N Advanced 3/4/14 11:59 AM Page 274

microprocessor and the registers used to implement CAN bus communication on it. The basic concepts and examples give the full process of setting up the CAN bus and its related registers and mailboxes for transmission and reception. The advanced concepts covered the modes of operation of the CAN bus along with how to change the speed of communication and using the CAN bus with interrupts.

10.8 REFERENCES

[1] Renesas Electronics, Inc. (February, 2013). *RX63N Group, RX631 Group User's Manual: Hardware,* Rev 1.60.

[2] Renesas Electronics, Inc. (March, 2013). *Application Note: RX600 Series CAN Application Programming Interface,* Rev. 2.03

10.9 EXERCISES

1. Write the code to set the bus to transmit at a baud rate as 500 kbps.
2. Write the code to set the bus to transmit the baud rate as 125 kbps.
3. Configure the CAN bus to send data with mailbox number 1 with Extended ID.
4. Write the code to transmit a 4-byte data on the CAN bus.
5. Write a code to transmit "Renesas Rulz!!" through the CAN bus of the RX63N and display it on the LCD on the reception board.
6. Write a code to receive 8-byte data from the terminal and transmit the inverted data through the CAN bus.
7. Connect three RX63N boards with standard IDs of 1, 2, and 3 and configure them to transmit frames. They should receive frames and store only the ones which have their respective IDs.
8. Take the X, Y, and Z co-ordinates from the accelerometer of the RX63N board and transfer it through the CAN bus using CAN interrupts.

Watchdog Timer and Brownout

11.1 LEARNING OBJECTIVES

Embedded systems are expected to work correctly, however, it is very difficult to completely test a system in all conceivable environments. Hence embedded system designers usually rely on a **watchdog timer (WDT)** to reset the processor if the program runs out of control. They also use a brownout (or low-voltage) detector to hold the processor in reset if the supply voltage is too low for correct operation. In this chapter the reader will learn:

- How to use the watchdog timer
- Brownout condition
- How to avoid brownout

11.2 BASIC CONCEPTS OF WATCHDOG TIMERS

Embedded systems must be able to cope with both hardware and software anomalies to be truly robust. In many cases, embedded devices operate in total isolation and are not accessible to an operator. Manually resetting a device in this scenario when its software "hangs" is not possible. In extreme cases, this can result in damaged hardware, a significant cost impact, or worse yet, create a human safety risk. A watchdog timer is a hardware timing device that triggers a system reset, or similar operation, after a designated amount of time has elapsed. A watchdog timer can be either a stand-alone hardware component or built into the processor itself. To avoid a reset, an application must periodically reset the watchdog timer before the specified interval elapses.

Note that a watchdog can also be useful in determining if other peripheral devices of the embedded system are functioning correctly. For example, if the system relies on a peripheral's regular activity, and that activity stops, the watchog can be useful in reseting the system to a state where it can report such inactivity and possibly correct it.

11.3 WATCHDOG TIMER IN RX63N

The Watchdog Timer (WDT) in the RX63N microcontroller is a 14-bit timer which outputs an overflow signal (WDTOVF) if the timer overflow occurs during general program flow,

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 276

and it can also be set up to reset the processor whenever the overflow signal occurs. The watchdog timer can also be used as an interval timer which generates an interrupt each time the counter overflows.

11.3.1 Register Description

The WDT has two start modes:

- Auto-start mode, in which counting automatically starts after release from the reset state.
- Register start mode, in which counting is started by refreshing the WDT (writing to the register).

The WDT control/status register (WDTCSR) is used for selecting the timer operation and clock source for the timer. The WDT reset control register (WDTRCR) is used to reset the timer.

TABLE 11.1 Specifications of WDT [1], page 1080–1081.

ITEM	SPECIFICATIONS
Count source	Peripheral clock (PCLK)
Clock division ratio	Divide by 4, 64, 128, 512, 2,048, or 8,192
Counter operation	Counting down using a 14-bit down-counter
Conditions for starting the counter	■ Counting automatically starts after a reset (auto-start mode) ■ Counting is started by refreshing the WDTRR register (writing 00h and then FFh) (register start mode)
Conditions for stopping the counter	■ Pin reset (the down-counter and registers return to their initial values) ■ A counter underflows or a refresh error is generated Count restarts automatically in auto-start mode, or by refreshing the counter in register start mode.
Window function	Window start and end positions can be specified (refresh-permitted and refresh-prohibited periods)
Reset-output sources	■ Down-counter underflows ■ Refreshing outside the refresh-permitted period (refresh error)
Interrupt request output sources	■ A non-maskable interrupt (WUNI) is generated by an underflow of the down-counter ■ Refreshing outside the refresh-permitted period (refresh error)
Reading the counter value	The down-counter value can be read by the WDTSR register.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 277

TABLE 11.1 Specifications of WDT [1], page 1080–1081.—*Continued*

ITEM	SPECIFICATIONS
Output signal (internal signal)	■ Reset output ■ Interrupt request output
Auto-start mode (controlled by the option function select register 0 (OFS0))	■ Selecting the clock frequency division ratio after a reset (OFS0.WDTCKS[3:0] bits) ■ Selecting the time-out period of the watchdog timer (OFS0.WDTTOPS[1:0] bits) ■ Selecting the window start position in the watchdog timer (OFS0.WDTRPSS[1:0] bits) ■ Selecting the window end position in the watchdog timer (OFS0.WDTRPES[1:0] bits) ■ Selecting the reset output or interrupt request output (OFS0.WDTRSTIRQS bit)
Register start mode (controlled by the WDT registers)	■ Selecting the clock frequency division ratio after refreshing (WDTCR.CKS[3:0] bits) ■ Selecting the time-out period of the watchdog timer (WDTCR.TOPS[1:0] bits) ■ Selecting the window start position in the watchdog timer (WDTCR.RPSS[1:0] bits) ■ Selecting the window end position in the watchdog timer (WDTCR.RPES[1:0] bits) ■ Selecting the reset output or interrupt request output (WDTRCR.RSTIRQS bit)

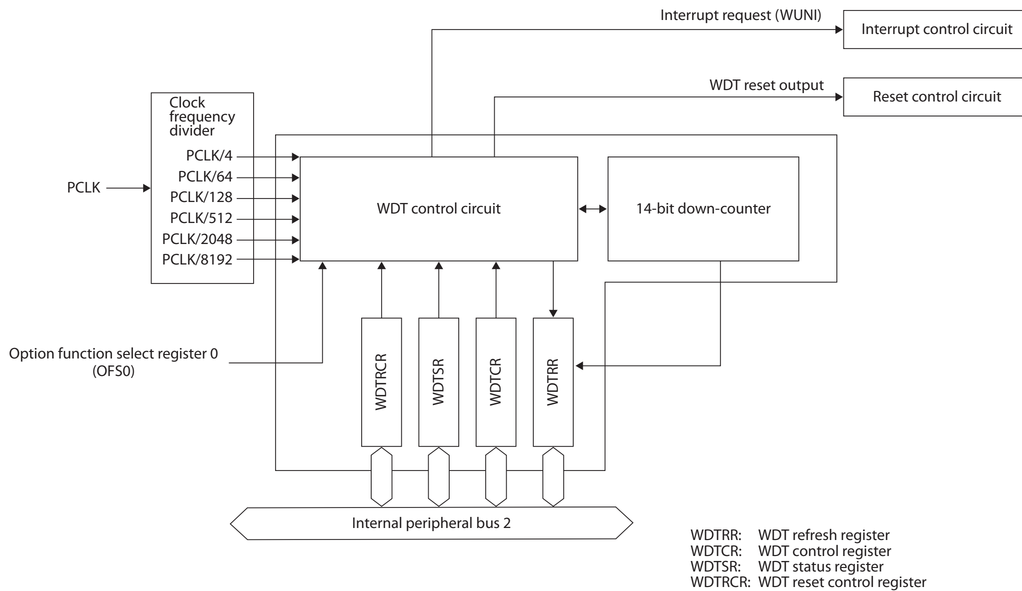
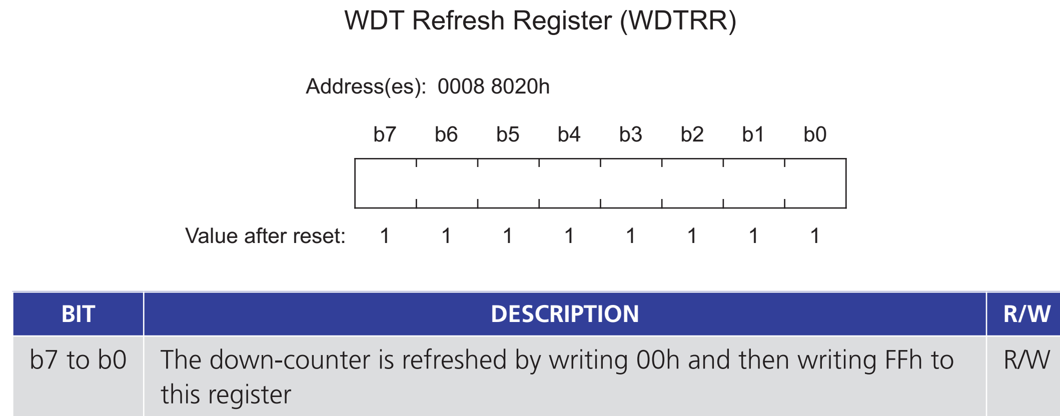


Figure 11.1 Block diagram of WDT [1], page 1081.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 278

WDT Refresh Register (WDTRR): The WDTRR is an 8-bit register which refreshes the down-counter of the WDT. The WDT down counter is refreshed by writing 00h and then writing FFh to WDTRR (refresh operation) within the refresh-permitted period.



BIT	DESCRIPTION	R/W
b7 to b0	The down-counter is refreshed by writing 00h and then writing FFh to this register	R/W

Figure 11.2 WDT Refresh Register [1], page 1082.

After the down counter has been refreshed, it starts counting down from the value selected by setting the WDT time-out period selection bits (OFS0.WDTTOPS [1:0]) in option function select register 0 in auto-start mode. In register start mode, counting down starts from the value selected by setting the time-out period selection bits (WDTCR.TOPS[1:0]) in the WDT control register by the first refresh operation after release from the reset state.

WDT Control Register (WDTCR): The WDT Control Register is a 16-bit register which is used to select the clock source for the timer and type of operation of the timer.

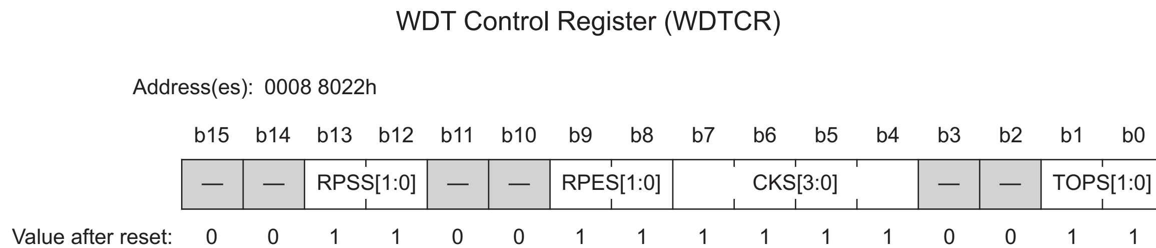


Figure 11.3 WDT Control Register [1], page 1083.

The bits of the WDTCR are set for the following functionality:

- **TOPS[1:0] Bits:** These bits are used to select the time-out period from among 1024, 4096, 8192, and 16384 cycles.
- **CKS[3:0] Bits:** These bits are used to select clock division ratio by 4, 64, 128, 512, 2048, and 8192.
- **RPES[1:0] Bits:** These bits are used to select window end position. The window start position should be a value greater than the window end position.
- **RPSS[1:0] Bits:** These bits are used to select window start position.

TABLE 11.2 Bit Description of WDTCR [1], page 1083.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b1, b0	TOPS[1:0]	Time-Out Period Selection	b1 b0	R/W
			0 0: 1,024 cycles (03FFh)	
			0 1: 4,096 cycles (0FFFh)	
			1 0: 8,192 cycles (1FFFh)	
			1 1: 16,384 cycles (3FFFh)	
b3, b2	—	Reserved	These bits are read as 0 and cannot be modified.	R
b7 to b4	CKS[3:0]	Clock Division Ratio Selection	b7 b4	R/W
			0 0 0 1: PCLK/4	
			0 1 0 0: PCLK/64	
			1 1 1 1: PCLK/128	
			0 1 1 0: PCLK/512	
			0 1 1 1: PCLK/2048	
			1 0 0 0: PCLK/8192	
			Other settings are prohibited.	
b9, b8	RPES[1:0]	Window End Position Selection	b9 b8	R/W
			0 0: 75%	
			0 1: 50%	
			1 0: 25%	
			1 1: 0% (window end position is not specified)	
b11, b10	—	Reserved	These bits are read as 0 and cannot be modified.	R
b13, b12	RPSS[1:0]	Window Start Position Selection	b13 b12	R/W
			0 0: 25%	
			0 1: 50%	
			1 0: 75%	
			1 1: 100% (window start position is not specified)	
b15, b14	—	Reserved	These bits are read as 0 and cannot be modified.	R

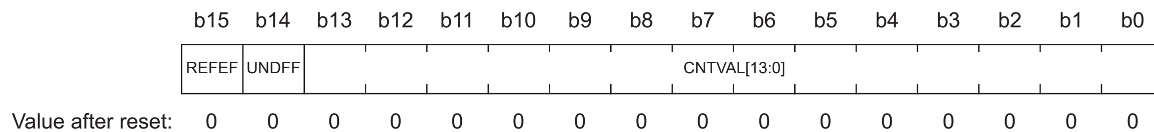
11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 280

WDT Status Register (WDTSR): The 16-bit WDT Status Register gives the status of the timer. The specific bit values are as follows:

- **CNTVAL[13:0] Bits (Down-Counter Value):** These 14 read bits give the value of the down counter, but the read value may differ from the actual count by a value of one count.
- **UNDFF Flag (Underflow Flag):** This bit is set to 1 when a down counter has underflowed. The value 0 indicates that the down counter has not underflowed. Writing 0 to the UNDFF flag sets the value to 0. Writing 1 has no effect.
- **REFEF Flag (Refresh Error Flag):** This bit is set to 1 when a refresh error has occurred. The value 0 indicates that no refresh error has occurred. REFEF flag is cleared by writing 0 to it. Writing 1 has no effect.

WDT Status Register (WDTSR)

Address(es): 0008 8024h



BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b13 to b0	CNTVAL[13:0]	Down-Counter Value	Value counted by the down-counter	R
b14	UNDFF	Underflow Flag	0: No underflow occurred 1: Underflow occurred	R(/W)*[1]
b15	REFEF	Refresh Error Flag	0: No refresh error occurred 1: Refresh error occurred	R(/W)*[1]
Note 1. Only 0 can be written to clear the flag.				

Figure 11.4 WDT Status Register [1], page 1086.

WDT Reset Control Register (WDTRCR): The WDT Reset Control Register (WDTRCR) is an 8-bit register used for control of the down counter and reset or interrupt request output. Writing to the WDT control register (WDTCR) or WDT reset control register (WDTRCR) is only possible once between the release from the reset state and the first refresh operation.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 281

WDT Reset Control Register (WDTRCR)

Address(es): 0008 8026h

	b7	b6	b5	b4	b3	b2	b1	b0
	RSTIRQS	—	—	—	—	—	—	—
Value after reset:	1	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b6 to b0	—	Reserved	These bits are read as 0 and cannot be modified.	R
b7	RSTIRQS	Reset Interrupt Request Selection	0: Non-maskable interrupt request output is enabled 1: Reset output is enabled	R/W

Figure 11.5 WDT Reset Control Register [1], page 1087.

Control over Writing to the WDTCR and WDTRCR Registers: Writing to the WDT control register (WDTCR) or WDT reset control register (WDTRCR) is only possible once between the release from the reset state and the first refresh operation.

11.4 ADVANCED CONCEPTS OF THE WATCHDOG TIMER

The WDT has two different count operations depending on the start mode selected: Register start mode or auto-start mode. Users select the WDT start mode by setting the WDT start mode selection bit OFS0.WDTSTRT in the option function select register 0.

When the register start mode OFS0.WDTSTRT bit is 1, the WDT control register (WDTCR) and WDT reset control register (WDTRCR) are enabled, and counting is started by refreshing (writing) the WDT refresh register (WDTRR).

When the auto-start mode OFS0.WDTSTRT bit is 0; the OFS0 register is enabled, and counting automatically starts after reset.

11.4.1 Register Start Mode

The WDT register start mode is selected by setting OFS0.WDTSTRT bit to 1, and the WDTCR and WDTRCR are enabled. After cancelling from the reset, set the clock division ratio, window start and end positions, time-out period in the WDTCR register, and the reset output or interrupt request output in the WDTRCR register. Then, refresh the down counter to start counting down from the value selected by setting the time-out period selection bits (WDTCR.TOPS[1:0]). Figure 11.6 shows an example of operation under these conditions.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 282

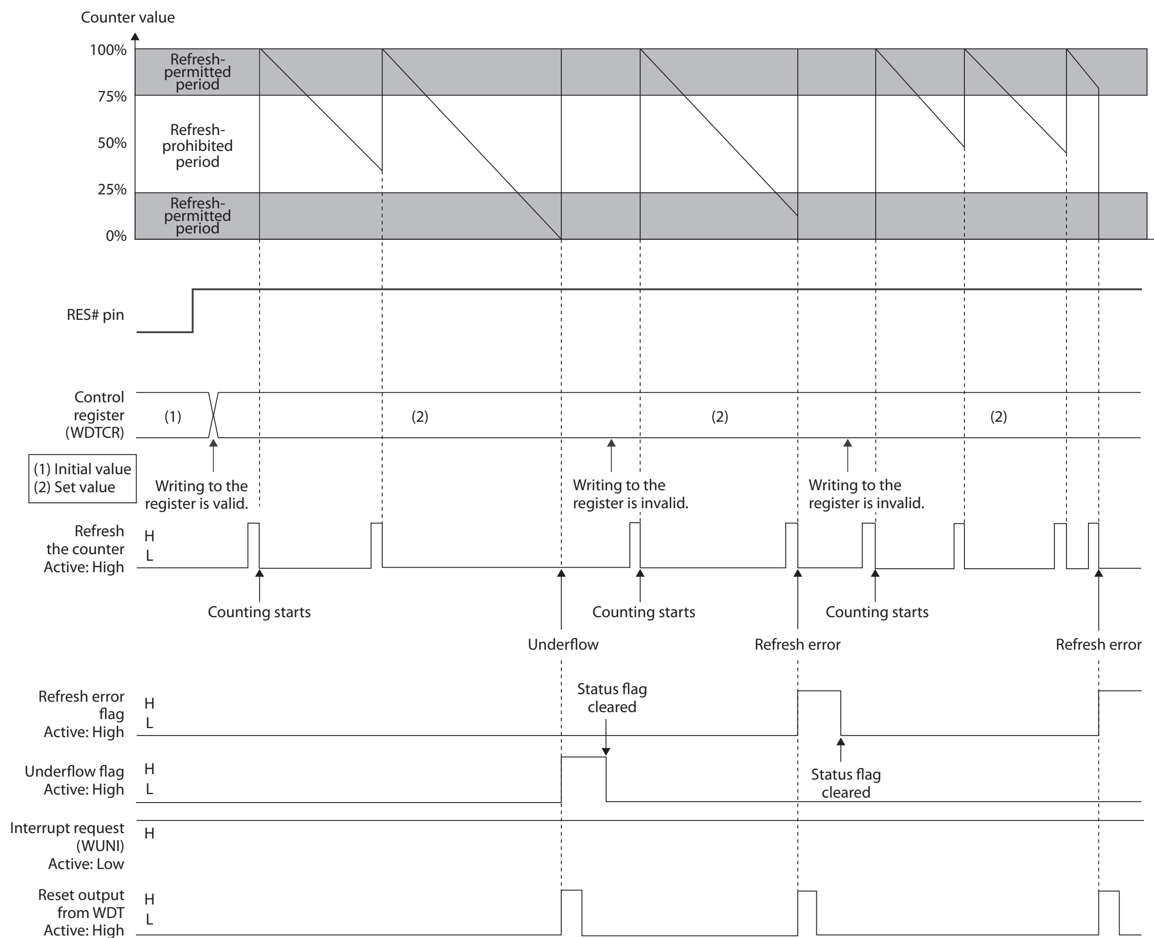


Figure 11.6 Operation example in register start mode [1], page 1089.

11.4.2 Auto-Start Mode

The WDT auto-start mode is selected by setting the OFS0.WDTSTRT bit to 0 in the option function select register 0, and when the WDT control register (WDTCR) and WDT reset control register (WDTRCR) are disabled. Within the reset state the clock division ratio, window start and end positions, time-out period, and reset output or interrupt request output are set by the option function select register (OFS0). When the reset state is canceled, the down counter automatically starts counting down from the value selected by the WDT time-out period selection bits (OFS0.WDTTOPS[1:0]).

Figure 11.7 shows an example of operation under these conditions.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 283

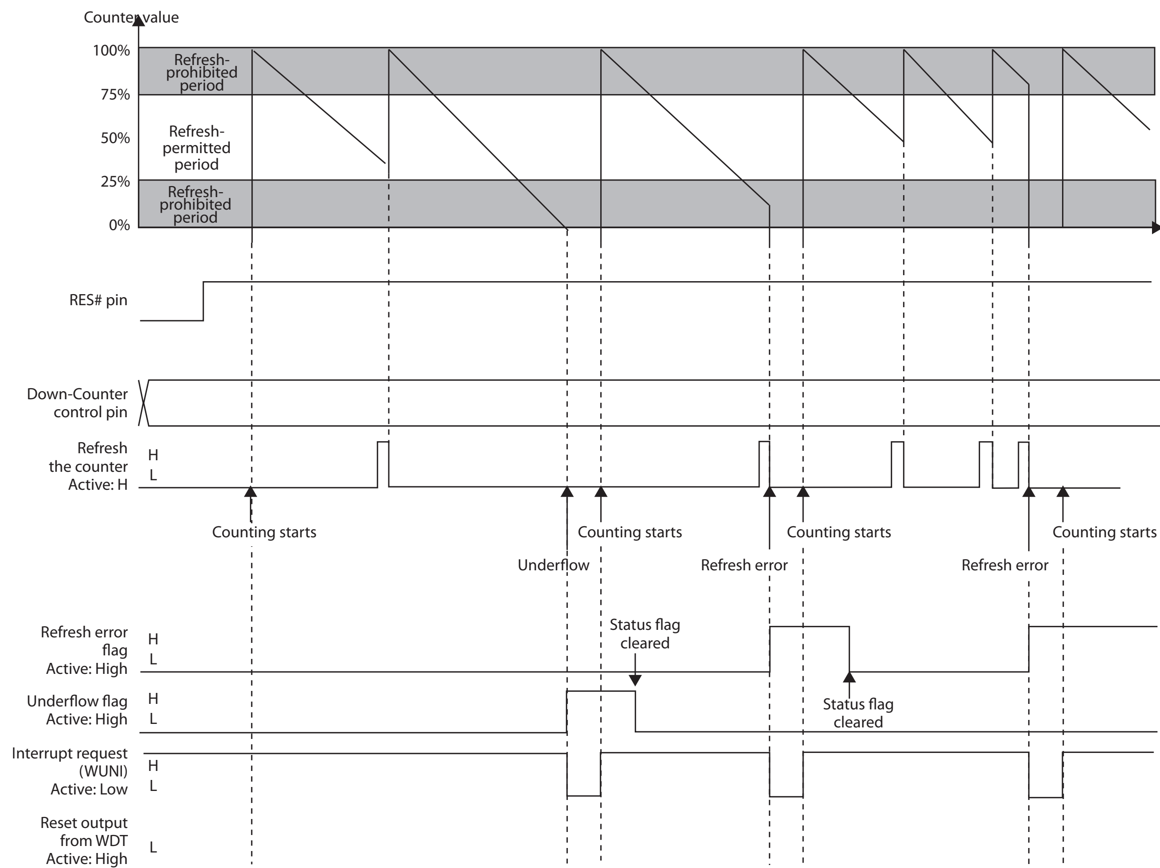


Figure 11.7 Operation in auto-start mode [1], page 1090.

Control over Writing to the WDTCR and WDTRCR Registers

Writing to the WDT control register (WDTCR) or WDT reset control register (WDTRCR) is only possible once between the release from the reset state and the first refresh operation. After a refresh operation (counting starts) or by writing to WDTCR or WDTRCR, the protection signal in the WDT becomes 1 to protect WDTCR and WDTRCR against subsequent attempts at writing. This protection is released by the reset source of the WDT. With other reset sources, the protection is not released.

11.5 INDEPENDENT WATCHDOG TIMER (IWDT)

The independent watchdog timer is used to detect a program entering into runaway conditions. The IWDT has a 14-bit down counter, and can be set up so that the chip is reset

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 284

by a reset output when counting down from the initial value causes an underflow of the counter. Alternatively, generation of an interrupt request is selectable when the counter underflows. The initial value for counting can be restored to the down counter by refreshing its value. The interval over which refreshing is possible can also be selected. Refreshing the counter during this interval will restore its initial value for counting, while attempting to refresh the counter beyond this interval leads to the output of a reset or interrupt request. The refresh interval can be adjusted and used to detect the program entering runaway conditions. The IWDT stops counting after an underflow or an attempt at refreshing the counter beyond the allowed interval. Counting is restarted by refreshing the counter when the IWDT is in register start mode. When the IWDT is in auto-start mode, counting is restarted automatically after output of the reset or interrupt request.

The IWDT has two start modes similar to WDT:

1. Auto-start mode, in which counting automatically starts after release from the reset state.
2. Register start mode, in which counting is started by refreshing the IWDT (writing to the register).

TABLE 11.3 Specifications of IWDT [1], page 1095.

ITEM	SPECIFICATIONS
Count source*[1]	IWDT-dedicated clock (IWDTCLK)
Clock division ratio	Division by 1, 16, 32, 64, 128, or 256
Counter operation	Counting down using a 14-bit down-counter
Conditions for starting the counter	■ Counting automatically starts after a reset (auto-start mode) ■ Counting is started by refreshing the IWDTRR register (writing 00h and then FFh) (register start mode)
Conditions for stopping the counter	■ Pin reset (the down-counter and other registers return to their initial values) ■ A counter underflows or a refresh error is generated Count restarts automatically in auto-start mode, or by refreshing the counter in register start mode
Window function	Window start and end positions can be specified (refresh-permitted and refresh-prohibited periods)
Reset-output sources	■ Down-counter underflows ■ Refreshing outside the refresh-permitted period (refresh error)
Interrupt request output sources	■ A non-maskable interrupt (WUNI) is generated by an underflow of the down-counter ■ When refreshing is done outside the refresh-permitted period (refresh error)

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 285

TABLE 11.3 Specifications of IWDT [1], page 1095.—*Continued*

ITEM	SPECIFICATIONS
Reading the counter value	The down-counter value can be read by the IWDTSR register.
Output signal (internal signal)	■ Reset output ■ Interrupt request output ■ Sleep-mode count stop control output
Auto-start mode (controlled by the option function select register 0 (OFS0))	■ Selecting the clock frequency division ratio after a reset (OFS0.IWDTCKS[3:0] bits) ■ Selecting the time-out period of the watchdog timer (OFS0.IWDTTOPS[1:0] bits) ■ Selecting the window start position in the watchdog timer (OFS0.IWDTRPSS[1:0] bits) ■ Selecting the window end position in the watchdog timer (OFS0.IWDTRPES[1:0] bits) ■ Selecting the reset output or interrupt request output (OFS0.IWDTRSTIRQS bit) ■ Selecting the down-count stop function at transition to sleep mode, software standby mode, deep software standby mode, or all-module clock stop mode (OFS0.IWDTSLCSTP bit)
Register start mode (controlled by the IWDT registers)	■ Selecting the clock frequency division ratio after refreshing (IWDTCR.CKS[3:0] bits) ■ Selecting the time-out period of the watchdog timer (IWDTCR.TOPS[1:0] bits) ■ Selecting the window start position in the watchdog timer (IWDTCR.RPSS[1:0] bits) ■ Selecting the window end position in the watchdog timer (IWDTCR.RPES[1:0] bits) ■ Selecting the reset output or interrupt request output (IWDTRCR.RSTIRQS bit) ■ Selecting the down-count stop function at transition to sleep mode, software standby mode, deep software standby mode, or all-module clock stop mode (IWDTCSTPR.SLCSTP bit)
Note	1. Set the count source so that the peripheral module clock frequency ≥ 4 x (the count source clock divided frequency).

To use the IWDT, two clocks (peripheral clock (PCLK) and IWDT-dedicated clock (IWDTCLK)) should be supplied so that the IWDT continues to function when the PCLK stops (see Figure 11.8). The bus interface and registers operate with PCLK, and the 14-bit down counter and control circuits operate with IWDTCLK. Signal lines between the blocks operating with the PCLK and IWDTCLK are connected through synchronization circuits.

11.5.1 Register Description

IWDT Refresh Register (IWDTRR): The IWDTRR refreshes the down counter of the IWDT. The down counter of the IWDT is refreshed by writing 00h and then writing FFh to IWDTRR (refresh operation) within the refresh-permitted period. After the down counter has been refreshed, it starts counting down from the value selected by the IWDT time-out period selection bits (OFS0.IWDTTOPS[1:0]) in option function select register 0 in

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 286

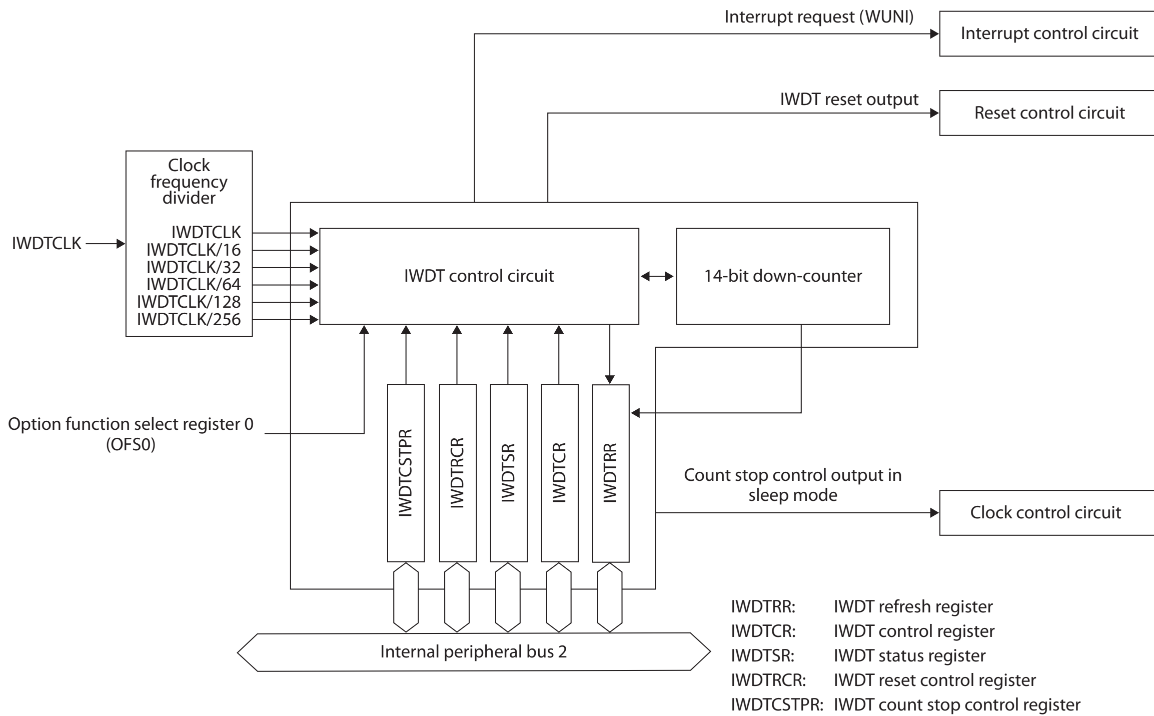
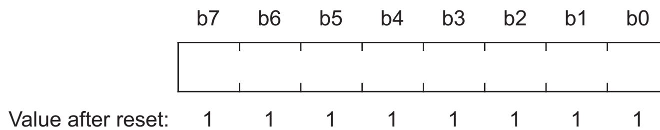


Figure 11.8 Block diagram of IWDT [1], page 1096.

auto-start mode. In register start mode, counting down starts from the value selected by setting the time-out period selection bits (TOPS[1:0]) in the IWDT control register (IWDTCR) in the first refresh operation after release from the reset state. When 00h is written, the read value is 00h. When a value other than 00h is written, the read value is FFh.

IWDT Refresh Register (IWDTRR)

Address(es): 0008 8030h



BIT	DESCRIPTION	R/W
b7 to b0	The down-counter is refreshed by writing 00h and then writing FFh to this register	R/W

Figure 11.9 Bit description of IWDTRR [1], page 1097.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 287

IWDT Control Register (IWDTCR): The IWDTCR is a 16-bit register. There are some restrictions on writing to the IWDTCR register. In auto-start mode, the settings in the IWDTCR register are disabled, and the settings in the option function select register 0 (OFS0) are enabled. The bit setting made to the IWDTCR register can also be made in the OFS0 register.

IWDT Control Register (IWDTCR)

Address(es): 0008 8032h

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	RPSS[1:0]		—	—	RPES[1:0]		CKS[3:0]				—	—	TOPS[1:0]	
Value after reset:	0	0	1	1	0	0	1	1	1	1	1	1	0	0	1	1

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b1, b0	TOPS[1:0]	Time-Out Period Selection	b1 b0	R/W
			0 0: 1,024 cycles (03FFh)	
			0 1: 4,096 cycles (0FFFh)	
			1 0: 8,192 cycles (1FFFh)	
			1 1: 16,384 cycles (3FFFh)	
b3, b2	—	Reserved	These bits are read as 0 and cannot be modified.	R
b7 to b4	CKS[3:0]	Clock Division Ratio Selection	b7 b4	R/W
			0 0 0 0: IWDTCLK	
			0 0 1 0: IWDTCLK/16	
			0 0 1 1: IWDTCLK/32	
			0 1 0 0: IWDTCLK/64	
			1 1 1 1: IWDTCLK/128	
			0 1 0 1: IWDTCLK/256	
			Other settings are prohibited.	
b9, b8	RPES[1:0]	Window End Position Selection	b9 b8	R/W
			0 0: 75%	
			0 1: 50%	
			1 0: 25%	
			1 1: 0% (window end position is not specified)	
b11, b10	—	Reserved	These bits are read as 0 and cannot be modified.	R
b13, b12	RPSS[1:0]	Window Start Position Selection	b13 b12	R/W
			0 0: 25%	
			0 1: 50%	
			1 0: 75%	
			1 1: 100% (window start position is not specified)	
b15, b14	—	Reserved	These bits are read as 0 and cannot be modified.	R

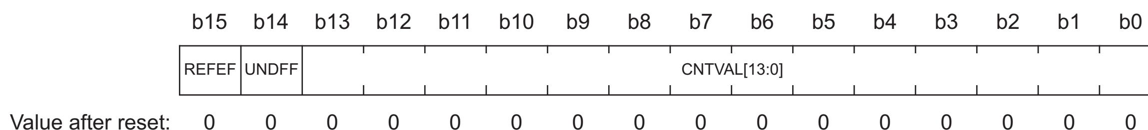
Figure 11.10 Bit description of IWDTCR [1], page 1098.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 288

IWDT Status Register (IWDTSR): The IWDTSR is a 16-bit register and is initialized by the reset source of the IWDT. IWDTSR is not initialized by other reset sources.

IWDT Status Register (IWDTSR)

Address(es): 0008 8034h



BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b13 to b0	CNTVAL[13:0]	Down-Counter Value	Value counted by the down-counter	R
b14	UNDFF	Underflow Flag	0: No underflow occurred 1: Underflow occurred	R(/W)*1
b15	REFEF	Refresh Error Flag	0: No refresh error occurred 1: Refresh error occurred	R(/W)*1
Note 1. Only 0 can be written to clear the flag.				

Figure 11.11 Bit description of IWDTSR [1], page 1101.

- **CNTVAL[13:0] Bits (Down-Counter Value):** Read these bits to confirm the value of the down counter, but note that the read value may differ from the actual count by a value of one count.
- **UNDFF Flag (Underflow Flag):** Read this bit to confirm whether or not an underflow has occurred in the down counter. The value 1 indicates that the down counter has underflowed. The value 0 indicates that the down counter has not underflowed. Write 0 to the UNDFF flag to set the value to 0. Writing 1 has no effect.
- **REFEF Flag (Refresh Error Flag):** Read this bit to confirm whether or not a refresh error (performing a refresh operation during a refresh-prohibited period) has occurred. The value 1 indicates that a refresh error has occurred. The value 0 indicates that no refresh error has occurred. Write 0 to the REFEF flag to set the value to 0. Writing 1 has no effect.

IWDT Reset Control Register (IWDTRCR): The IWDTRCR is an 8-bit reset control register. There are some restrictions on writing to the IWDTRCR register. In auto-start mode, the IWDTRCR register settings are disabled, and the settings in the option function select register 0 (OFS0) enabled. The bit setting mode to the IWDTRCR register can also be made in the OFS0 register.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 289

IWDT Reset Control Register (IWDTRCR)

Address(es): 0008 8036h

	b7	b6	b5	b4	b3	b2	b1	b0
	RSTIRQS	—	—	—	—	—	—	—
Value after reset:	1	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b6 to b0	—	Reserved	These bits are read as 0 and cannot be modified.	R
b7	RSTIRQS	Reset Interrupt Request Selection	0: Non-maskable interrupt request output is enabled 1: Reset output is enabled	R/W

Figure 11.12 Bit description of IWDTRCR [1], page 1102.

IWDT Count Stop Control Register (IWDTCSTPR): The IWDTCSTPR is an 8-bit register. In auto-start mode, the settings in the IWDTCSTPR register are ignored, and the settings in the option function select register 0 (OFS0) take effect. The bit setting mode to the IWDTCSTPR register can also be made in the OFS0 register.

IWDT Count Stop Control Register (IWDTCSTPR)

Address(es): 0008 8038h

	b7	b6	b5	b4	b3	b2	b1	b0
	SLCSTP	—	—	—	—	—	—	—
Value after reset:	1	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b6 to b0	—	Reserved	These bits are read as 0 and cannot be modified.	R
b7	SLCSTP	Sleep-Mode Count Stop Control	0: Count stop is disabled 1: Count is stopped at a transition to sleep mode, software standby mode, deep software standby mode, or allmodule clock stop mode	R/W

Figure 11.13 Bit description of IWDTCSTPR [1], page 1102.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 290

11.6 EXAMPLES

The following code explains the setting of the watchdog timer mode:

```
1. void wdt_init(void) {
2.    WDT.WDTRCR.BYTE = 0x80;
3.    ICU.NMIER.BIT.WDTEN = 1;
4.    WDT.WDTCR.WORD = 0x3383;
5.    WDT.WDTSR.WORD = 0x0000;
6. }
```

Code Explanation: In line 1 the watchdog timer function is initialized. In line 2 the WDTRCR register is set to a value of 0x80 and Reset (and not NMI) output is enabled. In line 3 the Non-Maskable Interrupt (NMI) with the watchdog is used by unmasking it using the Non-Maskable Interrupt Enable Register (NMIER). In line 4 the WDTC register is used to set PCLK = 48 MHz and the timeout period = 134,217,728/48,000,000 = 2.796 seconds. In line 5 the WDT Status register is set to zero for clearing the refresh error and underflow flags.

The watchdog timer needs to be refreshed to keep it from counting down to zero and underflowing. Historically, this was called “kicking the dog,” but we now use the more positive phrase called “feeding the dog.” A simple function to do this is shown in the following code:

```
1. void wdt_feed_watchdog(void) {
2.    WDT.WDTRR = 0x00;
3.    WDT.WDTRR = 0xFF;
4. }
```

In the above code we are feeding the watchdog by writing 00h and then writing FFh to WDTRR within the refresh-permitted period. The watchdog timer’s registers are intentionally more difficult to write to than most other processor registers. This makes it harder for a runaway rogue program to disable the watchdog.

```
1. void main() {
2.    int i;
3.    ENABLE_LEDS;
4.    while(1) {
5.       Init_WDT(void);
6.       ALL_LEDS_OFF();
7.       for(i = 0; i < 10000000; i++) {
8.          i--;  //This is a "infinite loop" software bug
9.       }
```

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 291


```
10.        ALL_LEDS_ON();
11.        WDT.WDTRR = 0x00;
12.        WDT.WDTRR = 0xFF;
13.    }
14. }
```

In the above program, the desired effect would be for the LEDs to switch on and switch off with some delay, but due to the software bug the program enters into an infinite loop which eventually causes the overflow of the watchdog timer—and the overflow causes the internal reset of the microcontroller. If there is no infinite loop, the count of the watchdog timer is reset frequently to prevent the overflow.

11.7 BASIC CONCEPTS OF BROWNOUT CONDITION

11.7.1 How Brownout Occurs

A brownout condition in a microcontroller occurs when the supply voltage for the microcontroller temporarily goes below a threshold value (V_{det}). Below this threshold value, the microcontroller may malfunction. There is also a blackout condition in which the microcontroller will have a total loss of electricity. In a brownout condition, some operations may work, but in a blackout condition none of the operations will be active.

11.7.2 Automatically Detecting a Brownout Condition

The main purpose of automatically detecting the brownout condition is to prevent the corruption of processor critical information. Whenever the brownout condition is detected, the internal reset from the voltage detection circuit should keep the processor in reset condition until the voltage increases above the threshold value (V_{det}).

The RX63N MCU has three low-voltage detection (LVD) circuits based on analog comparators which are voltage detection 0, voltage detection 1, and voltage detection 2. The voltage detection circuit (LVDA) monitors the voltage level input to the VCC pin using a program. The multiple detection levels allow the user to run interrupt service routines which could, for example, save critical data to EEPROM and properly shutdown the system before MCU goes into reset.

In voltage detection 0, whether to enable or disable the reset of voltage monitoring 0 can be selected after the reset using the option function select register 1 (OFS1).

In voltage detection 1 and voltage detection 2, the detection voltage is set using the voltage detection level select register (LVDLVLR). Reset of voltage monitoring 0, reset/interrupt of voltage monitoring 1, and reset/interrupt of voltage monitoring 2 can be used.

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 292

TABLE 11.4 Voltage Detection Circuit Specifications: [1], Page 227.

ITEM		VOLTAGE MONITORING 0	VOLTAGE MONITORING 1	VOLTAGE MONITORING 2
VCC monitoring	Monitored voltage	Vdet0	Vdet1	Vdet2
	Detected event	Voltage drops past Vdet0	Voltage rises or drops past Vdet1	Voltage rises or drops past Vdet2
	Detection voltage	One level fixed	Specify voltage using LVDLVLR.LVD1LVLR[3:0] bits	Specify voltage using LVDLVLR.LVD2LVL[3:0] bits
	Monitoring flag	None	LVD1SR.LVD1MON flag: Monitors whether voltage is higher or lower than Vdet1	LVD2SR.LVD2MON flag: Monitors whether voltage is higher or lower than Vdet2
			LVD1SR.LVD1DET flag: Vdet1 passage detection	LVD2SR.LVD2DET flag: Vdet2 passage detection
Process upon voltage detection	Reset	Voltage monitoring 0 reset	Voltage monitoring 1 reset	Voltage monitoring 2 reset
		Reset when Vdet0 > VCC CPU restart after specified time with VCC > Vdet0	Reset when Vdet1 > VCC CPU restart timing selectable: after specified time with VCC > Vdet1 or Vdet1 > VCC	Reset when Vdet2 > VCC CPU restart timing selectable: after specified time with VCC > Vdet2 or Vdet2 > VCC
	Interrupt	No interrupt	Voltage monitoring 1 interrupt	Voltage monitoring 2 interrupt
			Non-maskable interrupt	Non-maskable interrupt
			Interrupt request issued when Vdet1 > VCC and VCC > Vdet1 or either	Interrupt request issued when Vdet2 > VCC and VCC > Vdet2 or either
Digital filter	Enable/ Disable switching	Digital filter function not available	Available	Available
	Sampling time	—	1/n LOCO frequency × 2 (n: 1, 2, 4, 8)	1/n LOCO frequency × 2 (n: 1, 2, 4, 8)

11.ES_Conrad_RX63N_Advanced_CH11.qxd:RX63N Advanced 3/4/14 11:58 AM Page 293

11.8 RECAP

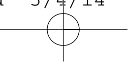
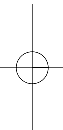
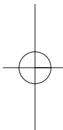
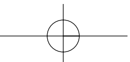
A watchdog timer is one of the tools in the microcontroller, which gets the system out of unexpected errors or infinite loops. The watchdog timer count is refreshed frequently so that the overflow does not occur in the general flow of the program. When the overflow occurs, it implies that the timer count has been not refreshed and the program is not functioning in the way it should. So the watchdog timer resets the system whenever it overflows.

11.9 REFERENCES

[1] Renesas Electronics, Inc. (February, 2013). *RX63N Group, RX631 Group User's Manual: Hardware,* Rev 1.60.

11.10 EXERCISES

1. List the registers used to set up the WDT and describe the importance of the registers.
2. List the registers used to set up the IWDT and describe the importance of the registers.
3. What registers are used to write data into WDTTRCR, WDTTCNT, and WDTTCSR registers?
4. Draw a flowchart explaining when and how a WDT resets the system.
5. Write pseudo code to reset the microcontroller when a brownout condition occurs.
6. Write a C code to set WDT to operate with a 48 MHz clock.
7. Write the Code to set up the WDT to reset the microcontroller every 1 s.
8. Write the Code to set up the IWDT to reset the microcontroller every 1 s.

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 295

Processor Settings and Running in Low Power Modes

12.1 LEARNING OBJECTIVES

In today's world of battery-operated devices, the proper use of the low-power/sleep modes provided in most embedded microcontrollers is critical in design success. The RX63N microcontroller has several functions, such as the capability to switch clock frequencies and the ability to stop modules to reduce its power consumption. Intelligent switching between a normal mode and a low power mode can significantly improve the battery life.

In this chapter the reader will learn:

- The startup process of the RX63N microcontroller.
- The general concept of low power consumption.
- How to switch the microcontroller into various low power modes.

12.2 RX63N STARTUP PROCESS

When the RX63N microcontroller starts in power-on mode, several initial settings must be configured in order to ensure its correct operation. This RX63N Demonstration Kit board (YRDKRX63N) is shipped with a user boot program that is stored in on-chip flash memory and set up with factory predefined specifications. The processor can be set to several operating modes based on pins 1 and 2 of the Switch 5 on the board. The available modes are single-chip, USB boot mode, and user boot mode. The RX63N is initially set to boot into single chip mode, which initially runs the startup code [1].

When the board is powered-on, several functions are executed based on the startup code prior to executing the main function. As shown in Figure 12.1, some of the steps include setting the clock, interrupts, and time capture control registers.

Upon powering up or resetting the board, a program counter points to the location in memory containing the functions defined in resetprg.c code, which can be found in the High-performance Embedded Workshop (HEW) inside all RX63N projects (the tool chain e2 studio has a similar mechanism). This program configures the microcontroller prior to the main program call.

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 296

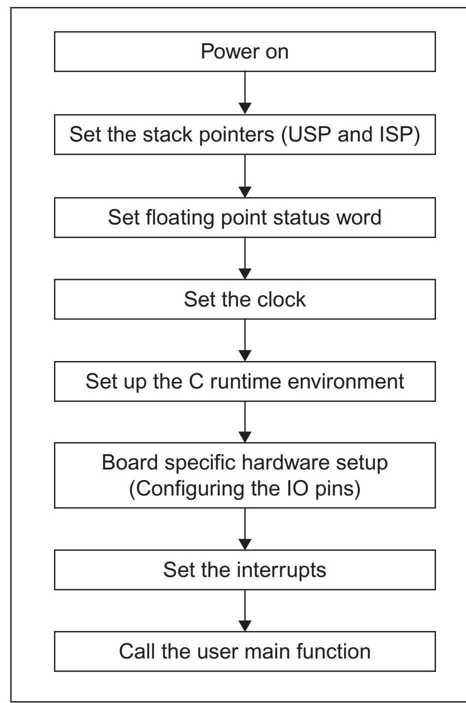


Figure 12.1 Initial settings performed by startup code before calling user main() function.

Consider the start-up function starting on the following page. Lines 12 through 22 configure the floating point unit, by initializing the FPSW register. Line 24 calls the operating frequency setup function. Lines 50 through 81 initialize the clocks based on the input clock frequency of 12 MHz. The default frequencies are shown in Table 12.1.

TABLE 12.1 Default Clock Frequencies found in resetprg.c.

PLL Frequency	192 MHz
Internal Clock Frequency	96 MHz
Peripheral Clock Frequency	48 MHz
USB Clock Frequency	48 MHz
External Bus Clock Frequency	24 MHz

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 297

Lines 25 and 26 initialize the C runtime environment and the I/O library respectively. Line 30 calls the hardware setup function shown in the hwsetup.c code below. In the hardware setup code, all the ports, interrupts, and peripherals are configured/enabled. Note that leaving unused IO pins floating should be avoided. Keeping them unused may lead to false recognition of pin as input pin due to induced noise. These pins should be handled as described in the hardware manual [1, p.630]. Lines 30 through 32 change the microcontroller from supervisor mode to user mode in order to protect some of the setup registers. Lines 38 through 41 enable the bus error interrupt in order to catch any attempts to access illegal/reserved areas of memory. Finally, in line 43 the main function is called, in which you should never return due to the infinite loop required in the main function.

```
 1. #include <machine.h>
 2. void main(void);
 3. static void operating_frequency_set(void);
 4.
 5. void PowerON_Reset_PC(void) {
 6.    #if RENESAS_VERSION >= 0x01010000
 7.       set_intb((void *) sectop("C$VECT"));
 8.    #else
 9.       set_intb((unsigned long) sectop("C$VECT"));
10.    #endif
11.
12.    #ifdef ROZ
13.       #define _ROUND 0x00000001
14.    #else
15.       #define _ROUND 0x00000000
16.    #endif
17.    #ifdef   DOFF
18.       #define _DENOM 0x00000100
19.    #else
20.       #define _DENOM 0x00000000
21.    #endif
22.    set_fpsw(FPSW_init | _ROUND | _DENOM);
23.
24.    operating_frequency_set();
25.    _INITSCT();
26.    _INIT_IOLIB();
27.    hardware_setup();
28.    nop();
```

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 298


```
29.    set_psw(PSW_init);
30.    #if RUN_IN_USER_MODE == 1
31.       #if RENESAS_VERSION >= 0x01010000
32.       chg_pmusr() ;
33.       #else
34.          Change_PSW_PM_to_UserMode();
35.       #endif
36.    #endif
37.
38.    IR(BSC,BUSERR) = 0;
39.    IPR(BSC,BUSERR) = 0x0F;
40.    IEN(BSC,BUSERR) = 1;
41.    BSC.BEREN.BIT.IGAEN = 1;
42.
43.    main();
44.
45.    _CLOSEALL();
46.    while(1){};
47. }
48.
49. void operating_frequency_set(void){
50.    volatile unsigned int i;
51.    SYSTEM.PRCR.WORD = 0xA50B;
52.    SYSTEM.SOSCCR.BYTE = 0x00;
53.    SYSTEM.MOSCWTCR.BYTE = 0x0D;
54.    SYSTEM.PLLWTCR.BYTE = 0x04;
55.    SYSTEM.PLLCR.WORD = 0x0F00;
56.    SYSTEM.MOSCCR.BYTE = 0x00;
57.    SYSTEM.PLLCR2.BYTE = 0x00;
58.    for(i = 0; i < 0x168; i++){
59.       nop() ;
60.    }
61.    SYSTEM.SCKCR2.WORD = 0x0031;
62.    SYSTEM.SCKCR3.WORD = 0x0400;
63.    SYSTEM.PRCR.WORD = 0xA500;
64. }
65.
66. #if RUN_IN_USER_MODE == 1
67. #if RENESAS_VERSION < 0x01010000
68.
```

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 299


```
69. static void Change_PSW_PM_to_UserMode(void) {
70.     MVFC PSW,R1
71.     OR #00100000h,R1
72.     PUSH.L R1
73.     MVFC   PC,R1
74.     ADD    #10,R1
75.     PUSH.L R1
76.     RTE
77.     NOP
78.     NOP
79. }
80.     #endif
81.     #endif
```

The following code is an excerpt taken from the hwsetup.c file, which is included in all projects that are created using HEW. As you can see in lines 3–5, the hardware setup function calls other functions that configure the I/O ports and interrupts, and enables the peripheral modules. If certain peripherals or interrupts need to be enabled on restart, the respective functions in the following code could be altered by adding the correct code.

```
 1. #include <stdint.h>
 2. void hardware_setup(void) {
 3.     output_ports_configure();
 4.     interrupts_configure();
 5.     peripheral_modules_enable()
 6. }
 7. void output_ports_configure(void) {
 8.     SYSTEM.PRCR.WORD = 0xA50B;
 9.     MPC.PWPR.BIT.B0WI = 0;
10.     MPC.PWPR.BIT.PFSWE = 1;
11.     MSTP(EDMAC) = 0;
12.     PORT0.PODR.BYTE = 0x00;
13.     PORT0.PDR.BYTE = 0x2F;
14.     .
15.     .
16.     .
17.     .
18.     PORTJ.PODR.BYTE = 0x08;
19.     PORTJ.PDR.BYTE = 0x28;
20. }
```

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 300


```
21. void interrupts_configure(void){};
22. void peripheral_modules_enable(void){};
```

12.3 BASIC CONCEPTS OF LOW POWER CONSUMPTION

12.3.1 Introduction to the Concept of Low Power Consumption

Power consumption is an important aspect in the design of any embedded system. Devices that are remotely operated with little or no human interaction are unlikely to have easily replaceable batteries. For example, a radio collar used to track a wild animal's location would not have an easily replaceable battery but would be tuned to last as long as possible to allow maximum use of the collar. The inability to either recharge or replace the batteries in some portable devices has highlighted the need for reducing power consumption when possible while still allowing the system to accomplish the given tasks. The idea is to perform the desired tasks as quickly as possible or with as little power consumption as possible.

The power consumption of a processor over a given interval is the sum of the power consumed in both active and standby modes. Most microcontrollers are based on CMOS logic in which power is consumed when the transistors are switched. The average power consumed when switching a microcontroller transistor can be defined in terms of the switching frequency (f) as $P = fCV^2$. This formula shows that the power consumption depends upon the switching frequency (clock), load capacitance, and the supply voltage [2]. Leakage also plays a role in power consumption. This power loss is much smaller than the losses consumed by switching frequencies.

Embedded systems normally have peripherals that perform some of the work for them and, upon completion, give the information to the processor to compute the results and act on them. During this time of waiting for a peripheral to respond, the processor can be put to sleep to help save power and then awake when the results are ready. Several peripherals often remain unused for a given instance of time, and they can be powered off. If the peripheral is not being used, turning off its power extends battery life. An equivalent concept can be seen in today's laptop computers, which comes with the feature of sleep mode or hibernation. In these modes, all the processes are suspended and the states of all the processes as well as the OS are stored in laptop's RAM/Hard Drive. The operation screen, hard drive, and other peripherals are suspended. After pressing any key board key or moving the mouse, the laptop will "wake up" from the sleep mode restoring its previous state from RAM/Hard Drive.

The RX63N microcontroller has several functions for reducing power consumption, including switching off clock signals to reduce power consumption; BCLK and SDCLK

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 301

output control functions; functions for stopping modules (peripherals); and functions for low power consumption in normal operation and transitions to and from low power consumption states [1].

12.3.2 Various Processor Settings to Achieve Low Power Consumption

In order to reduce power consumption in an embedded system, the designer must determine the minimum required speed and adjust the clock accordingly. The RX63N microcontroller has several different clocks that can be altered. When the SCKCR.FCK[3:0], ICK[3:0], BCK[3:0], PCKA[3:0], and PCKB[3:0] bits are set, the clock frequency for each related function changes. The CPU, DMAC, DTC, ROM, and RAM operate on the clock specified by the ICK[3:0] bits. Peripheral modules operate on the clock specified by the PCKA[3:0] or the PCKB[3:0] bits. The flash memory interface, either the ROM or E2 DataFlash, operates on the clock specified by the FCK[3:0] bits. The external bus operates on the clock specified by the BCK[3:0] bits [1]. Refer to the following section for the clock division settings for each register. Several other modes and functions are discussed in the following section, including the module stop function and various other lower operating power consumption control modes.

12.3.3 Overview: Various Low Power Consumption Modes

The Renesas RX63N microcontroller has four different low power modes. They are sleep mode, all module clock stop mode, software standby mode, and deep software standby mode—listed in descending order of power consumption. This processor can enter into any of these modes by setting certain registers and then executing a WAIT instruction. It can come out of a low power mode through the use of interrupts. Not all interrupts can cancel all low power modes. Refer to Section 12.4 for additional information about each mode.

12.3.4 Processor Settings: Various Registers Used

The following pages from the RX63N hardware manual [1] summarize the most important start-up registers to configure.

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 302

Standby Control Register (SBYCR)

Address(es): 0008 000Ch

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	SSBY	OPE	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Value after reset:	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b13 to b0	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b14	OPE	Output Port Enable	0: In software standby mode or deep software standby mode, the address bus and bus control signals are set to the high-impedance state. 1: In software standby mode or deep software standby mode, the address bus and bus control signals retain the output state.	R/W
b15	SSBY	Software Standby	0: Shifts to sleep mode or all-module clock stop mode after the WAIT instruction is executed 1: Shifts to software standby mode after the WAIT instruction is executed	R/W

OPE Bit (Output Port Enable)

The OPE bit specifies whether to retain the output of the address bus and bus control signals (CS0# to CS7#, RD#, WR0# to WR3#, WR#, BC0# to BC3#, ALE, CKE, SDCS#, RAS#, CAS#, WE#, and DQM0 to DQM3) in software standby mode or deep software standby mode, or to set the output to the high-impedance state.

SSBY Bit (Software Standby)

The SSBY bit specifies the transition destination after the WAIT instruction is executed.

When the SSBY bit is set to 1, the LSI enters software standby mode after execution of the WAIT instruction. When the LSI returns to normal mode after an interrupt has initiated release from software standby mode, the SSBY bit remains 1.

Write 0 to this bit to clear it.

When the oscillation stop detection function enable bit (OSTDCR.OSTDE) is 1, setting of the SSBY bit is invalid. Even if the SSBY bit is 1, the LSI will enter sleep mode or all module clock stop mode on execution of the WAIT instruction.

Figure 12.2 Standby Control Register Settings [2], p.281.

12.3.5 Functions: Description of Operation in Different Functions

Multi-clock functionality is the ability to utilize several different clocks, and therefore different clock speeds, in order to help lower power consumption. Peripherals in the RX63N operate on one of the two clocks, either PCLKA or PCLKB. Depending on the peripherals used and the speed needed, the board has the ability to run peripherals at the same time but possibly at different speeds, thus saving power by reducing clock switching. A clock is available for the memory unit (RAM and ROM) and the CPU, as well as a clock for flash memory and the external bus. Without all these different clocks, all the units would run at the highest speed, even though some of the peripherals and/or memory accesses may not need to operate at the highest speed, thus wasting unnecessary power. Refer to Section 12.4.1 for an example of setting some of the clocks to operate at different speeds.

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 303

Module Stop Control Register A (MSTPCRA)

Address(es): 0008 0010h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	ACSE	—	MSTPA 29	MSTPA 28	MSTPA 27	—	—	MSTPA 24	MSTPA 23	—	—	—	MSTPA 19	—	MSTPA 17	—
Value after reset:	0	1	0	0	0	1	1	0	1	1	1	1	1	1	1	1

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	MSTPA 15	MSTPA 14	MSTPA 13	MSTPA 12	MSTPA 11	MSTPA 10	MSTPA 9	—	—	—	MSTPA 5	MSTPA 4	—	—	—	—
Value after reset:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b3 to b0	—	Reserved	These bits are read as 1. The write value should be 1.	R/W
b4	MSTPA4	8-Bit Timer 3/2 (Unit 1) Module Stop	Target module: TMR3/TMR2 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b5	MSTPA5	8-Bit Timer 1/0 (Unit 0) Module Stop	Target module: TMR1/TMR0 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b8	—	Reserved	This bit is read as 1. The write value should be 1.	R/W
b9	MSTPA9	Multifunction Timer Pulse Unit 2 Module Stop	Target module: MTU (MTU0 to MTU5) 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b10	MSTPA10	Programmable Pulse Generator (Unit 1) Module Stop	Target module: PPG1 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b11	MSTPA11	Programmable Pulse Generator (Unit 0) Module Stop	Target module: PPG0 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b12	MSTPA12	16-Bit Timer Pulse Unit 1 (Unit 1) Module Stop	Target module: TPU unit 1 (TPU6 to TPU11) 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b13	MSTPA13	16-Bit Timer Pulse Unit 0 (Unit 0) Module Stop	Target module: TPU unit 0 (TPU0 to TPU5) 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b14	MSTPA14	Compare Match Timer (Unit 1) Module Stop	Target module: CMT unit 1 (CMT2, CMT3) 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W

Figure 12.3 Module Stop Control Register A Settings [1], pages 282–283.—*Continues*

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 304

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b15	MSTPA15	Compare Match Timer (Unit 0) Module Stop	Target module: CMT unit 0 (CMT0, CMT1)	R/W
			0: The module-stop state is canceled	
			1: Transition to the module-stop state is made	
b16	—	Reserved	This bit is read as 1. The write value should be 1.	R/W
b17	MSTPA17	12-bit A/D Converter (Unit 1) Module Stop	Target module: S12AD1	R/W
			0: The module-stop state is canceled	
			1: Transition to the module-stop state is made	
b18	—	Reserved	This bit is read as 1. The write value should be 1.	R/W
b19	MSTPA19	D/A Converter Module Stop	Target module: DA	R/W
			0: The module-stop state is canceled	
			1: Transition to the module-stop state is made	
b22 to b20	—	Reserved	These bits are read as 1. The write value should be 1.	R/W
b23	MSTPA23	10-bit A/D Converter Module Stop	Target module: AD	R/W
			0: The module-stop state is canceled	
			1: Transition to the module-stop state is made	
b24	MSTPA24	Module Stop A24	Writing to and reading from this bit is enabled. When a transition to all-module clock stop mode is made, be sure that 1 has been written to this bit.	R/W
b26, b25	—	Reserved	These bits are read as 1. The write value should be 1.	R/W
b27	MSTPA27	Module Stop A27	Writing to and reading from this bit is enabled. When a transition to all-module clock stop mode is made, be sure that 1 has been written to this bit.	R/W
b28	MSTPA28	DMA Controller/ Data Transfer Controller Module Stop	Target module: DMAC/DTC	R/W
			0: The module-stop state is canceled	
			1: Transition to the module-stop state is made	
b29	MSTPA29	EXDMA Controller Module Stop	Target module: EXDMAC	R/W
			0: The module-stop state is canceled	
			1: Transition to the module-stop state is made	
b30	—	Reserved	This bit is read as 1. The write value should be 1.	R/W
b31	ACSE	All-Module Clock Stop Mode Enable	0: All-module clock stop mode is disabled	R/W
			1: All-module clock stop mode is enabled	

ACSE Bit (All-Module Clock Stop Mode Enable)

The ACSE bit enables or disables a transition to all-module clock stop mode.

With the ACSE bit set to 1, when the CPU executes the WAIT instruction with the SBYCR.SSBY bit, MSTPCRA, MSTPCRB, and MSTPCRC satisfying specified conditions, the LSI enters all-module clock stop mode. For details, see section 11.6.2, All-Module Clock Stop Mode.

Whether to stop the 8-bit timers or not can be selected by the MSTPA5 and MSTPA4 bits.

When the MSTPCRA.ACSE bit = 0 while the SBYCR.SSBY = 0, a transition to sleep mode is made after the WAIT instruction is executed.

Figure 12.3 Module Stop Control Register A Settings [1], pages 282–283.—*Continued*

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 305

Module Stop Control Register B (MSTPCRB)

Address(es): 0008 0014h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	MSTPB 31	MSTPB 30	MSTPB 29	MSTPB 28	MSTPB 27	MSTPB 26	MSTPB 25	MSTPB 24	MSTPB 23	—	MSTPB 21	MSTPB 20	MSTPB 19	MSTPB 18	MSTPB 17	MSTPB 16
Value after reset:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	MSTPB 15	—	—	—	—	—	—	MSTPB 8	—	—	—	MSTPB 4	—	MSTPB 2	MSTPB 1	MSTPB 0
Value after reset:	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	MSTPB0	CAN Module 0 Module Stop*[1]	Target module: CAN0 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b1	MSTPB1	CAN Module 1 Module Stop*[1]	Target module: CAN1 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b2	MSTPB2	CAN Module 2 Module Stop*[1]	Target module: CAN2 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b3	—	Reserved	This bit is read as 1. The write value should be 1.	R/W
b4	MSTPB4	Serial Communication Interface SCId Module Stop	Target module: SCId (SCI12) 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b7 to b5	—	Reserved	These bits are read as 1. The write value should be 1.	R/W
b8	MSTPB8	Temperature Sensor Module Stop	Target module: Temperature sensor 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b14 to b9	—	Reserved	These bits are always read as 1. The write value should always be 1.	R/W
b15	MSTPB15	Ethernet Controller DMAC Module Stop	Target module: EDMAC 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b16	MSTPB16	Serial Peripheral Interface 1 Module Stop	Target module: RSPI1 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b17	MSTPB17	Serial Peripheral Interface 0 Module Stop	Target module: RSPI0 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b18	MSTPB18	Universal Serial Bus Interface (Port 1) Module Stop*[2]	Target module: USB1 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W

Figure 12.4 Module Stop Control Register B Settings [1], pages 284–285.—*Continues*

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 306

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b19	MSTPB19	Universal Serial Bus Interface (Port 0) Module Stop*2	Target module: USB0 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b20	MSTPB20	I^2C Bus Interface 1 Module Stop	Target module: RIIC1 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b21	MSTPB21	I^2C Bus Interface 0 Module Stop	Target module: RIIC0 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b22	—	Reserved	This bit is read as 1. The write value should be 1.	R/W
b23	MSTPB23	CRC Calculator Module Stop	Target module: CRC 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b24	MSTPB24	Serial Communication Interface 7 Module Stop	Target module: SCI7 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b25	MSTPB25	Serial Communication Interface 6 Module Stop	Target module: SCI6 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b26	MSTPB26	Serial Communication Interface 5 Module Stop	Target module: SCI5 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b27	MSTPB27	Serial Communication Interface 4 Module Stop	Target module: SCI4 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b28	MSTPB28	Serial Communication Interface 3 Module Stop	Target module: SCI3 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b29	MSTPB29	Serial Communication Interface 2 Module Stop	Target module: SCI2 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b30	MSTPB30	Serial Communication Interface 1 Module Stop	Target module: SCI1 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W
b31	MSTPB31	Serial Communication Interface 0 Module Stop	Target module: SCI0 0: The module-stop state is canceled 1: Transition to the module-stop state is made	R/W

Notes:
1. The MSTPBi bit should be rewritten while the oscillation of the clock controlled by MSTPBi is stabilized. For entering software standby mode after rewriting the MSTPBi bit, wait for two CAN clock (fCANCLK) cycles after rewriting, and execute the WAIT instruction (i = 0 to 2).
2. For entering software standby mode after rewriting the MSTPB1i bit, wait for two USB clock (UCLK) cycles after rewriting, and execute the WAIT instruction. (i = 8, 9)

Figure 12.4 Module Stop Control Register B Settings [1], pages 284–285.—*Continued*

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 307

Deep Standby Control Register (DPSBYCR)

Address(es): 0008 C280h

	b7	b6	b5	b4	b3	b2	b1	b0
	DPSBY	IOKEEP	—	—	—	—	DEEPCUT[1:0]	
Value after reset:	0	0	0	0	0	0	0	1

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b1, b0	DEEPCUT [1:0]	Deep Cut	b1 b0	R/W
			0 0: Power is supplied to the RAM (RAM0*1) and USB resume detecting unit in deep software standby mode	
			0 1: Power is not supplied to the RAM (RAM0*1) and USB resume detecting unit in deep software standby mode	
			1 0: Setting prohibited	
			1 1: Power is not supplied to the RAM (RAM0*1) and USB resume detecting unit in deep software standby mode. In addition, LVD is stopped and the low power consumption function in a power-on reset circuit is enabled.	
b5 to b2	—	Reserved	These bits are read as 0. The write value should be 0.	R/W
b6	IOKEEP	I/O Port Retention	0: Deep software standby mode and I/O port retention are canceled simultaneously.	R/W
			1: The I/O port state is retained even after deep software standby mode is canceled. Then, writing 0 to the IOKEEP bit cancels the I/O port retention.	
b7	DPSBY	Deep Software Standby	SSBY b7	R/W
			0 0: Transition to sleep mode or all-module clock stop mode is made after the WAIT instruction is executed	
			0 1: Transition to sleep mode or all-module clock stop mode is made after the WAIT instruction is executed	
			1 0: Transition to software standby mode is made after the WAIT instruction is executed	
			1 1: Transition to deep software standby mode is made after the WAIT instruction is executed	

Note: 1. For the RAM address space, see Table 11.2.

Figure 12.5 Deep Standby Control Register [1], page 294.—*Continues*

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:56 AM Page 308

DPSBYCR is not initialized by the internal reset signal that is the source to cancel the deep software standby mode. For details, see Table 6.2, Targets to be Initialized by Each Reset Source.

DEEPCUT[1:0] Bits (Deep Cut)

The DEEPCUT[1:0] bits control the internal power supply to the RAM and USB resume detecting unit in deep software standby mode. In addition, these bits control the state of LVD and power-on reset circuit in deep software standby mode.

The internal power supply of RAM0 and USB resume detecting unit can be controlled by the setting of the DEEPCUT[1:0] bits.

When a USB suspend/resume interrupt is used as a deep software standby mode canceling source, the DEEPCUT[1:0] bits must be set to 00b.

When an LVD interrupt is used in deep software standby mode, the DEEPCUT[1:0] bits must be set to 00b or 01b.

For lower power consumption, set the DEEPCUT[1:0] bits to 11b so that the LVD is stopped and the low power consumption function of the power-on reset circuit is enabled.

The internal power supply of RAM1 is stopped in deep software standby mode regardless of the setting of the DEEPCUT[1:0] bits.

IOKEEP Bit (I/O Port Retention)

In deep software standby mode, I/O ports keep retaining the same states from software standby mode. The IOKEEP bit specifies whether to keep retaining the I/O port states from deep software standby mode even after deep software standby mode is canceled, or to cancel retaining the I/O port states.

DPSBY Bit (Deep Software Standby)

The DPSBY bit controls transitions to deep software standby mode.

When the WAIT instruction is executed while the SBYCR.SSBY and DPSBY bits are both 1, the LSI enters deep software standby mode through software standby mode.

The DPSBY bit remains 1 when deep software standby mode is canceled by certain pins which are sources of external pin interrupts (NM1, IRQ0-DS to IRQ15-DS, SCL2-DS, SDA2-DS, and CRXI-DS) or a peripheral interrupt (RTC alarm, RTC interval, USB suspend/resume, voltage monitoring 1, or voltage monitoring 2). Write 0 to this bit to clear it.

The setting of the DPSBY bit becomes invalid when the IWDT is in auto-start mode and the OFS0.IWDTSLCSTP is 0 (counting continues) or the IWDT is in register start mode and the SLCSTP bit in IWDTCSTPR is 0.

Instead, even when the SBYCR.SSBY bit is 1 and the DPSBY bit 1, the transition after the execution of a WAIT instruction is to software standby mode.

The setting of the DPSBY bit becomes invalid when voltage monitoring 1 reset is enabled by the voltage monitoring 1 circuit mode select bit (LVD1CR0.LVD1RI = 1) or when a voltage monitoring 2 reset is selected by the voltage monitoring 2 circuit mode bit (LVD2CR0.LVD2RI = 1). In this case, even when the SBYCR.SSBY bit is 1 and the DPSBY bit is 1, the transition after the execution of a WAIT instruction is to software standby mode.

Figure 12.5 Deep Standby Control Register [1], page 294.—*Continued*

System Clock Control Register (SCKCR)

Address(es): 0008 0020h

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	FCK[3:0]				ICK[3:0]				PSTOP1	PSTOP0	—	—	BCK[3:0]			
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	PCKA[3:0]				PCKB[3:0]				—	—	—	—	—	—	—	—
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b3 to b0	—	Reserved	These bits should be set to 0001b.	R/W
b7 to b4	—	Reserved	These bits should be set to 0001b.	R/W
b11 to b8	PCKB[3:0]	Peripheral Module Clock B (PCLKB) Select*1, *5	b11 b8	R/W
			0 0 0 0: ×1/1	
			0 0 0 1: ×1/2	
			0 0 1 0: ×1/4	
			0 0 1 1: ×1/8	
			0 1 0 0: ×1/16	
			0 1 0 1: ×1/32	
			0 1 1 0: ×1/64	
			Settings other than above are prohibited.	
b15 to b12	PCKA[3:0]	Peripheral Module Clock A (PCLKA) Select*1, *5	b15 b12	R/W
			0 0 0 0: ×1/1	
			0 0 0 1: ×1/2	
			0 0 1 0: ×1/4	
			0 0 1 1: ×1/8	
			0 1 0 0: ×1/16	
			0 1 0 1: ×1/32	
			0 1 1 0: ×1/64	
			Settings other than those listed above are prohibited.	

Figure 12.6 System Clock Control Register [1], page 245.—*Continues*

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 310

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b19 to b16	BCK[3:0]	External Bus Clock (BCLK) Select*1, *2, *5	b19 b16	R/W
			0 0 0 0: ×1/1	
			0 0 0 1: ×1/2	
			0 0 1 0: ×1/4	
			0 0 1 1: ×1/8	
			0 1 0 0: ×1/16	
			0 1 0 1: ×1/32	
			0 1 1 0: ×1/64	
			Settings other than above are prohibited.	
b21 to b20	—	Reserved	These bits are always read as 0. The write value should always be 0.	R/W
b22	PSTOP0	SDCLK Pin Output Control	0: SDCLK pin output is enabled.	R/W
			1: SDCLK pin output is disabled. (Fixed high)	
b23	PSTOP1	BCLK Pin Output Control*3	0: BCLK pin output is enabled.	R/W
			1: BCLK pin output is disabled. (Fixed high)	
b27 to b24	ICK[3:0]	System Clock (ICLK) Select*1, *2, *4, *5	b27 b24	R/W
			0 0 0 0: ×1/1	
			0 0 0 1: ×1/2	
			0 0 1 0: ×1/4	
			0 0 1 1: ×1/8	
			0 1 0 0: ×1/16	
			0 1 0 1: ×1/32	
			0 1 1 0: ×1/64	
			Settings other than above are prohibited.	
b31 to b28	FCK[3:0]	FlashIF Clock (FCLK) Select*1, *4, *5	b31 b28	R/W
			0 0 0 0: ×1/1	
			0 0 0 1: ×1/2	
			0 0 1 0: ×1/4	
			0 0 1 1: ×1/8	
			0 1 0 0: ×1/16	
			0 1 0 1: ×1/32	
			0 1 1 0: ×1/64	
			Settings other than above are prohibited.	

Notes:
1. The setting for division by one is prohibited if the PLL is selected.
2. Do not make a setting such that the ICLK runs at a lower frequency than the external bus clock.
3. When operation of the external bus clock is selected, the P53 I/O port pin function is not available because it is multiplexed on the same pin as the BCLK pin function.
4. When the SCKCR3.CKSEL[2:0] bits are selecting the sub-clock oscillator in low-speed operating mode 2, division by 1 is the only frequency division setting allowed for the ICLK and FCLK.
5. The setting for division by 1 or 2 is prohibited if the SCKCR3.CKSEL[2:0] bits are set to 010b (the main clock oscillator is selected)

Figure 12.6 System Clock Control Register [1], page 245.—*Continued*

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 311

The module stop function has the capability to turn off the clocks for all peripherals or for the peripherals not needed at any given time. This ability saves power of the unused units. The unused units can drain power through unnecessary switching as well as leakage. When the board is powered up most of the peripherals are disabled and prior settings must be enabled. See Section 12.4 for more detail.

The SDCLK output control function has the ability to alter the clock that is used to read the SDRAM. The SDCLK is an operating clock for the external bus controller [1]. This clock must not be set higher than the frequency of the system clock, ICLK. If the frequency is set higher, the clock should be set to the same value as the ICLK. This clock should be adjusted based on the maximum speed needed by the system in order to be able to accomplish the memory accesses required for proper operation. If this clock is set to the maximum frequency, power is wasted due to unnecessary switching during times of non-use. Refer to Section 12.4 for examples of how to set this value.

12.4 BASIC EXAMPLES

12.4.1 Example 1: Setting the Multi Clock Function

The following code changes peripheral clocks A and B to different frequencies in order to allow them to run at different speeds. Line 2 removes the protection to the clock generation circuit register. Line 3 selects the PLL circuit to multiply the frequency from the 12 MHz oscillator provided on the board by the factor that was set in reset program described. The PLL circuit multiplies the input oscillator by 16 and sets it to $12 \times 16 = 192$ MHz. Line 4 sets the peripheral clock A by dividing the PLL by 4 and setting it to $192 \div 4 = 48$ MHz. Line 5 sets peripheral clock B by dividing the PLL by 8 and setting it to $192 \div 8 = 24$ MHz. Line 6 turns the protection back on to the clock generation circuit registers.

```
1. void clockSetup() {
2.     SYSTEM.PRCR.WORD = 0xA501;
3.     SYSTEM.SCKCR3.WORD = 0x0400;    //PLL Circuit(MP factor = 16)
4.     SYSTEM.SCKCR.BIT.PCKA = 0x02;
5.     SYSTEM.SCKCR.BIT.PCKB = 0x03;
6.     SYSTEM.PRCR.WORD = 0xA500;
7. }
```

12.4.2 Example 2: Setting the Module Clock Function

The `moduleStop` function is the general format used to disable the clock for a specific peripheral. As shown in the example, the first step is to enable the ability to modify the

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 312

Module Stop Control Register by writing 0xA5xx to the write protect register. The upper 8 bits is the PRC key code, which decides what is prohibited or permitted for writing. The lower bits (xx) are based on which registers are being modified. In line 2, the low power registers are enabled. In line 3, the 12-bit A/D converter module is stopped. Last, we ensure that the PRCR register is reset to disable writing to the low power registers in case of a runaway program.

The all-module clock stop mode should be used to disable the clock for most of the peripherals. There are some exceptions; for example, the 8-bit timers must be explicitly stopped. In line 8, writing to the registers related to low power mode is enabled. Line 9 enables all-module clock stop mode. Line 10 verifies that the software standby bit is set to enter all-module clock stop mode, it should be defaulted to this value. Line 11 write protects those registers. Last, line 12 executes a wait statement that shifts to all-module clock stop mode.

```
 1. void moduleStop() {
 2.    SYSTEM.PRCR.WORD = 0xA502;
 3.    SYSTEM.MSTPCRA.BIT.MSTPA17 = 1;
 4.    SYSTEM.PRCR.WORD = 0xA500;
 5. }
 6.
 7. void allModuleStop() {
 8.    SYSTEM.PRCR.WORD = 0xA502;
 9.    SYSTEM.MSTPCRA.BIT.ACSE = 1;
10.    SYSTEM.SBYCR.SSBY = 0;
11.    SYSTEM.PRCR.WORD = 0xA500;
12.    wait();
13. }
```

12.4.3 Example 3: Setting the SDCLK Output Control Function

The following function is an example of how to set the SDRAM clock speed, as well as enable it. Line 2 turns off the write protection. Line 3 sets the external bus clock speed to 48 MHz, assuming the PLL circuit is selected and is outputting 192 MHz. Line 4 stops the SDCLK so that line 5 can enable the SDCLK output. Line 6 starts the SDCLK output. Line 7 turns the register protection back on.

```
1. void SDCLKSetup() {
2.    SYSTEM.PRCR.WORD = 0xA501;
```

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 313


```
3.      SYSTEM.SCKCR.BIT.BCK = 0x02;
4.      SYSTEM.SCKCR.BIT.PSTOP0 = 1;
5.      SESTEM.PFBCR1.BIT.SDCLKE = 1;
6.      SYSTEM.SCKCR.BIT.PSTOP0 = 0;
7.      SYSTEM.PRCR.WORD = 0xA500;
8. }
```

12.5 ADVANCED CONCEPTS OF LOW POWER CONSUMPTION

Table 11.2 from the hardware manual [1, p. 256] clearly explains state of different peripherals in different low power modes.

12.5.1 Sleep Mode

In sleep mode the CPU stops operating but the contents of the CPU registers retain their values. Peripherals can be kept awake or in operation in this mode. Sleep mode uses the most power of all the low power modes but less power than the normal CPU operation.

To enter into Sleep Mode, first ensure that the interrupt to terminate this mode has been set up correctly in order to allow cancelling sleep mode. To configure the interrupt, enable the interrupt and set its priority level to the highest by setting the IPL[3:0] bits of the PSW register in the CPU prior to executing a WAIT statement. This setting causes the transition to sleep mode. Read the last I/O register to confirm the value written was reflected.

Sleep mode can be canceled using **any** interrupt, as long as the interrupt was set up prior to entering sleep mode. Sleep mode can be canceled by the reset pin, a power-on reset, a voltage monitoring reset, or a reset caused by an IWDT underflow. The WDT is stopped during sleep mode; therefore, it can be used in a program that uses WDT without the worry of it being reset. It is possible to switch the clock source to return from sleep mode. Refer to the Renesas Hardware Manual [1] on how to handle this operation.

12.5.2 All-Module Clock Stop Mode

The all-module clock stop mode is a method of reducing power consumption on the MCU/development board by disabling the bus controller, I/O ports, and most of the peripheral modules. The 8-bit timers, output port enable interrupts, the independent

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 314

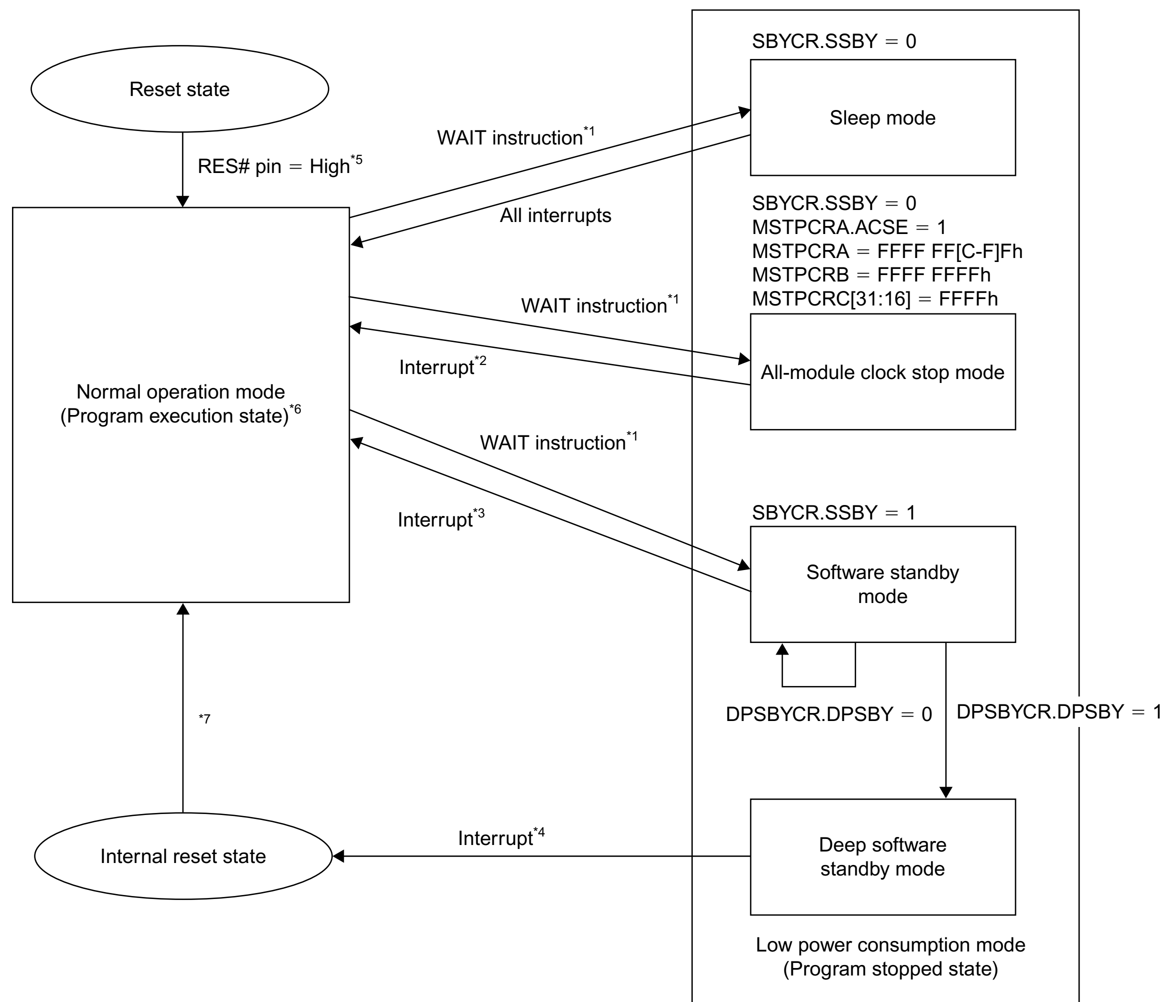


Figure 12.7 State Machine for Low Power Modes [1], page 280.

watchdog timer (IWDT), the power-on reset circuit, and the voltage detection circuits are all left active. The remaining peripherals are turned off to prevent wasting clock cycles and leakage. If the program is using the WDT, it is OK to place the processor in this state as the WDT will stop counting. Since most embedded systems use a WDT, it should stop counting during this mode. The all-module clock stop mode uses less power

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 315

than the sleep mode since it disables all peripheral clocks, where in sleep mode only the CPU clock is disabled.

Not all peripherals are enabled during power up. These peripherals must be enabled using the correct settings in the module stop control registers.

12.5.3 Software Standby Mode

Because one of its main functions is to stop the oscillator, software standby mode considerably reduces power consumption. In this mode the CPU on-chip peripheral and the on-chip oscillator functions stop. During this mode the contents of the CPU internal registers, RAM data, on-chip peripheral functions, and the states of the I/O ports are maintained. The WDT also stops counting and does not have to be disabled in this mode. This mode is a good candidate for any processes that wait on an interrupt from an external process or peripheral prior to performing a task.

How to cancel this mode is important. It can be cancelled by an external pin interrupt (the NMI or IRQ0-15), peripheral interrupt, RTC alarm, RTC Interval, IWDT, USB suspend/resume, voltage monitoring, and resets.

12.5.4 Deep Software Standby Mode

The Deep Software Standby mode saves the most power. However, the data in the CPU registers and most internal peripheral modules become undefined; therefore, needed data may be possibly lost. Data in RAM0 may become undefined depending on the setting of the DEPPCUT bits—a setting of '00' retains data in RAM0. The reason for this loss is due to the fact that the internal supply of power for these modules is stopped. In deep software standby mode the CPU, internal peripheral modules, RAM1-3, and all functions of the oscillators are stopped. The WDT stops counting when the power supply and oscillators are stopped. Similar to the Software Standby mode, the I/O port states are also maintained when the power and oscillators are stopped.

Obviously, a designer must consider that there is a tradeoff between power savings and start-up times when designing a system. For example, if a system is put in Deep Software Standby Mode, the software should write all valuable data stored in RAM to non-volatile memory before sleeping. Then some interrupting event (i.e. button press) can "wake up" the processor and have the software re-load the RAM and reactivate all peripherals to the "pre-sleep" state. Many embedded devices, like photocopiers and fax machines, use these energy savings processes.

This mode is cancelled using external interrupt source pins (the NMI, IRQ0-DS to IRQ15-DS, SCL2-DS, SDA2-DS or CRX1-DS), peripheral interrupts (RTC alarm, RTC interval, USB suspend/resume, voltage monitoring), and a processor reset.

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 316

Refer to the example in Section 12.6.3 for how to enter and cancel the Deep Software Standby mode.

Table 11.2 from the hardware manual [1, p. 256] clearly explains the states of the different peripherals in these four low power modes.

12.6 ADVANCED CONCEPT EXAMPLES

12.6.1 Example 1: Sleep Mode

The following example shows how to enter sleep mode after a switch is pressed. Once in low power mode, several types of interrupts can be used to cancel and return to sleep mode operation. A timer is used to cancel the low power mode and return to standard operation. Refer to Section 12.5.1 for more details.

```
 1. #include <stdint.h>
 2. #include <stdio.h>
 3. #include <machine.h>
 4. #include "platform.h"
 5. #include "r_switches.h"
 6. #include "mcu_mode.h"
 7.
 8. void tmrSetup(void);
 9. volatile int switchPressed = 0;
10.
11. void main(void) {
12.    int i;
13.    lcd_initialize();
14.    lcd_clear();
15.    R_SWITCHES_Init();
16.    while (1){
17.       switchPressed = 0;
18.       IR(ICU,IRQ8) = 0;
19.       IEN(ICU,IRQ8) = 1;
20.       lcd_display(LCD_LINE4, "Working");
21.       lcd_display(LCD_LINE5, "Press SW1");
22.       lcd_display(LCD_LINE6, "to sleep.");
23.       while(!switchPressed);
24.       lcd_display(LCD_LINE4, "Sleeping");
25.       lcd_display(LCD_LINE5, "Timer will");
```

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 317


```
26.         lcd_display(LCD_LINE6, "awake.");
27.         for(i = 0; i < 9800000; i++);
28.         if(0x100000 & get_psw()) {
29.            chg_pmsuper();
30.         }
31.         clrpsw_i();
32.         IR(ICU,IRQ8 ) = 0;
33.         IEN(ICU, IRQ8) = 0;
34.         SYSTEM.SBYCR.BIT.SSBY = 0;
35.         if(SYSTEM.SBYCR.WORD);
36.            tmrSetup();
37.            wait();
38.            chg_pmusr();
39.            for(i = 0; i < 9800000; i++);
40.         }
41.      }
42. void sw1_callback(void) {
43.      switchPressed = 1;
44. }
45.
46. void tmrSetup(void) {
47.      SYSTEM.PRCR.WORD = 0xA50B;
48.      SYSTEM.MSTPCRA.BIT.MSTPA5 = 0;
49.      TMR0.TCCR.BIT.CSS = 1;
50.      TMR0.TCCR.BIT.CKS = 6;
51.      TMR0.TCR.BIT.CCLR = 1;
52.      TMR0.TCORA = 0xFF;
53.      TMR1.TCCR.BIT.CSS = 3;
54.      TMR1.TCR.BIT.OVIE = 1;
55.      TMR1.TCNT = 0;
56.      TMR0.TCNT = 0;
57.      IEN(TMR1, OVI1) = 1;
58.      ICU.IPR[173].BIT.IPR = 3;
59. }
```

Lines 1 through 6 contain all the header files that are included in order to complete the task in the program. Line 8 declares the tmrSetup function prototype. Lines 12 and 13 then initialize and clear the LCD for use. Line 15 calls the function which sets the switch direction and interrupt control. Line 18 clears the switch1, IRQ8, and pending interrupts. Line 19 enables the interrupt associated with switch1 and the IRQ8. Lines 20 through 22 display LCD messages prior to pressing the switch.

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 318

Line 23 waits on a switch to be pressed. The switchPressed flag is set in the sw1_call-back function from the ISR. Once a switch is pressed, lines 24 through 26 display the "Sleeping Press SW1 to awake." Line 27 is used to allow the processor hardware to latch all settings into the registers prior to executing Line 28. Line 28 through 30 check the PSW register for user mode. If in user mode, the mode is changed to supervisor mode. Line 31 disables all of the maskable interrupts in order to disable the interrupt and to cancel sleep mode, for example, preventing the timer overflow from being triggered prior to entering the low power mode. Line 32 clears any pending interrupt requests prior to entering low power mode. Line 33 disables the switch interrupt while the timer is being used to cancel the low power mode. Line 34 sets the software standby mode to 0, telling the controller to switch into sleep mode once a wait statement is issued. Line 35 provides a force register read to ensure the software standby mode selection has completed. Line 36 calls the timer function that sets the timer to a predefined value for interruption. Line 37 tells the controller to enter into low power mode and enable interrupts. Line 38 changes the processor to user mode from supervisor mode. Line 39 provides a wait for the processor to stabilize upon wake up.

12.6.2 Example 2: Software Standby Mode

The following example demonstrates the transition to software standby mode after a switch is pressed. In this example the software standby mode is cancelled and returned to normal operation using a switch press. Refer to Section 12.5.3, *Software Standby Mode,* for other interrupts allowed to cancel this mode.

```
 1. #include <stdint.h>
 2. #include <stdio.h>
 3. #include <machine.h>
 4. #include "platform.h"
 5. #include "r_switches.h"
 6. #include "mcu_mode.h"
 7.
 8. void sw1_callback(void);
 9.
10. volatile int switchPressed = 0;
11.
12. void main(void) {
13.    int i;
14.    lcd_initialize();
15.    lcd_clear();
16.    R_SWITCHES_Init();
```

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 319


```
17.    while(1) {
18.       switchPressed = 0;
19.       IR(ICU,IRQ8 ) = 0;
20.       IEN(ICU, IRQ8) = 1;
21.
22.       lcd_display(LCD_LINE4, "Working");
23.       lcd_display(LCD_LINE5, "Press SW1");
24.       lcd_display(LCD_LINE6, "for Software");
25.       lcd_display(LCD_LINE7, "Standby.");
26.
27.       while(!switchPressed);
28.
29.       lcd_display(LCD_LINE4, "Software");
30.       lcd_display(LCD_LINE5, "Standby");
31.       lcd_display(LCD_LINE6, "Press SW1");
32.       lcd_display(LCD_LINE7, "to awake");
33.
34.       for(i = 0; i < 9800000; i++);
35.       if(0x100000 & get_psw()) {
36.          chg_pmsuper();
37.       }
38.       clrpsw_i();
39.       IR(ICU,IRQ8 ) = 0;
40.       IEN(ICU, IRQ8) = 1;
41.       SYSTEM.SBYCR.BIT.SSBY = 1;
42.       SYSTEM.SBYCR.BIT.OPE = 0;
43.       SYSTEM.DPSBYCR.BIT.DPSBY = 0;
44.       if(SYSTEM.OSTDCR.BYTE)
45.          nop();
46.       if(SYSTEM.SBYCR.WORD)
47.          nop();
48.       wait();
49.       chg_pmusr();
50.       for(i = 0; i < 9800000; i++);
51.    }
52. }
53.
54. void sw1_callback(void) {
55.    switchPressed = 1;
56. }
57.
```

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 320

Lines 1 through 7 are the header files that are included in order to complete the task in the program. Line 8 declares the switch1 callback function prototype. Lines 14 and 15 initialize and clear the LCD for use. Line 16 calls the function which sets the switch direction and interrupt controls. Line 19 is the code used to clear the switch1, IRQ8, and any pending interrupts. Line 20 enables the interrupt associated with the switch1, IRQ8. Lines 20 through 22 display LCD messages prior to a switch being pressed. Line 27 waits on a switch to be pressed. The switchPressed flag is set in the sw1_callback when that function is executed in an interrupt service routine. Once a switch is pressed, lines 29 through 32 display the "Software Standby." Line 34 is used to allow the processor hardware to latch all settings into the registers prior to executing line 35. Lines 35 through 36 check the PSW register for user mode. If in user mode, the mode is changed to supervisor mode. Line 37 disables all the maskable interrupts to ensure the interrupt to cancel software standby mode; for example, the switch press is not triggered prior to entering the low power mode. Line 38 clears any pending interrupt requests prior to entering the low power mode. Line 39 enables the switch interrupt that is used to cancel the software standby mode. Line 40 sets the software standby mode to 1, telling the controller to switch into software standby mode once a wait statement is issued. Line 42 determines whether or not to enter deep software standby mode. Lines 44 through 46 are a force register read that ensure the software standby mode selection has completed. Line 47 tells the controller to enter into the low power mode and enable interrupts. Line 49 changes the processor to user mode from supervisor mode. Line 50 issues a wait for the processor to stabilize upon wake up.

12.6.3 Example 3: Deep Software Standby Mode

The following example demonstrates switching the processor into the deep software standby mode at a switch press. The Real Time Clock Alarm Interrupt cancels the current mode and returns to normal operation. Refer to Section 12.5.4, *Deep Software Standby Mode,* for other interrupts allowed to cancel this mode.

```
1. #include <stdint.h>
2. #include <stdio.h>
3. #include <machine.h>
4. #include "platform.h"
5. #include "r_switches.h"
6. #include "mcu_mode.h"
7.
8. void RTCSetup(void);
9. volatile int switchPressed = 0;
```

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 321


```
10.
11. void main(void) {
12.    int i;
13.    lcd_initialize();
14.    lcd_clear();
15.
16.    R_SWITCHES_Init();
17.
18.    while (1) {
19.       switchPressed = 0;
20.       RTC.RCR2.BIT.START = 0;
21.       IR(ICU,IRQ8 ) = 0;;
22.       IEN(ICU, IRQ8) = 1;
23.       lcd_display(LCD_LINE4, "Working");
24.       lcd_display(LCD_LINE5, "Press SW1 4");
25.       lcd_display(LCD_LINE6, "Deep Soft.");
26.       lcd_display(LCD_LINE7, "Standby.");
27.       while(!switchPressed);
28.       lcd_display(LCD_LINE4, "Deep");
29.       lcd_display(LCD_LINE5, "Software");
30.       lcd_display(LCD_LINE6, "Standby");
31.       lcd_display(LCD_LINE7, "Mode");
32.       for(i = 0; i < 9800000; i++);
33.       if(0x100000 & get_psw()) {
34.          chg_pmsuper();
35.       }
36.       clrpsw_i();
37.       IR(ICU,IRQ8 ) = 0;
38.       IEN(ICU, IRQ8) = 0;
39.       SYSTEM.SBYCR.BIT.SSBY = 1;
40.       SYSTEM.SBYCR.BIT.OPE = 0;
41.       SYSTEM.DPSBYCR.BIT.DPSBY = 1;
42.       if (SYSTEM.OSTDCR.BYTE) {
43.          nop();
44.       }
45.       if (SYSTEM.SBYCR.WORD) {
46.          nop();
47.       }
48.       RTCSetup();
49.       wait();
50.       chg_pmusr();
```

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 322


```
51.         for(i = 0; i < 9800000; i++);
52.     }
53. }
54.
55. void sw1_callback(void) {
56.     switchPressed = 1;
57. }
58.
59. void RTCSetup(void) {
60.     RTC.RCR4.BIT.RCKSEL = 1;
61.     RTC.RCR3.BIT.RTCEN = 1;
62.     RTC.RCR2.BIT.START = 1;
63.     ICU.IER[11].BIT.IEN4 = 0;
64.     RTC.RSECAR.BYTE = 0x85;
65.     RTC.RCR1.BYTE = 1;
66.     while(RTC.RCR1.BYTE != 1);
67.     ICU.IR[92].BIT.IR = 0;
68.     ICU.IER[11].BIT.IEN4 = 1;
69.     ICU.IPR[92].BIT.IPR = 3;
70.     SYSTEM.PRCR.WORD = 0xA50B;
71.     SYSTEM.DPSIER2.BIT.DRTCAIE = 1;
72.     SYSTEM.DPSIFR2.BIT.DRTCAIF = 1;
73. }
```

Lines 1 through 6 are the header files included in order to complete the task in the program. Line 8 declares the Realtime Clock Setup function prototype. Lines 13 and 14 initialize and clear the LCD for use. Line 16 calls the function which sets the switch direction and interrupt controls. At line 20, the RTC is stopped. Line 21 is the code used to clear the switch1, IRQ8, and pending interrupts. Line 22 enables the interrupt associated with the switch1 and IRQ8. Lines 23 through 26 displays the LCD message prior to the switch press. Line 27 waits on a switch to be pressed. The switchPressed flag is set in the sw1_callback function from the ISR.

Once a switch is pressed lines 28 through 31 display the "Deep Software Standby Mode." Line 32 is used to allow the processor hardware to latch all settings into the registers prior to executing Line 33. Line 33 through 35 check the PSW register for user mode; if in user mode, the mode is changed to supervisor mode. Line 36 disables maskable interrupts to ensure the interrupts to cancel out deep software standby mode (i.e., the RTC alarm) is not triggered prior to entering the low power mode. Line 37 clears any pending interrupt requests prior to entering the low power mode and line 38 disables the switch interrupt as it will not be used to cancel the deep software standby mode. Line 39 sets the software standby mode to 1, telling the controller to switch into

12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 323

software standby mode once a wait statement is issued. Line 40 sets the address bus and bus control signals to a high impedance state in order to conserve power. Line 41 sets to enter deep software standby mode. Lines 42 through 47 are a force of register read to ensure the deep software standby mode selection has completed. Line 48 calls the RTC setup function which enables all interrupts, sets up the alarm to a certain second value, enables interrupts, and starts the clock. Line 49 tells the controller to enter into the low power mode and enable interrupts. Line 50 changes the processor to user mode from supervisor mode. Line 51 is used as a wait for the processor to stabilize upon wake up.

12.7 RECAP

Power consumption is an integral part of any embedded system design and must be accounted for early on in the process, even as early as MCU selection. Most of the today's modern processors have multiple modes to reduce power consumption depending on application needs.

The RX63N has four low power modes: all-module clock stop, software standby, sleep mode deep software standby mode, as well as the ability to change clock speeds to suit your application. The RX63N also has multiple clocks for bus and peripheral control. One can also save power by turning off one or more peripherals devices in the RX63N microcontroller.

12.8 REFERENCES

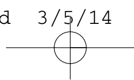
[1] Renesas Electronics, Inc. (February, 2013). *RX63N Group, RX631 Group User's Manual: Hardware,* Rev 1.60.

[2] Gupta, S and Kumar, M. (2012). Optimizing for Low Power in Embedded MCU Designs. *Embedded: Cracking the Code to Systems Development.* September 8, 2012. Web. March 12, 2012. http://www.embedded.com/design/power-optimization/4395689/Optimizing-for-low-power-in-embedded-MCU-designs

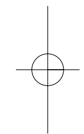
12.9 EXERCISES

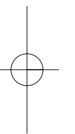
1. List out the tasks to be done upon power-on reset to configure the MCU.
2. What are the two modes in which the RX63N processor and other processors consume their power?
3. What are the two different areas of power loss for the RX63N processor as well as most others?

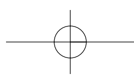
12.ES_Conrad_RX63N_Advanced_CH12.qxd:RX63N Advanced 3/5/14 7:57 AM Page 324



4. What should be an Embedded Systems Designer's focus in reducing power consumption in the RX63N processor?
5. Write a function to change the system clock frequency to 48 MHz. Do not assume that any of the registers are defaulted or set correctly.
6. How do you unprotect ALL the registers that are write-protected so that they can be written directly, using one line only?
7. List out the low power modes in the RX63N in descending order of their power consumption.
8. In All-Module Clock Stop Mode, are all of the peripheral modules turned off?
9. Which interrupts are allowed to cancel the Deep Software Standby state?
10. Does a reset of any kind cancel ALL low power modes?
11. In Deep Software Standby Mode do all values in RAM and computer registers remain valid?
12. Give one good reason, with an example, of why power consumption is a major requirement in embedded systems?
13. What is the difference between sleep mode and software standby mode?







Index

A
Acceptance filter, CAN, 243
ACK field, of data and remote frame, 237
Acknowledge check, 241
ACSE bit, 304
Activation record, 24
Addressing mode(s)
  immediate, 6
  register direct, 6–7
  register indirect, 7
  register relative, 7–8
All module clock mode, low power consumption, 301
All-module clock stop mode, low power consumption, 313–15
All-module clock stop mode enable (ACSE) bit, 304
Arbitration field, of data and remote frame, 236
Argument(s)
  concepts of, 25
  description of, 24
  passing
    methods for, 25
    rules for, 23–26
Arithmetic instructions, 9–10
Assembly language
  addresses in memory, 13–15
  addressing modes, 5–8
  concepts of, 1, 5–19
  data storage and use, 2–5
  inline assembly, 18–19
  instruction set (*see* instruction set)
  machine code, low *vs* high, 1
  notations, of RX family, 15–16
  registers of, 3
  set up ports and turn on LEDs, *ex,* 19–20
  simple software debounce for switch, *ex,* 20–21
  source code, writing of, 15–18
Auto-start mode, 282–83, 284

B
BCADR [10:0] bits, 162
BCK[3.0] bit, low power consumption, 301
BCLK, power consumption, reduction of, 300–301
BCR, 245, 257
Binary semaphore, 78, 87
Bit configuration register (BCR), 245, 257
Bit monitoring, 240
Bit rate
  automatic adjustment of, in boot mode, 180–81
  and CAN, 266
Bit stuffing, 240
Bit timing setting, CAN, 266
Bit(s). *see* individual bits
Bitwise operation, description of, 5
Blank check address setting, 162
Block transfer mode
  DMAC and, 123–24
  external data transfer, *ex,* 132–33
  internal data transfer, *ex,* 126
  setting transfer, *ex,* 116
Boot mode
  bit rate, automatic adjustment of, 180–81
  control code, 181
  flash memory and, 142, 178
  ID code protection, 181, 182–83
  startup process and, 295
  state transitions in, 179–80
  system configuration, 178
Brownout condition, 291
Buffer memory, FIFO. *see* FIFO memory buffer
Bulk transfers, of USB, 191
Bus arbitration, in CAN protocol, 238–39
Bus off state, in CAN protocol, 241–42
Bus topology, 210, 211

C
CAN. *see* Controller Area Network (CAN)
Carrier Sense Multiple Access with Collision Detection, 210
Cause flag(s), floating point status word, 61–62
CCI, 193
CDC. *see* Communication Device Class (CDC)
Checksum check, 241
CKS [3.0] bits, 279
Class driver(s). *see also* Human Interface Device (HID)
  communication device class, 192–94, 195
  mass storage device class, 194, 196
Clock setting, of CAN, 265

13.ES_Conrad_RX63N_Advanced_Index.qxd:Conrad 3/7/14 11:49 AM Page 326

Clock(s)
  default frequency, 296
  low power modes (*see* low power modes and consumption)
  module clock function, *ex,* 311–12
  multi-clock functionality, 311
  power consumption, reduction of, 301
Cluster transfer mode
  description of, 128
  external data transfer, *ex,* 133–35
  operation while in, *fig,* 130
  register update operation, 129
CMDLK bit, 150
CNTVAL [13.0] bits, 280, 288
Co-axial bus topology, 210
Code, writing rules and process of, 15–18
Command-lock state, and FCU, 177
Communication Class Interface (CCI), 193
Communication Device Class (CDC)
  description of, 192
  endpoints of, 194
  function examples for, 193
  groups of, 192–93
  subclasses of, 195
Complex number kernel(s), 94
Config_r_can_api.h, 270
Connectors, USB, 187–88
Control code, in boot mode, 181
Control field, of data and remote frame, 237
Control register (CTLR), 245
Control transfer instructions, 11–12
Controller Area Network (CAN)
  acceptance filtering, 266–68
  application programming interface (API), 270–73
  block diagram of, *fig,* 243
  bus arbitration, 238–39
  bus initialization, *ex,* 271–72
  bus send/receive, *ex,* 272–73
  bus standards, 242
  characteristics of, 234–35
  communication speed setting, 265–66
  concepts of, basic, 242–62
  data transfer synchronization, 240
  description of, 234, 241–42
  error detection, 240–42
  error state, confirmation of, 241–42
  fault confinement, 235, 241
  frame
    data and remote, 236–38
    error, 237–38
    overload, 238
  initialization of bus, *ex,* 255–58
  interrupt use, *ex,* 269–70
  interrupts of, 268–69
  mailbox modes of, 244
  main function, *ex,* 260–62
  masking functions, 266–68
  message broadcasting and, 235, 239–40
  nodes, states of, 241
  operating modes of, 262–64
  protocol, theory of, 234–42
  reception, 249–52, 253, 258–59
  registers of, 244–49
  transmission, 249–50, 252, 254, 259
  voltage levels of, 234
Co-operative scheduling, in FreeRTOS, 84
Counting semaphore, 78
CRC delimiter bit, 237
CRC field, of data and remote frame, 237
CRC sequence field, 237
CRXi, 243
CSMA/CD, 210
CTLR, 245
CTXi, 243

D

Data and remote frame, of Controller Area Network (CAN) protocol, 236–38
Data Class Interface (DCI), 193
Data communication equipment, 209
Data Length Code (DLC) field, 237
Data packet, USB transaction, 191
Data storage
  data, types of, 4–5
  description of, 2
  memory, important addresses in, 13–15
  memory features of RX63N, 2
  memory map, 14, 32–34
  registers for, 3
Data terminal equipment (DTEs), 209
Data transfer instructions, 8–9
Data transfer synchronization, and CAN protocol, 240
DCEs, 209
DCI, 193
Debounce for switch, *ex,* 20–21
Deep cut (DEEPCUT[1.0]) bit, 308
Deep software, low power consumption, 301
Deep software standby (DPSBY) bit, 308
Deep software standby mode, 315, 320–23

13.ES_Conrad_RX63N_Advanced_Index.qxd:Conrad 3/7/14 11:49 AM Page 327

Deep standby control register (DPSBYCR), 307–8
DEEPCUT[1.0] bit, 308
Denormalized number bit(s), 62
DFLAE bit, 150
DFLBCCNT, 162
DFLBCSTAT, 163
DFLRE0, 151–52
DFLRE1, 152
DFLREy, 177
DFLRPE bit, 150
DFLWE0, 153
DFLWE1, 154–55
DFLWEy, 177
DFLWPE bit, 150
DFT, 107
Digital signal processing (DSP)
  data, structures and types of, 95–96
  description of, 93
  executive attributes of, 95
  fast fourier transform (FFT), 107–9
  finite impulse response (FIR) filter, *ex,* 98–103
  floating point exception, 97
  function arguments, 98
  function naming convention, 97–98
  instructions of, 13
  kernel, defined, 93
  kernel handles, 96–97
  library, concepts of, 93–98
  library kernels, 94–95
  matrix multiplication, 103–7
  vector and matrices, 96
Direct memory access controller (DMAC)
  block diagram of, 113, 115
  cluster transfer mode, 128–30
  concepts of, 111–23
  description of, 112–13
  external data transfer, *ex,* 131–35
  external direct memory access controller, 126–30
  internal data transfer, *ex,* 123–27
  operation, modes of, 113, 116, 120–23
  registers of, 115–20, 122, 123, 135
  specifications of, 114
Direct memory access controller (DMAC)i, interrupts, examples with, 135–37
Discrete Fourier Transform (DFT), 107
Divide-by-zero exception (EZ), 56, 59, 62
DLC, 237
DMA block transfer count register (DMCRB), 117
DMA destination address register (DMDAR), 116
DMA interrupt setting register (DMINT), 117, 119
DMA source address register (DMSAR), 115–16
DMA transfer count register (DMCRA), 116–17
DMA transfer enable register (DMCNT), 117–20
DMA transfer mode register (DMTMD), 117, 118
DMAC. *see* direct memory access controller (DMAC)
DMCNT, 117–20
DMCRA, 116–17
DMCRB, 117
DMDAR, 116
DMINT, 117, 119
DMSAR, 115–16
DMTMD, 117, 118
Double precision, 47–49
Down-counter value, 280, 288
DPSBY bit, 308
DPSBYCR register, 307–8
DSP. *see* digital signal processing (DSP)
DTEs, 209

E

E stage, 50
E2 data flash
  internet protocol and, 212
  low power consumption, 301
E2 DataFlash access violation flag, 150
E2 DataFlash blank check control register (DFLBCCNT), 162
E2 DataFlash blank check status register (DFLBCSTAT), 163
E2 DataFlash lock-bit read mode, 146
E2 DataFlash memory. *see* flash memory
E2 DataFlash P/E enable register 0 (DFLWE0), 153
E2 DataFlash P/E enable register 1 (DFLWE1), 154–55
E2 DataFlash P/E modes
  description of, 146
  switching to, 164–65
E2 DataFlash P/E normal mode, 146
E2 DataFlash programming/erasure protection violation bit, 150
E2 DataFlash read enable register 0 (DFLRE0), 151–52
E2 DataFlash read enable register 1 (DFLRE1), 152
E2 DataFlash read protection violation flag, 150
E2 DataFlash status read mode, 146
ECMR, 216–19
ECSR, 220, 221

13.ES_Conrad_RX63N_Advanced_Index.qxd:Conrad 3/7/14 11:49 AM Page 328

EDMAC. *see* ethernet controller
EDMAC mode register (EDMR), 223
EDMAC receive request register (EDRRR), 224–25
EDMAC transmit request register (EDTRR), 224
EDMR, 223
EDRRR, 224–25
EDTRR, 224
EEPROM. *see* flash memory; Virtual EEPROM (VEE)
EHCI, 190–91
Electrically erasable programmable read-only memory. *see* flash memory; Virtual EEPROM (VEE)
Embedded TCP/IP Stack [4], 212
Enhanced Host Controller Interface, 190–91
EOF field, of data and remote frame, 236, 237
Equation(s)
  logic levels, 234
  power consumption, 300
  T-tap FIR filter, structure of, 99
Erasure. *see* flash memory
Erasure error flag, 157
Erasure suspend mode, 161
Erasure suspend status flag, 156
Error active state, in CAN protocol, 241
Error delimiter, field of error frame, 238
Error flag, field of error frame, 237
Error frame, in CAN protocol, 235, 237–38
Error passive state, in CAN protocol, 241
ERSERR bit, 157
ERSSPD bit, 156
ESUSPMD bit, 161
ETHERC, 214
Ethernet controller
  configuration of, 214
  controller chip, 213–14
  description of, 209
  direct memory access controller, 214–15, 222–26, 245
  driver API, 227–31
  frames
    reception of, 215–16
    transmission of, 214–15
  internet protocol, 212
  network topology, 209–11
  receiving frames, *ex,* 230–31
  registers of, 216–21, 223–26
  set up, *ex,* 212–13
  transmitting frames, *ex,* 229–30
Ethernet controller mode register (ECMR), 216–17
Ethernet controller status register (ECSR), 220, 221
Ethernet direct memory access controller (EDMAC). *see* ethernet controller
Ethernet mode control register, 217–18
Exception flag(s), floating point status word, 62
Exception handling enable bit(s), floating point status word, 62
EXDMAC. *see* external direct memory access controller (EXDMAC)
Execution stage (E stage), 50
External direct memory access controller (EXDMAC)
  cluster transfer mode, 128–30
  description of, 126
  operation, modes of, 127
  register update operation, 129
  registers of, 128
EZ, 56, 59, 62

F

FADD, 54
Fast Fourier Kernel. *see* Fast Fourier Transform (FFT)
Fast Fourier Transform (FFT), 93, 98
  coefficients for, 109
  description of, 107
  set up for, 107–9
FASTAT, 149–50, 177
FCMDR, 161
FCMP, 55
FCPSR, 161
FCRME bit, 155
FCU. *see* flash control unit (FCU)
FCU command register, 161
FCU command-lock flag, 150
FCU error flag, 158
FCU processing switching register (FCPSR), 161
FCU RAM enable bit, 155
FCU RAM enable register (FCURAME), 155
FCUERR bit, 158
FCURAME, 155
FENTRYR, 158–59, 177
FFT, 93, 98
FIFO mailbox mode, of CAN, 244
File Transfer Protocol (FTP), 212
Filter kernel(s), 94
Finite Impulse Response (FIR) filter, 93, 98–103
FIR filter, 93, 98–103
Fixed point math, 43, 66–70
Flash access status register (FASTAT), 149–50, 177
Flash control unit (FCU)
  commands of, 147
  description of, 145
  modes of, 145–46
  registers of, 148–64

13.ES_Conrad_RX63N_Advanced_Index.qxd:Conrad 3/7/14 11:49 AM Page 329

Flash memory
  block diagram of, 141
  boot mode (*see* boot mode)
  data flash area read and program permissions, *ex,* 169–70
  description of, 139
  E2 DataFlash, block configuration of, 144
  EEPROM, difference between, 139
  erasing, procedures for, 166–67
  FCU, description and modes of, 145–47
  FCU peripheral block initialization, *ex,* 168–69
  Flash API copying, *ex,* 170
  low power consumption, 301
  mode, transitions of, 164–65
  notification of clock, *ex,* 183–84
  operating modes of, 141–42
  program lock-bit, *ex,* 184–85
  programming, procedures for, 165–66
  ROM, block configuration of, 142–43
  Simple Flash API, 167–68
  software protection, 176–77
  specifications of, 140
  Virtual EEPROM (*see* Virtual EEPROM)
Flash mode register (FMODR), 149
Flash P/E mode entry register (FENTRYR), 158–59, 177
Flash P/E status register (FPESTAT), 162–63
Flash protection register (FPROTR), 159–60
Flash ready interrupt enable bit, 151
Flash ready interrupt enable register (FRDYIE), 150–51
Flash reset bit, 160–61
Flash reset register (FRESETR), 160
Flash status register 0 (FSTATR0), 155–56
Flash status register 1 (FSTATR1), 157
Flash write erase protection register (FWEPROR), 148, 177
Floating-point error summary flag, 63
Floating-point operation, 4, 11
Floating-point status word (FPSW), 51–52, 60–63
Floating-point unit (FPU)
  concepts of, 43–56
  dsp and, 97
  exceptions
    descriptions of, 55–56
    ex, 56–60
    handling of, 63–66
  FPSW bit definition, *tab,* 52–53
  instructions of, 54–55
  math, basics of, 43–49
  matrix multiplication time calculation, *ex,* 70–74
  registers of, 51–52
  representation of, 43–44
  in RX63N, 50–56
  standard of, 44–49
  time calculation, *ex,* 66–70
FLOCKST bit, 157
FMODR, 149
Fourier Transform, 107
FPESTAT, 162–63
FPROTCN bit, 160, 177
FPROTR, 159–60
FPU. *see* floating-point unit (FPU)
Frame buffer, 113
Frame check, 241
FRDYIE, 150–51
FRDYIE bit, 151
FreeRTOS. *see* operating system usage, advanced
FRESET bit, 160–61
FRESETR, 160
FSTATR0, 155–56
FSTATR1, 157
FSUB, 54
FTP, 212
Function call(s)
  arguments, rules for passing, 23–26
  description of, 23
  example, 29–32, 35–40
  interrupt service routine, difference between, 26
  memory mapping, 14, 32–34
  stack usage, allocation and deallocation, 28–29
  type conversion, 26–28
  variable declaration, 23–26
FWEPROR, 148, 177

G
Global variable(s), addresses in memory, 13
GNURX toolchain, 19
G_vee_RecordLocations array, 173
G_vee_Sectors array, 175

H
Halt mode, of CAN, 263
Handshaking packet, of USB transaction, 191
Hard real time requirement(s), 80
Hardware post-processing, 66
Hardware pre-processing, 64
Heap memory, addresses in memory, 13
HTTP, 212

13.ES_Conrad_RX63N_Advanced_Index.qxd:Conrad 3/7/14 11:49 AM Page 330

Human Interface Device (HID). *see also* class driver(s)
- architecture of, 201
- class endpoints usage, *tab,* 201
- description of, 200
- driver functions, *ex,* 201–6
- function description, *tab,* 202
- reports of, 201

Hypertext Transfer Protocol (HTTP), 212

I

ICD bit, 220
ICK[3.0] bit, low power consumption, 301
ICLK, low power modes, 311
ID code protection, 181, 182–83
ID Stage, 50
IDE bit, 237
Identifier extension (IDE) bit, 237
IEEE 754, 44–49, 51, 54, 55
IF Stage, 49
ILGLERR bit, 157
Illegal carrier detection, 220
Illegal command error flag, 157
Immediate addressing mode, 6
Independent watchdog timer (IWDT)
- block diagram of, 286
- description of, 283–84
- registers of, 285–
- specifications of, 284–85
- start modes of, 284
- use of, 285

Inexact exception, 56, 62
Inline assembly, 18–19
Instruction decode stage (ID stage), 50
Instruction fetch stage (IF stage), 49
Instruction set
- arithmetic, instructions of, 9–10
- of control transfer, 11–12
- of data transfer, 8–9
- DSP, 13
- of floating-point operations, 11
- logic and bit manipulation, 10–11
- no operation (NOP), 12
- string manipulation, 12–13

Instruction set architecture (ISA), 43
Integer, description of, 4
Interframe space
- field of error frame, 238
- of overload frame, 238

Internet protocol, 211
Interrupt generator, and CAN, 244
Interrupt management, in FreeRTOS, 86–87, 90–91
Interrupt transfers, of USB, 191–92
Invalid operation exception, 56, 60, 62
Inverse Fourier Transform, 107
I/O port retention (IOKEEP) bit, 308
IOKEEP bit, 308
IP, 212
ISO 11783, and CAN standards, 242
ISO 11898, and CAN standards, 242
ISO 11992, and CAN standards, 242
Isochronous transfers, of USB, 192
IWDT. *see* independent watchdog timer (IWDT)
IWDT control register, 287
IWDT count stop control register (IWDTCSTPR), 289
IWDT refresh register (IWDTRR), 285–86
IWDT reset control register (IWDTRCR), 288–89
IWDT status register, 288
IWDTCLK, 285
IWDTCR, 287
IWDTCSTPR, 289
IWDT-dedicated clock, 285
IWDTRCR, 288–89
IWDTRR, 285–86
IWDTSR, 288

K

Kernel. *see* digital signal processing (DSP)

L

LANs, 209–11
Large-Scale Integrated device (LSI), 214
LCHNG bit, 220
LEDs, turning on, *ex,* 19–20
Link signal change, 220
Local area networks (LANs), 209–11
Lock-bit protection cancel, 160, 177
Lock-bit status, 157
Logic and bit manipulation instructions, 10–11
Low power modes and consumption
- all-module clock stop mode, 313–15
- deep software standby mode, 315, 320–23
- description of, 300–301
- modes, overview of, 301
- module clock function, setting of, *ex,* 311–12
- multi-clock function, setting of, *ex,* 311–12
- power, loss of, 301
- processor settings, 301–2
- registers of, 301–2
- SDCLK output control function, setting of, *ex,* 312–13

13.ES_Conrad_RX63N_Advanced_Index.qxd:Conrad 3/7/14 11:49 AM Page 331

sleep mode, 313, 316–18
software standby mode, 315, 318–20
startup process, 295–300
state machine for, 314
Low-voltage detection (LVD), 291
LSI, 214
LVD, 291
LVDA, 291–92
LVDLVLR, 291

M
M stage, 50
MAC, 214, 245
Machine code. *see* assembly language
Magic Packet detection enable, 219
Mailbox interrupt enable register (MIER), 248
Mailbox register j (MBj), 247
Mask invalid register (MKIVLR), 246–47
Mask register k (MKRk), 246
Mass Storage Class (MSC), 194
Matrix kernel(s), 95
Matrix multiplication, 103–7
Matrix multiplication time calculation, 70–74
Matrix structure, 96
MBj, 247
MCTLj, 248
Media Access Control (MAC), 214, 245
Media Independent Interface, 214
Memory. *see* data storage
Memory access (M stage), 50
Memory map, 14, 32–34
Message box, and CAN, 243
Message broadcasting, 235, 239–40
Message control register j (MCTLj), 248
Message identifier, of data and remote frame, 236
MIER, 248
Mini connectors, USB, 188–89
MKIVLR, 246–47
MKRk, 246
Module clock function, setting of, *ex,* 311–12
Module stop control register A (MSTPCRA), 303–4
Module stop control register B (MSTPCRB), 305–6
MPDE bit, 219
MSC, 194, 196
MSTPCRA register, 303–4
MSTPCRB register, 305–6
Multi-clock functionality, 311–12
Multi-tasking, of operating system, 77–78

N
Networking devices. *see* class driver(s)
No operation (NOP) instruction, 12
Non-Return-to-Zero principle, 240
Normal mailbox mode, of CAN, 244
Normal transfer mode
DMAC and, 120–21
external data transfer, *ex,* 131
internal data transfer, *ex,* 123–26
setting transfer, *ex,* 116
Notations, of RX family, 16–17

O
OFS0.WDTTOPS[1:0], 278
OHCI, 190
On-The-Go (OTG) Micro Connectors, USB, 189
OPE bit, 302
Open Host Controller Interface, 190
Operating system usage, advanced
counting semaphore, *ex,* 80
FreeRTOS, getting started with, 80–81
interrupt management, 86–87, 90–91
memory footprint, 81
memory management, 79
multitasking, 77–78
port, features of, 81
queue management, 84–86, 89–90
semaphores, 78
binary for synchronization, 87
task communication, 79
task management, 82–84, 88–89
task structure, *ex,* 79
task synchronization, 79
Operation mode, of CAN, 264
Optimization, value of, *ex,* 38–40
OTG Connectors, USB, 189
Out-of-place normal order calucation, 109
Output port enable (OPE) bit, 302
Overflow exception, 55, 57, 62
Overload delimiter, field of overload frame, 238
Overload flag, field of overload frame, 238
Overload frame, in CAN protocol, 235, 238
Overrun mode, in CAN, 252, 253

P
Passing by value, 24
PAUSE frame retransmit retry over, 221
PAUSE frame usage with TIME, 219
PC, 29
PCKA [7:0] bits, 164

13.ES_Conrad_RX63N_Advanced_Index.qxd:Conrad 3/7/14 11:49 AM Page 332

PCKA[3.0] bit, low power consumption, 301
PCKAR, 163–64
PCKB[3.0] bit, low power consumption, 301
PCLK, IWDT, to use, 285
P/E error status, 163
PEERRST [7:0] bits, 163
Peripheral clock notification bits, 164
Peripheral clock notification register (PCKAR), 163–64
Peripheral(s), power consumption, 301
PHY-LSI, 214
Pipeline processing
  concepts of, 49–50
  example of, *fig,* 49
Pointer variable, description of, 32, 34
Point-to-point connection, 209–10
Ports, setting up, *ex,* 19–20
Power consumption. *see* low power modes and consumption
Pre-emptive scheduling, in FreeRTOS, 84
PRGERR bit, 157
PRGSPD bit, 156
PRM bit, 216, 219
Program code, processing of user-written code, 64
Program counter (PC), description of, 29
Programming error flag, 157
Programming suspend status flag, 156
Promiscuous mode, 216, 219
Protocol controller, and CAN, 243
PSRTO bit, 221

Q
Queue(s)
  defined, 84
  description of, 79
  management of, 84–86, 89–90

R
RAM, stack usage and, 28
R_can_api.c, 270
R_can_api.h, 270
RDLAR, 226
RE bit, 219
Real time clock alarm interrupt, 320
Real time requirement(s), 80
Receive descriptor list start address register (RDLAR), 226
Receive frame length 11 to 0, 219
Receive frame length register (RFLR), 219, 220
Reception enable bit, 219
Reduced Media Independent Interface, 214
REFEF flag, 280, 288
Refresh error flag (REFEF), 280, 288
Register direct addressing mode, 6–7
Register indirect addressing mode, 7
Register relative addressing mode, 7–8
Register start mode, 281–82, 284
Register(s)
  assembly language, 3
  CAN, 244–49
  cluster transfer mode, 129
  DMAC, 115–20, 122, 123, 135
  EDMAC, 223–26
  ethernet controller, 216–21
  EXDMAC, 128
  FCU, 51–52
  independent watchdog timer, 285–89
  lower power modes and consumption, 301–2
  rules to use, 25
  watchdog timer, 276–81
Relative address, defined, 7
Remote frame, in CAN protocol, 235
Remote transmission request (RTR), 236
Repeat transfer mode
  DMAC and, 121–22
  external data transfer, *ex,* 131–32
  internal data transfer, *ex,* 125
  setting transfer, *ex,* 116
Reserved bit(s), 63
Reset mode, of CAN, 262–63
Resetprg.c, 295
R_Ether_Close_ZC, 227
R_Ether_Open_ZC, 227
R_Ether_Read_ZC, 228
R_Ether_Write_ZC, 228
RFL [11:0] bits, 219
RFLR, 219, 220
RM bit(s), 61
ROM, block configuration of, 142–43
ROM access violation flag, 150
ROM lock-bit read mode, 146
ROM P/E modes, 145–46, 164
ROM read modes, 145, 164
ROM status read mode, 145–46
ROMAE bit, 150
ROM/E2 DataFlash Read Mode, 145, 164
Rounding-modes, in floating point status word, 61
RPES [1.0] bits, 279
RPSS [1.0] bits, 279
RTM bit, 219

13.ES_Conrad_RX63N_Advanced_Index.qxd:Conrad 3/7/14 11:49 AM Page 333

RX family notations, 16–17
RX_DSP library. *see* digital signal processing (DSP)

S
SBYCR register, 302
SCKCR register, 309–10
SCKCR.FCK[3.0] bit, low power consumption, 301
SCSI, 196
SDCLK
  low power modes, 311
  power consumption, reduction of, 301
  setting of, *ex,* 313
Semaphore(s), 78, 87
Simple Flash API, 167–68
Single-precision, 44–47
Size modifiers, and source code writing, 15
Sleep mode
  of CAN, 264
  how to enter, *ex,* 316–18
  low power consumption, 301, 313
Small Computer System Interface (SCSI), 196
SOF field, of data and remote frame, 236
Soft real time requirement(s), 80
Software standby (SSBY) bit, 302
Software standby mode
  low power consumption, 302, 315
  transition to, *ex,* 318–20
Source code, 15–18
SP, 28
SSBY bit, 302
Stack overflow, description of, 29
Stack point (SP), description of, 28
Stack(s). *see* function call(s)
Standby control register (SBYCR), 302
Standby mode
  low power consumption, 302, 315
  transition to, *ex,* 318–20
Star configuration, 211
Startup process. *see also* boot mode
  clock frequencies, 296
  initial settings of, 296
  low power consumption, 295–300
Statistical kernel(s), 94
Status register (STR), 249
Status return handle, 108
STR, 249
String manipulation instructions, 12–13
String(s), description of, 5
Suspend ready flag, 157
SUSRDY bit, 157
System clock control register (SCKCR), 309–10

T
Task management, in FreeRTOS, 82–84, 88–89
TCP, 212
TDLAR, 225–26
TE bit, 219
Telnet virtual terminal communications protocol, 212
Tick interrupt, 83
Timer, and CAN, 243
Token packet, of USB transaction, 191
Toolchain
  GNURX, 19
  inline assembly, 19
TOPS [1:0] bits, 279
Transfers, USB, 191
Transform kernel(s), 94
Transmission Control Protocol (TCP), 212
Transmission enable bit, 219
Transmission/reception bit, 219
Transmit descriptor list start address register (TDLAR), 225–26
TX ID, 258
Type casting, 26

U
UHCI, 190
Underflow exception, 55, 58, 62
UNDFF flag, 280, 288
Universal Host Controller Interface, 190
Universal Serial Bus (USB) Connectivity
  cabling and connectors of, 187–88
  class drivers (*see* class driver(s))
  device, 191
  device detection, *ex,* 196–97
  electrical specifications of, 189
  ending connection to USB device, *ex,* 197–98
  FIFO memory buffer (*see* FIFO memory buffer)
  frame breakdown, *tab,* 192
  host, 189–91
  human interface device (*see* Human Interface Device (HID))
  interface specifications, 187–89
  receiving data, *ex,* 198–200
  transactions of, 191
  transfers of, 191–92

13.ES_Conrad_RX63N_Advanced_Index.qxd:Conrad 3/7/14 11:49 AM Page 334

USB. *see* Universal Serial Bus (USB) Connectivity
UxInitialCount, 87
UxMaxCount, 87

V
Variable declaration, 23–26
Vector structure, 96
Virtual EEPROM (VEE)
  blocks, allocation of, 174–75
  data management, 172
  description of, 170–71
  project data configuration, 175–76
  record storage and blocks, 172–73
  records, assigning to sectors, 173
  records of, 172, 176
Voltage
  brownout condition, detection of, 291–92
  Controller Area Network, 234
Voltage detection circuit (LVDA), 291–92
Voltage detection level select register (LVDLVLR), 291

W
Watchdog timer (WDT)
  all-module clock stop mode, 313
  auto-start mode, 282–83
  block diagram of, 277
  brownout condition, 291–92
  concepts of, 275
  count operations of, 281
  description of, 275
  independent watchdog timer (*see* independent watchdog timer (IWDT))
  overflow signal, 275
  register start mode, 281–82
  registers of, 276–81
  setting of, *ex,* 290–91
  sleep mode, 313
  software standby mode, 315
  specifications of, 276–77
  start modes of, 276
WB stage, 50
WDT. *see* watchdog timer (WDT)
WDT control/status register (WDTCSR), 276
WDT overflow signal (WDTOVF), 275
WDT refresh register (WDTRR), 278
WDT reset control register (WDTRCR), 280–81, 283
WDT status register (WDTSR), 280
WDT time-out period selection bits, 278
WDTCR, 278–79, 281, 283
WDTCSR, 276
WDTOVF, 275
WDTRCR, 280–81, 283
WDTRR, 278
WDTSCR, 276
WDTSR, 280
Write protect register, 257
Write-back stage (WB stage), 50

X
XQueueMessagesWaiting, 85
XQueueRecieve, 85
XSemaphoreGiveFromISR, 87
XSemaphoreTake, 87
XTicksToWait, 85, 87

Z
ZPF bit, 219